UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

EDUARDO AUGUSTO DA COSTA

# Processamento Digital de Sinais para Simulação de Efeito 3D em Áudio em Fones de Ouvido

Porto Alegre

2018

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

EDUARDO AUGUSTO DA COSTA

# Processamento Digital de Sinais para Simulação de Efeito 3D em Áudio em Fones de Ouvido

Projeto de Pesquisa para o Projeto de Diplomação a ser apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

Orientador: Prof. Dr. Altamiro Amadeu Susin

Porto Alegre
2018

EDUARDO AUGUSTO DA COSTA

# Processamento Digital de Sinais para Simulação de Efeito 3D em Áudio em Fones de Ouvido

Projeto de Pesquisa para o Projeto de Diplomação a ser apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

**Prof. Dr. Altamiro Amadeu Susin**
Orientador - UFRGS

**Prof. Dr. Ály Ferreira Flores Filho**
Chefe do Departamento de Engenharia Elétrica (DELET) - UFRGS

Aprovado em 4 de julho de 2018.

BANCA EXAMINADORA

**Prof. Dr. Altamiro Amadeu Susin**
UFRGS

**Prof. Msc. Joel Augusto Luft**
IFRS

**Prof. Dr. Tiago Roberto Balen**
UFRGS

*A Margarete, minha mãe, Tenório, meu pai, e a todos que me ofereceram suporte nessa longa caminhada.*

# Agradecimentos

Aos colaboradores e pesquisadores do Laboratório de Processamento de Sinais em Imagens.

# Resumo

O efeito de áudio 3D, também conhecido como áudio *espacializado*, pode gerar a percepção de fontes de áudio virtuais, com origem em diferentes localizações no espaço ao redor do usuário. Algumas das aplicações para este efeito apresentadas na literatura são: realidade virtual para videogames – na qual pode ser usado para criar um ambiente imersivo melhor com áudio para o jogador – e na realidade aumentada – como, por exemplo, para sistemas melhorados de prevenção a obstáculos para pessoas com deficiência visual. Nesse projeto, então, desenvolveu-se um programa de computador que gera o efeito do áudio 3D para fones de ouvido em tempo real, para uma fonte de áudio virtual de posição programável, e que tem como entrada a posição da cabeça do usuário – detectada por um sensor. O desenvolvimento desse programa exigiu tantos conhecimentos de *psicoacústica* quanto de programação, bem como outros, que são abordados ao longo do texto.

**Palavras-chave**: áudio 3D, espacialização, áudio binaural, processamento digital de sinais.

# Abstract

The effect of 3D Audio, also known as spatialised audio, can be generated through signal processing and thus create the perception of virtual audio sources, with their origins in different positions in the space around the user. Some applications presented in literature are: virtual reality for digital video games – at which it can be used to create a better immersive environment with virtual audio cues for the player – and augmented reality — for improved systems of obstacle avoidance for visually impaired people, for example. In this project, a computer software was developed, capable of generating 3D audio in real time for ear phones, for a given virtual audio source with a programmable position, and for a given user's head position detected by a sensor. To develop this programme, a wide range of knowledge was required, such as *psycho-acoustics* as much as programming skills, and they are discussed in the following texts.

**Keywords**: 3D audio, spatialisation, binaural audio, digital signal processing.

# Lista de Figuras


Figura 1 – Efeito de Áudio 3D gerado com um arranjo de fontes de som em um ambiente. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
Figura 2 – Efeito de Áudio 3D gerado com uso de fones de ouvido. . . . . . . . . . 14
Figura 3 – Modelo de *ITD*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
Figura 4 – Modelo de *ILD*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
Figura 5 – *ILD* em função de ângulo e frequência. . . . . . . . . . . . . . . . . . . 26
Figura 6 – Mecanismo da Pinnae. . . . . . . . . . . . . . . . . . . . . . . . . . 28
Figura 7 – Processo de treinamento para corrigir a reversão frente-atrás. . . . . . . 29
Figura 8 – Modelo do ambiente traduzido para o fone de ouvido. . . . . . . . . . . 31
Figura 9 – Sistemas de coordenada relacionado à cabeça. . . . . . . . . . . . . . . 32
Figura 10 – Exemplo de HRTF e HRIR relacionados. . . . . . . . . . . . . . . . . 32
Figura 11 – Coordenadas retangulares (x,y,z). . . . . . . . . . . . . . . . . . . . . 34
Figura 12 – Coordenadas esféricas ($r$,$\theta$,$\phi$). . . . . . . . . . . . . . . . . . . . . . 36
Figura 13 – Blocos de um sistema para a geração de áudio 3D. . . . . . . . . . . . 39
Figura 14 – Diagrama de blocos da proposta de solução. . . . . . . . . . . . . . . . 43
Figura 15 – Modelo do ambiente traduzido para o fone de ouvido, antes e após o movimento do usuário. . . . . . . . . . . . . . . . . . . . . . . . . . . 44
Figura 16 – Áudio: trecho de voz humana gravada com microfone. . . . . . . . . . . 47
Figura 18 – Trecho de voz humana com o efeito 3D aplicado pelo segundo algoritmo em MATLAB. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
Figura 17 – Trecho de voz humana com o efeito 3D aplicado pelo primeiro algoritmo em MATLAB. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
Figura 19 – Fluxo de áudio. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
Figura 20 – Diagrama de blocos abstraído. . . . . . . . . . . . . . . . . . . . . . . 50
Figura 21 – Blocos do programa. . . . . . . . . . . . . . . . . . . . . . . . . . . 58
Figura 22 – Bloco *virtual channel*, parte 1. . . . . . . . . . . . . . . . . . . . . . 58
Figura 23 – Bloco *virtual channel*, parte 2. . . . . . . . . . . . . . . . . . . . . . 58

# Lista de Abreviaturas e Siglas

| | |
|---|---|
| AR | *Augmented Reality* |
| HRTF | *Head Related Transfer Function* |
| HRIR | *Head Related Impulse Response* |
| BRIR | *Binaural Room Impulse Response* |
| LTI | *Linear and Time Invariant* |
| ITD | *Interaural Time Differences* |
| ILD | *Interaural Level Differences* |
| DFT | *Discrete Fourier Transform* |
| FFT | *Fast Fourier Transform* |
| API | *Application Program Interface* |
| AES | *Audio Engineering Society* |
| SOFA | *Spatially Oriented Format for Acoustics* |

# Sumário


| | | |
|---|---|---|
| **1** | **INTRODUÇÃO** | **12** |
| **1.1** | **Justificativa** | **12** |
| **1.2** | **Objetivo** | **13** |
| **1.3** | **Objetivos Secundários** | **14** |
| **2** | **REVISÃO BIBLIOGRÁFICA** | **16** |
| **2.1** | **Sistemas e Sinais** | **16** |
| 2.1.1 | Representação de sinais discretos | 16 |
| 2.1.2 | Descrição de um Sistemas LTI pela Resposta ao Impulso | 16 |
| 2.1.3 | Resposta em Frequência e Transformada de Fourier | 18 |
| 2.1.4 | *Fast Fourier Transform (FFT)* | 18 |
| 2.1.5 | Processo de Filtragem com os algoritmos: *overlap and add* e *overlap and save* | 19 |
| 2.1.5.1 | *Overlap and save* | 20 |
| 2.1.5.2 | *Overlap and add* | 21 |
| **2.2** | ***Auralization*** | **21** |
| **2.3** | **Percepção auditória e Psicoacústica** | **22** |
| **2.4** | **Localização Sonora e Audição Binaural** | **22** |
| 2.4.1 | Áudio Estéreo e Áudio Binaural | 22 |
| 2.4.2 | Diferenças de Tempo Interaurais | 23 |
| 2.4.3 | Diferenças de Nível Interaural | 24 |
| 2.4.4 | Fones de Ouvido e a Lateralização | 26 |
| 2.4.5 | Lei da primeira onda | 27 |
| 2.4.6 | Questão de percepção de Localização Frente-Atrás | 27 |
| **2.5** | ***HRTF* e *HRIR*** | **30** |
| **2.6** | ***BRIR*** | **31** |
| **2.7** | **Arquivos *SOFA*** | **32** |
| 2.7.1 | *CIPIC* | 33 |
| **2.8** | **Posições e coordenadas** | **33** |
| 2.8.1 | Coordenadas retangulares | 33 |
| 2.8.2 | Coordenadas esféricas | 34 |
| 2.8.3 | Relação entre os sistemas de coordenadas | 35 |
| 2.8.4 | Rotacionando vetores em torno dos eixos (x,y,z) | 36 |
| **2.9** | **Trabalhos Anteriores** | **37** |
| 2.9.1 | Geração de áudio 3D através do seguimento do movimento da cabeça | 37 |
| 2.9.2 | *HRTF* e *head-tracker* | 38 |

2.9.3 *HRTF* e outras combinações de elementos para a geração de áudio 3D . . 38
**2.10 Tópicos de programação . . . 38**
2.10.1 *Threads* . . . 39
2.10.2 *Mutexes* . . . 40
2.10.3 Conectando com *Bluetooth* . . . 40

**3 IMPLEMENTAÇÃO . . . 42**
**3.1 Proposta de solução . . . 42**
**3.2 Ferramentas utilizadas . . . 45**
**3.3 Testes em MATLAB para a geração do efeito binaural . . . 45**
**3.4 Programa em C++ no macOS . . . 48**
3.4.1 Diagrama de blocos abstraído . . . 49
3.4.2 Considerações adicionais da implementação . . . 51
3.4.3 Bibliotecas de *software* . . . 51
3.4.3.1 Comunicação *bluetooth* . . . 51
3.4.3.2 Manipulação de Arquivos *SOFA* . . . 52
3.4.3.3 Fluxo de áudio dentro do sistema operacional . . . 53
3.4.3.4 Matemática por matrizes . . . 54
3.4.3.5 *Fast Fourier Transform* . . . 54
3.4.4 Particularidades da implementação algorítmica . . . 55
3.4.4.1 Posicionamento relativo . . . 55
3.4.4.1.1 Sensor de posição do usuário . . . 56
3.4.4.2 Linearização da convolução no domínio frequência . . . 56
3.4.4.3 Organização das *Threads* . . . 57
3.4.5 Implementação do Programa . . . 57
3.4.5.1 Interface com o usuário . . . 59
3.4.5.2 Bloco de captura e reprodução de áudio - port audio stream . . . 59
3.4.5.3 Bloco de utilidades - *utils* . . . 60
3.4.5.4 Bloco agregador - dois canais em um vetor de dados . . . 61
3.4.5.5 Bloco de aplicação de filtro . . . 61
3.4.5.6 Bloco *mySOFA* . . . 61
3.4.5.7 Bloco de *fast Fourier transform* . . . 61
3.4.5.8 Bloco do conector com a *HeadTracker*: *head position* . . . 62
3.4.5.9 Bloco de canais virtuais . . . 62

**4 RESULTADOS . . . 63**

**5 CONCLUSÃO . . . 65**

**REFERÊNCIAS BIBLIOGRÁFICAS . . . 66**

**ANEXOS** 69

ANEXO A – CÓDIGO FONTE DOS PROGRAMAS DE TESTE REALIZADOS EM *MATLAB* . . . . . . . . . . . . . . . 70

ANEXO B – CÓDIGO FONTE DO PROGRAMA C++ . . . . . . . . 73

**APÊNDICES** 110

APÊNDICE A – USO DA API SOFA . . . . . . . . . . . . . . . . . . . 111

APÊNDICE B – ALGORITMOS PARA SOM 3D . . . . . . . . . . . 112

# 1 Introdução

No presente projeto visa-se implementar algoritmos de processamento digital de áudio em tempo real que gerem a percepção de áudio 3D quando o usuário usar fones de ouvido. O efeito de áudio 3D, também conhecido como áudio *espacializado*, quando gerado para fones de ouvido pode gerar a percepção de fontes de áudio virtuais, com origem em diferentes localizações no espaço ao redor do usuário (ALGAZI et al., 2001). Existem muitas aplicações para este tipo de sistema, como será mostrado na justificativa. Intenciona-se criar um sistema para a geração desse efeito.

Inicialmente a justificativa é apresentada, seguida dos objetivos — geral e secundários. Segue-se com a revisão bibliográfica, implementação do programa, apresentação dos resultados, e finalmente a conclusão.

## 1.1 Justificativa

Conforme mencionado brevemente na introdução, existem muitas aplicações do efeito de áudio 3D, incluindo situações em que esse efeito pode melhorar a qualidade geral de um sistema. Algumas das aplicações apresentadas na literatura são realidade virtual e realidade aumentada. A realidade virtual é utilizada em videogames (FITZPATRICK et al., 2013) (WU; YU, 2016), nos quais o efeito pode ser usado para criar um ambiente imersivo melhor com pistas de áudio virtuais para o jogador. Para a realidade aumentada (do inglês *Augmented Reality*, *AR*) (SUNDARESWARAN et al., 2003) tem ampla aplicabilidade – como para o uso em sistemas melhorados de prevenção a obstáculo para pessoas com deficiência visual (PEŁCZYŃSKI; BOCHEŃSKA, 2012): onde um alerta para avisar que há um obstáculo é tocado, um aviso direcionado como vindo da posição do obstáculo, pode ser usado para melhorar esse sistema (SANDBERG et al., 2006), por exemplo. Além disso, a *AR* pode ser usado para ajudar na recuperação de desastres, situações apresentadas na literatura, como o uso de realidade aumentada para avaliação rápida de estruturas atingidas por terremoto (KAMAT; EL-TAWIL, 2007), na qual o áudio 3D não foi usado – é discutida a viabilidade de avaliar danos estruturais usando ambos imagem e *AR* –, o áudio 3D poderia melhorar o sistema: através da reprodução de áudio provindo, virtualmente, a partir da posição das áreas de interesse, poder-se-ia permitir ao usuário do dispositivo de realidade aumentada fazer uma identificação mais rápida de áreas a serem verificadas primeiro ou chamar a atenção do mesmo para áreas perigosamente danificadas. O efeito de áudio 3D também pode ser usado por bombeiros para identificar a posição de seus colegas de forma mais rápida e fácil em um incêndio (PEŁCZYŃSKI; BOCHEŃSKA, 2012): ao

usar um dispositivo de comunicação com áudio 3D, o mesmo pode reproduzir o áudio recebido em uma fonte virtual que fique na mesma direção da posição real do transmissor.

Além de sua aplicabilidade mais ampla, existem muitas ferramentas que podem ser usadas para implementar o efeito 3D de áudio. Alguns autores implementam com o uso de *HRTF* (Head Related Transfer Function – Função de Transferência da Cabeça), que com certeza é, conforme expresso na literatura (FITZPATRICK et al., 2013), um ponto chave na geração desse efeito. O *HRTF* contém as pistas que a onda sonora projetada sobre uma pessoa gera, usada pela pessoa para identificar essa origem de áudio. No entanto, traz uma complicação: cada pessoa possui suas *HRTF* específicas. Portanto, há uma grande complexidade envolvida em um sistema de áudio 3D com uso de *HRTF* para lhe proporcionar mais precisão. No entanto, existem outros elementos, como BRIP (Binaural Room Impulse Response – Resposta ao Impulso Binaural de uma Sala) (WABNITZ et al., 2010) que podem ser usados para melhorar a percepção geral de áudio 3D também. Além disso, há áreas a serem trabalhadas que podem melhorar muito a criação do efeito 3D, como, por exemplo, o problema de "reversão frontal". O problema de "reversão frontal"ocorre quando o usuário de um sistema de áudio 3D não pode distinguir se a fonte de áudio está atrás ou na frente do mesmo (CHO et al., 2006) (ZAHORIK et al., 2006). Em um estudo sobre a recalibração perceptual de localição sonora humana (ZAHORIK et al., 2006), propõe-se que o treinamento do usuário do sistema ajude a resolver esse problema de localização. Assim, é importante avaliar os diferentes algoritmos, sua influência na precisão da simulação de áudio 3D.

Finalizando, existem instrumentos (ou melhor, aplicações) que podem ser melhorados com o uso do efeito de áudio 3D, e um programa que gere esse efeito em tempo real, desenvolvido em C++ pode servir de base para demais projetos que podem se beneficiar do uso do áudio 3D.

## 1.2 Objetivo

Objetiva-se implementar algoritmos de processamento digital de áudio em tempo real que gerem a percepção de áudio 3D quando o usuário usar fones de ouvido. A direção de chegada do áudio, sendo a mesma conhecida, deverá ser corrigida conforme o usuário movimente a cabeça, para simular uma fonte fixa no espaço.

O efeito demonstrado na figura 1 – na qual é reprensentado uma pessoa no centro de um arranjo de fontes de áudio (ou ainda, *caixas de som*), de forma que o áudio pode, além de ser reproduzido das diversas fontes, gerar a sensação de movimento, quando, por exemplo, em um filme o áudio é deslocado de uma fonte sonora para outra – é gerado através de processamento digital, com aplicação de filtros. Com a utilização de um par de

fones de ouvidos, uma sensação similar a da imagem 1 é criada – essa nova situação está representado na figura 2.

Figura 1 – Efeito de Áudio 3D gerado com um arranjo de fontes de som em um ambiente.

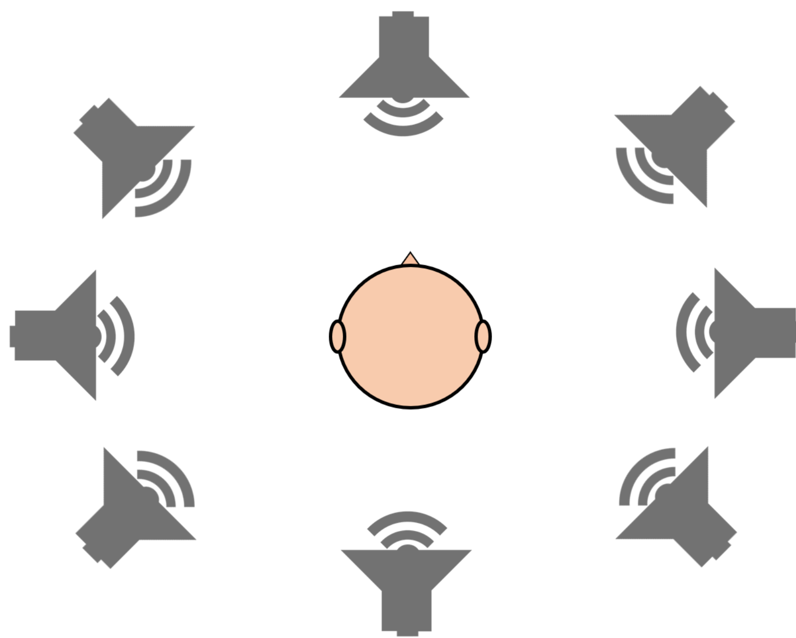
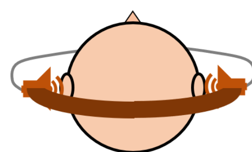


Figura 2 – Efeito de Áudio 3D gerado com uso de fones de ouvido.



## 1.3 Objetivos Secundários

Estudo de áudio digital, processamento de sinais digitais, funções de transferência, reverberação, transformadas para domínio frequência, psicoacústica; desenvolvimento de programa para microcomputador, aprendizado do uso de múltiplas *threads*, de *mutex*es e do *bluetooth*.

Desenvolvimento de uma versão em tempo real do sistema descrito no objetivo: criar um *player 3D* que processe um uma *stream* de áudio, e que sendo alimentado por um

sensor de posição de cabeça (*head-tracker*), e gere o efeito 3D com a correção da posição da fonte do som. O player deverá ser capaz de reproduzir um áudio de dois canais. Será feito o uso de um sensor de posição, com comunicação através de *bluetooth*, mas a construção do mesmo do dispositivo propriamente dita não está no escopo do presente trabalho.

# 2 Revisão Bibliográfica

Inicia-se esse capítulo revisando as propriedades de sistemas e sinais essenciais à esse trabalho. Seguindo, são feitas definições de conceitos de auralização (*auralization* do inglês) e psicoacústica. Após, são revisadas características adicionais sobre a geração de áudio 3D.

## 2.1 Sistemas e Sinais

Segundo o livro *Discrete-Time Signal Processing* (OPPENHEIM; SCHAFER, 2009), o termo *sinal* é genericamente aplicado a algo que transporte informação. Sinais são representados usando funções matemáticas dependentes de uma ou mais variáveis, e especificamente um sinal de áudio é representado matematicamente em função do tempo. Sinais podem ser contínuos ou discretos, e se contínuos serão portanto representados por uma variável independente contínua. Já se discretos, serão representados por uma variável independente de valores discretos. No presente trabalho são tratados sinais Digitais de áudio, que novamente segundo (OPPENHEIM; SCHAFER, 2009), por serem sinais digitais, são discretos tanto nos níveis de amplitude quanto tempo.

### 2.1.1 Representação de sinais discretos

Um sinal discreto provindo da amostragem de um sinal analógico pode ser representado no formato da equação 1 (OPPENHEIM; SCHAFER, 2009), para uma sequência de números $x$, no qual o $n$ésimo número é representado.

$$x[n] = x_a(nT), -\infty < n < \infty \tag{1}$$

na qual $n$ é a posição da sequência discreta de números discretos, $T$ é o período de amostragem e o seu inverso a frequência de amostragem, $x[n]$ é o valor da amostra da sequência (esse nome é usado mesmo quando o sinal não provém de uma amostragem de um sinal analógico).

### 2.1.2 Descrição de um Sistemas LTI pela Resposta ao Impulso

Um sistema LTI é um sistema que é simultaneamente linear e invariante no tempo (OPPENHEIM; SCHAFER, 2009). Um sistema invariante no tempo é um sistema no qual uma variação de fase ou atraso na sequência de entrada desse sistema causa uma mudança correspondente em sua saída (OPPENHEIM; SCHAFER, 2009), ou seja, para

um alteração na entrada no formato da equação 2, para toda constante $n_0$, tem-se uma saída correspondente na forma da equação 3. E um sistema linear é um sistema definido pela superposição: se uma entrada no formato da equação 4 for aplicada nesse sistema, uma saída no formato da equação 5 será gerada (OPPENHEIM; SCHAFER, 2009).

$$x_1[n] = x[n - n_0] \tag{2}$$

$$y_1[n] = y[n - n_0] \tag{3}$$

nas quais $x[n]$ é a entrada original do sistema, $n_0$ é uma constante de atraso, $x_1[n]$ é a nova entrada transladada por esse atraso. De forma análoga, $y[n]$ é a saída original do sistema, $y_1[n]$ é a nova saída transladada por esse atraso.

$$x[n] = \sum_k a_k x_k[n] \tag{4}$$

$$y[n] = \sum_k a_k y_k[n] \tag{5}$$

nas quais $x[n]$ é o resultado da soma dos $k$ sinais de entrada superpostos, $a_k$ são os coeficientes de cada sinal individual $x_k[n]$ de entrada, $y_k[n]$ é cada um dos sinais de saída superpostos pelo somatório que resulta no sinal $y[n]$.

Um sistema que seja LTI terá como propriedade que ele pode ser completamente caracterizado por sua resposta ao impulso (OPPENHEIM; SCHAFER, 2009). As equações 6, 7 e 8 descrevem essa propriedade.

$$y[n] = T\{\sum_{k=-\infty}^{\infty} x[k]\delta[n-k]\} \tag{6}$$

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]T\{\delta[n-k]\} = \sum_{k=-\infty}^{\infty} x[k]h_k[n] \tag{7}$$

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k], \forall n \tag{8}$$

nas quais T é uma transformação linear, que mapeia uma dada entrada em uma dada saída, $\delta$ é a função impulso, $y[n]$ é o sinal de saída do sistema, $x[k]$ é a entrada e $h[n]$ é a descrição impulsiva do sistema.

Uma forma alternativa de representar essa propriedade é pelo operador de convolução, presente na equação 9 (OPPENHEIM; SCHAFER, 2009).

$$y[n] = x[k] * h[n], \forall n \tag{9}$$

na qual $y[n]$ é o sinal de saída do sistema, $x[k]$ é a entrada e $h[n]$ é a descrição impulsiva do sistema.

### 2.1.3 Resposta em Frequência e Transformada de Fourier

Muitos sinais podem ser representados na forma do par de equações da *Transformada de Fourier* 10 e 11 (OPPENHEIM; SCHAFER, 2009), na qual a equação 11 é a transformada inversa de Fourier.

$$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\omega n})e^{j\omega n}d\omega \tag{10}$$

$$X(e^{j\omega n}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{11}$$

na qual $x[n]$ é o sinal discreto no tempo, $X(e^{j\omega n})$ é a transformada inversa de Fourier a dos sistema.

A reposta ao impulso pode ser obtida da resposta em frequência conforme a equação 12 (OPPENHEIM; SCHAFER, 2009). Utilizando as propriedades da transformada de Fourier tem-se também que a convolução torna-se uma multiplicação conforme mostrado na equação 13 (OPPENHEIM; SCHAFER, 2009).

$$h[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(e^{j\omega n})e^{j\omega n}d\omega \tag{12}$$

na qual $h[n]$ é a resposta ao impulso no domínio tempo $H(e^{j\omega n})$ é a resposta em frequência ao impulso do sistema.

$$Y(e^{j\omega n}) = H(e^{j\omega n})X(e^{j\omega n}) \tag{13}$$

na qual $Y(e^{j\omega n})$ é a saída do sistema, $H(e^{j\omega n})$ é a caracterização do sistema (a resposta ao impulso do mesmo), $X(e^{j\omega n})$ é o sinal de entrada.

### 2.1.4 *Fast Fourier Transform (FFT)*

Segundo a literatura (OPPENHEIM; SCHAFER, 2009), os algoritmos de *FFTs* – Transformada Rápida de Fourier, do inglês *Fast Fourier Transform* – são algoritmos que

calculam de forma rápida e eficiente a transformada discreta de Fourier ($DFT$), que é um caso especial para sistemas discretos da Transformada de Fourier. A equação 14 define a $DFT$ e a equação 15 é a transformada inversa da $DFT$. A inversa da Transformada Rápida de Fourier é conhecida como $iFFT$.

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, k = 0, 1, ..., N - 1. \tag{14}$$

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn}, n = 0, 1, ..., N - 1. \tag{15}$$

na qual $W_N = e^{-j(2\pi/N)}$, $N$ é o período ou o número de amostras, $x[n]$ é o sinal no domínio tempo e $X[k]$ é o sinal no domínio frequência.

Interessante notar que a $FFT$ é uma solução numérica exata e não uma aproximação numérica, e ela pode somente ser calculada em blocos de tamanho $N$ dado pela equação 16 (VORLÄNDER, 2007).

$$N = 2^m, (2, 4, 8, 16, 32, ...). \tag{16}$$

### 2.1.5 Processo de Filtragem com os algoritmos: *overlap and add* e *overlap and save*

No processo de filtragem aplica-se uma transformação linear a um sinal, que mapeia dadas entradas a dadas saídas do sistema. Esse processo é o de convolução, definido pela equação 9 (da seção 2.1.2). É descrito ainda pelas equações 12 e 13, a primeira no domínio tempo e a segunda no domínio frequência.

Com a realização da convolução de um sinal $x$ com um filtro impulsivo $h$ tem-se uma nova saída $y$, que terá o sinal de entrada com as características da resposta impulsiva aplicada. Para a geração do efeito de áudio 3D, o processo de filtragem é importate: é com o uso de filtros específicos (chamados de *HRTF*, a serem explicados nas próximas seções) que o efeito é aplicado a um áudio não-3D.

Quando uma convolução é realizada em um sinal no domínio frequência que foi gerado pela transformada rápida de Fourier (ou seja, uma multiplicação é realizada), segundo (SHYNK et al., 1992), ela será uma convolução circular. Uma característica dessa convolução é que a sequência de saída da mesma parece ser de um filtro com variação temporal periódica, conforme (PELKOWITZ, 1981 apud SHYNK et al., 1992) avaliou. Para que isso não ocorra, é desejável usar-se uma convolução linear. Sabe-se, da literatura, que existem algoritmos que proporcionam isso. De (OPPENHEIM; SCHAFER, 2009), assumindo uma sequência $x[n]$ de $L$ amostras, uma sequência $h[n]$ de $P$ amostras, para

que, ao convoluir as duas sequências através algoritmo de transformada discreta de Fourier – como a *FFT* –, obtenha-se $y[n]$ de $N$ amostras e tenha-se uma convolução linear, é necessário que $N$ satisfaça a equação 17.

$$N \geqslant L + P - 1 \tag{17}$$

Adicionalmente, também de (OPPENHEIM; SCHAFER, 2009), para que não seja necessário ter todo o sinal já na memória para computar a transformada discreta de Fourier, e portanto não tenha que ser esperado por todo o sinal estar pronto, usa-se convoluções de bloco, onde o sinal a ser processado é segmentado em blocos de tamanho $L$. Esses blocos então, são segmentados e processados, de forma a obter-se a convolução linear, nos métodos mais conhecidos: *overlap and save* e *overlap and add.*

#### 2.1.5.1 *Overlap and save*

No primeiro caso de convolução de bloco, adaptando-se de (KUNDUR, ) tem-se o algoritmo chamado de *overlap and save* (sobrepõe e salva).Considerando um sinal de entrada $x[n]$ de $L$ pontos e um filtro $h[n]$ de $P$ pontos, para obter-se a convolução $y[n]$ desses sinais, com $h$ preparado da seguinte forma:

- deve ter zeros adicionados até ficar do tamanho de $N = L + P - 1$ pontos.
- deve ter a transformada rápida $(FFT)$ $H$ de $N$ pontos calculada.

E com o sinal $x$ preparado da seguinte forma:

- deve ter inseridos $P - 1$ zeros no seu início.
- deve ser dividido em blocos $x_{bloco}$ de tamanho $N = L + P - 1$ pontos, com uma sobreposição entre os blocos de $P - 1$ pontos.

E para cada bloco $x_{bloco}$ ter-se-á:

- a transformada $(FFT)$ $X_{bloco}$ de $N$ pontos calculada.
- a saída $Y_{bloco}$ calculada, pelo produto de $X_{bloco}$ com $H$, conforme a equação 13.
- a transformada inversa de Fourier $(iFFT)$ $y_{bloco}$ de $Y_{bloco}$ calculada.
- a saída $y_{bloco}$ de cada bloco deve ter seus primeiros $P - 1$ pontos descartados.

Obtém-se a saída $y$, juntando os blocos $y_{bloco}$.

#### 2.1.5.2 *Overlap and add*

No segundo caso de convolução de bloco, adaptando-se de (KUNDUR, ), tem-se o algoritmo chamado de *overlap and add* (sobrepõe e soma). Considerando um sinal de entrada $x[n]$ de $L$ pontos e um filtro $h[n]$ de $P$ pontos, para obter-se a convolução $y[n]$ desses sinais, com $h$ preparado da seguinte forma:

- deve ter zeros adicionados até ficar do tamanho de $N = L + P - 1$ pontos.
- deve ter a transformada rápida $(FFT)$ $H$ de $N$ pontos calculada.

E com o sinal $x$ preparado da seguinte forma:

- deve ser dividido em blocos $x_{bloco}$ de tamanho $L$ pontos, sem sobreposição entre os blocos.
- deve ter zeros adicionados ao final de cada $x_{bloco}$ até ficar do tamanho de $N = L+P-1$ pontos.

E para cada bloco $x_{bloco}$ ter-se-á:

- a transformada $(FFT)$ $X_{bloco}$ de $N$ pontos calculada.
- a saída $Y_{bloco}$ calculada, pelo produto de $X_{bloco}$ com $H$, conforme a equação 13.
- a transformada inversa de Fourier $(iFFT)$ $y_{bloco}$ de $Y_{bloco}$ calculada.
- o bloco de saída $y_{saída}$ é formado através de dois blocos $y_{bloco}$ consecutivos, formado adicionando os últimos $P-1$ pontos de $y_{bloco}$ com os primeiros $P-1$ pontos de $y_{bloco}$ seguinte, ou seja $y_{saída[1]}[P-1:N] = y_{bloco[1]}[P-1:N] + y_{bloco[2]}[1:P-1]$.

Obtém-se a saída $y$, juntando os blocos $y_{saída}$.

## 2.2 *Auralization*

Segundo o livro que têm esse título em inglês, chamado *Auralization* (VORLÄNDER, 2007), os autores a definem como a técnica de criar arquivos de som audíveis a partir de dados numéricos simulados, medidos ou sintetizados. Esse é um importante conceito a ser considerado no presente trabalho, uma vez que a partir de modelos matemáticos, serão simulados os efeitos de espacialização do áudio, e esse é, portanto, um trabalho de *auralização*.

## 2.3 Percepção auditória e Psicoacústica

Segundo o dicionário de língua inglesa Oxford (OXFORD, 2003), acústica (*acoustics* em inglês) se refere às propriedades ou qualidades de uma sala ou construção, que determina como o som é transmitido nela. Para o presente trabalho, no entanto, apenas o estudo genérico de acústica não é interessante quanto o estudo da psicoacústica. Essa, novamente segundo o dicionário Oxford (OXFORD, 2003), (*psychoacoustics* em inglês) é o braço da psicologia interessada com a percepção do som e seus efeitos psicológicos. De forma selecionada, serão abordados os tópicos que se relacionam com o áudio 3D dentro da psicoacústica. De uma forma expandida, tem-se a definição presente no livro *Auditory Perception of Sound Sources* (YOST; FAY, 2007), no capítulo *Percebendo Fontes Sonoras*, no qual o autor diz que a sensação auditória pode ser vista como o processamento de atribuições físicas do som (frequência, nível...) pela pessoa, mas a percepção auditória adiciona o processamento dessas características; isso, então, permite a um organismo lidar com as fontes que produziram o som, assim determinando as fontes do mesmo.

## 2.4 Localização Sonora e Audição Binaural

A capacidade humana de localizar fontes sonoras, segundo (VORLÄNDER, 2007), (HOWARD, 2017) e (ZWICKER; FASTL, 2013) se dá por termos duas orelhas, e o nome desse *modo auditivo* é audição binaural. Como nesse trabalho busca-se a criação virtual dessas fontes, os principais efeitos que geram essa capacidade serão descritos nesse capítulo. E são eles: as Diferenças de Tempo Interaurais (*Interaural Time Differences*, *ITD*) e as Diferenças de Níveis Interaurais (*Interaural Level Differences*, *ILD*). Todavia, outras características conceituais importantes, como o efeito de lateralização – que busca-se ser anulado –, pertinente aos fones de ouvido, e a questão da localização frente-atrás também são vistos.

Interessante salientar que as obras consultadas diferem entre si ao nomear essas características. Para manter a coerência ao longo do presente trabalho, será usada a nomenclatura disponível em *Auralization* (VORLÄNDER, 2007).

### 2.4.1 Áudio Estéreo e Áudio Binaural

O áudio estéreo, é um áudio armazenado com duas faixas de áudio, geralmente uma faixa para a fonte de som da esquerda e outra para a fonte da direita. O áudio mono é um áudio com apenas uma faixa.

Já o áudio binaural é o áudio *localizável*. De (VORLÄNDER, 2007), os humanos podem localizar a origem dos sons graças a audição binaural. Em um exemplo citado no livro, o autor descreve um ambiente onde as ondas sonoras podem fluir normalmente, e uma

onda plana sonora é emitida a partir de uma direção particular. Essa onda é perturbada pelo ambiente, e pelo próprio corpo da pessoa. A partir das citadas *ITD* e *ILD* a pessoa identifica a origem do áudio.

A importante distinção a ser feita aqui é: um áudio binaural obrigatorimente tem duas faixas de áudio após processado, uma para cada ouvido da pessoa que a escutará, porém pode ser originado de um som mono, que, se ocorresse sem o fone de ouvido e originado em uma fonte de áudio distante do sujeito que está a escutar, geraria sinais diferentes para as duas orelhas desse sujeito. Esses mecanismos para a interpretação do áudio binaural serão detalhados nas próximas seções.

### 2.4.2 Diferenças de Tempo Interaurais

Diferenças de Tempo Interaurais, do inglês *Interaural Time Differences (ITD)*, é o intervalo de tempo que leva de uma onda sonora plana ser captada pelo primeiro ouvido (mais próximo da fonte de áudio) até ser captada pelo segundo ouvido (mais distante da fonte de áudio) (HOWARD, 2017). A *ITD* é uma pista que a mente humana usa para localizar a origem de uma fonte de áudio. Observe a figura 3, na qual uma fonte de áudio está a uma distância grande do receptor (maior que um metro) e emite uma onda de som – que ao chegar ao receptor é plana e portanto é representada na gravura como um grande ponto –, e essa onda é capturada pelos dois ouvidos do receptor em tempos diferentes. Esse tempo pode ser descrito pela equação 18, provinda da literatura (HOWARD, 2017).

$$ITD = \frac{r(\theta + sin(\theta))}{c} \tag{18}$$

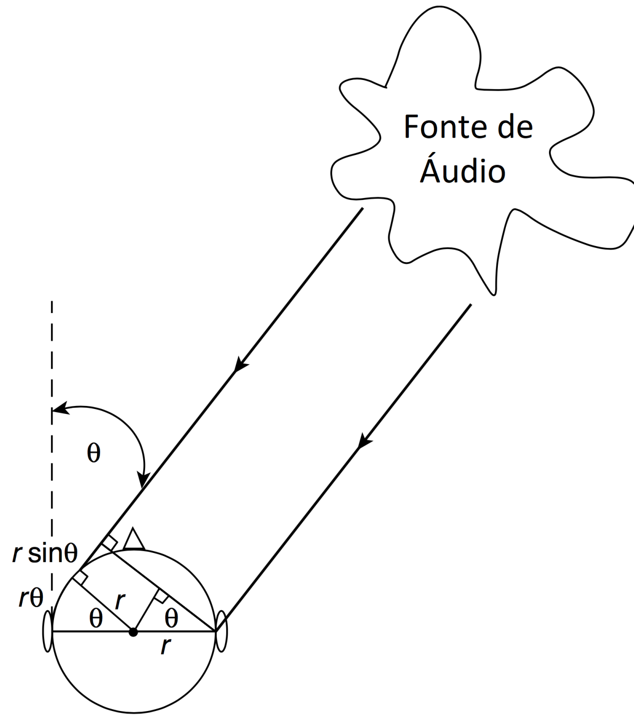
na qual $ITD$ é a diferença de tempo interaural, $r$ é a metade da distância entre os ouvidos, $\theta$ é o ângulo de incidência da onda sonora, em relação a uma reta normal à reta que liga os dois ouvidos (em radianos), $c$ é a velocidade do som (dado em $m/s$). Na literatura, o valor de *ITD* máximo é calculado, e para um $r = 0.09m$, esse valor é de $673\mu s$ (HOWARD, 2017).

Ainda segundo a literatura, essa característica presente no som, de detecção de posição, funciona em frequências menores que para o outro método – chamado *ILD* – (a frequência mais alta que a *ITD* atua pode ser determinada pela equação 20, a ser explicada). Isso ocorre porquê o sistema auditivo usa a mudança de fase entre um ouvido e outro para resolver a direção (HOWARD, 2017), e a fase é descrita na equação 19.

$$\phi_{ITD} = 2\pi fr(\theta + sin(\theta)) \tag{19}$$

na qual $\phi_{ITD}$ é fase interaural, $r$ é a metade da distância entre os ouvidos, $\theta$ é o ângulo de incidência da onda sonora, em relação a uma reta normal à reta que liga os dois ouvidos

Figura 3 – Modelo de *ITD*.




Adaptado de: (HOWARD, 2017).

(em radianos), $f$ é a frequência do som em questão (dado em $Hz$). Pode-se observar que quando essa mudança de fase for superior a 180°, haverá uma ambiguidade de posições possíveis para a fonte sonora.

Pode-se ainda estabelecer qual a frequência máxima, para um dado ângulo de incidência da onda do som, que esse método de localização funciona (HOWARD, 2017), calculado pela equação 20.

$$f_{max}(\theta) = \frac{c}{2 \times r \times (\theta + sin(\theta))} \tag{20}$$

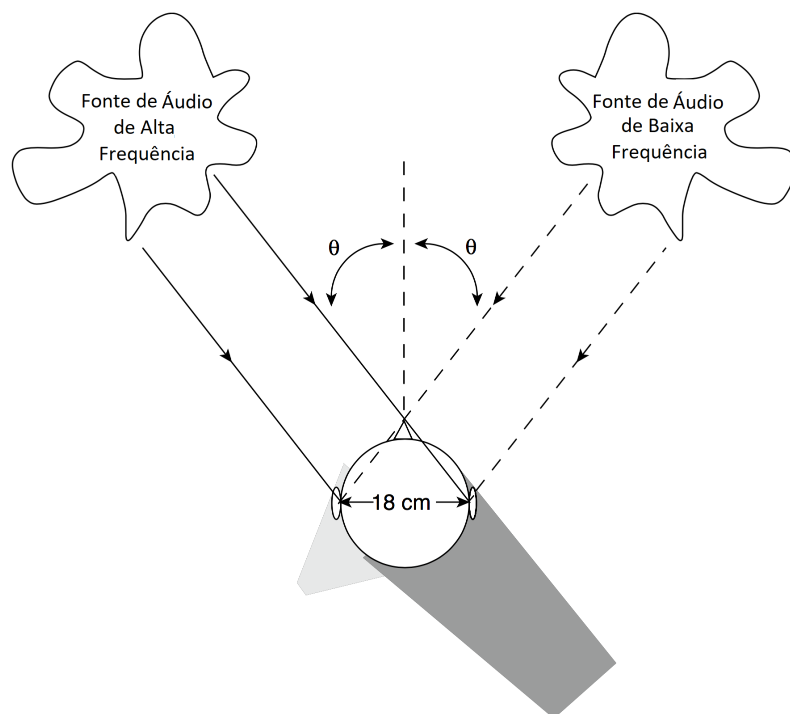
na qual $f_{max}(\theta)$ é a frequência máxima para um dado valor do ângulo de incidência da onda sonora, $r$ é a metade da distância entre os ouvidos, $c$ a velocidade do som em $m/s$ $\theta$ é o ângulo de chegada da onda sonora, em relação a uma reta normal à reta que liga os dois ouvidos (em radianos).

### 2.4.3 Diferenças de Nível Interaural

As Diferenças de Nível Interaural, do inglês *Interaural Level Differences* (*ILD*) (e também presente na literatura como *Interaural Intensity Differences – IID*), é diferença

de intensidade no som captado pelas orelhas que é causado pelo efeito de *shading* da cabeça (HOWARD, 2017), que está ilustrada na figura 4 – na qual vê-se que o efeito é mais relevante para frequências mais altas – um mínimo é calculável (a frequência mínima de atuação da *ILD* pode ser calculada pela equação 21, que será explicada). A *ILD* também é uma pista que o aparelho auditivo usa para localizar a origem da fonte de áudio. Segundo a própria literatura, esse é um efeito difícil de ser medido (HOWARD, 2017), contudo experimentos realizados indicam que há uma relação, na forma sinusoidal, entre a atenuação em dB e os ângulos de incidência para diferentes frequências. Na figura 5 está apresentada essa relação.

Figura 4 – Modelo de *ILD*.



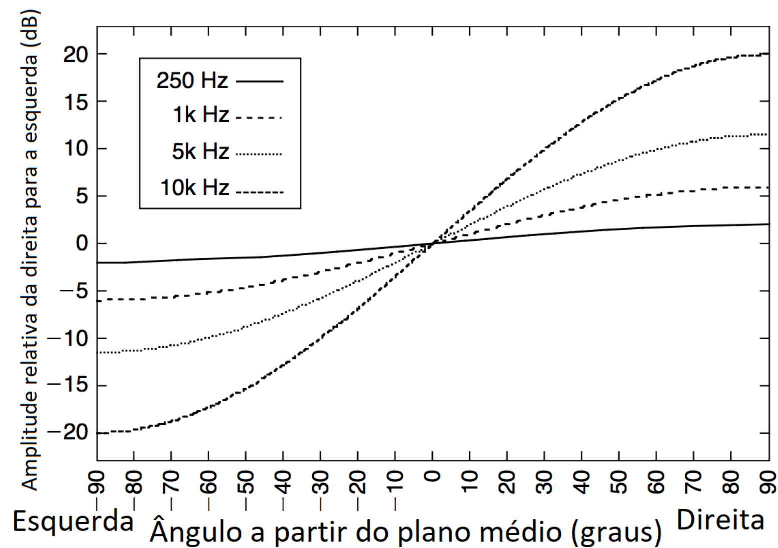

Adaptado de: (HOWARD, 2017).

Enquanto a *ITD* tem uma frequência máxima para cada ângulo na qual ela funciona adequadamente, a *ILD* tem uma frequência mínima. Isso ocorre porquê um corpo não é significativo para gerar *shading* a uma onda até que o seu tamanho seja de aproximadamente dois terços do comprimento de onda (HOWARD, 2017). Com a equação 21 que relaciona o tamanho da cabeça com a frequência mínima para esse efeito ser relevante.

$$f_{min}(\theta) = \frac{1}{3}\frac{c}{d} \tag{21}$$

na qual $f_{min}(\theta)$ é a frequência mínima para um dado valor do ângulo de incidência da onda sonora, $d$ é a distância entre os ouvidos, $\theta$ é o ângulo de incidência da onda sonora, em

Figura 5 – *ILD* em função de ângulo e frequência.




Adaptado de: (GULICK et al., 1971 apud HOWARD, 2017).

relação a uma reta normal à reta que liga os dois ouvidos (em radianos) e $c$ é a velocidade do som (em m/s).

Ainda segundo (HOWARD, 2017), a *ILD* é uma pista para frequências mais altas e a *ITD* para mais baixas e, há uma zona de cruzamento entre esses dois mecânismos que inicia em cerca de $700Hz$ e vai até aproximadamente $2,8KHz$ (que também depende do ângulo de incidência do áudio), e nessa zona a capacidade humana para distinguir a posição da fonte sonora não é boa como em outras frequências.

### 2.4.4 Fones de Ouvido e a Lateralização

Um fenômeno descrito na literatura ocorre quando uma pessoa usa fones de ouvido, e o som é percebido como se estivesse *dentro* da cabeça da pessoa. Esse fenômeno se opõe ao da localização sonora (ZWICKER; FASTL, 2013), sendo um fenômeno de apenas uma dimensão, que é a relação da presença do áudio ou mais no ouvido direito ou esquerdo do usuário do par de fones, enquanto o fenômeno da localização é um fenômeno tridimensional.

O posicionamento do som em um ou outro ouvido, no caso da lateralização, se dá através ou da diferença de amplitude ou da diferença de fase entre o áudio provindo das fontes do ouvido esquerdo e direito (ZWICKER; FASTL, 2013), lembrando os mesmos fenômenos apresentados para o som localizável.

Nesse trabalho, buscar-se-á evitar a lateralização e causar a percepção de localização binaural.

### 2.4.5 Lei da primeira onda

A lei da primeira onda (ZWICKER; FASTL, 2013), também encontrada na literatura como efeito *Haas* (HOWARD, 2017) (nome do pesquisador que identificou o efeito em um experimento), define que a fonte de áudio será dada pela primeira onda de som que chegar. Ondas que chegam em até $30ms$ são ignoradas para o posicionamento da fonte, e são percebidas como parte do primeiro som. Ondas que chegam após os $30ms$ são percebidas como eco.

### 2.4.6 Questão de percepção de Localização Frente-Atrás

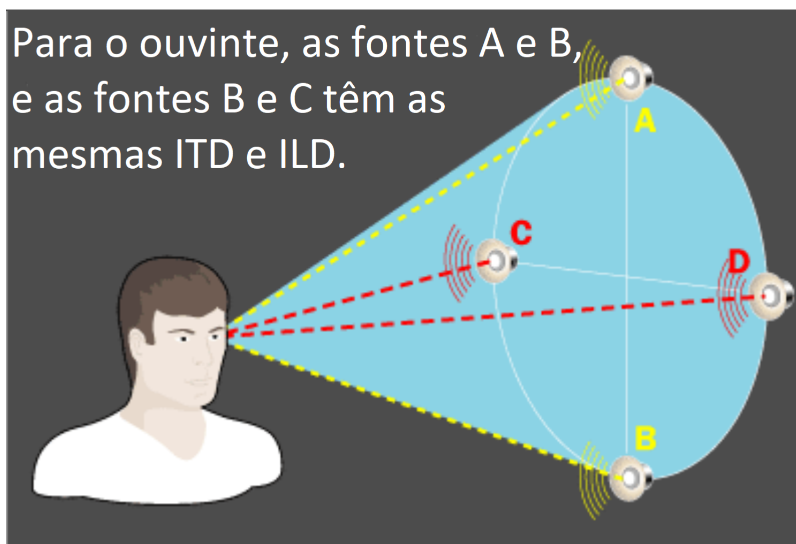
A questão da percepção de uma fonte sonora estando na frente ou atrás da pessoa é uma questão que ainda é pesquisada, e na literatura há diferentes aproximações e implicações interessantes. Os métodos explorados até agora nesse trabalho para a localização da fonte sonora, como o ITD e ILD, apresentam uma ambiguidade de posicionamento da fonte frente e trás, além da ambiguidade da elevação do som. Interessante notar que a questão da confusão frente-atrás é umas das principais dificuldades que precisa ser trabalhada para a geração correta de áudio externalizado em fones de ouvido.

Em (ZWICKER; FASTL, 2013) é apresentado o resultado de um estudo em que foi usado um tom de banda estreita. Esse estudo diz que, quando um som com a frequência central de 300 Hz ou 3 kHz é tocado, a pessoa que está escutando atribui a origem desse som como vinda da frente. Se for um áudio com frequência central de 8 kHz, ele é percebido como provindo de cima da pessoa. E, por fim, se for um áudio com frequência central de 1 ou 10 kHz, a origem é atribuída como provinda de trás da cabeça.

Em (HOWARD, 2017), é sugerido que há dois métodos principais para resolver essa questão de ambiguidade de posição. A primeira é relacionada com a o ouvido externo (ou em inglês, da *Pinnae*), e funciona geralmente para frequências acima de 5 kHz (há variação de pessoa para pessoa). Esse efeito ocorre através de reflexões que se originam no ouvido externo e causam um atraso no som, e esse atraso é função da posição de origem do som, de forma tridimensional. Em (FITZPATRICK et al., 2013), os autores trazem uma imagem elucidando essa afirmação, reproduzida na figura 6, na qual está escrito que, para o agente que escuta, as fontes de audio A e B e as fontes C e D tem ITD e ILD idênticas, porém, como é explicado nesse mesmo trabalho, as estruturas do ouvido externo são assimétricas, fazendo com que o áudio provindo de posições diferentes sejam filtrados de formas diferente, única para cada posição.

O segundo método presente em (HOWARD, 2017), sugere que a forma mais poderosa de podermos localizar uma fonte sonora se dá através de movimentos que realizamos com nossa cabeça. Quando queremos localizar a origem de uma fonte de áudio, nós movemos a nossa cabeça até que essa fonte fique em uma direção normal em relação à

Figura 6 – Mecanismo da Pinnae.



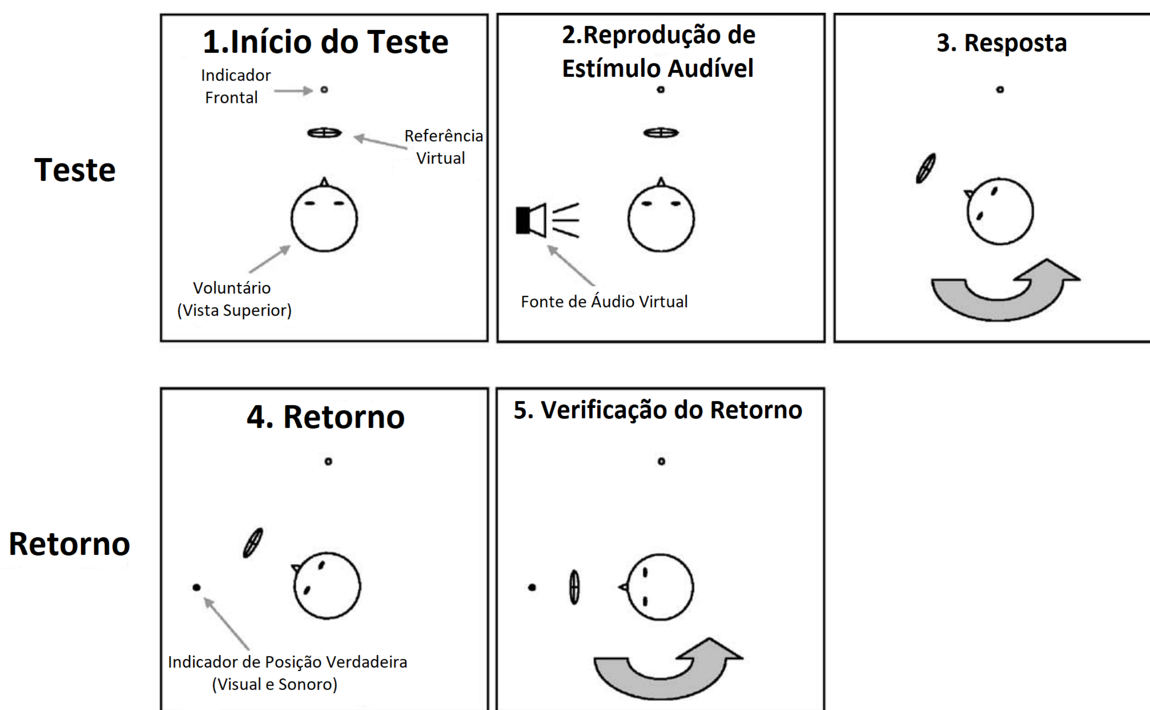

Adaptado de: (FITZPATRICK et al., 2013).

frente da mesma, posição na qual os atrasos e intensidades (*ITD* e *ILD*, respectivamente) são iguais. Isso, inclusive, é um dos fatores que causam a lateralização: ao movermos a nossa cabeça, a origem do som se move junto, fazendo com que a mesma seja projetada para "dentro"da cabeça.

Um efeito que acontece na prática, com o uso de fones de ouvido para a geração de som 3D é a reversão frente-atrás, descrita na literatura (ZAHORIK et al., 2006). Os autores desse artigo experimentaram um treinamento com voluntários, no qual um grupo foi treinado e outro não para comparar os efeitos do treinamento em relação à redução da reversão frente-trás. O treinamento consistia na reprodução de um som através de uma fonte virtual, utilizando um *HRTF* genérica (o conceito *HRTF* será explicado ainda neste capítulo) gerada pelo aparato de som. E então um ponto era exibido em um display na mesma posição que estava a fonte de áudio virtual, para o voluntário corrigir a posição (que podia ser em qualquer lugar dentro dos 360°). O método está descrito na figura 7. Para uma parte do grupo não foi apresentada a etapa do retorno (*feedback*) – onde o ponto da posição correta é mostrado. Para o grupo treinado, verificou-se que o treinamento persistiram ao menos por 4 meses e foram generalizados inclusive para posições não treinadas.

Na literatura (ALGAZI; DUDA, 2008), os autores criaram um sistema para geração de áudio 3D que considerava o seguimento do movimento da cabeça. Os autores, em sua revisão bibliográfica, resumiram três principais fatores sobre o uso de seguidor de cabeça para corrigir a questão do posicionamento frente-atrás, em relação aos erros no plano do azimute:

Figura 7 – Processo de treinamento para corrigir a reversão frente-atrás.




Adaptado de: (ZAHORIK et al., 2006).

- 1. Para estímulos auditivos menores que 0.3 segundos em duração, o movimento da cabeça realmente não melhora a qualidade da localização sonora.
- 2. Para estímulos auditivos de ao menos um segundo de duração, os erros causado pela confusão frente-trás são completamente eliminados.
- 3. Se a confusão frente/atrás é corrigida, a precisão da localização no plano horizontal é essencialmente não melhorada através do uso dos movimentos da cabeça.

Adicionalmente, os autores (ALGAZI; DUDA, 2008) também afirmaram que com o uso de seguidores de movimento da cabeça (em inglês, *Head Tracker*), as frequências baixas se tornaram um dos fatores mais importantes para a localização de fontes sonoras, e que o uso de *HRTF* (explicada na seção 2.5) – principalmente as personalizadas –, são menos críticas que no caso do sistema sem o detector de movimento. Tanto a *ITD* quanto a *ILD* se tornaram fatores importantes no trabalho descrito, na faixa de frequências menores que aproximadamente $3kHz$.

## 2.5 *HRTF* e *HRIR*

As Funções de Transferência Relacionada à Cabeça (*HRTF*), do inglês *Head Related Transfer Function* – ou sua versão no domínio tempo Resposta ao Impulso Relacionada à Cabeça, do inglês *Head Related Impulse Response* (*HRIR*) – são descritas como a forma completa e formal de descrever as distorções causadas no som pela cabeça e pelo torso de uma pessoa (VORLÄNDER, 2007). Cada pessoa tem uma função de transferência para cada posição de fonte de áudio possível no espaço, e também, a *HRTF*, por ser uma característica individual, terá diferença de pessoa para pessoa, de acordo com o formato de suas orelhas e o tamanho da sua cabeça dentre outras características particulares.

Como descrito em (VORLÄNDER, 2007), para a caracterização das *HRTF*, considera-se que está chegando no sujeito uma onda sonora plana. Também, deve-se notar que as pistas binaurais estudadas (como a *ITD* e a *ILD*) e outras de origem monaural, serão introduzidas no tímpano e carregam a difração do som incidente na cabeça e no torso da pessoa. A quantidade de difração é então descrita pela *HRTF*.

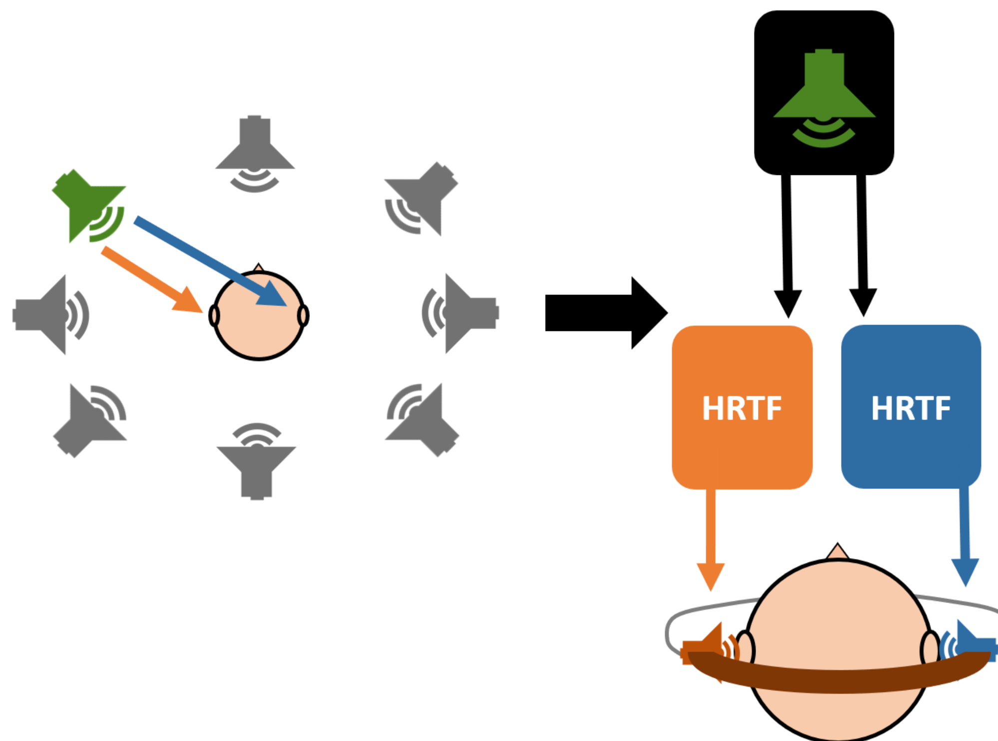
A unidade da *HRTF* é a medida de pressão sonora no tímpano da pessoa ou no canal de entrada, dividida pela pressão sonora medida na posição central da cabeça sem a cabeça presente.

Como mencionado, a *HRTF* dependerá da direção do som incidente. Na figura 8 está representada, à esquerda do leitor, uma situação na qual uma pessoa, no centro de um arranjo de fontes sonoroas (similar ao da figura 1) e apenas uma dada fonte sonora está reproduzindo áudio (em cor diferente das demais, em verde). Nota-se que para cada ouvido, o áudio segue um caminho diferente, com diferentes obstáculos, representados pelas setas coloridas, ou seja, um áudio binaural.

Quando as *HRTF*s são capturadas, utiliza-se câmaras anecoicas (ou seja, salas sem eco). Então, com microfones colocados nos dois ouvidos de uma pessoa, uma fonte sonora em uma posição estabelecida é acionada. E assim, captura-se as características que um áudio gerado em uma dada posição em relação a pessoa tem, em cada um dos ouvidos da mesma – ou seja, considerando um sistema linear descrito na seção 2.1.2, tem-se capturada a resposta ao impulso para essa dada posição. Se essas características forem aplicadas a outro áudio (através de filtragem), esse outro áudio receberá as características direcionais da *HRTF*.

Ainda de (VORLÄNDER, 2007), é estabelecido um sistema de coordenadas para as fontes sonoras, que considera os planos da figura 9, com o plano central descrito pelo ângulo de azimute, $\phi$, contado de forma anti-horária a partir da posição frontal (0°até 360°); e o ângulo polar $\theta$ de elevação, contado a partir da posição frontal para cima (de 0°até 90°); o ângulo polar se extende de forma negativa (para direção para baixo). Um

Figura 8 – Modelo do ambiente traduzido para o fone de ouvido.



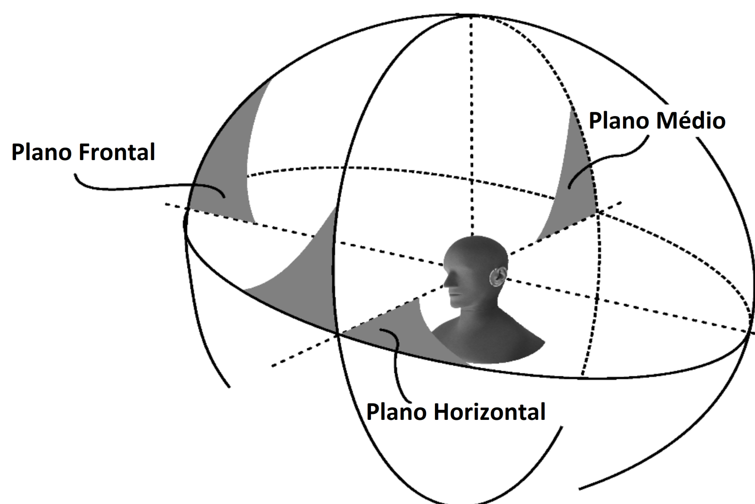

exemplo de um *HRTF* e um *HRIR* para $(\phi, \theta) = (90°, 0°)$ está na figura 10. Os sistemas de coordenadas são melhor explorados na seção 2.8.

Novamente na figura 8, no lado direito do leitor, a mesma situação do lado esquerdo é representada quando as *HRTF* são usadas: o áudio original que seria reproduzido pela fonte sonora posicionada, passa por dois filtros diferentes, um para cada ouvido, que contém as características de cada um dos caminhos diferentes que a onda sonora percorre. Então os filtros são aplicados no áudio original, ou seja, uma função de transferência (*HRTF*) é aplicada a um sinal, fazendo com que o sinal adquira as suas características. Dessa forma, a sensação de áudio espacializado é então gerada ao aplicar o filtro no áudio original.

## 2.6 *BRIR*

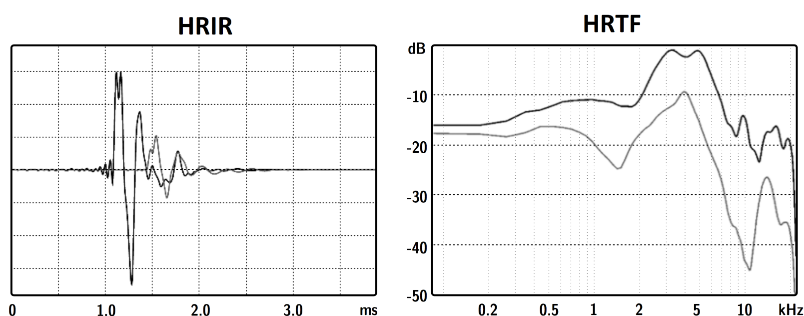
A Resposta ao Impulso Binaural de Sala, do inglês *Binaural Room Impulse Response* (*BRIR*), é uma forma de descrever a resposta de uma sala. De forma análoga ao *HRIR*, compreenderá as características que uma fonte sonora gera em uma dada sala em uma dada posição. Arquivos *BRIR* são compostos de sons diretos e sons refletidos, conforme (VORLÄNDER, 2007).

Figura 9 – Sistemas de coordenada relacionado à cabeça.




Adaptado de: (VORLÄNDER, 2007).

Figura 10 – Exemplo de HRTF e HRIR relacionados.




Adaptado de: (VORLÄNDER, 2007).

É essencial que seja uma reposta ao impulso binaural, para que o fator localização não seja perdido, novamente conforme (VORLÄNDER, 2007). Portanto, os arquivos com dados BRIR terão, igual aos arquivos com *HRTFs*, uma função de transferência para cada posição espacial possível, usando as mesmas coordenadas referidas na sub-seção 2.5: $(\phi, \theta)$, ângulo azimute e ângulo de elevação.

## 2.7 Arquivos *SOFA*

Conforme apresentado na literatura (MAJDAK et al., 2013), o formato SOFA visa unificar os diversos formatos de arquivo que as *HRTF* são disponibilizadas. A especificação

do arquivo também tem por objetivo a facilitação da criação de uma API para MATLAB e C++. É padronizado pela Audio Engineering Society (AES).

Um arquivo *SOFA* contém diversas funções de transferências, ou coeficientes de filtro *FIR* (análogo no domínio tempo da função de transferência que é dada no domínio frequência), em um formato de dados definido pelo documento (MAJDAK et al., 2013). Isso ocorre, por exemplo, em uma situação como a da figura 1, que pode-se desejar ter as funções de transferência respectivamente de todas fontes sonoras do ambiente para a pessoa.

O arquivo contém também dados adicionais, explicado os equipamentos e ambiente usados na captura dos dados de áudio. As coordenadas de captura das funções de transferência (ou os dados análogos no domínio tempo), são descritas com coordenadas retangulares ou esféricas, porém podem estar em uma geometria livre: não necessitam ser capturadas em uma geometria fixa e igualmente espaçada entre as capturas.

### 2.7.1 *CIPIC*

Segundo descrito na literatura (ALGAZI et al., 2001), o banco de dados CIPIC é uma biblioteca de domínio público de *HRTFs*. A versão 1.0 inclui a cabeça de 45 voluntários em 25 azimutes diferentes e 50 elevações diferentes com um ângulo aproximado de 5°de incremento. Está disponibilizada no formato *SOFA* na internet.

Esse banco de dados foi medido no *U.C. Davis CIPIC Interface Laboratory*, um laboratório americano. Esse banco de dados serve para que ao se gerar áudio espacializado (ou *áudio 3D*), o usuário desse sistema possa selecionar um arquivo capturado de uma pessoa ou manequim com uma cabeça com características similar a sua – dentre as 45 disponíveis nessa biblioteca, todas com características diferentes –, fazendo com que a geração de áudio 3D funcione melhor para o mesmo.

## 2.8 Posições e coordenadas

Nessa seção serão definidos os sistemas de coordenadas retangular e esférico, como são realizadas as transformações de um sistema para outro, e como são realizadas as rotações - através de matrizes - em torno dos eixo cartesianos.

### 2.8.1 Coordenadas retangulares

Segundo (MOON; SPENCER, 2012), o sistema de coordenadas retangulares possui como superfícies três planos mutuamente ortogonais, de valores constante $x = const$,

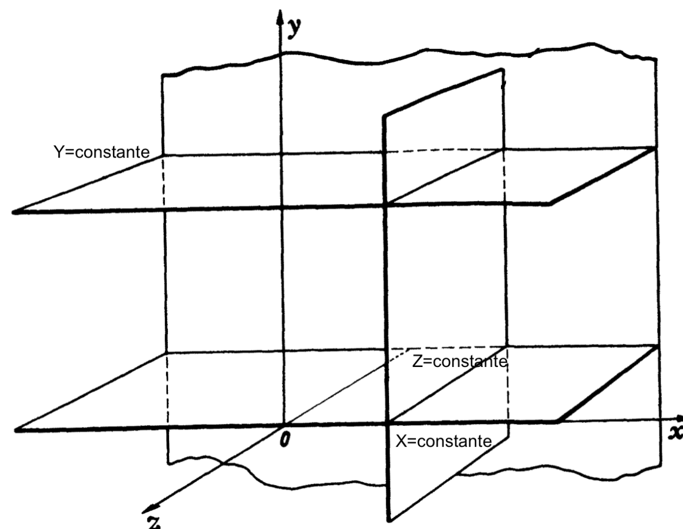
$y = const$, $z = const$, e pode ser visualizado na figura 11. Os domínios das variáveis de posição $x$, $y$, $z$ estão respectivamente definidos nas equações 22, 23, 24.

$$-\infty < x < \infty \tag{22}$$

$$-\infty < y < \infty \tag{23}$$

$$-\infty < z < \infty \tag{24}$$

Figura 11 – Coordenadas retangulares (x,y,z).




Adaptado de: (MOON; SPENCER, 2012).

### 2.8.2 Coordenadas esféricas

Segundo (MOON; SPENCER, 2012), o sistema de coordenadas esféricas possui como superfície uma esfera (de raio $r$ constante), cones circulares (de $\theta$ constante), e semi-planos (de $\phi$ constante), de valores retangulares definidos nas equações 28, 29, 30, e pode ser visualizado na figura 12. Os domínios das variáveis de posição $r$, $\theta$, $\phi$ estão respectivamente definidos nas equações 25, 26, 27.

$$0 < r < \infty \tag{25}$$

$$0 < \theta < \pi \tag{26}$$

$$0 < \phi < 2\pi \tag{27}$$

$$x + y + z = r^2 \tag{28}$$

$$tan(\theta) = \frac{\sqrt{(x^2 + y^2)}}{z} \tag{29}$$

$$tan(\phi) = \frac{y}{z} \tag{30}$$

Interessante notar que, conforme já descrito na seção 2.5, para questões das funções de transferência relacionadas à cabeça (*HRTF*), segundo (VORLÄNDER, 2007), o sistema de coordenadas esférico é estabelecido como um raio de $1m$ (salvo exceções), com ângulos $(\Phi, \Theta)$ (aqui em maiúsculo para evitar confusão) estabelecidos segundo às equações 31 e 32, dados em graus e não radianos. A posição $(\Phi, \Theta) = (0, 0)$ é definida a posição imediatamente a frente da pessoa, e o ângulo $\Phi$ é contado de forma anti-horária a partir dessa posição. Já o ângulo de elvação é positivo no plano superior e negativo no plano inferior. A relação com o $\theta$ está descrita na equação 33 (com ambos valores representados em graus). Mais detalhes estão na mencionada seção (2.5).

$$0° < \Phi < 360° \tag{31}$$

$$-90° < \Theta < 90° \tag{32}$$

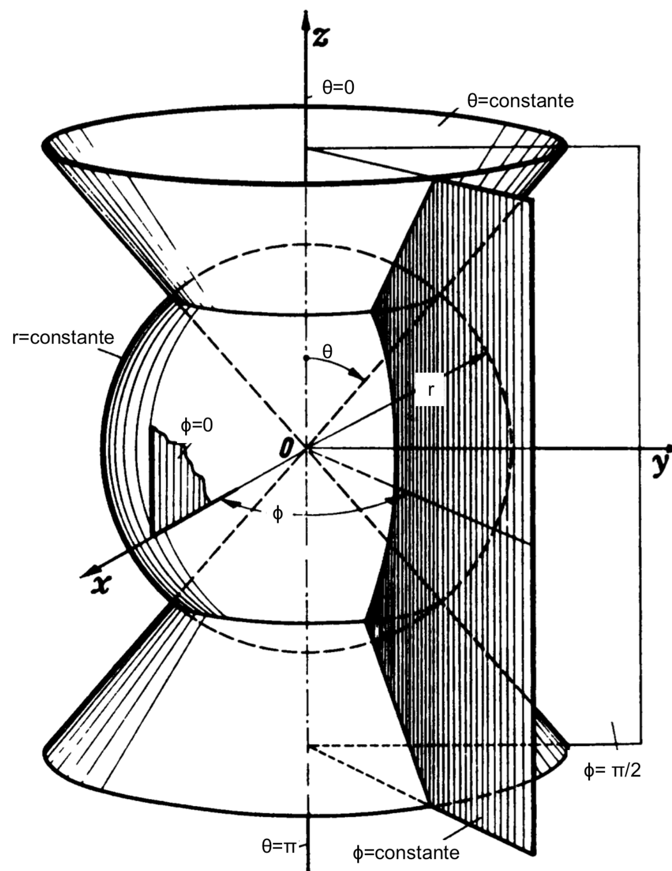
$$\theta = \begin{cases} 90° + |\Theta| & \text{if } \Theta \geq 0 \\ 90° - |\Theta| & \text{if } \Theta < 0 \end{cases} \tag{33}$$

### 2.8.3 Relação entre os sistemas de coordenadas

Nas equações 34, 35 36 (MOON; SPENCER, 2012)estão expressas as três relações necessárias entre os sistemas de coordenadas retangular e esférico, habilitando a transformação de um sistema para outro.

$$x = r \times sin(\theta) \times cos(\phi) \tag{34}$$

Figura 12 – Coordenadas esféricas ($r$,$\theta$,$\phi$).




Adaptado de: (MOON; SPENCER, 2012).

$$y = r \times sin(\theta) \times sin(\phi) \tag{35}$$

$$z = r \times cos(\theta) \tag{36}$$

### 2.8.4 Rotacionando vetores em torno dos eixos (x,y,z)

Adaptando de (MOON; SPENCER, 2012), assumindo uma matriz de transformação bi-dimensional $R$, em um sistema de coordenadas retangulares, ao realizar uma rotação de $\alpha$ graus em torno do ponto comum aos dois eixos $(0, 0)$ no caso bi-dimensional, tem-se a matriz de rotação expressa na equação 37, que pode ser estendida para o caso tri-

dimensional, considerando que o eixo que está sofrendo a rotação não sofre alteração, expresso na equação 38 para o eixo x, 39 para o eixo y e 40 para o eixo z.

$$R(\alpha) = \begin{bmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{bmatrix} \quad (37)$$

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{bmatrix} \quad (38)$$

$$R_y(\alpha) = \begin{bmatrix} cos(\alpha) & 0 & -sin(\alpha) \\ 0 & 1 & 0 \\ sin(\alpha) & 0 & cos(\alpha) \end{bmatrix} \quad (39)$$

$$R_z(\alpha) = \begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 \\ -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (40)$$

## 2.9 Trabalhos Anteriores

Na literatura existe uma boa quantidade de trabalhos sobre a geração do efeito de áudio 3D em fones de ouvido com mais ou menos qualidade e recursos. Para essa seção, três trabalhos foram selecionados e serão aqui apresentados.

### 2.9.1 Geração de áudio 3D através do seguimento do movimento da cabeça

No primeiro artigo destacado (ALGAZI; DUDA, 2008), já citado na seção 2.4.6, os autores utilizaram uma técnica para geração de áudio 3D através do seguimento do movimento da cabeça (MTB, motion-tracked binaural sound). Utilizando a consideração que o uso de *head-tracker* minimiza a dependência de frequências acima de $3kHz$, a ITD e a ILD foram as principais pistas sonoras utilizadas.

Enquanto o sistema utilizado, segundo os autores, tornou menos complexos as necessidades computacionais, uma vez que apenas as frequências baixas são processadas (as altas não são afetadas no processamento), por outro lado esse sistema requer um aparato de gravação mais complexo: com o uso de um arranjo de microfones, ao redor de um cilindro, diversas capturas são feitas simultaneamente, e ao usuário escutar a gravação, o sistema seleciona as fontes de áudio mais adequadas e interpola os mesmos. Portanto, ocorre de uma complexidade na gravação, e esse sistema se torna limitado para o uso em gravações normais de áudio, como músicas disponíveis em CD.

### 2.9.2 *HRTF* e *head-tracker*

Em um outro artigo (FITZPATRICK et al., 2013), os autores propõem (e de fato projetam) um sistema para a geração de áudio 3D com a utilização de um processador DSP e com um *head-tracker* baseado em vídeo. Para a etapa de processamento no DSP, foi optado pela utilização de um filtro do tipo *FIR*, diferente do presente trabalho onde se proporá o processamento no domínio frequência com o uso da *FFT*. Ainda em relação ao artigo, os autores afirmaram em sua conclusão que para um melhor resultado houve a necessidade de seleção de uma *HRTF* mais adequada pelos usuários, individualmente, através do teste de várias *HRTFs* diferentes. Como resultado, em uma sala com uma relativa liberdade, os usuários conseguiam localizar a fonte sonora virtual ao utilizar o aparato.

### 2.9.3 *HRTF* e outras combinações de elementos para a geração de áudio 3D

Em mais um artigo encontrado na literatura (HADAD et al., 2014), um sistema para a realização de experimentos com áudio 3D é projetado e utilizado. Dentre as principais características do sistema em questão, pode-se afirmar o uso de uma variação da *FFT* (especificamente Transformada de Fourier de Tempo Curto), um gerador de movimentos da cabeça (não foi utilizado um *head-tracker* real, e portanto foi feita essa substituição), uso de *HRTF* e de *BRIR*.

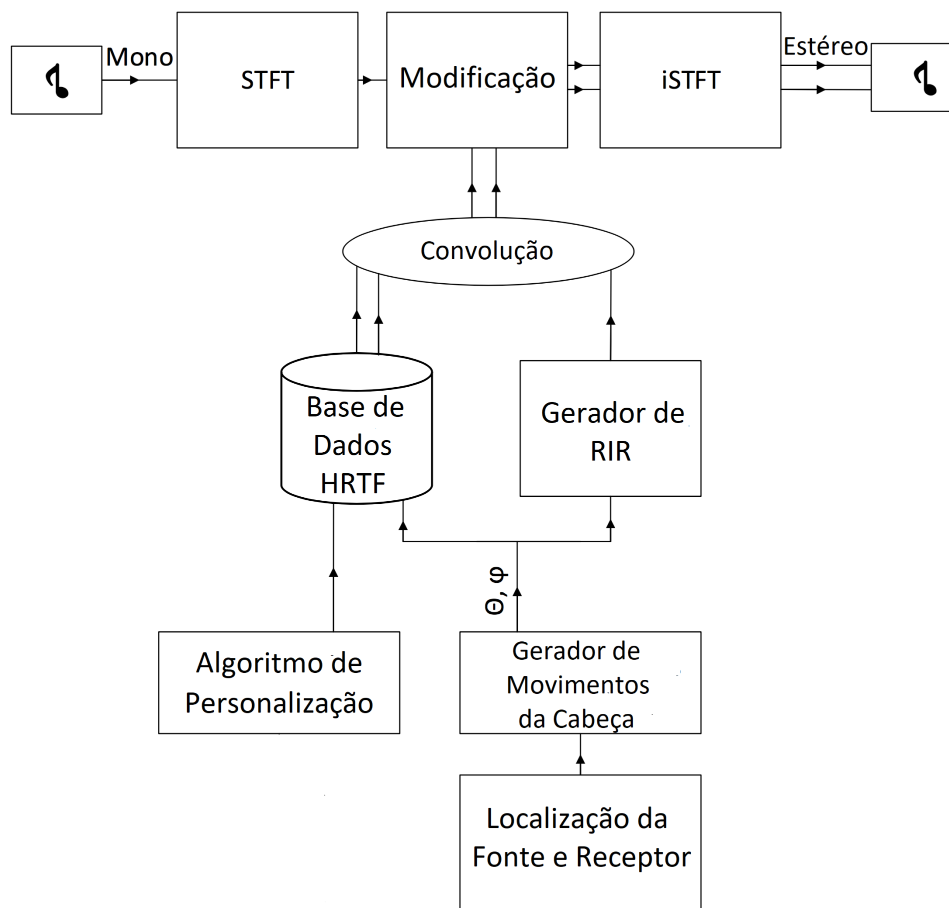
Dentre os principais fatores que os autores queriam avaliar, destaca-se: o efeito do uso das reflexões da sala (através do uso de BRIR), a personalização da *HRTF* de acordo com as medidas da antropomórficas do usuário, e o uso do seguidor de movimento da cabeça. O sistema está resumido na figura 13. Os principais resultados de um estudo com 20 voluntários, foram:

- 1. Adicionar a externalização o uso de *BRIR* e movimentos da cabeça (além da *HRTF*) reduz o erro de localização absoluto por 9,2%.
- 2. Com a adição da personalização das medidas antropomórficas, o erro absoluto de localização reduziu em 17.9%.

## 2.10 Tópicos de programação

Nessa seção serão abordados alguns tópicos específicos relacionados com a própria programação. Primeiro é abordado sobre o uso de *threads* e *mutexes*, e após sobre o perfil serial do *bluetooth*.

Figura 13 – Blocos de um sistema para a geração de áudio 3D.




Adaptado de: (HADAD et al., 2014).

### 2.10.1 *Threads*

De (TANENBAUM, 2009), um *processo* em um computador é uma forma de agrupar recursos relacionados juntos. Um processo contém um espaço de endereços onde estarão os dados do programa a ser executado, além de outros recursos como processos filhos, ou arquivos abertos. A *thread* (de *thread of execution*) adiciona a esse modelo, sendo ela a entidade que faz a execução dos programas. Adicionalmente, dentro de um processo, pode haver mais que um ponto de execução em paralelo, e portando mais uma *thread*.

Ainda de (TANENBAUM, 2009), existem diversas vantagens por se usar *threads*. A primeira é que em algumas aplicações múltiplas atividades precisam ser executadas ao mesmo tempo e, ao decompor a execução nessas atividades paralelas (ou *quasi-paralelas* como afirma o autor), o modelo de programação se torna mais simples. Outro argumento, acrescenta o autor, é que *threads* são mais fáceis e portanto leves de serem usadas em um sistema operacional se comparado a processos. Seguindo, o autor argumenta que o desempenho é outra razão: quando uma atividade usa uma quantidade substancial de

processamento e de entrada e saída de dados, *threads* permitindo que essas atividades se sobreponham, portanto elevando a velocidade de execução da aplicação. E, diz o autor, outra razão, é no uso de programas em processadores com múltiplos processadores, nos quais o paralelismo real é possível.

### 2.10.2 *Mutexes*

*Mutex*, segundo (TANENBAUM, 2009), é um caso simplificado de um semáforo. No semáforo geral, um dado processo (ou *thread* de um processo) é trancado quando o semáforo é zero, significando que ele não tem dados disponíveis para processar. Já o *mutex*, a variável assume apenas dois valores: 1 ou 0, marcando se um recurso compartilhado está sendo usado ou está livre. Então, utiliza-se *mutexes* para a sincronia de dados de múltiplas *threads*: quando a variável de *mutex* está liberada, o processo pode então marcar que está usando a mesma (colocado o valor para 1) e então acessar os dados compartilhados, garantindo que o mesmo é o único a realizar operações nestes dados, evitando conflito com outros processos.

### 2.10.3 Conectando com *Bluetooth*

O perfil a ser abordado nesse tópico é o chamado *Bluetooth Serial Port Profile* (BLUETOOTH, 2001), que segundo a própria especificação, serve para simular a conexão por porta serial RS232 (ou similar). Desta forma, o *bluetooth* serve como substituição para o cabo serial, através da abstração virtual de uma porta (essa, por sua vez, dependente do sistema operacional).

Ainda segundo a documentação, (BLUETOOTH, 2001), o *bluetooth* fornece as camadas 1 e 2 do modelo OSI (física e de enlace), e a camada de transporte. A cima dessa camada está uma API que simula o comportamento de uma porta serial na camada de aplicação. Conforme a especificação destaca, a camada rodando a cima dessa API necessita conectar-se por porta serial e não são capazes por si só de inicializar o que é necessário para a comunicação *bluetooth* ocorrer. Precisando, então, de um *software* que auxilie. Como será explicado, no caso desse programa a própria interface serial do sistema operacional resolvia a conexão.

No caso do presente trabalho, o sistema macOS fornece os recursos necessários para a conexão. O procedimento pode ser resumido em:

- Parear o dispositivo *bluetooth* com o computador, usando as interfaces do próprio sistema.
- Verificar os dispostivos disponíveis em */dev* com o comando *ls -l /dev/tty.** que listará os *terminais* seriais disponíveis no sistema.

- Conectar ao dispositivo para envio e recepção de texto através do comando *screen /dev/tty.**dispositivo** velocidade*, onde velocidade é um valor em *bits por segundo*.

Alternativamente, para o uso desse perfil em um *software*, a última etapa de conexão deve ser realizada pelo próprio programa.

# 3 Implementação

Neste capítulo primeiro é apresentada desde a proposta de solução até o objeto final produzido neste trabalho.

## 3.1 Proposta de solução

Considerando a literatura apresentada na revisão bibliográfica, há duas formas gerais que o efeito de áudio 3D pode ser gerado: uma é a gravação adequada do áudio através de microfones nos ouvidos, adequadamente posicionados, e a outra é através da aplicação das funções de transferência relacionadas à cabeça (e a sala). Nesse trabalho, foi utilizado o método da aplicação das funções de transferência, de forma a completar o objetivo descrito na seção 1.2.

A principal vantagem da utilização desse método é a generalização: com o aparato funcional, e a *HRTF* relacionada ao usuário do sistema, no momento que um som genérico for reproduzido no computador ele pode ter o efeito aplicado. Com o outro método, para cada áudio a ser reproduzido exigir-se-ia uma gravação específica, tornando esse sistema mais interessante para concertos por exemplo, mas não para o uso em computadores.

Elaborou-se um algoritmo de computador que recebe como entrada uma *stream* de áudio (ou seja, uma reprodução contínua no computador), aplica em tempo real o efeito binaural, e por fim reproduz o áudio com o efeito aplicado a fim de que seja reproduzido em fones de ouviudo. O *programa de computador* está genericamente representado na figura 14 – um áudio é gerado por um *software* genérico, nesse caso o navegador de internet *Google Chrome*, e para fins de exemplo, imagina-se esse som como sendo *mono*, ou seja apenas um canal, e este é capturado pelo programa, processado, e após reproduzido por um par de fones de ouvidos. Nota-se que para cada auto-falante do fone de ouvido um áudio diferente é reproduzido, ou seja, o áudio inicialmente de um canal foi transformado em um áudio de dois canais diferentes, cada um com seu próprio filtro aplicado – como foi explicado na ilustração da seção 2.5, e melhor detalhado no próximo parágrafo. O *head-tracker*, que é conectado através de *Bluetooth*, serve para que a posição da cabeça do usuário seja corrigida. O bloco *Processamento do Áudio 3D* busca aplicar a filtragem descrita pelas gravuras 8 e 15.

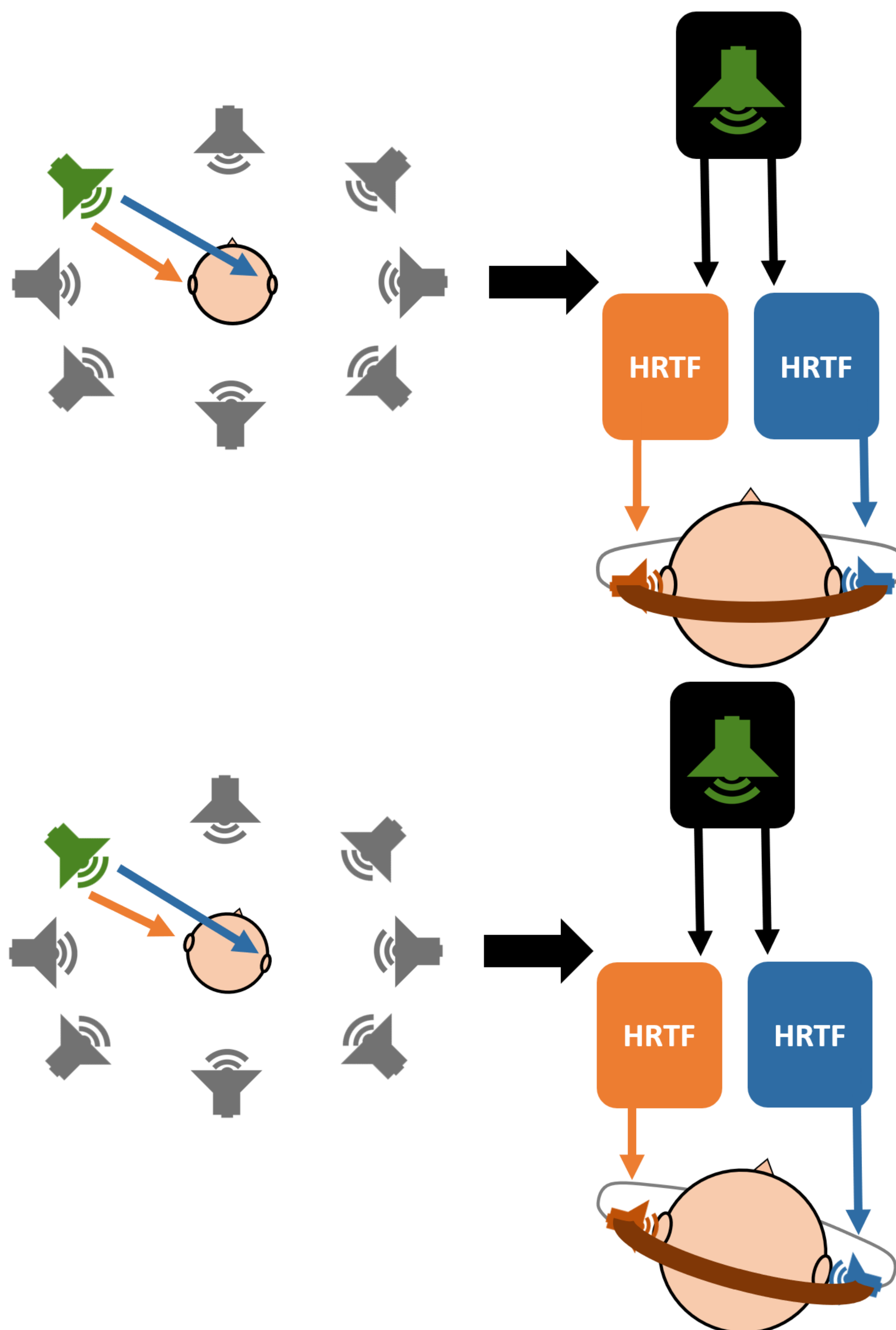
Para entender melhor o procedimento de filtragem, de forma similar ao explicado na seção 2.5, na figura 15 está representada, a esquerda do leitor na parte superior, uma situação na qual uma pessoa, no centro de um arranjo de fontes sonoroas (similar ao da figura 8) e apenas uma dada fonte sonora está reproduzindo áudio (em cor diferente das

Figura 14 – Diagrama de blocos da proposta de solução.

demais, em verde). Nota-se que para cada ouvido, o áudio segue um caminho diferente, com diferentes obstáculos, representados pelas setas coloridas. No lado direito do leitor, a mesma situação do lado esquerdo é representada quando as *HRTF* são usadas: o áudio original, sem aplicação de filtros, passa por dois filtros diferentes, um para cada ouvido do usuário, que aplica as características do caminho que o áudio percorriria ao áudio, e a sensação de áudio espacializado é então gerada.

Caso a pessoa se movimente, como representado na parte inferior da figura 15, os *caminhos* percorridos pela onda sonora mudará. Então, de forma que o a sensação de áudio 3D seja corretamente aplicada, um novo filtro deverá ser aplicado (representado por setas de tamanhos e posições diferentes nas partes superior e inferior da figura), para essa nova posição relativa. Para isso é utilizado o *head-tracker*, para que a correção seja corretamente realizada.

Figura 15 – Modelo do ambiente traduzido para o fone de ouvido, antes e após o movimento do usuário.



## 3.2 Ferramentas utilizadas

Como mencionado ao longo do corpo desse trabalho, o sistema de detecção da posição da cabeça do usuário (em inglês *head-tracker*) não fez parte deste trabalho. Há duas alternativas: a simulação da posição de entrada, ou ainda a utilização de um detector de posição pronto. No inicio do desenvolvimento utilizou-se de posição estática, seguindo então da inserção de um *head-tracker* existente no laboratório LaPSI (Laboratório de Processamento e Sinais da UFRGS). Esse *head-tracker* tem como saída graus de rotação nos eixos $x$, $y$ e $z$ do mesmo.

Quanto ao *hardware* que será utilizado, o computador é um *Apple Macbook Pro 11,5*, que será usado em conjunto com um fone de ouvido profissional *Shure SRH-440* – alternativamente também á usado o fone de ouvido *AKG K240*.

Também foram usados, no que é relacionado ao *software*, bibliotecas para manipulação de arquivos *SOFA* (explicados no capítulo 2.7), os próprios arquivos *SOFA* (escolhida a biblioteca do *CIPIC*, mencionada no capítulo 2.7.1), e o *MathWorks MATLAB©* para testes iniciais em conjunto e algoritmos elaborados em outros trabalhos do laboratório *LAPSI* (a serem explicados na seção 3.3).

Quanto ao programa final, como será apresentado nos próximos capítulos em detalhes, o mesmo foi desenvolvido na linguagem de programação *C++* versão 17.

## 3.3 Testes em MATLAB para a geração do efeito binaural

De forma a verificar as ferramentas disponíveis para auxiliar na implementação do algoritmo, foram inicialmente realizados testes com o uso do *software MathWorks MATLAB©*, com o uso de uma biblioteca para manipulção de arquivos *SOFA* em conjunto com um conjunto de algoritmos auxiliares (elaboradas por outro autor, com as funcionalidades explicadas em um apêndice).

Esses algoritmos e a própria biblioteca *SOFA* fornecem funções que servem carregar os arquivos *SOFA*, e obter os coefiecientes.

Com o uso destes, foram criados, então, dois algoritmos (com o código disponiblizado em anexo):

- O objetivo do primeiro algoritmo foi gerar o efeito de externalização para duas fontes de áudios virtuais em posições fixas. O nome é Teste1.m, e essa função retorna uma stream de áudio, que pode ser executado por *play*(*stream*); Pede como entrada o número do arquivo *SOFA* relacionado a uma cabeça (provindo da biblioteca do *CIPIC*), o Azimute, o volume da fonte de áudio 1, o volume da fonte de áudio 2 e o nome do arquivo *wave* a ser processado. As fontes de

áudio estão com 180 graus de diferença no Azimute. Exemplo de uso: $stream1 = teste1(11, 90, 0.4, 0.4,'sound_0.wav'); play(stream1)$.

- O objetivo do segundo algoritmo elaborado foi simular uma fonte de som virtual em movimento, através da variação do ângulo do Azimute ao longo da execução da stream de áudio. O nome do arquivo é Teste2.m, e essa função retorna uma stream de áudio, que pode ser executado por $play(stream)$; Pede como entrada o número do arquivo *SOFA* relacionado com a *HRTF* (provindo da biblioteca do *CIPIC*), o Azimute, o quanto o Azimute deve variar ao longo da execução do arquivo de áudio, a elevação, e o nome do arquivo *wave* a ser processado. Os dois canais de áudio do arquivos são somados em um único canal e uma fonte somente virtual é usada. Exemplo de uso: $stream2 = teste2(11, 0, 3600, 20,'sound_0.wav'); play(stream2)$.

Foram, com esses algoritmos, observados os seguintes resultados: percebe-se que foi gerado um efeito 3D, mas que pode ser muito melhorado. Pontualmente para o segundo algoritmo, quando há a movimentação das fontes sonoras, verifica-se uma *clipagem* no som. Há a necessidade de aplicação de um filtro mais flexível que os disponíveis no MATLAB para a troca dos coeficientes de filtragem em tempo de execução (que são relativos à posição da fonte de áudio) sem o aparecimento desse defeito.

Na figura 16 está um pequeno trecho de áudio, gravado através de microfone, de voz humana. Na figura 17 está o resultado desse mesmo áudio processado pelo primeiro algoritmo do autor, executado com os seguintes parâmetros: $stream1 = script1$ $(11, 90, 0.4, 0.4,'sound_1.wav')$. Na figura 18 o áudio agora é processado pelo algoritmo 2 com os parâmetros $stream1 = script2(11, 0, 180, 20,'sound_1.wav')$;.

Figura 16 – Áudio: trecho de voz humana gravada com microfone.

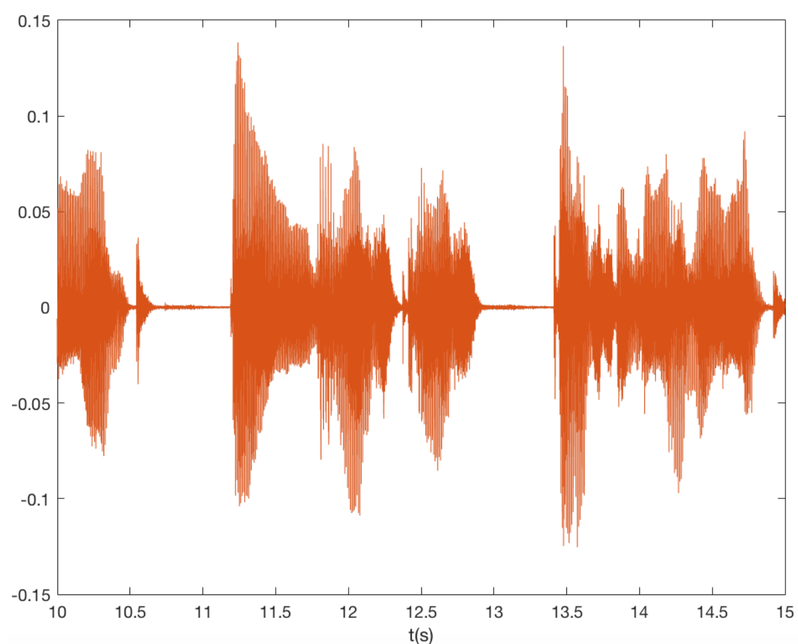
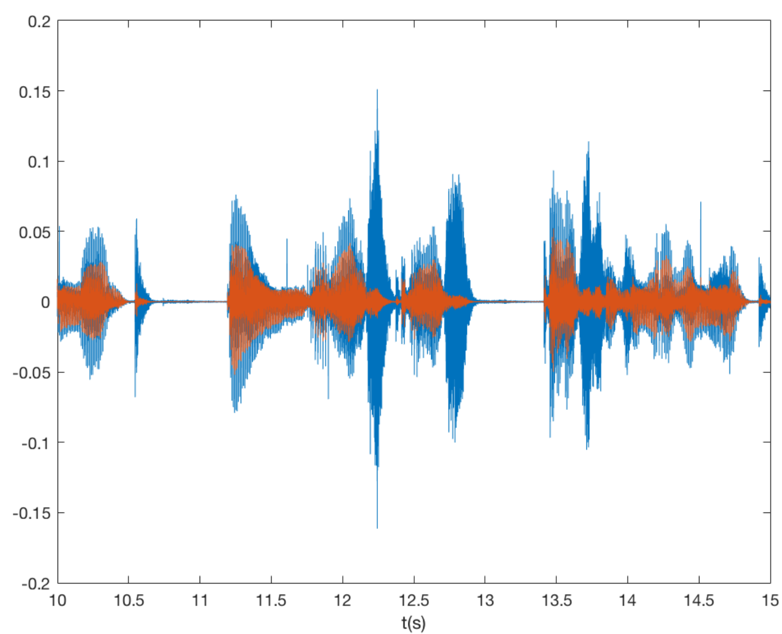
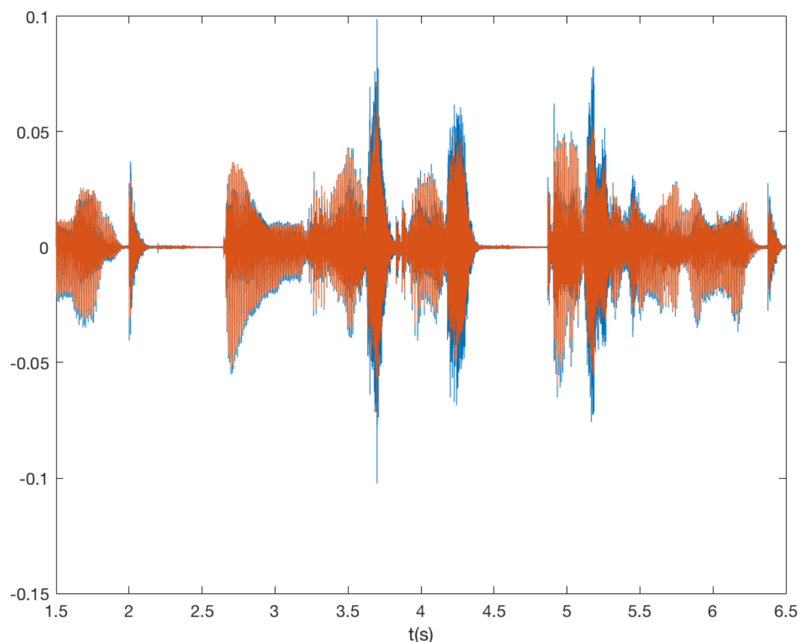



Figura 18 – Trecho de voz humana com o efeito 3D aplicado pelo segundo algoritmo em MATLAB.




Os equipamentos usados foram o mesmo listados na seção 3.2: um computador *Apple Macbook Pro 11,5* e um fone de ouvidos profissional *Shure SRH-440*.

Figura 17 – Trecho de voz humana com o efeito 3D aplicado pelo primeiro algoritmo em MATLAB.




Além do autor deste trabalho, três diferentes pessoas escutaram livremente o áudio representado na figura 18, gerado pelo segundo algoritmo, além de um áudio alternativo que era uma música que também foi processada pelo mesmo algoritmo. Pequenas conclusões puderam ser feitas dos comentários, que ajudaram as investigações posteriores:

- É mais fácil de perceber o efeito 3D quando apenas a voz humana é tocada.
- Em algumas posições de azimute o efeito 3D fica mais evidente, enquanto em outras desaparece (e ocorre a lateralização).
- Enquanto algumas das pessoas ouviram o efeito em muitas posições, uma elas reportou perceber o efeito somente em uma posição "ao lado da cabeça".
- Enquanto o algoritmo originalmente foi projetado para movimentar a fonte virtual no sentido anti-horário de azimute, uma das pessoas reportou perceber o movimento ocorrendo no sentido horário: houve uma clara reversão frente-atrás.

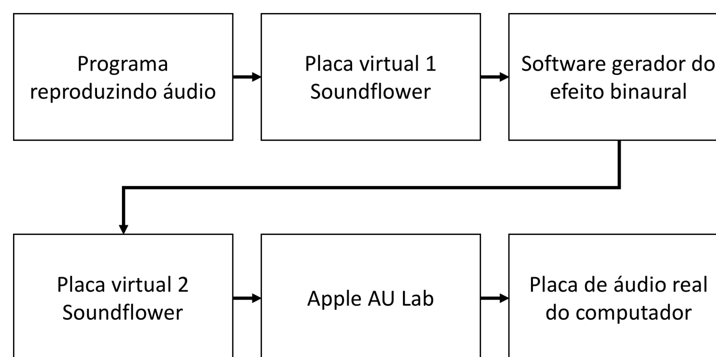
## 3.4 Programa em C++ no macOS

Desenvolveu-se um programa com o uso da linguagem de programação *C++* (versão *C++17*). Utilizou-se o *software CMake*, que é um utilitário que auxilia na compilação do código. Inicialmente utilizou-se a *API SOFA* para a manipulação dos arquivos *SOFA*

também no programa como fora anteriormente no *MATLAB*, porém, verificou-se que essa API é mais adequada para a geração desses arquivos e não para a extração dos parâmetros necessários para a geração do efeito binaural. Então, selecionou-se uma API chamada *mySofa*. Com parâmetros de entrada como o nome do arquivo *SOFA* e os ângulos azimute e de elevação (ou ângulos esféricos), a biblioteca retorna os coeficientes da *HRTF*. Adicionalmente, usou-se a biblioteca *PortAudio* para a manipulação de *streams* de áudio. Usou-se a biblioteca *AudioFFT* para as transformadas de *Fourier*, parametrizada para utilizar API aceleradora de *DSP* disponível no *macOS*, *Apple Accelerate*. Adicionalmente, utilizou-se da biblioteca *LibSerialPort* para a comunicação com o dispositivo externo seguidor de cabeça. Por fim, a biblioteca *OpenGL Mathematics* foi usada para calcular a rotação da cabeça do usuário do sistema e compensar a mesma.

Para desenvolvimento e também para a execução do programa, utilizou-se o *software* da *Apple* chamado *Au Lab©*, que serve para aplicar equalizadores e redirecionar saídas e entradas de áudio. Também utilizo-se o *software Soundflower©* que disponibiliza duas placas de som virtuais no *macOS*. Então, com o uso desses programas, utiliza-se o fluxo da figura 19. Interessante notar que o primeiro bloco, *programa reproduzindo áudio*, pode ser, por exemplo, um tocador de músicas ou um navegador de internet.

Figura 19 – Fluxo de áudio.

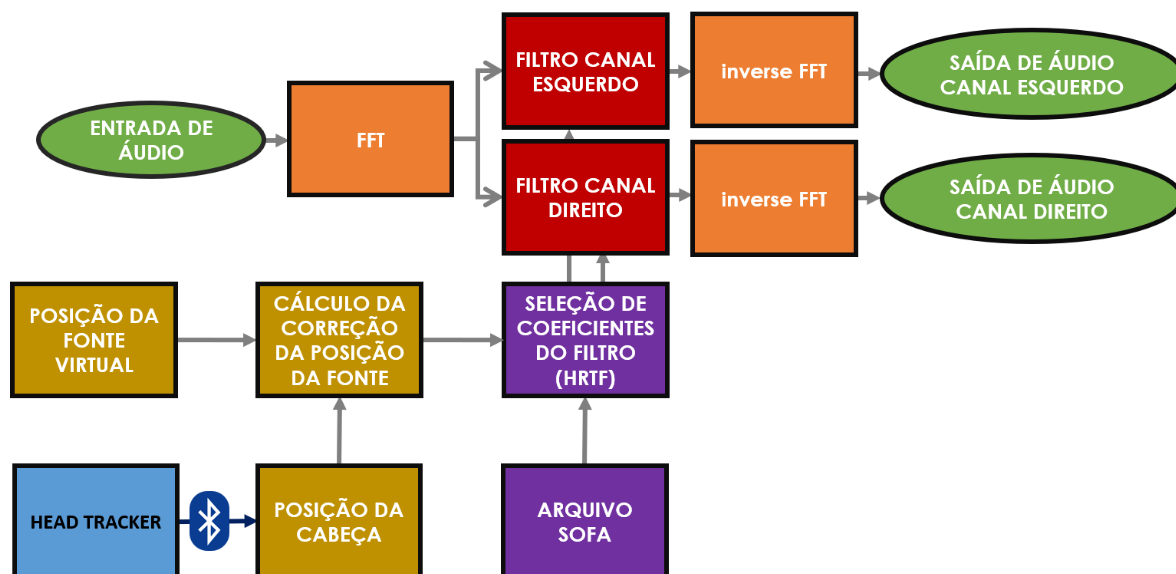



Na próxima sub-seção será apresentado um diagrama de blocos abstraído do programa, que salienta os principais pontos que são considerados em sua implementação. Na sub-seção seguinte, serão explicadas as bibliotecas usadas e suas funcionalidades, e em seguida particularidades da implementação. Seguindo isso, a implementação própriamente dita é abordada.

### 3.4.1 Diagrama de blocos abstraído

Na figura 20, é apresentado uma versão simplificada dos blocos implementados no *software* deste trabalho. Para fins de explicações, deve ser pensado que o áudio de entrada é de um canal apenas (ou seja, monoaural).

Figura 20 – Diagrama de blocos abstraído.




O áudio entra no sistema, em blocos de 512 amostras (configurável em constantes no programa), e é calculada a *FFT* desse bloco de amostras e tem-se o sinal no domínio frequência. Então, os dados no domínio frequência são duplicados, um para cada ouvido do usuário, e passados para os respectivos blocos de filtragem.

Para a seleção dos coeficientes do filtro, ou seja, a *HRTF* mais adequada, o programa considera duas entradas: a posição da fonte sonora virtual (ou seja, *onde deseja-se que a caixa de som virtual esteja*) – dada por um ângulo de azimute e um de elevação –, e a rotação da cabeça do usuário dada pelo *head-tracker* – a comunicação com o *head-tracker* dá-se através do *Bluetooth*, e o dispositivo, quando solicitado, retorna ao programa três ângulos de rotação: nos eixos $x$, $y$ e $z$ do sistema de coordenadas retagulares. Então, com essas duas informações e a rotação inicial da cabeça do usuário, é calculada a posição que a fonte de áudio virtual deve ficar, para que quando o usuário se mover, a mesma seja *corrigida*, ou seja, que a fonte sonora fique *parada* para o usuário.

Uma vez definida a posição da fonte sonora, uma *HRTF* para cada ouvido é extraída do arquivo *SOFA* aberto, para aquela posição (sempre dada em ângulos de azimute e elevação). Essas duas *HRTFs* são passadas para os seus respectivos blocos de filtragem. A *HRIR* dessas *HRTF* têm 200 amostras cada, e foi utilizado um arquivo *SOFA* da biblioteca *CIPIC*.

A filtragem propriamente dita, que é uma convolução no domínio tempo, é uma multiplicação no domínio frequência. Como em sinais nos quais a *FFT* é aplicada a multiplicação gera uma convolução circular e não linear como é desejado, o algorítmo

*overlap and add* foi aplicado de forma distribuída nos blocos, para que a convolução seja linear. Foi utilizado uma sobreposição de 50%.

Após filtrados, os sinais têm as suas respectivas *FFT* inversas calculadas, e então são encaminhados para o sistema operacional para que sejam reproduzidos nos fones de ouvidos do usuário.

### 3.4.2 Considerações adicionais da implementação

Apesar de descrito na seção 3.4.1 que o áudio é capturado em blocos de 512 amostras, como será sugerido pelas seções posteriores, o áudio é capturado em blocos de 4096 amostras, com uma frequência de amostragem de 48000 Hz, com os dados de áudio no formato *float*.

Esse tamanho de amostras dá-se pelo fato que, inicialmente, pretendia-se a aplicação de filtros *BRIR*, que, por exemplo, em (PIKE; ROMANOV, 2017) é utilizado uma resposta a um impulso de 16384 amostras, a uma taxa de amostragem de $48KHz$. Ou seja, dependendo o tipo de filtro *BRIR* aplicado, ter-se-ia que aumentar mais o tamanho do bloco de captura. Então, de forma que a correção da posição da cabeça não tivesse atrasos adicionais no processo de filtragem no qual a *HRTF* é aplicada, internamente o programa divide o bloco maior em blocos menores, finalmente de 512 amostras. Os filtros *FIR* (resposta ao impulso finita) da biblioteca *CIPIC* utilizados tem tamanho de 200 amostras.

O tamanho desses filtros foi o principal motivo que optou-se por realizar a filtragem no domínio frequência com o uso da FFT para a transformação do sinal.

### 3.4.3 Bibliotecas de *software*

Nesta sub-seção serão abordadas as bibliotecas usadas nesse programa.

#### 3.4.3.1 Comunicação *bluetooth*

Como explicado na seção 2.10.3, o perfil usado nesse programa, *Bluetooth Serial Port Profile*, abstrai uma porta serial, não havendo necessidade de um programa que gerencie o *bluetooth* propriamente dito.

De toda forma, inicialmente optou-se por usar uma biblioteca projetada para esse perfil, chamada *bluetooth-serial-port* (AGAMNENTZAR, 2018) e disponível no *GitHub*. No entanto, ao usar essa biblioteca verificou-se que o suporte dela ao sistema operacional *macOS* era limitado (por exemplo, ela não permite, quando usada nesse sistema, verificar se existem novos dados disponíveis para serem lidos). Então, ao se verificar as opções, notou-se as seguintes: ou usar uma biblioteca que abstrai a porta serial, ou fazer as operações de arquivos manualmente. Ao fazer a operação de arquivos manualmente, perder-se-ia

alguma possibilidade de multi-plataforma (verificar capítulo 5), pois dentro da família de sistemas *Unix* já muda um pouco esse procedimento. Além de que, consumiria muito tempo de desenvolvimento e testes de operações de arquivos específicas para porta serial, e não no objeto final deste projeto. Então, iniciou-se a procura ou por bibliotecas *bluetooth* alternativas, ou *serial*. Se usada as do primeiro tipo, elas poderiam eventualmente integrar no *software* a parte do pareamento.

Optou-se, entretanto, após algumas tentativas, pela biblioteca *libserialport* (SIGROK, ). Uma das vantagens dessa biblioteca é ela ser multi-plataforma, adicionando essa futura possibilidade. A segunda é que todas as funções necessárias funcionaram de forma adequada no sistema operacional *macOS*.

Foi criada uma classe, chamada *c_serial*, e todas as funções pertinentes a biblioteca foram inseridas nessa classe. As funcionalidades utilizadas foram:

- listagem de dispositios seriais disponíveis;
- conexão com um dispositivo serial, com seleção de velocidade;
- verificação de dados disponíveis;
- envio de dados;
- recepção de dados.

#### 3.4.3.2 Manipulação de Arquivos *SOFA*

O uso de arquivos SOFA, como mencionado na seção 2.7, é importante no presente trabalho uma vez que esses arquivos incluem as funções de transferências que permitem, enfim, a geração do áudio 3D. Inicialmente pensou-se em usar a biblioteca padronizada *libSOFA* (*Spatially Oriented Format for Acoustics, C++ library*) (SOFA, 2017), porém, com o uso dela, além de que a mesma pedia uma quantidade grande de dependência de outras bibliotecas, a mesma tinha uma documentação não desejável, e muitas funções que não seriam úteis – a biblioteca também serve para a geração e modificação desses arquivos Na procura por alternativas, encontrou-se a biblioteca *mySOFA* (HOENE et al., 2017), que segue a mesma especificação da anterior, a *AES69-2015* (MAJDAK; NOISTERNIG, 2015) (da sociedade dos engenheiros de áudio, *AES*), e que tem por objetivo ser uma biblioteca que, especificamente, possibilita a extração de coeficientes para a realização da filtragem.

Os recursos dessa bibliotecas são utilizados por uma classe chamada *c_master_sofa* (essa, como será visto nas próximas seções, é chamada pela classe *c_sofa*, que realiza a filtragem própriamente dita, no domínio frequência). As funcionalidades utilizadas da biblioteca foram:

- carregamento de um arquivo *SOFA* para uma dada frequência de amostragem, e para uma dada quantidade de coeficientes nos filtros a serem carregados;
- conversão para o sistema de coordenadas internas da biblioteca, a partir de coordenadas esféricas;
- carregamento do filtro para o ouvido direito e esquerdo para uma dada posição relativa da cabeça, e para um dado atraso para compensação da distância da fonte sonora.

Interessante notar que, como foi usado o raio fixo de 1 metro para a fonte sonora, o atraso tende a ser zero, a menos que o arquivo *SOFA* carregado utilize algum padrão diferente na captura dos coeficientes do filtro.

Essa biblioteca também é capaz de fazer a troca da frequência de amostragem dos filtros *FIR* ou das *HRTF* dos arquivos *SOFA*.

#### 3.4.3.3 Fluxo de áudio dentro do sistema operacional

Para a captura e reprodução de áudio, pensou-se em uma biblioteca que fosse capaz de efetivamente capturar o áudio sendo reproduzido em um sistema em tempo real, trazendo como vantagem o gerador de efeito 3D que pode ser aplicado, então, a qualquer áudio sendo reproduzido no computador: basta reproduzir o áudio através de qualquer programa (como por exemplo um navegador de internet), capturar, processar, e efetivamente reproduzir nos auto-falantes do fone de ouvido. Por isso, optou-se pela biblioteca *Port Audio* (BURK et al., ), que tem esses recursos. Foram usadas as funções dessa biblioteca em classes diferentes do sistema. Na classe *c_pa* foi apenas chamado o construtor e o destrutor da biblioteca, como a mesma requer. Essa classe é chamada no inicio da execução do programa, e portanto ao longo de todo o processo a biblioteca é usável. Na classe *c_config*, na qual é fornecido ao usuário as opções de configuração, que será explicada nas próximas seções, a funcionalidade de listar os dispositivos de áudio do sistema e suas características (quantidade de canais de entrada e de saída). Na classe *c_pa_stream* foram então usados os demais recursos, efetivamente para reprodução ou captura de *streams* de áudio. As funcionalidades, utilizadas nessa última, são:

- funções de *callback*, chamadas pelo sistema operacional para a cópia dos dados brutos de áudio para dentro do programa (captura de áudio), ou do programa para o sistema operacional, a fim de reproduzir. A função de *callback* recebe como parâmetros de entrada dois ponteiros de dados (um para entrada de dados, um para saída de dados), quantidade de quadros a serem copiados, informações de temporização, aviso de dados disponíveis, e um ponteiro para dados configurável pelo usuário;

- abertura de *stream*: que tem como opções gravação ou reprodução, taxa de amostragem, formato dos dados (inteiro, ou de ponto flutuante, e quantidade de bits), quantidade de canais, função de *callback* e ponteiro de dados para a função de *callback*.

Essa biblioteca é multi-plataforma.

#### 3.4.3.4 Matemática por matrizes

Para a realização de algumas operações matemáticas, poderiam serem utilizadas operações simples em sequência. Mas, como existem bibliotecas que otimizam a velocidade dos cálculos, optou-se por utilizar uma para o caso de cálculos matriciais (como os cálculos exposto na seção 3.4.5.6). A biblioteca utilizada, a *OpenGL Mathematics* (SOFTWARE..., b), fornece funções diversas para cálculos geométricos. Como a biblioteca tem disponível, para um dado vetor em uma posição em um sistema de coordenadas retângular 3D, a rotação em um dos eixos, a função foi utilizada diretamente. Essas funções são chamadas pela classe *c_sofa*, já mencionada, que realiza a seleção de filtros *HRTF*. Outros detalhes serão tratados na seção 3.4.4.1. Interessante lembrar que essa biblioteca é multi-plataforma.

#### 3.4.3.5 *Fast Fourier Transform*

Para o cálculo das *FFT*s utilizadas no programa, optou-se pelo uso da biblioteca *AudioFFT* (SOFTWARE..., a), que na prática encapsula outras bibliotecas de cálculo de *FFT* de forma transparante. Uma vantagem evidente de utilizar tal biblioteca é poder selecionar, dentre os algoritmos disponíveis, o mais eficiente em uma dada plataforma. As bibliotecas disponíveis internamente são: *Ooura*, *FFTW3*, *Apple Accelerate*, *KissFFT*. Conforme os testes de desempenho realizados pelo próprio desenvolvidor da biblioteca *AudioFFT*, a que traz mais desempenho é a biblioteca *Apple Accelerate*. Utilizou-se ela. Apesar da mesma reduzir a portabilidade – considerando que a *AudioFFT* é multi-plataforma – do projeto, ao portar o mesmo para outro sistema basta selecionar outra biblioteca, que será chamada de forma transparente pela *AudioFFT*. As funções em sua maior parte concentram-se na classe *c_fft*. Os recursos utilizados dessa biblioteca foram as funções que:

- realiza a incialização interna;
- calcula o número de amostras da *FFT* para uma quantidade de amostras no domínio tempo;
- efetivamente calcula a *FFT*, para um dado vetor de entrada com dados de números de ponto flutuante, e retorna dois vetores de dados – um com a parte real e outro com a imaginária – do resultado;

- calcula a inversa da *FFT*, para dois vetores de entrada – um com a parte real e outro com a parte imaginária dos dados –, retorna os dados no domínio tempo.

### 3.4.4 Particularidades da implementação algorítmica

Três pontos chaves da implementação do programa são discutidos nessa seção.

#### 3.4.4.1 Posicionamento relativo

Para que, no momento que uma pessoa usar esse programa, ela tenha uma experiência mais adequada, como discutido nas seções 2.9.2 e 2.9.3, optou-se, até como exposto no objetivo, pela utilização de um sensor de posição. O sensor utilizado está explicado na sub-seção a seguir (3.4.4.1.1).

Quanto ao método da correção da posição da fonte sonora, no exemplo a seguir com uma fonte apenas, o funcionamento é o seguinte:

- a posição da fonte de áudio é conhecida pelo programa (o usuário insere na incialização);
- no movimento do usuário – como por exemplo uma rotação de trinta graus para esquerda com o pescoço – o sensor acusará uma nova posição;
- o programa lerá essa posição, e fará o movimento contrário ao do usuário na fonte de áudio virtual, de forma que, a mesma fique, do ponto de vista do usuário, estática no ar.

Especificamente, detalhando esse procedimento genérico o algoritmos de posição funciona da seguinte maneira:

- a posição da fonte de áudio é conhecida pelo programa (o usuário insere na incialização);
- a posição do usuário é continuamente lida pelo sensor. Na mudança da mesma, as próximas ações são ativadas;
- é recebido do sensor três ângulos de rotação (ver 3.4.4.1.1), nos eixos X, Y e Z.
- a posição selecionada da virtual da fonte de áudio, que é dada no programa em azimute e elevação (com uma distância – raio – fixa de $1m$), é convertida para posição retângular através de cálculos matemáticos simples;
- a rotação, de forma inversa a do usuário, é realizada na fonte sonora, com o uso da biblioteca *OpenGL Mathematics*, um eixo de cada vez. Ou seja, se o usuário tem

uma rotação de trinta graus no eixo Z, por exemplo, a fonte de áudio virtual sofrerá uma rotação de menos 30 graus no mesmo eixo;

- a coordenada da fonte de áudio virtual é convertida novamente para posições dada em coordenadas esféricas;
- a nova posição é usada na fonte de áudio virtual, selecionando os coeficientes dessa nova posição e não mais da anterior (que poderia ser a original, que foi programada inicialmente).

Expandindo o exemplo incial, tem-se:

- a posição da fonte de áudio é conhecida pelo programa (o usuário insere na incialização) – por exemplo, 90 graus de azimute, zero graus de elevação.;
- no movimento do usuário – como por exemplo uma rotação de trinta graus para esquerda com o pescoço – o sensor acusará uma nova posição;
- a fonte terá como posição, após os cálculos, 60 graus de azimute, de forma que na percepção do usuário ela ficou parada.

###### 3.4.4.1.1 Sensor de posição do usuário

O sensor, utilizado *off-the-shelf*, retorna, através do *bluetooth* com o uso do perfil serial, três valores em ponto flutuante de 32 bits, que correspondem a rotação nos eixos X, Y e Z do mesmo. Para realizar uma leitura, basta enviar um pacote de dados como caractere *k*, e o mesmo enviará um caractere de alinhamento de dados (sempre o *e* comercial *&*), seguido dos três ângulos.

Internamente, o conjunto é um Arduíno com um acelerometro, giroscópio, e magnetômetro. E, claro, conexão *bluetooth*.

Ao utilizar-se o sensor, observou-se que o mesmo tem um tempo de instabilidade inicial, porém ao passar um intervalo de tempo de 10s a 30s, o mesmo estabiliza após a conexão inicial. Portanto, optou-se por esperar esse tempo no inicio da execução do programa.

##### 3.4.4.2 Linearização da convolução no domínio frequência

Como discutido na seção 2.1.5, ao realizar a convolução no domínio frequência, a mesma não será do tipo linear. Para resolver essa questão, comparou-se os dois métodos de linearização da convolução em blocos (é em blocos por que na medida que o sistema operacional disponibliza novos dados, os mesmos vão sendo processados, e reproduzidos, e simultaneamente novos dados continuam a chegar), e optou-se pelo *overlap and add*.

O principal motivo é que esse algoritmo envolvia menos modificações na estrutura do programa, sendo que a maior parte se concentrou em corrigir o tamanho dos *vectors* para terem o tamanho necessário de amostras e preenchimento de zeros onde necessário. No fim da *thread* de execução da classe da transformada de Fourier inversa, foi incluído, após a transformada, a soma dos dados como o algoritmo exige, e a seleção de apenas parte deles para serem encaminhados para a saída de dados.

#### 3.4.4.3 Organização das *Threads*

Como a maior parte dos algoritmos desse programa podem ser executados em paralelo (e algumas seriam de toda forma, como por exemplo as funções de *callbacks* exigidas pela biblioteca *Port Audio* que são executadas segundo a demanda do próprio sistema operacional), optou-se pela utilização de *threads* de execução para cada etapa. Para ilustrar o funcionamento do fluxo de dados, abaixo é mostrado um exemplo:

- o sistema operacional executa o *callback* da classe de entrada dos dados, alimentando a mesma;
- essa classe chama o *callback* da próxima função, encaminhando os dados. Interessante notar que nesse *callback* há um *mutex*, fazendo com que não haja conflito com as diferentes *threads* de execução que possam necessitar do mesmo dado e, esses dados, que chegam – em geral – na forma de vetor, são colocados em uma fila (do tipo primeiro a entrar, primeiro a sair) para a *thread* seguinte;
- a próxima *thread*, na mudança do estado do *mutex*, e com a variável de alerta ligada, é liberada para executar, tirando então as informações novas da fila e executando o processamento necessário. Então, chama o *callback* da próxima etapa;
- esse procedimento é repetido, com os dados passando pelas diversas etapas do processamento, até que chega na classe que prepara os dados para voltarem ao sistema operacional, e o áudio ser efetivamente reproduzido. O sistema operacional, de forma similar a primeira etapa, chamará o *callback* da classe de saída para que receba dados conforme for necessário.

### 3.4.5 Implementação do Programa

Nessa sub-seção, será apresentado o diagrama geral do programa, e após, em cada sub-seção, cada um dos blocos que o compõe é apresentado, salientando suas características. Cada um desses blocos é uma classe no programa. Cada classe, de forma geral, tem um construtor, e funções de entrada e saída de dados, e uma *thread* (ou seja, uma tarefa que executa em paralelo com as demais do programa) de execução da função propriamente dita do bloco.

Na figura 21 está esquematizado as conexões entre os *blocos* do sistema, ou seja suas classes, desde a captura do áudio de um dispositivo até a gravação final do áudio com o efeito de espacialização em outro dispositivo. O bloco *virtual channel* está destacado nas figuras 22 (parte 1) e 23 (parte 2), pois o próprio é composto por diversos outros blocos.

Figura 21 – Blocos do programa.

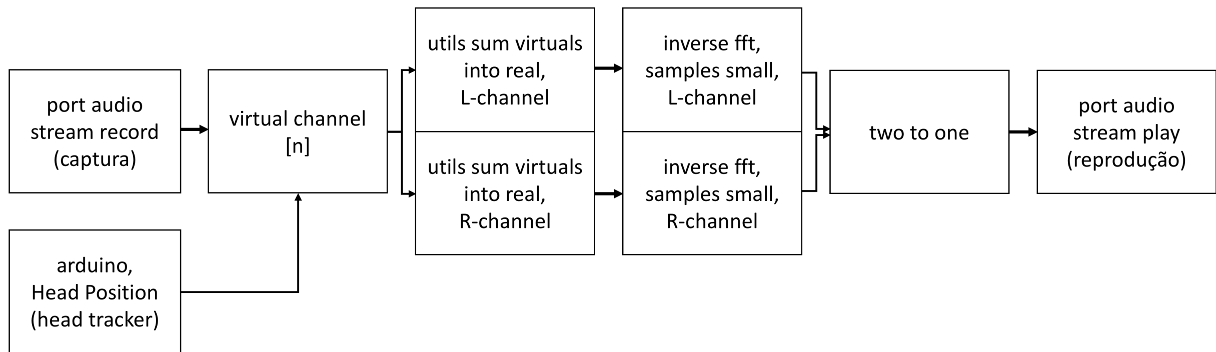



Figura 22 – Bloco *virtual channel*, parte 1.

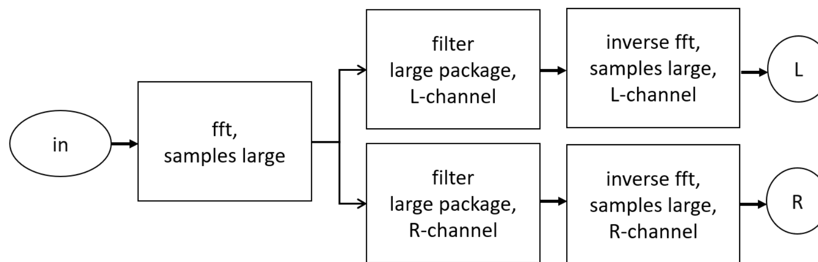
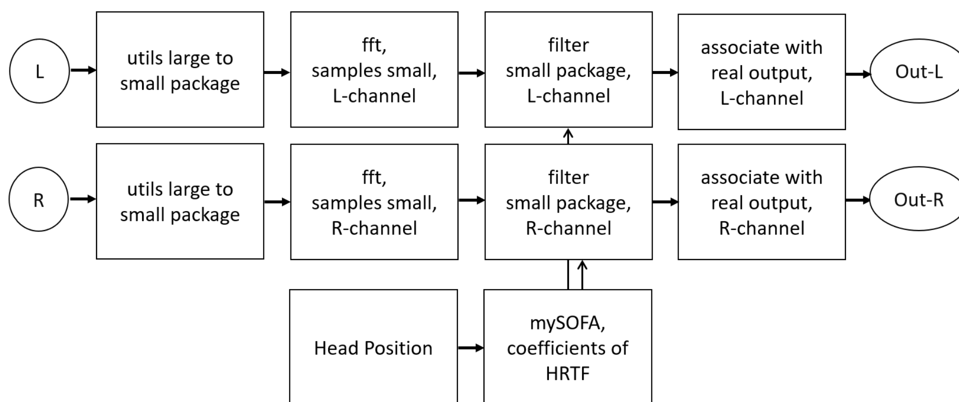



Figura 23 – Bloco *virtual channel*, parte 2.




No corpo do programa, há uma função principal que instancia todos os objetos presentes na figura 21, e associa os *callback* de cada classe: a função da classe que envia os

dados chama o *callback* da classe que recebe os dados. De forma geral, esses dados estão organizados dentro de um objeto do tipo *vector* do *C++*, ou mais de um objeto desse tipo em alguns casos. Essa função principal também instancia objetos do tipo *virtual audio*, que equivale a cada fonte de áudio virtual, e essa classe, por sua vez, instancia os objetos das figuras 22 e 23 e, inclusive, gera novos *callbacks* para a função principal poder enviar dados para o canal virtual. Na atual versão atual do programa, o usuário pode selecionar quantas fontes de áudio deseja e a posição destas.

#### 3.4.5.1 Interface com o usuário

Ao executar o programa, que é executado em terminal, os seguintes passos são seguidos para inicializar completamente o programa:

- é exibido uma lista de dispositivos de áudio disponíveis no computador;
- o usuário seleciona o dispositivo de entrada (ou seja, o fluxo que entrará no programa);
- o usuário seleciona o dispositivo de áudio de saída;
- é exibido uma lista com os dispositivos seriais encontrados no computador;
- o usuário seleciona o dispositivo que corresponde ao *head-tracker*;
- o usuário é perguntado de quantas fontes de áudio virtuais ele deseja;
- então, para cada fonte é perguntado a posição de azimute e de elevação;
- o programa começa a executar.

#### 3.4.5.2 Bloco de captura e reprodução de áudio - port audio stream

A classe responsável por chamar a *API PortAudio* tem duas formas de ser instanciada: em uma, ela faz a leitura dos dados de áudio de um dado dispositivo (gravação do ponto de vista do dispositivo – placa de som); na outra, ela armazena os dados de áudio em um dado dispositivo (do ponto de vista do dispositivo, reproduz).

Dentre os parâmetros utilizados para instanciar uma *stream* de áudio na biblioteca *PortAudio*, são necessários: o formato do dado (inteiro, ponto flutuante, quantos bits), quantos canais, latência, identificação do dispositivo, recursos extras da *API*, frequência de amostragem, amostras por pacote de dado (para cada canal), e o *callback*.

Para a captura dos dados (ou seja, leitura dos dados em um dado dispositivo), toda vez que há dados disponíveis, o sistema operacional chama a função de *callback* do programa, e a mesma recebe um ponteiro de dados para a primeira posição dos mesmos. Então, copia-se esses dados do dispositivo para uma variável local. Os dados de cada canal

estão organizados de forma alternada, da seguinte forma: *canal-1-byte-1, canal-2-byte-1, canal-n-byte-1, canal-1-byte-2, canal-2-byte-2....*

Para a reprodução (ou seja, quando os dados são armazenados no dispositivo), quando o sistema operacional precisa de dados ele chama de igual forma uma função de *callback*. E os dados, para os $n$ canais do dispositivo, devem ser gravados na mesma ordem que estão disponíveis para reprodução. O sistema operacional disponibiliza um ponteiro, e os dados devem ser gravados na posição indicada.

Importante notar que não há controle do programa na questão de quando o sistema operacional irá chamar cada *callback* da biblioteca *PortAudio*, isso será de acordo com a demanda do mesmo. Portanto, incluiu-se um atraso no programa de forma que o mecanismo de reprodução comece a ser executado após o de gravação, com a diferença de alguns milissegundos.

Interessante notar que os dados que saem e chegam nessa classe estão organizados em *vectors* do *C++*, e também fazem uso de *callbacks* da seguinte forma:

- para dados captados do dispositivo de áudio, ela encaminha para a próxima classe chamando o *callback* da mesma;
- para dados que deve ser enviado ao dispositivo de áudio, o bloco anterior (que é o somador de canais virtuais para canal real) chama o *callback*.

#### 3.4.5.3 Bloco de utilidades - *utils*

Essa classe, *utils*, faz as mediações no programa. Uma das utilidades dela é somar diferentes canais em um canal único. Explicando, cada fonte de áudio virtual gera duas saídas físicas: uma para cada ouvido do usuário do sistema. Portanto, essas saídas físicas devem ser colocadas conforme a ordem apresentada em 3.4.5.2 e, caso haja mais de uma fonte virtual, somadas, a fim de serem reproduzidas em conjunto no mesmo fone de ouvido.

Outra utilidade dessa classe é dividir um pacote de amostras grande (na atual compilação, 4096 amostras), para 512 amostras. Se os blocos das figuras 21, 22 e 23 forem observados, vê-se que são executadas duas operações de transformada normal e duas de inversa de Fourier, em conjunto com duas operações de filtragem. Isso ocorre porque, quando foi planejado esse programa, pensou-se em utilizar tanto as *HRIR* (que tem por inversa a *HRTF*) quanto os *BRIR*, sendo esses últimos conhecidos por terem mais coeficientes. Para a versão inicial, posteriormente decidiu-se utilizar apenas os coeficientes *HRIR*, ficando então para uma próxima iteração a adição do *BRIR*, com o código já organizado para tanto. Por fim, essa diferença de número de amostras para cada filtro, acabou gerando a necessidade de quebrar em duas filtragens, até para que a filtragem que

depende do *HRIR* seja atualizada de forma mais rápida ao receber uma nova posição do *head-tracker*.

#### 3.4.5.4 Bloco agregador - dois canais em um vetor de dados

Apesar de também ser uma utilidade, nesse caso optou-se por gerar uma classe independente da *utils*, pois essa classe não faz uso de uma *thread* de execução. A presente classe junta os dados dos canais de áudios reais para o ouvido esquerdo e direto em um único vetor para ser enviado para a classe do *PortAudio*.

#### 3.4.5.5 Bloco de aplicação de filtro

Esse bloco é o que realiza a filtragem. Como seria uma convolução entre os coeficientes do filtro e o áudio de entrada no domínio tempo, como os dados estão já transformados pelo bloco anterior do diagrama no domínio frequência, é realizada uma multiplicação dos coeficientes com a entrada.

Essa classe tem duas entradas de dados: uma para o áudio, atualizada a cada execução, e outra para os coeficientes do filtro, que são atualizados sobre demanda.

#### 3.4.5.6 Bloco *mySOFA*

Esse bloco recebe a posição da cabeça e da fonte de áudio virtual como entradas, abre o arquivo de cabeça *SOFA* selecionado, e através da biblioteca *mySOFA*, extrai os coeficientes do filtro HRTF para uma dada posição. Então, atualiza os coeficientes no filtro.

A posição da fonte de áudio recebida é dada no sistema de coordenadas esférico: azimute e elevação. Já a posição da cabeça é dada por três ângulos de rotação: no eixo x, eixo y e no eixo z.

No algoritmo interno da *thread* desta classe, as coordenadas esféricas são convertidas em posições cartesianas. Então, com o uso da biblioteca *OpenGL Mathematics*, é calculada a posição relativa da fonte através da rotação nos três eixos, como explicado na seção . Por fim, a posição cartesiana é novamente convertida em esférica para a seleção da *hrtf* correta. Então, os coeficientes do filtro são extraídos como mencionado.

#### 3.4.5.7 Bloco de *fast Fourier transform*

Essa classe realiza a transformada rápida de Fourier e a transformada inversa. Como entrada do bloco de transformada é um dado de uma dimensão, em um vetor. A saída é um vetor de duas dimensões no domínio frequência. Para realizar a transformada, é usada a *API AudioFFT*.

Para o bloco de transformada inversa, a entrada é o vetor bi-dimensional e a saída é um dado de uma dimensão. Esse bloco faz, também, a aplicação do algoritmo *overlap and add*, conforme explicado na seção 3.4.4.2.

#### 3.4.5.8 Bloco do conector com a *HeadTracker*: *head position*

Esse bloco é na realidade um conjunto de classes. Uma faz a conexão propriamente dita com o dispositivo externo. A outra classe, instancia a primeira, interpreta os dados recebidos e encaminha através de um *callback* para a classe seguinte. Um detalhe importante é que essa classe seguinte, referida aqui como *bloco mySofa*, está dentro de um canal virtual, ou seja, para cada canal inserido, a posição relativa virtual da cabeça deve ser calculada individualmente para uma dada posição de fonte de áudio virtual.

Em relação a primeira classe, a mesma conecta, usando a biblioteca *Libserialport*, com um dispositivo externo através de bluetooth. O bloco conector, recebe uma referência para o objeto de conexão, a ser explicado na seção x. Recebe os dados do bloco externo, de três ângulos diferentes, de rotações nos eixos x, y e z. Envia o ângulo recebido para a *thread* da classe que calcula a posição relativa da fonte de som virtual em relação ao usuário.

#### 3.4.5.9 Bloco de canais virtuais

Como pode ser observado nas figuras 22 e 23, o bloco de canais virtuais na realidade é composto de outros blocos. Isso se dá porque para cada fonte de áudio virtual um filtro diferente deve ser aplicado para cada saída de som real (ou ouvido do usuário). Então, em vez de instanciar cada bloco individualmente para cada canal, optou-se for criar uma classe que inclui eles, e com isso, o usuário pode parametrizar na execução do programa quantas fontes o mesmo desejar.

# 4 Resultados

Um *software* de computador foi desenvolvido, que aplica filtros em uma *stream* de áudio, de forma que a sensação de áudio espacializado a ser reproduzida em fones de ouvido é criada. As especificações desse programa incluem:

- suporta mais que uma fonte de som virtual (conforme o sistema operacional disponibilizar, no *macOS* com o uso do *Soundflower* são até 64) – o uso de até 4 fontes foi testado com sucesso;
- uso de um *head-tracker* para a correção de posição da cabeça do usuário em três eixos de rotação, ou seja, quando o usuário rotaciona a cabeça em qualquer exio do sistema retangular de coordenadas: $x$, $y$ e $z$;
- tempo real (um pequeno atraso de até um segundo pode ser notado) para qualquer programa sendo executado no sistema operacional (ou até uma fonte de som externa como um microfone);
- uso de arquivos *sofa* com *HRTFs* – o arquivo utilizado por padrão foi o que o autor considerou mais adequado para si, originário da biblioteca *CIPIC*;
- todas as bibliotecas usadas são *multi-plataformas*, sendo que o programa poderá ser futuramente portado.

Para testar a correção da posição nos três eixos com o uso do *head-tracker*, o autor posicionou uma fonte sonora, por vez, de forma que a mesma fosse afetada maximamente pelo movimento do eixo a ser testado: em $(\phi, \theta) = (90°, 0°)$, para a rotação em relação a um eixo que corta a cabeça de cima a baixo centralmente; em $(\phi, \theta) = (0°, 0°)$ para o eixo que corta entre as duas orelhas; e novamente em $(\phi, \theta) = (90°, 0°)$ para o eixo que corta a cabeça no eixo frente-atrás (aponta na direção do nariz). O autor teve a percepção de funcionamento do sistema.

Com a implementação do sistema completada, a fim de testar se o efeito de *externalização* (ou seja, o áudio 3D) era gerado no fone de ouvido, e não a sua qualidade, alguns colegas do *Laboratório de Processamento de Sinais e Imagens*, onde esse projeto foi desenvolvido, usaram o mesmo. Uma fonte de áudio virtual foi selecionada no programa, na posição $(\phi, \theta) = (135°, 0°)$, e um áudio no qual a voz humana era predominante foi reproduzido pelo computador. Mais da metade conseguiu adequadamente, ao usar e se movimentar com o fone de ouvido com o *head-tracker* acoplado, localizar a posição da fonte sonora virtual. Um grupo menor de pessoas localizou a fonte sonora em uma região maior no espaço e um colega não sentiu efeito de externalização.

Como usuário do próprio sistema, o autor notou que o sistema funcionou bem para si, entretanto, enquanto o sistema funciona muito bem para posições com azimute compreendidas aproximadamente entre $50° \leq \phi \leq 310°$, conforme a fonte virtual fica posicionada mais próximo da frente da cabeça (ou seja, aproximadamente entre $-50° < \phi < 50°$ de azimute), maior a chance do som ser *projetado* em cima da cabeça (similar a uma lateralização, porém frontal) e não na posição correta afastada do usuário.

O *head-tracker* utilizado também tem algumas limitações: no máximo 25 leituras por segundo e um atraso considerável (cerca de $40ms$). Isso somado ao fato que o programa não tenta *prever* a posição futura do usuário, quando o mesmo se movimenta muito rápido, este pode notar a fonte de som virtual se movimentando mais lentamente com um certo atraso em relação ao próprio movimento.

# 5 Conclusão

Iniciou-se esse trabalho apresentando o quão util é a geração de áudio 3D: de aplicações relacionadas ao entreterimento – como em jogos de computadores –, quanto assistivas – como em dispositivos para pessoas com deficiência visual. Então, houve o estudo de literatura prévia, que foi de igual forma também apresentada, de psico-acústica e estudos específicos sobre áudio 3D, a algoritmos de processamento de dados e especificamente áudio.

E finalmente, no capítulo *implementação* o objetivo descrito no primeiro capítulo foi desenvolvido. Um programa que efetivamente gera o efeito 3D, através da aplicação de filtros, em tempo real com a correção da fonte de áudio virtual com o uso de um *head-tracker*. Esses filtros têm coeficientes capturados de pessoas reais com características físicas diferentes, e estão disponíveis para o uso em bibliotecas padronizadas, como a utilizada *CIPIC* (ALGAZI et al., 2001). A primeira vantagem a ser destacada desse programa é que, como ele é em tempo real, além de que ele pode ser aplicado em qualquer áudio reproduzido no computador, ele não depende de um aparato para gravação especial: é aplicado o filtro, e o efeito 3D é gerado. Optou-se pelo uso de bibliotecas multi-plataforma sempre que possível, para que futuramente o programa possa ser portado para outras plataformas com facilidade e não exija extenso re-trabalho. A opção por *threads* se deu já que são atividades que naturalmente podem ocorrer de forma paralela, e conforme o mais canais virtuais de áudios são utilizados no programa, como o processamento de cada canal é paralelo e independente, não será atrasado o processamento dos outros canais. Adicionalmente, caso o sistema seja portado para uma plataforma móvel, por exemplo, com recursos de processamento limitados pelo consumo energético, porém usualmente paralelo, por utilizar mais de uma *thread*, esse programa pode ser executado em processadores multicores naturalmente. E, caso eventualmente se verifique que uma das etapas tem um consumo energético alto, essa parte pode ser portada para *hardware*, pois o programa é extremamente modular.

Como sugestão de continuídade, é interessante o estudo dos filtros *BRIR* e a adição dos mesmos. O programa já tem uma *espera* para o filtro baseado em *BRIR*, como descrito no capítulo *implementação*. Adicionalmente, a elaboração de uma rotina de treinamento para o usuário pode melhorar a percepção do áudio 3D sem grandes alterações estruturais no programa, como foi sugerido pela literatura apresentada.

# Referências Bibliográficas

AGAMNENTZAR. *Software: bluetooth-serial-port*. 2018. Disponível em: <https://github.com/Agamnentzar/bluetooth-serial-port>.

ALGAZI, V. R.; DUDA, R. O. Effective use of psychoacoustics in motion-tracked binaural audio. In: IEEE. *Multimedia, 2008. ISM 2008. Tenth IEEE International Symposium on*. [S.l.], 2008. p. 562–567.

ALGAZI, V. R. et al. The cipic hrtf database. In: IEEE. *Applications of Signal Processing to Audio and Acoustics, 2001 IEEE Workshop on the*. [S.l.], 2001. p. 99–102.

BLUETOOTH, S. Bluetooth specification version 1.1. 2001. Disponível em: <http://www.bluetooth.com>.

BURK, P.; OTHERS. *Software: PortAudio - an Open-Source Cross-Platform Audio API*. Disponível em: <http://www.portaudio.com/>.

CHO, S. J.; OVCHARENKO, A.; CHONG, U.-p. Front-back confusion resolution in 3d sound localization with hrtf databases. In: IEEE. *Strategic Technology, The 1st International Forum on*. [S.l.], 2006. p. 239–243.

FITZPATRICK, W.; WICKERT, M.; SEMWAL, S. 3d sound imaging with head tracking. In: IEEE. *Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE), 2013 IEEE*. [S.l.], 2013. p. 216–221.

GULICK, W. L. et al. Hearing: Physiology and psychophysics. Oxford University Press, 1971.

HADAD, E. et al. A study of 3d audio rendering by headphones. In: IEEE. *Electrical & Electronics Engineers in Israel (IEEEI), 2014 IEEE 28th Convention of*. [S.l.], 2014. p. 1–4.

HOENE, C.; MEJIA, I. C. P.; CACEROVSCHI, A. Mysofa—design your personal hrtf. In: AUDIO ENGINEERING SOCIETY. *Audio Engineering Society Convention 142*. [S.l.], 2017.

HOWARD, D. M. *Acoustics and psychoacoustics*. [S.l.]: Taylor & Francis, 2017.

KAMAT, V. R.; EL-TAWIL, S. Evaluation of augmented reality for rapid assessment of earthquake-induced building damage. *Journal of computing in civil engineering*, American Society of Civil Engineers, v. 21, n. 5, p. 303–310, 2007.

KUNDUR, D. *Presentation: Overlap-Save and Overlap-Add*. Disponível em: <https://pdfs.semanticscholar.org/presentation/6197/38ee171d1966471f4e739d166296e7080bc6.pdf>.

MAJDAK, P. et al. Spatially oriented format for acoustics: A data exchange format representing head-related transfer functions. In: AUDIO ENGINEERING SOCIETY. *Audio Engineering Society Convention 134*. [S.l.], 2013.

MAJDAK, P.; NOISTERNIG, M. Aes69-2015: Aes standard for file exchange-spatial acoustic data file format. *Audio Engineering Society*, 2015.

MOON, P.; SPENCER, D. E. *Field theory handbook: including coordinate systems, differential equations and their solutions.* [S.l.]: Springer, 2012.

OPPENHEIM, A. V.; SCHAFER, R. W. *Discrete-time Signal Processing.* [S.l.]: Pearson Education, 2009.

OXFORD. *Oxford English Dictionary.* [S.l.]: JSTOR, 2003.

PEŁCZYŃSKI, P.; BOCHEŃSKA, M. Real time implementation of dynamic 3d sound sources in a digital signal processor. *IFAC Proceedings Volumes*, Elsevier, v. 45, n. 7, p. 277–282, 2012.

PELKOWITZ, L. Frequency domain analysis of wraparound error in fast convolution algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, IEEE, v. 29, n. 3, p. 413–422, 1981.

PIKE, C.; ROMANOV, M. An impulse response dataset for dynamic data-based auralization of advanced sound systems. In: AUDIO ENGINEERING SOCIETY. *Audio Engineering Society Convention 142*. [S.l.], 2017.

SANDBERG, S. et al. Using 3d audio guidance to locate indoor static objects. In: SAGE PUBLICATIONS SAGE CA: LOS ANGELES, CA. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting.* [S.l.], 2006. v. 50, n. 16, p. 1581–1584.

SHYNK, J. J. et al. Frequency-domain and multirate adaptive filtering. *IEEE Signal Processing Magazine*, v. 9, n. 1, p. 14–37, 1992.

SIGROK. *Software: Libserialport: cross-platform library for accessing serial ports.* Disponível em: <https://sigrok.org/api/libserialport/0.1.1/>.

SOFA. *Software: [S]patially [O]riented [F]ormat for [A]coustics, C library.* 2017. Disponível em: <https://github.com/sofacoustics/API_Cpp/>.

SOFTWARE: AudioFFT. HiFi-LoFi. Disponível em: <https://github.com/HiFi-LoFi/AudioFFT>.

SOFTWARE: OpenGL Mathematics. G-Truc Creation. Disponível em: <https://glm.g-truc.net/0.9.9/index.html>.

SUNDARESWARAN, V. et al. 3d audio augmented reality: implementation and experiments. In: IEEE COMPUTER SOCIETY. *Proceedings of the 2nd IEEE/ACM international symposium on mixed and augmented reality.* [S.l.], 2003. p. 296.

TANENBAUM, A. S. *Modern operating system.* [S.l.]: Pearson Education, Inc, 2009.

VORLÄNDER, M. *Auralization: fundamentals of acoustics, modelling, simulation, algorithms and acoustic virtual reality.* [S.l.]: Springer Science & Business Media, 2007.

WABNITZ, A. et al. Room acoustics simulation for multichannel microphone arrays. In: *Proceedings of the International Symposium on Room Acoustics.* [S.l.: s.n.], 2010. p. 1–6.

WU, R.; YU, G. Improvements in hrtf dataset of 3d game audio application. In: IEEE. *Audio, Language and Image Processing (ICALIP), 2016 International Conference on*. [S.l.], 2016. p. 185–190.

YOST, W. A.; FAY, R. R. *Auditory perception of sound sources*. [S.l.]: Springer Science & Business Media, 2007. v. 29.

ZAHORIK, P. et al. Perceptual recalibration in human sound localization: Learning to remediate front-back reversals. *The Journal of the Acoustical Society of America*, ASA, v. 120, n. 1, p. 343–359, 2006.

ZWICKER, E.; FASTL, H. *Psychoacoustics: Facts and models*. [S.l.]: Springer Science & Business Media, 2013. v. 22.

# Anexos

# ANEXO A – Código fonte dos programas de teste realizados em *MATLAB*

Primeiro programa desenvolvido em Matlab.

```
%Funcao que retorna uma stream processada de um dado audio
%por uma dada cabeca
%function [stream1]=quemdiria(cabeca,angulo,caixa1,caixa2,arquivo)
%Pede:  cabeca (numero da cabeca cipic)
%   angulo (da caixa 1, a caixa 2 será 360-angulo)
%   caixa 1 (volume 0.0 a 1.0)
%   caixa 2 (volume 0.0 a 1.0)
%   arquivo (nome do arquivo .wav)
%retorna stream: play(stream1) para ouvir pause(stream1) para parar
%exemplo:
%stream1=teste1(11,90,0.4,0.4,)
%
function [stream1]=teste(cabeca,angulo,caixa1,caixa2,arquivo)
if caixa1>1
    caixa1=1;
end
if caixa2>1
    caixa2=1;
end
ang=angulo;
m=jl_sofa_load('CIPIC',cabeca);
x=jl_sofa_hrir_i(m,ang,0,1);
y=jl_sofa_hrir_i(m,(360-ang),0,1);
a=audioread(arquivo);
x1=x(:,1:1);
x2=x(:,2:2);
y1=y(:,1:1);
y2=y(:,2:2);
a1=a(:,1:1)*caixa1;
a2=a(:,2:2)*caixa2;
%para a1
s1_1=jl_fir(x1,a1);
s1_2=jl_fir(x2,a1);
%para a2
s2_1=jl_fir(y1,a2);
s2_2=jl_fir(y2,a2);
s_1=s1_1+s2_1;
```

```
s_2=s1_2+s2_2;
s=[s_1,s_2];
stream1=audioplayer(s,44100);
%stream2=audioplayer(a,44100);
%play(stream1);
```

Segundo programa desenvolvido em Matlab.

```
%Funcao que retorna uma stream processada de um dado audio
%por uma dada cabeca, fazendo o movimento definido por rangeangulo
%ou seja, o som anda do angulo até angulo+rangeangulo
%function [stream1]=quemdiria(cabeca,angulo,rangeangulo,altura,arquivoaudio
    )
%Pede:  cabeca (numero da cabeca cipic)
%    angulo (da caixa 1, a caixa 2 será zero nesse exemplo, e o som tocado é
     mono)
%    rangeangulo (o quanto o angulo deve incrementar até o fim do arquivo)

%    altura (angulo de altura)
%    arquivo (nome do arquivo .wav)
%retorna stream: play(stream1) para ouvir pause(stream1) para parar
%exemplo:
%stream1=quemdiria2(11,0,360*2,20,'sound_0.wav');play(stream1);
%Há uma clipagem terrivel no som

function [stream1,saida]=teste(cabeca,angulo,rangeangulo,altura,
    arquivoaudio)
ang=angulo;
ang2=0;
range=rangeangulo;
height=altura;
a=audioread(arquivoaudio);
if strcmp(arquivoaudio,'sound_1.wav')==1
    a=[a;a;a;a;a;a]; %improviso para aumentar a duracao do audio
end
a1=a(:,1:1)*0.3;
a1=a1+a(:,2:2)*0.3;
a2=a(:,2:2)*0.0;
m=jl_sofa_load('CIPIC',cabeca);
s_1=[];
s_2=[];
x_t=jl_sofa_hrir_i(m,0,0,1);
head_samples=size(x_t,1);
x_t=[];
size_a=size(a1,1);
slices=ceil(2*range);
i_a=0;
```

```
while (slices>range)
    i_a=i_a+1;
    slices = floor(size_a/(head_samples * i_a)) ;
end
samples_per_slice=head_samples*i_a;
for i=1:slices;
    ang0_l=ang+ang2+((range/slices))*i;
    ang1_l=ang0_l+180;
    while (ang0_l>360)
        ang0_l=ang0_l-360;
    end
    while (ang1_l>360)
        ang1_l=ang1_l-360;
    end
    x=jl_sofa_hrir_i(m,ang0_l,height,1);
    y=jl_sofa_hrir_i(m,ang1_l,height,1);
    x1=x(:,1:1);
    x2=x(:,2:2);
    y1=y(:,1:1);
    y2=y(:,2:2);
    start_s=(i-1)*samples_per_slice+1;
    end_s=(samples_per_slice*i);
    u1=a1(  start_s:end_s, : );
    u2=a2(  start_s:end_s, : );
    %para a1
    s1_1=jl_fir(x1,u1);
    s1_2=jl_fir(x2,u1);
    %para a2
    s2_1=jl_fir(y1,u2);
    s2_2=jl_fir(y2,u2);
    s_1=[s_1;s1_1+s2_1];
    s_2=[s_2;s1_2+s2_2];
end
s=[s_1,s_2];
s(s>1)=s(s>1)*0;
s(s<-1)=s(s<-1)*0;
stream1=audioplayer(s,44100);
saida=s;
%stream2=audioplayer(a,44100);
%play(stream1);
```

# ANEXO B – Código Fonte do Programa C++

Código fonte do arquivo CMAKE auxiliar para a compilação.

```
cmake_minimum_required(VERSION 3.6)
project(TCC)
set(CMAKE_MODULE_PATH "${PROJECT_SOURCE_DIR}/cmake;${CMAKE_MODULE_PATH}")
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fvectorize -ffast-math -Wall")
set(SOURCE_FILES main.cpp)
list(APPEND SOURCE_FILES ./AudioFFT/AudioFFT.cpp)
add_executable(TCC ${SOURCE_FILES})
find_package(Accelerate)
target_link_libraries(TCC ${Accelerate_LIBRARIES})
message(${Accelerate_LIBRARIES})
add_subdirectory(portaudio)
target_link_libraries(TCC portaudio)
find_library(libmysofa HINTS "/usr/local/lib/")
target_link_libraries(TCC mysofa)
find_library(libserialport HINTS "/usr/local/lib/")
target_link_libraries(TCC serialport)
```

Código fonte do programa desenvolvido. Licença BSD.

```
#include <iostream>
#include <list>
#include <vector>
#include <portaudio.h>
#include "./AudioFFT/AudioFFT.h"
#include <thread>
#include <queue>
#include <memory>
#include <ctime>
#include <sstream>
#include <algorithm>
#include <bitset>
#include <functional>
```

```
#include <cstdio>
#include <cstdlib>
#include <sys/ioctl.h>
#include <sys/syslimits.h>
#include <fstream>
#include <cmath>
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <random>
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::rotate, glm
    ::scale, glm::perspective
#include <glm/gtx/rotate_vector.hpp>
#include <libserialport.h>
#if __has_include(<optional>)
    #include <optional>
    template<typename T>
    using Optional = std::optional<T>;
#elif __has_include(<experimental/optional>)
    #include <experimental/optional>
    template<typename T>
    using Optional = std::experimental::optional<T>;
#else
    #error "No optional support in C++ standard library!"
#endif
#include <mysofa.h>
#define A_INPUT 2
#define A_OUTPUT 3
#define A_BLUETOOTH 0
#define A_BLUETOOTH_DELAY_1 0
#define A_BLUETOOTH_DELAY_2 2
#define A_BLUETOOTH_FAIL 400
#define SAMPLES_FFT_LARGE 4096
#define SAMPLES_FFT_SMALL 512
#define WINDOW_FFT_LARGE audiofft::AudioFFT::ComplexSize(SAMPLES_FFT_LARGE)
#define WINDOW_FFT_SMALL audiofft::AudioFFT::ComplexSize(SAMPLES_FFT_SMALL)
#define DELAY_SHORT 100us
#define DELAY_LONG 1ms
#define SAMPLE_RATE 48000
#define TOTAL_BUFFERS 5000
typedef float paSampleF;
using coordinate = float;
struct source_position{
    coordinate azimuth;
    coordinate elevation;
```

```
};
class c_text_flux{
    std::mutex mutex;
public:
    c_text_flux(){
    }
    void write_to_screen(std::string str){
        std::unique_lock<std::mutex> lock(mutex);
        std::cout << str << std::endl;
        return;
    }
};
class c_master_sofa{
private:
    std::string file;
    int32_t frequency;
    struct MYSOFA_EASY *hrtf;
    int filter_length;
    int err;
    int32_t fft_size;
    audiofft::AudioFFT fft;
public:
    c_master_sofa(std::string file_l, int32_t frequency_l, int32_t
       fft_size_):file(file_l),frequency(frequency_l){
        fft_size=fft_size_;
        fft.init(fft_size*2);
        hrtf = mysofa_open(file.data(), frequency, &filter_length, &err);
        if (hrtf==NULL){
            std::cout<< "ERR HRTF" <<std::endl;
            std::cout << "sai 2" << std::endl;
            exit(-28);
        }
    }
    ~c_master_sofa(){
        mysofa_close(hrtf);
    }
    Optional<std::tuple<std::vector<paSampleF>, std::vector<paSampleF>,
       float, float>> filter_coef(int32_t azimuth, int32_t elevation){
        if(!(hrtf==NULL)){
            std::vector<paSampleF> leftIR (filter_length);
            std::vector<paSampleF> rightIR (filter_length);
            float leftDelay;
            float rightDelay;
            float values[3];
            values[0]=(float) azimuth;
            values[1]=(float) elevation;
            values[2]=(float) 1;
```

```
            mysofa_s2c(values);
            mysofa_getfilter_float(hrtf, values[0], values[1], values[2],
                leftIR.data(), rightIR.data(), &leftDelay, &rightDelay);
            return {{std::move(leftIR),std::move(rightIR), rightDelay,
                leftDelay}};
        }
        return {};
    }
    Optional<std::tuple<std::vector<paSampleF>, std::vector<paSampleF>, std
        ::vector<paSampleF>, std::vector<paSampleF> >> filter_coef_fft(
        int32_t azimuth, int32_t elevation){
        auto coefficients = filter_coef(azimuth, elevation);
        std::vector<paSampleF> output_re_r (audiofft::AudioFFT::ComplexSize
            (fft_size*2));
        std::vector<paSampleF> output_im_r (audiofft::AudioFFT::ComplexSize
            (fft_size*2));
        std::vector<paSampleF> output_re_l (audiofft::AudioFFT::ComplexSize
            (fft_size*2));
        std::vector<paSampleF> output_im_l (audiofft::AudioFFT::ComplexSize
            (fft_size*2));
        auto & coefficients_value = *coefficients;
        auto input_r = std::move(std::get<0>(coefficients_value));
        auto input_l = std::move(std::get<1>(coefficients_value));
        input_r.resize(fft_size*2,0.0f);
        input_l.resize(fft_size*2,0.0f);
        fft.fft(input_r.data(), output_re_r.data(), output_im_r.data());
        fft.fft(input_l.data(), output_re_l.data(), output_im_l.data());
        return {{std::move(output_re_r), std::move(output_im_r), std::move(
            output_re_l), std::move(output_im_l)}};
    }
};
class c_sofa{
private:
    void sofa_function_start(){
        int32_t i=0;
    }
public:
    c_sofa (const c_sofa&) = delete;
    using filter_coeficients = std::vector<paSampleF>;
    using callback = std::function<void(filter_coeficients,
        filter_coeficients)>;
    using vector_callback = std::vector<callback>;
    std::function<void(std::string)> text_function;
    //pede como callback os n-filtros
    c_sofa(vector_callback functions, std::function<void(std::string)>
        text_function_v): callbacks(functions), text_function(
        text_function_v){
```

```
        //LARGE[re,im],small[re,im]
        new_position=false;
        head_w=0;
        head_x=0;
        head_y=0;
        head_z=0;
        source_azimuth=0;
        source_elevation=0;
        //inicializa sofa
        thread = std::thread([this]() { thread_(); });
        thread.detach();
    }
    void set_head_position(bool w, int32_t x, int32_t y, int32_t z){
        std::unique_lock<std::mutex> lock(mutex);
        head_w=w;
        head_x=x;
        head_y=y;
        head_z=z;
        new_position=true;
        cv.notify_one();
    }
    void set_source_position(int32_t azimuth, int32_t elevation){
        std::unique_lock<std::mutex> lock(mutex);
        source_azimuth=azimuth;
        source_elevation=elevation;
        new_position=true;
        cv.notify_one();
    }
private:
    vector_callback callbacks;
    int32_t source_elevation;
    int32_t source_azimuth;
    int32_t head_w;
    int32_t head_x;
    int32_t head_y;
    int32_t head_z;
    std::thread thread;
    std::mutex mutex;
    std::condition_variable cv;
    bool new_position;
    std::tuple<coordinate, coordinate> turn_sound_box(
            coordinate source_azimuth,
            coordinate source_elevation,
            coordinate subject_rotation_x,
            coordinate subject_rotation_y,
            coordinate subject_rotation_z
    ){
```

```
        coordinate theta_r, theta, phi_r, phi, x, y, z;//, rx, ry, rz;
        theta_r = glm::radians(90 - source_elevation);
        phi_r = glm::radians(source_azimuth);
        x = std::sin(theta_r) * std::cos(phi_r);
        y = std::sin(theta_r) * std::sin(phi_r);
        z = std::cos(theta_r);
        auto a = glm::rotate(glm::vec3(x,y,z), glm::radians(
            subject_rotation_x), glm::vec3(1.0, 0.0, 0.0));
        auto b = glm::rotate(a,glm::radians(subject_rotation_y), glm::vec3
            (0.0, 1.0, 0.0));
        auto c = glm::rotate(b,glm::radians(subject_rotation_z), glm::vec3
            (0.0, 0.0, 1.0));
        theta = 90 - glm::degrees( std::atan2( std::hypot(c.x , c.y), c.z))
            ;
        phi = glm::degrees(std::atan2(c.y,c.x));
        return {phi, theta};
    }
    void thread_(){
        size_t i;
        filter_coeficients filter_r_re_coeficients_large(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_LARGE*2));
        filter_coeficients filter_l_re_coeficients_large(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_LARGE*2));
        filter_coeficients filter_r_re_coeficients_small(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_SMALL*2));
        filter_coeficients filter_l_re_coeficients_small(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_SMALL*2));
        filter_coeficients filter_r_im_coeficients_large(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_LARGE*2));
        filter_coeficients filter_l_im_coeficients_large(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_LARGE*2));
        filter_coeficients filter_r_im_coeficients_small(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_SMALL*2));
        filter_coeficients filter_l_im_coeficients_small(audiofft::AudioFFT
            ::ComplexSize(SAMPLES_FFT_SMALL*2));
        c_master_sofa head("../sofa/cipic/subject_020.sofa", 48000,
            SAMPLES_FFT_SMALL);
        for (i=0; i<audiofft::AudioFFT::ComplexSize(SAMPLES_FFT_LARGE*2); i
            ++){
            filter_r_re_coeficients_large[i]=1.0f;
            filter_r_im_coeficients_large[i]=0.0f;
            filter_l_re_coeficients_large[i]=1.0f;
            filter_l_im_coeficients_large[i]=0.0f;
        }
        for (i=0; i<audiofft::AudioFFT::ComplexSize(SAMPLES_FFT_SMALL*2); i
            ++){
            filter_r_re_coeficients_small[i]=0.0f;
```

```
            filter_r_im_coeficients_small[i]=0.0f;
            filter_l_re_coeficients_small[i]=0.0f;
            filter_l_im_coeficients_small[i]=0.0f;
        }
        while(1) {
            std::unique_lock<std::mutex> lock(mutex);
            cv.wait(lock, [this]() {
                        return new_position;
            });
            if (new_position == true) {
                auto [relative_azimuth, relative_elevation] =
                    turn_sound_box(
                        (coordinate) source_azimuth,
                        (coordinate) source_elevation,
                        (coordinate) head_x,
                        (coordinate) head_y,
                        (coordinate) head_z
                );
                new_position=false;
                lock.unlock();
                {
                    std::stringstream a;
                    a << "Source_position (" << (int32_t) relative_azimuth
                        << "," << (int32_t) relative_elevation << ")" ;
                    text_function(a.str());
                    //hj=0;
                }
                if (relative_azimuth<0){
                    relative_azimuth+=360;
                }
                {
                    auto v_coefficients = head.filter_coef_fft((int32_t)
                        relative_azimuth, (int32_t) relative_elevation);
                    auto & coefficients = *v_coefficients;
                    memcpy(filter_r_re_coeficients_small.data(), std::get
                        <0>(coefficients).data(), audiofft::AudioFFT::
                        ComplexSize(SAMPLES_FFT_SMALL*2)*sizeof(paSampleF));
                    memcpy(filter_r_im_coeficients_small.data(), std::get
                        <1>(coefficients).data(), audiofft::AudioFFT::
                        ComplexSize(SAMPLES_FFT_SMALL*2)*sizeof(paSampleF));
                    memcpy(filter_l_re_coeficients_small.data(), std::get
                        <2>(coefficients).data(), audiofft::AudioFFT::
                        ComplexSize(SAMPLES_FFT_SMALL*2)*sizeof(paSampleF));
                    memcpy(filter_l_im_coeficients_small.data(), std::get
                        <3>(coefficients).data(), audiofft::AudioFFT::
                        ComplexSize(SAMPLES_FFT_SMALL*2)*sizeof(paSampleF));
                }
```

```
                callbacks.at(0)(filter_r_re_coeficients_large,
                    filter_r_im_coeficients_large);
                callbacks.at(1)(filter_l_re_coeficients_large,
                    filter_l_im_coeficients_large);
                callbacks.at(2)(filter_r_re_coeficients_small,
                    filter_r_im_coeficients_small);
                callbacks.at(3)(filter_l_re_coeficients_small,
                    filter_l_im_coeficients_small);
            }
        }
    }
};
class c_filter{
public:
    c_filter (const c_filter&) = delete;
    using callback_data_2 = std::vector<paSampleF>;
    using callback_f_2 = std::function<void(callback_data_2,callback_data_2
        )>;
    using vector_callback = std::vector<callback_f_2>;
    c_filter(size_t l_size, vector_callback callback_l): coefficients_re(
        l_size), coefficients_im(l_size), size_sample(l_size), callback(
        callback_l){
        //size_t i;
        for(auto & n: coefficients_re){
            n=0.0f;
        }
        for(auto & n: coefficients_im){
            n=0.0f;
        }
        new_coef=0;
        thread = std::thread([this]() { thread_(); });
        thread.detach();
    }
    void put_data(std::vector<paSampleF> re, std::vector<paSampleF> im){
        std::unique_lock<std::mutex> lock(mutex);
        data_re.push(std::move(re));
        data_im.push(std::move(im));
        cv.notify_one();
    }
    void set_coefficients(std::vector<paSampleF> re, std::vector<paSampleF>
        im){
        std::unique_lock<std::mutex> lock(mutex);
        new_coef=1;
        coefficients_re=std::move(re);
        coefficients_im=std::move(im);
    }
private:
```

```
    std::vector<paSampleF> coefficients_re;
    std::vector<paSampleF> coefficients_im;
    bool new_coef;
    std::queue<std::vector<paSampleF>> data_re;
    std::queue<std::vector<paSampleF>> data_im;
    size_t size_sample;
    std::thread thread;
    std::mutex mutex;
    std::condition_variable cv;
    vector_callback callback;
    void thread_(){
        size_t i;
        size_t debug=0;
        std::vector<paSampleF> input_re (0);
        std::vector<paSampleF> input_im (0);
        std::vector<paSampleF> output_re (size_sample);
        std::vector<paSampleF> output_im (size_sample);
        std::vector<paSampleF> coefficients_re_l(coefficients_re.size());
        std::vector<paSampleF> coefficients_im_l(coefficients_im.size());
        while(1) {
            std::unique_lock<std::mutex> lock(mutex);
            cv.wait(lock, [this]() { return ((!data_re.empty()) && (!
               data_im.empty())); });
            input_re = std::move(data_re.front());
            data_re.pop();
            input_im = std::move(data_im.front());
            data_im.pop();
            if (new_coef==1){
                coefficients_im_l=std::move(coefficients_im);
                coefficients_re_l=std::move(coefficients_re);
                new_coef=0;
            }
            lock.unlock();
            for (i = 0; i < size_sample; i++) {
                output_re[i] = input_re[i] * coefficients_re_l[i] -
                   input_im[i] * coefficients_im_l[i];
                output_im[i] = input_re[i] * coefficients_im_l[i] +
                   input_im[i] * coefficients_re_l[i];
            }
            for (auto & n: callback) {
                auto raw_re = output_re;
                auto raw_im = output_im;
                n(std::move(raw_re), std::move(raw_im));
            }
        }
    }
};
```

```
class c_id_assoc{
public:
    using callback_f_0 = std::function<size_t ()>;
    using vector_callback_0 = std::vector<callback_f_0>;
    using callback_data = std::vector<paSampleF>;
    using callback_f = std::function<void(callback_data)>;
    using vector_callback = std::vector<callback_f>;
    using callback_data_3 = std::vector<paSampleF>;
    using callback_f_3 = std::function<void(callback_data_3,callback_data_3
        ,size_t)>;
    using vector_callback_3 = std::vector<callback_f_3>;
    c_id_assoc(callback_f_0 f_register_with, callback_f_3 f_callback):
        function_callback(f_callback), function_register(f_register_with){
    }
    void forward_data(std::vector<paSampleF> data1, std::vector<paSampleF>
        data2){
        function_callback(std::move(data1),std::move(data2),id_);
    }
    void register_with(){
        id_ = function_register();
    }
private:
    size_t id_;
    callback_f_3 function_callback;
    callback_f_0 function_register;
};
class c_two_to_one{
public:
    c_two_to_one (const c_two_to_one&) = delete;
    using callback_data_2 = std::vector<paSampleF>;
    using callback_f_2 = std::function<void(callback_data_2,callback_data_2
        )>;
    using vector_callback = std::vector<callback_f_2>;
    c_two_to_one(callback_f_2 f_callback): function_callback(f_callback){
    }
    void forward_data_r(std::vector<paSampleF> raw_data_r) {
        std::unique_lock<std::mutex> lock(mutex);
        if (!data_l.empty()){
            auto raw_data_l = std::move(data_l.front());
            data_l.pop();
            function_callback(std::move(raw_data_r),std::move(raw_data_l));
            lock.unlock();//TODO VERIFY
        }else{
            data_r.push(raw_data_r);
            lock.unlock();//TODO VERIFY
        }
    }
```

```
    void forward_data_l(std::vector<paSampleF> raw_data_l){
        std::unique_lock<std::mutex> lock(mutex);
        if (!data_r.empty()){
            auto raw_data_r = std::move(data_r.front());
            data_r.pop();
            function_callback(std::move(raw_data_r),std::move(raw_data_l));
            lock.unlock();//TODO VERIFY
        }else{
            data_l.push(raw_data_l);
            lock.unlock();//TODO VERIFY
        }
    }
private:
    callback_f_2 function_callback;
    std::mutex mutex;
    std::queue<std::vector<paSampleF>> data_r;
    std::queue<std::vector<paSampleF>> data_l;
};
class c_utils{
public:
    c_utils (const c_utils&) = delete;
    using callback_data = std::vector<paSampleF>;
    using callback_f = std::function<void(callback_data)>;
    using vector_callback = std::vector<callback_f>;
    using callback_data_2 = std::vector<paSampleF>;
    using callback_f_2 = std::function<void(callback_data_2,callback_data_2
        )>;
    using vector_callback_2 = std::vector<callback_f_2>;
    enum class modes {
        divide_package,
        sum_multiple_channels
    };
    void put_data_divide(std::vector<paSampleF> data_l){
        std::unique_lock<std::mutex> lock(mutex);
        data.push(std::move(data_l));
        cv.notify_one();
    }
    void put_data_sum(std::vector<paSampleF> data_re_l, std::vector<
        paSampleF> data_im_l, size_t channel_id){
        std::unique_lock<std::mutex> lock(mutex);
        deque_im.at(channel_id).push(std::move(data_im_l));
        deque_re.at(channel_id).push(std::move(data_re_l));
        cv.notify_one();
    }
    c_utils(modes mode_l, size_t input_size_l, size_t ratio_l,
        vector_callback callback_l): callback(callback_l){
        sum_current_id=0;
```

```
        divide_ratio=ratio_l;
        input_size=input_size_l;
        if (mode_l==modes::divide_package){
            init_divide();
        }else{
            //std::cout << "de ruim";
            std::cout << "sai 3" << std::endl;
            exit(-3);
        }
    } //vector_callback [2 termos]
    c_utils(modes mode_l, size_t input_size_l, size_t channels_l,
        vector_callback_2 callback_l): callback_2(callback_l){
        sum_current_id=0;
        input_size=input_size_l;
        sum_channels=channels_l;
        if (mode_l==modes::sum_multiple_channels){
            init_sum();
        }else{
            //std::cout << "de ruim";
            std::cout << "sai 4" << std::endl;
            exit(-3);
        }
    }
    size_t generate_id(){
        size_t l_id=sum_current_id++;
        std::cout << "ID GERADO" << l_id << std::endl;
        return l_id;
    }
private:
    vector_callback callback;
    vector_callback_2 callback_2;
    size_t divide_ratio;
    size_t input_size;
    size_t sum_channels;
    std::atomic<size_t> sum_current_id;
    std::queue<std::vector<paSampleF>> data;
    std::thread thread;
    std::mutex mutex;
    std::condition_variable cv;
    std::deque<std::queue<std::vector<paSampleF>> > deque_re;
    std::deque<std::queue<std::vector<paSampleF>> > deque_im;
    paSampleF coef;
    void init_sum(){
        deque_re.resize(sum_channels);
        deque_im.resize(sum_channels);
        coef = 1.0f/((paSampleF) sum_channels);
        thread = std::thread([this]() { thread_sum(); });
```

```
        thread.detach();
    }
    void thread_sum(){
        size_t i,j;
        std::vector<std::vector<paSampleF>> input_im (sum_channels);
        std::vector<std::vector<paSampleF>> input_re (sum_channels);
        for (auto & n: input_im){
            n.resize(input_size);
        }
        for (auto & n: input_re){
            n.resize(input_size);
        }
        std::vector<paSampleF> output_im (input_size);
        std::vector<paSampleF> output_re (input_size);
        while(1) {
            std::unique_lock<std::mutex> lock(mutex);
            cv.wait(lock, [this]() {
                for (auto & n: deque_im){
                    if (n.empty()){
                        return false;
                    }
                }
                return true;
            });
            for (j=0; j < sum_channels; j++){
                input_im.at(j)=deque_im.at(j).front();
                deque_im.at(j).pop();
                input_re.at(j)=deque_re.at(j).front();
                deque_re.at(j).pop();
            }
            lock.unlock();
            for (j=0; j < input_size; j++){
                output_im.at(j)=0;
                output_re.at(j)=0;
                for (auto & n: input_im) {
                    output_im.at(j) += (n.at(j) * coef);
                }
                for (auto & n: input_re) {
                    output_re.at(j) += (n.at(j) * coef);
                }
            }
            for (auto & n: callback_2) {
                auto copy_re=output_re;
                auto copy_im=output_im;
                n(std::move(copy_re),std::move(copy_im));
            }
        }
```

```
    }
    void init_divide(){
        thread = std::thread([this]() { thread_divide(); });
        thread.detach();
    }
    void thread_divide(){
        size_t count = input_size/divide_ratio;
        size_t i,j;
        std::vector<paSampleF> input (0);
        std::vector<paSampleF> output (0);
        while(1) {
            std::unique_lock<std::mutex> lock(mutex);
            cv.wait(lock, [this]() { return (!data.empty()); });
            input = std::move(data.front());
            data.pop();
            lock.unlock();
            for (j=0; j<divide_ratio; j++){
                output.resize(count);
                memcpy(output.data() , input.data() + j * count , count*
                    sizeof(paSampleF));
                for (auto & n: callback){
                    n(std::move(output));
                }
            }
        }
    }
};
class c_pa{
public:
    c_pa (const c_pa&) = delete;
    c_pa(){
        Pa_Initialize();
    }
    ~c_pa(){
        Pa_Terminate();
    }
};
class c_pa_stream{
public:
    c_pa_stream (const c_pa_stream&) = delete;
    using callback_data = std::vector<paSampleF>;
    using callback = std::function<void(callback_data)>;
    using vector_callback = std::vector<callback>;
    enum class modes {
        play,
        record
    };
```

```
    struct play_raw_data{
        size_t * current_buffer_read;
        paSampleF * output [TOTAL_BUFFERS];
    };
    struct record_raw_data{
        size_t * current_buffer_write;
        uint32_t channels;
        paSampleF * input [TOTAL_BUFFERS];
    };
    c_pa_stream(int32_t device, modes mode, int32_t sample_rate_=48000,
       int32_t virtual_channels=2, vector_callback function =
       vector_callback()):
    channels(virtual_channels), vector_function(function), sample_rate(
       sample_rate_), device_id(device), output(0){
        if (mode == modes::play){
            init_play();
        }else if(mode == modes::record){
            init_record();
        }else{
            std::cout << "sai 5" << std::endl;
            exit(-44);
        }
    }
public:
    void put_data(std::vector<paSampleF> right, std::vector<paSampleF> left
       ){ //channel id, data //ONLY FOR THE FCK PLAY
        std::unique_lock<std::mutex> lock(mutex);
        data_queue_r.push(std::move(right));
        data_queue_l.push(std::move(left));
        cv.notify_one();
    }
    void start_stream(){
        Pa_StartStream( stream );
    }
private:
    PaStreamParameters parameters;
    int32_t channels;
    PaStream* stream;
    vector_callback vector_function;
    int32_t sample_rate;
    int32_t device_id;
    play_raw_data data;
    record_raw_data data_r;
    std::queue<std::vector<paSampleF>> data_queue_r;
    std::queue<std::vector<paSampleF>> data_queue_l;
    std::vector<std::vector<paSampleF>> output;
    size_t current_buffer_read;
```

```
size_t current_buffer_write;
std::thread thread;
std::mutex mutex;
std::condition_variable cv;
void init_play(){
    size_t i;
    current_buffer_read=0;
    current_buffer_write=0;
    data.current_buffer_read=&current_buffer_read;
    output.resize(TOTAL_BUFFERS);
    for (auto & n: output){
        n.resize(SAMPLES_FFT_SMALL*2, 0.0f);
    }
    for (i=0; i<TOTAL_BUFFERS; i++){
        data.output[i]=output.at(i).data();
    }
    parameters.device = device_id;
    parameters.channelCount = 2;
    parameters.sampleFormat = paFloat32;
    parameters.suggestedLatency = Pa_GetDeviceInfo( parameters.device )
       ->defaultLowInputLatency;
    parameters.hostApiSpecificStreamInfo = NULL;
    Pa_OpenStream( &stream,
                   NULL,
                   &parameters,
                   sample_rate,
                   //SAMPLES_FFT_LARGE,
                   SAMPLES_FFT_SMALL,
                   paNoFlag,
                   play_callback,
                   &data);
    thread = std::thread([this]() { thread_play(); });
    thread.detach();
}
static int play_callback(const void *input_buffer,
                         void *output_buffer,
                         unsigned long frame_count,
                         const PaStreamCallbackTimeInfo* timeInfo,
                         PaStreamCallbackFlags status_flags,
                         void *user_data ){
    auto * buffer = reinterpret_cast<play_raw_data*>(user_data);
    auto * buffer_output = reinterpret_cast<paSampleF*>(output_buffer);
    auto av = *buffer->current_buffer_read; //todo verify it
    auto * data = reinterpret_cast<paSampleF *>( buffer->output[ av ] )
       ;
    memcpy(buffer_output, data, frame_count*2*sizeof(paSampleF));
    av++;
```

```
        if (av >= TOTAL_BUFFERS){
            av = 0; //bug google compliant
        }
        *buffer->current_buffer_read=av;
        return paContinue;
    }
public:
    bool is_not_empty_play(){
        return ( (!data_queue_l.empty()) && (!data_queue_r.empty()));
    }
private:
    void thread_play(){
        using namespace std::chrono_literals;
        bool continue_;
        size_t debug=0;
        std::vector<paSampleF> output_l (0);
        std::vector<paSampleF> output_r (0);
        size_t i;
        while(1){
            std::unique_lock<std::mutex> lock(mutex);
            cv.wait(lock, [this]() { return ( (!data_queue_l.empty()) && (!
                data_queue_r.empty())); });
            output_l = std::move(data_queue_l.front());
            data_queue_l.pop();
            output_r = std::move(data_queue_r.front());
            data_queue_r.pop();
            lock.unlock();
            paSampleF * raw_samples = output.at(current_buffer_write).data
                ();
            paSampleF * raw_samples_l = output_l.data();
            paSampleF * raw_samples_r = output_r.data();
            for (i=0; i<SAMPLES_FFT_SMALL; i++){
                *(raw_samples+i*2) = *(raw_samples_r+i);
                *(raw_samples+i*2+1) = *(raw_samples_l+i);
            }
            current_buffer_write++;
            if (current_buffer_write >= TOTAL_BUFFERS){
                current_buffer_write=0;
            }
        }
    }
    void thread_record(){
        using namespace std::chrono_literals;
        bool continue_;
        std::vector<std::vector<paSampleF>> output_l (channels);
        for (auto & n: output_l){
            n.resize(SAMPLES_FFT_LARGE, 0.0f);
```

```
        }
        size_t i,j;
        while(1){
            while (current_buffer_read==current_buffer_write){
                std::this_thread::sleep_for(DELAY_LONG);
            }
            paSampleF * raw_samples = output.at(current_buffer_read).data()
                ;
            for (j=0; j<channels; j++){
                auto & n = output_l.at(j);
                auto * raw_store = n.data();
                for (i=0; i<SAMPLES_FFT_LARGE; i++) {
                    *(raw_store+i) = *(raw_samples + i * channels + j);
                }
            }
            for (j=0; j<channels; j++){
                auto & n = output_l.at(j);
                auto & m = vector_function.at(j);
                auto copy = n;
                m(std::move(copy));
            }
            current_buffer_read++;
            if (current_buffer_read >= TOTAL_BUFFERS){
                current_buffer_read=0;
            }
        }
    }
//public:
    static int record_callback(const void *input_buffer,
                               void *output_buffer,
                               unsigned long frame_count,
                               const PaStreamCallbackTimeInfo* timeInfo,
                               PaStreamCallbackFlags status_flags,
                               void *user_data ){
        auto * buffer = reinterpret_cast<record_raw_data*>(user_data);
        const paSampleF* buffer_input = reinterpret_cast<const paSampleF*>(
            input_buffer);
        auto av = *buffer->current_buffer_write; //todo verify it
        paSampleF * data = buffer->input[ av ] ;
        memcpy(data, buffer_input, frame_count * buffer->channels * sizeof(
            paSampleF));
        av++;
        if (av >= TOTAL_BUFFERS){
            av = 0; //bug google compliant
        }
        *buffer->current_buffer_write=av;
        return paContinue;
```

```
    }
//private:
    void init_record(){
        size_t i;
        data_r.current_buffer_write=&current_buffer_write;
        data_r.channels=channels;
        current_buffer_write=0;
        current_buffer_read=0;
        output.resize(TOTAL_BUFFERS); //TODO rever
        for (auto & n: output){
            n.resize(SAMPLES_FFT_LARGE*channels, 0.0f); //TODO rever
        }
        for (i=0; i<TOTAL_BUFFERS; i++){
            data_r.input[i] = output.at(i).data();
        }
        parameters.device=device_id;
        parameters.channelCount=channels;
        parameters.sampleFormat=paFloat32;
        parameters.suggestedLatency=Pa_GetDeviceInfo( parameters.device )->
           defaultLowInputLatency;
        parameters.hostApiSpecificStreamInfo = NULL;
        Pa_OpenStream( &stream,
                       &parameters,
                       NULL,
                       sample_rate,
                       SAMPLES_FFT_LARGE,
                       paNoFlag,
                       record_callback,
                       &data_r);
        thread = std::thread([this]() { thread_record(); });
        thread.detach();
    }
};
class c_fft{
public:
    c_fft (const c_fft&) = delete;
    using callback_data_ifft = std::vector<paSampleF>;
    using callback_ifft = std::function<void(callback_data_ifft)>;
    using vector_callback_ifft = std::vector<callback_ifft>;
    using callback_data_fft = std::vector<paSampleF>;
    using callback_fft = std::function<void(callback_data_fft,
       callback_data_fft)>;
    using vector_callback_fft = std::vector<callback_fft>;
    enum class modes {
        fft,
        ifft
    };
```

```
    c_fft(modes mode_l,
          size_t size_l,
          vector_callback_fft callback_l):
          f_callback_fft(callback_l) {
        size_sample=size_l;
        if (mode_l==modes::fft){
            init_fft();
        }else{
            std::cout<<"deu ruim fft"<<std::endl;
        }
    }
    c_fft(modes mode_l,
          size_t size_l,
          vector_callback_ifft icallback_l):
            f_callback_ifft(icallback_l){
        size_sample=size_l;
        if (mode_l==modes::ifft){
            init_ifft();
        }else{
            std::cout<<"deu ruim fft"<<std::endl;
        }
    }
    void put_data_fft(std::vector<paSampleF> data_l){
        std::unique_lock<std::mutex> lock(mutex);
        data.push(std::move(data_l));
        cv.notify_one();
    }
    void put_data_ifft(std::vector<paSampleF> data_re_l, std::vector<
        paSampleF> data_im_l){
        std::unique_lock<std::mutex> lock(mutex);
        data_re.push(std::move(data_re_l));
        data_im.push(std::move(data_im_l));
        cv.notify_one();
    }
private:
    size_t size_sample;
    std::thread thread;
    std::mutex mutex;
    std::condition_variable cv;
    std::queue<std::vector<paSampleF>> data;
    std::queue<std::vector<paSampleF>> data_re;
    std::queue<std::vector<paSampleF>> data_im;
    vector_callback_fft f_callback_fft;
    vector_callback_ifft f_callback_ifft;
    void init_fft(){
        thread = std::thread([this]() { fft_thread(); });
        thread.detach();
```

```
}
void init_ifft(){
    thread = std::thread([this]() { ifft_thread(); });
    thread.detach();
}
void fft_thread(){
    audiofft::AudioFFT fft;
    fft.init(size_sample*2);
    std::vector<paSampleF> output_re (audiofft::AudioFFT::ComplexSize(
        size_sample*2));
    std::vector<paSampleF> output_im (audiofft::AudioFFT::ComplexSize(
        size_sample*2));
    std::vector<paSampleF> input (0);
    while(1){
        std::unique_lock<std::mutex> lock(mutex);
        cv.wait(lock, [this]() { return ( !data.empty() ); });
        input = std::move(data.front());
        data.pop();
        lock.unlock();
        input.resize(size_sample*2,0.0f);
        fft.fft(input.data(), output_re.data(), output_im.data());
        for (auto & n: f_callback_fft){
            auto copy_re = output_re;
            auto copy_im = output_im;
            n(std::move(copy_re),std::move(copy_im));
        }
    }
}
void ifft_thread(){
    audiofft::AudioFFT fft;
    fft.init(size_sample*2);
    std::vector<paSampleF> output (0);
    std::vector<paSampleF> input_re (0);
    std::vector<paSampleF> input_im (0);
    output.resize(size_sample*2, 0.0f);
    std::vector<paSampleF> saved (size_sample, 0.0f);
    while(1){
        std::unique_lock<std::mutex> lock(mutex);
        cv.wait(lock, [this]() { return ( (!data_re.empty()) && (!
            data_im.empty()) ); });
        input_re = std::move(data_re.front());
        data_re.pop();
        input_im = std::move(data_im.front());
        data_im.pop();
        lock.unlock();
        fft.ifft(output.data(), input_re.data(), input_im.data());
        size_t i;
```

```
            for (i=0;i<size_sample;i++){
                output[i]+=saved[i];
            }
            for (i=0;i<size_sample;i++){
                saved[i]=output[size_sample+i];
            }
            for (auto & n: f_callback_ifft){
                auto copy = output;
                copy.resize(size_sample);
                n(std::move(copy));
            }
        }
    }
};
class c_virtual_channel{
public:
    c_virtual_channel (const c_virtual_channel&) = delete;
    c_virtual_channel(
        std::function<size_t()> callback_r_reg_lo,
        std::function<void( std::vector<paSampleF>, std::vector<paSampleF>,
            size_t)> callback_r_play_lo,
        std::function<size_t()> callback_l_reg_lo,
        std::function<void( std::vector<paSampleF>, std::vector<paSampleF>,
            size_t)> callback_l_play_lo,
        std::function<void(std::string)> text_function_v
    ):
        callback_r_reg(callback_r_reg_lo),
        callback_r_play(callback_r_play_lo),
        callback_l_reg(callback_l_reg_lo),
        callback_l_play(callback_l_play_lo),
        text_function(text_function_v)
    {
        thread = std::thread([this]() { thread_(); });
        thread.detach();
    }
    std::function<void(std::string)> text_function;
private:
    std::thread thread;
    std::function<size_t()> callback_r_reg;
    std::function<void( std::vector<paSampleF>, std::vector<paSampleF>,
        size_t)> callback_r_play;
    std::function<size_t()> callback_l_reg;
    std::function<void( std::vector<paSampleF>, std::vector<paSampleF>,
        size_t)> callback_l_play;
    void thread_(){
        c_id_assoc channel_v1(
                [&](){
```

```
                //std::cout << "a 1"<<std::endl;
                return callback_r_reg();
            },
            [&](auto vector3, auto vector4, auto vector5){
                //std::cout << "a 2"<<std::endl;
                callback_r_play(std::move(vector3),std::move(vector4),
                    vector5);
            });
    c_id_assoc channel_v2(
            [&](){
                //std::cout << "a 3"<<std::endl;
                return callback_l_reg();
            },
            [&](auto vector3, auto vector4, auto vector5){
                //std::cout << "a 4 "<<std::endl;
                callback_l_play(std::move(vector3),std::move(vector4),
                    vector5);
            });
    channel_v1.register_with();
    channel_v2.register_with();//TODO AUTOMATIZAR
    std::vector< std::function < void(std::vector<paSampleF>,std::
        vector<paSampleF>) > > function_filter1_r_sm (1);
    function_filter1_r_sm.at(0) = [&](auto vector1, auto vector2) {
         channel_v1.forward_data(std::move(vector1),std::move(vector2));
    };
    std::vector< std::function < void(std::vector<paSampleF>,std::
        vector<paSampleF>) > > function_filter1_l_sm (1);
    function_filter1_l_sm.at(0) = [&](auto vector1, auto vector2) {
         channel_v2.forward_data(std::move(vector1),std::move(vector2));
    };
    c_filter filter1_r_sm(audiofft::AudioFFT::ComplexSize(
        SAMPLES_FFT_SMALL*2), function_filter1_r_sm);
    c_filter filter1_l_sm(audiofft::AudioFFT::ComplexSize(
        SAMPLES_FFT_SMALL*2), function_filter1_l_sm);
    std::vector< std::function < void(std::vector<paSampleF>,std::
        vector<paSampleF>) > > function_fft_sm_1 (1);
    function_fft_sm_1.at(0) = [&](auto vector1, auto vector2) {
         filter1_r_sm.put_data(std::move(vector1),std::move(vector2));
    };
    std::vector< std::function < void(std::vector<paSampleF>,std::
        vector<paSampleF>) > > function_fft_sm_2 (1);
    function_fft_sm_2.at(0) = [&](auto vector1, auto vector2) {
         filter1_l_sm.put_data(std::move(vector1),std::move(vector2));
    };
    c_fft fft_sm_r_1(c_fft::modes::fft, SAMPLES_FFT_SMALL,
        function_fft_sm_1);
```

```
c_fft fft_sm_l_1(c_fft::modes::fft, SAMPLES_FFT_SMALL,
    function_fft_sm_2);
std::vector< std::function < void(std::vector<paSampleF>) > >
    function_divide_large_to_small_r_1 (1);
function_divide_large_to_small_r_1.at(0) = [&](auto vector1) {
     fft_sm_r_1.put_data_fft(std::move(vector1));
};
std::vector< std::function < void(std::vector<paSampleF>) > >
    function_divide_large_to_small_l_1 (1);
function_divide_large_to_small_l_1.at(0) = [&](auto vector1) {
     fft_sm_l_1.put_data_fft(std::move(vector1));
};
c_utils divide_large_to_small_l_1(c_utils::modes::divide_package,
    SAMPLES_FFT_LARGE, SAMPLES_FFT_LARGE/SAMPLES_FFT_SMALL,
    function_divide_large_to_small_r_1);
c_utils divide_large_to_small_r_1(c_utils::modes::divide_package,
    SAMPLES_FFT_LARGE, SAMPLES_FFT_LARGE/SAMPLES_FFT_SMALL,
    function_divide_large_to_small_l_1);
std::vector< std::function < void(std::vector<paSampleF>) > >
    function_ifft_r_la (1);
function_ifft_r_la.at(0) = [&](auto vector1) {
     divide_large_to_small_l_1.put_data_divide(std::move(vector1));
};
std::vector< std::function < void(std::vector<paSampleF>) > >
    function_ifft_l_la (1);
function_ifft_l_la.at(0) = [&](auto vector1) {
     divide_large_to_small_r_1.put_data_divide(std::move(vector1));
};
c_fft ifft_l_la(c_fft::modes::ifft, SAMPLES_FFT_LARGE,
    function_ifft_l_la);
c_fft ifft_r_la(c_fft::modes::ifft, SAMPLES_FFT_LARGE,
    function_ifft_r_la);
std::vector< std::function < void(std::vector<paSampleF>,std::
    vector<paSampleF>) > > function_filter1_r_la (1);
function_filter1_r_la.at(0) = [&](auto vector1, auto vector2) {
     ifft_r_la.put_data_ifft(std::move(vector1),std::move(vector2));
};
std::vector< std::function < void(std::vector<paSampleF>,std::
    vector<paSampleF>) > > function_filter1_l_la (1);
function_filter1_l_la.at(0) = [&](auto vector1, auto vector2) {
     ifft_l_la.put_data_ifft(std::move(vector1),std::move(vector2));
};
c_filter filter1_r_la(audiofft::AudioFFT::ComplexSize(
    SAMPLES_FFT_LARGE*2), function_filter1_r_la);
c_filter filter1_l_la(audiofft::AudioFFT::ComplexSize(
    SAMPLES_FFT_LARGE*2), function_filter1_l_la);
```

```
std::vector< std::function < void(std::vector<paSampleF>,std::
    vector<paSampleF>) > > function_sofa_gen (4);
function_sofa_gen.at(0) = [&](auto vector1, auto vector2) { //
    rlarge
     filter1_r_la.set_coefficients(std::move(vector1),std::move(
         vector2));
};
function_sofa_gen.at(1) = [&](auto vector1, auto vector2) {//l
    large
     filter1_l_la.set_coefficients(std::move(vector1),std::move(
         vector2));
};
function_sofa_gen.at(2) = [&](auto vector1, auto vector2) {//r
    small
     filter1_r_sm.set_coefficients(std::move(vector1),std::move(
         vector2));
};
function_sofa_gen.at(3) = [&](auto vector1, auto vector2) {//l
    small
     filter1_l_sm.set_coefficients(std::move(vector1),std::move(
         vector2));
};
c_sofa sofa_gen(function_sofa_gen, text_function);
sofa_callback_head= [&](auto vector1, auto vector2, auto vector3,
    auto vector4) {
     sofa_gen.set_head_position(vector1, vector2, vector3, vector4);
};
sofa_callback_source= [&](auto vector1, auto vector2) {
     sofa_gen.set_source_position(vector1,vector2);
};
std::vector< std::function < void(std::vector<paSampleF>,std::
    vector<paSampleF>) > > function_fft_la_1 (2);
function_fft_la_1.at(0) = [&](auto vector1, auto vector2) {
     filter1_r_la.put_data(std::move(vector1),std::move(vector2));
};
function_fft_la_1.at(1) = [&](auto vector1, auto vector2) {
     filter1_l_la.put_data(std::move(vector1),std::move(vector2));
};
c_fft fft_la_1(c_fft::modes::fft, SAMPLES_FFT_LARGE,
    function_fft_la_1);
fft_callback_sendto = [&](auto vector1) {
     fft_la_1.put_data_fft(std::move(vector1));
};
//}
while(1){
     using namespace std::chrono_literals;
     std::this_thread::sleep_for(10s);
```

```
        }
    }
    std::function<void(std::vector<paSampleF>)> fft_callback_sendto;
    std::function<void(int32_t,int32_t)> sofa_callback_source;
    std::function<void(bool,int32_t,int32_t,int32_t)> sofa_callback_head;
public:
    void callback_data(std::vector<paSampleF> vector1){
        fft_callback_sendto(std::move(vector1));
    }
    void callback_sofa_source(int32_t azimuth, int32_t elevation){
        sofa_callback_source(azimuth, elevation);
    }
    void callback_sofa_head(bool w, int32_t x, int32_t y, int32_t z){
        sofa_callback_head(w,x,y,z);
    }
};
class c_serial{
private:
    struct sp_port * port;
    bool is_open;
    bool fake;
    uint32_t fake_helper;
    std::tuple<bool,float,float,float> helper2;
public:
    c_serial(){
        is_open=0;
        fake=0;
        fake_helper=95;
        helper2={0,0,0,0};
    }
    ~c_serial(){
        if (is_open==1) {
            sp_close(port);
        }
    }
    void list_devices(){
        size_t i;
        struct sp_port **ports;
        sp_return error = sp_list_ports(&ports);
        if (error == SP_OK) {
            for (i = 0; ports[i]; i++) {
                printf("%d - Found port: '%s'\n", (int) i,sp_get_port_name(
                    ports[i]));
            }
            sp_free_port_list(ports);
        } else {
            printf("No serial devices detected\n");
```

```
        }
    }
    bool select_device(uint32_t id){
        if (id==99){fake=1;return 1;}
        int error=0;
        std::string device_name;
        struct sp_port **ports;
        error = sp_list_ports(&ports);
        uint32_t i;
        for (i = 0; ports[i]; i++) {
            if (id==i){
                device_name.reserve(strlen(sp_get_port_name(ports[i])));
                memcpy(device_name.data(), sp_get_port_name(ports[i]),
                    strlen(sp_get_port_name(ports[i])));
            }
        }
        sp_free_port_list(ports);
        error = sp_get_port_by_name(device_name.data(),&port);
        if (error == SP_OK) {
            error = sp_open(port,SP_MODE_READ_WRITE);
            if (error == SP_OK) {
                sp_set_baudrate(port, 57600);
                is_open=1;
                return 1;
            }
        }
        return 0;
    }
    bool is_valid(){
        return is_open;
    }
    void receive_until_empty(){
        if (fake==1){return;}
        int bytes_waiting = sp_input_waiting(port);
        int byte_num;
        char byte_buff[512];
        while (bytes_waiting>0) {
            /*byte_num = 0;
            byte_num = */
            sp_nonblocking_read(port, byte_buff, 512);
            {
                using namespace std::chrono_literals;
                std::this_thread::sleep_for(DELAY_LONG * 1000);
            }
            bytes_waiting = sp_input_waiting(port);
        }
    }
```

```
void fail(){
    //std::cout << "Something is wrong"<< std::endl;
}
bool request_data(){
    if (fake==1){return 1;}
    int err;
    err=sp_nonblocking_write(port,"k",1);
    if (err<1){
        fail();
        //std::cout <<"s"<<std::endl;
        return 0;
    }
    return 1;
}
bool set_zero(){
    if (fake==1){return 1;}
    int err;
    err=sp_nonblocking_write(port,"r",1);
    if (err<1){
        fail();
        //std::cout <<"s"<<std::endl;
        return 0;
    }
    return 1;
}
bool wait_for_data(){
    if (fake==1){
        {
            using namespace std::chrono_literals;
            std::this_thread::sleep_for(DELAY_LONG);
        }
        return 1;
    }
    size_t problem=0;
    while (sp_input_waiting(port) < 13){
        {
            using namespace std::chrono_literals;
            std::this_thread::sleep_for(DELAY_LONG *
                A_BLUETOOTH_DELAY_2);
        }
        problem++;
        if (problem>A_BLUETOOTH_FAIL){
            receive_until_empty();
            fail();
            return 0;
        }
    }
```

```
        return 1;
    }
    std::tuple<bool,float,float,float> receive_data(){
        if (fake==1){return fake_generator();}
        uint8_t byte_buff[15];
        int byte_num = sp_nonblocking_read(port, byte_buff, 13);
        if (byte_buff[0] == '&') {
            //auto result = convert(byte_buff[4], byte_buff[3], byte_buff
                [2], byte_buff[1]);
            auto result0 = convert (byte_buff[1]);
            auto result1 = convert (byte_buff[5]);
            auto result2 = convert (byte_buff[9]);
            return {1,result0,result1,result2};
            //std::cout << result << std::endl;
        }else{
            receive_until_empty();
            fail();
            return {0,0,0,0};
        }
    }
private:
    float convert(uint8_t & v3, uint8_t & v2, uint8_t & v1, uint8_t & v0){
        uint32_t data;
        data = (((uint32_t) v3)<<24) | (((uint32_t) v2)<<16) | (((uint32_t)
            v1)<<8) | (((uint32_t) v0));
        return (*(float *) (&data));
    }
    float convert(uint8_t & v){
        uint32_t * data = (uint32_t *) &v;
        return (*(float *) (data));
    }
    std::tuple<bool,float,float,float> fake_generator(){
        fake_helper++;
        if (fake_helper>100) {
            fake_helper = 0;
            std::random_device rd;  //Will be used to obtain a seed for the
                random number engine
            std::mt19937 gen(rd()); //Standard mersenne_twister_engine
                seeded with rd()
            std::uniform_int_distribution<> dis(0, 180);
            float result0 = (float) ((uint32_t) dis(gen)-90) * 1.0f;
            float result1 = (float) ((uint32_t) dis(gen)-90) * 1.0f;
            float result2 = (float) ((uint32_t) dis(gen)-90) * 1.0f;
            helper2 = {1, result0, result1, result2};
        }
        auto copy=helper2;
        return copy;
```

```
    }
};
class c_arduino{
public:
    c_arduino (const c_arduino&) = delete;
    using callback = std::function<void(bool,int32_t,int32_t, int32_t)>;
    using vector_callback = std::vector<callback>;
    c_arduino(int32_t device, vector_callback functions, c_serial &
        v_serial): callbacks (functions), serial(v_serial){
        thread = std::thread([this]() { thread_c(); });
        thread.detach();
    }
private:
    void thread_c(){
        bool new_p;
        elevation=0;
        azimuth=0;
        while(1){
            {
                using namespace std::chrono_literals;
                size_t i=0,j=0,k=0;
                int32_t x_ref=0, y_ref=0, z_ref=0;
                for (auto i=0; i<50; i++){
                    serial.request_data();
                    serial.wait_for_data();
                    auto [w, x, y, z] = serial.receive_data();
                    if (w==0){i--;}
                    if (i==20){
                        x_ref=(int32_t) x;
                        y_ref=(int32_t) y;
                        z_ref=(int32_t) z;
                        std::cout << "ref  (" << x_ref << "," << y_ref << "
                            ," << z_ref << ")" << std::endl;
                        break;
                    }
                    {
                        using namespace std::chrono_literals;
                        std::this_thread::sleep_for(1s);
                    }
                }
                {
                    using namespace std::chrono_literals;
                    std::this_thread::sleep_for(3s);
                }
                int32_t last_x=0, last_y=0, last_z=0;
                while(1){
                    serial.request_data();
```

```
                    serial.wait_for_data();
                    auto [w, x, y, z] = serial.receive_data();
                    if (w==1) {
                        int32_t x_n = (int32_t) x - x_ref;
                        int32_t y_n = (int32_t) y - y_ref;
                        int32_t z_n = (int32_t) z - z_ref;
                        if (!(x_n==last_x && y_n==last_y && z_n==last_z)) {
                            for (auto &v: callbacks) {
                                int32_t x_c = x_n, y_c = -y_n, z_c = z_n;
                                v(w,
                                  z_c,
                                  y_c,
                                  x_c
                                ); //vai para cada fonte virtual
                            }
                        }
                        last_x=x_n;
                        last_y=y_n;
                        last_z=z_n;
                    }
                }
            }
        }
    }
private:
    int32_t azimuth;
    int32_t elevation;
    vector_callback callbacks;
    std::thread thread;
    std::mutex mutex;
    std::condition_variable cv;
    c_serial & serial;
};
class c_config{
public:
    void list_audio_devices(){
        std::cout << std::endl << "Audio Configuration" <<std::endl;
        Pa_Initialize();
        int64_t i;
        int numDevices, defaultDisplayed;
        const   PaDeviceInfo *deviceInfo;
        PaStreamParameters inputParameters, outputParameters;
        PaError err;
        numDevices = Pa_GetDeviceCount();
        std::cout << "Number of devices" << numDevices << std::endl;
        for( i=0; i<numDevices; i++ )
        {
```

```
deviceInfo = Pa_GetDeviceInfo( i );
printf( "\n-> device #%d\n", i );
/* Mark global and API specific default devices */
defaultDisplayed = 0;
if( i == Pa_GetDefaultInputDevice() )
{
    printf( "[ Default Input" );
    defaultDisplayed = 1;
}
else if( i == Pa_GetHostApiInfo( deviceInfo->hostApi )->
   defaultInputDevice )
{
    const PaHostApiInfo *hostInfo = Pa_GetHostApiInfo(
       deviceInfo->hostApi );
    printf( "[ Default %s Input", hostInfo->name );
    defaultDisplayed = 1;
}
if( i == Pa_GetDefaultOutputDevice() )
{
    printf( (defaultDisplayed ? "," : "[") );
    printf( " Default Output" );
    defaultDisplayed = 1;
}
else if( i == Pa_GetHostApiInfo( deviceInfo->hostApi )->
   defaultOutputDevice )
{
    const PaHostApiInfo *hostInfo = Pa_GetHostApiInfo(
       deviceInfo->hostApi );
    printf( (defaultDisplayed ? "," : "[") );
    printf( " Default %s Output", hostInfo->name );
    defaultDisplayed = 1;
}
if( defaultDisplayed )
    printf( " ]\n" );
printf( "Max inputs = %d", deviceInfo->maxInputChannels  );
printf( ", Max outputs = %d\n", deviceInfo->maxOutputChannels
   );
inputParameters.device = i;
inputParameters.channelCount = deviceInfo->maxInputChannels;
inputParameters.sampleFormat = paInt16;
inputParameters.suggestedLatency = 0; /* ignored by
   Pa_IsFormatSupported() */
inputParameters.hostApiSpecificStreamInfo = NULL;
outputParameters.device = i;
outputParameters.channelCount = deviceInfo->maxOutputChannels;
outputParameters.sampleFormat = paInt16;
```

```
        outputParameters.suggestedLatency = 0; /* ignored by
            Pa_IsFormatSupported() */
        outputParameters.hostApiSpecificStreamInfo = NULL;
    }
    Pa_Terminate();
}
std::tuple<int, int> select_audio_device(){ //input, output
    std::cout << std::endl;
    int in,out;
    std::cout << "Type the ID of the input device." << std::endl;
    std::cin >> in;
    std::cout << "Type the ID of the output device." << std::endl;
    std::cin >> out;
    return {in, out};
}
void list_serial_devices(){
    c_serial serial;
    std::cout << std::endl << "Serial Configuration"<<std::endl;
    serial.list_devices();
}
void select_serial_device(c_serial & serial){
    //c_serial serial;
    std::cout << "Type the ID of the head-tracker device" << std::endl;
    uint32_t id;
    std::cin >> id;
    //std::cout << " " << serial.is_valid() << std::endl; //0
    if ((serial.select_device(id))==0){
        std::cout << "sai 6" << std::endl;
        exit(-1);
    }
    {
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(DELAY_LONG * 100);
    }
    //std::cout << "FOI" << std::endl;
    //std::cout << " " << serial.is_valid() << std::endl; //1
    serial.receive_until_empty();
    serial.request_data();
    {
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(DELAY_LONG * 1000);
    }
    //std::cout << " " << serial.is_valid() << std::endl; //1
    serial.receive_until_empty();
}
std::tuple<int32_t, std::vector<source_position>> sources(){
    std::cout << "Type the number of sources"<< std::endl;
```

```
        int32_t numberof;
        int number;
        std::cin >> number;
        numberof=(int32_t)number;
        std::vector<source_position> positions (numberof);
        int32_t i=0;
        for (auto & a: positions){
            std::cout << "Type the azimuth for the source " << (int)(++i)
               <<std::endl;
            std::cin >> a.azimuth;
            std::cout << "Type the elevation for the source " << (int)(i)
               <<std::endl;
            std::cin >> a.elevation;
        }
        return {numberof, std::move(positions)};
    };
};
void play_new(int32_t virtual_channels, int32_t play_device_id, int32_t
    record_device_id, c_serial & serial, std::vector<source_position>
    sources_position){
    c_pa start_audio_tools;
    c_pa_stream play(play_device_id, c_pa_stream::modes::play, 48000,
        virtual_channels);
    c_two_to_one forwarder_2_to_1(
            [&](auto vector1, auto vector2) {
                play.put_data(std::move(vector1),std::move(vector2));
            }
    );
    //duas ultimas iffts
    std::vector< std::function < void(std::vector<paSampleF>) > >
        function_ifft_r (1);
    function_ifft_r.at(0) = [&](auto vector1) {
        forwarder_2_to_1.forward_data_r(std::move(vector1));
    };
    std::vector< std::function < void(std::vector<paSampleF>) > >
        function_ifft_l (1);
    function_ifft_l.at(0) = [&](auto vector1) {
        forwarder_2_to_1.forward_data_l(std::move(vector1));
    };
    c_fft ifft_l_sm(c_fft::modes::ifft, SAMPLES_FFT_SMALL, function_ifft_l)
        ;
    c_fft ifft_r_sm(c_fft::modes::ifft, SAMPLES_FFT_SMALL, function_ifft_r)
        ;
    std::vector< std::function < void(std::vector<paSampleF>,std::vector<
        paSampleF>) > > function_final_sum_l (1);
    function_final_sum_l.at(0) = [&](auto vector1, auto vector2) {
        ifft_l_sm.put_data_ifft(std::move(vector1),std::move(vector2));
```

```
};
std::vector< std::function < void(std::vector<paSampleF>,std::vector<
    paSampleF>) > > function_final_sum_r (1);
function_final_sum_r.at(0) = [&](auto vector1, auto vector2) {
     ifft_r_sm.put_data_ifft(std::move(vector1),std::move(vector2));
};
c_utils final_sum_l(c_utils::modes::sum_multiple_channels, audiofft::
    AudioFFT::ComplexSize(SAMPLES_FFT_SMALL*2), virtual_channels,
    function_final_sum_l);
c_utils final_sum_r(c_utils::modes::sum_multiple_channels, audiofft::
    AudioFFT::ComplexSize(SAMPLES_FFT_SMALL*2), virtual_channels,
    function_final_sum_r);
c_text_flux text;
std::vector<c_virtual_channel *> virtual_ch;
virtual_ch.reserve(virtual_channels);
//c_virtual_channel * virtual_ch[virtual_channels];
for (auto i=0; i<virtual_channels; i++){
     auto * virtual_v = new c_virtual_channel  (
               [&](){
                    return final_sum_r.generate_id();
               },
               [&](auto vector3, auto vector4, auto vector5){
                    final_sum_r.put_data_sum(std::move(vector3),std::move(
                        vector4),vector5);
               },
               [&](){
                    return final_sum_l.generate_id();
               },
               [&](auto vector3, auto vector4, auto vector5){
                    final_sum_l.put_data_sum(std::move(vector3),std::move(
                        vector4),vector5);
               },
               [&](auto vector3){
                    text.write_to_screen(std::move(vector3));
               }
     );
     virtual_ch.push_back(virtual_v);
}
std::vector< std::function < void(std::vector<paSampleF>) > >
    function_record (virtual_channels);
for (auto i=0; i<virtual_channels; i++){
     auto & x = *virtual_ch.at(i);
     function_record.at(i) = [&](auto vector1) {
          x.callback_data(std::move(vector1));
     };
}
```

```
    c_pa_stream record(record_device_id, c_pa_stream::modes::record, 48000,
         virtual_channels, function_record);
    std::vector< std::function < void(bool,int32_t,int32_t,int32_t) > >
        function_external_head_tracker (virtual_channels);
    for (auto i=0; i<virtual_channels; i++){
        auto & x = *virtual_ch.at(i);
        function_external_head_tracker.at(i) = [&](auto vector1, auto
            vector2, auto vector3, auto vector4) {
            x.callback_sofa_head(vector1,vector2,vector3,vector4);
        };
    }
    for (auto i=0; i<virtual_channels; i++){
        virtual_ch.at(i)->callback_sofa_source(sources_position.at(i).
            azimuth,sources_position.at(i).elevation);
        virtual_ch.at(i)->callback_sofa_head(1,0,0,0);
    }
    c_arduino external_head_tracker(0,function_external_head_tracker,serial
        );
    record.start_stream();
    {
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(DELAY_SHORT*1);
    }
    while (!play.is_not_empty_play()){
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(DELAY_SHORT*1);
    }
    play.start_stream();
    while(1){
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(10s);
    }
}
void default_prog(){
    c_config config;
    config.list_audio_devices();
    auto [audio_input,audio_output] = config.select_audio_device();
    config.list_serial_devices();
    c_serial serial;
    config.select_serial_device(serial);
    auto [numberof, source_position_] = config.sources();
    play_new(numberof,audio_output,audio_input, serial, std::move(
        source_position_));
    while(1){
        {
            using namespace std::chrono_literals;
            std::this_thread::sleep_for(20s);
```

```
        }
    }
    std::cout << "sai 1" << std::endl;
    exit(0);
}
int main() {
    default_prog();
    return 0;
}
```

# Apêndices

# APÊNDICE A – Uso da API SOFA

Nessa seção, estão descritos alguns fatores importantes para o uso da API SOFA em MATLAB.

- *SOFAstart.m*: essa função inicializa a API. Em outras palavras, antes de utilizar qualquer função da API, deve-se adicionar a própria no *path* do MATLAB e chamar SOFAstart.
- *SOFAload.m*: essa função faz o carregamento propriamente dito do arquivo SOFA. Essa função retorna uma variável (do tipo matriz) com todos os dados disponíveis, na qual inclui-se dados como taxa de amostragem, e também todas as funções de transferências disponíveis. As funções que utilizei eram do tipo HRTF, e, portanto, numa sub-matriz estavam disponíveis duas informações de ângulo (azimute e elevação), e a distância que estava a fonte sonora quando cada uma das respostas foi obtida. E numa outra sub-matriz, os coeficientes das funções de transferência estavam disponíveis. Pede uma entrada que é o caminho do arquivo SOFA no sistema. Retorna, como citado, uma matriz.

# APÊNDICE B – Algoritmos para som 3D

As funções listadas abaixo foram escritas em um trabalho anterior com a finalidade de facilitar o acesso os arquivos *SOFA*, criadas para uso interno no Laboratório de Processamento de Sinais e Imagem da UFRGS.

- *espectro.m*: plota o gráfico no domínio frequência da HRTF, juntamente com a fase. Utilizando um parâmetro opcional, pode-se fazer com que sejam plotadas novamente função de transferência, porém dessa vez, juntamente com o atraso de grupo do sinal.
- *jl_sofa_load.m*: faz o carregamento, de forma similar à SOFAload, do arquivo de função de transferência da cabeça. O diferencial é que essa função já tem internamente uma lista de bibliotecas de HRTF e permite a seleção dessa função de transferência mais facilmente. Pede como entrada a identificação do banco de dados e qual o modelo de cabeça (as HRTF foram obtidas com diversos modelos – de bonecos ou de pessoas voluntárias). Retorna a matriz citada na SOFAload.
- *jl_sofa_hrir_i.m*: carrega a HRTF (na verdade, ela no tempo, portanto HRIR: Head Related Impulse Response, resposta ao impulso relacionada à cabeça). Pede como entrada a matriz obtida pela função de carregamento (jl_sofa_load), os dois ângulos de posicionamento da função (azimute e elevação), e a distância que a função foi obtida. Se não houver correspondência exata, essa função faz uma aproximação linear ponderada pela distância as funções disponíveis mais próximas. Retorna os coeficientes da HRIR para os dois ouvidos em uma matriz, geralmente de 200 ou 250 amostras.
- *jl_sofa_hrir.m*: similar à jl_sofa_hrir_i, porém sem a aproximação linear: se não houver função de transferência correspondente a solicitada, retornará a posição da função mais próxima.
- *jl_fir.m*: filtra um arquivo de áudio por um filtro. Pede como entrada os coeficientes do filtro, e uma matriz com o arquivo de áudio (um canal deste, na verdade). Retorna a matriz filtrada.
- *jl_hrtf_view.m*: chama a função espectro. É um atalho para a mesma.
- *jl_sofa_44_16.m*: converte um arquivo SOFA com HRIRs amostrado em 44kHz para 16kHz. Pede como entrada a matriz gerada pelo jl_sofa_load e retorna uma matriz de mesma estrutura, porém com os dados convertidos.

- *jl_sofa_44_48.m*: similar à jl_sofa_44_16, essa função converte arquivos com taxa de amostragem de 44kHz para 48kHz.
- *jl_sofa_48_16.m*: novamente similar à jl_sofa_48_16, essa função faz a conversão das amostras de 48kHz para 16kHz.