DAFX
Binaural Phase Vocoder Report

Jack Walters
j.h.walters@se19.qmul.ac.uk

Monday 4th May, 2020

Abstract

This report presents the Binaural Phase Vocoder, a VST/AU plugin written in C++ in the JUCE[4] framework whereby users can position a virtual sound source in a three-dimensional space relative to themselves and process the sound through an overlap-add phase vocoder. Sound is spatialised through through convolution with head-related transfer functions (HTRF) and interaural time difference (ITD), and the phase vocoding allows the user to alter the phase of the input signal spectrum.

1 Introduction

The architecture of the phase vocoder has been something that I have been interested in designing since learning about how efficiently it can be constructed through overlap-add buffer management. After covering the topic of spatial audio and reading about it in *Audio Effects*[3], I began to research how binaural audio was implemented in C++. After learning that binaural audio is achieved partially through convolution of head-related transfer functions (HRTF) with the input signal in the frequency domain, the similarities between the synthesis and re-synthesis of signals in a phase vocoder and the partitioned convolution in binaural audio became apparent. Furthermore, through utilising Lord Rayleigh's *Duplex Theory*, azimuth dependent delays can be introduced for each ear, introducing a memory dependent aspect to the plugin. In all instances aside from the "Whisperisation" effect, the system is phase vocoder and convolution-oriented, and hence linear and time invariant in nature.

2 Method

The architecture of this project is centred around Phase Vocoder synthesis, re-synthesis, HRIR loading and pipelining and interaural delay implementation. Figure 1 shows the block diagram representation of the signal and control flow of the project. This diagram represents the overarching control of the interface over the operation of subsequent components. Upon loading the plugin, the user is presented with a top-down image of a a head at the epicentre of a black circle surrounding it. To position the virtual sound source, the user can drag either the blue sound source inside the circle to adjust azimuth, the slider immediately to the right to adjust elevation or the slider below to adjust distance between the listener and the source. Inspiration for this interface layout and code segments, including the C arrays for the head image and virtual sound source image and the

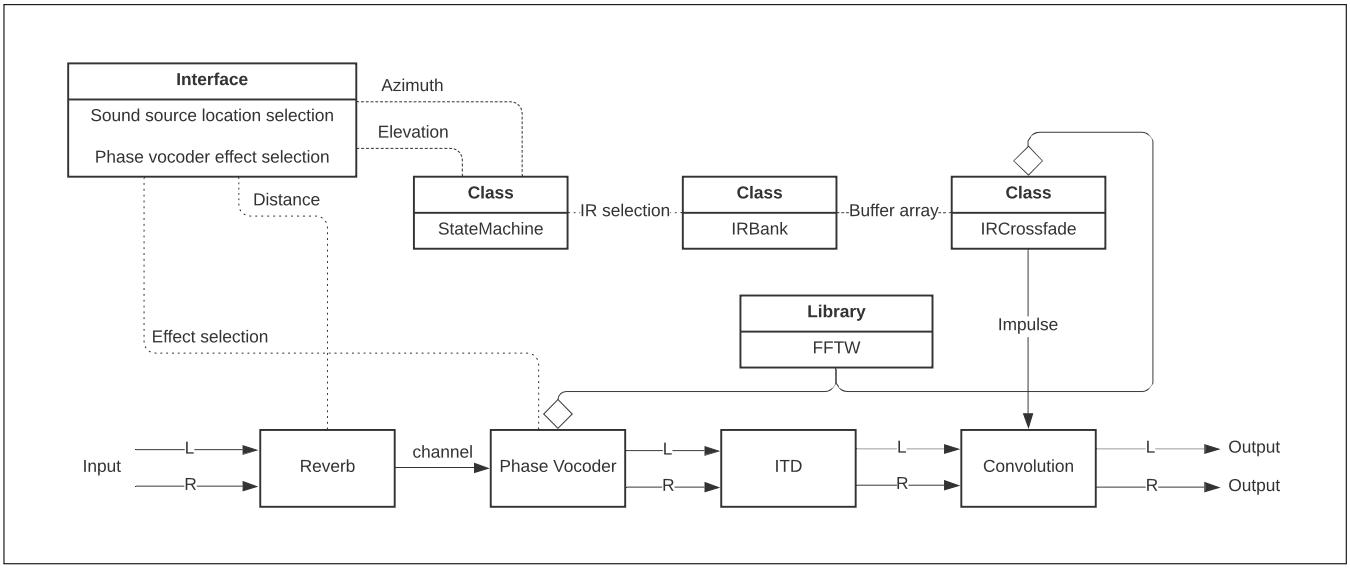

FIGURE 1: Block Diagram

`Util.h` header were taken from GitHub user "twoz", who designed an interface allowing for azimuth and elevation adjustment[5]. On the right-hand side of the interface, the user can also select one of three phase vocoder processing options: "Pass-through Vocoder", "Robotisation", "Whisperisation".

2.1 HRIR Pipeline

Following the user's selection of azimuth and elevation, a state machine inside `StateMachine.cpp` processes these values as parameters to determine the orientation state of the sound source. This design processes the sphere of combined azimuth and elevation values surrounding the listener as an array of combinations of impulse responses. A selection of HRIRs recorded on dummy head "D1" at various azimuths and elevations in an anechoic chamber as part of the SADIE II Database are used as the impulse responses that are selected dependent on the user's positioning of the sound source [1]. Wherever the positioning of the sound source, there are always 4 impulse responses in its immediate vicinity: to the lower-left (`LL`), upper-left (`UL`), lower-right (`LR`) and upper-right(`UR`); the state machine simply updates these 4 variables with the impulse response index numbers as and when the source is re-positioned.

These values correspond to indices of an `AudioSampleBuffer` array generated in `IRBank.cpp`. Listing 1 exhibits the `for()` loop structure whereby binary files are loaded into a `AudioSampleBuffer` array. At each index of the `for()` loop, a `char*` pointer is assigned the corresponding binary data and read into the memory stream `inputStream`. Line 15 shows object `reader` of type `AudioFormatReader` which will read samples from this audio file stream. Lines 17 to 25 demonstrate the reader "reading" the samples into the array `audioBuffer` of type `AudioSampleBuffer`.

LISTING 1: IRBank.cpp

```
1
2  for (int i = 0; i < BinaryData::namedResourceListSize; ++i)
3  {
4      const char* binaryData = 0;
5      int binaryDataSize = 0;
6
7      binaryDataSize = HRIR_SIZE_FILE_SIZE;
8      binaryData = BinaryData::getNamedResource
9          (BinaryData::namedResourceList[i], binaryDataSize);
10
11     auto* inputStream = new MemoryInputStream
12         (binaryData, binaryDataSize, false);
13
14     WavAudioFormat format;
15     reader = format.createReaderFor (inputStream, true);
16
17     if (reader)
18     {
19         int streamNumChannels = reader->numChannels;
20         int streamNumSamples = (int)reader->lengthInSamples;
21
22     bufferArray[i] = juce::AudioBuffer<float>
23         (streamNumChannels, streamNumSamples);
24     reader->read(&bufferArray[i], 0, streamNumSamples, 0, true, true);
25     }
26 }
```

Inside the audio processing function `DafxBinauralPhaseVocoderAudioProcessor`, the `IRCrossfade` class is instantiated inside the phase vocoder architecture and its method `loadImpulses` is fed the channel number (recall from Figure 1 that the phase vocoder processes each channel individually), and 4 instances of the `audioBuffer` array at separate indices assigned in `StateMachine.cpp`. Upon the calling of the method, these 4 buffers are copied into standard vectors before every sample index then is copied into the real dimension of a `fftw-complex` array before they are individually Fourier Transformed with the FFTW library[2]. `impulseFFTBlend` is concerned with combining the spectra of these 4 impulse responses through complex multiplication.

$$\begin{aligned}(a+bi)(c+di)(e+fi)(g+hi) = (eacg - ebdg - adfg - bcfg - eadh - ebch - acfh + bdfh) \\ + i(each + eadg + ebcg - ebdh + acfg - adfh - bcfh - bdfg)\end{aligned} \tag{1}$$

Equation 1 demonstrates the complex multiplication performed on the real (a, c, e, g) and imaginary (b, d, f, h) axes of the impulses. Not featured in the equation are the scale factors used to normalise the coefficients that arise as a result of the multiplication: in this project 1/K was used as a scaling factor. It is also worth noting that due to the conjugate symmetric nature of a transformed real signal, the `for()` loop this complex

multiplication is performed inside of iterates from 0 to K/2, and the conjugate symmetric pairs are concurrently written into the product array at index `K - i`. Following this, the `backwardFFTandStore` method performs an IFFT on the product array and copies it into the `AudioSampleBuffer` array `crossfadedImpulseData`.

2.2 Audio Pipeline

When a new audio block is passed into the `processBlock` method inside the `DafxBinauralPhaseVocoderAudioProcessor` class, both channels are passed into the JUCE `Reverb` class, the wet parameter of which is controlled by the user-specified distance. The wet signal is then copied to the `processBuffer` before being fed channel by channel into the phase vocoder architecture (partially inspired by Reiss and McPherson[3]). The samples from `processBuffer` are copied into an input buffer and when the number of samples copied reaches the hop size, the most recent K samples are windowed and copied into the real dimension of the `fftSignalTimeDomain` array (the imaginary dimension is also zeroed). A forward transform is then performed on the array, and the resultant magnitude and phase values are output into the `fftSignalFrequencyDomain` array. At this point we are able to apply phase vocoding effects to the phase values; Listing 2 shows the 3 phase processing options that can be selected by the user in the GUI. If "Pass-through Vocoder" is selected the phase is reconstructed to its original form through the arc-tangent function on line 3. If the user has selected either "Robotisation" or "Whisperisation", the phase is either zeroed or randomised inside a range of 0 - π. Similar to the `IRCrossfade` method described above, only amplitude and phase values between 0 - K/2 are calculated for greater efficiency, and the the conjugate symmetric mirrored values are simultaneously written back into the `fftSignalFrequencyDomain` array at index `K - i`. The sample's magnitude values are also calculated and multiplied by the reciprocal of the user-specified distance, meaning that as the distance increased the signal will sound quieter once it is re-synthesised back into the time domain. Following the phase vocoding the complex signal is transformed back into time domain through the IFFT and multiplied by the coefficient scaling factor (1/K) before being copied back into the `processBuffer`.

LISTING 2: Phase Vocoder Effects

```
1  if (passthrough)
2  {
3      phase = atan2(fftSignalFrequencyDomain_[i][1],
4          fftSignalFrequencyDomain_[i][0]);
5  }
6  else if (robotisation)
7  {
8      phase = 0.0;
9  }
10 else if (whisperisation)
11 {
12      phase = 2.0 * M_PI * (float)rand() / (float)RAND_MAX;
13 }
```

Following the phase vocoder architecture, the the left and right channels are assigned their own channel of the buffer `delayBuffer`, the length of which is 2 seconds. To calculate the interaural time difference (ITD), the delay for each ear is calculated separately. The azimuth-dependent delay for each ear is calculated differently dependent on which quadrant the sound source is in. Taking quadrant 1 as an example, the left and right ear delay times in milliseconds are calculated as demonstrated in equation 2:

$$\begin{aligned} T_{d,R}(\theta) &= a(1 - cos\theta)/c \\ T_{d,L}(\theta) &= a(|\theta| + 1 - \pi/2)c \end{aligned} \tag{2}$$

Where a is the radius of the head (which was as 0.07m), θ the azimuth angle and c the speed of sound. Following the calculation of the delay time for each ear, this millisecond value is converted to a read pointer value by means of multiplication with the sample rate in relation to the delay buffer write pointer, the result of which is kept inside the delay buffer length by means of modulo arithmetic. This delay buffer read pointer is then used to read samples from the delay buffer into the original buffer. At this final and perhaps most crucial stage of the signal chain, the `impulseResponseCrossfade` buffer created in `IRCrossfade` is loaded into the `Convolution` object, and the buffer is convolved with the impulse response by means of uniform-partition overlap-save convolution (UPOLS).

3 Results

Figure 2 to 7 show the left and right channel waveforms above and below each other for various azimuth, elevation and and distance values. Audio was rendered in DAW REAPER at 44.1kHz and plots were made in Sonic Visualiser. Figure 3 shows the that from a dynamics perspective, audio passes through the plugin relatively unchanged when it is non-bypassed to when the sound source is placed directly in front of the listener. Despite this, the audio does sound somewhat processed and by virtue of being convolved with an impulse response alone the audio sounds marginally more reverberant than when it is dry. Moving the sound source from around the listener from 0 - 180° and back from -180° to 0 azimuth (i.e. clockwise), the sound source clearly starts to move, with minor jumps being heard when the azimuth crosses boundaries in the `ImpulseSelectionStateMachine` class. The sound source sounds slightly louder when positioned behind the listener, although to naked eye Figures 3 and 5 look almost identical. Adjusting elevation also introduces a discernable effect, however it is not as easy to locate the sound source when it is being adjusted horizontally. Adjusting distance introduces very light reverberation in a satisfying way, although the drop-off in level is slightly more steep in the 5.0-15.0 meter range than one would imagine it should be. Engaging either of the phase-altering vocoder effects produces a very noticeable change in the audio's quality. The "Robotisaion" effect introduces a perceptible buzz and monotone quality to the audio, while the "whisperisation" effect makes the audio sound grainy and whisper-like.

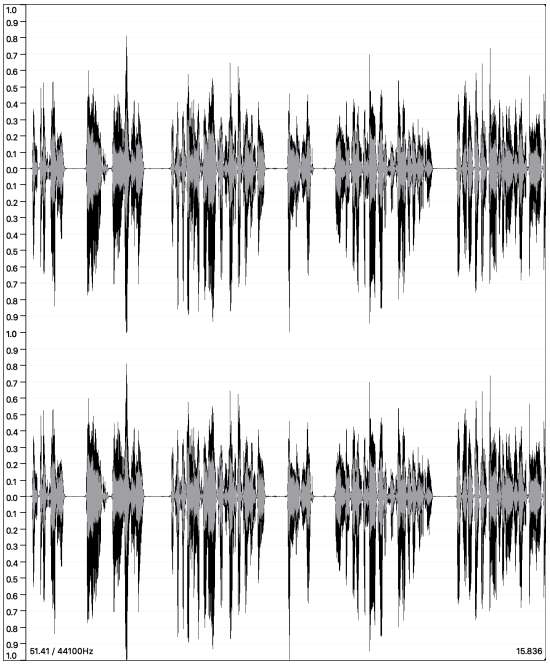

FIGURE 2: Original

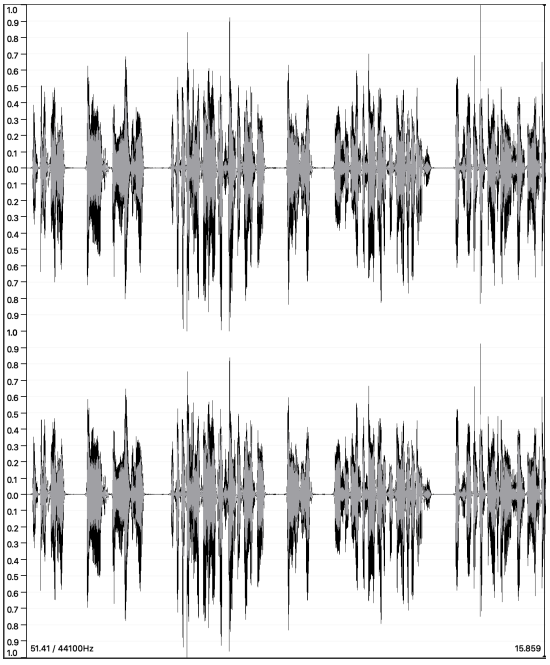

FIGURE 3: 0° Azimuth, 0° Elevation, 2.0m

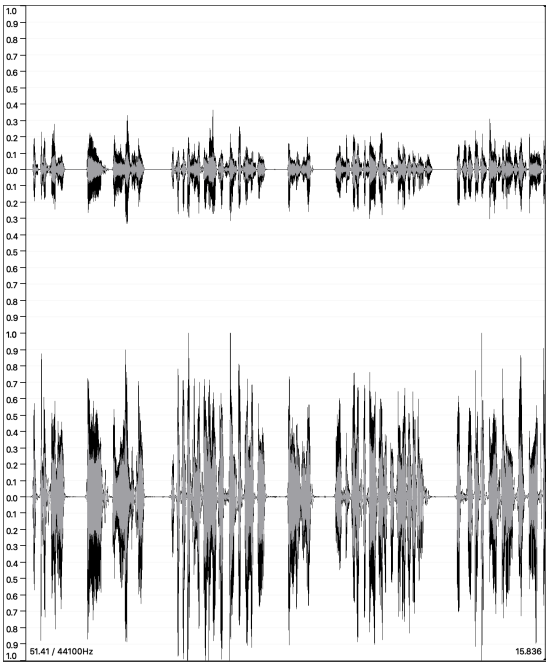

FIGURE 4: +90.23° Azimuth, 0° Elevation, 2.0m

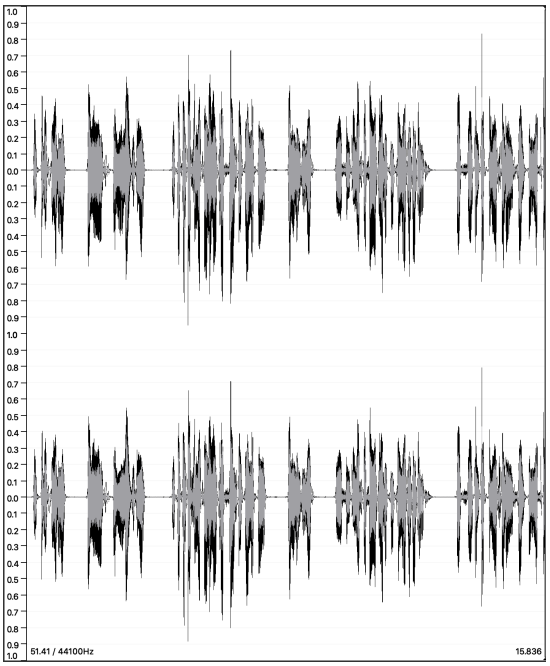

FIGURE 5: +180° Azimuth, 0° Elevation, 2.0m

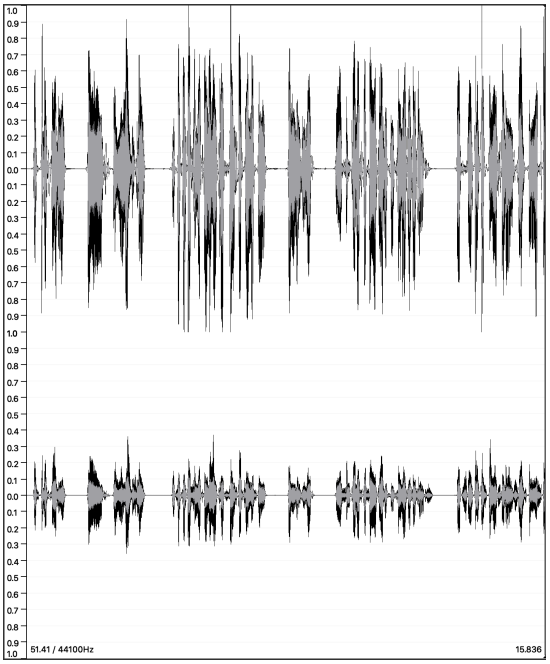

FIGURE 6: -90.17° Azimuth, 0° Elevation, 2.0m

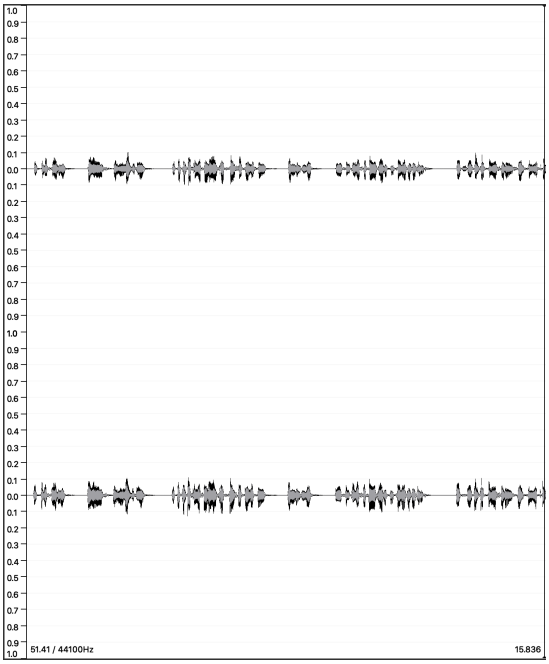

FIGURE 7: +141.20° Azimuth, -90° Elevation, 17.2m

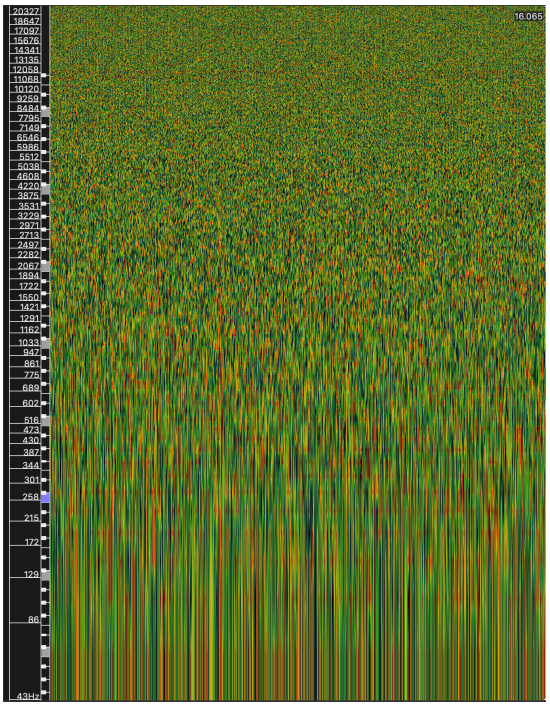

FIGURE 8: Original phase. 0-Fs/2, log scale

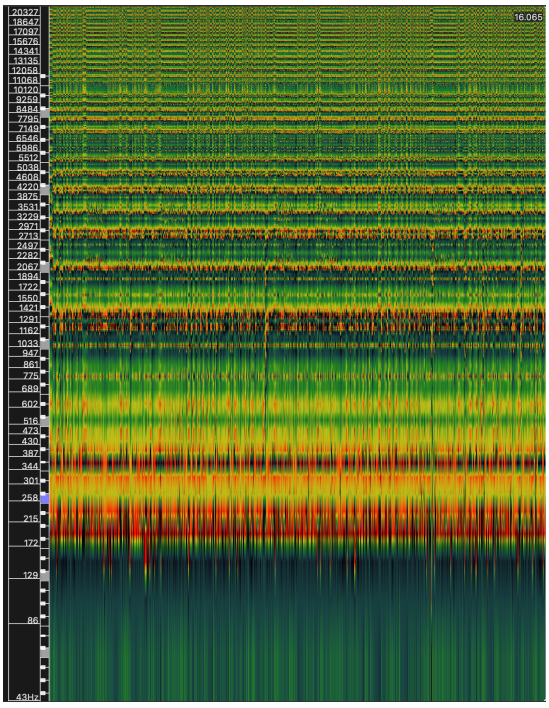

FIGURE 9: Robotisation phase. 0-Fs/2, log scale

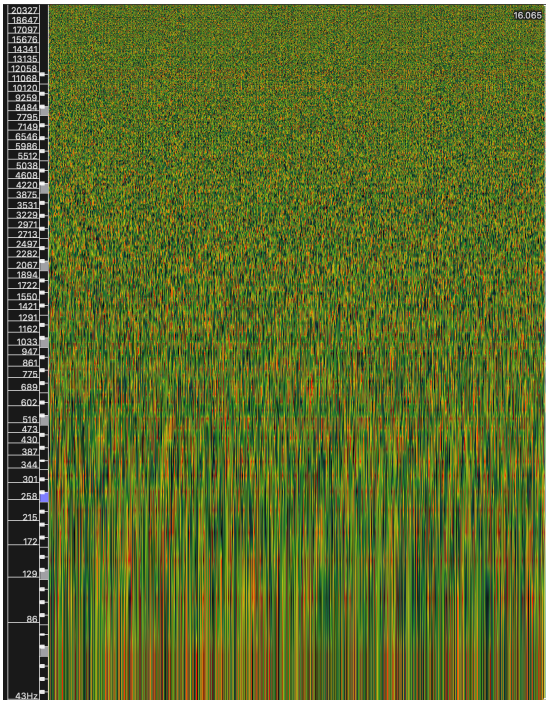

FIGURE 10: Whisperisation phase. 0-Fs/2, log scale

4 Discussion

Overall I believe the project to be successful in its implementation. The sound is effectively spatialised dependent on the user's selected azimuth, elevation and distance. Figures 2-7 show how the inter-aural level differences (ILD) are altered according to where the sound source is located. If the sound source is located either directly to the right or left of the listener, the sound level in the opposite ear drops and the sound level in the ear closer to the source increases. I believe the effect to sound convincing, however there is a more noticeable transition from one set of impulse responses to another when rotating the sound source around the head. As was mentioned in the results section, I believe the sound source to sound louder when it was located directly behind the listener, however the plots do not seem to reflect this. The distance parameter also effectively conveys the sound source moving further away, however between 5.0 and 15.0 meters the volume seems to drop off too quickly. Perhaps implementing a non-linear inverse relationship between the distance and magnitude of the signal would be something to look into for future work.

In terms of the overall architecture, I was also hoping to implement a UPOLS architecture myself but implementing it as effectively as JUCE's in-built `Convolution` class proved extremely difficult. Being able to do so would have meaning refining the architecture for more efficiency, combining the convolution process of the signal with the impulse, with the convolution of a more reverberant impulse response with the already convolved signal for the reverb effect. One area of the project I also wish to refine is the state machine in the `ImpulseSelectionStateMachine` class. Building a structure which utilises reg-

ular expression to parse through the binary file names for azimuth and elevation values rather than having to manually build a state machine would make the system far more flexible for future use. Although neighbouring impulse responses are blended together to simulate a smoother sphere of impulses surrounding the listener, they do not adaptively blend dependent on where the user has chosen the sound source to be. Rather, they are pre-fixed according to the design of the state machine meaning that crossing the boundary between say 30° and 60° azimuth produces a minor but noticeable audible transition to the immediate neighbour array of impulses. I was not surprised to find the sound was not as effectively spatialised when the elevation was adjusted rather than the azimuth. This would be down to two reasons: the human ear being less sensitive to height change and a lack of impulse responses elevation measurements compared to azimtuh. The phase vocoder effects also effectively combined into the binaural space, and there is no noticeable clash of the two effects running in parallel. In future I would like to expand on the functionality of the phase vocoder and allow the user to fine tune their effects to a greater degree.

4.1 Future Work

As was mentioned in the discussion, this project would benefit from an overhaul of the impulse response selection architecture. To simulate an audio source moving around the listener's head more effectively a larger set of impulse responses needs to be utilised in the design. This would require dynamic memory allocation and reallocation that allows the structure to load impulse responses into memory and then delete them from memory when they are not needed. Attempting to load a larger data set than mine in its entirety into memory would saturate the memory and slow the program down. To further simulate a moving sound source, the complex multiplication present in `IRCrossfade` would need to function more as a weighted avergae than a direct complex multiplication that is the equal product of all 4 impulse responses. The weighting of this average would be adjusted according to where between the loaded impulses the user had positioned their sound source. Finally, I would also like to include data sets from other impulse response dummies, which would allow the program to account for the variability of ILD and head-shadowing that occurs between people with differently shaped heads; this would also allow the ITD calculations to be refined, whereby the variable a denoting difference is adjusted according to head shape.

References

[1] Cal Armstrong, Lewis Thresh, Damian Murphy, and Gavin Kearney. A perceptual evaluation of individual and non-individual hrtfs: A case study of the sadie ii database. *Applied Sciences*, 8(11):2029, 2018.

[2] Steven G Johnson and Matteo Frigo. *JUCE 3.3.8*, 22 February 2020. http://www.fftw.org/.

[3] Joshua D Reiss and Andrew McPherson. *Audio effects: theory, implementation and application*. CRC Press, 2014.

[4] ROLI. *JUCE 5.4.7*, 10 February 2020. https://juce.com/.

[5] twoz. *binaural-vst*, 2016 (accessed 10 April, 2020). https://github.com/twoz/binaural-vst.