An Audio Units Plug-in to Simulate Spatial Audio Synthesis

Juszkiewicz, R. S.; Reddy, V.A., Tankanow, A. S.

1 Introduction

Using the fundamentals of head-related transfer functions (HRTFs) an Audio Units plug-in to binauralize a mono sound for accurate spatial perception through headphones is presented. This binauralization process is dependent upon the user to input the desired location and is performed in real time. In order to provide the background information necessary for understanding the implementation of the plug-in a summary of the relevant research conducted by the authors is presented in the first part of this paper. In the next section the motivation for the creation of such a plug-in is explained. The subsequent section details the steps of the plug-in's algorithm and finally, in the concluding section, possibilities for future improvements and additions to the presented work are discussed.

2 Background & Literature Survey

The use of sonic cues, by humans, for the estimation of the spatial location of an auditory object has been an area of sporadic interest in the past but has recently been the focus of large amounts research due to advances in computer and recording technology. It is now possible to empirically measure, analyze, and synthesize these spectral cues, which are also known as head-related transfer functions (HRTFs).

One of the initial theories describing sound source localization is known as the duplex theory. It attributes human localization abilities to two binaural cues: interaural time differences (ITD) and interaural amplitude (or level or intensity) differences (IAD, ILD and IID respectively). The former is defined as difference in arrival times of a sound's wavefront at the left and right ears and the latter as the amplitude difference generated between the right and left ears by a sound in the free field (Cheng and Wakefield 2001) Most mixing and panning in multi-channel environments is done by exploiting the principles of IAD. Sound is generally perceived by the human auditory system as originating from the point in space which is closer to the ear receiving it earlier with the largest amplitude. Due to the shape of the head, there may not be a direct path from the source to the ear which may cause problems in calculating the ITDs and IIDs. A solution to this problem was proposed by Lord Rayleigh (Rayleigh 1945). He modeled the head as a rigid sphere and then solved equations for wave

propagation based on this assumption. The results of his math lead to some interesting discoveries the first of which demonstrated how ITDs vary non-linearly with frequency. This is because the diameter of the head is comparable to the wavelength of a 1.5kHz sine wave which results in the attenuation of frequencies at and above 1.5kHz. For pure sinusoids below this frequency, the ITD is simply the phase difference of the received wave fronts at both ears. At frequencies above 1.5kHz the ITDs may match up with distances that are longer than one wavelength thus creating aliasing. It is because of this aliasing, caused by the shadowing effects of the head, that the phase difference no longer corresponds to a unique spatial location. This phenomenon is illustrated in figures 1 and 2.

In three-dimensional space the azimuth is the term used to refer to the lateral location of the sound, from straight in front of the listener to directly behind the listener. The aptly named elevation is the term used to describe the height of the sound. For a given frequency it is understood that the perceived azimuth varies approximately linearly with the logarithm of the IID. A major drawback of the duplex theory is in the inability to account for the elevation and distance of a sound in three-dimensional space. This localization ambiguity is due to the presence of an infinite number of points sharing the same ITD and/or IID. The locus of all these points is known as the "cone of confusion" because a sound emanating from anywhere inside of this cone appear to be coming from the same location according to the duplex theory. Figure 3 shows the cone of confusion for a constant ITD and IID.

After further research it became a common assumption that effects of the irregular shapes of the head, body and pinnae can be modeled by LTI filters that are dependent upon the location of the source. Empirical measurements, among other methods, are used to examine the frequency response of the ear. These measurements are then equalized and mathematically modeled as minimum-phase FIR filters that accurately model the ITDs and IIDs of the measured locations. The ITD is represented by the phase of the filter and the IID is captured by power in the filter's spectrum, i.e. its frequency response. If necessary, since the filters are of the minimum phase nature, they can be represented solely by their magnitude responses because the phase response of a minimum phase causal system and its magnitude response form a Hilbert transform pair. A filter pair that consists of a one filter for the left ear and one for the right ear, at a given spatial location, forms an HRTF.

The empirical measurements of HRTFs are done by inserting tiny microphones into the ear canals of a subject. A stimulus is then played from a specified azimuth and elevation at a constant radius from the head and its response is recorded. Some commonly used stimuli are clicks, impulses,

wideband white noise, MLS codes, Golay codes or sine tone frequency sweeps and the subjects are either humans or dummy heads. After the recordings are complete the effects of the recording and playback equipment must be removed from the HRTFs. This is done by using high precision equipment to measure the transfer functions of the loudspeakers and microphones used in the measurements and then the inverses of these functions are used to equalize the HRTFs. The time domain representations of these signals are known as Head Related Impulse Responses (HRIRs).

An alternative, and extremely useful, representation of the data contained in HRTFs is known as a Spatial Frequency Response Surface (SFRS). These surfaces indicate the energy received by the ear at a particular location for a fixed frequency. Due to the irregularity in the spatial sampling pattern utilized during HRTF measurement, triangulation and linear interpolation is used to construct these continuous surfaces. Figure 4 shows this process. These plots are similar to spectrograms in that they plot three variables on a two dimensional plane. In this case the three variables are azimuth, elevation and energy. Azimuth and elevation are the abscissa and ordinate, respectively, and energy is represented by the darkness (or sometimes color intensity) of the surface. A dark area at a specific azimuth and elevation indicates that a large amount of energy is received by the ear at that location for the frequency range that the plot represents. An example SRFS pair is shown in Figure 5 for the frequency range of 2343.8-2441.4Hz.

In theory, to use HRTFs or HRIRs to synthesize spatial audio is relatively simple. A monophonic sound is either convolved with or filtered by the HRIR or HRTF pair, respectively, that represents the desired location of the sound; this process will result in the inputted sound coming from the specified location in space. There are, however, a number of shortcomings to this theory, some of which have been actively researched. Listeners often complain about a "lack of presence" that occurs with sounds near the median plane (0° azimuth). This results in sounds that appear to be located inside of the head instead of in front of the head. Another related problem is one of front-back confusion. Some sounds that are supposed to be behind a listener appear to be in front of the listener.

Another major problem with HRTFs is that they are listener dependent. The effects that the pinnae, torso and head have on the sound that eventually reaches the tympanic membrane are unique for every individual. Listeners using the HRTF of a different subject may experience the lack of presence and front back confusion that was explained earlier. Averaging the results of many human subjects offers a theoretically more accurate representation for these individuals that have not had their own HRTFs measured but these substituted HRTFs are

obviously are not as precise as the real thing and a degradation in localization--even though it may be slightly minimized--still remains. Another possible explanation of the issue addressed in the previous paragraph is that current HRTF measuring techniques assume both ears to be symmetric and this is rarely the case for most people. Sometimes there are considerable differences between both ears which has an effect on the way that individual perceives the spatial locations of sounds. Researching new methods to facilitate the measuring of HRTFs is currently the focus of many people. A mathematical solution to the localization problem was the focus of some work done in the late 1990s (Zhang et. al. 1998). This algorithm will be discussed in the implementation section.

Depending on how many locations at which the HRTFs were recorded the memory requirements for implementing spatial audio synthesis algorithms on DSP hardware are relatively demanding. Along the same lines, if seamless panning of a sound from one location to the other is desired then interpolation must be used to create a continuous effect; this drastically increases the number of computations per second that are required to do this operation in real-time. Some early interpolation algorithms uses SFRSs to produce interpolated HRTFs. Based on weighted values from the SFRSs the magnitude response for a desired spatial location is able to be calculated.

An obvious problem with spatial audio synthesis lies in the very process of using HRTFs-- the sounds are filtered with the transfer functions impulse responses for given locations. This filtering operation often colors the sound. This coloration is suffered at the expense of having it in the desired location. The frequency response of an HRTF is shown in Figure 6. It can be seen in this figure that the certain frequencies are attenuated by more than 20dB which, depending on the nature of the sound, results in a severe alteration of the original. There have been some algorithms proposed to ameliorate this effect. One such algorithm transforms a given HRTF pair into its sum and difference (also commonly referred to as Mid-Side, or simply M-S) coordinates and then performs equalization based on those values (Hawksford 2002). Four different equalization methods using the M-S coordinates are proposed but are beyond the scope of this text.

The inability for perception of spatial location to be objectively measured is a limitation to the progress of work done in the HRTF field. This closely related to the limitations of our aural sense. If the source of the sound is not visible then it is more difficult to perceive its spatial location. To counter this it is advised to listen to the three dimensional sounds with closed eyes. This aural-visual discrepancy and the variation of the hearing sense among people makes evaluation of the quality of three dimensional sound synthesis algorithms dependent on listener's perception.

3 Motivation

A large motivation for the need of spatial synthesis using HRTFs can be attributed to the increased use of headphones due to the proliferation of portable audio devices. It is believed that headphones offer the purest reproduction of sound possible (Toole 1984); most recorded material, however, is optimized for playback through loudspeakers and when listened to on headphones the localization of instruments in the mix are often inaccurate. This is most obvious with sounds mixed to extreme locations: left, right and center. On headphones, sounds located to the left and right are perceived as coming from inside of the listener's ear canals and sounds in the center appear to becoming from inside the listener's head. Clearly this is not accurate but for a very long time it was tolerated. One way to solve this problem is to record the material binaurally using a procedure similar to that of HRTF measurement. This method is not very practical because the recorded material requires exclusively headphone playback. An alternative to this method is to use HRTFs to create a binaural mix at the studio level. The former approach may be more accurate but it is not very cost effective and it is impractical. The plug-in presented lays the groundwork for a more complex application that could be used to test the effectiveness of binaural mixes.

4 Implementation

Before the coding processed commenced the Ircam HRIR database[1] was selected to be used in the plug-in. This database contains 187 locations worth of impulse responses as 24bit/44.1kHz wav files. The files must be stored on the computer that is running the plug-in. They are read into the plug-in using the libsndfile API and then de-interleaved into two 512 length arrays, one for each channel. Real-time convolution using these impulse responses was attempted but it was not very computationally efficient, nor did it work correctly. To solve this problem the inputted audio was filtered with the left and right impulse responses using an FIR architecture. This method was much more computationally efficient and more importantly, it worked. The audio stream is then outputted to the host computer's headphone jack and the exhilarating experience of spatial audio can be enjoyed by the user.

[1] This database is publicly available at http://recherche.ircam.fr/equipes/salles/listen/download.html.

The azimuth and elevation are selectable by the user via sliders. Since all of the impulse response files are named according to a convention the integer values of the sliders are converted into strings, concatenated with the string that is common to all file names and then that file is loaded. Because there are a finite amount of impulse responses in the databases and no interpolation is implemented there is a short discontinuity in the sound when a new file is loaded.

Once the plug-in worked it was determined that the localization was lacking at certain azimuth and elevation values. Some of the same problems that were addressed in the background section were encountered. The algorithm introduced by Zhang et. al. was then implemented in MATLAB on all of the impulse response files. This computation was offloaded to MATLAB because it is a one time calculation and it would only waste processing power if it was carried out in real time.

This algorithm enhances the localization of sounds by applying weighting functions to the HRTFs. The weighting functions amplify the spectral differences of the reflections and diffractions of sounds due to the pinnae that vary as a function of sound direction. The original HRTF is then multiplied by this weighting function to create a more accurate HRTF. In the equations below $H'_P(f)$ represents the new HRTF, $W_P(f)$ is the weighting function, $H_P(f)$ is the original HRTF and m is a constant given in the paper to have an optimal value of 0.6.

$$H'_P(f) = W_P(f)\, H_P(f)$$
$$W_P(f) = |\hat{H}_P(f)|^m$$
$$\hat{H}_P(f) = H_P(f)/(\max_f |H_P(f)|)$$

Figure 7 shows the original frequency response of an of the left part of the HRTF shown in Figure 6 and the frequency response of it after it has been multiplied by the weighting function described above. It is obvious from this figure that the zeros of the response get 'pulled down' and the maxima remain in tact. This will color the sound more but, according to the authors and subjects of Zhang's listening tests, it improves localization. Elevation is perceived better and small (15°) changes in the azimuth are more apparent than with the original HRTFs.

5 Future Work & Conclusion

The plug-in presented only represents a very small portion of the vision of the authors. The possibilities for future additions to this plug-in are numerous. Interpolation can be added to eliminate the subtle discontinuities and improve panning, multiple sources can be added to simulate a true mixing environment (a prototype of this addition was written in MATLAB but the lack of a real time environment in MATLAB renders this implementation relatively worthless), a grandiose Open GL GUI can be added so that the user can better visualize where sources are being placed, one of the coloration reduction algorithms can be implemented to minimize the timbre changes introduced by the filters while maintaining localization and finally, a way for the user to enter his/her own HRTF via the GUI can be added. This would allow for the weighting function algorithm currently done in MATLAB to be done when a subject's HRTF is initially loaded into the program prior to real time calculations.

References

Aarts, R. M. (2000). Phantom sources applied to stereo-base widening. *Journal of the. Audio Engineering Society*, 48(3), 181 - 189

Baumgarte, F., and Faller, C. (2002). Design and evaluation of binaural cue coding schemes. In *Audio Engineering Society 113th Conference*, New York.

Cheng, C. I., and Wakefield, G. H. (1999). Introduction to head-related transfer functions (HRTF's): Representations of HRTFs in time, frequency, and space. In *Audio Engineering Society 107th Conference*, New York.

Cheng, C. I., and Wakefield, G. H. (2001). Moving sound source synthesis for binaural electroacoustic music using interpoplated head-related transfer functions (HRTFs)." *Computer Music Journal* 25(4), 57 – 80.

Hawksford, M. O. J. (2002). Scalable multichannel coding with HRTF enhancement for DVD and virtual sound systems. *Journal of the. Audio Engineering Society*, 50(11), 894 – 913.

Huopaniemi, J. and Karjalainen, M. (1997). Review of digital filter design and implementation methods for 3-D sound. In *Audio Engineering Society 102nd Convention*, Munich.

Rayleigh, L. (1945). *The Theory of Sound*. New York, NY: Dover Publications.

Toole, F. E. (1984). The acoustics and psychoacoustics of headphones. In *Audio Engineering Society 2nd Convention*, Anaheim.

Zhang, M., Tan, K. and Er, M. H. (1998). Three-dimensional sound synthesis based on head-related transfer functions. *Journal of the. Audio Engineering Society*, 46(10), 836 – 844.

Figure 1. Head Shadowing Effect on IID and ITD. The aliasing effects

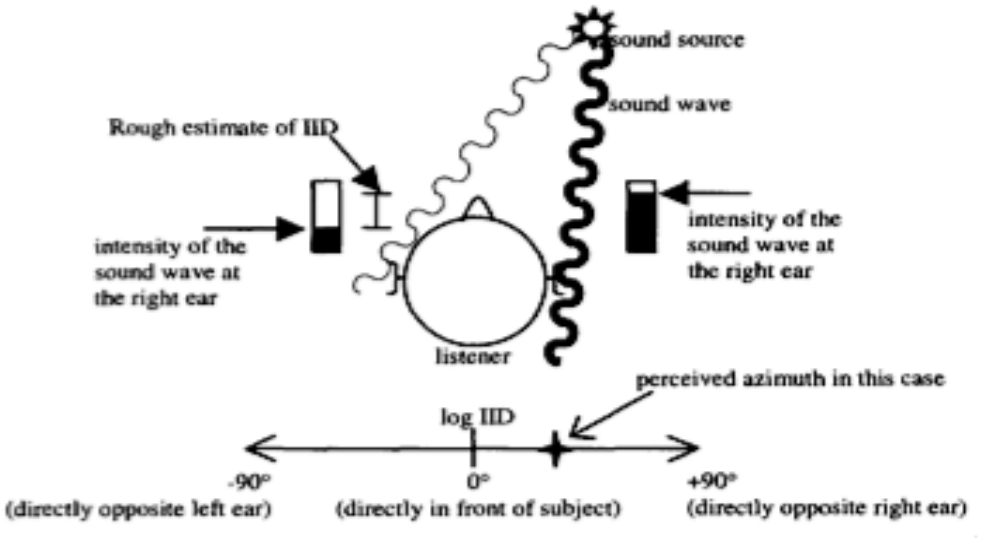


Figure 2. Head Shadowing Effect on ITD resulting in aliasing

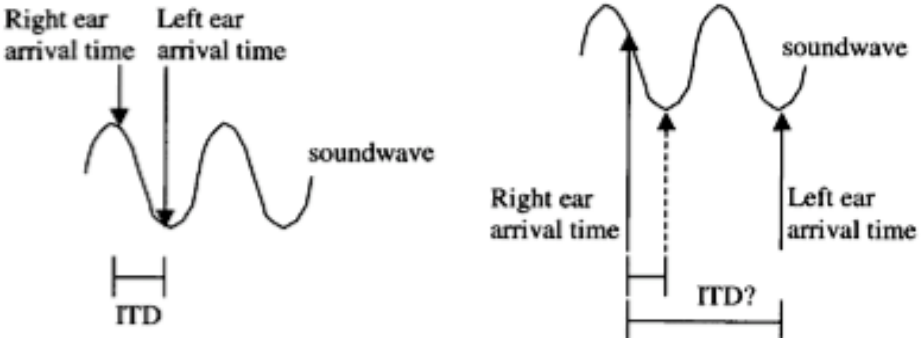


Figure 3. Cone of Confusion for a fixed IID and ITD

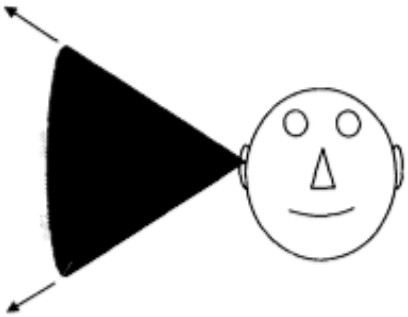


Figure 4. Construction of SFRSs from existing HRTF data of a particular frequency using Triangulation and linear interpolation of existing data.

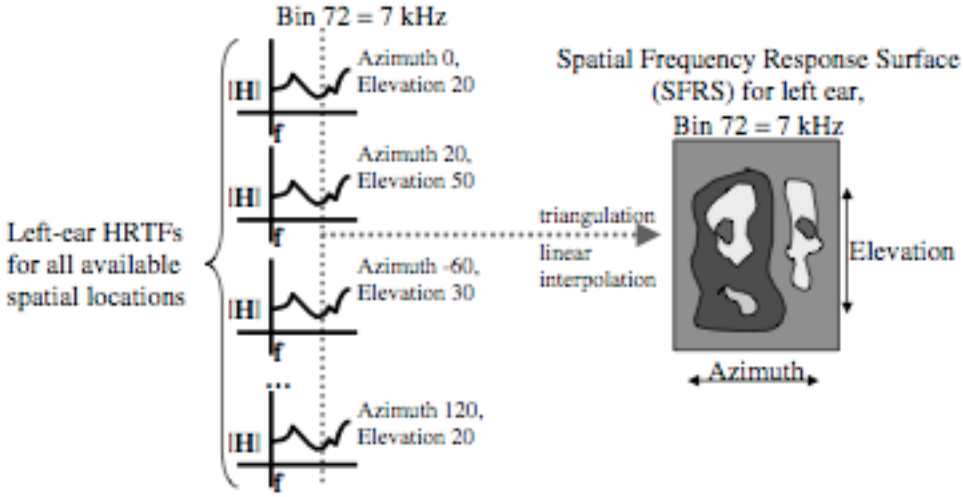


Figure 5. Example of an SFRS pair

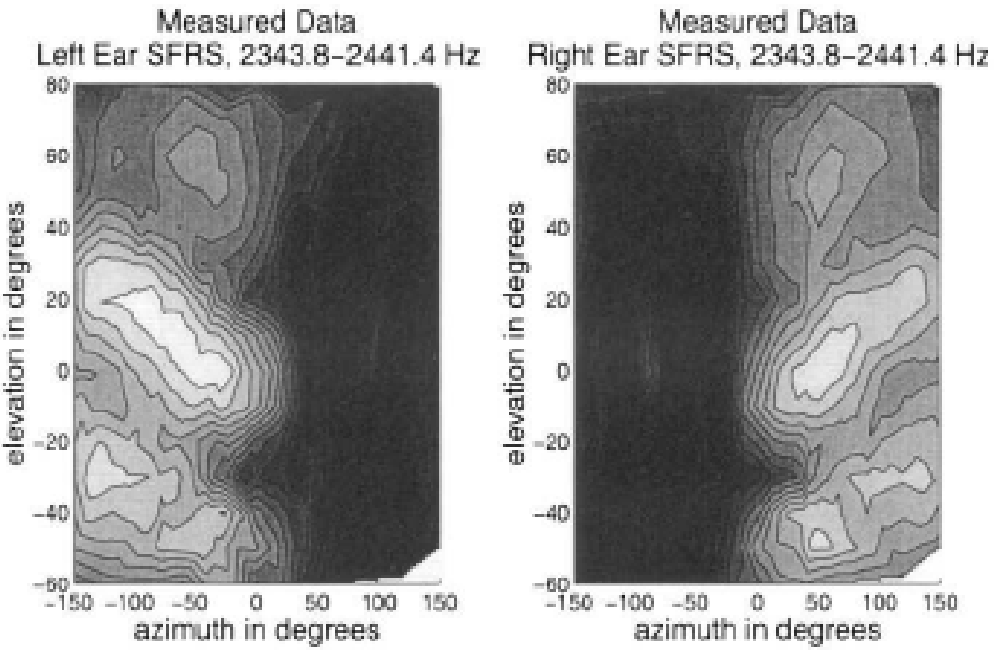


Figure 6.Frequency response of the left and right HRTFs for Azimuth and Elevation of 0^o

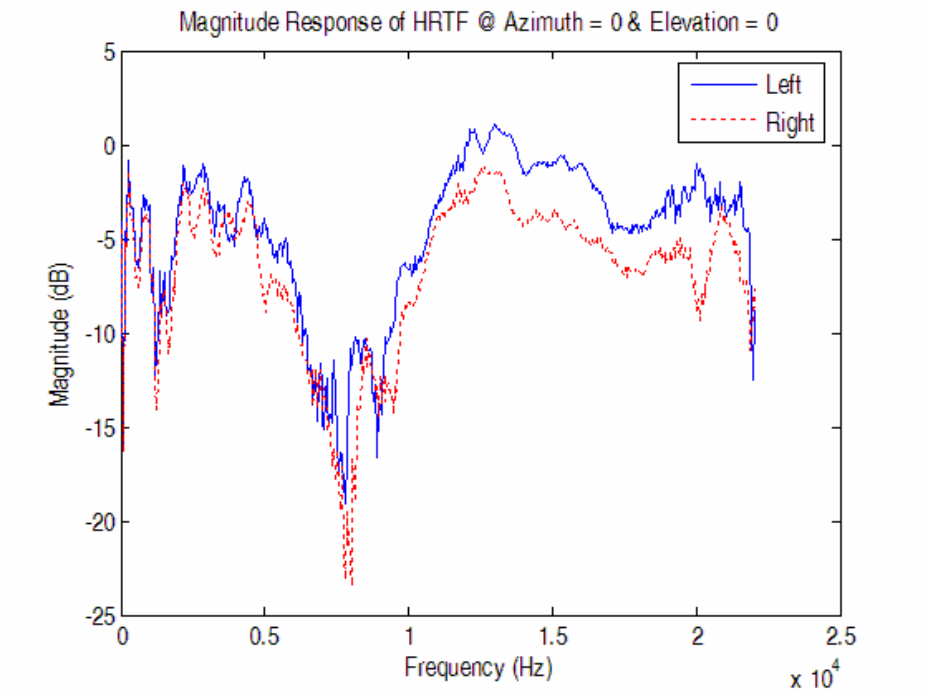
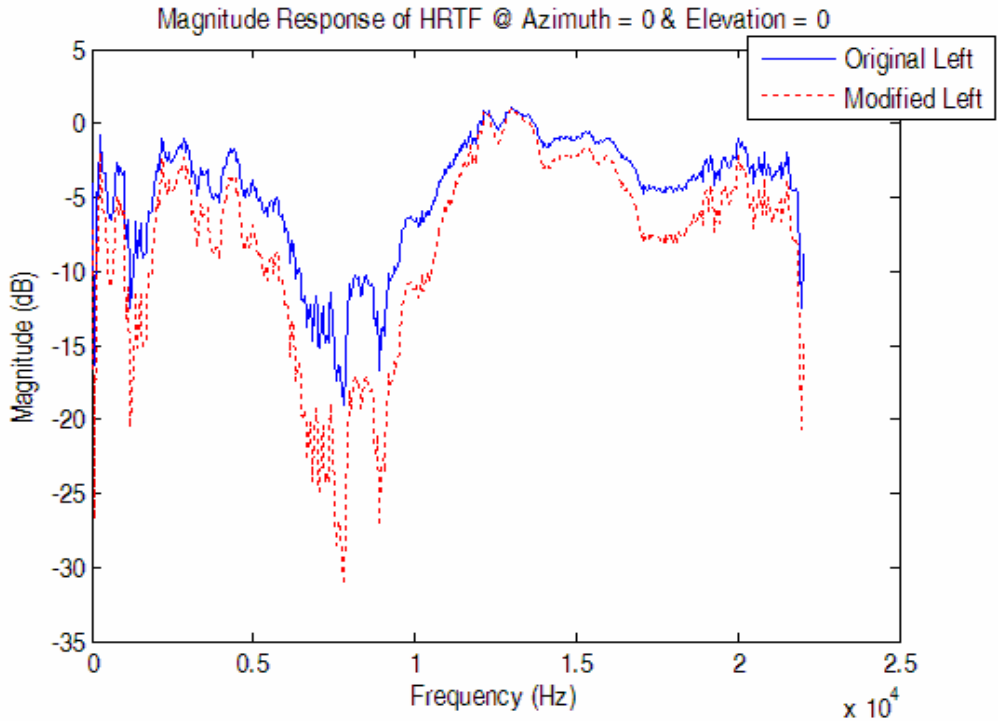


Figure 7. Original frequency response of left part of the HRTF shown in Figure 6 and the frequency response of it after it has been multiplied by the weighting function



APPENDIX

C++ CODE

```
/*
 *  fileIO.cpp
 *  wavIO
 *
 *  Created by Adam Tankanow on 4/18/06.
 *  Copyright 2006 __MyCompanyName__. All rights reserved.
 *
 */

#include "fileIO.h"

#include <iostream>
#include <stddef.h>
#include <stdlib.h>
using std::cout;
using std::cin;
using std::endl;

fileIO::fileIO()
{
    infilename = "Default.wav";
}

void fileIO::setInfilename(char *isFileName)
{
    infilename = isFileName;
}

double * fileIO::openFileAndGetData()
{
    if (! (infile = sf_open (infilename, SFM_READ, &sfinfo)))
    {   /* Open failed so print an error message. */
        printf ("Not able to open input file %s.\n", infilename) ;
        /* Print the error message from libsndfile. */
        puts (sf_strerror (NULL)) ;
    }

    readcount = sf_read_double (infile, data, BUFFER_LEN);

    return data;
}

void fileIO::getMonoData(int biFlag,double * dPtr)
{
    //double dTemp[BUFFER_LEN/2];
    int index = 0;
    if(biFlag==0)
    {
        for(int i=0;i<BUFFER_LEN;i=i+2)
        {
            dPtr[index] = data[i];
            index++;
        }
    }
    else
    if(biFlag==1)
    {
        for(int i=1;i<BUFFER_LEN;i=i+2)
        {
```

```
            dPtr[index] = data[i];
            index++;
        }
    }

}

void fileIO::conv(const Float32 * input,const double * impulse, Float32
* out, const int lengthinput, const int length_impulse)
{
    //const float impulse[] = {2,3};
    //const float input[] = {1,2,3,5,100};
    //const int lengthinput = sizeof(input)/sizeof(input[0]);
    //const int length_impulse = sizeof(impulse)/sizeof(double);
    const int LEN = length_impulse+lengthinput-1;

    Float32 sum = 0;
    Float32 fTemp;
    //Float32 out[LEN];
     for(int in=0;in<LEN;in++)
     {
        for(int i=0;i<length_impulse;i++)
        {
            for(int j=0;j<lengthinput;j++)
            {
                if((i+j)==in)
                {
                    fTemp = 0;
                    fTemp = (Float32) impulse[i];
                    //cout << "fTemp: " << fTemp << " input[j]: " <<
input[j];
                    sum = sum + (fTemp*input[j]);
                }
            }
        }
        out[in]=sum;
        //cout<< endl << out[in]<<endl;
        sum=0;


    }
    //Float32 * ptr;
    //ptr = out;
    //return out;
}

void fileIO::closeFile()
{
    /**
     *  Close input file
     **/
     sf_close (infile) ;}
/*
 *  fileIO.h
 *  wavIO
 *
 *  Created by Adam Tankanow on 4/18/06.
 *  Copyright 2006 __MyCompanyName__. All rights reserved.
 *
 */

//#include <Carbon/Carbon.h>
#include    <sndfile.h>
```

```
#define     BUFFER_LEN  1024
#define     MAX_CHANNELS    6

#ifndef fileIO_h
#define fileIO_h

class fileIO
{
    private:
        double data[BUFFER_LEN];
        SNDFILE     *infile;
        SNDFILE     *outfile;
        SF_INFO     sfinfo ;
        int         readcount ;
        char        *infilename; //=
"/Users/adamtankanow/Desktop/IRC_1002/RAW/WAV/IRC_1002_R/IRC_1002_R_R019
5_T000_P000.wav" ;

    public:
        fileIO();
        void setInfilename(char *);
        double * openFileAndGetData();
        void conv(const Float32 *, const double *, Float32 *,const int,
const int);
        void getMonoData(int,double *);
        void closeFile();
};

#endif
```

```
/*
*   File:       PlugInTest.h
*
*   Version:    1.0
*
*   Created:    4/19/06
*
*   Copyright:  Copyright © 2006 __MyCompanyName__, All Rights Reserved
*
*   Disclaimer: IMPORTANT:  This Apple software is supplied to you by
Apple Computer, Inc. ("Apple") in
*               consideration of your agreement to the following terms,
and your use, installation, modification
*               or redistribution of this Apple software constitutes
acceptance of these terms.  If you do
*               not agree with these terms, please do not use, install,
modify or redistribute this Apple
*               software.
```

/*===
    PlugInTest.h

===*/
#include "PlugInTest.h"
#include <iostream>
#include <stdlib.h>

#include <string>
using std::string;

#include <sstream>
using std::ostringstream;

using std::cout;

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~

COMPONENT_ENTRY(PlugInTest)


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
//  PlugInTest::PlugInTest
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
PlugInTest::PlugInTest(AudioUnit component)
    : AUEffectBase(component)
{
    CreateElements();
    Globals()->UseIndexedParameters(kNumberOfParameters);
    SetParameter(kParam_One, kDefaultValue_ParamOne );
    SetParameter(kParam_Two, kDefaultValue_ParamOne );

#if AU_DEBUG_DISPATCHER
    mDebugDispatcher = new AUDebugDispatcher (this);
#endif

    /**
     *  Initialize Buffer
     **/
    fBuffer = new Float32[512];
    for(int i=0;i<512;i++)
        fBuffer[i] = 0;

}


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
//  PlugInTest::GetParameterValueStrings
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
ComponentResult     PlugInTest::GetParameterValueStrings(AudioUnitScope     inScope,

AudioUnitParameterID    inParameterID,

CFArrayRef *        outStrings)
{

    return kAudioUnitErr_InvalidProperty;
}



//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
//  PlugInTest::GetParameterInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
ComponentResult     PlugInTest::GetParameterInfo(AudioUnitScope
    inScope,

AudioUnitParameterID    inParameterID,

AudioUnitParameterInfo  &outParameterInfo )
{
```

```
    ComponentResult result = noErr;

    outParameterInfo.flags =    kAudioUnitParameterFlag_IsWritable
                        |       kAudioUnitParameterFlag_IsReadable;

    if (inScope == kAudioUnitScope_Global) {
        switch(inParameterID)
        {
            case kParam_One:
                AUBase::FillInParameterName (outParameterInfo,
kParameterOneName, false);
                outParameterInfo.unit =
kAudioUnitParameterUnit_LinearGain;
                outParameterInfo.minValue = 0;
                outParameterInfo.maxValue = 23;
                outParameterInfo.defaultValue = kDefaultValue_ParamOne;
                break;
            case kParam_Two:
                AUBase::FillInParameterName (outParameterInfo,
kParameterTwoName, false);
                outParameterInfo.unit =
kAudioUnitParameterUnit_LinearGain;
                outParameterInfo.minValue = 0;
                outParameterInfo.maxValue = 3;
                outParameterInfo.defaultValue = kDefaultValue_ParamOne;
                break;
            default:
                result = kAudioUnitErr_InvalidParameter;
                break;
            }
    } else {
        result = kAudioUnitErr_InvalidParameter;
    }

    return result;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
//  PlugInTest::GetPropertyInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
ComponentResult     PlugInTest::GetPropertyInfo (AudioUnitPropertyID    inID,
                                                 AudioUnitScope      inScope,
                                                 AudioUnitElement
    inElement,
                                                 UInt32 &        outDataSize,
                                                 Boolean &       outWritable)
{
    return AUEffectBase::GetPropertyInfo (inID, inScope, inElement,
outDataSize, outWritable);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
//  PlugInTest::GetProperty
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
ComponentResult     PlugInTest::GetProperty(    AudioUnitPropertyID
inID,
                                                 AudioUnitScope      inScope,
                                                 AudioUnitElement
    inElement,
                                                 void *          outData )
```

```
{
    return AUEffectBase::GetProperty (inID, inScope, inElement,
outData);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
//  This is where we are going to do the processing
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
OSStatus PlugInTest::ProcessBufferLists(AudioUnitRenderActionFlags &
ioActionFlags,
                                const AudioBufferList & inBuffer,
AudioBufferList & outBuffer,
                                UInt32 inFramesToProcess)
{
    //cout << "Frames size: " << inFramesToProcess <<"\n";
    float * in1 = (float*)(inBuffer.mBuffers[0].mData);
    float * in2 = (float*)(inBuffer.mBuffers[1].mData);
    float * out1 = (float*)(outBuffer.mBuffers[0].mData);
    float * out2 = (float*)(outBuffer.mBuffers[1].mData);

    //processHRTF(in1,out1,out2,inFramesToProcess);
    processHRTF(in2,out1,out2,inFramesToProcess);
    return noErr;
}

void PlugInTest::processHRTF(const Float32 *sourcePtr,
                                 Float32 *leftPtr,
                                 Float32 *rightPtr,
                                 UInt32  inFramesToProcess)
{
    int iAzimuth = ((int)GetParameter( kParam_One ))*15;
    int iElevation = ((int)GetParameter( kParam_Two ))*15;

    loadHRTFData(iAzimuth,iElevation);
    UInt32 nSampleFrames = inFramesToProcess;
    Float32 *outL = leftPtr;
    Float32 *outR = rightPtr;
    /**
     *  Initialize Buffer
     *  This has been temporarily moved to the global scope
    Float32 *fBuffer = new Float32[nSampleFrames];
    for(int i=0;i<nSampleFrames;i++)
        fBuffer[i] = 0;
    */

    while(nSampleFrames-- > 0)
    {
        // cout << nSampleFrames << " % 511: " << nSampleFrames % 511 <<
"\n";
        Float32 fSample = *sourcePtr;
        sourcePtr++;
        fBuffer[nSampleFrames] = fSample;
        Float32 fLeftOut = 0;
        Float32 fRightOut = 0;
        Float32 fZero = 0;
        int iHRTFindex = 0;
        // for(int i=nSampleFrames;i<512;i++) changed for global buffer
        for(int i=nSampleFrames;i<nSampleFrames+512;i++)
        {
                fLeftOut += fBuffer[i % 512]*dHRTFLeftData[iHRTFindex];
                fRightOut += fBuffer[i %
512]*dHRTFRightData[iHRTFindex];
```

```
                iHRTFindex++;
            }
            *outL = fLeftOut;
            *outR = fRightOut;
            outL++;
            outR++;
        }
    }

    void PlugInTest::loadHRTFData(int iAzimuth, int iElevation)
    {

        /**
         *  Load in HRTF data
         **/

        /** convert azimuth to integer  */
    /*
        char a[3];
        itoa(iAzimuth,a,10);
        char cPad[ 10 ] = "000";
        strcat(cPad,a);
        int j=0;
        for(int i=strlen(cPad)-3;i<strlen(cPad);i++)
        {
            a[j] = *(cPad+i);
            j++;
        }
        a[3] = '\0';
    */
        ostringstream os;
        string sPad("000");
        string sAzimuth;
        os << sPad << iAzimuth;
        sAzimuth = os.str();


        ostringstream os2;
        string sElevation;
        os2 << sPad << iElevation;
        sElevation = os2.str();

        //b = "090\0";
        //cout << "a: " << b << "\n";
        // IRC_1047_C_R0195_T330_P345new.wav
        char filename[100] = "";
        char filebase[] =
    "/Users/adamtankanow/Desktop/HRTFs/IRC_1047_C_R0195_T\0";//IRC_1002_C_R0
    195_T\0";
        char filemid[] = "_P\0";
        char fileext[] = "new.wav\0";
        strcat(filename,filebase);
        strcat(filename,(sAzimuth.substr(sAzimuth.length()-3,3)).c_str());
        strcat(filename,filemid);
        strcat(filename,(sElevation.substr(sElevation.length()-
    3,3)).c_str());
        strcat(filename,fileext);
        f.setInfilename(filename);
        f.openFileAndGetData();
        f.getMonoData(0,dHRTFLeftData);
        f.getMonoData(1,dHRTFRightData);
        f.closeFile();

    }
```

```
char* PlugInTest::itoa( int value, char* result, int base ) {

    // check that the base if valid

    if (base < 2 || base > 16) { *result = 0; return result; }


    char* out = result;
    //char *out;

    int quotient = value;


    do {

        *out = "0123456789abcdef"[ std::abs( quotient % base ) ];

        ++out;

        quotient /= base;

    } while ( quotient );


    // Only apply negative sign for base 10

    if ( value < 0 && base == 10) *out++ = '-';


    std::reverse( result, out );

    *out = 0;

    return result;

}
/*
#pragma mark ____PlugInTestEffectKernel


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
//  PlugInTest::PlugInTestKernel::Reset()
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
void        PlugInTest::PlugInTestKernel::Reset()
{

    //f.setInfilename("/Users/adamtankanow/Desktop/IRC_1002/COMPENSATED/
WAV/IRC_2002_C/IRC_1002_C_R0195_T090_P000.wav");
        f.setInfilename("/IRC_1002_C_R0195_T090_P000.wav");
        f.openFileAndGetData();
        f.getMonoData(0,dHRTFLeftData);
        f.getMonoData(1,dHRTFRightData);

}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~
```

```
//  PlugInTest::PlugInTestKernel::Process
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~
void        PlugInTest::PlugInTestKernel::Process(  const Float32
    *inSourceP,
                                                    Float32
    *inDestP,
                                                    UInt32
    inFramesToProcess,
                                                    UInt32
    inNumChannels, // for version 2 AudioUnits inNumChannels is always 1
                                                    bool
    &ioSilence )
{

    //This code will pass-thru the audio data.
    //This is where you want to process data to produce an effect.

    const int iStereo = 2;
    UInt32 nSampleFrames = inFramesToProcess;
    const Float32 *sourceP = inSourceP;
    Float32 *destP = inDestP;
    Float32 gain = GetParameter( kParam_One );

    ///**
     *  Initialize Buffer
     //*
    fBuffer = new Float32[nSampleFrames];
    for(int i=0;i<nSampleFrames;i++)
        fBuffer[i] = 0;

    //Float32 sourceArray[nSampleFrames];
    //int index=0;
    while(nSampleFrames-- > 0)
    {
        Float32 fSample = *sourceP;
        //sourceArray[index] = fSample;
        //index++;
        sourceP++;


    //double d[] = {1,2,3};
    //Float32 fT[] = {3,2,1};
    //Float32 * testOut;
    //f.conv(fT,d,testOut,inFramesToProcess,512);
    //for(int z=0;z<6;z++)
    //  cout << "testOut[" << z << "]: " << *(testOut+z) << "\n";
    ///**
    *  Convolution:  this cannot be done in realtime
    int iBufferLength = inFramesToProcess + 512 - 1;
    Float32 fLeftConv[iBufferLength];
    f.conv(sourceArray,dHRTFLeftData,fLeftConv,inFramesToProcess,512);
    Float32 fRightConv[iBufferLength];
    f.conv(sourceArray,dHRTFRightData,fRightConv,inFramesToProcess,512);


    ///**
     *  Lets try making the impulse an fir filter
    //

        fBuffer[nSampleFrames] = fSample;
        Float32 fLeftOut = 0;
        Float32 fRightOut = 0;
        Float32 fZero = 0;
```

```
        for(int i=nSampleFrames;i<512;i++)
        {
            if( (nSampleFrames + i) < 512)
            {
                fLeftOut += fBuffer[i]*dHRTFLeftData[i];
                fRightOut += fBuffer[i]*dHRTFRightData[i];
            }
            else
            {
                fLeftOut = fZero;
                fRightOut = fZero;
            }
        }
        *destP = fLeftOut;//(fRightOut + fLeftOut) / 2;
        *(destP+1) = fRightOut;
        destP += iStereo;

    }
    // Interleave our mono convolutions into one stereo array
    ///*Float32 fTempOut[iBufferLength*2];
    int ii = 0;
    for(int i=0;i<iBufferLength;i++)
    {
        fTempOut[ii] = fLeftConv[i];
        fTempOut[ii+1] = fRightConv[i];
        ii+=2;
    }
    destP = fLeftConv;
    for(int i=0;i<10;i++)
        cout << "Final Output[" << i << "]: " << destP[i] << "\n";

    ///*while (nSampleFrames-- > 0) {
        Float32 inputSample = *sourceP;

        //The current (version 2) AudioUnit specification *requires*
        //non-interleaved format for all inputs and outputs. Therefore
inNumChannels is always 1

        sourceP += inNumChannels;   // advance to next frame (e.g. if
stereo, we're advancing 2 samples);
                                    // we're only processing one of an
arbitrary number of interleaved channels

            // here's where you do your DSP work
                //Float32 outputSample = f.conv(inputSample,dHRTFData);

        *destP = outputSample;
        destP += inNumChannels;
    }//
}
*/



/*
*   File:       PlugInTest.h
*
*   Version:    1.0
*
*   Created:    4/19/06
*
*   Copyright:  Copyright © 2006 __MyCompanyName__, All Rights Reserved
*
*   Disclaimer: IMPORTANT:  This Apple software is supplied to you by
```

```
Apple Computer, Inc. ("Apple") in
*               consideration of your agreement to the following terms,
and your use, installation, modification
*               or redistribution of this Apple software constitutes
acceptance of these terms.  If you do
*               not agree with these terms, please do not use, install,
modify or redistribute this Apple
*               software.
*
*               In consideration of your agreement to abide by the
following terms, and subject to these terms,
*               Apple grants you a personal, non-exclusive license,
under Apple's copyrights in this
*               original Apple software (the "Apple Software"), to use,
reproduce, modify and redistribute the
*               Apple Software, with or without modifications, in source
and/or binary forms; provided that if you
*               redistribute the Apple Software in its entirety and
without modifications, you must retain this
*               notice and the following text and disclaimers in all
such redistributions of the Apple Software.
*               Neither the name, trademarks, service marks or logos of
Apple Computer, Inc. may be used to
*               endorse or promote products derived from the Apple
Software without specific prior written
*               permission from Apple.  Except as expressly stated in
this notice, no other rights or
*               licenses, express or implied, are granted by Apple
herein, including but not limited to any
*               patent rights that may be infringed by your derivative
works or by other works in which the
*               Apple Software may be incorporated.
*
*               The Apple Software is provided by Apple on an "AS IS"
basis.  APPLE MAKES NO WARRANTIES, EXPRESS OR
*               IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED
WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY
*               AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE
APPLE SOFTWARE OR ITS USE AND OPERATION ALONE
*               OR IN COMBINATION WITH YOUR PRODUCTS.
*
*               IN NO EVENT SHALL APPLE BE LIABLE FOR ANY SPECIAL,
INDIRECT, INCIDENTAL OR CONSEQUENTIAL
*               DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS
*               OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
ARISING IN ANY WAY OUT OF THE USE,
*               REPRODUCTION, MODIFICATION AND/OR DISTRIBUTION OF THE
APPLE SOFTWARE, HOWEVER CAUSED AND WHETHER
*               UNDER THEORY OF CONTRACT, TORT (INCLUDING NEGLIGENCE),
STRICT LIABILITY OR OTHERWISE, EVEN
*               IF APPLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
*
*/
#include "AUEffectBase.h"
#include "PlugInTestVersion.h"
#include "fileIO.h"

#if AU_DEBUG_DISPATCHER
    #include "AUDebugDispatcher.h"
#endif
```

```
#ifndef __PlugInTest_h__
#define __PlugInTest_h__


#pragma mark ____PlugInTest Parameters

/**
 *  Buffer for storing outputs
 **/
//static Float32 *fBuffer;

// parameters
static const float kDefaultValue_ParamOne = 0.5;

static CFStringRef kParameterOneName = CFSTR("Azimuth");
static CFStringRef kParameterTwoName = CFSTR("Elevation");

enum {
    kParam_One =0,
    //Add your parameters here...
    kParam_Two =1,
    kNumberOfParameters=2
};

#pragma mark ____PlugInTest
class PlugInTest : public AUEffectBase
{
public:
    PlugInTest(AudioUnit component);
#if AU_DEBUG_DISPATCHER
    virtual ~PlugInTest () { delete mDebugDispatcher; }
#endif

    //virtual AUKernelBase *        NewKernel() { return new
PlugInTestKernel(this); }

    virtual ComponentResult GetParameterValueStrings(AudioUnitScope         inScope,

AudioUnitParameterID        inParameterID,
                                                     CFArrayRef *   outStrings);

    virtual ComponentResult     GetParameterInfo(AudioUnitScope         inScope,
    AudioUnitParameterID    inParameterID,

    AudioUnitParameterInfo &outParameterInfo);

    virtual ComponentResult     GetPropertyInfo(AudioUnitPropertyID     inID,
                                                AudioUnitScope          inScope,
                                                AudioUnitElement        inElement,
                                                UInt32 &
    outDataSize,
                                                Boolean &
    outWritable );

    virtual ComponentResult     GetProperty(AudioUnitPropertyID inID,
                                            AudioUnitScope      inScope,
                                            AudioUnitElement
    inElement,
                                            void *          outData);

    virtual bool                SupportsTail () { return false; }

    /*! @method Version */
    virtual ComponentResult Version() { return kPlugInTestVersion; }
```

```
    /**
     *  For multichannel processing
     */
    virtual OSStatus ProcessBufferLists(AudioUnitRenderActionFlags &
ioActionFlags,
                        const AudioBufferList & inBuffer,
AudioBufferList & outBuffer,
                        UInt32 inFramesToProcess);
    virtual void    processHRTF(    const Float32 *sourcePtr,
                                    Float32 *leftPtr,
                                    Float32 *rightPtr,
                                    UInt32  inFramesToProcess);

    virtual void loadHRTFData(int,int);
    virtual char* itoa( int, char*, int );
private:
        double dHRTFLeftData[512];
        double dHRTFRightData[512];
        fileIO f;
        Float32 *fBuffer;

/*
protected:
        class PlugInTestKernel : public AUKernelBase        // most of
the real work happens here
    {
public:
        PlugInTestKernel(AUEffectBase *inAudioUnit )
        : AUKernelBase(inAudioUnit)
    {
    }

        // *Required* overides for the process method for this effect
        // processes one channel of interleaved samples
        virtual void        Process(    const Float32   *inSourceP,
                                        Float32         *inDestP,
                                        UInt32
    inFramesToProcess,
                                        UInt32          inNumChannels,
                                        bool            &ioSilence);

        virtual void        Reset();

        //private: //state variables...
        double dHRTFLeftData[512];
        double dHRTFRightData[512];
        fileIO f;
    };
*/
};


#endif
```

MATLAB Code for the algorithm introduced by Zhang et al.

```
Clear all;clc;

azimuth = ['000'; '015'; '030'; '045'; '060'; '075';'090' ;'105';'120';
'135' ;'150'; '165'; '180' ;'195'; '210'; '225' ;'240'; '255';
```

```
'270';'285' ;'300'; '315'; '330'; '345'];
elevation =
['000';'015';'030';'045';'060';'075';'090';'315';'330';'345'];

%%%%%%%%%Reading in each HRTF wav file

for i=1:length(azimuth)
    for j = 1:length(elevation)

file =
sprintf('IRC_1047_C_R0195_T%s_P%s.wav',azimuth(i,:),elevation(j,:));
fid = fopen(file);
while(fid==-1)
    j=j+1
    file =
sprintf('IRC_1047_C_R0195_T%s_P%s.wav',azimuth(i,:),elevation(j,:));
    fid = fopen(file)
end

[inputi fsi n] = wavread(file);
bins = 1024;
m = .6;

data = inputi;

%%%%Implementation of the Algorithm

Hp = fft(data(:,1), bins);
magFreqresp = abs(Hp(1:bins/2));
freqvector = [fsi/(bins/2):fsi/(bins/2):fsi];

Hphat = Hp/(max(abs(Hp)));
Wp = abs(Hphat).^m;
HpPrime = Wp.*Hp;

newleft = ifft(HpPrime);
newleft = newleft(1:bins/2);

Hp = fft(data(:,2), bins);
magFreqresp = abs(Hp(1:bins/2));
freqvector = [fsi/(bins/2):fsi/(bins/2):fsi];

Hphat = Hp/(max(abs(Hp)));
Wp = abs(Hphat).^m;
HpPrime = Wp.*Hp;

newright = ifft(HpPrime);
newright = newright(1:bins/2);

newleft = filter(newleft,1,data(:,1));
newright = filter(newright,1,data(:,2));

newstereo = [newleft newright];

%%%%%Writing the new improved HRTF wav file

newfile = sprintf('IRC_1047_C_R0195_T%s_P%snew.wav',azimuth(i,:),
elevation(j,:));
wavwrite(newstereo,fsi,n,newfile);

    end
end
```