MATHEUS MÁLTARO PERPÉTUO

ÁUDIO DSP UFLA:

SUÍTE DE PLUG-INS DE ÁUDIO EM C++ / JUCE

1^{a} edição

LAVRAS – MG

2021

MATHEUS MÁLTARO PERPÉTUO

ÁUDIO DSP UFLA:

SUÍTE DE PLUG-INS DE ÁUDIO EM C++ / JUCE

1^{a} edição

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências da Graduação em Engenharia de Controle e Automação, para a obtenção do título de Bacharel.

Prof. Dr. Thomaz Chaves de Andrade Oliveira

Orientador

LAVRAS – MG

2021

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos da Biblioteca Universitária da UFLA, com dados informados pelo próprio autor.

Perpétuo, Matheus Máltaro
Áudio DSP UFLA : Suíte de Plug-ins de Áudio em C++ / JUCE / Matheus Máltaro Perpétuo. 1^a ed. – Lavras : UFLA, 2021.
193 p. : il.

TCC(Graduação)–Universidade Federal de Lavras, 2021.
Orientador: Prof. Dr. Thomaz Chaves de Andrade Oliveira.
Bibliografia.

1. DSP. 2. Áudio. 3. Plug-ins. I. Oliveira, Thomaz Chaves de Andrade. II. Título.

MATHEUS MÁLTARO PERPÉTUO

ÁUDIO DSP UFLA: SUÍTE DE PLUG-INS DE ÁUDIO EM C++ / JUCE
AUDIO DSP UFLA: C++ / JUCE AUDIO PLUG-IN SUITE

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências da Graduação em Engenharia de Controle e Automação, para a obtenção do título de Bacharel.

APROVADA em XX de Novembro de 2021.

Prof. Dr. Bruno de Abreu Silva UFLA
Nelton Vinícius Matioli Santos UFLA

Prof. Dr. Thomaz Chaves de Andrade Oliveira
Orientador

LAVRAS – MG
2021

Dedico este trabalho aos meus pais, família, amigos e à Isabela, que me incentivaram e acompanharam nessa jornada.

AGRADECIMENTOS

Agradeço a Deus, pela minha vida, saúde e força para alcançar os meus objetivos. Aos meus pais, por sempre priorizarem a minha formação e por todo apoio durante todo esse processo. Aos meus avós, de sangue e de coração, pela confiança depositada e pelo suporte dado em fases importantes da minha vida. À minha família, por terem sido minha base forte nos momentos necessários. Aos meus amigos, por estarem presentes nos melhores e piores momentos, trazendo grandes aprendizados para minha vida. À Isabela, por todo amor, companheirismo e cumplicidade. À minha prima Giuvane, pelo suporte financeiro nos últimos anos da graduação. À República Cabaré, por esses cinco anos de grande aprendizado e pelas amizades construídas. Ao meu orientador Thomaz, pelo apoio e pelas conversas durante a realização deste trabalho. Ao Bruno e Nelton, por aceitarem o convite de participarem da banca deste trabalho. À Universidade Federal de Lavras, por todo o aprendizado acadêmico e para a vida.

Sem música, a vida seria um erro.
(Friedrich Nietzsche)

RESUMO

O avanço da computação fez com que os computadores se tornassem ferramentas presentes nos estúdios de música. Uma das tecnologias mais usadas para produção musical são os *plug-ins* de áudio, que são processadores digitais de sinal. Este trabalho teve o objetivo de desenvolver *plug-ins* de áudio utilizando o *framework* JUCE, os *plug-ins* implementados foram um equalizador, um *waveshaper*, um reverberador e um *delay*, com sucesso. Em especial para o equalizador, foi feita uma análise matemática no domínio da frequência para definição dos parâmetros. Para os outros *plug-ins*, foi feita uma revisão bibliográfica para tratar dos assuntos teóricos referentes aos fenômenos sonoros e à ação desses processadores nos sinais de entrada. Os *plug-ins* desenvolvidos foram enviados para produtores musicais para validação como ferramenta de trabalho. Os resultados da pesquisa mostraram que os *plug-ins* são passíveis de serem usados profissionalmente, após ajustes solicitados pelos produtores. O desenvolvimento dessas ferramentas se justifica por possibilitar um custo mais acessível a elas.

Palavras-chave: DSP, Áudio, Engenharia, Plug-ins, JUCE, C++, Filtros, Processamento Linear, Processamento Não Linear

ABSTRACT

The advancement of computing made computers become tools present in music studios. One of the most used technologies for music production is audio plug-ins, which are digital signal processors. This work had the objective of developing audio plug-ins using the JUCE framework, the implemented plug-ins were an equalizer, a waveshaper, a reverberator and a delay, with success. In particular for the equalizer, a mathematical analysis was carried out in the frequency domain to define the parameters. For the other plug-ins, a literature review was carried out to deal with theoretical issues related to sound phenomena and the action of these processors on input signals. The developed plug-ins were sent to music producers for validation as a working tool. The survey results showed that the plug-ins are likely to be used professionally, after adjustments requested by the producers. The development of these tools is justified by making them more affordable.

Keywords: DSP, Audio, Engineering, Plug-ins, JUCE, C++, Filters, Linear Processing, Non Linear Processing

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A música acompanha os seres humanos há muito tempo, ela é conhecida e praticada desde a pré-história e, "juntamente com os costumes, hábitos e expressões artísticas, constitui a manifestação cultural de um povo, que forma-se a partir de processos históricos e sociais"(MATIAS, 2020).

Até a década de 1970, a gravação e reprodução de música era feita somente de forma analógica (MITCHELL; SANTIAGO, 2016). As formas de onda eram armazenadas da maneira mais fiel possível em um meio físico (vinil) ou magnético (fita), mas a partir desse momento, com o advento e avanço da computação, se inicializou a digitalização do áudio. Com evolução dos *hardwares* e *software*, começaram a surgir equipamentos e algoritmos capazes de processar o áudio na forma digital. O desenvolvimento dos algoritmos básicos para o processamento de sinais digitais se deu a partir da década de 1970, causando uma mudança fundamental na maneira em que eram produzidas as obras fonográficas, ou os registros de música.

No final da década de 1970 e início da década de 1980, os computadores aos poucos foram se tornando ferramentas presentes nos estúdios de música. Um exemplo é o da empresa Yamaha, que foi uma das primeiras a realizar grandes investimentos no desenvolvimento de ferramentas digitais, como a Interface Digital de Instrumentos Musicais (MIDI - *Musical Instrument Digital Interface*), um protocolo de comunicação digital por comandos, que ainda hoje é utilizado para realizar a comunicação entre instrumentos digitais e computadores. A Yamaha também foi pioneira no uso de computadores para gerar sequências automatizadas, como linhas de bateria ou de teclado armazenados em MIDI. Na década de 1980, havia computadores que podiam realizar a comunicação com esses instrumentos, para que a programação das automações fossem realizadas por meio do computador (MIDI, 2021).

Com essa evolução, foram surgindo outros equipamentos, como *racks* de efeitos digitais e pedaleiras digitais. No início da década de 1990, surgiu o *software* Pro Tools, que possibilitou a utilização de computadores para realizar gravação e pós-produção musical (PRO. . . , 2021). Hoje, a produção musical em quase toda a sua totalidade é realizada por meio de computadores e as ferramentas disponíveis não param de evoluir. O que antes só era possível de realizar com *racks* repletos de equipamentos, hoje é possível de ser feito utilizando apenas um notebook ou até mesmo um *smartphone*. Uma das tecnologias mais disseminadas no mercado de produção musical hoje são os *plug-ins* de processamento de áudio, processadores digitais de sinal que possibilitam realizar as mais variadas modificações nos sinais a serem processados.

Assim como toda mudança, também tiveram pessoas contrárias às transformações dos meios de produção musical. Ainda hoje temos pessoas do ramo musical que preferem utilizar equipamentos analógicos em todas as etapas de gravação e pós-produção musical, em vez dos equipamentos digitais. Outros, preferem a utilização de LPs em vez de mídias digitais. Entretanto, é difícil negar o quanto a transformação digital na música facilitou e reduziu os custos das produções.

Este trabalho teve por objetivo principal o desenvolvimento de ferramentas voltadas à produção digital de música, buscando uma interação com os profissionais da área para direcionar melhor os esforços e desenvolver aplicações que se tornem cada vez mais úteis para esse mercado. Esse desenvolvimento se justifica por tornar o custo dessas ferramentas mais acessível e permitir uma melhor personalização, a partir do contato feito com esses profissionais. Os objetivos específicos foram: abordar a teoria básica e implementar quatro *plug-ins* simples de processamento de áudio (um equalizador, um *waveshaper*, um reverberador e um *delay*), que foram escolhidos para provar a possibilidade de criação de uma grande variadade de *plug-ins* adotando a mesma metodologia utilizada, realizar testes nesses *plug-ins* para verificar o correto funcionamento e coletar *feedbacks* de profissionais do mercado de áudio que realizaram testes utilizando essas ferramentas.

1.1 Organização desta monografia

O restante desta monografia está organizado da seguinte maneira: no Capítulo 2 são apresentados os conceitos necessários para o desenvolvimento do trabalho, no Capítulo 3 são apresentados os materiais e métodos utilizados para implementação dos *plug-ins* e realização dos testes, no Capítulo 4 são apresentados os resultados obtidos e discussões pertinentes sobre esses resultados e no Capítulo 5 são apresentadas as conclusões sobre o trabalho.

2 REFERENCIAL TEÓRICO

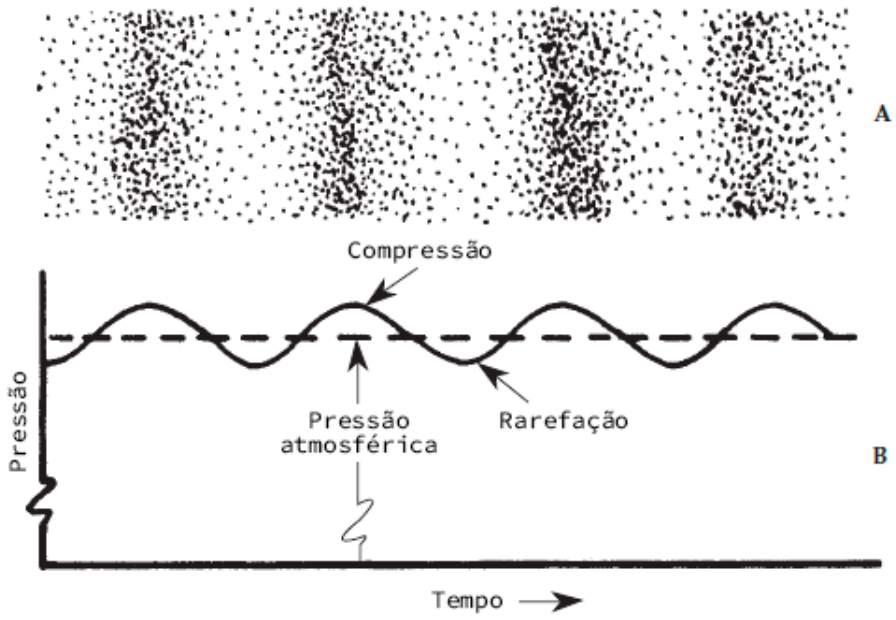
2.1 O som

O som é uma onda mecânica longitudinal, que se propaga em todas as direções e em qualquer meio material (o som não se propaga no vácuo).

Uma onda é produzida quando um meio é perturbado. O meio pode ser ar, água, aço, terra, etc. A perturbação produz uma flutuação na condição ambiente do meio que se propaga como uma onda que irradia para fora da fonte da perturbação (BALLOU et al., 2008).

Essa flutuação é uma série de compressões e rarefações do meio, que é sucessivamente transmitida pela matéria. Na Figura 2.1, vemos uma representação das partículas de ar sob efeito de compressão e refração e a representação desse fenômeno em formato gráfico de onda longitudinal.

Figura 2.1 – Variações de pressão da onda sonora. (A) No meio em que ela se encontra e (B) em representação gráfica



Fonte: Everest e Pohlmann (2009, p. 6).

As principais características que nos permitem diferenciar um som de qualquer outro são a intensidade, a altura e o timbre. A intensidade se refere à quantidade de energia que uma onda sonora transmite, sua amplitude. Quando aumentamos o volume do rádio ou da televisão, por exemplo, estamos mudando a intensidade do som.

Segundo Everest e Pohlmann (2009, p. 21), a medida de intensidade é adimensional, mas atribuímos a ela a unidade de um bel (de Alexander Graham Bell). Para tornar o intervalo

mais fácil de usar, geralmente expressamos os valores em decibéis. Um decibel equivale a $1/10bel$. Um decibel (dB) é 10 vezes o logaritmo à base 10 da razão de duas grandezas de intensidade (ou potência).

A equação 2.1 mostra a relação de ganho K em decibéis entre uma potência P e outra potência de referência P_{ref}.

$$K(dB) = 10 \times log \frac{P}{P_{ref}} \tag{2.1}$$

Em suma, o decibel é uma forma de expressar a intensidade de uma forma que seja relevante para a percepção humana de volume (BALLOU et al., 2008, p. 23). A cada variação de 3dB para cima ou para baixo, a percepção da intensidade do som pelos seres humanos dobra ou cai pela metade. O cérebro relativiza e se adapta à intensidade dos sons que estão presentes no ambiente; em lugares silenciosos, por exemplo, é possível ouvir sons com intensidade muito menor que em lugares com muito barulho.

A altura é a frequência fundamental (conceito explicado na Seção 2.2.3) na qual o som se propaga, a quantidade de ciclos de onda por segundo, sendo Hz (hertz) a unidade de medida; essa caraterística nos permite distinguir um som mais agudo (alto) de um som mais grave (baixo).

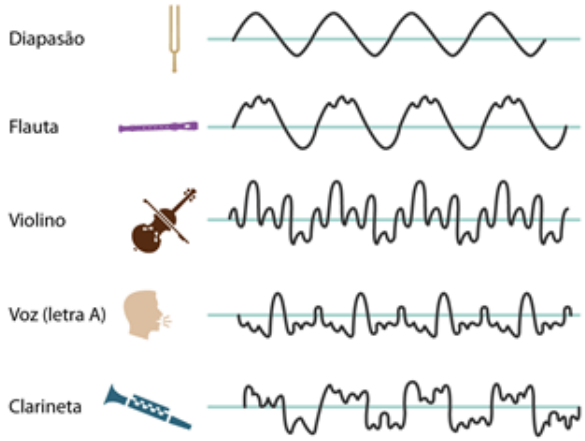
O escopo comumente aceito do espectro audível é de 20 Hz a 20 kHz; sendo o alcance uma das características específicas do ouvido humano. Há sons com frequência muito baixa, chamados infrassons, e sons com frequência muito alta, chamados ultrassons, para serem ouvidos pelo ouvido humano. (EVEREST; POHLMANN, 2009, p. 14)

O timbre remete à natureza do som, ele é caracterizado pelo restante das frequências, além da fundamental, que compõem o som. O timbre é o que nos permite distinguir um som de intensidade e altura iguais produzidos por fontes diferentes, como um violão e um piano, por exemplo. Na Figura 2.2, pode ser vista a diferença entre as formas de onda de sons de mesma altura e timbres diferentes.

2.1.1 A gravação e a reprodução do som

A gravação e a reprodução sonora são realizadas utilizando-se de equipamentos mecânicos ou eletromecânicos. No processo de gravação, as vibrações produzidas no ar pela fonte sonora são registradas como informação. Na reprodução, as informações que foram previamente registradas são convertidas novamente em ondas sonoras.

Figura 2.2 – Sons de alturas iguais e timbres diferentes



Fonte: (DENIS, 2019)

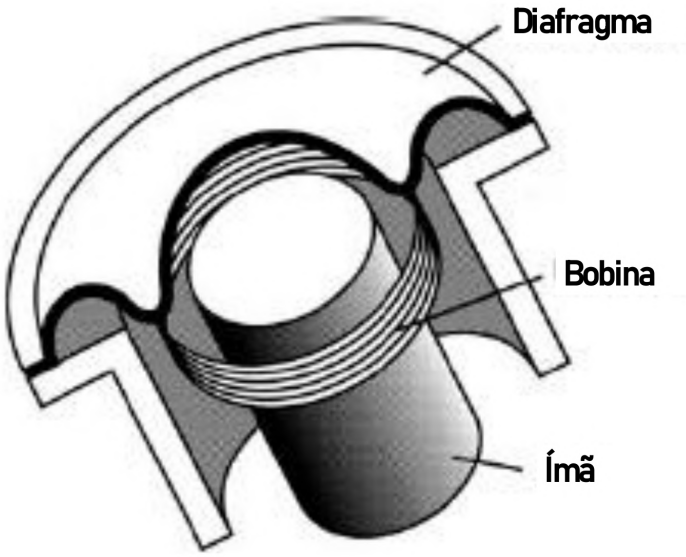
> A história da gravação e reprodução de som é recente. O primeiro protótipo do fonógrafo foi obtido pelo francês Léon Scott em 1857, quando estudava as características do som. Somente vinte anos depois, no entanto, graças a uma máquina inventada por Thomas Alva Edison, foi possível ouvir a reprodução de uma gravação. No século XX, desenvolveram-se muito as técnicas de gravação e reprodução acústica, o que resultou numa série de aparelhos domésticos destinados ao lazer (MITCHELL; SANTIAGO, 2016).

Para converter as vibrações sonoras em sinais elétricos, utiliza-se o microfone. Ele é um equipamento transdutor mecânico/elétrico, sensível às variações de pressão causadas pelo som, por meio de um dispositivo chamado diafragma. O diafragma fica acoplado a um ímã e este ímã é envolto por uma bobina condutora (solenoide); quando a variação de pressão do ar faz com que o conjunto diafragma e ímã se movimente, é gerada uma corrente elétrica na bobina e por consequência uma tensão nos terminais dessa bobina, devido à interação entre o ímã e a solenoide, seguindo o princípio da indução eletromagnética descoberto por Michael Faraday. Na Figura 2.3 temos um desenho que mostra os dispositivos internos de um microfone do tipo dinâmico.

Para o processo de conversão inverso é utilizado um outro tipo de transdutor, o alto falante, ele transforma o sinal elétrico em movimento mecânico, o que faz dele uma fonte sonora. Na Figura 2.4 temos um desenho mostrando os dispositivos internos de um alto falante: cone, terminal de conexão, aranha, bobina, suporte da bobina, peça de pólo e ímã.

As técnicas de gravação e reprodução do som evoluíram muito com o passar dos anos, a indústria fonográfica já foi dominada por discos de vinil, fitas cassete, *Compact Discs* (CDs) e dispositivos de memória *flash*; atualmente o mercado se reinventa mais uma vez com os serviços de *streaming*. Essas transformações tornaram cada vez mais prática e acessível a disponibiliza-

Figura 2.3 – Desenho Simples de um Transdutor de um Microfone do Tipo Dinâmico



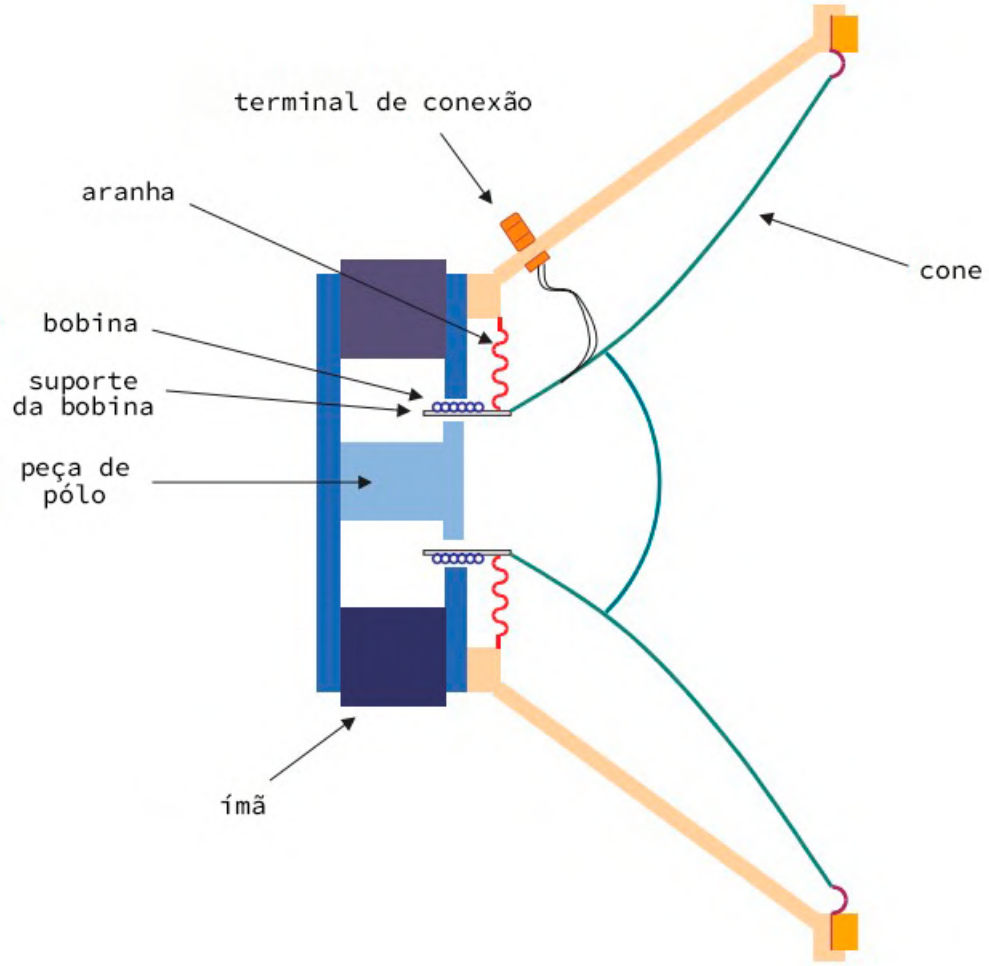
Legenda: A parte interna de um microfone dinâmico é composta por três partes: o diafragma, um ímã e uma bobina.

Fonte: Ballou et al. (2008, p. 504)

ção de produtos fonográficos, fazendo que seja cada vez mais fácil para que uma pessoa com conhecimento e o mínimo de equipamento consiga disponibilizar ao público músicas, *podcasts* ou qualquer outro tipo de material sonoro.

Em cada uma dessas mídias, o som está representado de formas diferentes. Nos discos de vinil as ranhuras desenhadas são os registros das ondas sonoras, elas fazem a agulha vibrar e essa vibração é transformada em sinal elétrico. Nas fitas cassetes o áudio é gravado de forma sequencial em uma camada composta por minúsculas partículas magnéticas, gerando regiões magnetizadas; essas regiões podem, posteriormente, ser lidas por um sensor e transformadas em sinais elétricos. Nos CDs as informações são gravadas na forma de bits em uma espiral, por meio de depressões e planos que fazem o registro digital da informação; essas informações podem ser recuperadas utilizando-se de um leitor a *laser*. Dispositivos de memória *flash* armazenam informações por meio de transistores elétricos, que podem ser lidos por microprocessadores. Os serviços de *streaming* armazenam seus dados em servidores, esses dados, geralmente processados por algoritmos que realizam a compressão dos arquivos para redução de tamanho, são enviados por meio de protocolos de internet para que possam ser reproduzidos nos dispositivos dos usuários.

Figura 2.4 – Desenho de um Alto Falante Eletrodinâmico



Fonte: Adam (2002)

2.1.2 O áudio digital

O áudio digital é simplesmente um meio alternativo de transportar informações de áudio, permitindo que computadores consigam trabalhar com elas. Um gravador de áudio digital ideal tem as mesmas características que um gravador analógico ideal.

A maioria dos sinais de interesse prático surge como sinais de tempo contínuo. No entanto, o uso da tecnologia de processamento de sinal digital requer uma representação de sinal em tempo discreto. Isso geralmente é feito por amostragem de um sinal de tempo contínuo em pontos isolados e igualmente espaçados no tempo (amostragem periódica).

> Historicamente, sistemas em tempo discreto têm sido realizados através de computadores digitais, nos quais sinais contínuos no tempo são processados por amostras digitalizadas ao invés de amostras não quantizadas. Portanto, os termos filtro digital e sistema em tempo discreto são usados como sinônimos na literatura. (LATHI, 2008, p. 243)

Os dois processos de transformação de um sinal analógico contínuo em um sinal digital são chamados de amostragem e quantização, eles resultam em uma aproximação da informação real, o que acaba por restringir a quantidade de informação que um sinal digital pode conter. Logo, sempre existe erro atribuído, mas a magnitude deste erro pode ser minimizada até que ele se torne insignificante. Os processos que permitem aos computadores interagir com esses sinais são a conversão analógico-digital (ADC – *Analog Digital Conversion*) e a conversão digital-analógico (DAC – *Digital Analog Conversion*).

Segundo Zölzer et al. (2011, p. 20), a amostragem das amplitudes do sinal analógico $x(t)$ é realizada pelo ADC em uma grade equidistante ao longo do eixo do tempo horizontal; a quantização das amplitudes é representada por números $x[n]$ ao longo do eixo da amplitude vertical. O sinal digital (tempo discreto e amplitude quantizada) é uma sequência de amostras $x[n]$ representadas por números sobre o índice de tempo discreto n. A distância de tempo entre duas amostras consecutivas é denominada intervalo de amostragem T (período de amostragem) e a frequência de amostragem é $1/T$. A relação entre o período e a frequência de amostragem pode ser vista na Equação 2.2.

$$f_s = \frac{1}{T_s} \tag{2.2}$$

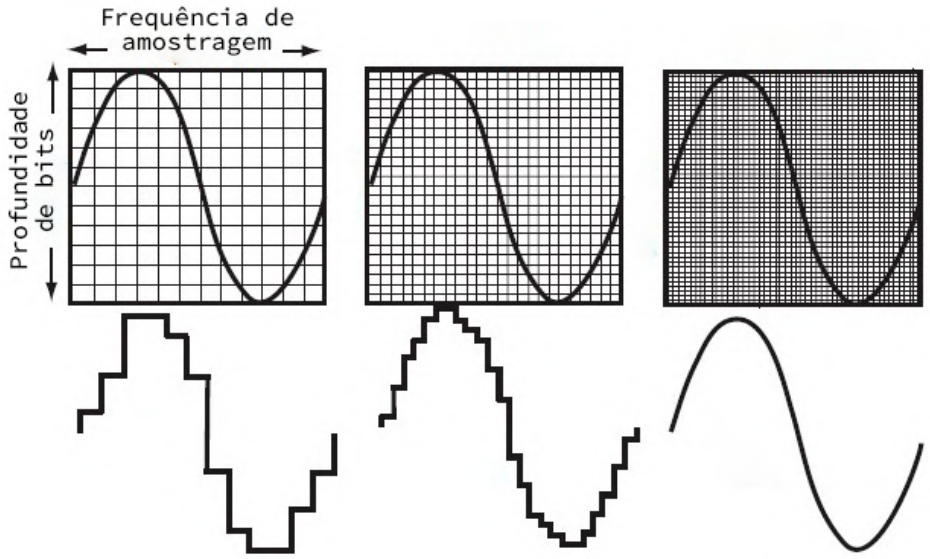
Ainda segundo Zölzer et al. (2011, p. 21), de acordo com o teorema da amostragem, essa frequência deve ser escolhida com valor duas vezes maior que o da frequência mais alta contida no sinal analógico, como mostra a equação 2.3; essa relação é baseada no teorema de Nyquist. Como o ouvido humano não detecta sons acima de 20kHz, a indústria fonográfica tem utilizado os padrões de 44.1kHz, 48kHz, 96kHz e até mesmo valores superiores a estes para a frequência de amostragem.

$$f_s \geq 2 \times f_0 \tag{2.3}$$

Além da frequência de amostragem, a outra propriedade de um sinal digital é a profundidade de bit, que determina o tamanho da informação em código binário utilizada para cada amostra do sinal original; é o número de valores possíveis que uma amostra pode assumir após a quantização. Já foram usados diversos padrões durantes os anos, como o de 16 bits em CDs, atualmente tem-se adotado como padrão da indústria uma quantização com 24 bits de profundidade, com esse padrão uma amostra pode assumir 16.777.216 valores diferentes.

A sucessão de amostras em um sistema digital é análoga à forma de onda original. A Figura 2.5 exemplifica a digitalização de um sinal analógico e mostra como a qualidade final do sinal é afetada pela variação dos parâmetros de amostragem e quantização: na parte de cima da imagem tem-se o sinal original e uma grade representando a profundidade de bit (eixo vertical) e a frequência de amostragem (eixo horizontal), na parte de baixo temos o sinal digitalizado equivalente. Quanto maior a frequência de amostragem e a profundidade de bit (da esquerda para a direita) maior é a fidelidade do sinal amostrado e quantizado em relação ao sinal contínuo analógico.

Figura 2.5 – Transformação do sinal contínuo analógico em sinal digital



Fonte: Davis, Patronis e Brown (2013, p. 36)

Como os dois eixos do sinal representado digitalmente são discretos, ele pode ser restaurado com precisão a partir de números, pois uma vez que um parâmetro é expresso como um número discreto, uma série desses números pode ser transmitida inalterada. A frequência de amostragem e a precisão da quantização (profundidade de bits) são os únicos fatores que determinam a qualidade da representação digital de um sinal que não passou por nenhum processo de compressão.

2.2 Processamento de sinais

O processamento de sinais é uma área interdisciplinar relacionada à aquisição, representação, manipulação e transformação de sinais, sendo essencial para um grande número de aplicações práticas, podendo ser feito de forma analógica ou digital. Segundo Moura (2009), este

processo utiliza matemática, estatística, computação, heurística e representações linguísticas, formalismos e técnicas de representação, modelagem, análise, síntese, descoberta, recuperação, detecção, aquisição, extração, aprendizagem, segurança e ciência forense.

Os objetivos principais do processamento de sinais são melhorar a qualidade ou extrair informações úteis de um sinal, separar sinais previamente combinados e preparar sinais para armazenamento e transmissão.

2.2.1 Processamento analógico de sinais

O processamento de sinal analógico é um tipo de processamento de sinal realizado em sinais analógicos. As grandezas físicas são normalmente representadas como tensão ou corrente elétrica em torno dos componentes nos dispositivos eletrônicos.

Os elementos de processamento analógico comuns incluem capacitores, resistores e indutores (como elementos passivos) e transistores, válvulas e amplificadores operacionais (como elementos ativos). Alguns exemplos de processamento de sinal analógico em áudio incluem equalizadores analógicos, pré-amplificadores analógicos e compressores analógicos. Na Figura 2.6, pode-se ver um exemplo de um processador analógico.

Figura 2.6 – Processador analógico de sinal

Fonte: Do autor (2021)

2.2.2 Processamento digital de sinais

A rápida evolução da computação marcou a transição do processamento analógico para o processamento digital de sinais (DSP – *Digital Signal Processing*). Segundo Zölzer et al. (2011, p. 20), os fundamentos do processamento digital de sinais consistem na descrição de si-

nais digitais e na descrição de sistemas digitais, que são representados como uma sequência de números com representação numérica apropriada e algoritmos capazes de calcular uma sequência de números de saída a partir de uma sequência de números de entrada, respectivamente.

O DSP está entre as tecnologias mais poderosas, que causaram mudanças revolucionárias na ciência e na engenharia. Evoluções já foram alcançadas em uma ampla gama de campos por causa dessa tecnologia: comunicações, imagens, tratamento de áudio, sensoriamento remoto, medicina, entre outros. Cada uma dessas áreas desenvolveu uma profunda tecnologia de DSP, com seus próprios algoritmos, matemática e técnicas especializadas. Este trabalho foca no processamento digital de sinais de áudio.

Segundo Lathi (2008, p. 244), o processamento digital de sinais tem muitas vantagens em relação ao processamento analógico:

a) sistemas digitais são menos sensíveis às mudanças nos parâmetros dos componentes causadas pela variação da temperatura, idade e outros fatores, pois podem tolerar uma variação considerável nos valores do sinal. Por isso, esses sistemas têm um alto grau de precisão e estabilidade. Essa precisão pode ser aumentada usando circuitos com palavras binárias de tamanho maior;

b) sistemas digitais não necessitam de nenhum ajuste de fábrica e podem ser facilmente duplicados em volume sem nos preocuparmos com valores precisos de componentes, podendo também ser substituídos por um único chip usando circuitos VLSI (*very large scale integrated*);

c) as características dos filtros digitais podem ser facilmente alteradas simplesmente mudando o programa. A implementação em *hardware* digital permite o uso de microprocessadores, miniprocessadores, chaves digitais e circuitos VLSI;

d) uma grande variedade de filtros pode ser implementada por sistemas digitais;

e) sinais digitais podem ser facilmente armazenados sem deterioração da qualidade do sinal. Inclusive em servidores distantes;

f) sinais digitais podem ser codificados, levando a taxas de erro extremamente baixas e alta fidelidade, além de privacidade;

g) algoritmos de processamento de sinais mais sofisticados podem ser utilizados para processar sinais digitais;

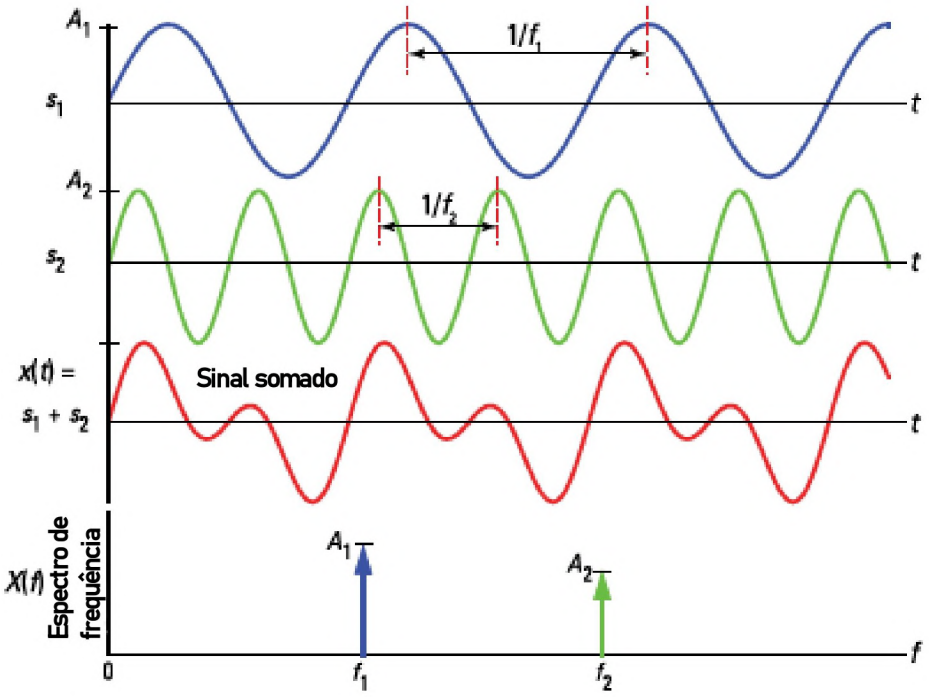
h) filtros digitais podem servir para várias entradas simultaneamente, por serem facilmente compartilhados. É mais fácil e mais eficiente multiplexar diversos sinais digitais em um mesmo canal;

i) a reprodução de mensagens digitais é extremamente confiável sem deterioração.

2.2.3 Representação de sinais no domínio da frequência

Como citado na Seção 2.1, uma das características do som é o timbre, que é definido pelas frequências além da fundamental que o compõem. Segundo Lathi (2008, p. 528) "[..] um sinal periódico pode ser representado como a soma de senoides (ou exponenciais) de várias frequências.". A frequência fundamental é a frequência mais baixa de uma forma de onda periódica; na música, é a nota percebida de um som. A representação de sinais no domínio da frequência é uma forma de visualizá-los, ela permite que vejamos quais as componentes de frequência em um sinal e qual a contribuição de cada uma delas na formação dele. Na Figura 2.7 podemos ver como diversas ondas somadas resultam em outra e a representação desse sinal no domínio da frequência.

Figura 2.7 – Composição de uma onda.



Fonte: Wickert (2013, p. 17)

Esse tipo de representação é muito útil para trabalharmos com estes sinais, pois na maioria das vezes o processamento do sinal tem o objetivo de atuar em uma frequência ou faixa de

frequência específica. “Engenheiros eletricistas instintivamente pensam em sinais em termos de seu espectro de frequência e pensam em sistemas em termos da sua resposta em frequência.” (LATHI, 2008, p. 528).

A frequência é uma transformação do tempo, portanto observar variações no domínio da frequência é apenas outra maneira de analisar as variações dos dados de séries temporais. Ao analisar um sinal dessa forma, pode-se observar componentes espectrais de um sinal que não são facilmente detectáveis nas formas de onda no domínio do tempo, como frequência dominante, potência, distorção, harmônicos, largura de banda, entre outros.

2.2.4 Espectro de sinal

O espectro de sinal é um tipo de gráfico utilizado para analisar sinais no domínio da frequência, ele mostra a força das variações do sinal em função das frequências que estão presentes nele. As unidades utilizadas são energia (dB) no eixo vertical e frequência (Hz) no eixo horizontal e geralmente o eixo horizontal é disposto em escala logarítmica. Um analisador de espectro é uma ferramenta que permite visualizar o espectro de um sinal, seu uso primário é medir a potência do espectro de sinais conhecidos e desconhecidos. A Figura 2.8 mostra um exemplo de analisador de espectro.

Figura 2.8 – Analisador de Espectro.

Fonte: Do autor (2021)

Esse tipo de ferramenta é muito utilizada pelas pessoas que trabalham com o processamento de áudio para avaliar seu trabalho. Nessas aplicações, ele mostra os níveis de intensidade

de faixas de frequência em toda a faixa típica da audição humana, em vez de exibir uma onda. O uso do analisador de espectro auxilia o profissional a balancear os sinais, criando harmonia para proporcionar conforto ao ouvido humano.

Uma ferramenta moderna deste tipo tem um alto grau de capacidade. Como a maioria usa técnicas digitais, o processamento de sinal é realizado usando o processamento de sinal digital usando a Transformada Rápida de Fourier (FFT – *Fast Fourier Transform*), esse método permite analisar um sinal não periódico no domínio da frequência. Ela permite que a partir de um sinal qualquer consiga-se extrair as frequências que o compõem.

2.2.5 Processamento de áudio

O caminho que vai do microfone ou dos instrumentos do músico ao alto-falante é muito longo. Um trabalho musical pode ser gravado em um estúdio de som ou em apresentações ao vivo, em ambas as formas é feita a captação de vozes e instrumentos em vários canais ou faixas. Fazendo-se dessa forma é possível que o produtor musical tenha maior flexibilidade e mais possibilidades para se chegar ao produto final, que pode ser uma música, um filme, um jogo de videogame, um *podcast*, entre outros.

A modificação da característica de som do sinal de entrada é o objetivo principal dos efeitos de áudio digital. As configurações dos parâmetros de controle geralmente são feitas por engenheiros de som, músicos (intérpretes, compositores ou fabricantes de instrumentos digitais) ou simplesmente pelo ouvinte de música, mas também podem fazer parte de um nível específico na cadeia de processamento de sinal do efeito de áudio digital. De acordo com os critérios acústicos, o usuário define seus parâmetros de controle para o efeito sonoro que deseja obter. (ZÖLZER et al., 2011, p. 1)

O complexo processo de combinar as faixas individuais em um produto final é chamado mixagem, um profissional de mixagem de áudio utiliza das mais variadas ferramentas para tratar os canais em sua individualidade ou em conjunto. Essas ferramentas utilizadas são os processadores de sinais, que podem fornecer várias funcionalidades importantes durante a mixagem, dentre elas pode-se destacar: filtragem, compressão, efeitos (como reverbs e delays), edição de sinais, entre outros.

2.3 Filtros

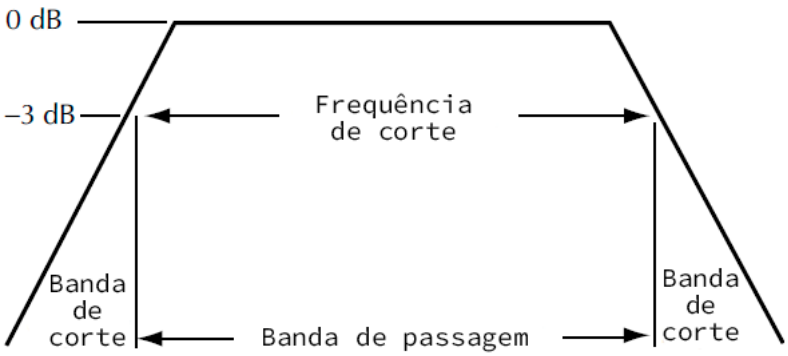
O termo filtro, em geral, pode ser definido como uma maneira de selecionar certos elementos com propriedades desejadas a partir de um conjunto maior de elementos. No campo particular dos efeitos de áudio digital, o sinal pode ser visto como um conjunto de harmônicos com diferentes frequências e amplitudes. Um filtro faz uma seleção dos harmônicos de acordo com as frequências que queremos rejeitar, reter ou enfatizar, ele modifica a amplitude desses harmônicos de acordo com sua frequência.

Podemos, então, considerar um filtro como um dispositivo que consegue separar sinais com base nas suas frequências. Estes dispositivos podem ser definidos em termos de sua banda passante, na qual apenas as frequências de interesse são permitidas, ou em termos de sua banda filtrada, na qual determinadas frequências são removidas.

> Um filtro ideal possui uma banda passante (ganho unitário) e uma banda filtrada (ganho zero) com alguma transição repentina entre a banda passante e a banda filtrada. Não existe banda (ou faixa) de transição. Para filtros práticos (ou realizáveis), por outro lado, a transição da banda passante para a banda filtrada (ou vice-versa) é gradual e ocorre durante uma faixa finita de frequências. (LATHI, 2008, p. 407)

Assim, considera-se a banda filtrada como a banda de frequências que após passar por um filtro tem uma perda superior a 3 dB em relação ao seu ganho nominal. Analogamente, considera-se a banda passante de um filtro como a banda específica de frequências que passam por ele com uma perda inferior a 3 dB em relação ao seu ganho nominal do filtro. Portanto, as bandas filtrada e passante são delimitadas pela frequência de corte do filtro, que é a frequência na qual o ganho fica 3 dB abaixo o ganho nominal. A Figura 2.9 ilustra as bandas passante e filtrada, assim como a frequência de corte de um filtro.

Figura 2.9 – Frequência de corte e bandas de um filtro.



Fonte: Ballou et al. (2008, p. 785)

Abaixo alguns dos tipos de filtro mais utilizados quando se trata de processamento de áudio:

a) passa-altas: atenua o ganho das frequências que estão abaixo da frequência de corte e não altera o das frequências que estão acima dela;

b) passa-baixas: atenua o ganho das frequências que estão acima da frequência de corte e não altera o das frequências que estão abaixo dela;

c) filtros *shelving*: atenuam ou aumentam o ganho, até um nível pré-determinado, da faixa de frequência que está abaixo ou acima de uma frequência escolhida; mantendo o ganho das frequências restantes inalterado;

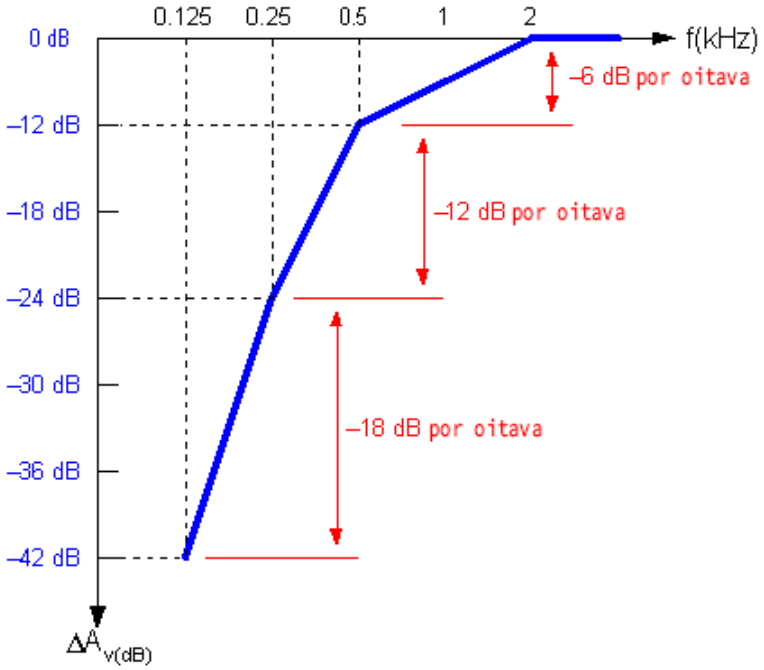
d) filtros paramétricos: atenuam ou aumentam o ganho das frequências que estão na vizinhança de uma frequência central definida (onde ocorrerá o ganho máximo definido), mantendo inalterado o ganho do restante das frequências.

Outra maneira de classificar os tipos de filtro é pela ordem deles. A ordem de um filtro está relacionada com a intensidade que o decréscimo no ganho ocorre quando se percorre o domínio das frequências. Em um filtro analógico, a ordem é determinada pelo número de elementos reativos do circuito (capacitores e indutores) que são adicionados para fins de alteração da resposta em frequência: em um filtro de primeira ordem temos apenas um elemento, em um de segunda ordem temos dois, assim por diante. A medida da atenuação geralmente se dá em decibéis por oitava, sendo que uma oitava é uma medida na qual uma frequência dobra ou cai pela metade em relação a outra. Filtros do tipo passa-altas e passa-baixas de primeira ordem têm atenuação de 6 dB por oitava, de segunda ordem 12 dB por oitava. Na Figura 2.10 estão ilustradas, de forma conceitual, curvas de atenuação para filtros passa-altas de primeira, segunda e terceira ordem.

2.3.1 Filtros digitais

Os filtros podem ser implementados usando meios inteiramente matemáticos de suas representações de funções de transferência no domínio do tempo. A função de transferência de um filtro no domínio de tempo discreto (z) pode ser mapeada a partir da função de transferência no domínio de tempo contínuo (s), por meio da transformação bilinear (Equação 2.4), como ilustrado na Figura 2.11.

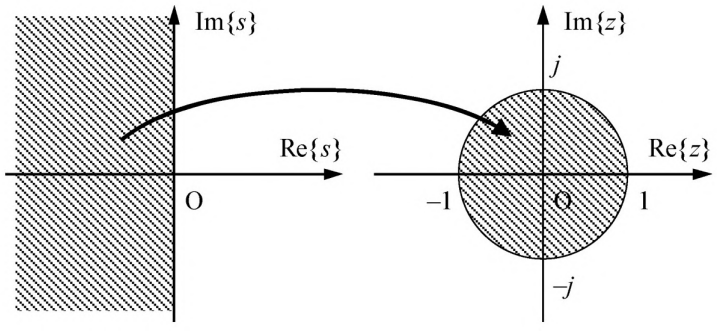
Figura 2.10 – Desenho conceitual para curvas de filtros passa-altas de primeira, segunda e terceira ordem.



Fonte: Paynter (2010)

$$s = \frac{2}{T} \times \frac{1 - z^{-1}}{1 + z^{-1}} \tag{2.4}$$

Figura 2.11 – Transformação bilinear.



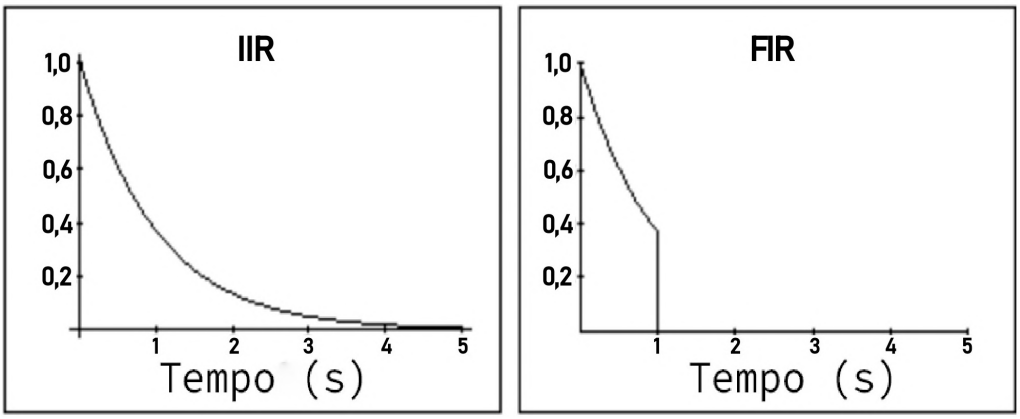
Fonte: Naredo et al. (2007)

Os filtros digitais utilizam algoritmos para os quais os DSPs (processadores de sinais digitais) são otimizados. Eles podem ser classificados de acordo com a natureza de sua resposta ao impulso: podem ter resposta ao impulso finita (FIR - *Finite impulse response*) ou resposta ao impulso infinita (IIR – *Infinite impulse response*). Essa natureza advém da maneira como esse filtro é implementado: um filtro FIR é implementado utilizando-se do método de convolução, enquanto um filtro IIR é implementado utilizando-se do método de recursão.

Os filtros de resposta ao impulso finito (FIR) são sempre estáveis e respondem ao sinal de entrada apenas uma vez, esse sinal percorre apenas um caminho, da entrada para a saída do filtro. No domínio temporal, o tempo durante o qual esse tipo de filtro responde a uma entrada é finito, fixo e prontamente estabelecido. Já os filtros de resposta ao impulso infinito (IIR) respondem a um impulso indefinidamente, mas não são necessariamente estáveis, pois o sinal é realimentado da saída para a entrada. (WATKINSON, 2001, p. 127 a 129)

Na Figura 2.12 encontram-se, ilustradas de forma didática, representações de respostas ao impulso infinita e finita, nota-se que a resposta ao impulso infinita continua indefinidamente pelo eixo do tempo, enquanto a resposta finita tem duração delimitada.

Figura 2.12 – Respostas ao impulso infinita e finita.



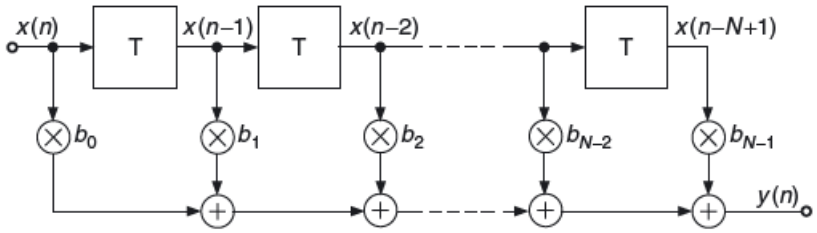
Fonte: Smith (2007)

2.3.1.1 Filtro FIR

Embora os filtros FIR sejam simples em conceito e fáceis de projetar, eles podem acabar usando grandes quantidades de poder de processamento em relação aos filtros IIR, pois são necessárias muitas operações por amostra. Na Figura 2.13 podemos ver a estrutura de um filtro FIR. Cada bloco T representa um atraso de um período de amostragem, as seções de multiplicação e adição são repetidas para cada amostra, a soma dos resultados de cada amostra sucessiva por superposição gera o sinal de saída completo. Isso requer o armazenamento de $N-1$ amostras de entrada anteriores e a execução é feita em N operações de multiplicação e adição por amostra. Essas operações são operações de convolução de sinais, a equação da convolução entre dois sinais está expressa na Equação 2.5.

$$s[n] * r[n] = \sum_{k=-\infty}^{\infty} s[k].r[n-k] \tag{2.5}$$

Figura 2.13 – Estrutura de um filtro FIR.



Fonte: Zölzer et al. (2011, p. 57)

2.3.1.2 Filtro IIR

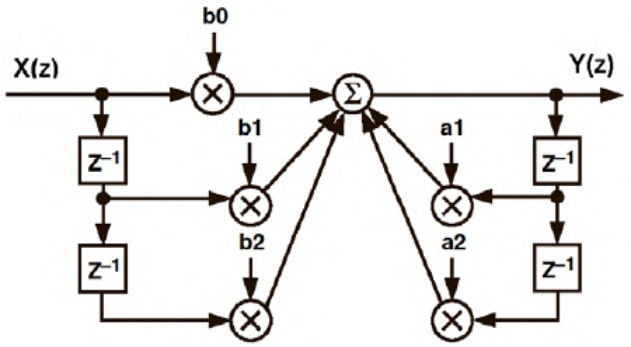
Um filtro IIR é basicamente um filtro FIR com *feedback* adicionado. Além dos dados de entrada, ele também alimenta o fluxo de dados de saída de volta por meio de outra série de multiplicadores e coeficientes; por isso, esses filtros são implementados por meio de algoritmos recursivos. Eles são uma maneira eficiente de obter uma resposta de impulso longa, sem ter que realizar uma convolução longa, suas respostas ao impulso são compostas por exponenciais em decomposição. Segundo Manolakis e Ingle (2011, p. 624), esses filtros podem ser projetados usando protótipos de filtro de tempo contínuo, pois eles têm respostas infinitamente longas.

Como a natureza dos componentes analógicos (capacitores, indutores, amplificadores operacionais) tende a ser naturalmente adequada para designs de filtro recursivo, a maioria dos designs de filtros IIR é derivada de designs de filtro analógico. O procedimento básico de projeto é pegar um projeto de filtro analógico e convertê-lo em uma implementação de filtro digital IIR.

Segundo Parker (2010, p. 75), quando um sistema filtro analógico está sendo atualizado para implementação digital, é importante preservar suas características de desempenho, um exemplo disso pode ser um equipamento de áudio profissional, e os filtros IIR podem aproximar melhor o desempenho de um filtro analógico.

Existem muitas configurações possíveis de filtros de resposta ao impulso infinito (IIR), a Figura 2.14 mostra a forma direta de um filtro biquadrático (filtro de segunda ordem, linear e recursivo, que contém dois polos e dois zeros), no qual as amostras de entrada e saída são passadas para a linha de atraso. Cada bloco Z^{-1} representa um atraso de um período de amostragem, a soma das entradas e saídas multiplicadas pelos coeficientes gera o sinal de saída completo. Essa forma é mais adequada para a iteração de ponto fixo, na qual é importante que os termos atrasados mantenham o máximo de precisão possível.

Figura 2.14 – Estrutura de um filtro IIR.



Fonte: Herrera (2004, p. 48)

Esse filtro é chamado de biquadrático, pois sua função de transferência no domínio Z é composta por duas funções quadráticas, uma no numerador e outra no denominador, como mostra a Equação 2.6 abaixo:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} \tag{2.6}$$

Para que esse filtro seja estável, é necessário que seus dois polos se encontrem dentro do círculo unitário, conforme Figura 2.11. Tratando-se de filtro discretos, em geral isso é verdade, ou seja, todos os polos devem estar dentro do círculo unitário no domínio Z para que o filtro seja estável.

2.3.1.3 Filtro *peaking*

Enquanto os filtro Passa-altas e Passa-baixas atenuam as frequências que estejam abaixo ou acima de uma frequência de corte, os filtros *peaking* alteram o ganho de uma banda de frequências definidas pelos parâmetros escolhidos, são eles:

a) frequência central (f_c): é a frequência onde ocorrerá o máximo ganho ou atenuação;

b) ganho (K): é a magnitude de aumento ou atenuação na frequência central (f_c), que pode ser expresso na forma absoluta ou em decibéis;

c) largura de banda (BW): a gama de frequências a partir de (f_c) na qual será aplicado o ganho. Frequentemente este parâmetro de largura de banda é substituído pelo fator de qualidade Q, derivado de BW e f_c.

Os filtros desse tipo atenuam ou aumentam o ganho das frequências que estão na vizinhança da frequência central definida (onde ocorrerá o ganho máximo definido), mantendo inalterado o ganho do restante das frequências. Eles produzem resposta em frequência na qual a magnitude é igual à do sinal original em todo o espectro, exceto nas frequências próximas a f_c, esta inclusa. Um filtro *peaking* pode ser usado de várias maneiras, isso é possível por causa da característica de determinação individual dos parâmetros dele.

Segundo Välimäki e Reiss (2016, p. 3), adicionar um pico de ganho pode ser útil para destacar um instrumento de outros ou para adicionar coloração ao som de um instrumento, aumentando ou reduzindo uma faixa de frequência específica. As atenuações podem ser usadas para reduzir sons indesejados, incluindo a remoção do zumbido da linha de força (50 Hz ou 60 Hz e seus harmônicos) e redução do *feedback*. Para remover artefatos sem afetar o restante do sinal, uma largura de banda estreita pode ser usada.

2.3.1.3.1 Parâmetro *Q*

O fator de qualidade Q de um filtro *peaking* é relacionado com sua frequência central f_c e sua largura de banda BW, de modo que fixando um Q qualquer e alterando o ganho K, BW também seja alterada. O objetivo é fazer com que o decréscimo do ganho (dB por oitava) se mantenha igual ao do K anterior ao percorrer o eixo das frequências.

Do ponto de vista do usuário, seria desejável que dois filtros que possuíssem Q e f_c idênticos e ganhos opostos, ou seja, “x” dB e “-x” dB, se somassem em uma resposta plana. Dessa forma, caso primeiro filtro tenha uma função de transferência $H_1(s)$, o último deve ter função de transferência $H_2(s) = 1/H_1(s)$. Se as curvas de resposta em frequência para a mesma quantidade de ganho e corte são imagens espelhadas entre si no eixo de ganho da unidade, a característica de resposta é chamada recíproca ou simétrica.

Para atender à propriedade desejada acima, neste trabalho foi utilizada a definição de Q desenvolvida por Robert Bristow-Johnson. Como Bristow-Johnson (2001) demonstrou em seus trabalhos , os parâmetros da Equação 2.6 podem ser encontrados a partir da transformação bilinear da função de transferência do filtro *peaking* no tempo contínuo (Equação 2.7).

$$H(s) = \frac{s^2 + s\frac{A}{Q} + 1}{s^2 + s\frac{s}{AQ} + 1} \tag{2.7}$$

Sendo:

$$A = \sqrt{K} \tag{2.8}$$

$$\omega_0 = \frac{2\pi f_0}{F_s} \tag{2.9}$$

$$\alpha = \frac{sen\omega_0}{2Q} \tag{2.10}$$

onde K é o ganho absoluto, f_0 a frequência central, F_s a frequência de amostragem e Q o parâmetro fator de qualidade do filtro (*quality factor*). Os coeficientes b_0, b_1, b_2, a_0, a_1 e a_2 podem ser definidos como:

$$b_0 = 1 + \alpha A \tag{2.11}$$

$$b_1 = -2cos\omega_0 \tag{2.12}$$

$$b_2 = 1 - \alpha A \tag{2.13}$$

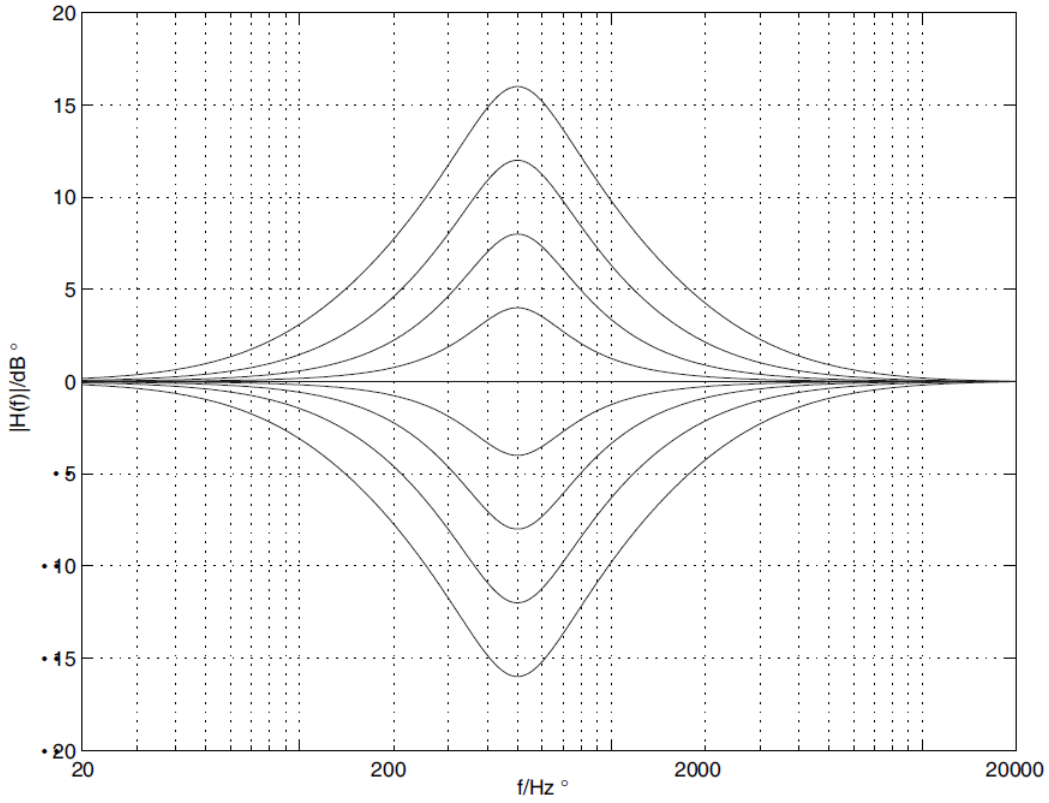
$$a_0 = 1 + \frac{\alpha}{A} \tag{2.14}$$

$$a_1 = -2cos\omega_0 \tag{2.15}$$

$$a_2 = 1 - \frac{\alpha}{A} \tag{2.16}$$

Nas figuras Figura 2.15, Figura 2.16 e Figura 2.17 estão ilustradas respostas em frequência de filtros *peaking* com a variação dos três parâmetros, K, Q e f_c. Nelas podemos observar que os filtros *peaking* de mesmo Q e ganhos K opostos são simétricos em relação ao eixo de ganho, dessa forma, quando somados eles se anulam. Esse é o tipo de resposta desejada para este tipo de filtro, por ser mais útil musicalmente.

Figura 2.15 – Resposta em frequência de filtros *peaking* com variação de K.



Legenda: Filtros *peaking* com f_c de 500Hz, Q de 1,25 e variando-se K.
Fonte: Zölzer (2008, p. 126)

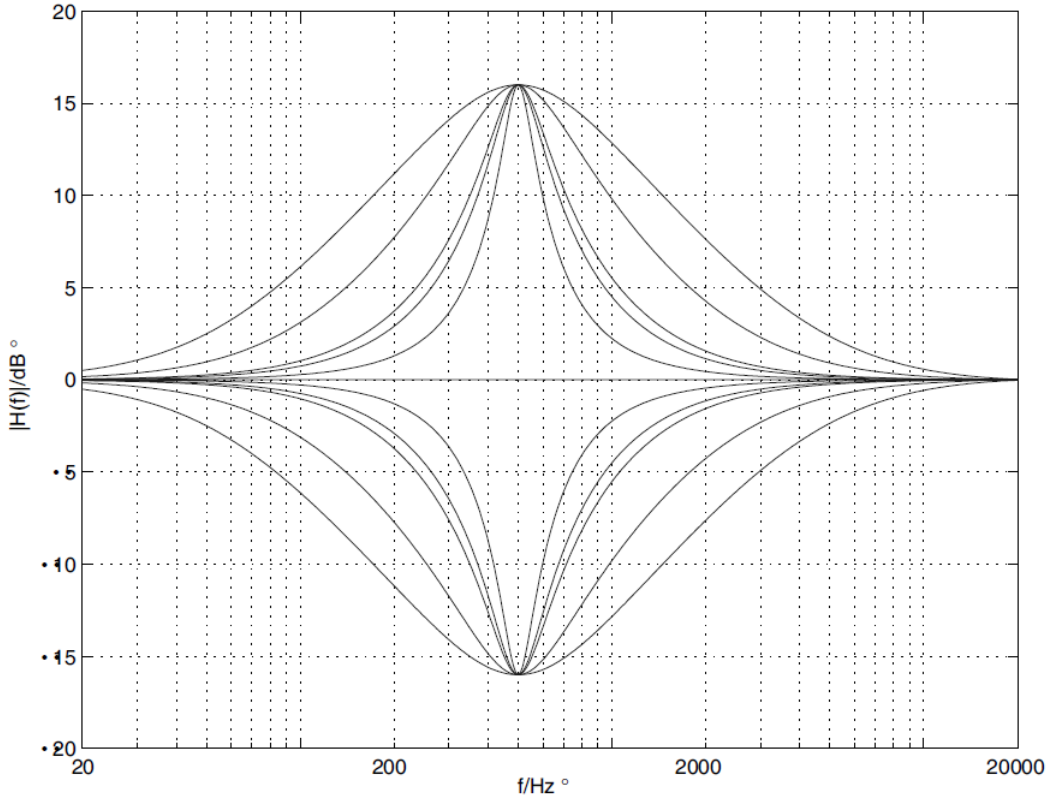
2.4 Equalização

A equalização é um dos métodos mais importantes para o processamento de sinais de áudio, podendo ser feita de forma analógica ou digital. Segundo Välimäki e Reiss (2016, p. 1), ela é uma área de pesquisa vasta e ativa, o termo tem sido usado para qualquer procedimento que envolva alterar ou ajustar a magnitude da resposta em frequência.

Equalizadores são dispositivos ou componentes projetados para remover características indesejáveis na magnitude ou resposta de fase do sistema, tornando a resposta igual novamente. Esses dispositivos consistem em filtros implementados de maneira a fornecer controle sobre a resposta de frequência, permitindo ao usuário modificar os parâmetros para chegar à resposta que está tentando recriar. (BALLOU et al., 2008, p.800)

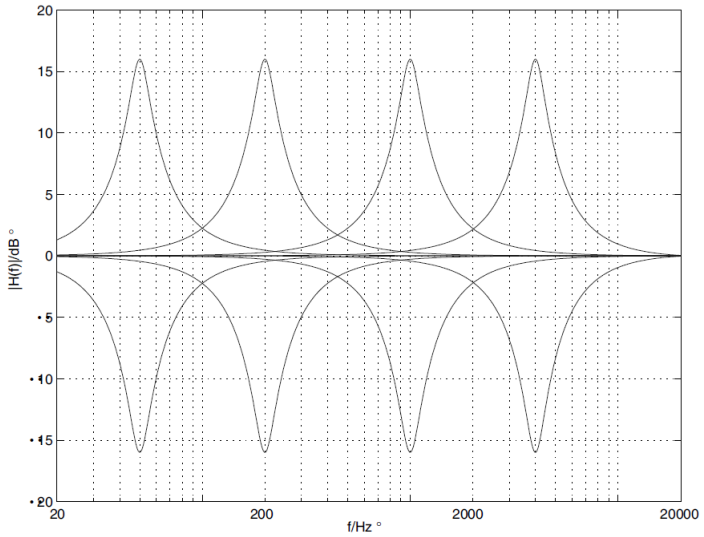
"Um equalizador é constituído por um banco de filtros cujas funções de transferência (FT) podem ser ajustadas para atender a situações específicas. Os principais tipos de filtros empregados são passa baixas, passa altas e paramétricos (*peaking*)."(HERRERA, 2004, p. 43) Esses filtros são projetados para faixas de frequência distintas umas das outras, que geralmente

Figura 2.16 – Resposta em frequência de filtros *peaking* com variação de Q.



Legenda: Filtros *peaking*, com f_c de 500 Hz, K de +-16 dB e variando-se Q.
Fonte: Zölzer (2008, p. 127)

Figura 2.17 – Resposta em frequência de filtros *peaking* com variação de f_c.



Legenda: Filtros *peaking*, com K de +-16 dB, Q de 1,25 e variando-se f_c.
Fonte: Zölzer (2008, p. 127)

estão contidas entre 20 Hz e 20 kHz. Os controles são organizados em termos de frequências

centrais, larguras de banda e ganhos, ou outros parâmetros derivados destes (como no caso do Q do filtro *peaking*), em vez dos valores reais dos parâmetros de suas funções de transferência.

A equalização é relevante em qualquer situação na qual seja benéfico moldar um espectro, desde os usos mais comuns até os mais específicos. Apesar de serem aplicados de maneira mais sofisticada nos estúdios de som, os filtros são usados para equalização de som em quase todos os produtos de consumo, como rádios de carros e amplificadores de alta fidelidade. Alguns exemplos de uso de equalizadores são:

a) controle de tom: contido em uma vasta gama de aparelhos reprodutores de som, permite o fácil ajuste das frequências do áudio reproduzido. Geralmente possui controle de grave e agudo, sendo que alguns também possuem controle de médio. Um exemplo de controle de tom é a aplicação em guitarras, outro exemplo pode ser visto na Figura 2.18;

b) equalização de alto-falantes: tem por objetivo buscar uma resposta desejada, não danificar os alto-falantes reproduzindo frequências além da faixa possível e dividir a gama de frequências entre dois ou mais falantes, sendo esta aplicação diferente do *crossover* passivo[1], que também pode ser utilizado para este fim;

c) equalização de sala: utilizado para compensar as ressonâncias de um local, para obter uma resposta sonora mais plana e consistente em sua extensão;

d) equalização de fones de ouvido: tem o mesmo princípio da equalização de alto-falantes, mas também busca compensar as ressonâncias do canal auditivo;

e) equalização para combater o ruído ambiente: como o ruído ambiente geralmente tem mais energia em baixas frequências, a equalização é utilizada para combater o fenômeno de mascaramento auditivo. Atualmente, estão se tornando mais comuns os fones de ouvido com cancelamento de ruído;

f) produção de áudio: um uso bem estabelecido, com métodos que vêm sendo aperfeiçoados para atingir os objetivos técnicos e artísticos de uma produção. Nos últimos anos, a tecnologia dos equalizadores vem evoluindo e oferecendo novas possibilidades aos criadores de conteúdo de áudio.

[1] Circuito eletrônico com elementos passivos, que divide um sinal de áudio em duas ou mais faixas de frequência.

Figura 2.18 – Controle de tom de um aparelho de som doméstico.

Legenda: Neste modelo, o controle de tom apresenta possibilidade de ajuste de agudos (*treble*) e graves (*bass*).

Fonte: Do autor (2021)

Segundo Davis, Patronis e Brown (2013, p.568), uma discussão de quando e onde é apropriado usar um equalizador e quando e onde ele não deve ser empregado é útil, já que a equalização, assim como a maioria dos componentes do sistema de som, pode ser mal aplicada. O uso excessivo de filtros pode gerar grandes defasagens no sinal, podendo resultar em cancelamento de fase.

2.4.1 Equalizadores paramétricos

Muitas aplicações de processamento de sinal envolvem aumentar ou atenuar apenas uma pequena parte do espectro de frequência de um sinal, sem afetar o restante do espectro. Este efeito é comumente obtido em aplicações de áudio usando filtros biquadráticos do tipo *peaking*. Na equalização de áudio digital, qualquer resposta de frequência desejada pode ser realizada pelo agrupamento de várias seções em cascata desses filtros, fazendo com que os efeitos de cada um deles sejam cumulativos em uma escala de decibéis. Além disso, eles também são frequentemente associados a outros tipos de filtro, como passa-altas, passa-baixas e *shelving*. A esse conjunto de filtros é dado o nome de equalizador paramétrico.

O equalizador paramétrico é o mais poderoso e flexível dos tipos de equalizador. Ele permite o acesso direto aos parâmetros da função de transferência por meio do controle dos coeficientes associados, portanto, não é necessário calcular um conjunto completo de coeficientes para uma função de transferência de segunda ordem. Com os seus três parâmetros de ajuste

em cada seção – ganho K, frequência central f_c e fator de qualidade Q – o operador consegue adicionar um pico ou atenuação em um local arbitrário no espectro de áudio sem modificar o conteúdo espectral do restante das frequências, onde sua resposta em magnitude é unitária (0 dB). Um exemplo de equalizador paramétrico pode ser visto na Figura 2.19.

A frequência central de cada banda do equalizador é ajustável em uma faixa de frequência limitada. Isso é feito para que a independência dos parâmetros possa ser mantida. Cada seção normalmente cobre uma faixa de frequência ligeiramente diferente. Além disso, normalmente é fornecido um ajuste maior para cortar níveis do que para aumentar. Em termos de Q, o intervalo está normalmente entre 0,3 e 3, sendo que a posição inicial do controle geralmente fica em 0,707, já que este é o valor criticamente amortecido. (BALLOU et al., 2008, p. 801)

Figura 2.19 – Equalizador paramétrico Waves Q10 Equalizer.

Legenda: Neste modelo, o equalizador apresenta possibilidade de controle de 10 bandas de filtros.
Fonte: Do autor (2021)

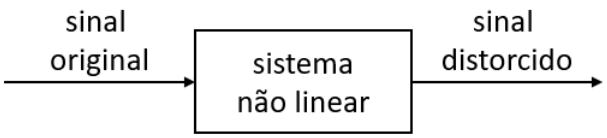
Este tipo de equalizador é muito útil em aplicações musicais, pois de acordo com o ajuste dos parâmetros pode ser possível realizar alterações precisas e pontuais no espectro de frequência de um sinal. Adicionar um pico pode ser útil para destacar um instrumento em uma mistura complexa, ou para modificar deliberadamente o timbre de um instrumento, aumentando

ou reduzindo uma faixa de frequência específica. As atenuações podem ser usadas para remover artefatos indesejados, incluindo a remoção do ruído da rede elétrica (50Hz ou 60Hz) e redução de ressonância, sem afetar o restante do sinal, bastando utilizar uma largura de banda estreita para fazer estas atenuações.

2.5 *Waveshaping*

Waveshaping é o processo que consiste em modular determinado sinal em um diferente por meio da aplicação de uma função de transferência escolhida ao sinal original. A função aplicada cria uma distorção de maneira eficaz no sinal de entrada, transformando em uma nova forma de onda ao alterar a proporção de seus componentes. Segundo Oliveira (2013, p. 133), o efeito da distorção é obtido introduzindo o sinal em um bloco de distorção, que contém uma função não linear (*waveshaping*). Na Figura 2.20 está ilustrado um sistema que demonstra o fluxo de sinal em um *waveshaper* simples.

Figura 2.20 – Sistema de processamento não linear.



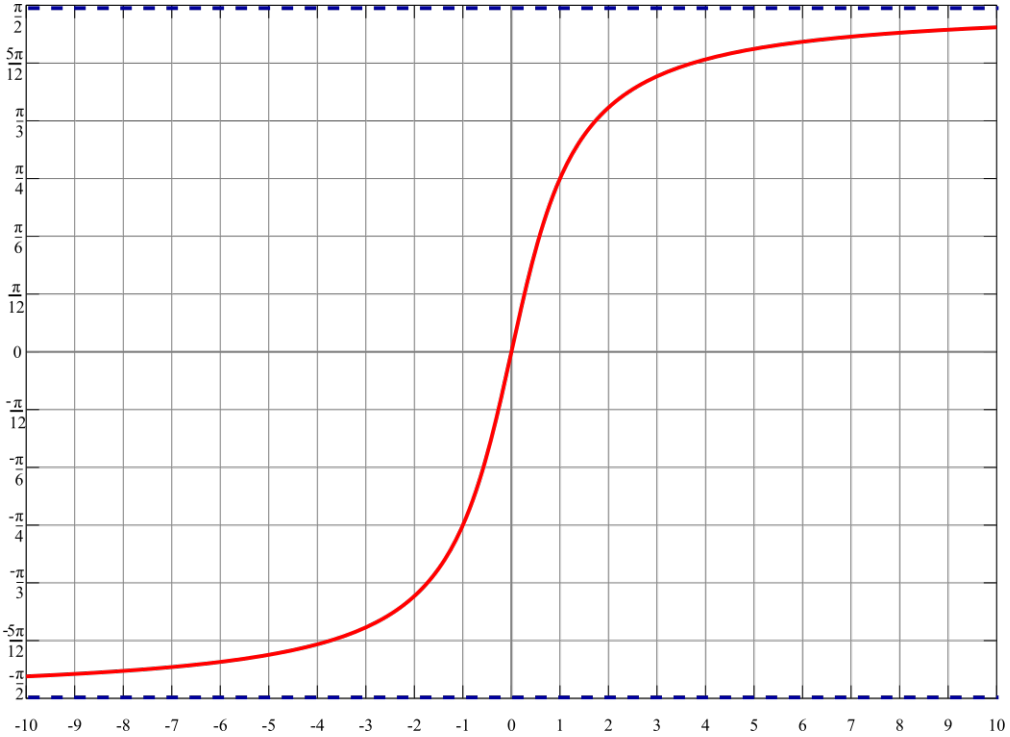
Fonte: Do autor (2021)

A distorção ocorre quando o sistema opera na região não linear da função aplicada, na qual o sinal atinge o nível de entrada máximo e então o nível de saída é limitado aos limiares dessa função. Os resultados do processamento não linear são complexos e muitas vezes sonoramente mais ricos, mas por outro lado também são mais difíceis de entender ou prever.

Ao aplicar uma determinada função de transferência, os *waveshapers* adicionam conteúdo harmônico ao sinal original. As possibilidades de formação de ondas podem ser infinitas, pois qualquer tipo de função de transferência pode ser aplicado ao sinal de entrada. Por exemplo, uma onda senoidal simples pode ser moldada em uma forma de onda diferente aplicando-se uma função matemática a ela, como a função arco tangente, por exemplo, mostrada na Figura 2.21.

A nova forma de onda gerada depende da forma da onda de entrada, da função de transferência e da amplitude do sinal de entrada. Como a amplitude de entrada afeta a forma da forma de onda de saída, tem-se uma maneira simples de alcançar timbres diferentes, simples-

Figura 2.21 – Gráfico da função $f(x) = arctan(x)$.



Fonte: Funções... (2021)

mente variando o nível de entrada. Por isso, é comum incluir um controle de amplitude de entrada como parte do sistema de *waveshaping*. Uma amplitude pequena leva a relativamente pouca distorção e uma amplitude maior resulta em um timbre com mais distorção; quanto mais elevado o ganho de entrada mais harmônicos são gerados no sinal. Além do controle de amplitude de entrada, também é comum incluir um controle de *blend*, que faz uma mistura do sinal original com o sinal distorcido na proporção desejada, aumentando ainda mais as possibilidades de timbre a ser atingido.

A distorção pode ser adicionada a um sinal digital quando se busca trazer a ele características de sinais analógicos. Segundo Oliveira (2013, p. 131), pode-se modelar matematicamente uma distorção não linear como uma série temporal gerada por alguma função não linear.

Processadores não lineares são usados para a simulação de amplificadores valvulados ou sistemas de áudio não lineares, nos quais a não linearidade fornece uma distorção de som com uma característica específica desejada. Várias aplicações demonstram a importância dos processadores não lineares.

Segundo Oliveira (2013, p. 131), muitas são as aplicações de algoritmos que processam o som de maneira não-linear no ramo de algoritmos de efeitos sonoros. As mais comuns são algoritmos de processamento dinâmicos, simuladores de circuitos distorcedores (como pedais

e válvulas), simulações de gravadores analógicos e processamentos sonoros de realçamento baseados em psicoacústica.

Estes processamentos criam componentes de frequência harmônicos ou inarmônicos, intencionais ou não, que não estão presentes no sinal de entrada. A distorção é introduzida por meio de não linearidades do sistema. A aplicação desses dispositivos é uma arte e é uma das principais ferramentas para músicos e produtores. (ZÖLZER et al., 2011)

Os processadores não lineares, como os dispositivos de distorção, são ferramentas desafiadoras para músicos e engenheiros de som. O sucesso da aplicação desses processadores de áudio depende do controle adequado desses dispositivos. Uma variedade de parâmetros de controle interativo influencia a qualidade do som resultante.

2.6 *Delay*

Delay é uma técnica de processamento de sinal de áudio que consiste na gravação e armazenamento de um sinal de entrada e sua reprodução após um período de tempo. Quando este sinal atrasado é somado ao sinal original, cria um efeito semelhante a um eco, pois é ouvido após áudio original. Segundo Ballou et al. (2008, p. 809), a implementação de um *delay* requer meios de armazenar o sinal e, em seguida, reinserir o sinal armazenado após um período de tempo controlado. Isso pode ser feito armazenando um registro contínuo do som ou dividindo-o em amostras, armazenadas separadamente.

Devido à distância que existe entre nossos dois ouvidos, existe um atraso para o mesmo som que chega em um e no outro, isso acontece por causa da velocidade de propagação do som pelo ar. Utilizando-se desse atraso, nosso cérebro consegue definir a localização das fontes sonoras, pois o som chega primeiro ao ouvido que está mais próximo da origem. Quando um *delay* constante é aplicado a um par de alto-falantes com configuração estéreo, há uma percepção de que as fontes sonoras estão próximas do alto-falante que emite o sinal sonoro anterior.

O *delay* é relativo. Para que um *delay* tenha efeito sobre um som, ele deve ser ouvido em conjunto com o som original sem atrasos. Esse fenômeno pode acontecer de duas maneiras. Um mesmo som pode chegar ao ouvinte após percorrer dois caminhos de comprimento diferentes, como um som direto e um som refletido, ou o sinal podem atrasado artificialmente e reinserido para ser ouvido de um único local. (BALLOU et al., 2008, p. 805)

Além da utilização para a percepção direcional do som, o *delay* pode ser utilizado de outras formas no áudio, gerando outros tipos de efeito. O sinal atrasado pode ser reproduzido várias vezes, ou realimentado na gravação, para criar o som de um eco repetitivo e decadente. Também pode ser utilizado para alinhar os sons emitidos por sistemas sonoros que utilizam mais de uma fonte de som que estão em locais distintos; o atraso nesse caso é utilizado para compensar o tempo que o som leva para percorrer a distância entre uma fonte sonora, a outra e o ouvinte. Além disso, segundo Zölzer et al. (2011, p. 80), pode ser usado como um bloco de construção básico para implementar efeitos de áudio baseados em atraso, reverberação artificial e modelos físicos para simulação de instrumentos.

Existem vários tipos de implementação de delay possíveis, alguns dos exemplos mais conhecidos são:

a) *small delays*: fazem uso do pequeno atraso de fase de um filtro passa tudo, geralmente utilizados para sistemas de *crossover* ativos[2] para alinhamento de caixas de som com mais de um alto falante;

b) *delay* acústico: envia um sinal sonoro por um meio físico, como um tubo, fazendo a captação na outra ponta, gerando assim um atraso;

c) *tape delay*: utiliza um *loop* de fita magnética, o som é gravado na fita pela cabeça de gravação e lido de volta por uma ou mais cabeças de reprodução, então a fita é apagada e volta ao início, a distância entre essas cabeças gera o atraso desejado. Um exemplo de *delay* de fita pode ser visto na Figura 2.22;

d) *shift register delay*: utilizam capacitores para mover amostras de sinais na forma de carga elétrica de um estágio de armazenamento de carga para outro, sob controle de um sinal temporizador.

2.6.1 O *delay* digital

Os *delays* digitais utilizam de *buffers* de dados, a memória é geralmente organizada em um *buffer* circular; quando os dados que devem ser registrados alcançam o final do *buffer*, as amostras subsequentes são registradas novamente no início do *buffer*, sobrescrevendo os dados

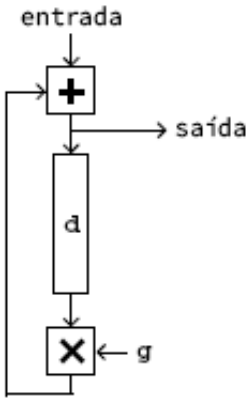
[2] Circuito eletrônico com elementos ativos, que divide um sinal de áudio em duas ou mais faixas de frequência.

Figura 2.22 – *Tape delay*.

Fonte: Jouaud (2020)

que estavam armazenados anteriormente. Da mesma maneira, o *buffer* é lido posteriormente após o tempo de atraso definido e adicionado ao sinal original. O diagrama de blocos de uma realimentação de um sinal atrasado utilizando um *buffer* circular pode ser visto na Figura 2.23.

Figura 2.23 – Realimentação com *buffer* circular.



Fonte: Puckette (2006, p. 185)

O sinal de entrada é armazenado em um *buffer* atrasado em d amostras, esse *buffer* é realimentado na entrada com ganho g. Devido ao *loop* de *feedback*, esse filtro tem um tempo de resposta infinito, podemos ver sua resposta a um impulso na Figura 2.24.

O ganho g aplicado deve ser menor que a unidade em valor absoluto, pois com um ganho maior que um cada atraso teria uma magnitude maior que o anterior. Isso faria com que a amplitude do sinal crescesse exponencialmente, tornando o sistema instável.

Figura 2.24 – Resposta de um *delay* com *buffer* circular.

Fonte: Puckette (2006, p. 186)

2.7 Reverberadores

A reverberação é a persistência do som em um espaço depois que o som original foi removido. Ela permite que os seres humanos diferenciem sons que são produzidos em ambientes diferentes e estimar a qual distância a fonte sonora se encontra. As fontes de reverberação são variáveis. Em ambientes acústicos reais, a reverberação é o somatório de reflexões com atrasos aleatórios e reflexões de reflexões de pisos, paredes, tetos e obstáculos.

A reverberação é o resultado do somatório de muitos reflexos do som original. A reverberação é composta primeiro por um som direto, seguido por um curto intervalo, conhecido como intervalo de tempo inicial (ITG – *Inicial Time Gap*). Em seguida, vêm os primeiros ecos de reflexão iniciais distintos, causados pela reflexão do som em superfícies próximas à fonte ou ao ouvinte. Depois, os sons refletidos começam a refletidos novamente, até que o nível de energia comece a decair de forma constante. Essa taxa de decaimento está relacionada às distâncias percorridas e à quantidade de absorção na sala. (BALLOU et al., 2008, p. 808)

A reverberação é necessária na maioria dos sons, principalmente quando se trata de música, e adição de reverberação artificial a gravações é uma prática padrão na indústria de áudio. Um som completamente seco ouvido em um conjunto de fones de ouvido dá a impressão de originar-se dentro da própria cabeça. Uma reverberação apropriada adiciona uma sensação de espaço ao som, a adição gradual desse efeito faz com que o som pareça estar se afastando à sua frente. Por outro lado, reverberação em excesso pode fazer com que o som perca a clareza e a definição.

Segundo Kahrs e Brandenburg (2002, p. 86), a importância da reverberação na música gravada resultou na criação de reverberadores artificiais, dispositivos eletroacústicos que simulam a reverberação de salas. Os primeiros dispositivos usavam molas ou placas de aço equipadas com transdutores. Com o advento da eletrônica digital, esses dispositivos foram substituídos pelo moderno reverberador digital, um processador de sinal que simula a reverberação usando um filtro linear de tempo discreto. Esses dispositivos estão muito presentes na indústria de produção de áudio.

Existem muitas maneiras de gerar reverberação artificial, o desafio é encontrar o método que imite salas de música reais e não introduza aberrações de resposta de frequência no sinal. Historicamente, uma sala de reverberação dedicada foi empregada; o som original é reproduzido na sala e o sinal reverberado é captado e inserido de volta ao original na quantidade necessária para atingir o efeito desejado. Outros tipos de reverberadores utilizados foram o *plate*, no qual um falante emite ondas em uma placa metálica fazendo a vibrar, e o de mola (*spring*), que utiliza molas metálicas para produzir a reverberação. Com o avanço das tecnologias, passou-se a utilizar filtros digitais para produzir o efeito de reverberação desejado, barateando os custos e proporcionando maior versatilidade, já que com uma simples alteração dos parâmetros é possível se produzir uma infinidade de respostas diferentes.

O problema de projetar um reverberador pode ser abordado de um ponto de vista físico ou perceptivo. A abordagem física busca simular exatamente a propagação do som da fonte ao ouvinte por uma determinada sala, simplesmente medindo a resposta de impulso binaural de uma sala existente e, em seguida, renderizando a reverberação por convolução. A vantagem dessa abordagem é que ela oferece uma relação direta entre a especificação física da sala e a reverberação resultante. No entanto, essa abordagem é computacionalmente cara e bastante inflexível. Já a abordagem perceptiva, busca reproduzir apenas as características perceptualmente salientes de reverberação. Se cada atributo perceptivo pode ser associado a uma característica física da resposta ao impulso, então podemos tentar construir um filtro digital com N parâmetros que reproduz exatamente esses N atributos. O reverberador deve então produzir uma reverberação muito parecida com a natural, mesmo que os detalhes das respostas ao impulso sejam diferentes. (KAHRS; BRANDENBURG, 2002, p. 87, 88)

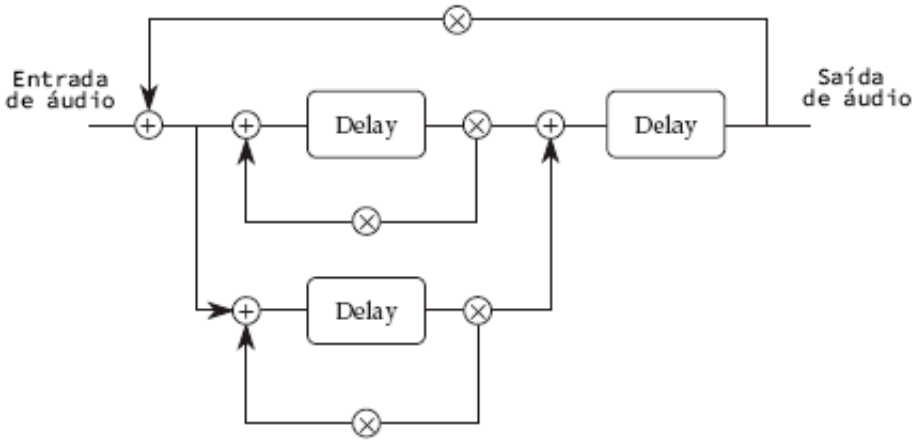
A abordagem perceptiva tem como principais vantagens poder ser baseada em filtros IIR eficientes, proporcionar controle em tempo real dos parâmetros perceptivos relevantes e conseguir simular diversos tipos de reverberação com apenas um algoritmo. Essa tem sido a

abordagem mais utilizada para projetos de algoritmos de reverberação. O *delay* é usado como base para a reverberação, pois com ele podemos facilmente armazenar o sinal e liberá-lo posteriormente, assim como os reflexos de um ambiente chegam ao ouvinte atrasados do som direto.

Portanto, um *reverb* pode ser emulado usando variações de atraso de tempo, combinando atrasos longos e curtos para emular as diferentes reflexões e difrações presentes em um ambiente. As realimentações aplicadas podem também passar por uma equalização, buscando imitar a absorção de frequências mais altas em ambientes naturais.

Um sistema de reverberação digital pode ser visto na Figura 2.25. Nela podemos ver os atrasos do sinal de entrada e a realimentação desses sinais atrasados no sinal original.

Figura 2.25 – Sistema demonstrando um reverberador simples.

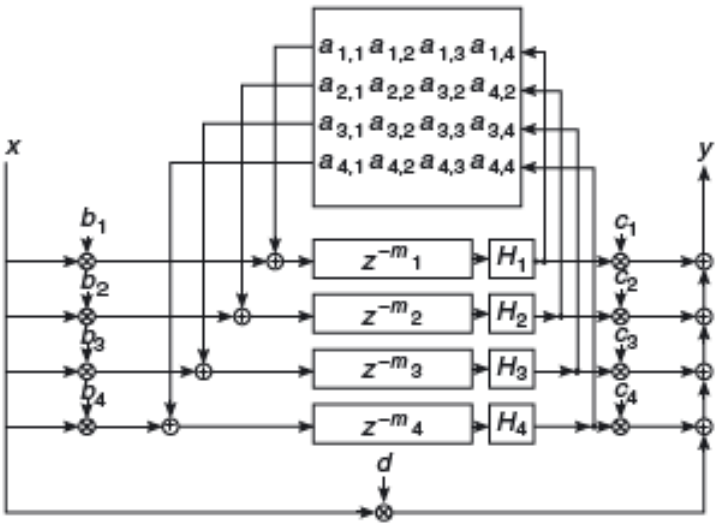


Fonte: Everest e Pohlmann (2009, p. 176)

Segundo Zölzer et al. (2011, p. 169), em 1982, J. Stautner e M. Puckette introduziram uma estrutura para reverberação artificial baseada em linhas de atraso interconectadas em um *loop* de *feedback* por meio de uma matriz, como pode ser visto na Figura 2.26. Mais tarde, estruturas como essa foram chamadas de redes de atraso de *feedback* (FDNs – *feedback delay networks*).

Como comentado por Ballou et al. (2008, p. 808), alguns parâmetros que precisam ser considerados para ajudar a alcançar o realismo na simulação de reverberação, como a distância da fonte, o tamanho da sala, as frequências absorvidas e as demais características do ambiente. Os tempos de atraso não devem ser produtos harmônicos uns dos outros para minimizar o acúmulo de ondas estacionárias e *comb filters*. Assim como no *delay*, qualquer ganho deve ser menor que um, caso contrário, o som aumentará exponencialmente até que ocorra distorção.

Figura 2.26 – Rede de atraso de *feedback* de quarta ordem.



Fonte: Zölzer et al. (2011, p. 170)

3 MATERIAIS E MÉTODOS

3.1 As ferramentas utilizadas

3.1.1 JUCE

Para desenvolvimento dos *plug-ins* deste trabalho, foi utilizado um *framework* em C++ voltado para aplicações de áudio, chamado JUCE.

JUCE é um *framework* C++ multiplataforma de código aberto para a criação de aplicativos de alta qualidade para *desktop* e dispositivos móveis, dentre eles *plug-ins* de áudio VST, VST3, AU, AUv3, RTAS e AAX. O JUCE pode ser facilmente integrado com projetos existentes via CMake, ou pode ser usado como uma ferramenta de geração de projetos via Projucer. Os projetos podem ser exportados para as IDEs Xcode (macOS e iOS), Visual Studio, Android Studio, Code::Blocks e Linux Makefiles, além disso o Projucer tem um editor de código-fonte próprio (JUCE, 2021).

Foi originalmente desenvolvido como parte da DAW 'Tracktion' e posteriormente extraído como uma estrutura autônoma, por isso uma de suas características quando comparado a outros *frameworks* semelhantes é seu grande conjunto de funcionalidades de áudio.

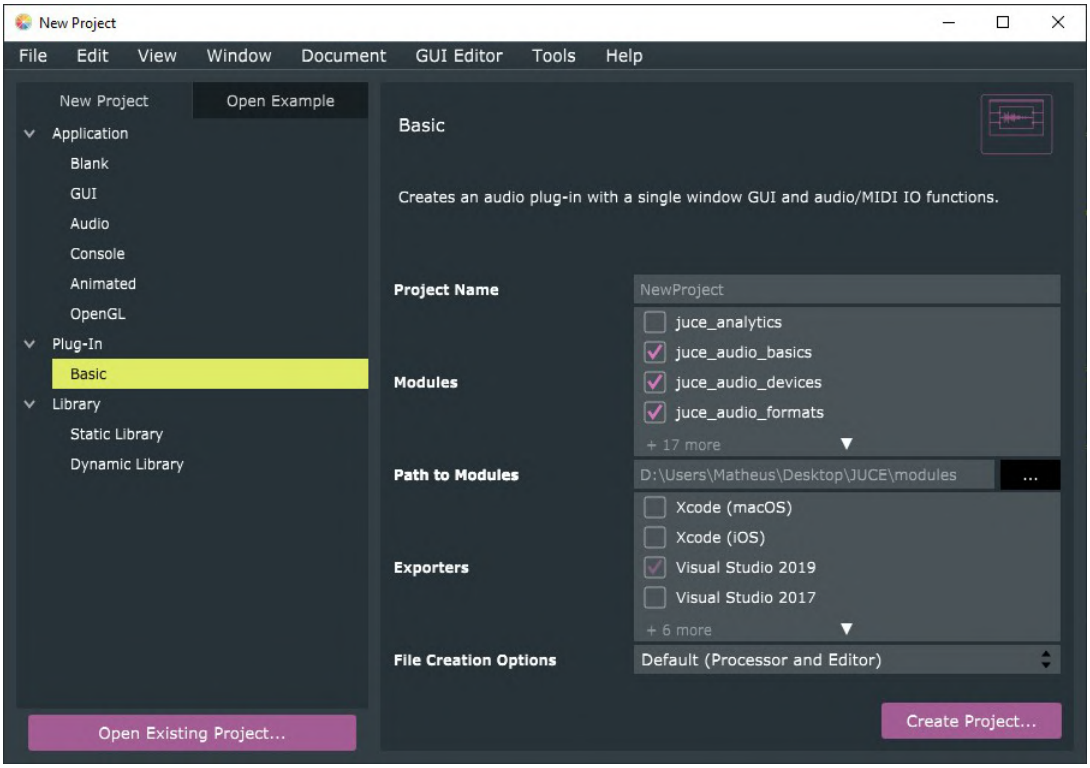
Segundo Robinson (2013), existem várias estruturas disponíveis para o desenvolvimento de softwares voltados ao áudio, mas o JUCE combina consistência, facilidade de uso e uma gama ampla de funcionalidades. Ele o incentiva a escrever códigos consistentes e é particularmente otimizado para implementação de interfaces gráficas complexas personalizadas e processamento de áudio/MIDI.

O JUCE é composto por uma ampla gama de classes que resolvem problemas comuns encontrados durante o desenvolvimento de *softwares*. Isso inclui manipulação de gráficos, som, interação do usuário, redes e assim por diante. Devido ao seu nível de suporte de áudio, ele é popular para o desenvolvimento de aplicativos de áudio e *plug-ins* de áudio, mas isso não limita o seu uso apenas para estes fins.

Além das classes, também é fornecido o Projucer, uma IDE que permite criar e gerenciar os projetos desenvolvidos com o JUCE, como pode ser visto na Figura 3.1. Ele gera automaticamente uma coleção de arquivos de projeto para as IDEs suportadas, permitindo que o projeto seja compilado nativamente em cada uma delas, as configurações de exportação podem ser definidas quando o projeto é criado e também podem ser modificadas posteriormente. O Projucer também possui integrados um editor de código, que pode ser visto na Figura 3.2, e um editor de

interface com o usuário, que pode ser visto na Figura 3.3, isso facilita na criação dos *plug-ins* e aplicativos caso o desenvolvedor não deseje utilizar outras ferramentas para esse fim.

Figura 3.1 – Interface para criação de um projeto no Projucer.



Fonte: Do autor (2021)

Alguns motivos que levaram à escolha do *framework* JUCE para o desenvolvimento deste trabalho:

a) é gratuito para Estudantes e Educadores;

b) tem uma comunidade ativa e fóruns de discussão para sanar dúvidas dos usuários, além de uma documentação vasta, didática e bem estruturada;

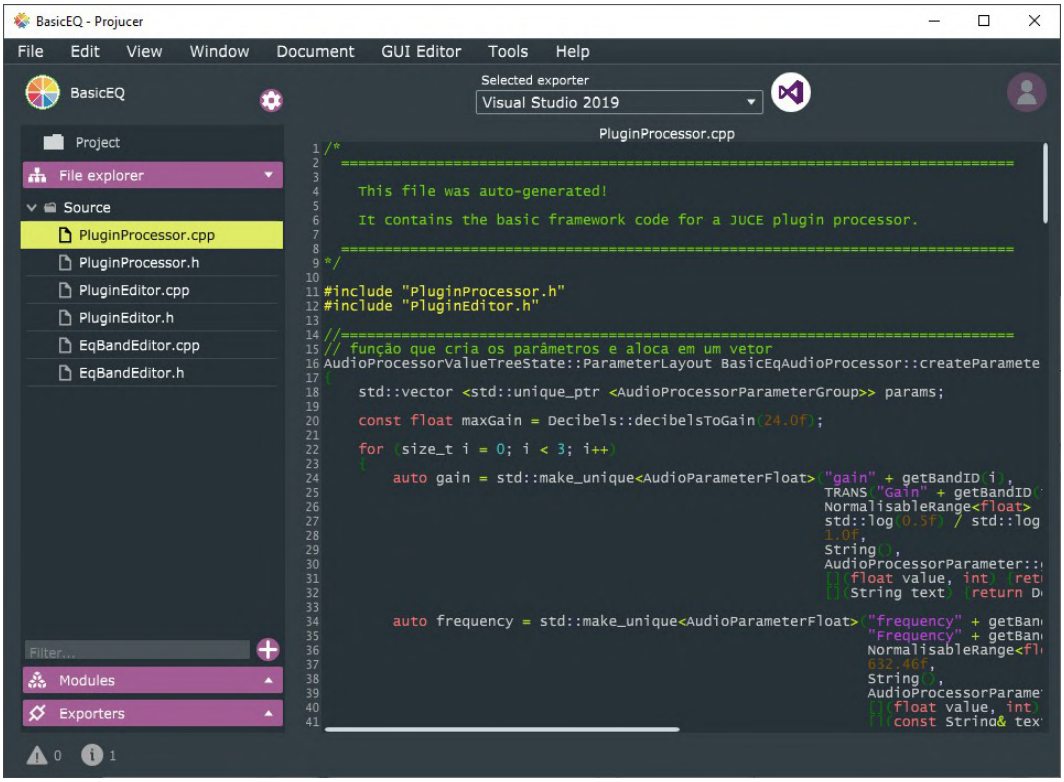
c) há materiais disponibilizados pelo próprio time de desenvolvedores e por outros usuários na internet para quem deseja aprender a utilizar;

d) tem um time de desenvolvimento ativo, entregando novas funções aos usuários a cada atualização.

3.1.2 Microsoft Visual Studio

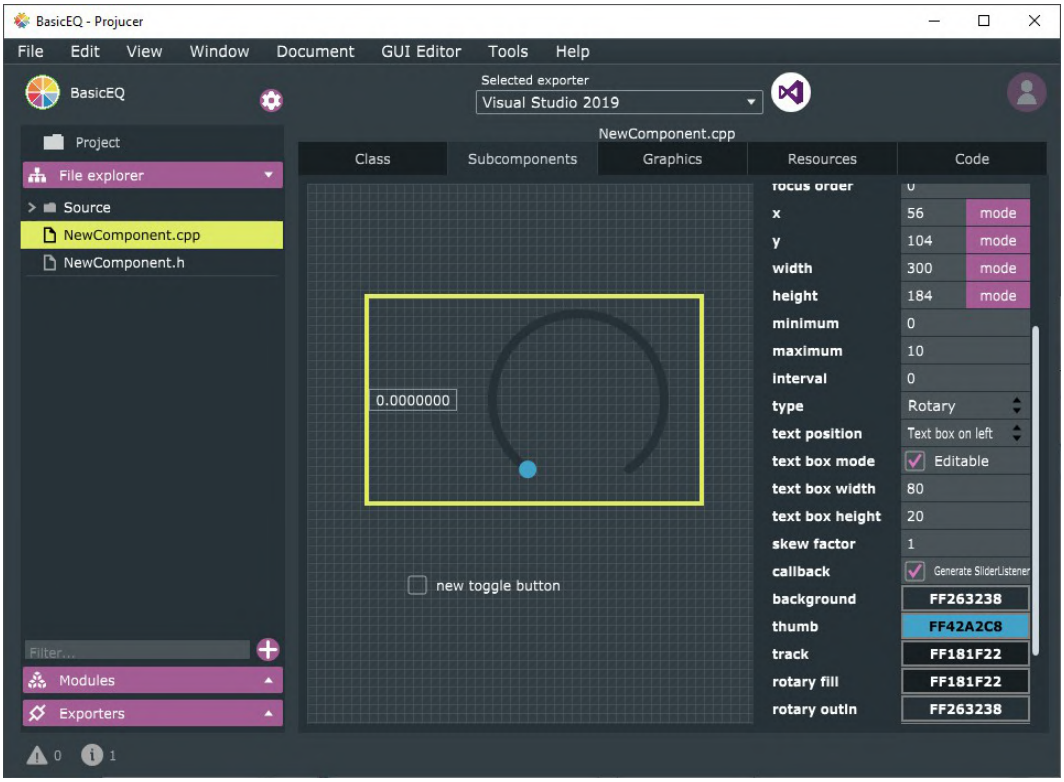
O Microsoft Visual Studio é um ambiente de desenvolvimento integrado (IDE – *Integrated Development Environment*) da empresa Microsoft. Pode ser utilizado para desenvolvimento

Figura 3.2 – Editor de código do Projucer.



Fonte: Do autor (2021)

Figura 3.3 – Editor de interface com o usuário do Projucer.



Fonte: Do autor (2021)

de softwares em diferentes linguagens de programação, mas é especialmente dedicado às linguagens .NET Framework, Visual Basic (VB), C, C++, C# (C Sharp) e F# (F Sharp). Também pode ser utilizado para desenvolvimento para plataformas *web*, como *websites*, aplicativos *web*, serviços *web* e aplicativos móveis, usando a plataforma do ASP.NET. Outras linguagens suportadas por esse ambiente são Python, JavaScript, TpeScript, R, entre outras.

É um *software* que reúne características e ferramentas de apoio ao desenvolvimento com o objetivo de agilizar este processo, que, segundo Lee (2021), pode ser usado para criar, editar, depurar códigos e publicar o aplicativo desenvolvido. O Visual Studio inclui compiladores, ferramentas de conclusão de código, designers gráficos e muitos outros recursos para facilitar o processo de desenvolvimento de *software*, além do editor e do depurador padrão que a maioria dos IDEs fornece.

Há três edições do Visual Studio disponíveis, Community, Professional e Enterprise, destinadas a tipos de usuários diferentes. Enquanto as edições Professional e Enterprise são versões pagas e voltadas para uso empresarial, a edição Community é gratuita e voltada para estudantes, colaboradores de *software* livre e para uso pessoal.

Neste trabalho o Microsoft Visual Studio Community foi a ferramenta utilizada para desenvolver os *plug-ins* voltados para o sistema operacional Windows, da própria Microsoft. Ele foi escolhido por facilitar o desenvolvimento em C++ e pela sua integração com o Projucer. Principalmente por depurar os códigos escritos e construir os *plug-ins* utilizando o JUCE na própria IDE, permitindo a visualização desses *plug-ins* antes de testar nas plataformas de produção musical de destino, para as quais eles foram desenvolvidos.

3.1.3 Xcode

O Xcode é uma IDE e *software* livre da empresa Apple. Assim como o Visual Studio, pode ser utilizado para desenvolvimento de *softwares* em diferentes linguagens de programação, mas é especialmente dedicado às linguagens Swift e C/C++/Objective-C.

Segundo a própria página, Xcode (2021), consiste em um conjunto de ferramentas que os desenvolvedores podem usar para construir aplicativos para as plataformas Apple, com recursos para gerenciar todo o fluxo de trabalho de desenvolvimento, desde a criação do aplicativo até testagem, otimização e publicação na loja de aplicativos App Store, da Apple.

O Xcode é um *software* poderoso para desenvolvimento de aplicativos e pode ser obtido gratuitamente para usuários do sistema operacional da Apple, o MacOS, diretamente da loja de aplicativos App Store.

Neste trabalho o Xcode foi a ferramenta utilizada para desenvolver os *plug-ins* voltados para o sistema operacional MacOs, da própria Apple. Ele foi escolhido pelos mesmos motivos do Visual Studio em relação ao Windows, a utilização desta IDE facilita o desenvolvimento em C++ nesse sistema operacional e tem integração com o Projucer.

3.1.4 REAPER

O REAPER é uma estação de trabalho de áudio digital (DAW – *Digital Audio Workstation*), um *software* voltado para produção de áudio digital, que oferece um conjunto completo de ferramentas de gravação, edição, processamento, mixagem e masterização de áudio. Suporta uma vasta gama de *hardware*, formatos digitais de áudio e *plug-ins*, permitindo a reprodução dos áudios e alterações em tempo real nos processamentos. Há uma única versão desse *software*, com recursos completos e sem limitações artificiais, e ela pode ser avaliada gratuitamente por um período de sessenta dias.

Segundo a página do fabricante, Reaper (2021), o REAPER faz o processamento de áudio interno em 64 bits, possibilita a importação, gravação e renderização em muitos formatos de mídia, profundidade de bits e taxa de amostragem. Possui suporte a *hardware* e *software* de Interface Digital de Instrumentos Musicais (MIDI - *Musical Instrument Digital Interface*), *plug-ins* de terceiros e instrumentos virtuais nos formatos VST, VST3, LV2, AU, DX e JS. A DAW também suporta automação, modulação, agrupamento, VCA, *surround*, macros, OSC, *scripts*, superfícies de controle, *skins* e *layouts* personalizados. Usando o REAPER com um PC ou Mac, é possível importar áudio e MIDI, sintetizar, amostrar, compor, arranjar, editar, mixar e masterizar músicas ou quaisquer outros projetos de áudio. Adicionando uma interface de áudio e um microfone ou instrumento, também é possível realizar gravações diretamente na plataforma.

Neste trabalho, o REAPER foi a DAW escolhida para testar os *plug-ins*. Apesar de não ser a principal ferramenta utilizada no mercado de produção musical, já está consolidada como uma alternativa boa e mais barata, além disso permite a realização do teste gratuito por sessenta dias, que foram suficientes para realizar os testes necessários para finalizar os *plug-ins*

desenvolvidos. Essa DAW poderia ser substituída por qualquer outra DAW que tenha suporte para *plug-ins* VST3 ou AU, como a DAW gratuita Waveform Free, da empresa Tracktion.

3.1.5 Freesound

Freesound, Figura é um site que disponibiliza um banco de dados colaborativo com trechos de áudio, samples, gravações e outros tipos de sons sob licenças Creative Commons, que oferecem permissão para usar um trabalho de acordo com a lei de direitos autorais.. Segundo o próprio site, About... (2021), eles também pretendem criar uma base de dados aberta de sons, que poderá ser usada para pesquisas científicas e ser integrada em aplicações de terceiros. Este site foi criado em 2005 na Universitat Pompeu Fabra, em Barcelona, Espanha, e é atualmente mantido pelo time Freesound.

Este site foi utilizado para encontrar amostras de áudio que pudessem ser utilizadas para realizar os testes nos *plug-ins* desenvolvidos. Foi escolhido pela sua vasta quantidade de sons disponíveis e principalmente pela liberdade de utilização que a licença Creative Commons proporciona.

3.1.6 Google Forms

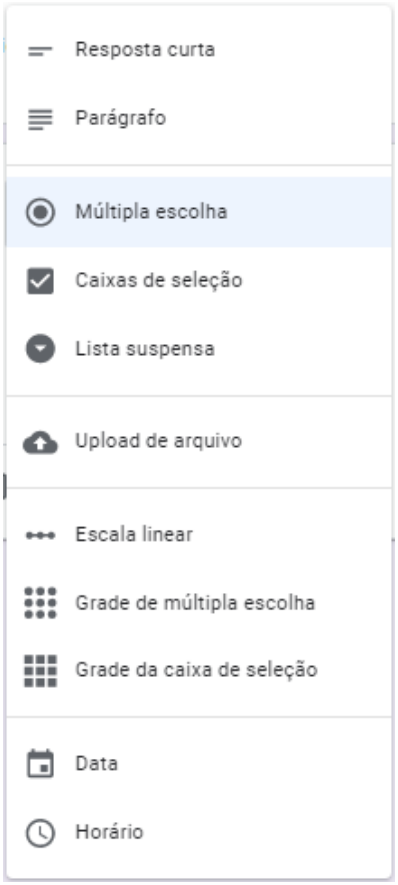
O Google Forms é uma ferramenta de criação de formulários da empresa Google que permite a criação de formulários personalizados e a disponibilização deste a outras pessoas, para coletar as respostas. Essa plataforma disponibiliza uma ferramenta simples, mas eficaz, com possibilidade de dividir o formulário em seções, personalizar com imagens diferentes tipos de resposta possíveis e exportar as respostas em planilhas. A Figura 3.4 mostra a caixa de seleção dos tipos de perguntas disponíveis.

Essa ferramenta foi utilizada para coletar *feedback* de profissionais do mercado de áudio sobre os *plug-ins* que foram desenvolvidos no trabalho. Foi escolhida pela facilidade de uso e por ser fornecida aos alunos da UFLA, devido à utilização da plataforma Google pela universidade.

3.1.7 Google Drive

O Google Drive é uma ferramenta de armazenamento em nuvem da empresa Google, no qual o usuário pode armazenar arquivos e compartilhá-los com outras pessoas pela internet.

Figura 3.4 – Tipos de perguntas do Google Forms.



Fonte: Do autor (2021)

Além de armazenar esses arquivos, a ferramenta também consegue realizar a leitura de alguns formatos, possibilitando que o usuário reproduza sem a necessidade de fazer o *download*.

3.2 Um projeto JUCE

3.2.1 Criando um novo projeto

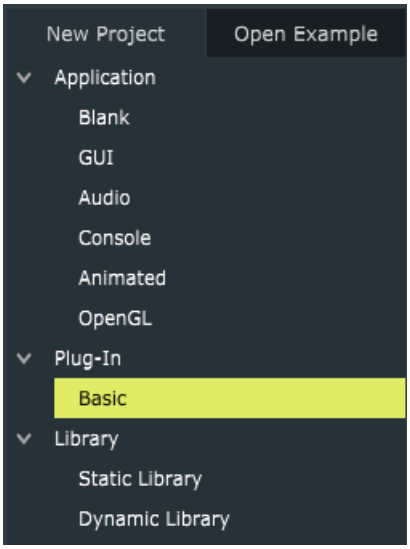
Para iniciar um projeto no JUCE, basta executar o Projucer que é aberta a janela para criação de um novo projeto, como mostra a Figura 3.5.

Abaixo o que cada uma desses tipos de projeto nos oferece por padrão:

a) Application:

- Blank: cria uma interface gráfica do usuário (GUI – *Graphical User Interface*) em branco;
- GUI: cria uma GUI com um único componente de janela em branco;

Figura 3.5 – Escolhendo o tipo de projeto no Projucer.



Fonte: Do autor (2021)

- Audio: cria uma aplicação GUI com um único componente de janela e funções de entrada e saída de áudio e MIDI;
- Console: cria uma aplicação por linhas de comando, sem suporte a GUI;
- Animated: cria uma aplicação GUI que desenha um *display* gráfico animado;
- OpenGL: cria uma aplicação JUCE em branco com um único componente de janela, que suporta recursos OpenGL;

b) *plug-in*:

- Basic: cria um *plug-in* de áudio com uma única janela GUI e funções de entrada e saída de áudio e MIDI;

c) Library:

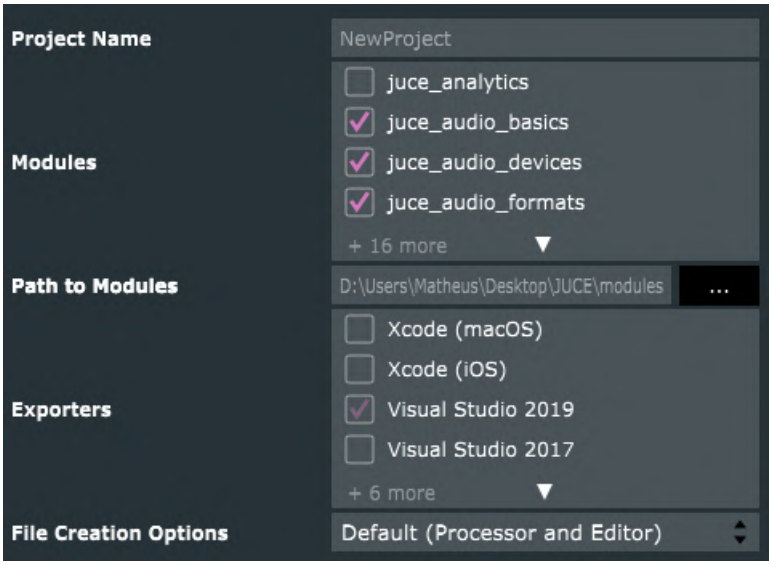
- Static library: cria uma biblioteca estática;
- Dynamic library: cria uma biblioteca dinâmica.

Como o objetivo desse trabalho é o desenvolvimento de *plug-ins*, escolheu-se o projeto do tipo Basic, do grupo *plug-in*. A partir disso o Projucer nos dá algumas opções para configurar o projeto, conforme Figura 3.6.

Essas opções nos permitem ajustar:

a) Project Name: o nome do projeto;

Figura 3.6 – Configurações de criação do projeto no Projucer.



Fonte: Do autor (2021)

b) Modules: quais os módulos JUCE devem ser incluídos no projeto;

c) Path to Modules: qual o caminho no qual a pasta de módulos está localizada no computador;

d) Exporters: para quais IDEs o projeto deve ser exportado;

e) File Creation Options: quais arquivos padrões do JUCE deverão ser criados com o projeto.

Para desenvolvimento dos *plug-ins* deste trabalho, inicialmente foram utilizadas as configurações que vieram pré-carregadas pelo Projucer, alterando somente o nome do projeto de acordo com cada *plug-in* desenvolvido. As configurações padrão de módulos JUCE para criação de um *plug-in* são: juce_audio_basics, juce_audio_devices, juce_audio_formats, juce_audio_plugin_client, juce_audio_processors, juce_audio_utils, juce_core, juce_data_structures, juce_events, juce_graphics, juce_gui_basics, juce_gui_extra.

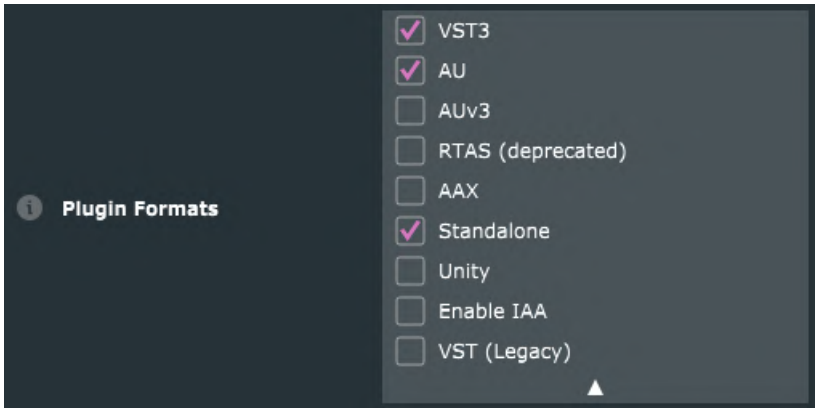
Ao clicar em criar projeto é permitido escolher o caminho no qual se deseja salvar o projeto, escolhe-se o caminho e então o projeto é criado.

3.2.2 Configurando o projeto

Ao abrir o projeto, algumas configurações devem ser alteradas. Clicando no símbolo da engrenagem, no painel esquerdo ao lado do símbolo do JUCE , são mostradas as configurações

adicionais. Aqui é importante definir quais formatos de *plug-ins* serão exportados ao compilar o projeto, isso pode ser feito no campo plug-in Formats, mostrado na Figura 3.7; para a plataforma Windows foi utilizado o formato VST3, para a plataforma Mac OS foram utilizados os formatos AU e VST3. Além destes formatos de *plug-in*, que são lidos pelas DAWs, também é importante selecionar o formato Standalone, que permite a execução do *plug-in* diretamente na IDE escolhida.

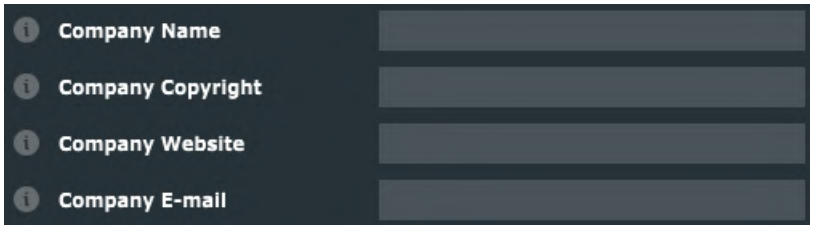
Figura 3.7 – Escolhendo os formatos dos *plug-ins*.



Fonte: Do autor (2021)

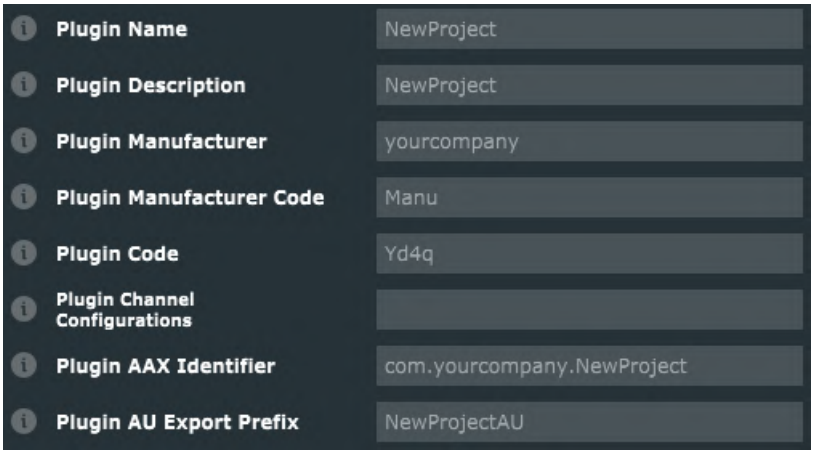
Ao idealizar o trabalho, também era desejada a utilização do formato AAX, padrão da principal DAW do mercado, o Pro Tools. Porém, para exportar um *plug-in* neste formato é necessária uma assinatura digital para cada *plug-in*. Essa assinatura é emitida pela empresa Avid, criadora da DAW e do formato, e não foi possível de ser obtida. Outras configurações importantes de serem alteradas são referentes à classificação quanto à categoria dos *plug-ins*. Cada *plug-in* terá uma categoria específica e ela deve ser definida para cada formato de exportação. Como os formatos utilizados foram AU e VST3, os campos nos quais deve ser escolhida a categoria são plug-in AU Main Type e plug-in VST3 Category. Caso seja desejado, também podem ser preenchidos os campos com as informações referentes aos desenvolvedores e ao *plug-in*, conforme mostram as Figuras 3.8 e 3.9.

Figura 3.8 – Informações sobre o *plug-in*.



Fonte: Do autor (2021)

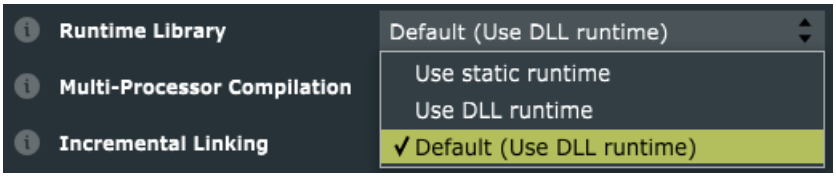
Figura 3.9 – Outras informações sobre o *plug-in*.



Fonte: Do autor (2021)

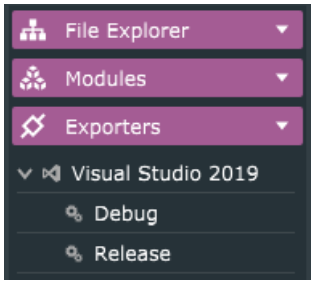
Também deve ser alterada uma configuração referente à biblioteca de tempo de execução utilizada na exportação dos *plug-ins* pelo Visual Studio. O campo Runtime Library, Figura 3.10, deve ser alterado de Use DLL runtime para Use static runtime. Isso é feito para evitar que os *plug-ins* exportados não funcionem em computadores que não têm o ambiente configurado corretamente para tempo de execução por DLL. Essa configuração deve ser alterada tanto na opção Debug quanto na opção Release referentes ao Visual Studio e pode ser acessada a partir do painel lateral esquerdo, na aba Exporters, conforme Figura 3.11.

Figura 3.10 – Alterando o campo Runtime Library.



Fonte: Do autor (2021)

Figura 3.11 – Aba Exporters.

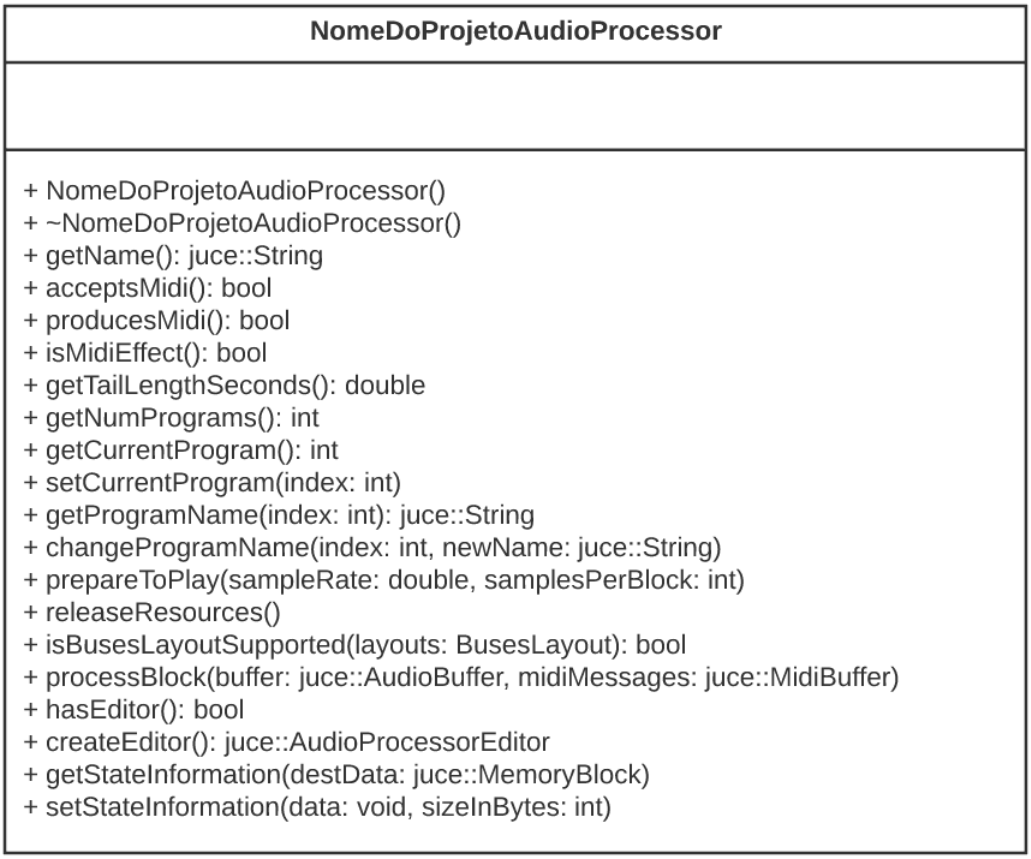


Fonte: Do autor (2021)

3.2.3 Arquivos criados pelo Projucer

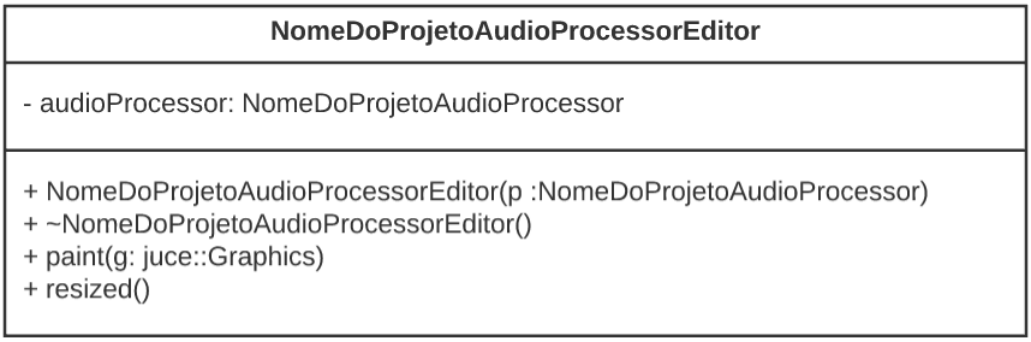
Ao escolher o projeto do tipo plugin: basic, o Projucer cria duas classes padrões com algumas funções predefinidas, para facilitar a criação desses softwares. A primeira classe é a NomeDoProjetoAudioProcessor, que tem as funções necessárias para realizar o processamento dos sinais de áudio a nível de sistema, conforme mostrado pela Figura 3.12. Essa classe é dividida entre os arquivos de *header* pluginProcessor.h e de implementação pluginProcessor.cpp. A segunda classe é a NomeDoProjetoAudioProcessorEditor, que tem as funções necessárias para construir a GUI do *plug-in*; inclusive os controles para os parâmetros dos *plug-ins*, que permitem ajustes em tempo real durante o processamento. A estrutura dessa classe pode ser vista na Figura 3.13, ela é dividida entre os arquivos de *header* pluginEditor.h e de implementação pluginEditor.cpp. Os arquivos criados pelo Projucer podem ser visualizados no painel lateral esquerdo, aba File Explorer, como mostra a Figura 3.14.

Figura 3.12 – NomeDoProjetoAudioProcessor.



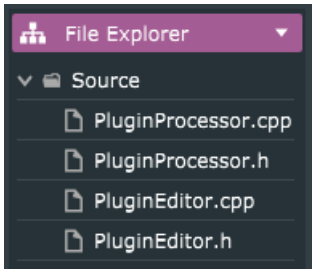
Fonte: Do autor (2021)

Figura 3.13 – NomeDoProjetoAudioProcessorEditor.



Fonte: Do autor (2021)

Figura 3.14 – Arquivos criados pelo Projucer.



Fonte: Do autor (2021)

3.2.3.1 A classe NomeDoProjetoAudioProcessor

Dentre os diversos métodos criados para a classe NomeDoProjetoAudioProcessor, alguns merecem atenção especial, pois serão alterados na maioria dos *plug-ins* que se deseje implementar. São eles:

a) NomeDoProjetoAudioProcessor: este método é o construtor da classe;

b) prepareToPlay: este método é executado apenas uma vez, antes que as iterações de processamento de áudio propriamente ditas realizadas comecem a ser executadas. Nele devem ser inseridas quaisquer inicializações necessárias antes da reprodução;

c) processBlock: este é o método no qual o processamento dos sinais de áudio é realizado. Todas as funções que realizam as modificações no sinal de entrada devem ser executadas dentro deste método;

d) getStateInformation: este método é usado para armazenar os parâmetros do *plug-in* em um bloco de memória;

e) setStateInformation: este método é usado para restaurar os parâmetros do *plug-in* a partir do bloco de memória no qual eles foram armazenados.

3.2.3.2 A classe NomeDoProjetoAudioProcessorEditor

Aqui provavelmente todos os métodos, exceto o destrutor da classe, serão utilizados ao desenvolver um *plug-in*, mas eles são poucos:

a) NomeDoProjetoAudioProcessorEditor: o construtor da classe. Importante notar que o este construtor recebe como parâmetro um ponteiro para a classe NomeDoProjetoAudioProcessor, isso faz com que seja possível controlar os parâmetros implementados no processamento dos *plug-ins* por meio da GUI criada. Os elementos gráficos criados para o *plug-in* devem ser iniciados nesse método;

b) paint: geralmente usada caso se deseje utilizar elementos gráficos como linhas, formas geométricas e textos na GUI;

c) resized: utilizada para determinar ou ajustar o tamanho dos elementos dentro da janela quando esta é redimensionada.

3.3 O desenvolvimento dos *plug-ins*

Decidiu-se por desenvolver quatro tipos de *plug-in* neste trabalho: um equalizador paramétrico de sete bandas, um de distorção, um *delay* e um reverberador. Poderiam ter sido feitos mais tipos que isso ou menos, porém entendeu-se que este número seria o suficiente para provar a possibilidade de criar uma grande variedade de *plug-ins* usando a mesma metodologia aplicada, devido à praticidade e versatilidade do *Framework* JUCE.

Para torná-los atrativos, optou-se por usar a criatividade em busca de definir nomes mais comerciais. Os nomes escolhidos foram:

a) DigiQ 7B: o equalizador foi nomeado dessa forma por ser digital (Digi), ser paramétrico (referente ao parâmetro Q) e ter sete bandas (7B);

b) BuzzTortion: o *plug-in* de distorção foi nomeado dessa forma para referir-se ao zumbido de uma abelha (*buzz* em tradução para inglês) e à terminação da palavra distorção em inglês (*distortion*);

c) BigLoop; o *delay* foi nomeado dessa forma para referir-se a um grande ciclo (*big loop* em tradução para o inglês), remetendo ao *buffer* circular utilizado;

d) WonderVerb: o reverberador foi nomeado dessa forma, referindo se à palavra maravilhoso (*wonderful* em tradução para o inglês) e reverberação (*reverb* ema tradução para o inglês).

3.3.1 DigiQ 7B

Para implementar o equalizador, utilizou-se o módulo de DSP do JUCE, então foi necessário adicionar mais este módulo no projeto. Para adicionar um novo módulo a um projeto já criado deve-se acessar a aba Modules, no painel esquerdo do Projucer, clicar no ícone com o símbolo de adição na parte inferior da aba e acessar pelo menu mostrado a opção Add a module, depois Global JUCE modules path e selecionar o módulo juce_dsp.

3.3.1.1 A classe DigiQ7BAudioProcessor

3.3.1.1.1 Atributos da classe DigiQ7BAudioProcessor

O módulo DSP fornece diversas classes e estruturas muito úteis para se trabalhar com o processamento de sinais, evitando que o desenvolvedor tenha que se preocupar com as manipulações dos dados a nível de sistema. Para o equalizador em questão foi utilizada a classe juce::dsp::IIR::Filter, que permite a implementação de um filtro IIR.

Para construir o filtro IIR utilizado é necessário passar como parâmetros os coeficientes que serão utilizados em sua função, que são os parâmetros já apresentados na Equação 2.6. O JUCE possui uma *struct* específica para armazenar esses coeficientes, que é a juce::dsp::IIR::Coefficients.

Também foi utilizada a *struct* juce::dsp::ProcessorDuplicator, que multiplica um processador mono pelo número de canais que serão utilizados, transformando-o em um processador multicanal. Essa *struct* recebe como parâmetros um processador, que nesse caso será o filtro IIR, e o estado do processador, que nesse caso são os coeficientes.

Como o equalizador tem sete bandas, tiveram que ser criados sete filtros IIR. Para facilitar a manipulação de muitos processadores de sinal, o módulo DSP disponibiliza a classe juce::dsp::ProcessorChain, que agrupa todos os processadores necessários em uma só estrutura, mas permitindo que posteriormente eles sejam acessados de forma individual. A declaração desses atributos é feita no arquivo pluginProcessor.h, dentro da seção do modificador private.

Nessa seção também foram declaradas duas variáveis, o inteiro eqBandIndex, usado para identificação da banda do equalizador, e o ponto flutuante lastSampleRate, usado para armaze-

nar a frequência de amostragem utilizada. Além destes, também foi declarado o objeto apvts, da classe juce::AudioProcessorValueTreeState. O uso de todos esses atributos será explicado mais adiante. A declaração dos atributos mencionados está ilustrada na Figura 3.15.

Figura 3.15 – Atributos da classe DigiQ7BAudioProcessor.

```
// árvore de valores usada para gerenciar todo o estado do AudioProcessor
juce::AudioProcessorValueTreeState apvts;

// multiplicando o processador pelo número de canais usados
using eqBand = juce::dsp::ProcessorDuplicator<juce::dsp::IIR::Filter<float>,
juce::dsp::IIR::Coefficients<float>>;

// alocando todos os DSP's para serem processados em conjunto
juce::dsp::ProcessorChain <eqBand, eqBand, eqBand, eqBand, eqBand, eqBand,
eqBand> processorChain;

// armazenará o índice da banda do equalizador
int eqBandIndex;

// armazenará frequência de amostragem
float lastSampleRate;
```

Fonte: Do autor (2021)

3.3.1.1.2 Implementando o método prepareToPlay com a classe juce_dsp

O método prepareToPlay recebe como parâmetros o ponto flutuante sampleRate, frequência de amostragem utilizada, e samplesPerBlock, a quantidade de amostras em um contidas em um bloco de processamento. Essas informações são passadas automaticamente pelo JUCE a esse método, elas são extraídas da DAW em que o *plug-in* está sendo usado.

Primeiramente atribui-se à variável lastSampleRate criada o valor de sampleRate, que foi recebido por parâmetro. Após isso utiliza-se a *struct* spec (juce::dsp::ProcessSpec), que recebe os valores de lastSampleRate, samplesPerBlock e o número de canais de saída do *plug-in*, que pode ser acessado utilizando o método getMainBusNumOutputChannels. O processo acima está ilustrado na Figura 3.16.

Ainda no método prepareToPlay, o processorChain deve ser inicializado. Primeiramente é utilizado o método reset para reinicializá-lo, evitando valores de lixo. Depois, é chamado o método updateFilter (esse método foi criado na classe DigiQ7BAudioProcessor e será explicado mais adiante) para atualizar os coeficientes do filtro IIR com os valores iniciais. Finalmente, é

Figura 3.16 – Preenchendo o ProcessSpec.

```
lastSampleRate = sampleRate;

// faz a comunicação entre \textit{plug-in} e módulo dsp
juce::dsp::ProcessSpec spec;

spec.sampleRate = lastSampleRate;
spec.maximumBlockSize = samplesPerBlock;
spec.numChannels = getMainBusNumOutputChannels();
```

Fonte: Do autor (2021)

utilizado o método prepare, que recebe como parâmetro o ProcessSpec criado, para inicializar o ProcessorChain. O processo está ilustrado na Figura 3.17.

Figura 3.17 – Inicializando o ProcessorChain.

```
processorChain.reset(); // evita valores de lixo
updateFilter();
// inicia o processor duplicator com as propriedades passadas
processorChain.prepare(spec);
```

Fonte: Do autor (2021)

Assim, foi finalizada a implementação do método prepareToPlay. Com exceção do método updateFilter, a implementação do método prepareToPlay foi utilizada como padrão para todos os *plug-ins* que utilizaram o módulo DSP e será suprimida nas explicações destes.

3.3.1.1.3 Implementando o método processBlock com a classe juce_dsp

O método processBlock recebe como parâmetros um ponteiro juce::AudioBuffer, que é o *buffer* de amostras de áudio, e um ponteiro juce::MidiBuffer, que é um *buffer* para instrumentos MIDI. Esse método já vem por padrão com algumas linhas de código implementadas, porém, como está sendo utilizado o juce_dsp, algumas dessas iterações não foram utilizadas e tiveram de ser apagadas, as linhas que foram removidas são as mostradas na Figura 3.18.

O *loop* for que foi excluído faria o processamento do sinal se implementássemos o *plug-in* sem utilizar o módulo juce_dsp. Utilizando o módulo, a implementação foi feita passando um bloco para o método process do ProcessorChain, por meio da struct juce::dsp::ProcessContextReplacing. Criou-se a variável block (juce::dsp::AudioBlock) para receber o ponteiro de *buffer*, chamou-se

Figura 3.18 – Iteração não necessária em *plug-ins* que utilizam o módulo juce_dsp.

```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
        auto* channelData = buffer.getWritePointer (channel);
}
```

Fonte: Do autor (2021)

novamente o método que atualiza os coeficientes do filtro e executou-se o método process. A implementação do método processBlock ficou conforme a Figura 3.19.

Figura 3.19 – Implementação do processBlock utilizando o módulo juce_dsp.

```
juce::ScopedNoDenormals noDenormals;
auto totalNumInputChannels = getTotalNumInputChannels();
auto totalNumOutputChannels = getTotalNumOutputChannels();

for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

// como está sendo utilizado o juce_dsp, não é necessário fazer iterações
// de processamento aqui, tudo é realizado pelo dsp
juce::dsp::AudioBlock<float> block(buffer); // o bloco de áudio é o buffer
updateFilter();
// processa o bloco por meio do dsp
processorChain.process(juce::dsp::ProcessContextReplacing<float>(block));
```

Fonte: Do autor (2021)

Assim, foi finalizada a implementação do método processBlock. Com exceção do método updateFilter, a implementação do método processBlock foi utilizada como padrão para todos os *plug-ins* que utilizaram o módulo DSP e será suprimida nas explicações destes.

3.3.1.1.4 O AudioProcessorValueTreeState

A classe juce::AudioProcessorValueTreeState contém uma árvore de valores que é utilizada para gerenciar o estado do AudioProcessor, os parâmetros do *plug-in* são armazenados em um objeto desse tipo. Dessa forma, esse objeto é utilizado tanto para fazer a comunicação entre os parâmetros do AudioProcessor e os controles da GUI do AudioProcessorEditor, quanto para armazenar e recuperar os parâmetros utilizando um bloco de memória por meio dos métodos getStateInformation e setStateInformation.

Esse objeto pode conter um conjunto de RangedAudioParameters ou um conjunto de AudioProcessorParameterGroups, que contém RangedAudioParameters. Esses conjuntos são identificados por meio de uma juce::String.

O atributo apvts é instanciado no construtor do AudioProcessor, a árvore de valores é definida pelo objeto ParameterLayout. Como esse código é muito grande para colocar no corpo do trabalho, poderá ser visto no Apêndice A com os comentários pertinentes para o entendimento. Basicamente foi criado um ParameterLayout para o *plug-in* utilizando o método createParameterLayout, implementado pelo autor, que é passado por parâmetro ao instanciar o apvts no construtor do AudioProcessor.

3.3.1.1.5 Os métodos getBandID e getBandFreq

São dois métodos simples, criados pelo autor. O getBandID retorna uma juce::String de identificação da banda do equalizador de acordo com um índice recebido por parâmetro. O getBandFreq tem funcionamento similar, mas retorna um ponto flutuante que representa um valor para a frequência de cada banda do *plug-in*. Esses métodos são mostrados na Figura 3.20 e 3.21.

Figura 3.20 – getBandID.

```
switch (index)
    {
case 0: return "Sub";
case 1: return "Low";
case 2: return "LowMid";
case 3: return "Mid";
case 4: return "MidHigh";
case 5: return "High";
case 6: return "Brigth";
default: break;
}
return "unknown
```

Fonte: Do autor (2021)

3.3.1.1.6 O método updateFilter

Esse método é utilizado para atualizar o estado dos filtros IIR do equalizador. Utilizando o método getRawParameterValue busca na apvts os valores dos parâmetros desejados. Com o método makePeakFilter, gera os coeficientes do filtro IIR a partir dos valores de frequência de

Figura 3.21 – getBandFreq.

```
switch (index)
{
case 0: return 50;
case 1: return 150;
case 2: return 400;
case 3: return 1000;
case 4: return 2500;
case 5: return 7500;
case 6: return 15000;
default: break;
}
return 0;
```

Fonte: Do autor (2021)

amostragem, frequência do filtro, fator Q e ganho. Atualiza o estado da banda do filtro, que pôde ser obtida por meio do ProcessorChain. Esse procedimento é repetido para as sete bandas do equalizador e pode ser visto na Figura 3.22.

3.3.1.1.7 Os métodos getStateInformation e setStateInformation

Esses métodos fazem a comunicação entre o objeto apvts e o bloco de memória onde os parâmetros do *plug-in* são salvos, para que em uma reabertura de uma sessão na DAW eles recuperem o mesmo valor que tinham quando a DAW foi fechada. A implementação desses métodos pode ser vista nas Figuras 3.23 e 3.24.

3.3.1.2 A classe EqBandEditor

Para esse *plug-in* em específico, foi decidido criar uma nova classe, a classe EqBandEditor é herdada da classe juce::component. O raciocínio por trás da criação desta classe veio da percepção de que a estrutura gráfica das bandas deveria manter um mesmo padrão. Sendo assim, esta classe foi utilizada para criar essa estrutura padrão de banda, que pode ser replicada quantas vezes for necessário na DigiQ7BAudioProcessorEditor, que é a classe principal da GUI.

3.3.1.2.1 Atributos da classe EqBandEditor

Cada banda do *plug-in* tem a necessidade de disponibilizar ao usuário três *sliders*, para permitir o controle dos três parâmetros de uma seção de filtro paramétrico. Então foram declarados três objetos da classe juce::Slider. Para fazer a comunicação entre a GUI e o AudioProcessor

Figura 3.22 – getBandFreq.

```
for (size_t i = 0; i < 7; i++)
{
        eqBandIndex = i;
        float freq = *apvts.getRawParameterValue("frequency" + getBandID(i));
        float qual = *apvts.getRawParameterValue("quality" + getBandID(i));
        float gain = *apvts.getRawParameterValue("gain" + getBandID(i));

        if (i == 0) *processorChain.get<0>().state =
                *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
                        lastSampleRate, freq, qual, gain);
        else if (i == 1) *processorChain.get<1>().state =
                *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
                        lastSampleRate, freq, qual, gain);
        else if (i == 2) *processorChain.get<2>().state =
                *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
                        lastSampleRate, freq, qual, gain);
        else if (i == 3) *processorChain.get<3>().state =
                *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
                        lastSampleRate, freq, qual, gain);
        else if (i == 4) *processorChain.get<4>().state =
                *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
                        lastSampleRate, freq, qual, gain);
        else if (i == 5) *processorChain.get<5>().state =
                *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
                        lastSampleRate, freq, qual, gain);
        else if (i == 6) *processorChain.get<6>().state =
                *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
                        lastSampleRate, freq, qual, gain);
}
```

Fonte: Do autor (2021)

Figura 3.23 – getStateInformation.

```
// o bloco de memória no qual o estado será escrito
juce::MemoryOutputStream stream(destData, false);
// escrevendo no bloco de memória
apvts.state.writeToStream(stream);
```

Fonte: Do autor (2021)

utilizando a classe juce::AudioProcessorValueTreeState foi criada uma juce::OwnedArray chamada sliderAttachments. Além desses atributos referentes aos *sliders*, também foram criados o atributo bandID para índice da banda e os atributos area, textPosition, textHeight e textWidth para realizar a manipulação dos espaços na GUI. Na Figura 3.25 estão as declarações dos atributos dessa classe. O restante da estrutura da classe foi baseada na classe AudioProcessorEditor.

Figura 3.24 – setStateInformation.

```
// lendo do bloco de memória
juce::ValueTree tree = juce::ValueTree::readFromData(data, sizeInBytes);
// passando os estados para a apvts
if (tree.isValid()) apvts.state = tree;
```

Fonte: Do autor (2021)

Figura 3.25 – Atributos private da classe EqBandEditor.

```
size_t bandID;

juce::Slider gainSlider;
juce::Slider freqSlider;
juce::Slider qSlider;

juce::OwnedArray<juce::AudioProcessorValueTreeState::
        SliderAttachment> sliderAttachments;

juce::Rectangle<int> area;
int textPosition;
int textHeight = 30;
int textWidth = 200;
```

Fonte: Do autor (2021)

3.3.1.2.2 O construtor EqBandEditor

O construtor da classe EqBandEditor, além de um AudioProcessor (parâmetro padrão dos AudioProcessorEditor), também recebe como parâmetro um size_t i, que identifica o índice da banda, o atributo bandID recebe o valor desse parâmetro.

No construtor também são implementados os métodos que possibilitam adicionar, exibir e associar os *sliders* aos parâmetros da apvts, seguindo o padrão da Figura 3.26.

3.3.1.2.3 Os métodos paint e resized

O método paint foi utilizado para adicionar elementos gráficos à GUI e o método resized para posicionar os *sliders*. Devido à extensão do métodos e facilidade para entendimento dos códigos, eles poderão ser visualizados no Apêndice A, com os comentários necessários para o entendimento.

Figura 3.26 – Adicionando, exibindo e associando o slider.

```
// adiciona o slider ao componente e torna visível
addAndMakeVisible(variavelDoSlider);
// define estilo do slider como rotatório
gainSlider.setSliderStyle(juce::Slider::Rotary);
// define a posição e tamanho da caixa de texto do slider
gainSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 80, 20);
// define a cor de fundo
gainSlider.setColour(juce::Slider::backgroundColourId, juce::Colour(0x00000000));
// define a cor do indicador da posição
gainSlider.setColour(juce::Slider::thumbColourId, juce::Colours::yellow);
// define a cor do contorno da caixa de texto
gainSlider.setColour(juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
// faz a associação com a apvts, utilizando o método getAPVTS,
        //implementado para retornar este objeto
sliderAttachments.add(new juce::AudioProcessorValueTreeState::SliderAttachment(
        processor.getAPVTS(), "gain" + processor.getBandID(i), gainSlider));
```

Fonte: Do autor (2021)

3.3.1.3 A classe DigiQ7BAudioProcessorEditor

A classe AudioProcessorEditor na maioria das vezes é implementada para definir os parâmetros dos elementos da GUI. Neste *plug-in* foi utilizada apenas para adicionar as bandas do equalizador à janela, já que que as definições visuais foram feitas na classe EqBandEditor.

Os atributos declarados para a classe foram os sete objetos da classer eqBandEditor, utilizados para implementar cada banda.

No construtor da classe, foi necessário instanciar cada banda, passando como parâmetro o AudioProcessor e o índice correspondente. Aqui também foi definido o tamanho inicial da janela e que ela seria redimensionável. Além disso também se fez necessário adicionar e tornar visível cada objeto da classe eqBandEditor. A implementação do construtor pode ser vista na Figura 3.27

No método paint toda a janela foi preenchida com apenas a cor preta. No método resized foi apenas definido um tamanho igual para cada banda na janela.

3.3.2 BuzzTortion

Para implementar o *plug-in* de distorção, também foi utilizado o módulo juce_dsp, como no DigiQ 7B. Os métodos prepareToPlay e processBlock diferem dos implementados no equa-

Figura 3.27 – Construtor do DigiQ7BAudioProcessorEditor.

```
DigiQ7BAudioProcessorEditor::DigiQ7BAudioProcessorEditor
        (DigiQ7BAudioProcessor& p)
        : AudioProcessorEditor(&p), audioProcessor(p),
        eqBand1(p, 0), eqBand2(p, 1), eqBand3(p, 2), eqBand4(p, 3),
        eqBand5(p, 4), eqBand6(p, 5), eqBand7(p, 6)
{
            setSize(1200, 400);
            setResizable(true, false);

           addAndMakeVisible(&eqBand1);
            addAndMakeVisible(&eqBand2);
            addAndMakeVisible(&eqBand3);
            addAndMakeVisible(&eqBand4);
            addAndMakeVisible(&eqBand5);
            addAndMakeVisible(&eqBand6);
            addAndMakeVisible(&eqBand7);
}
```

Fonte: Do autor (2021)

lizador apenas pela substituição do método updateFilter pelo método updateWaveshaper, que será mostrado mais adiante. O ParameterLayout segue a mesma ideia do implementado no equalizador, mas se adequando aos parâmetros necessários ao BuzzTortion. O construtor da classe BuzzTortionAudioProcessor e os métodos getStateInformation e setStateInformation são idênticos aos implementados no equalizador.

3.3.2.1 A classe BuzzTortionAudioProcessor

3.3.2.1.1 Atributos da classe BuzzTortionAudioProcessor

Para implementar o processador de distorção foi utilizada a classe juce::dsp::Waveshaper, que permite a implementação de um *waveshaper*. Essa classe recebe como parâmetro a função (std::function) que será aplicada ao sinal para gerar a distorção. Criou-se também um objeto juce::dsp::ProcessorChain, recebendo o *waveshaper* como parâmetro. Como no DigiQ 7B, também foram declarados o objeto apvts e o ponto flutuante lastSampleRate. Na Figura 3.28 está ilustrado o código utilizado para declarar os atributos.

Figura 3.28 – Atributos da classe BuzzTortionAudioProcessor.

```
// árvore de valores usada para gerenciar todo o estado do AudioProcessor
juce::AudioProcessorValueTreeState apvts;
// alocando todos o DSP no ProcessorChain
juce::dsp::ProcessorChain<juce::dsp::WaveShaper<double,
        std::function<double(double)>>> processorChain;
// armazenará frequência de amostragem
float lastSampleRate;
```

Fonte: Do autor (2021)

3.3.2.1.2 O método updateWaveshaper

Esse método é utilizado para atualizar o estado do *waveshaper*. Primeiro é realizada a busca dos valores dos parâmetros na apvts. Com o método functionToUse foi determinada a função utilizada pelo *waveshaping* para gerar a distorção, que foi dividida em duas partes, mostradas nas Equações 3.1 e 3.2 abaixo:

$$distortedSignal = \frac{inputGain \times arctan(drive \times in)}{\pi \times (\frac{drive}{2} + 0,5)^{0,4}} \tag{3.1}$$

$$out = (distortedSignal \times blend + in \times (1 - blend)) \times outputGain \tag{3.2}$$

onde *in* é o sinal de entrada, *inputGain* é o parâmetro que controla o ganho de entrada, *drive* é o parâmetro que controla a quantidade de distorção adicionada, *blend* é o parâmetro que faz o ajuste da proporção entre sinal distorcido e original que compõe o sinal de saída e *outputGain* é o parâmetro que controla o ganho de saída.

A função arco tangente, Figura 2.21 foi utilizada para dar a característica de não linearidade ao processamento. O denominador da equação foi utilizado para compensar o ganho obtido ao se alterar o parâmetro *drive*, já que esse parâmetro altera diretamente o ganho do sinal de entrada. Pode-se notar na Equação 3.2 que a soma proporcional entre os sinais original e distorcido é sempre um. A implementação do método pode ser visualizada na Figura 3.29.

3.3.2.2 A classe BuzzTortionAudioProcessorEditor

Diferentemente do DigiQ 7B, o BuzzTortion não possui estruturas repetitivas na GUI, portanto todo o código da GUI foi implementado diretamente nesta classe.

Figura 3.29 – Código do método updateWaveshaper.

```
// buscando os valores na apvts
float inputGain = *apvts.getRawParameterValue("inputGain");
float drive = *apvts.getRawParameterValue("drive");
float blend = *apvts.getRawParameterValue("blend");
float outputGain = *apvts.getRawParameterValue("outputGain");

// atualizando o estado do waveshaper
auto& waveshaper = processorChain.get<0>();
waveshaper.functionToUse =
        [inputGain, drive, blend, outputGain](double in)
{
        // função que aplica a distorção no sinal original
        float distortedSignal = inputGain * atan(drive * in) /
                (juce::float_Pi * std::pow(drive + .5f, .4f));
        // função que mescla o sinal distorcido com o original
        float out = ((distortedSignal * blend) + (in * (1.f - blend))) *
                outputGain;
        return out;
}
```

Fonte: Do autor (2021)

3.3.2.2.1 Atributos da classe BuzzTortionAudioProcessorEditor

Como o *plug-in* tem quatro parâmetros ajustáveis, foram declarados quatro juce::Slider. Também foram declarados os atributos sliderAttachments, area, textPosition, textHeight e textWidth, que possuem os mesmos objetivos já mencionados no equalizador.

3.3.2.2.2 O construtor e os métodos paint e resized

No construtor, foi implementado o código para adicionar, exibir e associar os *sliders* ao apvts. O método paint foi utilizado para adicionar elementos gráficos à GUI e o método resized para posicionar os *sliders*. Devido a esse tipo de implementação já ter sido explicado, os códigos poderão ser visualizados no Apêndice B.

3.3.3 WonderVerb

Para implementar o *plug-in* de reverberação, também foi utilizado o módulo juce_dsp, como no DigiQ 7B e BuzzTortion. Os métodos prepareToPlay e processBlock diferem dos implementados apenas pelo método de atualização do estado do processador, que aqui se chama updateReverb e será mostrado mais adiante. O ParameterLayout segue a mesma ideia dos ante-

riores, se adequando aos parâmetros necessários ao BuzzTortion. O construtor da classe BuzzTortionAudioProcessor e os métodos getStateInformation e setStateInformation são idênticos aos implementados nos dois *plug-ins* anteriores.

3.3.3.1 A classe WonderVerbAudioProcessor

3.3.3.1.1 Atributos da classe WonderVerbAudioProcessor

Para implementar o processador de distorção foi utilizada a classe juce::dsp::Reverb, que permite a implementação de um reverberador. Um objeto da classe Reverb não recebe parâmetros em seu construtor, os parâmetros são passados posteriormente por meio do método setParameters e isso será mostrado adiante. Criou-se também um objeto juce::dsp::ProcessorChain, recebendo o reverberador como parâmetro. Como nos *plug-ins* anteriores, também foram declarados o objeto apvts e o ponto flutuante lastSampleRate. Na Figura 3.30 está ilustrado o código utilizado para declarar os atributos.

Figura 3.30 – Atributos da classe WonderVerbAudioProcessor.

```
// árvore de valores usada para gerenciar todo o estado do AudioProcessor
juce::AudioProcessorValueTreeState apvts;
// alocando o processador no ProcessorChain
juce::dsp::ProcessorChain<juce::dsp::Reverb> processorChain;
// armazenará frequência de amostragem
float lastSampleRate;
```

Fonte: Do autor (2021)

3.3.3.1.2 O método updateReverb

Esse método é utilizado para atualizar o estado do reverberador. Primeiro é realizada a busca dos valores dos parâmetros na apvts. Depois os valores são passados ao reverberador por meio do método setParameters, que recebe uma struct com os parâmetros, como ilustrado na Figura 3.31.

Os parâmetros que o reverberador recebe são aqueles referentes à abordagem perceptiva, são eles:

a) roomSize: o tamanho da sala simulada;

b) damping: a absorção das altas frequências na sala simulada;

Figura 3.31 – Código do método updateReverb.

```
// buscando os valores na apvts
float roomSize = *apvts.getRawParameterValue("roomSize");
float damping = *apvts.getRawParameterValue("damping");
float wetLevel = *apvts.getRawParameterValue("wetLevel");
float dryLevel = *apvts.getRawParameterValue("dryLevel");
float width = *apvts.getRawParameterValue("width");
float freezeMode = 0;

// atualizando o estado do reverberador
auto& reverb = processorChain.get<0>();
reverb.setParameters({ roomSize, damping, wetLevel, dryLevel, width, freezeMode });
```

Fonte: Do autor (2021)

c) wetLevel: o nível de saída do sinal processado;

d) dryLevel: o nível de saída do sinal original;

e) width: a amplitude da imagem estéreo;

f) freezeMode: esse parâmetro insere sustentação infinita à reverberação, foi definido o valor zero, pois foi escolhido não utilizar este efeito.

3.3.3.2 A classe WonderVerbAudioProcessorEditor

A implementação dessa classe foi muito semelhante à classe equivalente do BuzzTortion. Foram declarados cinco juce::Slider para os parâmetros e novamente os atributos sliderAttachments, area, textPosition, textHeight e textWidth. No construtor, foi implementado o código para adicionar, exibir e associar os *sliders* ao apvts. O método paint foi utilizado para adicionar elementos gráficos à GUI e o método resized para posicionar os *sliders*. Devido a esse tipo de implementação já ter sido explicado, os códigos poderão ser visualizados no Apêndice C.

3.3.4 BigLoop

Diferentemente dos *plug-ins* anteriores, para implementação do *delay* não foi utilizado o módulo juce_dsp, portanto as iterações foram feitas no próprio código do *plug-in*. O ParameterLayout segue a mesma ideia dos anteriores, se adequando aos parâmetros necessários ao

BigLoop. O construtor da classe BuzzTortionAudioProcessor e os métodos getStateInformation e setStateInformation são idênticos aos implementados nos *plug-ins* anteriores.

3.3.4.1 A classe BigLoopAudioProcessor

3.3.4.1.1 Atributos da classe BigLoopAudioProcessor

Para implementar um *delay*, foi necessário criar um *buffer* extra destinado a armazenar os valores para serem lidos com atraso, isso foi feito utilizando a classe juce::AudioBuffer. Além do *buffer*, os inteiros writePosition e lastDelayTime e os pontos flutuantes lastInputGain e lastFeedBackGain foram declarados. Aqui também foi usado o atributo lastSampleRate, já mencionado nos outros *plug-ins*. A Figura 3.32 mostra a declaração desses atributos. O BigLoop também utiliza o objeto apvts para lidar com os parâmetros do *plug-in*.

Figura 3.32 – Atributos da classe BigLoopAudioProcessor.

```
// árvore de valores usada para gerenciar todo o estado do AudioProcessor
juce::AudioProcessorValueTreeState apvts;
// delayBuffer para armazenar as amostras
juce::AudioBuffer<float> delayBuffer;

// armazenará frequência de amostragem
int lastSampleRate;
// posição onde será escrito nos delayBuffer
int writePosition;
// armazenará o tempo de delay
int lastDelayTime;
// armazenará o ganho de entrada
float lastInputGain;
// armazenará ganho de feedback
int lastFeedbackGain;
```

Fonte: Do autor (2021)

3.3.4.1.2 Implementando o método prepareToPlay do BigLoop

No método prepareToPlay, são inicializados os atributos lastSampleRate com a frequência de amostragem, writePosition com valor zero e os atributos lastDelayTime, lastInputGain e lastFeedbackGain com os valores lidos da apvts. Também é inicializado o delayBuffer com o método setSize, com um tamanho de dois segundos somados com dois *buffers* padrão por segurança; além disso utiliza-se o método clear para evitar valores de lixo no delayBuffer. Essa implementação pode ser vista na Figura 3.33.

Figura 3.33 – Implementando o prepareToPlay para o BigLoop.

```
lastSampleRate = sampleRate;
writePosition = 0;

// inicializa com os valores dos parâmetros do delay
lastDelayTime = *apvts.getRawParameterValue("delayTime");
lastInputGain = *apvts.getRawParameterValue("inputGain");
lastFeedbackGain = *apvts.getRawParameterValue("feedbackGain");

// tamanho do delayBuffer, 2 segundos + 2 buffers de segurança
const int delayBufferSize = 2 * (sampleRate + samplesPerBlock);
// define o tamanho do delayBuffer
delayBuffer.setSize(getTotalNumInputChannels(), (int)delayBufferSize);
//evita valores de lixo
delayBuffer.clear();
```

Fonte: Do autor (2021)

3.3.4.1.3 Implementando o método processBlock do BigLoop

No BigLoop, como não foi utilizada a classe juce_dsp, as iterações de processamento dos sinais são realizadas dentro do método processBlock. Foi utilizado um *loop* for para percorrer os canais de áudio e realizar o processamento em todos eles. Dentro desse *loop* foram chamadas as funções fillDelayBuffer, getFromDelayBuffer e feedbackDelay para implementar o *delay* por *buffer* circular; essas funções recebem por como parâmetros o *buffer* do *plug-in*, o delayBuffer criado e o canal no qual deve ocorrer a iteração, o funcionamento dessas funções será exposto em seguida. O código da classe ficou como mostra a Figura 3.34.

3.3.4.1.4 Implementando o método fillDelayBuffer

O método fillDelayBuffer realiza o preenchimento do delayBuffer a partir dos valores presentes no *buffer* principal. Como esses dois *buffers* possuem tamanhos diferentes, alguns cuidados especiais tiveram de ser tomados.

Primeiramente foram criadas algumas variáveis facilitar a implementação e entendimento do código: channelData é um ponteiro para o canal do *buffer* principal, bufferSize armazena o tamanho do *buffer* principal, delayBufferSize armazena o tamanho do delayBuffer e inputGain recebe o valor de inputGain da apvts. Essas variáveis podem ser vistas na Figura 3.35.

Figura 3.34 – Implementando o processBlock para o BigLoop.

```
juce::ScopedNoDenormals noDenormals;
auto totalNumInputChannels = getTotalNumInputChannels();
auto totalNumOutputChannels = getTotalNumOutputChannels();

for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
        // preenche o buffer de delay
        fillDelayBuffer(buffer, delayBuffer, channel);
        // lê do buffer de delay
        getFromDelayBuffer(buffer, delayBuffer, channel);
        // adiciona o delay no delayBuffer
        feedbackDelay(buffer, delayBuffer, channel);
}

// aumenta a posição de escrita no tamanho do buffer principal
writePosition += buffer.getNumSamples();
// garantindo que a posição estará dentro do tamanho do buffer
writePosition \%= delayBuffer.getNumChannels();
```

Fonte: Do autor (2021)

Figura 3.35 – Variáveis locais do método fillDelayBuffer.

```
// ponteiro para o canal do buffer
auto* channelData = buffer.getReadPointer(channel);
// tamanho do buffer
auto bufferSize = buffer.getNumSamples();
// tamanho do delayBuffer
auto delayBufferSize = delayBuffer.getNumSamples();

// lê o valor do ganho de entrada
float inputGain = *apvts.getRawParameterValue("inputGain");
```

Fonte: Do autor (2021)

Nesta próxima parte, o delayBuffer será preenchido com os valores do *buffer* principal, utilizando o método copyFromWithRamp. Esse método recebe como parâmetro o canal do *buffer* a ser preenchido, a posição inicial de escrita no *buffer* de destino, o *buffer* de onde serão lidos os valores, o número de valores que deverão ser copiados e os ganhos inicial e final aplicados ao conteúdo copiado, respectivamente. Contudo, devido aos tamanhos divergentes dos *buffers*, tem-se duas possibilidades: uma quando o *buffer* principal cabe completamente no delayBuffer e outra quando o o tamanho do *buffer* principal ultrapassa o limite de amostras do delayBuffer

Considerando as duas possibilidades, é verificado se a quantidade de valores presente no *buffer* principal ultrapassa os limites de escrita no delayBuffer. Caso não ultrapasse, o método copyFromWithRamp é executado apenas uma vez, copiando todos os valores do *buffer* principal para o delayBuffer. Caso ultrapasse, o método deve ser executado duas vezes, uma preenchendo o espaço restante do delayBuffer e outra sobrescrevendo os valores iniciais do delayBuffer com os valores restantes do *buffer* principal. Antes de finalizar a execução do método fillDelayBuffer, a o valor da variável lastInputGain é atualizado. A implementação pode ser vista na Figura 3.36.

Figura 3.36 – Preenchendo o delayBuffer.

```
// se as posições que serão preenchidas não ultrapassam a última posição do
        // delayBuffer
if (bufferSize + writePosition < delayBufferSize)
{
        // preenche o delayBuffer com os valores do buffer principal, aplicando
                // uma rampa de ganho
        delayBuffer.copyFromWithRamp(channel, writePosition, channelData,
                bufferSize, lastInputGain, inputGain);
}
else // se ultrapassam
{
        // quantos valores ainda podem ser escritos no delayBuffer
        auto numSamplesToEnd = delayBufferSize - writePosition;
        // preenche o delayBuffer até a última posição, aplicando uma rampa
                // de ganho
        delayBuffer.copyFromWithRamp(channel, writePosition, channelData,
                numSamplesToEnd, lastInputGain, inputGain);
        // quantos valores sobraram no buffer
        auto numSamplesAtStart = bufferSize - numSamplesToEnd;
        // preenche o delayBuffer com o que sobrou no buffer pricipal,
                // aplicando uma rampa de ganho
        delayBuffer.copyFromWithRamp(channel, 0, channelData, numSamplesAtStart,
                lastInputGain, inputGain);
}
// atualiza o valor do lastInputGain
lastInputGain = inputGain;
```

Fonte: Do autor (2021)

3.3.4.1.5 Implementando o método getFromDelayBuffer

O método getFromDelayBuffer realiza o preenchimento do *buffer* principal a partir dos valores presentes no delayBuffer, por isso houve de se tomar cuidados parecidos com do método

fillDelayBuffer. Também foram criadas variáveis facilitar a implementação e entendimento do código: bufferSize, delayBufferSize e delayTime, que recebe o valor de delaTime da apvts. Implementou-se também uma condição para zerar os valores do delayBuffer caso haja alteração no delayTime, evitando valores de lixo que causam sons não desejados no processamento. A implementação pode ser vista na Figura 3.37.

Figura 3.37 – Variáveis locais do método getFromDelayBuffer.

```
// tamanho do buffer
auto bufferSize = buffer.getNumSamples();
// tamanho do delayBuffer
auto delayBufferSize = delayBuffer.getNumSamples();

// lê o valor do tempo de delay
int delayTime = *apvts.getRawParameterValue("delayTime"); // ms

// evita lixo ao alterar o tempo de delay
if (delayTime != lastDelayTime)
{
        delayBuffer.clear();
}
```

Fonte: Do autor (2021)

Antes de preencher o *buffer* principal, foi definida a posição de leitura (readPosition) no delayBuffer. Para obter o atraso desejado em segundos, subtraiu-se da posição de escrita (tempo sem atraso) o tempo do *delay* em segundos multiplicado pela frequência de amostragem (essa multiplicação resulta no número de amostras que deve ser atrasadas para obter o atraso em segundos desejado). Também é feita uma verificação, se o valor obtido para a readPosition é negativo, soma-se a ela o tamanho do delayBuffer.

Agora o *buffer* principal pode ser preenchido com os valores lidos do delayBuffer, utilizando o método copyFrom. Esse método é parecido com o copyFromWithRamp, mas não aplica ganho ao sinal. Como no fillDelayBuffer, tem-se duas possibilidades: uma quando as posições lidas não ultrapassam o limite do delayBuffer e outra quando ultrapassam.

Verifica-se essa condição. Caso não ultrapasse, o método copyFrom é executado apenas uma vez, preenchendo completamente o *buffer* principal. Caso ultrapasse, o método deve ser executado duas vezes, uma preenchendo uma parte do *buffer* principal com os valores que restam no delayBuffer e outra finalizando o preenchimento do *buffer* principal a partir da posição

inicial do delayBuffer. Antes de finalizar a execução do método getFromDelayBuffer, o valor da variável lastDelayTime é atualizado. A implementação pode ser vista na Figura 3.38.

Figura 3.38 – Preenchendo o *buffer* principal com os valores atrasados.

```
// a posição que irá ser lida do delayBuffer
auto readPosition = static_cast<int>(writePosition -
        (getSampleRate() * delayTime / 1000));

// assegurar que não serão lidas posições negativas
if (readPosition < 0) readPosition += delayBufferSize;

// se as posições que serão lidas não ultrapassam a última posição do delayBuffer
if (readPosition + bufferSize < delayBufferSize)
{
        // preenche o buffer inteiro
        buffer.copyFrom(channel, 0, delayBuffer.getReadPointer(
                channel, readPosition), bufferSize);
}
else // se ultrapassam
{
        // quantos valores ainda podem ser lidos do delayBuffer
        auto numSamplesToEnd = delayBufferSize - readPosition;
        // adiciona valores até finalizar o delayBuffer
        buffer.copyFrom(channel, 0, delayBuffer.getReadPointer(
                channel, readPosition), numSamplesToEnd);
        // quantos valores faltam para completar o buffer
        auto numSamplesAtStart = bufferSize - numSamplesToEnd;
        // termina de preencher o buffer
        buffer.copyFrom(channel, numSamplesToEnd,
                delayBuffer.getReadPointer(channel), numSamplesAtStart);
    }
// atualiza o valor do lastDelayTime
lastDelayTime = delayTime;
```

Fonte: Do autor (2021)

3.3.4.1.6 Implementando o método feedbackDelay

O método feedbackDelay realiza a realimentação do delayBuffer com os valores atrasados, presentes no *buffer* principal. A implementação deste método foi bem parecida com a do método fillDelayBuffer, as diferenças foram o método utilizado para preencher o delayBuffer e o valor do ganho. O método utilizado foi o addFromWithRamp, que recebe os mesmos parâmetros do copyFromWithRamp; a diferença se dá pela operação de soma dos valores do *buffer* de origem aos do *buffer* de destino, em vez da substituição. O valor do ganho utilizado foi o

do parâmetro feedbackGain, buscado na apvts. Na Figura 3.39 é mostrada a implementação do código do feedbackDelay.

Figura 3.39 – Código do método feedbackDelay.

```
// posição que será lida do buffer
auto* channelData = buffer.getReadPointer(channel);
// tamanho do buffer
auto bufferSize = buffer.getNumSamples();
// tamanho do delayBuffer
auto delayBufferSize = delayBuffer.getNumSamples();

// lê o valor do ganho de feedback
float feedbackGain = *apvts.getRawParameterValue("feedbackGain");

// se as posições que serão preenchidas não ultrapassam a última posição do
        // delayBuffer
if (bufferSize + writePosition < delayBufferSize)
{
        // adiciona no delayBuffer os valores do buffer principal,
                // aplicando uma rampa de ganho
        delayBuffer.addFromWithRamp(channel, writePosition, channelData,
                bufferSize, lastFeedbackGain, feedbackGain);
}
else // se ultrapassa
{
        // quantos valores ainda podem ser escritos no delayBuffer
        auto numSamplesToEnd = delayBufferSize - writePosition;
        // adiciona no delayBuffer até a última posição,
                // aplicando uma rampa de ganho
        delayBuffer.addFromWithRamp(channel, writePosition, channelData,
                numSamplesToEnd, lastFeedbackGain, feedbackGain);
        // quantos valores sobraram no buffer
        auto numSamplesAtStart = bufferSize - numSamplesToEnd;
        // adiciona no delayBuffer com o que sobrou no buffer pricipal,
                // aplicando uma rampa de ganho
        delayBuffer.addFromWithRamp(channel, 0, channelData, numSamplesAtStart,
                lastFeedbackGain, feedbackGain);
}
// atualiza o valor do lastFeedbackGain
lastFeedbackGain = feedbackGain;
```

Fonte: Do autor (2021)

3.3.4.2 A classe BigLoopAudioProcessorEditor

A implementação dessa classe foi muito semelhante às classe equivalentes dos *plug-ins* BuzzTortion e WonderVerb. Foram declarados três juce::Slider para os parâmetros e os atributos

sliderAttachments, area, textPosition, textHeight e textWidth. No construtor, foi implementado o código para adicionar, exibir e associar os *sliders* ao apvts. O método paint foi utilizado para adicionar elementos gráficos à GUI e o método resized para posicionar os sliders. Devido a esse tipo de implementação já ter sido explicado, os códigos poderão ser visualizados no Apêndice D.

3.4 Teste dos *plug-ins*

3.4.1 Testes das funcionalidades

Ao compilar os *plug-ins* nas IDEs, os arquivos de *plug-in* são gerados, com eles é possível executar esses *plug-ins* nas DAWs que tenham suporte ao formato. Neste trabalho, foi utilizada a DAW REAPER para executar e realizar os testes com os *plug-ins*.

O primeiro teste realizado foi da abertura e correta visualização dos *plug-ins* na DAW. Os *plug-ins* foram abertos no REAPER para testar a exibição correta da GUI e as respostas dos controles às ações do usuário.

O segundo teste realizado foi do processamento dos sinais pelos *plug-ins*. Para realizar estes testes era necessário ter sinais de áudio para serem processados, que foram obtidos da plataforma Freesound. Para testar o DigiQ 7B foi escolhida uma gravação contendo mais de um instrumento, pois dessa forma se torna mais fácil a percepção das diversas frequências que compõem o sinal e as mudanças causadas pelo filtro são mais evidentes. Para testar o BuzzTortion foi escolhida uma gravação de um solo de guitarra com um som mais limpo (sem a aplicação de efeitos ou com o mínimo possível de efeitos), pois dessa forma fica mais nítida a distorção causada no sinal. Para testar o WonderVerb foi escolhida a gravação de um solo de violão com um som mais seco (com pouca ou nenhuma característica de reverberação), para que a reverberação se tornasse evidente no sinal, adicionando sensação espacial ao som. Para testar o BigLoop, foi escolhida uma gravação de uma batida única em um bumbo[1], para que ficassem bem notáveis as repetições causadas pelo *plug-in*. Por fim, utilizou-se o mesmo solo de guitarra do teste do BuzzTortion para testar o funcionamento dos quatro *plug-ins* em conjunto.

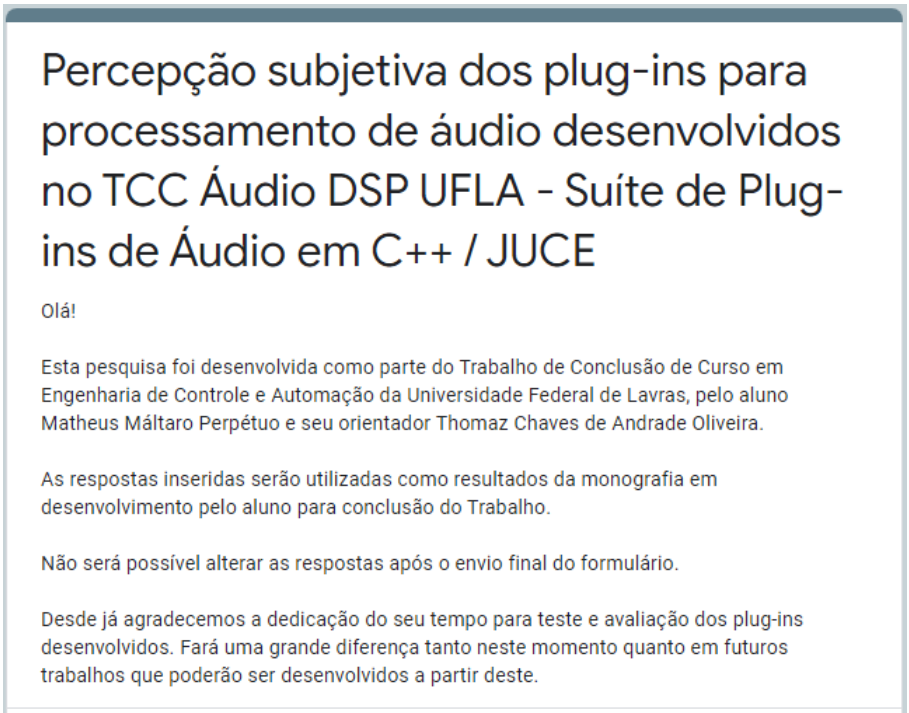
[1] Tambor cilíndrico de grande dimensão, de som grave e seco.

3.4.2 Testes de percepção subjetiva

O terceiro teste realizado foi convidando produtores de áudio, que são os profissionais para os quais os *plug-ins* desenvolvidos se destinam, para testar os *plug-ins*. Para esses testes, não foi definida qual DAW deveria ser utilizada, cada produtor utilizou uma DAW por escolha própria. Para identificação das respostas, foi feita a nominação Produtor A, Produtor B, Produtor C, Produtor D e Produtor E.

Para facilitar e padronizar o *feedback* dos profissionais, foi criado um formulário para que respondessem após realizarem os testes. Este formulário contou com perguntas introdutórias sobre o cenário de desenvolvimento de *plug-ins* de áudio no Brasil e nas universidades brasileiras, perguntas sobre as características e desempenho de cada *plug-in* e perguntas sobre o resultado do trabalho em geral. O título dado ao formulário e o texto de apresentação podem ser vistos na Figura 3.40

Figura 3.40 – Título e texto de apresentação do formulário.



Fonte: Do autor (2021)

3.4.2.1 Seção introdutória

A seção introdutória teve como título "Percepção geral do cenário de desenvolvimento de *plug-ins* de áudio no Brasil e nas universidades brasileiras". As questões foram as seguintes:

a) Pergunta: Você já conhecia a Universidade Federal de Lavras? Respostas possíveis: Sim, Não.

b) Pergunta: Você sabia que as instituições de ensino superior públicas desenvolvem tecnologias voltadas para o áudio? Respostas possíveis: Sim, Não;

c) Pergunta: Como você avalia o desenvolvimento desse tipo de tecnologia em uma Universidade Brasileira? Respostas possíveis: Muito positivo, Positivo, Indiferente, Negativo, Muito negativo;

d) Pergunta: Sinta-se à vontade para justificar sua resposta anterior (campo não obrigatório). Resposta aberta;

e) Pergunta: Você acredita que o Brasil possa se tornar referência em desenvolvimento de ferramentas voltadas para o áudio? Respostas possíveis: Sim, Não, Talvez;

f) Pergunta: Sinta-se à vontade para justificar sua resposta anterior (campo não obrigatório). Resposta aberta;

g) Pergunta: Você acredita que uma proximidade entre desenvolvedores de *software* e os produtores musicais e donos de estúdio seria benéfica para o desenvolvimento de novas ferramentas voltadas para o áudio? Respostas possíveis: Sim, Não, Talvez;

h) Pergunta: Sinta-se à vontade para justificar sua resposta anterior (campo não obrigatório). Resposta aberta;

i) Pergunta: Exponha sua opinião sobre a iniciativa de produzir ferramentas voltadas para o áudio dentro das instituições de ensino públicas brasileiras. Resposta aberta;

j) Pergunta: Sugestões para desenvolvimento de novas ferramentas voltadas para o áudio (campo não obrigatório). Resposta aberta.

3.4.2.2 Seção para cada *plug-in*

Nessa seção foram formuladas questões sobre as características e desempenho e foi replicada para obter respostas separadas para cada *plug-in*. As questões foram as seguintes:

a) Pergunta: Como você avalia a utilidade do *plug-in*? (Ele faz sentido ou facilita o seu trabalho?). Respostas possíveis: de 1 a 10, sendo 1 Pouco útil e 10 muito útil;

b) Pergunta: Como você avalia a identidade sonora do *plug-in*? (O timbre característico do *plug-in*). Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Boa;

c) Pergunta: O processamento soa natural ou artificial? Respostas possíveis: Natural, Artificial;

d) Pergunta: Como você avalia a o desempenho do *software*? (A execução ocorre sem falhas?). Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Bom;

e) Pergunta: Como você avalia a performance do software? (Consumo de recursos do sistema). Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Boa;

f) Pergunta: O *plug-in* atende à proposta? (Suas características estão de acordo com o que se espera desse tipo de *plug-in*?). Respostas possíveis: Sim, Parcialmente, Não;

g) Pergunta: Sinta-se à vontade para justificar sua resposta anterior (campo não obrigatório). Resposta aberta;

h) Pergunta: Como você avalia a usabilidade da interface? (Facilidade na utilização do *plug-in*). Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Boa;

i) Pergunta: A interface traz todos os parâmetros necessários? (Caso tenha faltado de algum parâmetro, inserir no campo outros). Respostas possíveis: Sim, Outros (resposta aberta);

j) Pergunta: O *plug-in* realiza de maneira satisfatória a mesma tarefa de outros *plug-ins* consolidados no mercado de áudio? Respostas possíveis: Sim, Parcialmente, Não;

k) Pergunta: Sinta-se à vontade para justificar sua resposta anterior (campo não obrigatório). Resposta aberta;

l) Pergunta: Você continuaria a utilizar este *plug-in* em suas produções musicais? Respostas possíveis: Sim, Não, Talvez;

m) Pergunta: Sinta-se à vontade para justificar sua resposta anterior (campo não obrigatório). Resposta aberta;

n) Pergunta: Qual a sua nota geral para o desempenho sonoro do *plug-in*? Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Bom;

o) Pergunta: Qual a sua nota geral para a interface do *plug-in*? Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Boa;

p) Pergunta: Qual a sua nota geral para o *plug-in*? Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Bom;

q) Pergunta: O que você achou muito bom no *plug-in*? (campo não obrigatório). Resposta aberta;

r) Pergunta: O que você achou muito ruim no *plug-in*? (campo não obrigatório). Resposta aberta;

s) Pergunta: Conte um pouco sobre sua experiência ao utilizar o *plug-in*. Resposta aberta;

t) Pergunta: Você tem alguma sugestão para melhoria deste *plug-in*? (campo não obrigatório). Resposta aberta.

3.4.2.3 Seção de conclusão

Nesta seção foram feitas perguntas sobre o trabalho como um todo e pedidos de sugestões. As questões foram as seguintes:

a) Pergunta: Qual a sua nota geral para o trabalho desenvolvido? Respostas possíveis: de 1 a 10, sendo 1 Ruim e 10 Bom;

b) Pergunta: Qual sua sugestão para outros tipos de *plug-in* que poderiam ser desenvolvidos em futuros trabalhos voltados a este tema? (campo não obrigatório). Resposta aberta;

c) Pergunta: Críticas, sugestões e comentários (Muito obrigado por ter concluído os testes e a pesquisa! Fique à vontade para deixar as suas críticas, sugestões e comentários) (campo não obrigatório). Resposta aberta.

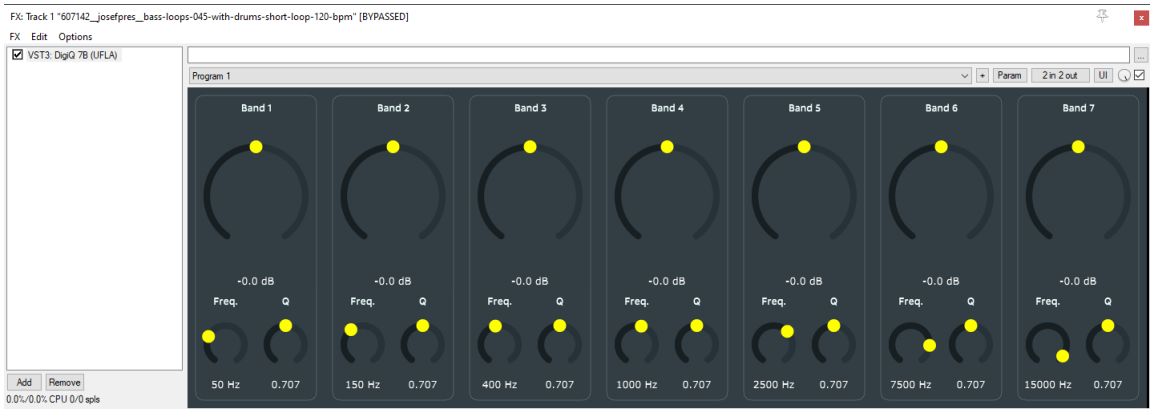
4 RESULTADOS E DISCUSSÕES

4.1 Resultados dos testes de funcionalidade dos *plug-ins*

4.1.1 GUI

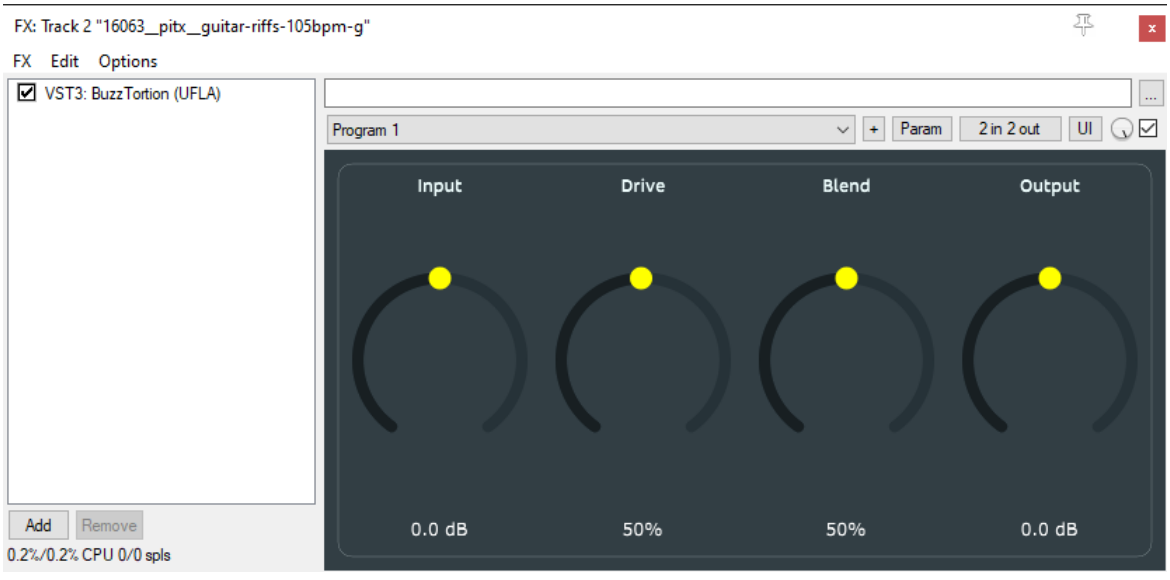
Os testes da GUI ocorreram da maneira que era esperado. Os *plug-ins* foram exibidos corretamente na tela e os controles deslizantes responderam bem às ações executadas. Nas imagens 4.1, 4.2, 4.3 e 4.4 são mostradas as interfaces geradas para os *plug-ins*, sendo executados dentro do REAPER.

Figura 4.1 – GUI do DigiQ 7B.



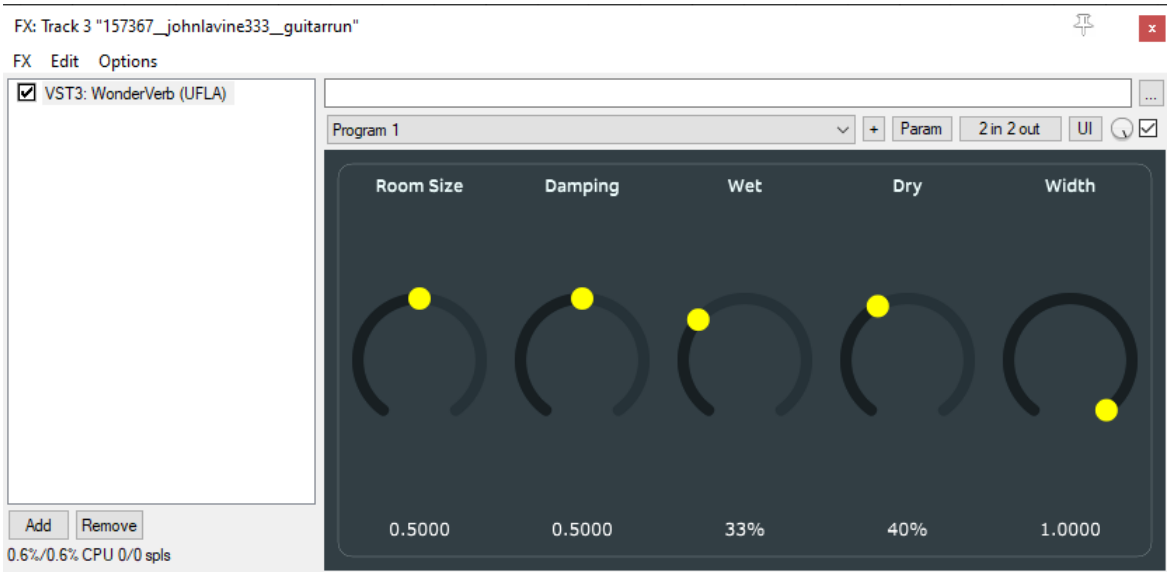
Fonte: Do autor (2021)

Figura 4.2 – GUI do BuzzTortion.



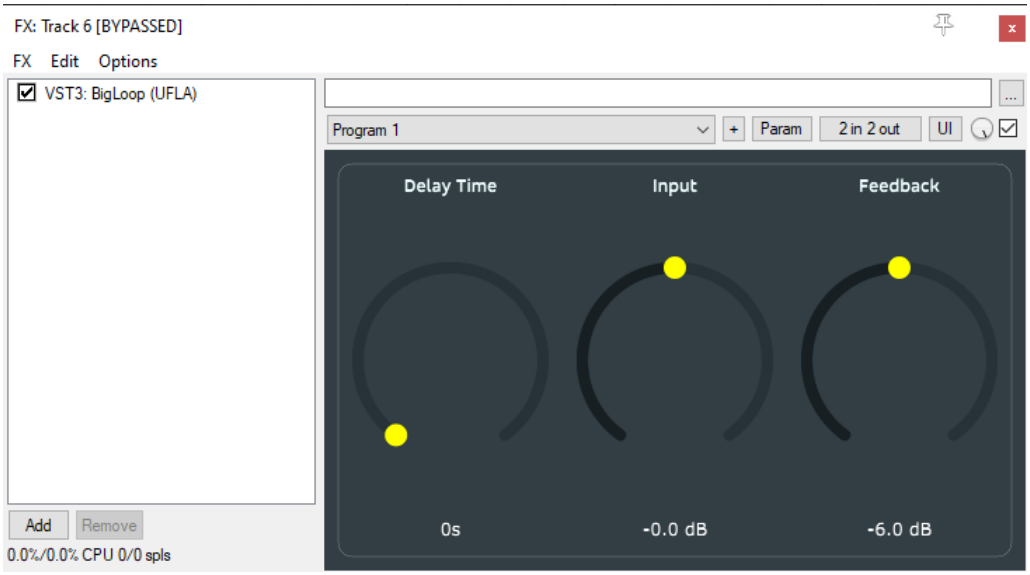
Fonte: Do autor (2021)

Figura 4.3 – GUI do WonderVerb.



Fonte: Do autor (2021)

Figura 4.4 – GUI do BigLoop.



Fonte: Do autor (2021)

4.1.2 Sonoridade

4.1.2.1 DigiQ 7B

Para o teste de sonoridade desse *plug-in* foi utilizado o áudio "Bass loops 045 with drums short loop 120 bpm.wav"[1], uma gravação de baixo e bateria disponibilizada no Freesound pelo usuário josefpres. Foi feito um primeiro teste realizando cortes nas frequências mais baixas: -18dB em 50Hz com Q 0.079, -18dB em 150Hz com Q 0.119 e -18dB em 400Hz com Q 0.556.

[1] Disponível em <https://freesound.org/people/josefpres/sounds/607142/>.

Um segundo teste foi feito realizando cortes nas frequências mais altas: -11.7dB em 1500Hz com Q 0.416, -15.5dB em 7500Hz com Q 0.252 e -17.9dB em 15000Hz com Q 0.134.

Em ambos os testes o equalizador funcionou como esperado e foi possível realizar um bom isolamento das frequências, mas sentiu-se falta de parâmetros para controle dos ganhos de entrada e saída, para compensar os ganhos e cortes aplicados. Os sinais de áudio antes e depois da aplicação do DigiQ 7B estão disponíveis para audição no Google Drive[2].

4.1.2.2 BuzzTortion

Para o teste de sonoridade desse *plug-in* foi utilizado o áudio "guitar_riffs_105bpm_G.wav"[3], uma gravação de guitarra, aparentemente sem efeitos aplicados, diponibilizada no Freesound pelo usuário pitx. O teste foi feito com os seguintes valores fixados nos parâmetros: Input em 6dB, Drive em 100%, Blend em 100% e Output em -6dB. Os parâmetros foram utilizados com esses valores para evidenciar ao máximo o efeito de distorção que se pode obter com esse *plug-in*.

O *plug-in* funcionou como esperado, mas percebeu-se que ele causa uma distorção excessiva nas frequências mais altas devido à não utilização de um método de *oversampling*. No entanto, como esse tipo de *plug-in* é destinado à utilização de forma mais artística e criativa, entendeu-se que ele atingiu ao objetivo. Os sinais de áudio antes e depois da aplicação do BuzzTortion estão disponíveis para audição no Google Drive[4].

4.1.2.3 WonderVerb

Para o teste de sonoridade desse *plug-in* foi utilizado o áudio "GuitarRun.wav"[5], uma gravação de violão, aparentemente sem efeitos aplicados, diponibilizada no Freesound pelo usuário JohnLaVine333. O teste foi feito com os seguintes valores fixados nos parâmetros: Room Size em 0.3, Damping em 0.5, Wet em 33%, Dry em 40% e Width em 1. Os parâmetros foram utilizados com esses valores para evidenciar o efeito de reverberação, sem descaracterizar o som original por reverberação excessiva.

---

[2] Disponível em <https://drive.google.com/drive/folders/1cd9aRxkUqB-ZmSSQqfwvWQdBejJrZrIH?usp=sharing>.

[3] Disponível em <https://freesound.org/people/pitx/sounds/16063/>.

[4] Disponível em <https://drive.google.com/drive/folders/1vsqhAGCvshCtwKJKdj68oqI5n5Q8F7dY?usp=sharing>.

[5] Disponível em <https://freesound.org/people/JohnLaVine333/sounds/157367/>.

O *plug-in* funcionou como esperado, uma alteração que poderia tornar o uso mais intuitivo seria transformar os parâmetros Wet e Dry em um parâmetro de Blend, como o que existe no BuzzTortion; também poderia ser alterado o valor de Width para percentual, para torná-lo mais intuitivo. O *plug-in* respondeu bem, gerando um bom efeito de ambientação, então entendeu-se que ele atendeu ao objetivo. Os sinais de áudio antes e depois da aplicação do WonderVerb estão disponíveis para audição no Google Drive[6].

4.1.2.4 BigLoop

Para o teste de sonoridade desse *plug-in* foi utilizado o áudio "kick swedish.wav"[7], uma gravação de uma batida única de bumbo, diponibilizada no Freesound pelo usuário Veiler. O teste foi feito com os seguintes valores fixados nos parâmetros: Delay Time em 103s, Input em -1.8dB e Feedback em -12.3dB. Os parâmetros foram utilizados com esses valores para evidenciar o efeito de repetição, mas sem gerar uma excessividade desnecessária.

O *plug-in* não funcionou como esperado. Ao ouvir o áudio processado, percebeu-se que ele teve o ganho reduzido ao ser processado pelo *plug-in*, o que não deveria ter acontecido; além disso, após algumas repetições aparecem artefatos não desejados no sinal, prováveis frutos de alguma inconsistência na transferência dos dados entre os *buffers*. Sentiu-se falta de um parâmetro Blend, como o do BuzzTortion, para controle da combinação entre o sinal original e o sinal atrasado; a falta desse parâmetro faz com que o *plug-in* deva sempre ser utilizado em processamento paralelo. Também deveria ser alterada a forma de controlar o atraso, em vez de segundos, deveria ser utilizado um atraso por figuras rítmicas, para torná-lo mais intuitivo. Entendeu-se que esse *plug-in* atingiu o objetivo de inserir atrasos no sinal, mas para ser usado artisticamente deve ser melhorado. Os sinais de áudio antes e depois da aplicação do BigLoop estão disponíveis para audição no Google Drive[8].

4.1.2.5 Vários *plug-ins* em conjunto

Foi realizado um último teste, combinando mais de um *plug-in*, para explorar as possibilidades de modificação de um sinal de áudio utilizando os *plug-ins* desenvolvidos neste

[6] Disponível em <https://drive.google.com/drive/folders/14P_GTMltGuT28M15YKSITtTFcNsAkcv2?usp=sharing>.
[7] Disponível em <https://freesound.org/people/Veiler/sounds/264601/>.
[8] Disponível em <https://drive.google.com/drive/folders/1bTKl4M3J_zU9kT0nHyAOojyx_kMxzv3j?usp=sharing>.

trabalho. Para o teste de sonoridade dos *plug-ins* em conjunto foi utilizado novamente o áudio "guitar_riffs_105bpm_G.wav", diponibilizado no Freesound pelo usuário pitx.

Primeiramente, utilizou-se o DigiQ 7B para ajustar o timbre, limpar as frequências mais baixas e reduzir o ganho nas frequências mais altas, antes de utilizar o BuzzTortion; as modificações feitas pelo equalizador foram: -17.7dB em 50Hz, 6.1dB em 150Hz, -12.2dB em 400Hz, -9.1dB em 1000Hz, 3.6dB em 2500Hz, -1.8dB em 7500Hz e -14.3dB em 15000Hz, todos com Q de 0.707.

Depois utilizou-se o BuzzTortion para adicionar distorção ao sinal. Os parâmetros utilizados foram: Input em 0dB, Drive em 82.7%, Blend em 59.69% e Output em 0dB. Após isso utilizou-se o WonderVerb para adicionar ambientação. Os parâmetros utilizados foram: Room Size em 0.2472, Damping em 0.3346, Wet em 24.72%, Dry em 48.7% e Width em 0.7127.

Para finalizar, utilizou-se mais um DigiQ 7B para as frequências altas após terem sido removidas para utilização do BuzzTortion. As modificações aplicadas ao sinal foram: 4.8dB em 7500Hz e 9.1dB em 15000Hz, ambas com Q de 0.707.

O *plug-in* BigLoop não foi utilizado neste último teste devido aos problemas que apresentou, já mencionados anteriormente.

Após a aplicação de todos esses processamentos ao sinal original, obteve-se uma sonoridade mais artística e mais agradável, o que permitiu deduzir que os *plug-ins* poderiam ser utilizados de forma satisfatória. Os sinais de áudio antes e depois da aplicação dos *plug-ins* estão disponíveis para audição no Google Drive[9].

4.2 Resultados da pesquisa realizada com os profissionais do mercado de áudio

4.2.1 Compilação dos resultados

Para melhor visualização dos resultados, as respostas das pesquisas foram compiladas em tabelas, que podem ser vistas nas Figuras 4.5, 4.6, 4.7, e 4.8.

4.2.2 Seção introdutória

As respostas para as perguntas da seção introdutória foram as seguintes:

a) Pergunta: Você já conhecia a Universidade Federal de Lavras? Respostas: 80% Sim, 20% Não;

[9] Disponível em <https://drive.google.com/drive/folders/1SjfhArUydgcMuhwB7CVZEa2OipqG_G3l?usp=sharing>.

Figura 4.5 – Compilação dos resultados da seção introdutória.

PERGUNTAS GERAIS					
Você já conhecia a Universidade Federal de Lavras?	Sim	Não			
	80%	20%			
Você sabia que as instituições de ensino superior públicas desenvolvem tecnologias voltadas para o áudio?	Sim	Não			
	0%	100%			
Como você avalia o desenvolvimento desse tipo de tecnologia em uma Universidade Brasileira?	Muito positivo	Positivo	Indiferente	Negativo	Muito negativo
	60%	40%	0%	0%	0%
Você acredita que o Brasil possa se tornar referência em desenvolvimento de ferramentas voltadas para o áudio?	Sim	Não	Talvez		
	80%	0%	20%		
Você acredita que uma proximidade entre desenvolvedores de software e os produtores musicais e donos de estúdio seria benéfica para o desenvolvimento de novas ferramentas voltadas para o áudio?	Sim	Não	Talvez		
	100%	0%	0%		

Fonte: Do autor (2021)

Figura 4.6 – Compilação dos resultados das questões de múltipla escolha das seções dos *plug-ins*.

	DigiQ 7B			BuzzTortion		
O plugin atende à proposta?	Sim	Parcialmente	Não	Sim	Parcialmente	Não
	20%	80%	0%	60%	20%	20%
O plugin realiza de maneira satisfatória a mesma tarefa de outros plugins consolidados no mercado de áudio?	Sim	Parcialmente	Não	Sim	Parcialmente	Não
	20%	60%	20%	40%	40%	20%
Você continuaria a utilizar este plugin em suas produções musicais?	Sim	Não	Talvez	Sim	Não	Talvez
	40%	0%	60%	60%	40%	0%
O processamento soa natural ou artificial?	Natural	Artificial		Natural	Artificial	
	60%	40%		40%	60%	

	WonderVerb			BigLoop		
O plugin atende à proposta?	Sim	Parcialmente	Não	Sim	Parcialmente	Não
	80%	20%	0%	40%	40%	20%
O plugin realiza de maneira satisfatória a mesma tarefa de outros plugins consolidados no mercado de áudio?	Sim	Parcialmente	Não	Sim	Parcialmente	Não
	80%	20%	0%	20%	40%	40%
Você continuaria a utilizar este plugin em suas produções musicais?	Sim	Não	Talvez	Sim	Não	Talvez
	100%	0%	0%	20%	40%	40%
O processamento soa natural ou artificial?	Natural	Artificial		Natural	Artificial	
	80%	20%		60%	40%	

Fonte: Do autor (2021)

b) Pergunta: Você sabia que as instituições de ensino superior públicas desenvolvem tecnologias voltadas para o áudio? Respostas: 0% Sim, 100% Não;

c) Pergunta: Como você avalia o desenvolvimento desse tipo de tecnologia em uma Universidade Brasileira? Respostas: 60% Muito positivo, 40% Positivo, 0% Indiferente, 0% Negativo, 0% Muito negativo;

Figura 4.7 – Compilação dos resultados das notas para os *plug-ins*.

	DigiQ 7B	BuzzTortion	WonderVerb	BigLoop
Como você avalia a utilidade do plug-in?	7,6	6,8	8,8	7
Como você avalia a identidade sonora do plug-in?	6,8	6,2	8,4	7,2
Como você avalia a o desempenho do software?	8,2	8	8,2	7,8
Como você avalia a performance do software?	8,2	8,2	8,2	8
Como você avalia a usabilidade da interface?	7,6	7,4	8	6,8
Qual a sua nota geral para o desempenho sonoro do plug-in?	7,6	6,6	8,8	7,2
Qual a sua nota geral para a interface do plug-in?	7,4	7,4	8,4	6,2
Qual a sua nota geral para o plug-in?	7,4	6,6	8,4	6,8

Fonte: Do autor (2021)

Figura 4.8 – Compilação dos resultados da seção de conclusão.

CONCLUSÃO
Qual a sua nota geral para o trabalho desenvolvido?
9,2

Fonte: Do autor (2021)

d) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

   - "Toda forma de desenvolvimento em tecnologia no Brasil é de extrema importância!" – Produtor B;
   - "Eu acredito que, como centro de excelência de tecnologia, seria uma incubadora para projetos de tenologia em geral. O que não seria diferente para o áudio." – Produtor D;

e) Pergunta: Você acredita que o Brasil possa se tornar referência em desenvolvimento de ferramentas voltadas para o áudio? Respostas: 80% Sim, 0% Não, 20% Talvez;

f) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

   - "Depende da comunidade do áudio brasileiro abraçar os projetos." – Produtor B;
   - "Acredito que, como o processo de difusão de conhecimento é algo globalizado, acredito que não existam motivos para o nosso país não vir a ser referência no campo." – Produtor D;

g) Pergunta: Você acredita que uma proximidade entre desenvolvedores de *software* e os produtores musicais e donos de estúdio seria benéfica para o desenvolvimento de novas ferramentas voltadas para o áudio? Respostas: 100% Sim, 0% Não, 0% Talvez;

h) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

- "Porque os parâmetros podem ser ajustados conciliando prática e teoria, de uma forma mais personalizada." – Produtor A;
- "Softwares de áudio são ferramentas de trabalho para produtores musicais, músicos que se gravam, engenheiros etc... Sem o *feedback* e parceria de todos pra melhorar os projetos, seria impossível avançar com eficiência." – Produtor B;
- "Eu acredito que, todo o desenvolvimento de *plug-ins* e *software*, tem que obrigatoriamente passar por pessoas que possuem experiência e experiência musical. Muitos equipamentos que testei no passado sofreram muito pela falta de comunicação entre os desenvolvedores e usuários." – Produtor D;

i) Pergunta: Exponha sua opinião sobre a iniciativa de produzir ferramentas voltadas para o áudio dentro das instituições de ensino públicas brasileiras. Respostas:

- "Extremamente positivo pois pode propiciar o acesso a este tipo de ferramenta para pequenos usuários com um acompanhamento próximo e dentro de uma linguagem que a maioria entenderá." – Produtor A;
- "Toda a forma de conhecimento que resulte em produtos pra melhorar a vida dos profissionais é muito importante. É difícil, mas com trabalho sério é possível." – Produtor B;
- "Desenvolver ferramentas é sempre interessante. Ter ferramentas objetivas ajuda no dia a dia." – Produtor C;
- "Acredito que seja muito válido, pois acredito que a universidade pública, como bem público, deve servir como fomentadora do desenvolvimento tecnológico nacional." – Produtor D;
- "Muito positiva." – Produtor E;

j) Pergunta: Sugestões para desenvolvimento de novas ferramentas voltadas para o áudio. Respostas:

- "Equipamentos analógicos de áudio com preço acessível" – Produtor A;
- "Ferramentas pra viabilidade do Dolby Atmos. Áudio espacial. *Surround*. Que já é realidade no cinema e na música." – Produtor B;
- "Simulador de amplificador de guitarra." – Produtor D.

Analisando as respostas da seção introdutória, percebeu-se que em geral os profissionais do áudio têm pouco conhecimento sobre as tecnologias voltadas ao áudio produzidas em instituições de ensino superior públicas. Esse tipo de trabalho foi avaliado como positivo, devido à importância do desenvolvimento de tecnologia no país. Também notou-se a confiança no desenvolvimento dessa área no país, devido ao aumento da acessibilidade ao conhecimento, porém dependendo de um apoio do mercado. Foi opinião unânime que seria benéfica a proximidade entre desenvolvedores de *software* e profissionais do áudio, acreditando que a comunicação entre as partes traria melhorias para as tecnologias desenvolvidas. A iniciativa foi avaliada positivamente por todos, acreditando na possibilidade da melhoria das tecnologias e do acesso a elas. Dentre as sugestões para novos trabalhos estão equipamentos analógicos a preços acessíveis, ferramentas voltadas ao áudio espacial e simulações de equipamentos analógicos.

4.2.3 DigiQ 7B

As respostas para as perguntas referentes ao DigiQ 7B foram as seguintes:

a) Pergunta: Como você avalia a utilidade do *plug-in*? Resposta média: 7,6; sendo 1 Pouco útil e 10 muito útil;

b) Pergunta: Como você avalia a identidade sonora do *plug-in*? Resposta média: 6,8; sendo 1 Ruim e 10 Boa;

c) Pergunta: O processamento soa natural ou artificial? Respostas: 60% Natural, 40% Artificial;

d) Pergunta: Como você avalia a o desempenho do *software*? Resposta média: 8,2; sendo 1 Ruim e 10 Bom;

e) Pergunta: Como você avalia a performance do *software*? Resposta média: 8,2; sendo 1 Ruim e 10 Boa;

f) Pergunta: O *plug-in* atende à proposta? Respostas: 20% Sim, 80% Parcialmente, 0% Não;

g) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

   - "Apesar de ter uma interface simples e fácil, as respostas durante as regulagens soaram muito agressivas nas frequências." − Produtor A;

- "A falta de recurso ganho de saída e entrada do sinal, faz muita falta, tem em todo tipo de *plug-ins*. O *plug-in* distorce fácil por falta de controle do ganho de entrada de sinal da fonte." − Produtor B;
- "Falta ajuste de tipos de curvas , controle de *low cut*, *high cut*." − Produtor E;

h) Pergunta: Como você avalia a usabilidade da interface? Resposta média: 7,6; sendo 1 Ruim e 10 Boa;

i) Pergunta: A interface traz todos os parâmetros necessários? Respostas: 20% Sim, 80% Outros:

- "Leitor de RTA." − Produtor A;
- "Faltou controles de ganho de saída e entrada." − Produtor B;
- "Gostaria de ter um controle de entrada e saída do sinal com medidores e de uma opção de solo das frequências para auxiliar na equalização corretiva." − Produtor C;
- "Controles de *Shelving filters*." − Produtor E;

j) Pergunta: O *plug-in* realiza de maneira satisfatória a mesma tarefa de outros *plug-ins* consolidados no mercado de áudio? Respostas: 20% Sim, 60% Parcialmente, 20% Não;

k) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

- "Para uso profissional e ajuste fino, além de textura ..deixou a desejar por falta de parâmetros importantes como citados anteriormente. Uma boa referência seria os equalizadores da Fabfilter." − Produtor B;
- "Eu sinto falta de uma interface um pouco mais visual, apesar do mesmo ocorrer com outros *plug-ins* de *channel strip* ssl, por exemplo." − Produtor D;
- "Controles de *Shelving filters* são muito úteis em equalizadores, a ausência deles, limita o *plugin* e seu uso pleno." − Produtor E;

l) Pergunta: Você continuaria a utilizar este *plug-in* em suas produções musicais? Respostas: 40% Sim, 0% Não, 60% Talvez;

m) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

- "Uso especificamente *software* Pro Tools, mas este *plug-in* ainda não está homologado, compatível." – Produtor B;
- "Compatibilidade com a DAW Pro Tools é fundamental." – Produtor E;

n) Pergunta: Qual a sua nota geral para o desempenho sonoro do *plug-in*? Resposta média: 7,6; sendo 1 Ruim e 10 Bom;

o) Pergunta: Qual a sua nota geral para a interface do *plug-in*? Resposta média: 7,4; sendo 1 Ruim e 10 Boa;

p) Pergunta: Qual a sua nota geral para o *plug-in*? Resposta média: 7,4; sendo 1 Ruim e 10 Bom;

q) Pergunta: O que você achou muito bom no *plug-in*? Respostas:

- "Interface fácil." – Produtor A;
- "Boa interface, simples funcional." – Produtor B;
- "Gostei da suavidade da interface aliadas a sonoridade." – Produtor C;
- "Sonoridade. Curva agressiva de equalização." – Produtor D;
- "Leve, simples e funcional." – Produtor E;

r) Pergunta: O que você achou muito ruim no *plug-in*? Respostas:

- "Resposta agressiva nas frequência e falta de RTA." – Produtor A;
- "Falta de parâmetros importantes. Timbre, textura sem personalidade." – Produtor B;
- "Interface pouco intuitiva." – Produtor D;
- "Ausência dos controles de *Shelving filters.*" – Produtor E;

s) Pergunta: Conte um pouco sobre sua experiência ao utilizar o *plug-in*. Respostas:

- "Movimentando os *knobs* achei muito a resposta muito agressiva." – Produtor A;
- "Foi divertido testar um produto feito por estudante brasileiro, e percebi que é possível sim que melhorem tendo como referência as melhores marcas do mercado mundial." – Produtor B;

- "Me surpreendeu a questão de soar bem, sem necessidade de girar muitos parâmetros de forma agressiva." – Produtor C;
- "Eu utilizei em uma trilha de guitarra e observei uma boa resposta de frequência, com uma curva de frequência bastante presente e agressiva. Acredito que em produções em que uma equalização mais agressiva seria bastante bem vindo." – Produtor D;
- "Não ser compatível com o Pro Tools dificultou, assim como a ausência dos filtros, principalmente ao ajustar cortes como *low cut* ou *high cut*, as bandas acabam por se somarem." – Produtor E;

t) Pergunta: Você tem alguma sugestão para melhoria deste *plug-in*? Respostas:

- "Melhorar a sensibilidade dos controles." – Produtor A;
- "Use as referências como Fabfilter, Waves, UAD, *plug-ins* Alliance etc..." – Produtor B;
- "*High Pass filter* e *Low Pass Filter*. Interface gráfica mais intuitiva." – Produtor D;
- "Compatibilidade com a DAW Pro Tools, adicionar os filtros *shelving*, talvez separar as bandas de equalização por cores diferentes." – Produtor E.

A partir das respostas recebidas, pôde-se notar que o *plug-in* apresentou bom funcionamento em geral. As principais críticas e sugestões recebidas foram quanto à falta de controle de ganho de entrada e saída do sinal, leitor de RTA, medidores, opção de "solo"nas frequências, separação das frequências por cores e adição filtros passa altas, passa baixas e *shelving*; também foi solicitada compatibilidade com a DAW Pro Tools e sugerido utilizar como referência os equalizadores da FabFilter. Algumas características tiveram avaliações que foram divergentes entre os produtores, como a resposta agressiva dos parâmetros, que agradou a alguns e desagradou a outros.

4.2.4 BuzzTortion

As respostas para as perguntas da seção introdutória foram as seguintes:

a) Pergunta: Como você avalia a utilidade do *plug-in*? Resposta média: 6,8; sendo 1 Pouco útil e 10 muito útil;

b) Pergunta: Como você avalia a identidade sonora do *plug-in*? Resposta média: 6,2; sendo 1 Ruim e 10 Boa;

c) Pergunta: O processamento soa natural ou artificial? Respostas: 40% Natural, 60% Artificial;

d) Pergunta: Como você avalia a o desempenho do *software*? Resposta média: 8; sendo 1 Ruim e 10 Bom;

e) Pergunta: Como você avalia a performance do *software*? Resposta média: 8,2; sendo 1 Ruim e 10 Boa;

f) Pergunta: O *plug-in* atende à proposta? Respostas: 60% Sim, 20% Parcialmente, 20% Não;

g) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

- "Houve um ruído não muito natural nas frequências mais agudas." − Produtor D;
- "Me pareceu um pouco agressivo e destrutivo no som." − Produtor E;

h) Pergunta: Como você avalia a usabilidade da interface? Resposta média: 7,4; sendo 1 Ruim e 10 Boa;

i) Pergunta: A interface traz todos os parâmetros necessários? Respostas: 60% Sim, 40% Outros:

- "Mais controles para timbragem." − Produtor A;
- "Existem muitos parâmetros nesse tipo de efeito que poderiam entrar em futuras atualizações." − Produtor B;

j) Pergunta: O *plug-in* realiza de maneira satisfatória a mesma tarefa de outros *plug-ins* consolidados no mercado de áudio? Respostas: 40% Sim, 40% Parcialmente, 20% Não;

k) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas: sem respostas;

l) Pergunta: Você continuaria a utilizar este *plug-in* em suas produções musicais? Respostas: 60% Sim, 40% Não, 0% Talvez;

m) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

- "De forma criativa, toda distorção é bem vinda." − Produtor B;
- "O ruído nas frequências altas impossibilita a utilização." − Produtor D;

n) Pergunta: Qual a sua nota geral para o desempenho sonoro do *plug-in*? Resposta média: 6,6; sendo 1 Ruim e 10 Bom;

o) Pergunta: Qual a sua nota geral para a interface do *plug-in*? Resposta média: 7,4; sendo 1 Ruim e 10 Boa;

p) Pergunta: Qual a sua nota geral para o *plug-in*? Resposta média: 6,6; sendo 1 Ruim e 10 Bom;

q) Pergunta: O que você achou muito bom no *plug-in*? Respostas:

- "Sonoridade." − Produtor A;
- "Distorções são sempre divertidas." − Produtor B;

r) Pergunta: O que você achou muito ruim no *plug-in*? Respostas:

- "O ruído." − Produtor D;
- "Sonoridade não me agradou." − Produtor E;

s) Pergunta: Conte um pouco sobre sua experiência ao utilizar o *plug-in*. Respostas:

- "Sonoridade próxima do real com timbre agradável." − Produtor A;
- "Se mostrou agressivo e digital, ardido, mas perfeitamente usável com outros *plug-ins* fazendo a mix. *Plug-in* para *sound designer*. Mudam o som. Cria algo novo. Isso é bom." − Produtor B;
- "Me agradou o fato da distorção dele somar no contexto da mix de forma natural, valorizando o timbre natural." − Produtor C;
- "Eu o utilizei como *boost* em uma guitarra distorcida juntamente com um simulador da Neural DSP. Ele cumpriu a sua função, porém eu senti que o ruído de artefatos desvirtuou a proposta." − Produtor D;
- "Soou ríspido e agressivo, de certa forma destrutivo." − Produtor E;

t) Pergunta: Você tem alguma sugestão para melhoria deste *plug-in*? Respostas:

- "Mais controles para timbragem." − Produtor A;
- "Use referências de grandes marcas pra ajustes de novos parâmetros." − Produtor B;
- "Aumentar a taxa de amostragem para eliminar o ruído." − Produtor D;
- "Compatibilidade de DAW Pro Tools, ser menos agressivo, menos ríspido." − Produtor E.

A partir das respostas recebidas, pôde-se notar que o *plug-in* apresentou bom funcionamento em geral. A principal crítica recebida foi referente aos ruídos gerados em frequências mais altas, mostrando uma necessidade de se utilizar *oversampling*, de modo a evitar uma distorção excessiva nessas frequências. Também foi sugerido algumas vezes a adição de outros parâmetros de controle para a distorção, mas sem especificá-los, e compatibilidade com a DAW Pro Tools. Algumas características tiveram avaliações que foram divergentes entre os produtores, como a sonoridade da distorção, que agradou a alguns e desagradou a outros.

4.2.5 WonderVerb

As respostas para as perguntas da seção introdutória foram as seguintes:

a) Pergunta: Como você avalia a utilidade do *plug-in*? Resposta média: 8,8; sendo 1 Pouco útil e 10 muito útil;

b) Pergunta: Como você avalia a identidade sonora do *plug-in*? Resposta média: 8,4; sendo 1 Ruim e 10 Boa;

c) Pergunta: O processamento soa natural ou artificial? Respostas: 80% Natural, 20% Artificial;

d) Pergunta: Como você avalia a o desempenho do *software*? Resposta média: 8,2; sendo 1 Ruim e 10 Bom;

e) Pergunta: Como você avalia a performance do *software*? Resposta média: 8,2; sendo 1 Ruim e 10 Boa;

f) Pergunta: O *plug-in* atende à proposta? Respostas: 80% Sim, 20% Parcialmente, % Não;

g) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

   - "Timbre um pouco cavernoso, pouco natural." – Produtor B;
   - "Bom timbre de *reverb*, de personalidade mais escura, e de boa sonoridade." – Produtor D;

h) Pergunta: Como você avalia a usabilidade da interface? Resposta média: 8; sendo 1 Ruim e 10 Boa;

i) Pergunta: A interface traz todos os parâmetros necessários? Respostas: 40% Sim, 60% Outros:

   - "*Reverbs* são complexos, então alguns parâmetros extras podem ajudar." – Produtor B;
   - "Gostaria de ter um filtro de graves e agudos pra ajudar na timbragem." – Produtor C;
   - "Eu senti falta de especificação do tipo de *reverb*, mas atendeu bem dentro daquilo que se propunha." – Produtor D;

j) Pergunta: O *plug-in* realiza de maneira satisfatória a mesma tarefa de outros *plug-ins* consolidados no mercado de áudio? Respostas: 80% Sim, 20% Parcialmente, % Não;

k) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas: sem respostas;

l) Pergunta: Você continuaria a utilizar este *plug-in* em suas produções musicais? Respostas: 100% Sim, 0% Não, 0% Talvez;

m) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

   - "Usaria em canais individuais pra um efeito específico." – Produtor B;
   - "O Timbre é interessante." – Produtor D;
   - "Torná-lo compatível com Pro Tools é importante!" – Produtor E;

n) Pergunta: Qual a sua nota geral para o desempenho sonoro do *plug-in*? Resposta média: 8,8; sendo 1 Ruim e 10 Bom;

o) Pergunta: Qual a sua nota geral para a interface do *plug-in*? Resposta média: 8,4; sendo 1 Ruim e 10 Boa;

p) Pergunta: Qual a sua nota geral para o *plug-in*? Resposta média: 8,4; sendo 1 Ruim e 10 Bom;

q) Pergunta: O que você achou muito bom no *plug-in*? Respostas:

- "Facilidade no uso e respostas reais." − Produtor A;
- "Simplicidade." − Produtor B;
- "Sonoridade." − Produtor D;
- "Naturalidade do *reverb*, bem trabalhado." − Produtor E;

r) Pergunta: O que você achou muito ruim no *plug-in*? Respostas:

- "Timbre ainda deixa a desejar." − Produtor B;
- "Interface/recursos." − Produtor D;
- "Não ter compatibilidade com Pro Tools." − Produtor E;

s) Pergunta: Conte um pouco sobre sua experiência ao utilizar o *plug-in*. Respostas:

- "Sonoridade agradável e facilidade no uso." − Produtor A;
- "O *plug-in* é extremamente simples e intuitivo." − Produtor B;
- "*Reverb* com profundidade muito legal!" − Produtor C;
- "Utilizei em uma trilha de solo de guitarra e logo obtive timbres bastante interessantes e boa sonoridade." − Produtor D;
- "Muito interessante a sonoridade, sem rispidez digital, cumpre muito bem o seu papel." − Produtor E;

t) Pergunta: Você tem alguma sugestão para melhoria deste *plug-in*? Respostas:

- "Tente se aproximar da referência como os *plug-ins* da Valhalla" − Produtor B;
- "Acréscimo de diferentes tipos de *reverb*." − Produtor D;
- "Estar disponível para Pro Tools." − Produtor E.

A partir das respostas recebidas, pôde-se notar que o *plug-in* apresentou bom funcionamento em geral. Para este também houve divergência quanto à sonoridade do *plug-in*, mostrando mais uma vez que essa característica tende a ter avaliação particular. Foi sugerida a adição de outros tipos de reverberação, já que este tipo de efeito possui muitas variantes, como citado no referencial teórico; também houve sugestão de tomar como referência os *plug-ins* da empresa Valhalla e tornar compatível com a DAW Pro Tools. Também pediu-se a adição de outros parâmetros de controle, sem especificação.

4.2.6 BigLoop

As respostas para as perguntas da seção introdutória foram as seguintes:

a) Pergunta: Como você avalia a utilidade do *plug-in*? Resposta média: 7; sendo 1 Pouco útil e 10 muito útil;

b) Pergunta: Como você avalia a identidade sonora do *plug-in*? Resposta média: 7,2; sendo 1 Ruim e 10 Boa;

c) Pergunta: O processamento soa natural ou artificial? Respostas: 60% Natural, 40% Artificial;

d) Pergunta: Como você avalia a o desempenho do *software*? Resposta média: 7,8; sendo 1 Ruim e 10 Bom;

e) Pergunta: Como você avalia a performance do *software*? Resposta média: 8; sendo 1 Ruim e 10 Boa;

f) Pergunta: O *plug-in* atende à proposta? Respostas: 40% Sim, 40% Parcialmente, 20% Não;

g) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

   - "Senti a falta do controle *dry/wet*." – Produtor D;

h) Pergunta: Como você avalia a usabilidade da interface? Resposta média: 6,8; sendo 1 Ruim e 10 Boa;

i) Pergunta: A interface traz todos os parâmetros necessários? Respostas: 20% Sim, 80% Outros:

- "Filtros *high pass*, *low pass* e *tap* para configuração do *timing*." – Produtor A;
- "Ainda é bem simples." – Produtor B;
- "Gostaria de um parâmetro de equalização da cauda do *delay*." – Produtor C;
- "*Tap delay*." – Produtor E;

j) Pergunta: O *plug-in* realiza de maneira satisfatória a mesma tarefa de outros *plug-ins* consolidados no mercado de áudio? Respostas: 20% Sim, 40% Parcialmente, 40% Não;

k) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

- "Em todos os *delays* sempre existe o *dry/wet*... nesse eu senti falta." – Produtor D;

l) Pergunta: Você continuaria a utilizar este *plug-in* em suas produções musicais? Respostas: 20% Sim, 40% Não, 40% Talvez;

m) Pergunta: Sinta-se à vontade para justificar sua resposta anterior. Respostas:

- "Sem *dry/wet*, não rola" – Produtor D;

n) Pergunta: Qual a sua nota geral para o desempenho sonoro do *plug-in*? Resposta média: 7,2; sendo 1 Ruim e 10 Bom;

o) Pergunta: Qual a sua nota geral para a interface do *plug-in*? Resposta média: 6,2; sendo 1 Ruim e 10 Boa;

p) Pergunta: Qual a sua nota geral para o *plug-in*? Resposta média: 6,8; sendo 1 Ruim e 10 Bom;

q) Pergunta: O que você achou muito bom no *plug-in*? Respostas:

- "Simplicidade." – Produtor B;
- "Sonoridade." – Produtor D;
- "Leve e simples." – Produtor E;

r) Pergunta: O que você achou muito ruim no *plug-in*? Respostas:

- "Limitado pra controle efetivo, mas usável sim." – Produtor B;
- "Recursos." – Produtor D;

- "Unidade de referência do *feedback* estranha, pouco intuitivo, ausência de um controle de *tap* tempo" − Produtor E;

s) Pergunta: Conte um pouco sobre sua experiência ao utilizar o *plug-in*. Respostas:

- "Configuração exige conhecimento de *timing* para *delay*, sonoridade não agradável." − Produtor A;
- "É bem simples, intuitivo, com poucas opções de *delays*, formas e timbre q precisam ser tratados posteriormente com equalizadores e etc." − Produtor B;
- "Coloração muito legal." − Produtor C;
- "Utilizei em uma faixa de guitarra e funcionou como *plug-in* de uso em paralelo, porém sem o controle de *dry/wet* foi difícil de regular, apesar da boa sonoridade das repetições." − Produtor D;
- "Ele é muito natural na resposta, rápido, porém os controles estão um pouco confusos." − Produtor E;

t) Pergunta: Você tem alguma sugestão para melhoria deste *plug-in*? Respostas:

- "*Plug-ins* de *delay* geralmente têm filtros *high pass*, *low pass* e *tap* para configuração do *timing*." − Produtor A;
- "Sempre buscar nas referências das grandes marcas." − Produtor B;
- "*Dry/wet*" − Produtor D;
- "Controle de *tap* tempo, melhorar e ajustar a interface gráfica." − Produtor E.

A partir das respostas recebidas, pôde-se notar que o *plug-in* apresentou bom funcionamento em geral. Porém, nesse *plug-in* os produtores sentiram falta de alguns parâmetros e acharam outros um pouco difíceis de serem utilizados. As principais críticas foram quanto ao controle de tempo do *delay*, que no formato de segundos dificulta a utilização para criação musical; essa questão já havia sido notada nos testes de funcionalidade executados. Outra sugestão foi o controle de *dry/wet*, que poderia ser implementado na forma de *blend*, outro ponto também notado nos testes de funcionalidade. Também foi sugerida a adição de filtros passa altas e passa baixas para a resposta do *delay*, assim como tomar como referência as grandes marcas do mercado.

Um ponto importante a ser observado é que o problema ocorrido nos testes de funcionalidade do BigLoop não foi apontado por nenhum outro usuário, levantando a hipótese de que a falha pode ser no sistema Do autor.

4.2.7 Seção de conclusão

As respostas para as perguntas da seção de conclusão foram as seguintes:

a) Pergunta: Qual a sua nota geral para o trabalho desenvolvido? Resposta média: 9,2; sendo 1 Ruim e 10 Bom;

b) Pergunta: Qual sua sugestão para outros tipos de *plug-in* que poderiam ser desenvolvidos em futuros trabalhos voltados a este tema? Respostas:

- "Masterização." − Produtor A;
- "*Plug-ins* de criação de música, como samples de instrumentos, *loops*, etc." − Produtor B;
- "Compressor, compressor multibanda, *DeEsser* e *AutoPan* são alguns que utilizo bastante e seria interessante ter opções com funcionalidade próximas as apresentadas nos *plug-ins* testados." − Produtor C;
- "Simulador de *channel strip*, compressor, simulação de amplificador." − Produtor D;
- "Compressores." − Produtor E;

c) Pergunta: Críticas, sugestões e comentários. Respostas:

- "Excelente trabalho" − Produtor A;
- "Nunca desista e continue trocando ideias com profissionais da área." − Produtor B;
- "Crítica alguma, inclusive deixo meus parabéns aos envolvidos! Se pudesse pontuar uma questão somente, seria a questão de ter os mesmos em versão AAX para quem usa Pro Tools. Obrigado." − Produtor C;
- "Muito bom o trabalho, surpreende. Sugiro manter a compatibilidade com as principais DAW do áudio, trabalhar o padrão gráfico pra diferenciar entre os *plug-ins* e

> buscar criar identidade aliada a boa sonoridade que já desenvolveram até aqui." – Produtor E.

Os profissionais tiveram grande apreço pela iniciativa do trabalho. Tendo em vista que no geral eles desconheciam qualquer trabalho acadêmico voltado a essa área, notou-se o entusiasmo em conceber avanços nas tecnologias e participar da criação destas por meio de testes e comentários. Percebe-se também que o campo para desenvolver novos trabalhos nessa área é amplo, devido à quantidade de sugestões recebidas.

4.3 Considerações finais

Ao receber as respostas da pesquisa, alguns pontos notados nos testes de funcionalidades foram convergentes com a percepção dos profissionais consultados. São eles: a falta de controles de ganho de entrada e saída no DigiQ 7B; a distorção excessiva das frequências mais altas, devida à não utilização de *oversampling* no BuzzTortion; a falta de um controle de *blend* e a necessidade de melhoria no parâmetro de controle do atraso no BigLoop.

Um ponto que deve ser priorizado em trabalhos futuros envolvendo este tema é a obtenção da assinatura digital para utilização dos *plug-ins* no formato AAX. A necessidade de se ter os *plug-ins* neste formato, compatível com a DAW Pro Tools, foi apontada por diversas vezes pelos produtores.

Percebeu-se que algumas características do processamento são de questão muito particular, cada produtor tem uma preferência quanto à forma de se trabalhar e teve uma percepção diferente quanto aos mesmos pontos avaliados. Isso mostra que a qualidade de processamento de áudio não é uma característica totalmente objetiva, é também subjetiva.

Um ponto importante a se enfatizar é a qualidade das observações feitas pelos produtores, pessoas que estão habituadas a trabalhar com as ferramentas semelhantes às desenvolvidas. Isso confirma a importância de se aumentar o contato entre pesquisadores e profissionais de mercado, facilitando e melhorando o direcionamento das pesquisas. Os comentários recebidos deram uma direção bem clara de quais devem ser os próximos passos a serem seguidos, tanto para melhorar a qualidade dos *plug-ins* já desenvolvidos, quanto para desenvolvimento de novos *plug-ins*. Devido ao grande interesse despertado nos profissionais, percebeu-se que essa é uma área que pode ser interessante para desenvolver mais estudos, visto que há grande demanda por novas tecnologias.

5 CONCLUSÃO

O mercado do áudio está sempre em busca de inovações e novas soluções que facilitem o processo produtivo. O objetivo dessa busca é deixar os artistas mais desprendidos, liberando-os para dedicar a maior parte do tempo a serem criativos. Ferramentas como as desenvolvidas neste trabalho vão ao encontro desse objetivo, tornando as tarefas do processo mais simples e intuitivas.

De modo geral, pode-se afirmar que o principal objetivo deste trabalho foi atingido com sucesso: desenvolver DSPs para áudio voltados à produção musical na UFLA, buscando no mercado as direções a serem seguidas e servindo de base para o desenvolvimento de possíveis trabalhos futuros.

Os próximos passos a serem dados ficaram claros a partir dos resultados da pesquisa realizada com os produtores musicais. As respostas recebidas indicaram as direções a serem seguidas e mostraram a grande necessidade e interesse por trabalhos na área de DSP voltado ao áudio.

REFERÊNCIAS

ABOUT Freesound. 2021. Disponível em: <https://freesound.org/help/about/>. Acesso em: 10 nov. 2021.

ADAM, V. Loudspeaker behaviour under incident sound fields. 01 2002.

BALLOU, G. M. et al. **Handbook for Sound Engineers**. 4. ed. [S.l.]: Focal Press, 2008.

BRISTOW-JOHNSON, R. The equivalence of various methods of computing biquad coefficients for audio parametric equalizers. 11 2001.

DAVIS, D.; PATRONIS, J. E.; BROWN, P. **Sound System Engineering**. 4. ed. [S.l.]: Focal Press, 2013.

DENIS. **Música: Ondas Mecânicas que se transformam em Arte**. 2019. Disponível em: <https://blog.aprovatotal.com.br/musica-ondas-mecanicas/>. Acesso em: 27 nov. 2021.

EVEREST, F. A.; POHLMANN, K. C. **Master Handbook of Acoustics**. 5. ed. [S.l.]: Mc Graw Hill, 2009.

FUNçõES trigonométricas inversas. 2021. Disponível em: <https://pt.wikipedia.org/wiki/Fun%C3%A7%C3%B5es_trigonom%C3%A9tricas_inversas>. Acesso em: 15 nov. 2021.

HERRERA, C. G. **Projeto de Sistemas de Processamento Digital de Sinais de Áudio Utilizando Metodologia Científica**. Dissertação (Mestrado em Engenharia Elétrica) — Universidade Federal de Minas Gerais, 2004.

JOUAUD, L. **The analog delay, or rather… the analog-voiced delay**. 2020. Disponível em: <https://anasounds.com/analog-delay-analog-voiced-delay/>. Acesso em: 08 nov. 2021.

JUCE. 2021. Disponível em: <https://github.com/juce-framework/JUCE>. Acesso em: 24 oct. 2021.

KAHRS, M.; BRANDENBURG, K. **Applications of Digital Signal Processing to Audio and Acoustics**. 1. ed. [S.l.]: Kluwer Academic Publishers, 2002.

LATHI, B. P. **Sinais e Sistemas Lineares**. 2. ed. [S.l.]: Bookman, 2008.

LEE, T. G. **Bem-vindo ao IDE do Visual Studio**. 2021. Disponível em: <https://docs.microsoft.com/pt-br/visualstudio/get-started/visual-studio-ide?view=vs-2022>. Acesso em: 10 nov. 2021.

MANOLAKIS, D. G.; INGLE, V. K. **Applied Digital Signal Processing Theory and Practice**. 1. ed. [S.l.]: Cambridge, 2011.

MATIAS, J. **A música como manifestação cultural e sua massificação**. 2020. Disponível em: <http://jornalismojunior.com.br/a-musica-como-manifestacao-cultural-e-sua-massificacao/>. Acesso em: 27 nov. 2021.

MIDI. 2021. Disponível em: <https://en.wikipedia.org/wiki/MIDI>. Acesso em: 27 nov. 2021.

MITCHELL, P. W.; SANTIAGO, G. L. A. **Gravação e Reprodução do Som**. 2016. Disponível em: <https://www.portalsaofrancisco.com.br/fisica/gravacao-e-reproducao-do-som>. Acesso em: 28 mar. 2021.

Moura, J. What is Signal Processing? [President's Message]. **IEEE Signal Processing Magazine**, v. 26, n. 6, p. 6–6, 2009.

NAREDO, L. et al. z - transform - based methods for electromagnetic transient simulations. **Power Delivery, IEEE Transactions on**, p. 1799 – 1805, 08 2007.

OLIVEIRA, T. C. de A. **Modelagem Computacional de Amplificadores Valvulados**. Tese (Doutorado em Engenharia Elétrica) — Universidade Estadual de Campinas, 2013.

PARKER, M. **Digital Signal Processing Everything you need to know to get started**. 1. ed. [S.l.]: Newnes, 2010.

PAYNTER, R. T. **Introductory Electronic Devices and Circuits**. 2010. Disponível em: <https://wps.prenhall.com/chet_paynter_introduct_6/0,5779,426214-,00.html>. Acesso em: 27 oct. 2021.

PRO Tools. 2021. Disponível em: <https://en.wikipedia.org/wiki/Pro_Tools>. Acesso em: 27 nov. 2021.

PUCKETTE, M. **The Theory and Technique of Electronic Music**. [S.l.: s.n.], 2006.

ROBINSON, M. **Getting started with JUCE**. 2013. Disponível em: <https://www.packtpub.com/product/getting-started-with-juce/9781783283316>. Acesso em: 24 oct. 2021.

SMITH, J. O. **Introduction to Digital Filters with Audio Applications**. [S.l.]: `http://-ccrma.stanford.edu/~jos/filters/`, 2007. Online book. Acesso em: 15 nov. 2021.

VäLIMäKI, V.; REISS, J. D. All about audio equalization: Solutions and frontiers. **Applied Sciences**, v. 6, n. 5, 2016. Disponível em: <https://www.mdpi.com/2076-3417/6/5/129>.

WATKINSON, J. **The Art of Digital Audio**. 3. ed. [S.l.]: Focal Press, 2001.

WICKERT, M. **Signals and Systems for Dummies**. 1. ed. [S.l.]: John Wiley & Sons, Inc, 2013.

XCODE. 2021. Disponível em: <https://developer.apple.com/documentation/xcode>. Acesso em: 10 nov. 2021.

ZÖLZER, U. **Digital Audio Signal Processing**. 2. ed. [S.l.]: John Wiley & Sons, Inc, 2008.

ZÖLZER, U. et al. **DAFX: Digital Audio Effects**. 2. ed. [S.l.]: John Wiley and Sons, 2011.

APÊNDICE A – Código do DigiQ 7B

Abaixo o código completo do arquivo PluginProcessor.h:

```
#pragma once

#include <JuceHeader.h>

//==============================================================================
class DigiQ7BAudioProcessor  : public juce::AudioProcessor
{
public:
    //==========================================================================
    DigiQ7BAudioProcessor();
    ~DigiQ7BAudioProcessor() override;

    //==========================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock)
        override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts)
        const override;
   #endif

    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&)
        override;

    //==========================================================================
    juce::AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==========================================================================
    const juce::String getName() const override;
```

```
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const juce::String getProgramName (int index) override;
    void changeProgramName (int index, const juce::String& newName)
        override;

    //==============================================================
    void getStateInformation (juce::MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes)
        override;

    //==============================================================
    void updateFilter();

    //==============================================================
    juce::AudioProcessorValueTreeState::ParameterLayout
        createParameterLayout();
    juce::AudioProcessorValueTreeState& getAPVTS();
    static juce::String getBandID(size_t index);
    static float DigiQ7BAudioProcessor::getBandFreq(size_t index);

private:
    //==============================================================
    //  rvore  de valores usada para gerenciar todo o estado do
    // AudioProcessor
    juce::AudioProcessorValueTreeState apvts;
```

```
    // multiplicando o processador pelo n mero de canais usados
    using eqBand = juce::dsp::ProcessorDuplicator
        <juce::dsp::IIR::Filter<float>,
        juce::dsp::IIR::Coefficients<float>>;

    // alocando todos os DSP's para serem processados em conjunto
    juce::dsp::ProcessorChain <eqBand, eqBand, eqBand, eqBand, eqBand,
        eqBand, eqBand> processorChain;

    // armazenar  o  ndice  da banda do equalizador
    int eqBandIndex;

    // armazenar  frequ ncia de amostragem
    float lastSampleRate;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR
        (DigiQ7BAudioProcessor)
};
```

Abaixo o código completo do arquivo PluginProcessor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================
// fun  o que cria o layout dos par metros e aloca em um vetor
juce::AudioProcessorValueTreeState::ParameterLayout
    DigiQ7BAudioProcessor::createParameterLayout()
{
    std::vector <std::unique_ptr <juce::AudioProcessorParameterGroup>>
        params;

    const float minGain = juce::Decibels::decibelsToGain(-18.f);
    const float maxGain = juce::Decibels::decibelsToGain(18.f);
```

```
for (size_t i = 0; i < 7; i++)
{
    auto gain = std::make_unique<juce::AudioParameterFloat>
        ("gain" + getBandID(i), // ID
        TRANS("Gain" + getBandID(i)), // Nome
        juce::NormalisableRange<float> {1.f / maxGain, maxGain,
            0.001f, std::log(.5f) / std::log((1.f - minGain) /
            (maxGain - minGain))}, // range
        1.0f, // Valor Padr o
        juce::String(), // Legenda (opcional)
        // Categoria (opcional)
        juce::AudioProcessorParameter::genericParameter,
        // converte um valor n o normalizado para uma string com
        //tamanho limitado, usado pelos hosts para mostrar
        //o valor
        [](float value, int) {return juce::String
            (juce::Decibels::gainToDecibels(value), 1) + " dB"; },
        // o contr rio do de cima, para o usu rio escrever um
        // valor para o par metro
        [](juce::String text) {return juce::Decibels::
            decibelsToGain(text.dropLastCharacters(3).
            getFloatValue()); });

    auto frequency = std::make_unique<juce::AudioParameterFloat>
        ("frequency" + getBandID(i),
        "Frequency" + getBandID(i),
        juce::NormalisableRange<float> {20.0f, 20000.0, 0.0f,
        1.0f / std::log2(1.0f + std::sqrt(20000.0f / 20.0f))},
        getBandFreq(i),
        juce::String(),
        juce::AudioProcessorParameter::genericParameter,
        [](float value, int) { return juce::String(value, 0) +
            " Hz"; },
        [](const juce::String& text) { return
```

```
                text.getFloatValue(); });

        juce::NormalisableRange<float> qualityRange{ 0.025f, 40.0f,
            0.0f };
        qualityRange.setSkewForCentre(1.f / std::sqrt(2.f));
        auto quality = std::make_unique<juce::AudioParameterFloat>
            ("quality" + getBandID(i),
            "Quality" + getBandID(i),
            qualityRange,
            1.0f / std::sqrt(2.0f),
            juce::String(),
            juce::AudioProcessorParameter::genericParameter,
            [](float value, int) { return juce::String(value, 3); },
            [](const juce::String& text) {
                return text.getFloatValue(); });


        //cria um grupo de parametros
        auto group = std::make_unique
            <juce::AudioProcessorParameterGroup>("band" + getBandID(i),
            "Band" + getBandID(i), // groupName
            "|", // Separador de subgrupo
            std::move(gain), // parametro filho
            std::move(frequency),
            std::move(quality));


        params.push_back(std::move(group));
    }

    return { params.begin(), params.end() };
}

juce::AudioProcessorValueTreeState& DigiQ7BAudioProcessor::getAPVTS()
    { return apvts; }
```

```
juce::String DigiQ7BAudioProcessor::getBandID(size_t index)
{
    switch (index)
    {
    case 0: return "Sub";
    case 1: return "Low";
    case 2: return "LowMid";
    case 3: return "Mid";
    case 4: return "MidHigh";
    case 5: return "High";
    case 7: return "Brigth";
    default: break;
    }
    return "unknown";
}

float DigiQ7BAudioProcessor::getBandFreq(size_t index)
{
    switch (index)
    {
    case 0: return 50;
    case 1: return 150;
    case 2: return 400;
    case 3: return 1000;
    case 4: return 2500;
    case 5: return 7500;
    case 6: return 15000;
    default: break;
    }
    return 0;
}

//==============================================================
```

```
DigiQ7BAudioProcessor::DigiQ7BAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                      #if ! JucePlugin_IsMidiEffect
                       #if ! JucePlugin_IsSynth
                        .withInput  ("Input",  juce::AudioChannelSet::
                             stereo(), true)
                       #endif
                        .withOutput ("Output", juce::AudioChannelSet::
                             stereo(), true)
                      #endif
                        ),
#endif

    // adiciona os parmetros na treeState para serem
    // automatizados na daw
    apvts(*this, nullptr, juce::Identifier("PARAMETER"),
        createParameterLayout())
{
}

DigiQ7BAudioProcessor::~DigiQ7BAudioProcessor()
{
}

//==============================================================================
const juce::String DigiQ7BAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool DigiQ7BAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
```

```
    return true;
   #else
    return false;
   #endif
}

bool DigiQ7BAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool DigiQ7BAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}

double DigiQ7BAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int DigiQ7BAudioProcessor::getNumPrograms()
{
    return 1;   // NB: some hosts don't cope very well if you tell
                // them there are 0 programs,
                // so this should be at least 1, even if you're not
```

```
                //really implementing programs.
}

int DigiQ7BAudioProcessor::getCurrentProgram()
{
    return 0;
}

void DigiQ7BAudioProcessor::setCurrentProgram (int index)
{
}

const juce::String DigiQ7BAudioProcessor::getProgramName (int index)
{
    return {};
}

void DigiQ7BAudioProcessor::changeProgramName (int index,
    const juce::String& newName)
{
}

//==============================================================================
void DigiQ7BAudioProcessor::prepareToPlay (double sampleRate,
    int samplesPerBlock)
{
    lastSampleRate = sampleRate;

    // faz a comunica  o entre plugin e m dulo dsp
    juce::dsp::ProcessSpec spec;

    spec.sampleRate = lastSampleRate;
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = getMainBusNumOutputChannels();
```

```
    processorChain.reset(); // evita valores de lixo
    updateFilter();
    // inicia o processor duplicator com as propriedades passadas
    processorChain.prepare(spec);
}

void DigiQ7BAudioProcessor::releaseResources()
{
    // When playback stops, you can use this as an opportunity to
    // free up any spare memory, etc.
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool DigiQ7BAudioProcessor::isBusesLayoutSupported (
    const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    juce::ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions,
    // will only load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::mono()
        && layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::stereo())
        return false;

    // This checks if the input layout matches the output layout
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() !=
```

```
        layouts.getMainInputChannelSet())
        return false;
   #endif

    return true;
  #endif
}
#endif

// atualiza o estado do filtro
void DigiQ7BAudioProcessor::updateFilter()
{
    for (size_t i = 0; i < 7; i++)
    {
        eqBandIndex = i;
        float freq = *apvts.getRawParameterValue(
            "frequency" + getBandID(i)); // frequencia central
        float qual = *apvts.getRawParameterValue(
            "quality" + getBandID(i)); // qualidade
        float gain = *apvts.getRawParameterValue(
            "gain" + getBandID(i)); // qualidade

        if (i == 0) *processorChain.get<0>().state =
            *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
            lastSampleRate, freq, qual, gain);
        else if (i == 1) *processorChain.get<1>().state =
            *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
            lastSampleRate, freq, qual, gain);
        else if (i == 2) *processorChain.get<2>().state =
            *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
            lastSampleRate, freq, qual, gain);
        else if (i == 3) *processorChain.get<3>().state =
            *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
            lastSampleRate, freq, qual, gain);
```

```
        else if (i == 4) *processorChain.get<4>().state =
            *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
            lastSampleRate, freq, qual, gain);
        else if (i == 5) *processorChain.get<5>().state =
            *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
            lastSampleRate, freq, qual, gain);
        else if (i == 6) *processorChain.get<6>().state =
            *juce::dsp::IIR::Coefficients<float>::makePeakFilter(
            lastSampleRate, freq, qual, gain);
    }
}

void DigiQ7BAudioProcessor::processBlock (
    juce::AudioBuffer<float>& buffer,
    juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels;
        ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

    // como est  sendo utilizado o juce_dsp, n o    necess rio
    // fazer itera  es de processamento aqui,
    // tudo    realizado pelo dsp

    // o bloco de  udio     o buffer
    juce::dsp::AudioBlock<float> block(buffer);
    updateFilter();
    // processa o bloco por meio do dsp
    processorChain.process(
        juce::dsp::ProcessContextReplacing<float>(block));
```

```
}

//==============================================================
bool DigiQ7BAudioProcessor::hasEditor() const
{
    // change this to false if you choose to not supply an editor
    return true;
}

juce::AudioProcessorEditor* DigiQ7BAudioProcessor::createEditor()
{
    return new DigiQ7BAudioProcessorEditor (*this);
}

//==============================================================
void DigiQ7BAudioProcessor::getStateInformation (
    juce::MemoryBlock& destData)
{
    // o bloco de mem ria no qual o estado ser  escrito
    juce::MemoryOutputStream stream(destData, false);
    // escrevendo no bloco de mem ria
    apvts.state.writeToStream(stream);
}

void DigiQ7BAudioProcessor::setStateInformation (
    const void* data, int sizeInBytes)
{
    // lendo do bloco de mem ria
    juce::ValueTree tree = juce::ValueTree::readFromData(
        data, sizeInBytes);
    // passando os estados para a apvts
    if (tree.isValid()) apvts.state = tree;
}
```

```
//==============================================================================
// This creates new instances of the plugin..
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new DigiQ7BAudioProcessor();
}
```

Abaixo o código completo do arquivo EqBandEditor.h:

```
#pragma once

#include <JuceHeader.h>
#include "PluginProcessor.h"

//==============================================================================
class EqBandEditor : public juce::Component
{
public:
    EqBandEditor(DigiQ7BAudioProcessor&, size_t);
    ~EqBandEditor() override;

    void paint(juce::Graphics&) override;
    void resized() override;

private:
    //==============================================================
    size_t bandID;

    juce::Slider gainSlider;
    juce::Slider freqSlider;
    juce::Slider qSlider;

    juce::OwnedArray<juce::AudioProcessorValueTreeState::
        SliderAttachment> sliderAttachments;

    juce::Rectangle<int> area;
```

```
    int textPosition;
    int textHeight = 30;
    int textWidth = 200;


    DigiQ7BAudioProcessor& processor;

    //==============================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(EqBandEditor)
};
```

Abaixo o código completo do arquivo EqBandEditor.cpp:

```
#include "EqBandEditor.h"

//==============================================================================
EqBandEditor::EqBandEditor(DigiQ7BAudioProcessor& p, size_t i)
    : processor(p)
{
    bandID = i;

    addAndMakeVisible(gainSlider);
    gainSlider.setSliderStyle(juce::Slider::Rotary);
    gainSlider.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
    gainSlider.setColour(juce::Slider::backgroundColourId,
        juce::Colour(0x00000000));
    gainSlider.setColour(juce::Slider::thumbColourId,
        juce::Colours::yellow);
    gainSlider.setColour(juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        processor.getAPVTS(),
        "gain" + processor.getBandID(i), gainSlider));
```

```
    addAndMakeVisible(freqSlider);
    freqSlider.setSliderStyle(juce::Slider::Rotary);
    freqSlider.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
    freqSlider.setColour(juce::Slider::backgroundColourId,
        juce::Colour(0x00000000));
    freqSlider.setColour(juce::Slider::thumbColourId,
        juce::Colours::yellow);
    freqSlider.setColour(juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        processor.getAPVTS(),
        "frequency" + processor.getBandID(i), freqSlider));

    addAndMakeVisible(qSlider);
    qSlider.setSliderStyle(juce::Slider::Rotary);
    qSlider.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
    qSlider.setColour(juce::Slider::backgroundColourId,
        juce::Colour(0x00000000));
    qSlider.setColour(juce::Slider::thumbColourId,
        juce::Colours::yellow);
    qSlider.setColour(juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        processor.getAPVTS(),
        "quality" + processor.getBandID(i), qSlider));
}

EqBandEditor::~EqBandEditor()
{
}
```

```
void EqBandEditor::paint(juce::Graphics& g)
{
    g.fillAll(juce::Colour(0xff323e44));

    area = getLocalBounds();

    //ret ngulo arredondado desenhado
    {
        area.removeFromTop(10);
        area.removeFromBottom(10);
        area.removeFromLeft(10);
        area.removeFromRight(10);

        juce::Colour strokeColour = juce::Colours::azure;

        g.setColour(strokeColour);
        g.drawRoundedRectangle(area.getX(), area.getY(),
            area.getWidth(), area.getHeight(), 10.f, .2f);
    }

    //texto band
    {
        int x = proportionOfWidth(0.5000f) - (200 / 2);
        // pega a p o s i  o inicial da area que sobrou
        int y = area.getY();
        int width = 200;
        int height = textHeight;

        juce::String text(TRANS("Band ") +
            juce::String(bandID + 1));
        juce::Colour fillColour = juce::Colours::azure;

        g.setColour(fillColour);
```

```
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Freq.
{
    int x = area.getX();
    int y = textPosition;
    int width = area.getWidth() / 2;
    int height = textHeight;

    juce::String text(TRANS("Freq."));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Q
{
    int x = area.getX() + area.getWidth() / 2;
    int y = textPosition;
    int width = area.getWidth() / 2;
    int height = textHeight;

    juce::String text(TRANS("Q"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
```

```
        g.setFont(juce::Font("Bradesco Sans", 15.00f,
            juce::Font::plain).withTypefaceStyle("SemiBold"));
        g.drawText(text, x, y, width, height,
            juce::Justification::centred, true);
    }
}

void EqBandEditor::resized()
{
    area = getLocalBounds();
    area.removeFromTop(10);
    area.removeFromBottom(10);
    area.removeFromLeft(10);
    area.removeFromRight(10);

    area.removeFromTop(textHeight);
    area.removeFromBottom(10);


    const int gainHeight = (area.getHeight() * 2 / 3) - 15;
    gainSlider.setBounds(area.removeFromTop(gainHeight));

    textPosition = gainSlider.getBottom();

    area.removeFromTop(textHeight);

    const int freqWidth = area.getWidth() / 2;
    freqSlider.setBounds(area.removeFromLeft(freqWidth));

    qSlider.setBounds(area.removeFromLeft(freqWidth));

    textWidth = freqWidth;
}
```

Abaixo o código completo do arquivo PluginEditor.h:

```
#pragma once

#include <JuceHeader.h>
#include "PluginProcessor.h"
#include "EqBandEditor.h"

//==============================================================================
/**
*/
class DigiQ7BAudioProcessorEditor : public juce::AudioProcessorEditor
{
public:
    DigiQ7BAudioProcessorEditor (DigiQ7BAudioProcessor&);
    ~DigiQ7BAudioProcessorEditor() override;

    //==============================================================================
    void paint (juce::Graphics&) override;
    void resized() override;

private:
    DigiQ7BAudioProcessor& audioProcessor;

    EqBandEditor eqBand1, eqBand2, eqBand3, eqBand4, eqBand5,
        eqBand6, eqBand7;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
        DigiQ7BAudioProcessorEditor)
};
```

Abaixo o código completo do arquivo PluginEditor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
DigiQ7BAudioProcessorEditor::DigiQ7BAudioProcessorEditor (
```

```
    DigiQ7BAudioProcessor& p)
    : AudioProcessorEditor(&p), audioProcessor(p),
    eqBand1(p, 0), eqBand2(p, 1), eqBand3(p, 2), eqBand4(p, 3),
    eqBand5(p, 4), eqBand6(p, 5), eqBand7(p, 6)
{
    setSize(1200, 400);
    setResizable(true, false);

    addAndMakeVisible(&eqBand1);
    addAndMakeVisible(&eqBand2);
    addAndMakeVisible(&eqBand3);
    addAndMakeVisible(&eqBand4);
    addAndMakeVisible(&eqBand5);
    addAndMakeVisible(&eqBand6);
    addAndMakeVisible(&eqBand7);
}

DigiQ7BAudioProcessorEditor::~DigiQ7BAudioProcessorEditor()
{
}

//==============================================================================
void DigiQ7BAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll(juce::Colours::black);
}

void DigiQ7BAudioProcessorEditor::resized()
{
    juce::Rectangle<int> area = getLocalBounds();
    const int bandWidth = getWidth() / 7;

    eqBand1.setBounds(area.removeFromLeft(bandWidth));
    eqBand2.setBounds(area.removeFromLeft(bandWidth));
```

```
        eqBand3.setBounds(area.removeFromLeft(bandWidth));
        eqBand4.setBounds(area.removeFromLeft(bandWidth));
        eqBand5.setBounds(area.removeFromLeft(bandWidth));
        eqBand6.setBounds(area.removeFromLeft(bandWidth));
        eqBand7.setBounds(area.removeFromLeft(bandWidth));
    }
```

APÊNDICE B – Código do BuzzTortion

Abaixo o código completo do arquivo PluginProcessor.h:

```
#pragma once

#include <JuceHeader.h>

//==============================================================================
class BuzzTortionAudioProcessor  : public juce::AudioProcessor
{
public:
    //==============================================================
    BuzzTortionAudioProcessor();
    ~BuzzTortionAudioProcessor() override;

    //==============================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock)
        override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts)
        const override;
   #endif

    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&)
        override;

    //==============================================================
    juce::AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================
    const juce::String getName() const override;
```

```
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const juce::String getProgramName (int index) override;
    void changeProgramName (int index, const juce::String& newName)
        override;

    //==============================================================
    void getStateInformation (juce::MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes)
        override;

    //==============================================================
    void updateWaveshaper();

    //==============================================================
    juce::AudioProcessorValueTreeState::ParameterLayout
        createParameterLayout();
    juce::AudioProcessorValueTreeState& getAPVTS();

private:
    //==============================================================
    //  rvore  de valores usada para gerenciar todo o estado do
    // AudioProcessor
    juce::AudioProcessorValueTreeState apvts;
    // alocando todos o DSP no ProcessorChain
    juce::dsp::ProcessorChain<juce::dsp::WaveShaper<double,
        std::function<double(double)>>> processorChain;
```

```
    // armazenar  frequ ncia de amostragem
    float lastSampleRate;

    //==============================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
        BuzzTortionAudioProcessor)
};
```

Abaixo o código completo do arquivo PluginProcessor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"
//==============================================================

juce::AudioProcessorValueTreeState::ParameterLayout
    BuzzTortionAudioProcessor::createParameterLayout()
{
    std::vector <std::unique_ptr <juce::RangedAudioParameter>>
        params;

    const float minGain = juce::Decibels::decibelsToGain(-6.f);
    const float maxGain = juce::Decibels::decibelsToGain(6.f);

    auto inputGain = std::make_unique<juce::AudioParameterFloat>
        ("inputGain", // ID
            TRANS("Input Gain"), // Nome
            juce::NormalisableRange<float> { //Range
                1.f / maxGain, maxGain, 0.001f, std::log(.5f) /
                std::log((1.f - minGain) / (maxGain - minGain))},
            1.0f,
            juce::String(), // Legenda (opcional)
            // Categoria (opcional)
            juce::AudioProcessorParameter::genericParameter,
            // converte um valor n o normalizado para uma string
            // com tamanho limitado, usado pelos hosts para mostrar
            // o valor
```

```
        [](float value, int) {return juce::String(
            juce::Decibels::gainToDecibels(value), 1) + " dB"; },
        // o contrário do de cima, para o usuário escrever um
        // valor para o parâmetro
        [](juce::String text) {
            return juce::Decibels::decibelsToGain(
            text.dropLastCharacters(3).getFloatValue()); }
);
params.push_back(std::move(inputGain));

auto drive = std::make_unique<juce::AudioParameterFloat>
    ("drive",
        TRANS("drive"),
        juce::NormalisableRange<float> {1.f, 100.f, 0.1},
        50.0f, // Valor Padrão
        juce::String(),
        juce::AudioProcessorParameter::genericParameter,
        [](float value, int) {
            return juce::String(value, 0) + "%"; },
        [](const juce::String& text) {
            return text.getFloatValue(); });
params.push_back(std::move(drive));

auto blend = std::make_unique<juce::AudioParameterFloat>
    ("blend",
        TRANS("Blend"),
        juce::NormalisableRange<float> {0.f, 1.f, 0.0001},
        .5f,
        juce::String(),
        juce::AudioProcessorParameter::genericParameter,
        [](float value, int) {
            return juce::String(value * 100, 0) + "%"; },
        [](const juce::String& text) {
            return text.getFloatValue() / 100; });
```

```
    params.push_back(std::move(blend));

    auto outputGain = std::make_unique<juce::AudioParameterFloat>
        ("outputGain",
            TRANS("Output Gain"),
            juce::NormalisableRange<float> {
                1.f / maxGain, maxGain, 0.001f, std::log(.5f) /
                std::log((1.f - minGain) / (maxGain - minGain))},
            1.0f,
            juce::String(), // Legenda (opcional)
            juce::AudioProcessorParameter::genericParameter,
            [](float value, int) {return juce::String(
                juce::Decibels::gainToDecibels(value), 1) + " dB"; },
            [](juce::String text) {
                return juce::Decibels::decibelsToGain(
                    text.dropLastCharacters(3).getFloatValue()); }
    );
    params.push_back(std::move(outputGain));

    return { params.begin(), params.end() };

}

juce::AudioProcessorValueTreeState&
    BuzzTortionAudioProcessor::getAPVTS() { return apvts; }

//==============================================================
BuzzTortionAudioProcessor::BuzzTortionAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",
                        juce::AudioChannelSet::stereo(), true)
```

```
                        #endif
                         .withOutput ("Output",
                          juce::AudioChannelSet::stereo(), true)
                       #endif
                         ),
#endif

    // adiciona os par metros na treeState para serem
    // automatizados na daw
    apvts(*this, nullptr, juce::Identifier("PARAMETER"),
        createParameterLayout())
{
}

BuzzTortionAudioProcessor::~BuzzTortionAudioProcessor()
{
}

//==============================================================================
const juce::String BuzzTortionAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool BuzzTortionAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool BuzzTortionAudioProcessor::producesMidi() const
```

```
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool BuzzTortionAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}

double BuzzTortionAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int BuzzTortionAudioProcessor::getNumPrograms()
{
    return 1;   // NB: some hosts don't cope very well if you tell
                // them there are 0 programs,
                // so this should be at least 1, even if you're not
                // really implementing programs.
}

int BuzzTortionAudioProcessor::getCurrentProgram()
{
    return 0;
}
```

```
void BuzzTortionAudioProcessor::setCurrentProgram (int index)
{
}

const juce::String BuzzTortionAudioProcessor::getProgramName (
    int index)
{
    return {};
}

void BuzzTortionAudioProcessor::changeProgramName (
    int index, const juce::String& newName)
{
}

//==============================================================
void BuzzTortionAudioProcessor::prepareToPlay (
    double sampleRate, int samplesPerBlock)
{
    lastSampleRate = sampleRate;

    // feito para comunica  o entre plugin e dsp
    juce::dsp::ProcessSpec spec;
    spec.sampleRate = lastSampleRate;
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = getMainBusNumOutputChannels();

    // evita valores de lixo
    processorChain.reset();
    updateWaveshaper();
    // inicia o processor duplicator com as propriedades passadas
    processorChain.prepare(spec);
}
```

```
void BuzzTortionAudioProcessor::releaseResources()
{
    // When playback stops, you can use this as an opportunity to
    // free up any spare memory, etc.
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool BuzzTortionAudioProcessor::isBusesLayoutSupported (
    const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    juce::ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions,
    // will only load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::mono()
        && layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::stereo())
        return false;

    // This checks if the input layout matches the output layout
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() !=
        layouts.getMainInputChannelSet())
        return false;
   #endif

    return true;
  #endif
```

```
}
#endif

void BuzzTortionAudioProcessor::updateWaveshaper()
{
    // buscando os valores na apvts
    float inputGain = *apvts.getRawParameterValue("inputGain");
    float drive = *apvts.getRawParameterValue("drive");
    float blend = *apvts.getRawParameterValue("blend");
    float outputGain = *apvts.getRawParameterValue("outputGain");

    // atualizando o estado do waveshaper
    auto& waveshaper = processorChain.get<0>();
    waveshaper.functionToUse =
        [inputGain, drive, blend, outputGain](double in)
    {
        // fun  o que aplica a distor  o no sinal original
        float distortedSignal = inputGain * atan(drive * in) /
            (juce::float_Pi * std::pow(drive + .5f, .4f));
        // fun  o que mescla o sinal distorcido com o original
        float out = ((distortedSignal * blend) +
            (in * (1.f - blend))) * outputGain;
        return out;
    };
}

void BuzzTortionAudioProcessor::processBlock (
    juce::AudioBuffer<float>& buffer,
    juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
```

```
    for (auto i = totalNumInputChannels; i <
        totalNumOutputChannels; ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

    // o bloco de  udio     o buffer
    juce::dsp::AudioBlock<float> block(buffer);
    updateWaveshaper();
    // processa o bloco por meio do dsp
    processorChain.process(
        juce::dsp::ProcessContextReplacing<float>(block));
}

//==============================================================================
bool BuzzTortionAudioProcessor::hasEditor() const
{
    // (change this to false if you choose to not supply an editor)
    return true;
}

juce::AudioProcessorEditor*
    BuzzTortionAudioProcessor::createEditor()
{
    return new BuzzTortionAudioProcessorEditor (*this);
}

//==============================================================================
void BuzzTortionAudioProcessor::getStateInformation (
    juce::MemoryBlock& destData)
{
    juce::MemoryOutputStream stream(destData, false);
    apvts.state.writeToStream(stream);
}

void BuzzTortionAudioProcessor::setStateInformation (
```

```
    const void* data, int sizeInBytes)
{
    juce::ValueTree tree = juce::ValueTree::readFromData(
        data, sizeInBytes);
    if (tree.isValid()) apvts.state = tree;
}

//==============================================================
// This creates new instances of the plugin..
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new BuzzTortionAudioProcessor();
}
```

Abaixo o código completo do arquivo PluginEditor.h:

```
#pragma once

#include <JuceHeader.h>
#include "PluginProcessor.h"

//==============================================================
class BuzzTortionAudioProcessorEditor
    : public juce::AudioProcessorEditor
{
public:
    BuzzTortionAudioProcessorEditor (BuzzTortionAudioProcessor&);
    ~BuzzTortionAudioProcessorEditor() override;

    //==============================================================
    void paint (juce::Graphics&) override;
    void resized() override;

private:
    juce::Slider inputGainKnob;
    juce::Slider driveKnob;
```

```
    juce::Slider blendKnob;
    juce::Slider outputGainKnob;

    juce::Rectangle<int> area;

    int textPosition;
    int textHeight = 30;
    int textWidth = 200;

    juce::OwnedArray<
        juce::AudioProcessorValueTreeState::SliderAttachment>
        sliderAttachments;

    BuzzTortionAudioProcessor& audioProcessor;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
        BuzzTortionAudioProcessorEditor)
};
```

Abaixo o código completo do arquivo PluginEditor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================
BuzzTortionAudioProcessorEditor::BuzzTortionAudioProcessorEditor (
    BuzzTortionAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    setSize(600, 300);
    setResizable(true, false);

    addAndMakeVisible(inputGainKnob);
    inputGainKnob.setSliderStyle(juce::Slider::Rotary);
    inputGainKnob.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
```

```
inputGainKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
inputGainKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
inputGainKnob.setColour(
    juce::Slider::textBoxOutlineColourId,
    juce::Colour(0x00000000));
sliderAttachments.add(
    new juce::AudioProcessorValueTreeState::SliderAttachment(
    audioProcessor.getAPVTS(), "inputGain", inputGainKnob));

addAndMakeVisible(driveKnob);
driveKnob.setSliderStyle(juce::Slider::Rotary);
driveKnob.setTextBoxStyle(
    juce::Slider::TextBoxBelow, false, 80, 20);
driveKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
driveKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
driveKnob.setColour(
    juce::Slider::textBoxOutlineColourId,
    juce::Colour(0x00000000));
sliderAttachments.add(
    new juce::AudioProcessorValueTreeState::SliderAttachment(
    audioProcessor.getAPVTS(), "drive", driveKnob));

addAndMakeVisible(blendKnob);
blendKnob.setSliderStyle(juce::Slider::Rotary);
blendKnob.setTextBoxStyle(
    juce::Slider::TextBoxBelow, false, 80, 20);
blendKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
blendKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
```

```
    blendKnob.setColour(
        juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        audioProcessor.getAPVTS(), "blend", blendKnob));

    addAndMakeVisible(outputGainKnob);
    outputGainKnob.setSliderStyle(juce::Slider::Rotary);
    outputGainKnob.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
    outputGainKnob.setColour(
        juce::Slider::backgroundColourId, juce::Colour(0x00000000));
    outputGainKnob.setColour(
        juce::Slider::thumbColourId, juce::Colours::yellow);
    outputGainKnob.setColour(
        juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        audioProcessor.getAPVTS(), "outputGain", outputGainKnob));
}

BuzzTortionAudioProcessorEditor::~BuzzTortionAudioProcessorEditor()
{
}

//==============================================================================
void BuzzTortionAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll(juce::Colour(0xff323e44));

    area = getLocalBounds();
```

```
//ret ngulo arredondado desenhado
{
    area.removeFromTop(10);
    area.removeFromBottom(10);
    area.removeFromLeft(10);
    area.removeFromRight(10);

    juce::Colour strokeColour = juce::Colours::azure;

    g.setColour(strokeColour);
    g.drawRoundedRectangle(area.getX(), area.getY(),
        area.getWidth(), area.getHeight(), 10.f, .2f);
}

//texto Input
{
    int x = area.getX();
    int y = area.getY();
    int width = area.getWidth() / 4;
    int height = textHeight;

    juce::String text(TRANS("Input"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Drive
{
    int x = area.getX() + area.getWidth() / 4;
```

```
    int y = area.getY();
    int width = area.getWidth() / 4;
    int height = textHeight;

    juce::String text(TRANS("Drive"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Blend
{
    int x = area.getX() + (area.getWidth() / 4) * 2;
    int y = area.getY();
    int width = area.getWidth() / 4;
    int height = textHeight;

    juce::String text(TRANS("Blend"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Output
{
    int x = area.getX() + (area.getWidth() / 4) * 3;
```

```
        int y = area.getY();
        int width = area.getWidth() / 4;
        int height = textHeight;

        juce::String text(TRANS("Output"));
        juce::Colour fillColour = juce::Colours::azure;

        g.setColour(fillColour);
        g.setFont(juce::Font("Bradesco Sans", 15.00f,
            juce::Font::plain).withTypefaceStyle("SemiBold"));
        g.drawText(text, x, y, width, height,
            juce::Justification::centred, true);
    }
}

void BuzzTortionAudioProcessorEditor::resized()
{
    area = getLocalBounds();
    area.removeFromTop(10);
    area.removeFromBottom(10);
    area.removeFromLeft(10);
    area.removeFromRight(10);

    area.removeFromTop(textHeight);
    area.removeFromBottom(10);

    const int bandWidth = area.getWidth() / 4;

    inputGainKnob.setBounds(area.removeFromLeft(bandWidth));
    driveKnob.setBounds(area.removeFromLeft(bandWidth));
    blendKnob.setBounds(area.removeFromLeft(bandWidth));
    outputGainKnob.setBounds(area.removeFromLeft(bandWidth));
}
```

APÊNDICE C – Código do WonderVerb

Abaixo o código completo do arquivo PluginProcessor.h:

```
#pragma once

#include <JuceHeader.h>

//==============================================================================
class WonderVerbAudioProcessor  : public juce::AudioProcessor
{
public:
    //==========================================================
    WonderVerbAudioProcessor();
    ~WonderVerbAudioProcessor() override;

    //==========================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock)
        override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts)
        const override;
   #endif

    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&)
        override;

    //==========================================================
    juce::AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==========================================================
    const juce::String getName() const override;
```

```
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const juce::String getProgramName (int index) override;
    void changeProgramName (int index, const juce::String& newName)
        override;

    //==============================================================
    void getStateInformation (juce::MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes)
        override;

    //==============================================================
    void updateReverb();

    //==============================================================
    juce::AudioProcessorValueTreeState::ParameterLayout
        createParameterLayout();
    juce::AudioProcessorValueTreeState& getAPVTS();

private:
    //==============================================================
    //  rvore  de valores usada para gerenciar todo o estado do
    // AudioProcessor
    juce::AudioProcessorValueTreeState apvts;
    // alocando o processador no ProcessorChain
    juce::dsp::ProcessorChain<juce::dsp::Reverb> processorChain;
    // armazenar  frequ ncia de amostragem
```

```
    float lastSampleRate;

    //==============================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
        WonderVerbAudioProcessor)
};
```

Abaixo o código completo do arquivo PluginProcessor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================
juce::AudioProcessorValueTreeState::ParameterLayout
    WonderVerbAudioProcessor::createParameterLayout()
{
    std::vector <std::unique_ptr <
        juce::RangedAudioParameter>> params;

    auto roomSize = std::make_unique<juce::AudioParameterFloat>
        ("roomSize", // ID
            TRANS("Room Size"), // Nome
            // normalizableRange
            juce::NormalisableRange<float> {0.f, 1.f, 0.0001},
            0.5f); // Valor Padr o
    params.push_back(std::move(roomSize));

    auto damping = std::make_unique<juce::AudioParameterFloat>
        ("damping",
            TRANS("Damping"),
            juce::NormalisableRange<float> {0.f, 1.f, 0.0001},
            0.5f);
    params.push_back(std::move(damping));

    auto wetLevel = std::make_unique<juce::AudioParameterFloat>
        ("wetLevel",
```

```
            TRANS("Wet Level"),
            juce::NormalisableRange<float> {0.f, 1.f, 0.0001},
            0.33f,
            juce::String(),
            juce::AudioProcessorParameter::genericParameter,
            [](float value, int) {
                return juce::String(value * 100, 0) + "%"; },
            [](const juce::String& text) {
                return text.getFloatValue() / 100; });
    params.push_back(std::move(wetLevel));

    auto dryLevel = std::make_unique<juce::AudioParameterFloat>
        ("dryLevel",
            TRANS("Dry Level"),
            juce::NormalisableRange<float> {0.f, 1.f, 0.0001},
            0.4f,
            juce::String(),
            juce::AudioProcessorParameter::genericParameter,
            [](float value, int) {
                return juce::String(value * 100, 0) + "%"; },
            [](const juce::String& text) {
                return text.getFloatValue() / 100; });
    params.push_back(std::move(dryLevel));

    auto width = std::make_unique<juce::AudioParameterFloat>
        ("width",
            TRANS("Width"),
            juce::NormalisableRange<float> {0.f, 1.f, 0.0001},
            1.0f);
    params.push_back(std::move(width));

    return { params.begin(), params.end() };
}
```

```
juce::AudioProcessorValueTreeState&
    WonderVerbAudioProcessor::getAPVTS() { return apvts; }

//==============================================================
WonderVerbAudioProcessor::WonderVerbAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
    : AudioProcessor(BusesProperties()
#if ! JucePlugin_IsMidiEffect
#if ! JucePlugin_IsSynth
        .withInput("Input", juce::AudioChannelSet::stereo(), true)
#endif
        .withOutput("Output", juce::AudioChannelSet::stereo(), true)
#endif
    ),
#endif

    // adiciona os par metros na treeState para serem
    // automatizados na daw
    apvts(*this, nullptr, juce::Identifier("PARAMETER"),
        createParameterLayout())
{
}

WonderVerbAudioProcessor::~WonderVerbAudioProcessor()
{
}

//==============================================================
const juce::String WonderVerbAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool WonderVerbAudioProcessor::acceptsMidi() const
```

```
{
    #if JucePlugin_WantsMidiInput
     return true;
    #else
     return false;
    #endif
}

bool WonderVerbAudioProcessor::producesMidi() const
{
    #if JucePlugin_ProducesMidiOutput
     return true;
    #else
     return false;
    #endif
}

bool WonderVerbAudioProcessor::isMidiEffect() const
{
    #if JucePlugin_IsMidiEffect
     return true;
    #else
     return false;
    #endif
}

double WonderVerbAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int WonderVerbAudioProcessor::getNumPrograms()
{
    return 1;   // NB: some hosts don't cope very well
```

```
                // if you tell them there are 0 programs,
                // so this should be at least 1, even if
                // you're not really implementing programs.
}

int WonderVerbAudioProcessor::getCurrentProgram()
{
    return 0;
}

void WonderVerbAudioProcessor::setCurrentProgram (int index)
{
}

const juce::String WonderVerbAudioProcessor::getProgramName (
    int index)
{
    return {};
}

void WonderVerbAudioProcessor::changeProgramName (
    int index, const juce::String& newName)
{
}

//==============================================================
void WonderVerbAudioProcessor::prepareToPlay (
    double sampleRate, int samplesPerBlock)
{
    lastSampleRate = sampleRate;

    // feito para comunica  o entre plugin e dsp
    juce::dsp::ProcessSpec spec;
    spec.sampleRate = lastSampleRate;
```

```
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = getMainBusNumOutputChannels();

    processorChain.reset(); // evita valores de lixo
    updateReverb();
    // inicia o processor duplicator com as propriedades passadas
    processorChain.prepare(spec);
}

void WonderVerbAudioProcessor::releaseResources()
{
    // When playback stops, you can use this as an opportunity
    // to free up any spare memory, etc.
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool WonderVerbAudioProcessor::isBusesLayoutSupported (
    const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    juce::ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions,
    // will only load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::mono()
        && layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::stereo())
        return false;

    // This checks if the input layout matches the output layout
```

```
    #if ! JucePlugin_IsSynth
     if (layouts.getMainOutputChannelSet() !=
         layouts.getMainInputChannelSet())
         return false;
    #endif

     return true;
   #endif
}
#endif

void WonderVerbAudioProcessor::updateReverb()
{
    // buscando os valores na apvts
    float roomSize = *apvts.getRawParameterValue("roomSize");
    float damping = *apvts.getRawParameterValue("damping");
    float wetLevel = *apvts.getRawParameterValue("wetLevel");
    float dryLevel = *apvts.getRawParameterValue("dryLevel");
    float width = *apvts.getRawParameterValue("width");
    float freezeMode = 0;

    // atualizando o estado do reverberador
    auto& reverb = processorChain.get<0>();
    reverb.setParameters({
        roomSize, damping, wetLevel, dryLevel, width, freezeMode });
}

void WonderVerbAudioProcessor::processBlock (
    juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
```

```
    for (auto i = totalNumInputChannels; i <
        totalNumOutputChannels; ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

    // o bloco de  udio     o buffer
    juce::dsp::AudioBlock<float> block(buffer);
    updateReverb();
    // processa o bloco por meio do dsp
    processorChain.process(
        juce::dsp::ProcessContextReplacing<float>(block));
}

//==============================================================
bool WonderVerbAudioProcessor::hasEditor() const
{
    // (change this to false if you choose to not supply an editor)
    return true;
}

juce::AudioProcessorEditor* WonderVerbAudioProcessor::createEditor()
{
    return new WonderVerbAudioProcessorEditor (*this);
}

//==============================================================
void WonderVerbAudioProcessor::getStateInformation (
    juce::MemoryBlock& destData)
{
    juce::MemoryOutputStream stream(destData, false);
    apvts.state.writeToStream(stream);
}

void WonderVerbAudioProcessor::setStateInformation (
    const void* data, int sizeInBytes)
```

```
{
    juce::ValueTree tree = juce::ValueTree::readFromData(
        data, sizeInBytes);
    if (tree.isValid()) apvts.state = tree;
}

//==============================================================================
// This creates new instances of the plugin..
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new WonderVerbAudioProcessor();
}
```

Abaixo o código completo do arquivo PluginEditor.h:

```
#pragma once

#include <JuceHeader.h>
#include "PluginProcessor.h"

//==============================================================================
class WonderVerbAudioProcessorEditor
    : public juce::AudioProcessorEditor
{
public:
    WonderVerbAudioProcessorEditor (WonderVerbAudioProcessor&);
    ~WonderVerbAudioProcessorEditor() override;

    //==============================================================================
    void paint (juce::Graphics&) override;
    void resized() override;

private:
    juce::Slider roomSizeKnob;
    juce::Slider dampingKnob;
    juce::Slider wetLevelKnob;
```

```
    juce::Slider dryLevelKnob;
    juce::Slider widthKnob;

    juce::Rectangle<int> area;

    int textPosition;
    int textHeight = 30;
    int textWidth = 200;

    juce::OwnedArray<
        juce::AudioProcessorValueTreeState::SliderAttachment>
        sliderAttachments;

    WonderVerbAudioProcessor& audioProcessor;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
        WonderVerbAudioProcessorEditor)
};
```

Abaixo o código completo do arquivo PluginEditor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================
WonderVerbAudioProcessorEditor::WonderVerbAudioProcessorEditor (
    WonderVerbAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    setSize(600, 300);
    setResizable(true, false);

    addAndMakeVisible(roomSizeKnob);
    roomSizeKnob.setSliderStyle(juce::Slider::Rotary);
    roomSizeKnob.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
```

```
roomSizeKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
roomSizeKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
roomSizeKnob.setColour(
    juce::Slider::textBoxOutlineColourId,
    juce::Colour(0x00000000));
sliderAttachments.add(
    new juce::AudioProcessorValueTreeState::SliderAttachment(
    audioProcessor.getAPVTS(), "roomSize", roomSizeKnob));

addAndMakeVisible(dampingKnob);
dampingKnob.setSliderStyle(juce::Slider::Rotary);
dampingKnob.setTextBoxStyle(
    juce::Slider::TextBoxBelow, false, 80, 20);
dampingKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
dampingKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
dampingKnob.setColour(
    juce::Slider::textBoxOutlineColourId,
    juce::Colour(0x00000000));
sliderAttachments.add(
    new juce::AudioProcessorValueTreeState::SliderAttachment(
    audioProcessor.getAPVTS(), "damping", dampingKnob));

addAndMakeVisible(wetLevelKnob);
wetLevelKnob.setSliderStyle(juce::Slider::Rotary);
wetLevelKnob.setTextBoxStyle(
    juce::Slider::TextBoxBelow, false, 80, 20);
wetLevelKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
wetLevelKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
```

```
wetLevelKnob.setColour(
    juce::Slider::textBoxOutlineColourId,
    juce::Colour(0x00000000));
sliderAttachments.add(
    new juce::AudioProcessorValueTreeState::SliderAttachment(
    audioProcessor.getAPVTS(), "wetLevel", wetLevelKnob));

addAndMakeVisible(dryLevelKnob);
dryLevelKnob.setSliderStyle(juce::Slider::Rotary);
dryLevelKnob.setTextBoxStyle(
    juce::Slider::TextBoxBelow, false, 80, 20);
dryLevelKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
dryLevelKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
dryLevelKnob.setColour(
    juce::Slider::textBoxOutlineColourId,
    juce::Colour(0x00000000));
sliderAttachments.add(
    new juce::AudioProcessorValueTreeState::SliderAttachment(
    audioProcessor.getAPVTS(), "dryLevel", dryLevelKnob));

addAndMakeVisible(widthKnob);
widthKnob.setSliderStyle(juce::Slider::Rotary);
widthKnob.setTextBoxStyle(
    juce::Slider::TextBoxBelow, false, 80, 20);
widthKnob.setColour(
    juce::Slider::backgroundColourId, juce::Colour(0x00000000));
widthKnob.setColour(
    juce::Slider::thumbColourId, juce::Colours::yellow);
widthKnob.setColour(
    juce::Slider::textBoxOutlineColourId,
    juce::Colour(0x00000000));
sliderAttachments.add(
```

```
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        audioProcessor.getAPVTS(), "width", widthKnob));
}

WonderVerbAudioProcessorEditor::~WonderVerbAudioProcessorEditor()
{
}

//==============================================================================
void WonderVerbAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll(juce::Colour(0xff323e44));

    area = getLocalBounds();

    //ret ngulo arredondado desenhado
    {
        area.removeFromTop(10);
        area.removeFromBottom(10);
        area.removeFromLeft(10);
        area.removeFromRight(10);

        juce::Colour strokeColour = juce::Colours::azure;

        g.setColour(strokeColour);
        g.drawRoundedRectangle(area.getX(), area.getY(),
            area.getWidth(), area.getHeight(), 10.f, .2f);
    }

    //texto Room Size
    {
        int x = area.getX();
        int y = area.getY();
        int width = area.getWidth() / 5;
```

```
    int height = textHeight;

    juce::String text(TRANS("Room Size"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Damping
{
    int x = area.getX() + area.getWidth() / 5;
    int y = area.getY();
    int width = area.getWidth() / 5;
    int height = textHeight;

    juce::String text(TRANS("Damping"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Wet
{
    int x = area.getX() + (area.getWidth() / 5) * 2;
    int y = area.getY();
    int width = area.getWidth() / 5;
```

```
    int height = textHeight;

    juce::String text(TRANS("Wet"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Dry
{
    int x = area.getX() + (area.getWidth() / 5) * 3;
    int y = area.getY();
    int width = area.getWidth() / 5;
    int height = textHeight;

    juce::String text(TRANS("Dry"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Width
{
    int x = area.getX() + (area.getWidth() / 5) * 4;
    int y = area.getY();
    int width = area.getWidth() / 5;
```

```
        int height = textHeight;

        juce::String text(TRANS("Width"));
        juce::Colour fillColour = juce::Colours::azure;

        g.setColour(fillColour);
        g.setFont(juce::Font("Bradesco Sans", 15.00f,
            juce::Font::plain).withTypefaceStyle("SemiBold"));
        g.drawText(text, x, y, width, height,
            juce::Justification::centred, true);
    }
}

void WonderVerbAudioProcessorEditor::resized()
{
    area = getLocalBounds();
    area.removeFromTop(10);
    area.removeFromBottom(10);
    area.removeFromLeft(10);
    area.removeFromRight(10);

    area.removeFromTop(textHeight);
    area.removeFromBottom(10);

    const int bandWidth = area.getWidth() / 5;

    roomSizeKnob.setBounds(area.removeFromLeft(bandWidth));
    dampingKnob.setBounds(area.removeFromLeft(bandWidth));
    wetLevelKnob.setBounds(area.removeFromLeft(bandWidth));
    dryLevelKnob.setBounds(area.removeFromLeft(bandWidth));
    widthKnob.setBounds(area.removeFromLeft(bandWidth));
}
```

APÊNDICE D – Código do BigLoop

Abaixo o código completo do arquivo PluginProcessor.h:

```
#pragma once

#include <JuceHeader.h>

//==============================================================================
class BigLoopAudioProcessor  : public juce::AudioProcessor
{
public:
    //==========================================================
    BigLoopAudioProcessor();
    ~BigLoopAudioProcessor() override;

    //==========================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock)
        override;
    void releaseResources() override;

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts)
        const override;
   #endif

    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&)
        override;

    //==========================================================
    juce::AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==========================================================
    const juce::String getName() const override;
```

```
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

    //==============================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const juce::String getProgramName (int index) override;
    void changeProgramName (int index, const juce::String& newName)
        override;

    //==============================================================
    void getStateInformation (juce::MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes)
        override;

    //==============================================================
    void fillDelayBuffer(juce::AudioBuffer<float>& buffer,
        juce::AudioBuffer<float>& delayBuffer, int channel);
    void getFromDelayBuffer(juce::AudioBuffer<float>& buffer,
        juce::AudioBuffer<float>& delayBuffer, int channel);
    void feedbackDelay(juce::AudioBuffer<float>& buffer,
        juce::AudioBuffer<float>& delayBuffer, int channel);

    //==============================================================
    juce::AudioProcessorValueTreeState::ParameterLayout
        createParameterLayout();
    juce::AudioProcessorValueTreeState& getAPVTS();

private:
    //==============================================================
    //  rvore  de valores usada para gerenciar todo o estado do
```

```
    // AudioProcessor
    juce::AudioProcessorValueTreeState apvts;
    // delayBuffer para armazenar as amostras
    juce::AudioBuffer<float> delayBuffer;

    // armazenar  frequ ncia de amostragem
    int lastSampleRate;
    // p o s i   o onde s e r  escrito nos delayBuffer
    int writePosition;
    // armazenar  o tempo de delay
    int lastDelayTime;
    // armazenar  o ganho de entrada
    float lastInputGain;
    // armazenar  ganho de feedback
    int lastFeedbackGain;

    //==============================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
        BigLoopAudioProcessor)
};
```

Abaixo o código completo do arquivo PluginProcessor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================
juce::AudioProcessorValueTreeState::ParameterLayout
    BigLoopAudioProcessor::createParameterLayout()
{
    std::vector <std::unique_ptr <
        juce::RangedAudioParameter>> params;

    const float minGain = juce::Decibels::decibelsToGain(-3.f);
    const float maxGain = juce::Decibels::decibelsToGain(3.f);
```

```
auto delayTime = std::make_unique<juce::AudioParameterInt>
    ("delayTime", // ID
        TRANS("Delay Time"), // Nome
        0, // min
        1000, //max
        0, // Valor Padr o
        juce::String(),
        [](float value, int) {
            return juce::String(value, 0) + "s"; },
        [](const juce::String& text) {
            return text.getFloatValue(); });
params.push_back(std::move(delayTime));

auto inputGain = std::make_unique<juce::AudioParameterFloat>
    ("inputGain",
        TRANS("Input Gain"),
        juce::NormalisableRange<float> {  // range
            minGain, maxGain, 0.001f, std::log(.5f) /
            std::log((1.f - minGain) / (maxGain - minGain))},
        1.f,
        juce::String(), // Legenda (opcional)
        // Categoria (opcional)
        juce::AudioProcessorParameter::genericParameter,
        // converte um valor n o normalizado para uma string
        // com tamanho limitado, usado pelos hosts para
        // mostrar o valor
        [](float value, int) {return juce::String(
            juce::Decibels::gainToDecibels(value), 1) + " dB"; },
        // o contr rio do de cima, para o usu rio escrever
        // um valor para o par metro
        [](juce::String text) {
            return juce::Decibels::decibelsToGain(
            text.dropLastCharacters(3).getFloatValue()); }
);
```

```
    params.push_back(std::move(inputGain));

    juce::NormalisableRange<float> feedbackRange{ 0.f, 1.f, 0.001f };
    feedbackRange.setSkewForCentre(.5f);
    auto feedbackGain = std::make_unique<juce::AudioParameterFloat>
        ("feedbackGain",
            TRANS("Feedback Gain"),
            feedbackRange,
            .5f,
            juce::String(), // Legenda (opcional)
            juce::AudioProcessorParameter::genericParameter,
            [](float value, int) {return juce::String(
                juce::Decibels::gainToDecibels(value), 1) + " dB"; },
            [](juce::String text) {
                return juce::Decibels::decibelsToGain(
                text.dropLastCharacters(3).getFloatValue()); }
    );
    params.push_back(std::move(feedbackGain));

    return { params.begin(), params.end() };
}

juce::AudioProcessorValueTreeState& BigLoopAudioProcessor::getAPVTS()
    { return apvts; }

//==============================================================================
BigLoopAudioProcessor::BigLoopAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",
                         juce::AudioChannelSet::stereo(), true)
                      #endif
```

```
                        .withOutput ("Output",
                         juce::AudioChannelSet::stereo(), true)
                      #endif
                        ),
#endif

    // adiciona os parmetros na treeState para
    // serem automatizados na daw
    apvts(*this, nullptr, juce::Identifier("PARAMETER"),
        createParameterLayout())
{
}

BigLoopAudioProcessor::~BigLoopAudioProcessor()
{
}

//==============================================================================
const juce::String BigLoopAudioProcessor::getName() const
{
    return JucePlugin_Name;
}

bool BigLoopAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool BigLoopAudioProcessor::producesMidi() const
{
```

```
    #if JucePlugin_ProducesMidiOutput
     return true;
    #else
     return false;
    #endif
}

bool BigLoopAudioProcessor::isMidiEffect() const
{
    #if JucePlugin_IsMidiEffect
     return true;
    #else
     return false;
    #endif
}

double BigLoopAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}

int BigLoopAudioProcessor::getNumPrograms()
{
    return 1;   // NB: some hosts don't cope very well if
                // you tell them there are 0 programs,
                // so this should be at least 1, even if
                // you're not really implementing programs.
}

int BigLoopAudioProcessor::getCurrentProgram()
{
    return 0;
}
```

```
void BigLoopAudioProcessor::setCurrentProgram (int index)
{
}

const juce::String BigLoopAudioProcessor::getProgramName (int index)
{
    return {};
}

void BigLoopAudioProcessor::changeProgramName (
    int index, const juce::String& newName)
{
}

//==============================================================
void BigLoopAudioProcessor::prepareToPlay (
    double sampleRate, int samplesPerBlock)
{
    lastSampleRate = sampleRate;
    writePosition = 0;

    // inicializa com os valores dos par metros do delay
    lastDelayTime = *apvts.getRawParameterValue("delayTime");
    lastInputGain = *apvts.getRawParameterValue("inputGain");
    lastFeedbackGain = *apvts.getRawParameterValue("feedbackGain");

    // tamanho do delayBuffer, 2 segundos + 2 buffers de seguran a
    const int delayBufferSize = 2 * (sampleRate + samplesPerBlock);
    // define o tamanho do delayBuffer
    delayBuffer.setSize(getTotalNumInputChannels(),
        (int)delayBufferSize);
    //evita valores de lixo
    delayBuffer.clear();
}
```

```
void BigLoopAudioProcessor::releaseResources()
{
    // When playback stops, you can use this as an opportunity
    // to free up any spare memory, etc.
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool BigLoopAudioProcessor::isBusesLayoutSupported (
    const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    juce::ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions,
    //will only load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::mono()
        && layouts.getMainOutputChannelSet() !=
        juce::AudioChannelSet::stereo())
        return false;

    // This checks if the input layout matches the output layout
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() !=
        layouts.getMainInputChannelSet())
        return false;
   #endif

    return true;
  #endif
```

```
}
#endif

void BigLoopAudioProcessor::processBlock (
    juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i <
        totalNumOutputChannels; ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

    for (int channel = 0; channel < totalNumInputChannels; ++channel)
    {
        // preenche o buffer de delay
        fillDelayBuffer(buffer, delayBuffer, channel);
        // l   do buffer de delay
        getFromDelayBuffer(buffer, delayBuffer, channel);
        // adiciona o delay no delayBuffer
        feedbackDelay(buffer, delayBuffer, channel);
    }

    // aumenta a posi  o de escrita no tamanho do buffer principal
    writePosition += buffer.getNumSamples();
    // garantindo que a posi  o estar  dentro do tamanho do buffer
    writePosition %= delayBuffer.getNumChannels();
}

//==============================================================
void BigLoopAudioProcessor::fillDelayBuffer(
    juce::AudioBuffer<float>& buffer,
    juce::AudioBuffer<float>& delayBuffer, int channel)
```

```
{
    // ponteiro para o canal do buffer
    auto* channelData = buffer.getReadPointer(channel);
    // tamanho do buffer
    auto bufferSize = buffer.getNumSamples();
    // tamanho do delayBuffer
    auto delayBufferSize = delayBuffer.getNumSamples();

    // l   o valor do ganho de entrada
    float inputGain = *apvts.getRawParameterValue("inputGain");


    // se as p o s i   e s  que  s e r  o preenchidas  n  o ultrapassam a
    // ltima   p o s i   o do delayBuffer
    if (bufferSize + writePosition < delayBufferSize)
    {
        // preenche o delayBuffer com os valores do buffer principal,
        // aplicando uma rampa de ganho
        delayBuffer.copyFromWithRamp(channel, writePosition,
            channelData, bufferSize, lastInputGain, inputGain);
    }
    else // se ultrapassam
    {
        // quantos valores ainda podem ser escritos no delayBuffer
        auto numSamplesToEnd = delayBufferSize - writePosition;
        // preenche o delayBuffer at   a  ltima   p o s i   o ,
        // aplicando uma rampa de ganho
        delayBuffer.copyFromWithRamp(channel, writePosition,
            channelData, numSamplesToEnd, lastInputGain, inputGain);
        // quantos valores sobraram no buffer
        auto numSamplesAtStart = bufferSize - numSamplesToEnd;
        // preenche o delayBuffer com o que sobrou no buffer
        // principal, aplicando uma rampa de ganho
        delayBuffer.copyFromWithRamp(channel, 0, channelData,
```

```
            numSamplesAtStart, lastInputGain, inputGain);
    }
    // atualiza o valor do lastInputGain
    lastInputGain = inputGain;
}

void BigLoopAudioProcessor::getFromDelayBuffer(
    juce::AudioBuffer<float>& buffer,
    juce::AudioBuffer<float>& delayBuffer, int channel)
{
    // tamanho do buffer
    auto bufferSize = buffer.getNumSamples();
    // tamanho do delayBuffer
    auto delayBufferSize = delayBuffer.getNumSamples();

    // l   o valor do tempo de delay
    int delayTime = *apvts.getRawParameterValue("delayTime"); // ms

    // evita lixo ao alterar o tempo de delay
    if (delayTime != lastDelayTime)
    {
        delayBuffer.clear();
    }

    // a posi  o que ir  ser lida do delayBuffer
    auto readPosition = static_cast<int>(writePosition -
        (getSampleRate() * delayTime / 1000));

    // assegurar que n o ser o lidas posi  es negativas
    if (readPosition < 0) readPosition += delayBufferSize;

    // se as posi  es que ser o lidas n o ultrapassam a
    //  ltima  posi  o do delayBuffer
    if (readPosition + bufferSize < delayBufferSize)
```

```
    {
        // preenche o buffer inteiro
        buffer.copyFrom(channel, 0, delayBuffer.getReadPointer(
            channel, readPosition), bufferSize);
    }
    else // se ultrapassam
    {
        // quantos valores ainda podem ser lidos do delayBuffer
        auto numSamplesToEnd = delayBufferSize - readPosition;
        // adiciona valores at  finalizar o delayBuffer
        buffer.copyFrom(channel, 0, delayBuffer.getReadPointer(
            channel, readPosition), numSamplesToEnd);
        // quantos valores faltam para completar o buffer
        auto numSamplesAtStart = bufferSize - numSamplesToEnd;
        // termina de preencher o buffer
        buffer.copyFrom(channel, numSamplesToEnd,
            delayBuffer.getReadPointer(channel), numSamplesAtStart);
    }
    // atualiza o valor do lastDelayTime
    lastDelayTime = delayTime;
}

void BigLoopAudioProcessor::feedbackDelay(
    juce::AudioBuffer<float>& buffer,
    juce::AudioBuffer<float>& delayBuffer, int channel)
{
    // p o s i  o que s e r  lida do buffer
    auto* channelData = buffer.getReadPointer(channel);
    // tamanho do buffer
    auto bufferSize = buffer.getNumSamples();
    // tamanho do delayBuffer
    auto delayBufferSize = delayBuffer.getNumSamples();

    // l  o valor do ganho de feedback
```

```
    float feedbackGain = *apvts.getRawParameterValue("feedbackGain");

    // se as posi  es que ser o preenchidas n o ultrapassam a
    //  ltima  posi  o do delayBuffer
    if (bufferSize + writePosition < delayBufferSize)
    {
        // adiciona no delayBuffer os valores do buffer principal,
            // aplicando uma rampa de ganho
        delayBuffer.addFromWithRamp(channel, writePosition,
            channelData, bufferSize, lastFeedbackGain, feedbackGain);
    }
    else // se ultrapassa
    {
        // quantos valores ainda podem ser escritos no delayBuffer
        auto numSamplesToEnd = delayBufferSize - writePosition;
        // adiciona no delayBuffer at  a  ltima  posi  o ,
            // aplicando uma rampa de ganho
        delayBuffer.addFromWithRamp(channel, writePosition,
            channelData, numSamplesToEnd,
            lastFeedbackGain, feedbackGain);
        // quantos valores sobraram no buffer
        auto numSamplesAtStart = bufferSize - numSamplesToEnd;
        // adiciona no delayBuffer com o que sobrou no
        // buffer pricipal, aplicando uma rampa de ganho
        delayBuffer.addFromWithRamp(channel, 0, channelData,
            numSamplesAtStart, lastFeedbackGain, feedbackGain);
    }
    // atualiza o valor do lastFeedbackGain
    lastFeedbackGain = feedbackGain;
}


//==============================================================
bool BigLoopAudioProcessor::hasEditor() const
```

```
{
    // (change this to false if you choose to not supply an editor)
    return true;
}

juce::AudioProcessorEditor* BigLoopAudioProcessor::createEditor()
{
    return new BigLoopAudioProcessorEditor (*this);
}

//==============================================================
void BigLoopAudioProcessor::getStateInformation (
    juce::MemoryBlock& destData)
{
    juce::MemoryOutputStream stream(destData, false);
    apvts.state.writeToStream(stream);
}

void BigLoopAudioProcessor::setStateInformation (
    const void* data, int sizeInBytes)
{
    juce::ValueTree tree = juce::ValueTree::readFromData(
        data, sizeInBytes);
    if (tree.isValid()) apvts.state = tree;
}

//==============================================================
// This creates new instances of the plugin..
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new BigLoopAudioProcessor();
}
```

Abaixo o código completo do arquivo PluginEditor.h:

```
#pragma once
```

```
#include <JuceHeader.h>
#include "PluginProcessor.h"

//==============================================================================
class BigLoopAudioProcessorEditor
    : public juce::AudioProcessorEditor
{
public:
    BigLoopAudioProcessorEditor (BigLoopAudioProcessor&);
    ~BigLoopAudioProcessorEditor() override;

    //==============================================================================
    void paint (juce::Graphics&) override;
    void resized() override;

private:

    juce::Slider delayTimeKnob;
    juce::Slider inputGainKnob;
    juce::Slider feedbackGainKnob;

    juce::OwnedArray<
        juce::AudioProcessorValueTreeState::SliderAttachment>
        sliderAttachments;

    juce::Rectangle<int> area;

    int textPosition;
    int textHeight = 30;
    int textWidth = 200;

    BigLoopAudioProcessor& audioProcessor;
```

```
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (
        BigLoopAudioProcessorEditor)
};
```

Abaixo o código completo do arquivo PluginEditor.cpp:

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================
BigLoopAudioProcessorEditor::BigLoopAudioProcessorEditor (
    BigLoopAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    setSize(500, 300);
    setResizable(true, false);

    addAndMakeVisible(delayTimeKnob);
    delayTimeKnob.setSliderStyle(juce::Slider::Rotary);
    delayTimeKnob.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
    delayTimeKnob.setColour(
        juce::Slider::backgroundColourId, juce::Colour(0x00000000));
    delayTimeKnob.setColour(
        juce::Slider::thumbColourId, juce::Colours::yellow);
    delayTimeKnob.setColour(
        juce::Slider::textBoxOutlineColourId,
            juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        audioProcessor.getAPVTS(), "delayTime", delayTimeKnob));

    addAndMakeVisible(inputGainKnob);
    inputGainKnob.setSliderStyle(juce::Slider::Rotary);
    inputGainKnob.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
```

```
    inputGainKnob.setColour(
        juce::Slider::backgroundColourId, juce::Colour(0x00000000));
    inputGainKnob.setColour(
        juce::Slider::thumbColourId, juce::Colours::yellow);
    inputGainKnob.setColour(
        juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        audioProcessor.getAPVTS(), "inputGain", inputGainKnob));

    addAndMakeVisible(feedbackGainKnob);
    feedbackGainKnob.setSliderStyle(juce::Slider::Rotary);
    feedbackGainKnob.setTextBoxStyle(
        juce::Slider::TextBoxBelow, false, 80, 20);
    feedbackGainKnob.setColour(
        juce::Slider::backgroundColourId, juce::Colour(0x00000000));
    feedbackGainKnob.setColour(
        juce::Slider::thumbColourId, juce::Colours::yellow);
    feedbackGainKnob.setColour(
        juce::Slider::textBoxOutlineColourId,
        juce::Colour(0x00000000));
    sliderAttachments.add(
        new juce::AudioProcessorValueTreeState::SliderAttachment(
        audioProcessor.getAPVTS(), "feedbackGain", feedbackGainKnob));
}

BigLoopAudioProcessorEditor::~BigLoopAudioProcessorEditor()
{
}

//==============================================================
void BigLoopAudioProcessorEditor::paint (juce::Graphics& g)
{
```

```
g.fillAll(juce::Colour(0xff323e44));

area = getLocalBounds();

//ret ngulo arredondado desenhado
{
    area.removeFromTop(10);
    area.removeFromBottom(10);
    area.removeFromLeft(10);
    area.removeFromRight(10);

    juce::Colour strokeColour = juce::Colours::azure;

    g.setColour(strokeColour);
    g.drawRoundedRectangle(area.getX(), area.getY(),
        area.getWidth(), area.getHeight(), 10.f, .2f);
}

//texto Delay Time
{
    int x = area.getX();
    int y = area.getY();
    int width = area.getWidth() / 3;
    int height = textHeight;

    juce::String text(TRANS("Delay Time"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}
```

```
//texto Input
{
    int x = area.getX() + area.getWidth() / 3;
    int y = area.getY();
    int width = area.getWidth() / 3;
    int height = textHeight;

    juce::String text(TRANS("Input"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}

//texto Feedback
{
    int x = area.getX() + (area.getWidth() / 3) * 2;
    int y = area.getY();
    int width = area.getWidth() / 3;
    int height = textHeight;

    juce::String text(TRANS("Feedback"));
    juce::Colour fillColour = juce::Colours::azure;

    g.setColour(fillColour);
    g.setFont(juce::Font("Bradesco Sans", 15.00f,
        juce::Font::plain).withTypefaceStyle("SemiBold"));
    g.drawText(text, x, y, width, height,
        juce::Justification::centred, true);
}
```

```
}

void BigLoopAudioProcessorEditor::resized()
{
    area = getLocalBounds();
    area.removeFromTop(10);
    area.removeFromBottom(10);
    area.removeFromLeft(10);
    area.removeFromRight(10);

    area.removeFromTop(textHeight);
    area.removeFromBottom(10);

    const int bandWidth = area.getWidth() / 3;

    delayTimeKnob.setBounds(area.removeFromLeft(bandWidth));
    inputGainKnob.setBounds(area.removeFromLeft(bandWidth));
    feedbackGainKnob.setBounds(area.removeFromLeft(bandWidth));
}
```