Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Vitor Samora da Graça

Efeitos Sonoros em Tempo Real

São Paulo
Novembro de 2016

Efeitos Sonoros em Tempo Real

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Marcelo Gomes de Queiroz

São Paulo
Novembro de 2016

Agradecimentos

Agradeço, primeiramente, aos meus familiares por todo o apoio, paciência, carinho e amor que me foi concedido durante todas as fases de nossas vidas. Não consigo encontrar palavras que expressem o quanto vocês são importantes para mim e o quão grato sou por tê-los em minha vida.

Agradeço aos meus amigos de escola pelo apoio, diversão e companheirismo durante todos esses anos. Graças a vocês eu pude suportar alguns dos piores momentos da minha vida e ter alguns dos melhores.

Agradeço aos meus colegas de turma pela ajuda e pelos momentos que partilhamos ao longo desses anos de curso.

Agradeço a todos os professores e funcionários do IME-USP que contribuíram direta ou indiretamente na minha formação.

Por fim, agradeço a todos os integrantes do grupo de computação musical, principalmente ao meu supervisor, Prof. Marcelo Queiroz, por toda sua a paciência e contribuição no processo de aprendizado que possibilitou o desenvolvimento deste trabalho.

Resumo

Efeitos sonoros em tempo real têm aplicação no desenvolvimento musical e podem ser implementados por meio de algoritmos de processamento de sinais segmentados em blocos de tamanho fixo. Nesse trabalho apresentamos a fundamentação teórica de técnicas de processamento de áudio associadas a efeitos populares como *overdrive*, *pitch shifting* e *wah-wah*, sendo também feita a implementação de *plugins* para cada um deles. Tais efeitos fazem uso de diversas formas de representação de sinais digitais, que podem ser processados diretamente na forma de onda (domínio do tempo) ou através de seus espectros (domínio da frequência), sendo que a conversão entre essas duas representações é feita através da transformada de Fourier. Para as técnicas abordadas temos duas categorias: as técnicas não-lineares e os filtros lineares. Da primeira categoria, estudamos as técnicas de *waveshaping*, *time stretching* e *pitch shifting*. Para filtros lineares, apresentamos noções sobre polos e zeros, e seus reflexos em relação às componentes de frequência que são enfatizadas ou retidas nas saídas dos filtros considerados. A bibliografia selecionada conta com livros introdutórios de computação musical como *Elements of Computer Music* e *The Theory and Technique of Electronic Music*, e com o livro *DAFX: Digital Audio Effects*, que é mais específico sobre efeitos digitais de áudio e processamento de sinais digitais. A verificação das implementações pode ser feita por meio da comparação entre métricas de saídas geradas para entradas simples. Comparamos as métricas que são fornecidas pela bibliografia, no caso dos modelos estudados, com as que são calculadas através das saídas geradas pelos *plugins*.

Palavras-chave: efeitos digitais de áudio, filtros lineares e não-lineares, processamento sonoro em tempo real.

Abstract

Realtime audio effects have application in musical development and can be implemented through signal processing algorithms where the signal is segmented in blocks of fixed size. In this final course assignment we present the theoretical foundation of sound processing techniques related to popular audio effects such as overdrive, pitch shifting and wah-wah, implementing plugins for each one of them. These effects use several forms of representation for a digital signal, that can be processed directly in its waveform (time domain) or in its spectrum (frequency domain); and we can transform one representation into another through the Fourier transform. We have two categories for the covered techniques: linear and nonlinear filters. For the nonlinear category, we study waveshaping, time stretching and pitch shifting techniques. For linear filters, we present poles and zeros notions and their reflexes in the frequency components that can be emphasized or retained in the output of the studied filters. The selected bibliography contains introductory books of computer music, such as *Elements of computer music* and *The theory and technique of electronic music*, and the book *DAFX: Digital Audio Effects*, which is more specific about audio effects and sound processing. The validation of the implementations can be made by the comparison of metrics taken from the outputs generated from simple inputs. We compare the metrics provided by the bibliography, from the studied models, with the metrics calculated from the outputs of the plugins.

Keywords: digital audio effects, linear and nonlinear filters, realtime sound processing.

Sumário

Capítulo 1

Introdução

1.1 Contextualização

Efeitos sonoros em tempo real têm grande aplicação no desenvolvimento musical e são popularmente associados a instrumentos de corda e teclados. Isso se dá tanto pela difusão de estilos musicais que são marcados pelos timbres desses instrumentos, como pelo aparecimento de artistas icônicos que os utilizaram. Em meio a esse contexto, a criação de novos efeitos sonoros se tornou relevante e diversos pedais de efeitos analógicos (unidades que alteram propriedades de um som de entrada através de circuitos elétricos) passaram a ser desenvolvidos.

1.2 Justificativa

Com a evolução da computação musical, tornou-se possível a concepção de diversas famílias de efeitos sonoros por meio de modelos matemáticos, compondo uma série de técnicas para manipulação de sinais digitais. Suas implementações, especialmente aquelas em código aberto, em ambientes desktop e dispositivos embarcados permitem uma disseminação e utilização por parte de potenciais interessados de muito maior alcance do que as versões implementadas em hardware, custosas e menos flexíveis. Algumas dessas técnicas estão dispostas no livro *DAFX: Digital Audio Effects* [Zölzer(2011)], enquanto que os fundamentos de computação musical necessários para a aplicação e entendimento dessas técnicas podem ser encontrados em livros como *Elements of Computer Music* [Moore(1990)] e *The Theory and Technique of Electronic Music* [Puckette(2007)].

1.3 Motivação

Sendo um guitarrista nas horas vagas há 9 anos, existe a curiosidade de aprender como são implementados os efeitos digitais mais populares entre os guitarristas, com a possibilidade

de futuramente contribuir com a comunidade através da criação de novos efeitos ou sugestão de novas formas de implementação para efeitos já conhecidos.

1.4 Objetivo

O principal objetivo deste trabalho é estudar e aprender os fundamentos de computação musical necessários para o desenvolvimento de famílias de efeitos. As famílias abordadas foram as de filtros lineares e a de efeitos não-lineares, sendo feita a implementação de *plugins* que utilizam as técnicas estudadas para a concepção dos efeitos, além da validação desses *plugins* por meio de experimentos.

1.5 Implementações e experimentos

A implementação de efeitos sonoros digitais se dá pela criação de *plugins* que, por seguirem um determinado padrão, podem ser empregados em vários *hosts* através de uma interface em comum. Nas implementações dos *plugins* foi utilizado o padrão LV2 [LV2()], no qual os efeitos podem ser escritos em linguagem C e apresentam interface de dados definida por sintaxe Turtle [Tur()].

Para testes durante o período de desenvolvimento foi utilizado como *host* o *software* Ardour [Ard()], sendo que muitas das métricas perceptuais puderam ser obtidas em tempo real através de *plugins* de análise, tais como os da ferramenta CALF [CAL()]. Os experimentos foram realizados com entradas simplificadas (como senoides, por exemplo) que possuem saídas conhecidas para praticamente todos os efeitos implementados. A parte experimental é exibida na forma de gráficos que foram gerados com o auxílio do programa Sonic Visualiser [Son()].

1.6 Estrutura

Esse texto foi separado em dois capítulos principais, sendo exposta no capítulo 2 toda a fundamentação teórica utilizada no desenvolvimento do trabalho. Iniciamos discutindo e definindo na seção 2.1 alguns dos conceitos que serão utilizados ao longo de todo o trabalho. Na seção 2.2 é feita a apresentação da família de filtros lineares através de definições que permitem a descrição de filtros por meio de fórmulas matemáticas; também apresentamos análises de métricas que traduzem o comportamento da resposta de filtros, além de exemplos de filtros clássicos. Finalmente, na seção 2.3 apresentamos com detalhes algumas das técnicas usadas na implementação de efeitos não-lineares. No capítulo 3 é feita a apresentação de implementações das técnicas expostas no capítulo anterior, discutindo suas limitações em ambientes de tempo real, assim como experimentos que verificam se os *plugins* desenvolvidos estão de acordo com os modelos propostos.

Capítulo 2

Fundamentação teórica

2.1 Conceitos utilizados

Antes de discutirmos a concepção de efeitos de áudio em tempo real, necessitamos de um *background* em conceitos básicos de computação musical. A seguir apresentaremos alguns desses conceitos que serão utilizados diversas vezes no desenvolvimento tanto teórico como prático do trabalho.

2.1.1 Representação digital e teorema de Nyquist

A representação digital de um sinal analógico pode ser dada através de sua forma de onda ou de seu espectro. A forma de onda é dada pela variação da amplitude do sinal ao longo do tempo, enquanto na representação pelo espectro temos o sinal descrito no domínio de componentes de frequência.

Devemos ressaltar que em nenhuma dessas formas de representação o sinal analógico pode ser representado explicitamente, pois em qualquer intervalo de tempo maior que zero temos infinitos valores de amplitude. Portanto, a representação de um sinal analógico no computador pressupõe um processo de amostragem (ou *sampling*), possuindo uma taxa associada que deve condizer com o teorema de Nyquist.

Teorema de Nyquist: Para representar digitalmente um sinal contendo componentes de frequência de até X Hz é necessário usar uma taxa de amostragem de pelo menos 2X amostras (ou *samples*) por segundo [Moore(1990)].

O processo de amostragem possui três estágios em sua realização:

- Filtragem do sinal, para que quaisquer componentes de frequência maiores que metade da taxa de amostragem sejam removidas;
- Obtenção de valores de amplitude instantânea, em intervalos de tempo igualmente espaçados;

- Conversão de cada medida feita em um código numérico discreto (por exemplo um valor em ponto flutuante entre -1 e 1).

O processo descrito é realizado por um sistema elétrico chamado ADC (*Analog to Digital Converter*), tendo como saída uma sequência de valores chamada de sinal digital. A partir do sinal digital é possível construir o sinal analógico associado através de um sistema DAC (*Digital to Analog Converter*), que basicamente desfaz os três estágios anteriormente descritos em ordem reversa (converte códigos em valores de amplitude, preenche os intervalos com valores constantes e suaviza o resultado).

2.1.2 Transformada discreta de Fourier

O processo de amostragem descrito anteriormente produz uma representação digital do sinal analógico em forma de onda, mas para muitas aplicações a representação espectral é mais conveniente ou até mesmo necessária. A transformada discreta de Fourier (DFT) pode ser vista como uma ferramenta matemática que permite essa conversão, traduzindo a representação digital de forma de onda para o domínio da frequência (ou vice-versa) sem perda de informação.

Dado que estamos lidando com um sinal digital $x(n)$ de N amostras, a DFT e sua inversa (IDFT) são definidas respectivamente por:

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-2\pi jkn/N} \qquad k = 0, 1, ..., N-1$$

$$x(n) = \sum_{k=0}^{N-1} X(k)e^{2\pi jkn/N} \qquad n = 0, 1, ..., N-1,$$

onde $X(f)$ é o espectro de $x(t)$ e seu domínio se refere a valores de frequência, e é a base dos logaritmos naturais, j é a unidade imaginária e k é o índice da amostra no domínio da frequência.

Assim como $x(n)$ representa N amostras de uma forma de onda contínua no tempo, $X(k)$ representa N amostras de um espectro contínuo em relação à frequência. Isto é, a DFT é amostrada no domínio da frequência da mesma forma que a forma de onda digital é amostrada no domínio do tempo, sendo possível transitar de uma representação para a outra sem perder nem ganhar informações e em tempo de computação proporcional a N^2 quando aplicamos diretamente as equações apresentadas.

Para entendermos melhor as equações apresentadas podemos considerar a relação de Euler [Moore(1990)]:

$$e^{j\theta} = \cos\theta + j\sin\theta$$

A relação de Euler nos diz que a transformada discreta de Fourier retorna valores que estão no domínio dos números complexos. Com isso, cada valor complexo $X(k)$ pode ser

mapeado em um vetor partindo da origem do plano complexo, do qual podemos tirar informação sobre a amplitude e a fase da componente senoidal de frequência associada ao índice k (sendo que a DFT representa o sinal através de N dessas componentes).

A amplitude da componente informa quão presente é a frequência correspondente no sinal, e é indicada pelo comprimento do vetor $X(k)$ no plano complexo. Já a fase indica quão distante o sinal está do início do período para aquela frequência, sendo indicada pelo ângulo que o vetor se encontra em relação ao eixo real do plano, em sentido anti-horário. Dado $X(k) = a + ib$, esses valores podem ser calculados por:

$$|X(k)| = \sqrt{a^2 + b^2}$$
$$\theta(X(k)) = \tan^{-1} \frac{b}{a}$$

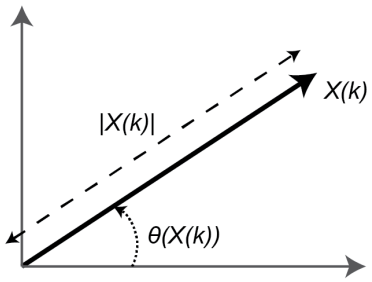

Figura 2.1: Representação gráfica de amplitude e fase para uma componente $X(k)$.

Na prática, a DFT é normalmente calculada utilizando uma técnica chamada *transformada rápida de Fourier* (FFT), que é uma forma eficiente de calcular a DFT, mas apenas para certos valores de N. Quando restringimos N para valores de potências de 2, a FFT é capaz de obter o mesmo resultado que a DFT em tempo de computação proporcional a $N \log_2 N$ [Moore(1990)].

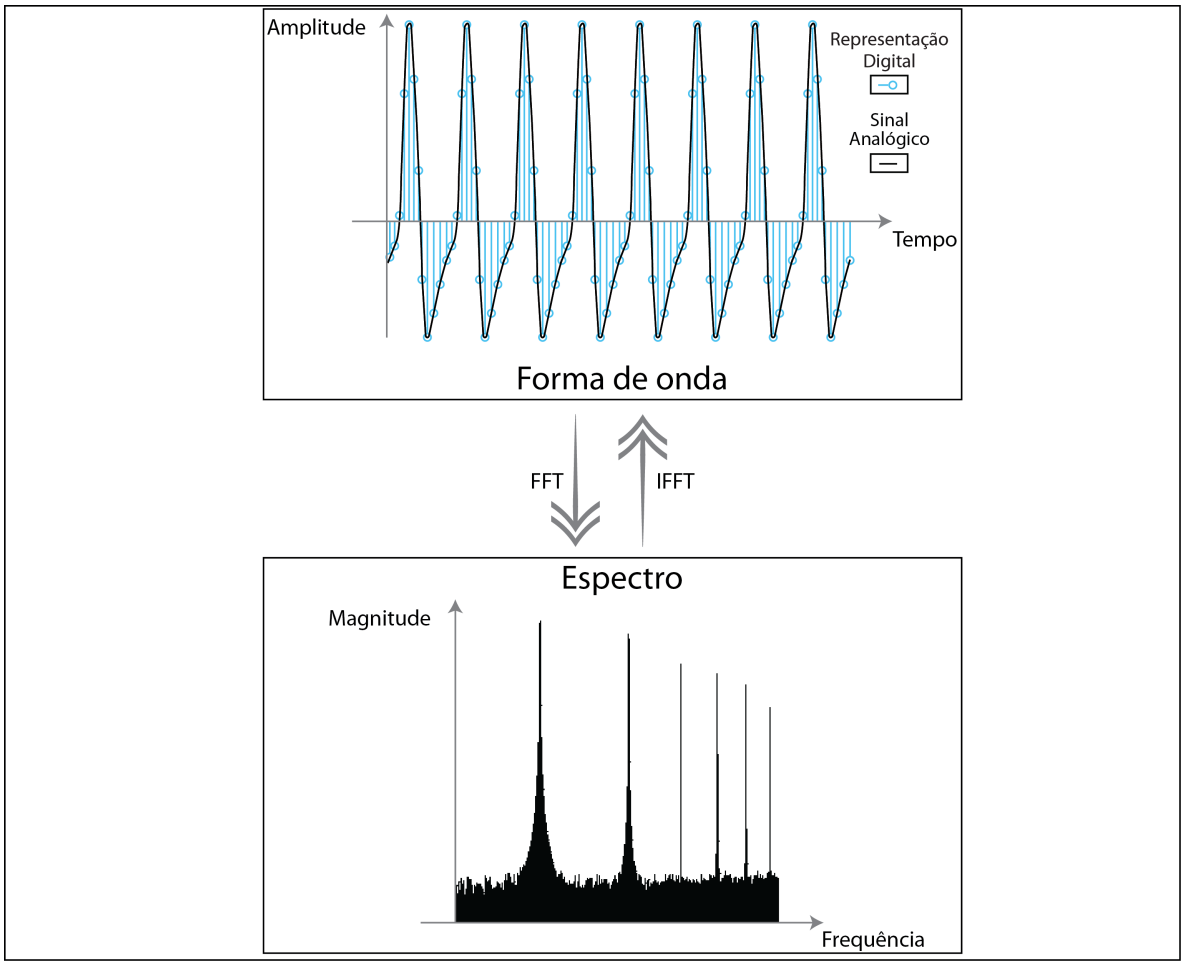

Figura 2.2: Representação digital e FFT.

2.1.3 Processamento de sinal em bloco

O processamento de determinados efeitos pode ser feito de forma particionada. Isto é, um sinal de entrada pode ser quebrado em blocos de tamanho N, tendo suas N amostras processadas e gerando uma saída, normalmente também de tamanho N.

Essa forma de processamento é conveniente para casos em que o arquivo de áudio a ser processado é grande, não havendo a necessidade de armazenar todas as amostras em memória. Além disso, para a implementação de efeitos de áudio em tempo real o processamento em bloco é uma necessidade, uma vez que a resposta para o sinal gerado deve ser dada com atraso pouco perceptível.

2.2 Filtros lineares

Filtros são uma forma de selecionar determinados elementos de um conjunto. Como já vimos, um sinal pode ser visto como um conjunto de parciais (componentes senoidais referentes às possíveis frequências dentro do espectro de um som). Basicamente, um filtro realiza a seleção das parciais que queremos rejeitar, reter, atenuar ou enfatizar [Zölzer(2011)].

Mais formalmente, um filtro pode ser caracterizado pela sua função de transferência, normalmente denotada por $H(z)$, que se encontra no domínio dos números complexos (z representa a frequência nesse domínio). Utilizando a substituição $z = e^{j\omega}$, com ω variando entre mais ou menos metade da taxa de amostragem, podemos calcular a resposta em frequência e em fase de um filtro. Isso significa que o domínio complexo de frequências (ou domínio z) é similar ao domínio de frequência que temos acesso através da transformada de Fourier.
Transformada z: Dada uma forma de onda discreta $x(n)$ definida para todos os valores de n, sua transformada z é definida por [Moore(1990)]:

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n},$$

onde z é uma variável complexa. A quantia z^{-1} é interpretado como um operador de *delay* unitário na amostra, pelo fato de que se $y(n) = x(n-1)$ então $Y(z) = \sum_{n=-\infty}^{\infty} x(n-1)z^{-n} = \sum_{m=-\infty}^{\infty} x(m)z^{-(m+1)} = z^{-1}X(z)$. Generalizando, a definição acima implica que se $X(z)$ é a transformada z de $x(n)$ então a transformada z de $x(n-k)$ é $z^{-k}X(z)$.

Além disso, fazendo a substituição $z = e^{j\omega}$, temos:

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n},$$

que nada mais é do que a transformada de Fourier para $x(n)$ no contexto de sinais em tempo discreto infinito, mostrando que a transformada de Fourier é um caso especial da transformada z.

A praticidade de $H(z)$ se dá por poder descrever simplificadamente a ação de qualquer filtro. Se sabemos a função de transferência $H(z)$ de um filtro, podemos calcular a transformada z de resposta para qualquer entrada ao simplesmente multiplicar $H(z)$ pela transformada z da entrada. $H(z)$ descreve uma relação entre a entrada e a saída de um filtro:

$$H(z) = \frac{Y(z)}{X(z)}$$

Vale ressaltar que $H(z)$ corresponde a uma forma de onda $h(n)$ e, como a multiplicação no domínio da frequência corresponde à convolução (definida abaixo) no domínio do tempo, podemos calcular a saída de um filtro ao realizar a convolução do sinal de entrada com $h(n)$.
Convolução: A convolução linear de duas sequências $f(n)$ e $g(n)$ de tamanhos finitos N_f e N_g, respectivamente, é definida como [Moore(1990)]:

$$h(n) = f(n) * g(n) = \sum_{m=0}^{N_f} f(m)g(n-m),$$

onde tanto f como g possuem zeros em todos os valores fora dos intervalos de 0 até $N_f - 1$

e de 0 até $N_g - 1$, respectivamente.

2.2.1 Equação do filtro

Alguns filtros possuem a propriedade de serem definidos por simples relações matemáticas. A seguir definiremos algumas classes de filtros que nos possibilitam a definição de uma equação que os representa e que é implementável em tempo real:

- **Filtro linear:** Dizemos que um filtro é linear se ele obedece o chamado princípio de sobreposição.

  Princípio de sobreposição: Se $y_1(n)$ é a saída produzida por um filtro em resposta ao sinal de entrada $x_1(n)$ e $y_2(n)$ é a resposta do filtro à entrada $x_2(n)$, então um filtro é dito linear se sua resposta à entrada $ax_1(n) + bx_2(n)$ é $ay_1(n) + by_2(n)$, com a e b sendo constantes arbitrárias [Moore(1990)].

- **Filtro causal:** Se a saída de um filtro depende apenas de saídas e entradas passadas e presentes, ele é chamado de filtro causal. Para um filtro causal não é necessário que o sinal digital esteja inteiramente armazenado na memória do computador, pois para uma determinada saída não iremos considerar entradas e saídas que ainda não aconteceram [Moore(1990)].

- **Filtro invariante no tempo:** Se a resposta de um filtro não muda ao longo do tempo, dizemos que ele é invariante no tempo [Moore(1990)]; tecnicamente, isso equivale a dizer que para qualquer entrada $x(n)$ com saída correspondente $y(n)$, vale que $x(n-k)$ produz como saída $y(n-k)$, independentemente de k.

Uma importante classe de filtros lineares, causais e invariantes no tempo corresponde à família de filtros descritos por uma equação (chamada equação do filtro) da forma:

$$y(n) = \sum_{i=0}^{M} a_i x(n-i) - \sum_{j=1}^{N} b_j y(n-j),$$

onde $x(n)$ é a entrada, $y(n)$ a saída e a_i e b_j são coeficientes constantes que determinam as características do filtro. O primeiro somatório especifica como a saída do filtro depende das entradas passadas e da entrada presente, enquanto o segundo somatório especifica como a saída atual depende das saídas passadas.

2.2.2 Polos e zeros

Em termos gerais, polos estão localizados no plano complexo em frequências que um filtro tende a enfatizar, enquanto zeros estão localizados em frequências que o filtro tende a reter ou rejeitar. Na equação do filtro apresentada anteriormente os $M + 1$ coeficientes a_i

determinam a posição dos M zeros no domínio complexo de frequências e os N coeficientes b_j determinam a posição dos N polos.

Utilizando a propriedade de que a transformada z de $x(n-k)$ é $z^{-k}X(z)$, juntamente com a linearidade dessa transformada, temos que a equação do filtro pode ser expressa como (demonstração retirada do livro *Elements of Computer Music* [Moore(1990)]):

$$Y(z) = \sum_{i=0}^{M} a_i z^{-i} X(z) - \sum_{j=1}^{N} b_j z^{-j} Y(z).$$

Fatorando $Y(z)$ do lado esquerdo e $X(z)$ do lado direito, temos:

$$Y(z)(1 + b_1 z^{-1} + ... + b_N z^{-N}) = X(z)(a_0 + a_1 z^{-1} + ... + a_M z^{-M}),$$

e colocando na forma da função de transferência:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(a_0 + a_1 z^{-1} + ... + a_M z^{-M})}{(1 + b_1 z^{-1} + ... + b_N z^{-N})} = \frac{N(z)}{D(z)}$$

Onde $N(z)$ e $D(z)$ são polinômios cujas raízes representam respectivamente os zeros e polos da função de transferência. Qualquer z que seja raiz de $N(z)$ terá o efeito de tornar $H(z)$ zero, definindo um zero da função de transferência. Em contrapartida, qualquer z que seja raiz de $D(z)$ terá o efeito de tornar $H(z)$ infinito, definindo um polo.

O teorema fundamental da álgebra nos diz que qualquer polinômio de grau k possui exatamente k raízes complexas (que podem ser distintas entre si ou possuir multiplicidade maior do que 1). Isso significa que um filtro que possui função de transferência como a apresentada tem exatamente N polos e M zeros.

2.2.3 Exemplos de filtros

Além do sinal de entrada, filtros podem ter como entrada parâmetros que expressam intervalos de frequências (ou bandas) a serem enfatizadas ou rejeitadas. Esses parâmetros definem valores dos N coeficientes referentes aos polos e dos M coeficientes referentes aos zeros do filtro. Essa estrutura possibilita um ajuste simplificado do filtro, sem introduzir complexidade computacional, o que é importante para implementações que possibilitam controle em tempo real. A seguir apresentaremos de forma simplificada alguns filtros elementares que dão origem a diversos efeitos lineares.

Filtros *Allpass* (AP)

Em termos gerais, filtros *allpass* deixam passar todas as componentes senoidais que estão presentes na entrada sem alterações de amplitude, modificando apenas a fase do sinal de entrada. Como veremos, essa propriedade permite a implementação de filtros que possuem resposta de magnitude variável. Analisaremos filtros *allpass* de primeira e segunda ordem,

que possuem respectivamente um e dois parâmetros responsáveis por modificar a resposta de fase do filtro.

- ***Allpass* de primeira ordem:**

  O filtro AP de primeira ordem pode ser descrito pela função de transferência:

$$A(z) = \frac{z^{-1} + c}{1 + cz^{-1}}$$
$$c = \frac{\tan(\pi f_c/f_s) - 1}{\tan(\pi f_c/f_s) + 1},$$

  onde f_s é a taxa de amostragem e f_c é a frequência de corte, um valor que deve estar entre zero e metade da taxa de amostragem. Observemos que por construção da função de transferência temos um polo em $z = -c$ e um zero em $z = -c^{-1}$.

  Analizando a resposta da magnitude para $z = e^{j\omega}$:

$$|A(e^{j\omega})| = \frac{|e^{-j\omega} + c|}{|1 + ce^{-j\omega}|}.$$

  Utilizando que $|e^{j\omega}| = 1$, temos:

$$|A(e^{j\omega})| = \frac{|e^{-j\omega} + c|}{|1 + ce^{-j\omega}||e^{j\omega}|} = \frac{|e^{-j\omega} + c|}{|e^{j\omega} + c|}.$$

  Pela relação de Euler:

$$|A(e^{j\omega})| = \frac{|(\cos\omega + c) - i\sin\omega|}{|(\cos\omega + c) + i\sin\omega|}.$$

  Como $|z| = |a \pm ib| = \sqrt{a^2 + b^2}$ para z complexo:

$$|A(e^{j\omega})| = \frac{\sqrt{(\cos\omega + c)^2 + \sin^2\omega}}{\sqrt{(\cos\omega + c)^2 + \sin^2\omega}} = 1$$

  Isso significa que o filtro não intensifica nem atenua nenhuma frequência $z = e^{j\omega}$, como especificamos anteriormente. Além disso, dissemos que filtros AP alteram a fase do sinal de entrada. A resposta de fase do filtro é definida por:

$$pha[A(e^{j\omega})] = pha\left[\frac{e^{-j\omega} + c}{1 + ce^{-j\omega}}\right] = \tan^{-1}\left(\frac{-\sin\omega}{\cos\omega + c}\right) - \tan^{-1}\left(\frac{-c\sin\omega}{c\cos\omega + 1}\right),$$

  mostrando que a fase é modificada de forma não-linear.

  Da função de transferência chegamos na equação correspondente:

$$y(n) = cx(n) + x(n-1) - cy(n-1).$$

  O diagrama de bloco da equação acima é dado pela figura 2.3. Para $f_c = 0.1f_s$ temos

as respostas de fase e magnitude representadas na figura 2.4.

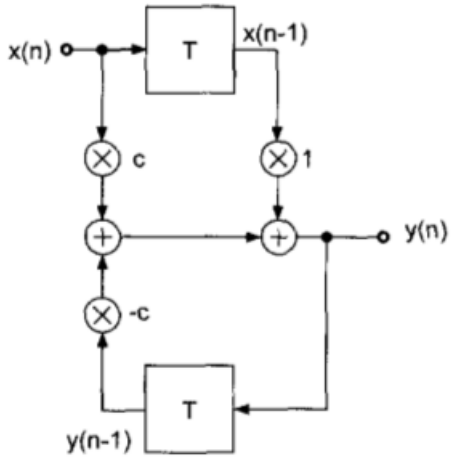

Figura 2.3: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

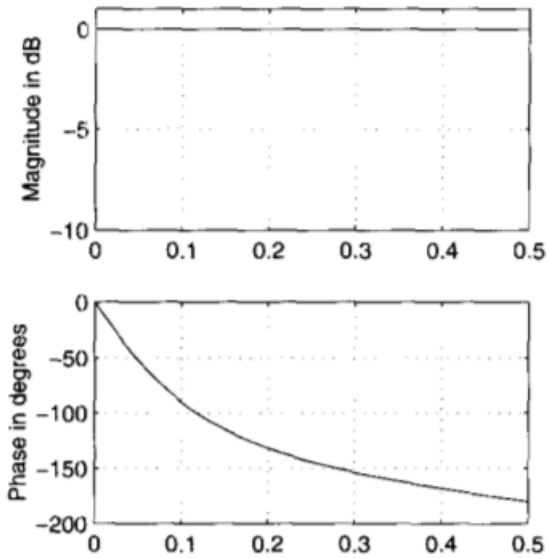

Figura 2.4: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

- ***Allpass* de segunda ordem:**

  O filtro AP de segunda ordem pode ser descrito pela função de transferência:

$$A(z) = \frac{-c + d(1-c)z^{-1} + z^{-2}}{1 + d(1-c)z^{-1} - cz^{-2}} \tag{2.1}$$

$$c = \frac{\tan(\pi f_b/f_s) - 1}{\tan(\pi f_b/f_s) + 1}$$

$$d = -\cos(2\pi f_c/f_s),$$

  onde f_s é a taxa de amostragem, f_c é a frequência de corte e f_b é a largura da banda. Os dois últimos valores devem estar entre zero e metade da taxa de amostragem.

Similarmente, podemos representar o filtro AP de segunda ordem pela sua função de transferência na forma polar:

$$A(z) = a_0 \frac{(1 - Re^{j\theta}z^{-1})(1 - Re^{-j\theta}z^{-1})}{(1 - (1/R)e^{j\theta}z^{-1})(1 - (1/R)e^{-j\theta}z^{-1})}. \tag{2.2}$$

Podemos colocar a função de transferência 2.1 na forma polar 2.2 através da substituição das variáveis c e d por:

$$c = -\frac{1}{R^2}$$
$$d = -\frac{2R\cos(\theta)}{R^2 + 1}.$$

Os pares de polos e zeros da função de transferência 2.2 estão representados na figura 2.5 respectivamente por X e O. Podemos notar que existe uma semelhança na estrutura dos filtros AP de primeira e segunda ordem. No primeiro caso, como estamos lidando apenas com um polo e um zero, ambos são valores puramente reais que possuem a relação $z_p = z_z^{-1}$, isto é, $R_p = R_z^{-1}$, pois $\theta = 0$. No segundo caso temos algo parecido mas para dois pares de polos e zeros complexos, que continuam nos dando $R_p = R_z^{-1}$ mas para dois valores de θ, onde $\theta_1 = -\theta_2$.

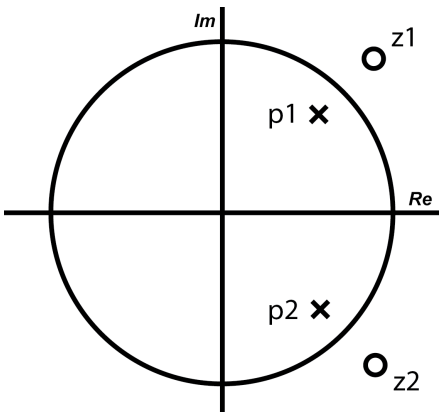

Figura 2.5: Adaptada de *Elements of Computer Music* [Moore(1990)].

Mostraremos que a resposta em magnitude de 2.2 é constante para $z = e^{j\omega}$ (demonstração retirada do livro *Elements of Computer Music* [Moore(1990)]), considerando $a_0 = 1$, sem perda de generalidade:

$$|A(e^{j\omega})| = \left| \frac{(1 - Re^{j\theta}e^{-j\omega})(1 - Re^{-j\theta}e^{-j\omega})}{(1 - (1/R)e^{j\theta}e^{-j\omega})(1 - (1/R)e^{-j\theta}e^{-j\omega})} \right|.$$

Separando a equação em dois fatores:

$$
\begin{aligned}
|A(e^{j\omega})| &= |A_1(e^{j\omega})||A_2(e^{j\omega})| \\
|A_1(e^{j\omega})| &= \left| \frac{(1 - Re^{j(\theta-\omega)})}{(1 - (1/R)e^{j(\theta-\omega)})} \right| \\
|A_2(e^{j\omega})| &= \left| \frac{(1 - Re^{j(\theta+\omega)})}{(1 - (1/R)e^{j(\theta+\omega)})} \right|.
\end{aligned}
$$

Multiplicando cada fator por seu conjugado temos a magnitude ao quadrado, logo:

$$
\begin{aligned}
|A_1(e^{j\omega})|^2 &= \frac{(1 - Re^{j(\theta-\omega)})}{(1 - (1/R)e^{j(\theta-\omega)})} \frac{(1 - Re^{-j(\theta-\omega)})}{(1 - (1/R)e^{-j(\theta-\omega)})} \\
|A_2(e^{j\omega})|^2 &= \frac{(1 - Re^{j(\theta+\omega)})}{(1 - (1/R)e^{j(\theta+\omega)})} \frac{(1 - Re^{-j(\theta+\omega)})}{(1 - (1/R)e^{-j(\theta+\omega)})}
\end{aligned}
$$

$$
\begin{aligned}
|A_1(e^{j\omega})|^2 &= \frac{1 + R^2 - 2R\cos(\theta - \omega)}{1 + 1/R^2 - (2/R)\cos(\theta - \omega)} = R^2 \\
|A_2(e^{j\omega})|^2 &= \frac{1 + R^2 - 2R\cos(\theta + \omega)}{1 + 1/R^2 - (2/R)\cos(\theta + \omega)} = R^2
\end{aligned}
$$

$$
|A(e^{j\omega})|^2 = R^4.
$$

Isso significa que $|A(e^{j\omega})| = R^2$, que é um valor constante. Como no caso do *allpass* de primeira ordem, o filtro não intensifica nem atenua nenhuma frequência $z = e^{j\omega}$. Mas apresenta resposta de fase variável, dada pela fórmula:

$$
\begin{aligned}
pha[A(e^{j\omega})] &= pha\left[\frac{N(e^{j\omega})}{D(e^{j\omega})}\right] \\
pha[A(e^{j\omega})] &= \tan^{-1}\left(\frac{Im[N(e^{j\omega})]}{Re[N(e^{j\omega})]}\right) - \tan^{-1}\left(\frac{Im[D(e^{j\omega})]}{Re[D(e^{j\omega})]}\right).
\end{aligned}
$$

Para $f_c = 0.1f_s$ e $f_b = 0.022f_s$ temos as respostas de fase e magnitude representadas na forma de gráfico na figura 2.6. Da função de transferência 2.1 chegamos na equação 2.3 e diagrama de bloco 2.7 correspondentes.

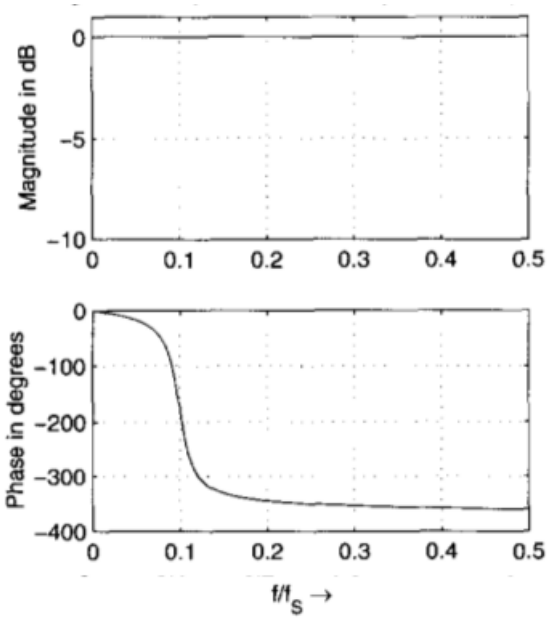

Figura 2.6: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

$$y(n) = -cx(n) + d(1-c)x(n-1) + x(n-2) - d(1-c)y(n-1) + cy(n-2) \quad (2.3)$$

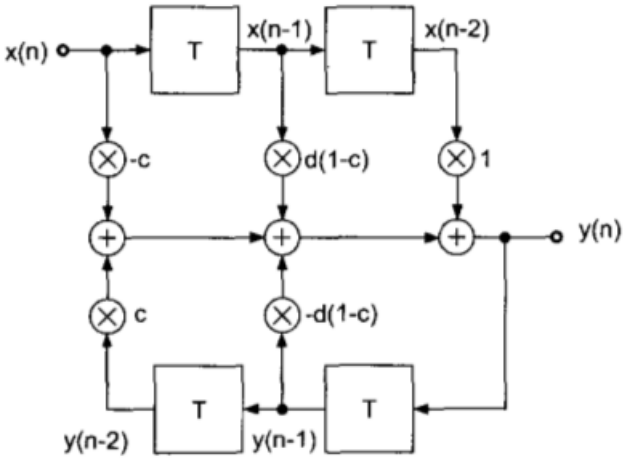

Figura 2.7: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

***Lowpass* (LP) e *highpass* (HP)**

Filtros *lowpass* selecionam as frequências abaixo de uma frequência de corte f_c, atenuando as frequências que estão acima de f_c, enquanto filtros *highpass* selecionam as frequências acima de uma frequência de corte f_c, atenuando as frequências que estão abaixo de f_c. Os filtros LP e HP podem ser implementados respectivamente ao adicionarmos e subtrairmos o sinal de entrada $x(n)$ com o sinal de saída do filtro AP de primeira ordem. Suas funções de

transferência são dadas por:

$$H(z) = \frac{1}{2}(1 \pm A(z)) \qquad \text{(LP/HP +/-)}$$
$$A(z) = \frac{z^{-1} + c}{1 + cz^{-1}}$$
$$c = \frac{\tan(\pi f_c/f_s) - 1}{\tan(\pi f_c/f_s) + 1},$$

onde $A(z)$ representa a função de transferência do filtro AP de primeira ordem, f_s é a taxa de amostragem e f_c é a frequência de corte, que deve estar entre zero e metade da taxa de amostragem. Como vimos, a saída do filtro AP de primeira ordem possui uma mudança de fase que varia de 0 grau para frequências baixas até -180 graus para frequências altas; isso faz com que as operações de soma e subtração desse sinal com o sinal de entrada levem a uma filtragem *lowpass* e *highpass* através de interferências construtivas e destrutivas induzidas pela mudança de fase de acordo com cada parcial. Multiplicamos esse sinal por $\frac{1}{2}$ para que ele continue no intervalo de -1 até 1.

Das funções de transferência chegamos nas equações correspondentes:

$$y(n) = \frac{1}{2}(x(n) \pm (cx(n) + x(n-1) - cy(n-1))). \qquad \text{(LP/HP +/-)}$$

O diagrama de bloco das equações é dado pela figura 2.8. Para $f_c = 0.1 f_s$ temos as respostas de fase e magnitude representadas na figura 2.9.

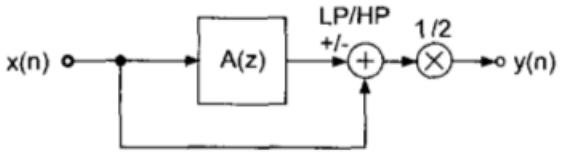

Figura 2.8: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

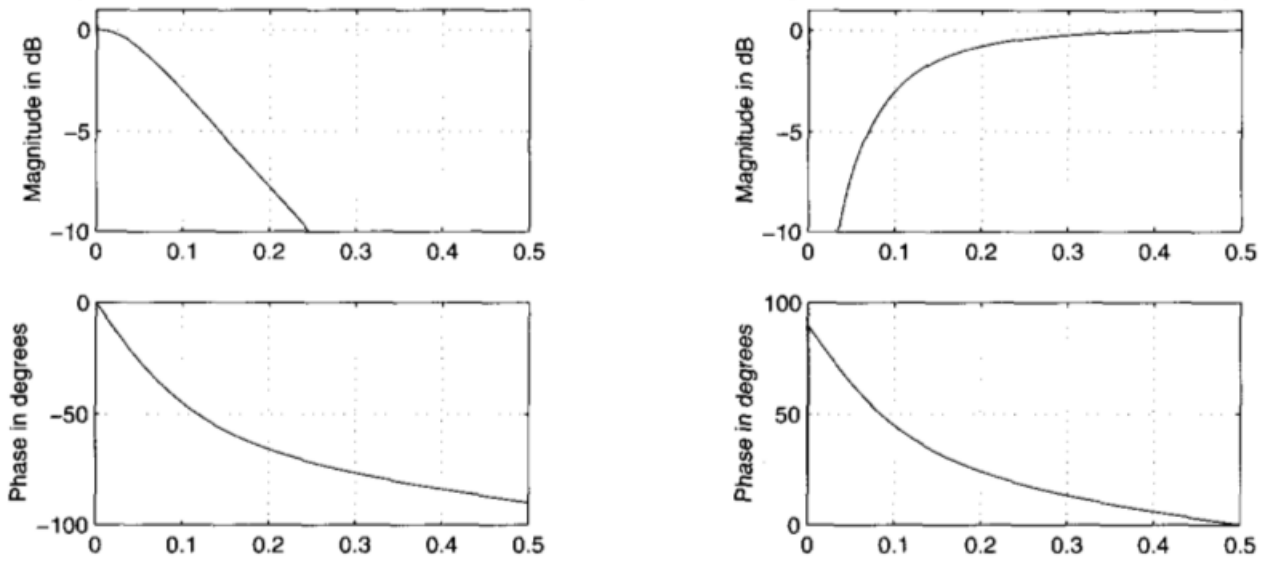

Figura 2.9: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

Bandpass (BP) e *bandreject* (BR)

Filtros *bandpass* selecionam as frequências que estão entre duas frequências de corte f_{cl} e f_{ch}, onde $f_{cl} < f_{ch}$, atenuando frequências que estão abaixo de f_{cl} e acima de f_{ch}. Em contrapartida, filtros *bandreject* atenuam as frequências que estão entre duas frequências de corte f_{cl} e f_{ch}, deixando passar frequências que estão abaixo de f_{cl} e acima de f_{ch}. Os filtros BR e BP podem ser implementados respectivamente ao adicionarmos e subtrairmos o sinal de entrada $x(n)$ com o sinal de saída do filtro AP de segunda ordem. Suas funções de transferência são dadas por:

$$
\begin{aligned}
H(z) &= \frac{1}{2}(1 \pm A(z)) && \text{(BR/BP +/-)} \\
A(z) &= \frac{-c + d(1-c)z^{-1} + z^{-2}}{1 + d(1-c)z^{-1} - cz^{-2}} \\
c &= \frac{\tan(\pi f_b/f_s) - 1}{\tan(2\pi f_b/f_s) + 1} \\
d &= -\cos(2\pi f_c/f_s),
\end{aligned}
$$

onde $A(z)$ representa a função de transferência do filtro AP de segunda ordem apresentado anteriormente, f_s é a taxa de amostragem, f_c é a frequência de corte e f_b é a largura da banda. Os dois últimos valores devem estar entre zero e metade da taxa de amostragem. Como vimos, a saída do filtro AP de segunda ordem possui uma mudança de fase que varia de 0 grau para frequências baixas até -360 graus para frequências altas, isso faz com que as operações de soma e subtração desse sinal com o sinal de entrada levem a uma filtragem *bandpass* e *bandreject* através de interferências construtivas e destrutivas induzidas pela mudança de fase de acordo com cada parcial. Multiplicamos esse sinal por $\frac{1}{2}$ para que ele continue no intervalo de -1 até 1.

O diagrama de bloco é dado pela figura 2.10. Para $f_c = 0.1f_s$ e $f_b = 0.022f_s$ temos as respostas de fase e magnitude representadas na figura 2.11.

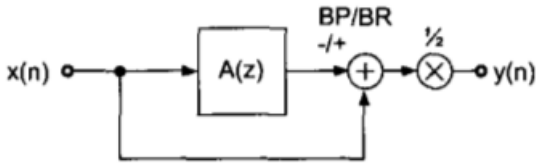

Figura 2.10: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

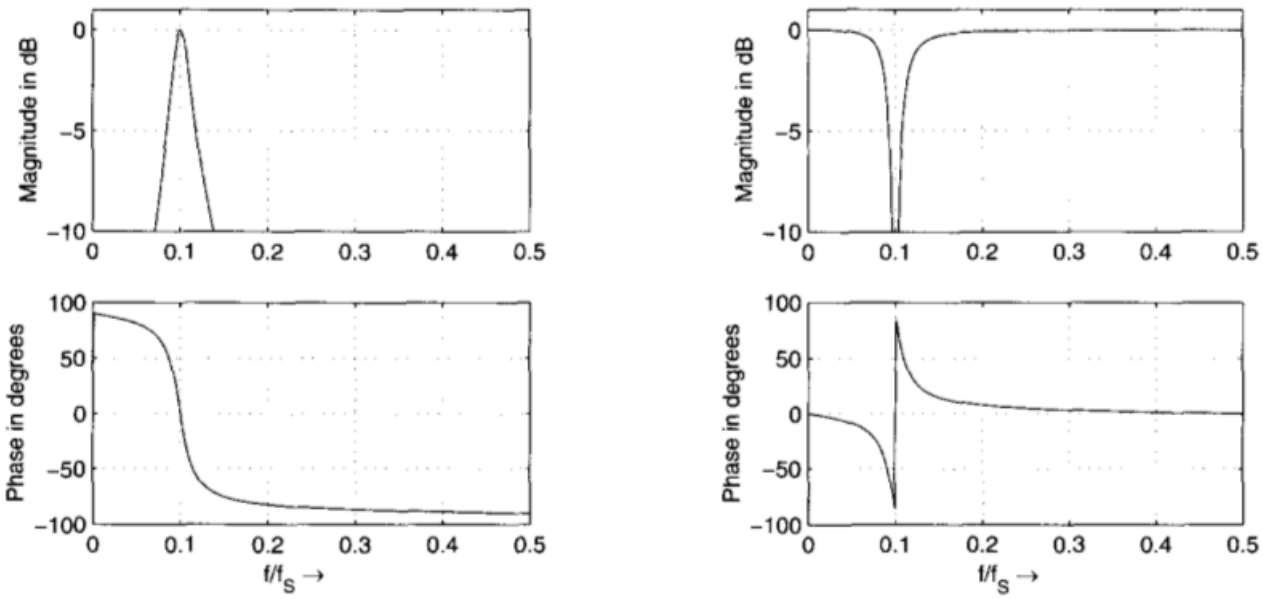

Figura 2.11: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

Wah-wah

O *wah-wah* é um filtro dinâmico, normalmente controlado por pedal, que pode ser implementado a partir da mixagem do sinal de entrada puro $x(n)$ com o sinal de saída de um filtro *bandpass* de largura de banda pequena e uma frequência central f_c dada pela abertura do pedal. O filtro apresenta um efeito parecido com a fala da palavra "wah" quando variamos a frequência central, algo parecido com o que ocorre no processo de fala humana, passando de um som mais "fechado" (com menos parciais) para um som mais "aberto" (com mais parciais). O efeito pode ser representado pelo diagrama de bloco da figura 2.12.

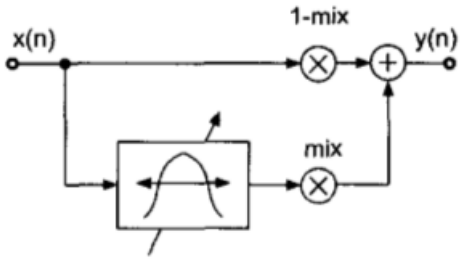

Figura 2.12: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

Diferentemente dos filtros apresentados anteriormente, podemos perceber que o *wah-wah* é um efeito atingido pela variação de seus parâmetros ao longo do tempo. Isso traz uma dificuldade para processamento de sinais segmentados em blocos quando estes têm tamanho grande, ou quando a mudança dos parâmetros é feita de forma abrupta, pois para cada processamento de bloco os parâmetros de entrada são fixos. Uma solução simples para esse problema é guardar os parâmetros de entrada para o bloco anterior, retornando uma saída que transita linearmente da resposta gerada pelos parâmetros do bloco anterior para a resposta gerada com os parâmetros do bloco atual.

2.3 Técnicas não-lineares

Em relação aos filtros lineares apresentados na seção anterior, técnicas não-lineares se diferenciam por possibilitar a criação intencional ou involuntária de componentes de frequência (harmônicas ou não) que não estão presentes no sinal de entrada original. Esses processos são usados na implementação de efeitos como *pitch shifting*, *time stretching*, *overdrive* e *distortion*.

Neste trabalho exploraremos as técnicas de *waveshaping*, *time stretching* e *pitch shifting*. Na definição de *waveshaping* utilizaremos a operação de composição de funções, mostrando algumas classes e subclasses dessa técnica, bem como seus reflexos em relação às representações digitais discutidas anteriormente. Para *time stretching* e *pitch shifting* utilizaremos os processos de reamostragem, segmentação e interpolação como base, chegando a uma forma de modificar o tempo de duração e a frequência de um sinal de forma independente.

2.3.1 *Waveshaping*

Waveshaping é uma forma de moldar o sinal de entrada através de uma função de mapeamento não-linear f. Esse processo é uma composição de funções, onde cada amostra $x(n)$ da entrada é mapeada em um novo valor de amplitude de acordo com a função de transferência na forma $f(x(n))$.

A função de transferência f pode ser definida por uma expressão analítica ou por mais de uma expressão. Nesse caso, particionamos o domínio de amplitude em seções disjuntas

e associamos a cada seção uma função, que se conecta continuamente com as funções das seções vizinhas.

Corte simétrico

A técnica de corte simétrico (ou *symmetrical clipping*) consiste em definir uma função de transferência f que apresenta uma expressão linear para níveis baixos de amplitude de entrada e uma constante para níveis mais altos. Por conta da simetria, f é uma função ímpar que define um valor max constante para valores de amplitude maiores ou iguais a um determinado θ e $-max$ para valores de amplitude menores ou igual a $-\theta$.

Como nesse caso f é uma função ímpar, podemos simplificar sua definição ao representarmos apenas a parte positiva do mapeamento. Na figura 2.13 é representada graficamente uma função de transferência possível para corte simétrico com $max = 0.3$ e $\theta = 0.3$ em (b), além de um exemplo de entrada (a) e saída (c). A função é definida por:

$$f(x) = \begin{cases} x & \text{para } 0 \leq x \leq 0.3 \\ 0.3 & \text{para } 0.3 \leq x \leq 1. \end{cases}$$

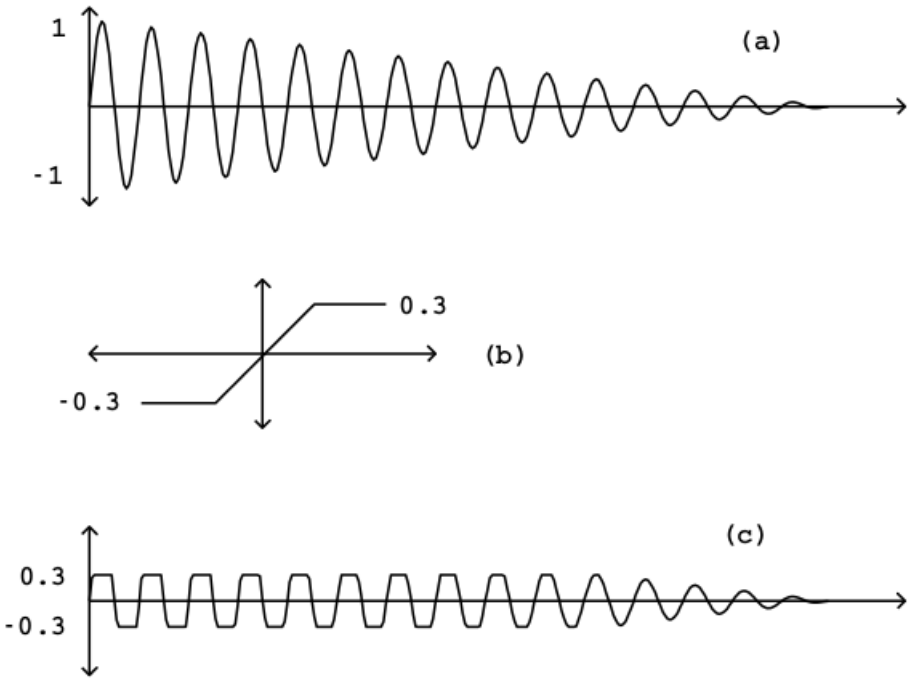

Figura 2.13: Retirada de *The Theory and Technique of Electronic Music* [Puckette(2007)].

A função representada na imagem 2.13 implementa o que podemos chamar de *symmetrical hard clipping*. Essa subclasse de corte simétrico define um corte para valores de amplitude maiores que max sem suavizar a saída na transição de uma sentença para a outra, gerando uma onda aproximadamente quadrada. Como a onda resultante possui descontinuidades em sua derivada, seu espectro apresenta harmônicos de frequências altas com maior ênfase, sendo que este efeito desaparece abruptamente uma vez que a onda se torne mais fraca

(com amplitude menor do que θ). Podemos observar esse comportamento na figura 2.14, que apresenta o espectro de uma senoide de 1 kHz que decai ao passar do tempo.

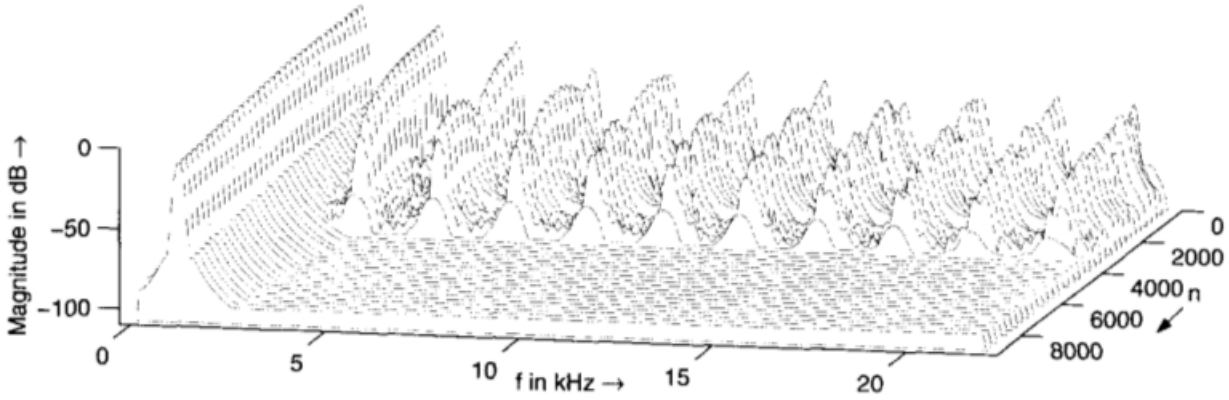

Figura 2.14: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

Uma outra subclasse possível é chamada *symmetrical soft clipping*. Nesse caso é feita a suavização da saída na transição entre as sentenças que definem a função de transferência. Isso pode ser feito com a adição de uma expressão intermediária que tem característica não-linear. Uma função possível [Zölzer(2011)] para $max = 2/3$ e $\theta = 1/3$ é:

$$f(x) = \begin{cases} 2x & \text{para } 0 \leq x \leq 1/3 \\ \frac{3-(2-3x)^2}{3} & \text{para } 1/3 \leq x \leq 2/3 \\ 1 & \text{para } 2/3 \leq x \leq 1. \end{cases} \tag{2.4}$$

O comportamento espectral do *symmetrical soft clipping* para uma senoide de 1 kHz que decai ao passar do tempo é dado pela figura 2.15. Podemos observar que os harmônicos de frequências mais altas têm intensidades que decaem de forma suave, conforme a entrada vai decaindo.

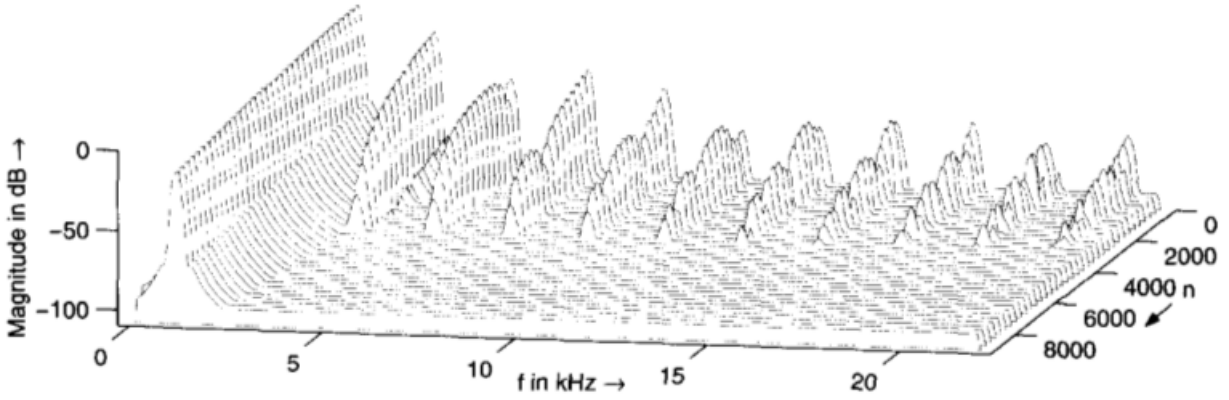

Figura 2.15: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

Uma forma de encontrar uma função de transferência como a que foi dada por 2.4 seria considerar que temos uma expressão linear l, uma de segundo grau q e outra que é constante

em 1. Como restrições iniciais temos que essas sentenças devem se tocar em seus extremos θ_1 e θ_2, com $\theta_1 < \theta_2$. Além disso, queremos transições suavizadas entre as três sentenças, o que é equivalente a igualar as derivadas das sentenças em seus pontos extremos. Quando consideramos essas restrições, podemos montar um sistema linear contendo como variáveis os coeficientes de l e q em termos de θ_1 e θ_2 dados.

Queremos encontrar:

$$\begin{aligned} l(x) &= ax + b \\ q(x) &= cx^2 + dx + e, \end{aligned}$$

cujas restrições são:

$$\begin{aligned} l(\theta_1) = q(\theta_1) &\implies c\theta_1^2 + (d-a)\theta_1 + (e-b) = 0 \\ q(\theta_2) = 1 &\implies c\theta_2^2 + d\theta_2 + e = 1 \\ l'(\theta_1) = q'(\theta_1) &\implies 2c\theta_1 + (d-a) = 0 \\ q'(\theta_2) = 0 &\implies 2c\theta_2 + d = 0, \end{aligned}$$

que é um sistema linear equivalente a:

$$\begin{bmatrix} -\theta_1 & -1 & \theta_1^2 & \theta_1 & 1 \\ 0 & 0 & \theta_2^2 & \theta_2 & 1 \\ -1 & 0 & 2\theta_1 & 1 & 0 \\ 0 & 0 & 2\theta_2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}. \tag{2.5}$$

Resolvendo o sistema e tomando b como parâmetro, temos:

$$\begin{aligned} e &= \frac{\theta_1^2 - 2b(\theta_2 - \theta_2^2/2)}{\theta_1^2 - 2(\theta_2 - \theta_2^2/2)} \\ d &= \frac{2(b-e)}{\theta_1^2} \\ c &= -d/2 \\ a &= d + 2\theta_1 c. \end{aligned}$$

As fórmulas que calculam os coeficientes dados acima já estão dispostas de forma a minimizar o número de operações que serão realizadas pelo computador. Além disso, podemos considerar que o coeficiente b é sempre 0, o que significa que temos saída 0 para entrada 0, e é conveniente por diminuir o número de operações realizadas no cálculo de e e d.

Corte assimétrico

Diferentemente do corte simétrico, a técnica de corte assimétrico (ou *asymmetrical clipping*) é definida por um mapeamento assimétrico do sinal quando comparamos sua parcela positiva com sua parcela negativa. A função de transferência que implementa essa característica é não ímpar e normalmente possui apenas uma expressão não-linear.

Uma possível função de transferência para corte assimétrico é dada pela equação 2.6 [Zölzer(2011)]. Essa função implementa um efeito que simula a distorção gerada por um amplificador de tubo. Na figura 2.16 temos o comportamento espectral da função aplicada a uma onda de 1kHz. Podemos observar que quanto maiores forem as frequências dos harmônicos adicionados ao sinal, menores serão suas intensidades.

$$f(x) = \frac{x - Q}{1 - e^{-dist \cdot (x-Q)}} + \frac{Q}{1 - e^{dist \cdot (x-Q)}}. \tag{2.6}$$

O parâmetro Q controla quão próximo de um mapeamento linear será a função para valores baixos de entrada, tendo resposta "mais linear" para valores mais negativos de Q. Já o parâmetro $dist > 0$ controla o nível de distorção. Quanto maior seu valor, mais próximo de uma onda quadrada estará o sinal de saída de uma senoide.

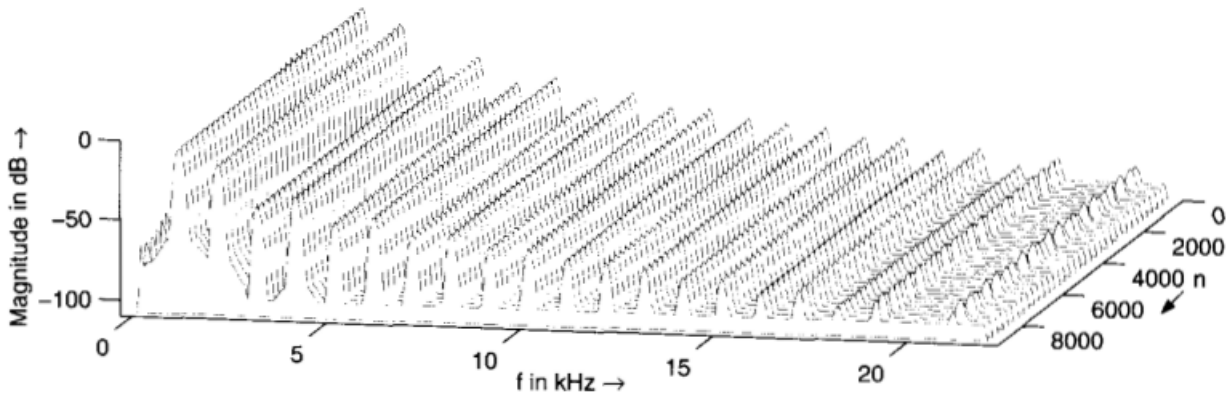

Figura 2.16: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

2.3.2 *Time stretching*

Para algumas aplicações podemos estar interessados em prolongar ou diminuir a duração de uma informação de áudio. A implementação mais direta nesse caso é a realização da reamostragem do conjunto de amostras em uma frequência diferente da original. Porém, esse processo não só altera a duração do áudio em questão como também altera sua frequência. *Time stretching* é um processo que realiza a compressão ou extensão da duração de um arquivo de áudio sem que sejam alteradas as informações referentes à frequência do mesmo. Em outras palavras, queremos escalar o número de amostras do arquivo de áudio em relação a um fator α (normalmente entre 0.25 e 2) sem que ocorram mudanças significativas em seu espectro.

Um algoritmo que implementa o *time stretching* é o SOLA (*Synchronous Overlap and Add*). Ele é baseado em técnicas de correlação entre vetores através da ideia de processar o sinal de entrada em segmentos. Quando queremos aumentar o número de amostras ($\alpha > 1$) alguns segmentos são repetidos, e quando queremos comprimir a duração do áudio ($\alpha < 1$) alguns segmentos são descartados. Um esboço visual do algoritmo é dado pela figura 2.17. Nela temos S_a representando o tamanho do segmento original e $S_s = \alpha S_a$ representando o tamanho do segmento resultante, dimensionado pelo fator α. A área em preto representa o intervalo de sobreposição de tamanho L dos segmentos dimensionados. Encontramos o melhor ponto de sobreposição entre os dois segmentos dimensionados através do cálculo da máxima similaridade entre pedaços deles. Somamos um *fade out* do pedaço do final da sequência anterior a um *fade in* do pedaço do início da sequência seguinte para suavizar a transição entre os segmentos, que podem ser consideravelmente distintos.

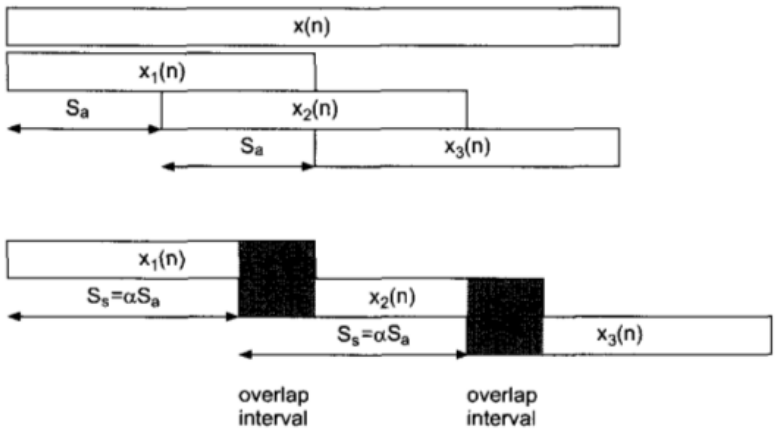

Figura 2.17: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

Exemplos de saídas do algoritmo para diferentes valores de α são apresentados na figura 2.18. Os sinais de saída podem ser analisados pelo ponto de vista do número de amostras e do seu espectro, respectivamente dos lados esquerdo e direito da figura. Podemos observar que o número de amostras é dimensionado por α em todas as saídas, com poucas diferenças entre os espectros para diferentes valores de α.

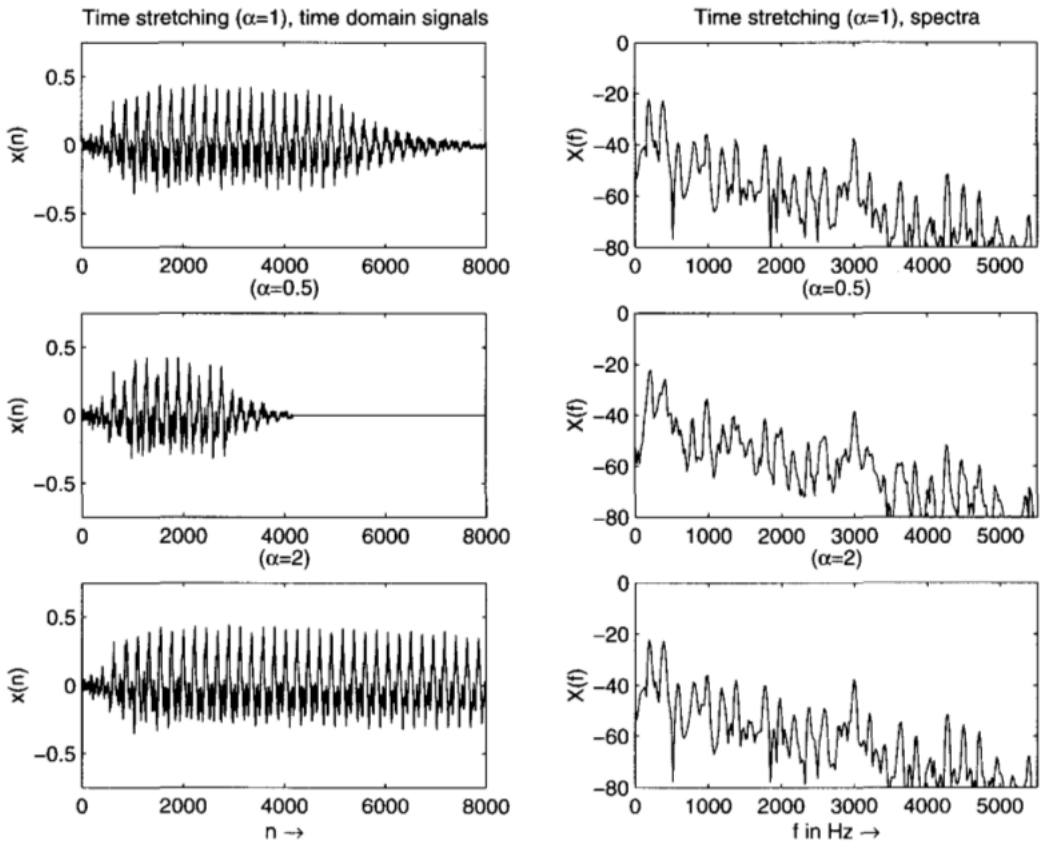

Figura 2.18: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

2.3.3 *Pitch shifting*

Pitch shifting é uma forma de transpor a entrada de áudio para uma afinação diferente da que está sendo tocada, mantendo a duração de tempo original. Em outras palavras, queremos escalar cada frequência presente no sinal de entrada por um fator α entre 0.25 e 2, sem alterar a duração do arquivo de áudio. Uma aplicação típica desse efeito é a correção da intonação musical (*autotune*) tanto para cantores como para instrumentos musicais [Zölzer(2011)]. Assim como no caso do *time stretching*, esse é um efeito que pode ser parcialmente obtido ao fazermos a reamostragem do sinal original em uma frequência diferente. Mas simplesmente com uma reamostragem, o sinal de saída terá duração diferente da entrada, não gerando uma transposição condizente com a entrada.

Uma forma de implementar o *pitch shifting* é através do auxílio do algoritmo SOLA apresentado anteriormente. Com ele podemos realizar a expansão ou compressão do tamanho da entrada antes de fazer o processo de reamostragem. Quando queremos transpor a entrada para uma frequência mais alta ($\alpha > 1$), por exemplo, basta escalarmos a duração N_1 da entrada para $N_2 = \alpha N_1$ antes (ou depois) de fazermos a reamostragem da entrada a uma taxa dimensionada por $N_1/N_2 = \alpha$. Esse processo gera um sinal de saída que não apenas tem a mesma dimensão N_1 da entrada original, como também possui todas as suas componentes de frequência com um *shift* proporcional a α.

Exemplos de saídas do algoritmo para diferentes valores de α são apresentados na figura

2.19. Os sinais de saída podem ser analisados pelo ponto de vista da forma de onda e do seu espectro, respectivamente dos lados esquerdo e direito da figura. Podemos observar que, apesar da mudança no aspecto do sinal (referente ao *shift* na frequência), o número de amostras é o mesmo para todas as saídas. Do ponto de vista dos espectros, podemos observar que a saída para $\alpha = 0.5$ apresenta espectro concentrado em frequências mais baixas, enquanto a saída para $\alpha = 2$ apresenta espectro com componentes de frequências altas mais intensas que no sinal original.

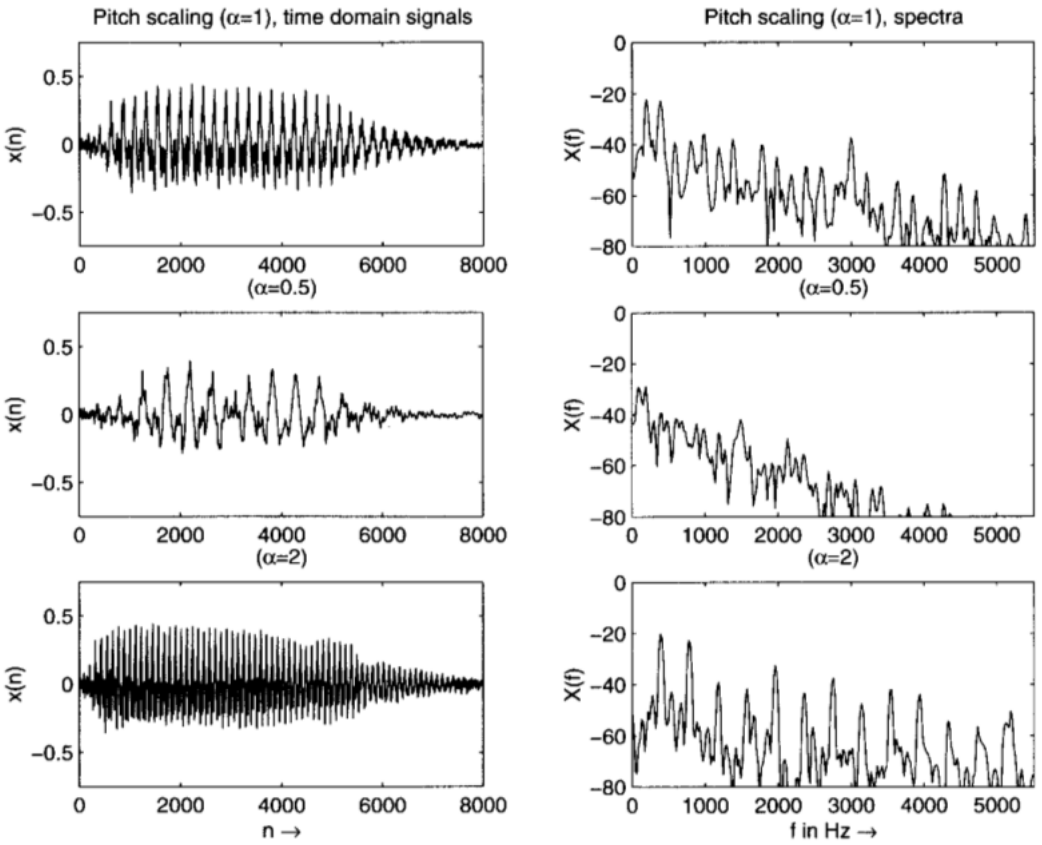

Figura 2.19: Retirada de *DAFX: Digital Audio Effects* [Zölzer(2011)].

Capítulo 3

Implementação e experimentos

A implementação de efeitos digitais de áudio em tempo real se dá pela criação de *plugins* de áudio, que podem ser empregados em diversos *hosts* por meio de um protocolo. Os *hosts* mais utilizados são *softwares* comerciais referentes à mixagem e edição musical, também chamados de DAWs (*Digital Audio Workstations*). Entre os mais famosos, podemos citar: *Pro Tools* [Pro()], *REAPER* [REA()], *Steinberg Cubase* [Cub()], *Audacity* [Aud()] e *Ardour* [Ard()]. Para a implementação desse trabalho foi escolhida a interface definida pelo padrão LV2 [LV2()], principalmente por se tratar de um padrão aberto que é aceito pela maioria dos *softwares* musicais, proprietários ou não. Essa interface será explicada em maiores detalhes na seção 3.1.

Os efeitos implementados em padrão LV2 serão apresentados na seção 3.2. Como veremos, alguns desses efeitos podem ser implementados de forma direta ao partirmos das definições apresentadas no capítulo 2, enquanto outros possuem implementação não trivial para aplicações em tempo real. Na seção 3.3 serão apresentados experimentos sobre os efeitos implementados para verificarmos se os resultados obtidos se encontram de acordo com os modelos definidos no capítulo 2.

3.1 Padrão LV2

Plugins no formato LV2 são instalados em diretórios estruturados, chamados *bundles*. Os *plugins* são tipicamente separados em duas partes: código e dados. O código deve estar em uma linguagem compatível com C, pois a API desse padrão consiste de um *header* em C contendo métodos típicos de plugins de áudio. Já os dados que são trocados entre o *plugin* e o *host* durante o processamento devem estar definidos em sintaxe *Turtle* [Tur()]. Dentro de um *bundle* existem três arquivos: `manifest.ttl`, `nome-plugin.ttl` e `nome-plugin.so`. Os dois primeiros definem a interface de dados e o terceiro o executável das funções que foram implementadas.

3.1.1 Interface de dados

Hosts verificam os arquivos `manifest.ttl` de todos os *bundles* presentes num determinado diretório do computador para descobrirem quais *plugins* estão presentes. O arquivo `manifest.ttl` apresenta uma definição simplificada do *plugin*, informando seu URI (*Uniform Resource Identifier*), o nome de seu executável .so e o nome de um outro arquivo .ttl que detalha melhor quais serão os buffers e parâmetros compartilhados entre o *plugin* e o *host*. No código 3.1, em sintaxe *Turtle*, expressamos que o URI meu-uri se referencia a um *plugin* LV2 que possui executável `nome-plugin.so` e interface de dados definida em `nome-plugin.ttl`.

Código 3.1: Exemplo de um arquivo manifest.ttl

```
@prefix lv2:  <http://lv2plug.in/ns/lv2core#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<meu-uri>
  a lv2:Plugin ;
  lv2:binary <nome-plugin.so>  ;
  rdfs:seeAlso <nome-plugin.ttl> .
```

A descrição completa do *plugin* se encontra no arquivo `nome-plugin.ttl`. Aqui, além de definirmos que meu-uri é um *plugin*, podemos definir sua licença, e associá-lo a um projeto e a alguma classe de *plugins*, como por exemplo dizer que ele implementa um efeito de distorção. Cada uma das portas de entrada, saída e controle devem ser listadas e especificadas com um nome, um símbolo, um índice, e um intervalo de valores aceitos. No código 3.2, em sintaxe *Turtle*, temos a definição da interface de dados do *plugin* de nome "Exemplo", com porta de áudio de entrada, porta de áudio de saída e porta de controle de ganho que aceita valores entre 0.0 e 1.0, com valor *default* 0.5.

Código 3.2: Exemplo de um arquivo nome-plugin.ttl

```
@prefix doap:  <http://usefulinc.com/ns/doap#> .
@prefix lv2:   <http://lv2plug.in/ns/lv2core#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

<meu-uri>
  a lv2:Plugin ,
    lv2:DistortionPlugin ;

  lv2:project <meu-projeto> ;

  doap:name "Exemplo" ;
  doap:license <http://opensource.org/licenses/isc> ;

  lv2:port [
    a lv2:InputPort ,
      lv2:ControlPort ;
    lv2:index 0 ;
    lv2:symbol "gain" ;
    lv2:name "Gain" ;
    lv2:default 0.5 ;
```

```
    lv2:minimum 0.0 ;
    lv2:maximum 1.0 ;
  ] , [
    a lv2:AudioPort ,
      lv2:InputPort ;
    lv2:index 1 ;
    lv2:symbol "input" ;
    lv2:name "Input"
  ] , [
    a lv2:AudioPort ,
      lv2:OutputPort ;
    lv2:index 2 ;
    lv2:symbol "output" ;
    lv2:name "Output"
  ] .
```

3.1.2 Código

O código que implementa alguma aplicação deve ser escrito em linguagem compatível com C, sendo posteriormente compilado em um executável .so que será adicionado ao *bundle* da aplicação. Códigos em C que implementam métodos referentes a *plugins* no padrão LV2 possuem um *template* como o ilustrado no código 3.3. Vale ressaltar que não podemos alocar memória no contexto da função *run*, pois essa é uma operação demorada e será nesse contexto que geraremos a saída, em tempo real, para uma entrada de áudio dada, sendo necessário minimizar o atraso na resposta. O mais usual é declarar no escopo da função todas as variáveis que serão utilizadas quando não for necessário utilizar informação de entradas passadas. Quando informações passadas são necessárias, podemos adicionar variáveis à estrutura do *plugin* e atualizá-las a cada chamada da função *run*.

Código 3.3: Template do código de um plugin LV2 em C

```
#include "lv2/lv2plug.in/ns/lv2core/lv2.h"

#define URI "meu-uri"

typedef enum {
  /* Indices das portas definidas pela interface de dados */
} PortIndex;

typedef struct {
  /* Estrutura dos buffers e parametros utilizados */
} Plugin;

static LV2_Handle
instantiate(const LV2_Descriptor*     descriptor,
            double                    rate,
            const char*               bundle_path,
            const LV2_Feature* const* features)
{
  /* Alocacao da estrutura definida, retornando uma instancia do plugin */
}

static void
connect_port(LV2_Handle instance,
```

```
              uint32_t    port,
              void*       data)
{
  /* Atribuicao de um dado data a estrutura, dependendo do indice port passado */
}

static void
activate(LV2_Handle instance)
{
  /* Funcao executada quando o plugin muda seu estado para ativo */
}

static void
run(LV2_Handle instance, uint32_t n_samples)
{
  /* Rotina principal do plugin */
}

static void
deactivate(LV2_Handle instance)
{
  /* Funcao executada quando o plugin muda seu estado para desativado */
}

static void
cleanup(LV2_Handle instance)
{
  /* Liberacao da memoria alocada */
}

static const void*
extension_data(const char* uri)
{
  /* Retorna possiveis dados adicionais fornecidos pelo plugin */
}

static const LV2_Descriptor descriptor = {
  /* Descritor estatico do plugin */
  URI,
  instantiate,
  connect_port,
  activate,
  run,
  deactivate,
  cleanup,
  extension_data
};
```

3.2 Efeitos implementados

A maior parte de *plugins* desenvolvidos em padrão LV2 vieram da família de efeitos não-lineares. Entre eles, foram implementados: *fuzz*, simulador de tubo, *pitch shifting* e duas versões de *overdrive*. Da família de filtros lineares, foram desenvolvidas duas versões do efeito *wah-wah*. Por questões de praticidade serão apresentados códigos em Python das rotinas principais de cada *plugin*, sendo citada em texto a interface de dados necessária para

o funcionamento. Para simplificar o código, consideramos que as variáveis passadas como parâmetro armazenam globalmente as alterações feitas no interior das funções.

3.2.1 *Overdrive*

O *overdrive* é um efeito não-linear que pode ser implementado com a técnica de corte simétrico apresentada em 2.3.1. Sua equação de transferência é normalmente definida por três expressões, e a característica do som gerado vem principalmente da expressão não-linear [Zölzer(2011)].

O primeiro *plugin* implementado foi baseado na função de corte simétrico definida por 2.4. Como a função não apresenta parâmetros, a interface de dados utilizada apenas consiste de uma entrada e uma saída de áudio. O código 3.4 apresenta a rotina principal do *plugin* implementado, com interface de dados definida pelos parâmetros de entrada da função.

Código 3.4: Rotina principal do *overdrive* sem parâmetros

```
def od (inp, output):
  theta = 1.0/3.0

  # Aplicacao do mapeamento
  for i in range(len(inp)):
    if abs(inp[i]) > 2*theta:
      if inp[i] > 0:
        output[i] = 1
      else:
        output[i] = -1
    elif abs(inp[i]) < theta:
      output[i] = 2*inp[i]
    else: # abs(inp[i]) >= theta
      if inp[i] > 0:
        output[i] = (3 - pow((2 - inp[i]*3), 2))/3.0
      else:
        output[i] = -(3 - pow((2 + inp[i]*3), 2))/3.0
```

Pedais de *overdrive* normalmente possuem um parâmetro de ganho que permite a regulagem da intensidade do efeito, saindo de um mapeamento completamente linear até um outro que gera uma onda aproximadamente quadrada. Uma modelagem de corte simétrico que contém parâmetros de regulagem foi apresentada pelo sistema linear 2.5, cujo a resolução retorna os coeficientes das expressões da função de mapeamento. Para a implementação escolhemos o valor $b = 0$, pois para entrada nula queremos saída nula. A interface de dados utilizada consiste de entrada e saída de áudio, além de dois parâmetros th e th2 que estão entre 0 e 1 e determinam os valores de θ_1 e θ_2 presentes no sistema linear. O código 3.5 apresenta a rotina principal do *plugin*, que possui interface de dados definida pelos parâmetros de entrada da função.

Código 3.5: Rotina principal do *overdrive* com dois parâmetros

```
def od2params (inp, output, th1, th2):
  # Calculo dos coeficientes para b = 0
  b = 0
```

```
  e = (pow(th1, 2) - 2*b*(th2 - pow(th2, 2)/2)) / (pow(th1, 2) - 2*(th2 - pow(th2, 2)/2))
  d = 2*(b - e)/pow(th1, 2)
  c = -d/2
  a = d + 2*th1*c

  # Aplicacao do mapeamento
  for i in range(len(inp)):
    if abs(inp[i]) > th2:
      if inp[i] > 0:
        output[i] = 1
      else:
        output[i] = -1
    elif abs(inp[i]) < th:
      output[i] = a*inp[i] + b
    else: # abs(inp[i]) >= theta
      if inp[i] > 0:
        output[i] = e + (d * inp[i]) + (c * pow(inp[i], 2))
      else:
        output[i] = -(e - (d * inp[i]) + (c * pow(inp[i], 2)))
```

3.2.2 *Fuzz*

O *fuzz* é um efeito não-linear que pode ser implementado com a técnica de *waveshaping* apresentada em 2.3.1. Sua equação de transferência pode ser definida por uma equação não-linear como a apresentada em 3.1 [Zölzer(2011)].

$$f(x) = \frac{x}{|x|}\left(1 - e^{x^2/|x|}\right) \tag{3.1}$$

Na interface de dados do *plugin* implementado, além da entrada e saída de áudio, foram incluídos os parâmetros *mix* e *gain*. O *mix* deve estar no intervalo de 0 a 1 e define o peso de cada um dos sinais na saída. Essa é uma forma de obter resultados intermediários entre o sinal de áudio original e o sinal processado pela função de transferência. Já o *gain* é um parâmetro que nos levará a valores maiores no exponencial dado pela equação de transferência, o que faz com que a característica não-linear do sinal de saída seja mais intensa. O código 3.6 apresenta a rotina principal do *plugin*, que possui interface de dados definida pelos parâmetros de entrada da função.

Código 3.6: Rotina principal do *fuzz*

```
from math import exp

def sign(x): return 1 if x >= 0 else -1

def fuzz (inp, output, gain, mix):
  # Normalizacao do sinal de entrada e inclusao do ganho dado pelo parametro gain
  norm_signal = [float(sample)*gain/max(inp) for sample in inp]

  # Aplicacao do waveshaping
  shaped_signal = map(lambda x: sign(x)*(1.0 - exp(sign(-x)*x)), norm_signal)

  # Normalizacao do sinal gerado e mixagem entre o sinal com waveshaping e o sinal original
  mixed_signal = [mix*shaped_signal[i]*max(inp)/max(shaped_signal) + (1.0 - mix)*inp[i] \
```

```
                for i in range(len(inp))]

    # Normalizacao do sinal mixado
    norm_signal = [float(sample)*max(inp)/max(mixed_signal) for sample in mixed_signal]

    # Saida
    for i in range(len(norm_signal)):
            output[i] = norm_signal[i]
```

3.2.3 Simulador de tubo

O simulador de tubo é um efeito não-linear que pode ser implementado com a técnica de corte assimétrico apresentada em 2.3.1. Uma proposição para sua equação de transferência foi dada pela equação 2.6, que em sua concepção já possui os parâmetros *dist* e Q.

Além dos parâmetros *dist* e Q e a entrada e saída de áudio, a interface de dados do *plugin* implementado possui os parâmetros *gain*, *mix*, *rh* e *rl*, onde *gain* e *mix* possuem a mesma interpretação dada em 3.2.2, e *rh* e *rl* são coeficientes que se referem respectivamente a filtragem *highpass* e *lowpass*. Por conta dos filtros, a estrutura utilizada também contém valores anteriores de entradas e saídas necessárias para o processo de filtragem. O código 3.7 apresenta a rotina principal do *plugin*, que possui interface de dados definida pelos parâmetros de entrada da função.

Código 3.7: Rotina principal do simulador de tubo

```
def tube (inp, output, gain, q, dist, rh, rl, mix, lastX, lastX2, lastY, lastY2, lastY_LP):
  # Normalizacao do sinal de entrada e inclusao do ganho dado pelo parametro gain
  norm_signal = [float(sample)*gain/max(inp) for sample in inp]

  # Aplicacao do waveshaping dependendo do valor de q
  if q == 0:
    shaped_signal = map(lambda x: x/(1 - exp(-dist*x)), norm_signal)
    for i in range(len(shaped_signal)):
      if norm_signal[i] == q:
        shaped_signal[i] = 1/dist
  else:
        shaped_signal = map(lambda x: (x - q)/(1 - exp(-dist*x - q)) + q/(1 - exp(dist*q)),\
                    norm_signal)
    for i in range(len(shaped_signal)):
      if norm_signal[i] == q:
        shaped_signal[i] = 1/dist + Q/(1 - exp(dist*q));

  # Normalizacao do sinal gerado e mixagem entre o sinal com waveshaping e o sinal original
  mixed_signal = [mix*shaped_signal[i]*max(inp)/max(shaped_signal) + \
                  (1.0 - mix)*inp[i] for i in range(len(inp))]

  # Normalizacao do sinal mixado
  norm_signal = [float(sample)*max(inp)/max(mixed_signal) for sample in mixed_signal]

  # Filtragem HP: y(n) = x(n) - 2*x(n-1) + x(n-2) + 2*rh*y(n-1) - rh*rh*y(n-2)
  for i in range(len(norm_signal)):
        y = norm_signal[i] - 2*lastX + lastX2 + 2*rh*lastY - rh*rh*lastY2
        lastX2 = lastX
        lastX = norm_signal[i]
        lastY2 = lastY
```

```
        lastY = y
        norm_signal[i] = y

    # Filtragem LP: y(n) = (1-rl)*x(n) + rl*y(n-1)
    for i in range(len(norm_signal)):
        y = (1 - rl)*norm_signal[i] + rl*lastY_LP
        lastY_LP = y
        output[i] = y
```

3.2.4 *Pitch shifting*

O *pitch shifting* é um efeito não-linear que pode ser implementado com a técnica apresentada em 2.3.3. A técnica consiste basicamente de uma contração ou expansão da duração do sinal de entrada de acordo com um fator α pelo algoritmo SOLA, seguido de uma reamostragem, também proporcional a α, do sinal gerado, causando uma transposição das frequências parciais presentes no sinal de entrada de acordo com o fator α.

Observemos porém que o algoritmo SOLA não é implementável em tempo real pois, dado um bloco a ser processado, não temos como estimar quanta memória é necessária para armazenar o sinal gerado no caso de $\alpha < 1$, da mesma forma que não sabemos quanto devemos atrasar a resposta do *plugin* até termos conteúdo suficiente para gerar uma saída condizente no caso de $\alpha > 1$. Apesar disso, o *pitch shifting* é implementável em tempo real porque as dimensões dos blocos de entrada e saída são iguais.

A primeira tentativa de implementação do efeito consistiu de fazer o processo descrito em 2.3.3 para cada bloco de entrada. Localmente o efeito funcionava, pois cada bloco tinha suas parciais de frequência transpostas de acordo com α. Porém foi observado que as saídas geravam descontinuidades nas transições entre os blocos. Uma ideia para resolver esse problema foi introduzir um *delay* de um bloco na resposta do *plugin* para que a cada chamada da função *run* tivéssemos o conteúdo do bloco atual e do bloco anterior. Com isso, geramos a saída referente a dois blocos, encontrando a parte desse sinal que mais se assemelha ao final da saída anterior e fazendo o processo de sobreposição e soma para suavizar o começo de cada bloco de saída em relação ao bloco de saída anterior. Note que isso é compatível com a hipótese de que $\alpha \leq 2$, de onde no máximo o conteúdo relativo a 2 blocos será usado na produção da saída de 1 bloco. O código 3.8 apresenta a rotina principal do *plugin* adaptado, que possui interface de dados definida pelos parâmetros de entrada da função.

Código 3.8: Rotina principal do *pitch shifting*

```
from math import ceil, floor

def ps (inp, output, alpha, last_inp, last_L_out, is_first, last_L):
  Sa = 128, N = 512
  if is_first:
    last_inp = inp
    last_L_out = last_L*[0]
    is_first = False
    last_L = round(Sa*alpha/2)
    output = len(inp)*[0]
```

```
    return

  data_size = 2*len(inp)
 M = ceil(data_size/Sa)
 Ss = round(Sa*alpha)
 L = round(Sa*alpha/2)
 resLen = N
 result = last_inp + inp

 # Time Stretching
 for i in range(M):
    grain = inp[i*Sa:(i+1)*Sa]
    overlap = result[i*Ss:i*Ss + L]
    max_index = maxSimIndex(grain[0:L], overlap) # Retorna o indice de overlap que melhor
                                                 # se encaixa com o comeco de grainL
    cut = i*Ss + max_index
    tail = result[cut:resLen]
    # Sebreposicao e soma
    tail[0:resLen - cut] = [tail[j - cut] * (1 - (j - cut)/(resLen - cut)) + grain[j \
                           - cut] * (j - cut)/(resLen - cut) for j in range(cut, resLen)]
    result = result + tail[0:resLen - cut] + grain[resLen - cut:N]
    resLen = cut + N

 # Reamostragem
 scaled_size = floor(data_size * alpha)
 x = [i * scaled_size / data_size for i in range(data_size)]
 for i in range(data_size):
    mix = x[i] - floor(x[i])
    tail[i] = result[int(floor(x[i]))] * (1.0 - mix)
    tail[i] = tail[i] + result[int(floor(x[i]) + 1)] * mix
 max_index = maxSimIndex(last_L_out, tail[N - L:N]) # Retorna o indice de last_L_out que
                                                    # melhor se encaixa com o meio de tail
 # Sebreposicao e soma
 tail[max_index:max_index + last_L] = [tail[j - cut] * (j/(last_L - 1)) + last_L_out[j] \
                                      * (1 - j / (last_L - 1)) for j in range(last_L)]
 last_inp = inp
 last_L_out = tail[len(inp) + max_index:len(inp) + max_index + L]
 last_L = L

 # Saida
 output = tail[max_index:max_index + len(inp)]
```

3.2.5 *Wah-wah*

O *wah-wah* é um efeito linear que pode ser implementado através da mixagem do sinal de entrada com o sinal de saída de um filtro *bandpass* de largura de banda estreita e frequência central parametrizada, como apresentamos em 2.2.3. Além da entrada e saída de áudio, podemos ter na interface de dados: a largura de banda do filtro, um controle com valor no intervalo entre a mínima e máxima frequência de corte aceitas, e um parâmetro *mix*.

O código 3.9 é uma implementação direta do diagrama de bloco da imagem 2.12, sendo $y_{ap}(n)$ a equação do filtro *allpass* de segunda ordem dada pela equação 2.3, o filtro *bandpass* possui equação de filtro dada por:

$$y(n) = 0.5 * (x(n) - y_{ap}(n)).$$

Código 3.9: Rotina principal do *wah-wah*

```
from math import cos, pi, tan

def wah (inp, output, fs, fb, fc, mix, lastX, lastX2, lastY, lastY2):
  # Definicao dos parametros do filtro
  d = -cos(2*pi*fc/fs)
  c = (tan(pi*fb/fs) - 1)/(tan(2*pi*fb/fs) + 1)

  # Filtragem
  for i in range(len(inp)):
    y_ap = -c*inp[i] + d*(1 - c)*lastX + lastX2 - d*(1 - c)*lastY + c*lastY2
    y = 0.5 * (inp[i] - y_ap)
    lastX2 = lastX
    lastX = inp[i]
    lastY2 = lastY
    lastY = y_ap
    output[i] = inp[i]*(1.0 - mix) + y*mix
```

Uma outra possibilidade é o efeito chamado *auto-wah*, onde variamos a frequência central de acordo com a amplitude máxima do bloco de entrada. Isto é, amplitude máxima baixa tem o mesmo efeito que o pedal fechado para o *wah-wah* comum, enquanto amplitude máxima alta tem o mesmo efeito que o pedal aberto. Nesse caso podemos abrir mão do parâmetro de controle da estrutura anterior, pois ele é dado implicitamente pela intensidade do ataque das notas no instrumento. O código 3.10 é uma implementação do esquema explicado, com os parâmetros adicionais: *lastFc*, que armazena qual o último valor de frequência central, para variarmos linearmente; e *maxAt*, que nos diz qual valor de amplitude deve ser considerado como máximo.

Código 3.10: Rotina principal do *auto-wah*

```
from math import cos, pi, tan

def autowah (inp, output, fs, fb, fc, maxAt, last_fc, mix, lastX, lastX2, lastY, lastY2):
  maximum = 0.10
  minimum = 0.01

  # Encontra a frequencia central conforme a maxima amplitude da entrada
  max_amp = max(inp)
  if max_amp > maxAt:
    max_amp = maxAt
  new_fc = ((maximum - minimum)*(max_amp/maxAt) + minimum)*fs

  for i in range(len(inp)):
    # Definicao dos parametros do filtro
    current_fc = ((new_fc - last_fc)/len(inp))*i + last_fc
    d = -cos(2*pi*fc/fs)
    c = (tan(pi*fb/fs) - 1)/(tan(2*pi*fb/fs) + 1)

    # Filtragem
    y_ap = -c*inp[i] + d*(1 - c)*lastX + lastX2 - d*(1 - c)*lastY + c*lastY2
    y = 0.5 * (inp[i] - y_ap)
    lastX2 = lastX
    lastX = inp[i]
    lastY2 = lastY
    lastY = y_ap
    output[i] = inp[i]*(1.0 - mix) + y*mix
```

3.3 Experimentos

Para verificar se os *plugins* implementados têm características que estão de acordo com os modelos propostos no capítulo 2, foram feitos experimentos que permitem a comparação visual entre formas de onda e espectrogramas das entradas e saídas dos *plugins*. Espectrogramas nos dizem qual o comportamento espectral de um sinal de áudio ao longo do tempo e consistem de um gráfico colorido da frequência pelo tempo, onde as cores definem uma escala de intensidade das componentes em frequências presentes no sinal. As imagens que serão apresentadas foram geradas com o auxílio do *software* Sonic Visualiser [Son()].

3.3.1 Efeitos não-lineares

Para efeitos não-lineares os experimentos foram realizados com uma senoide de 1 kHz, que possui espectrograma dado pela figura 3.1. Escolhemos uma senoide porque ela faz com que a característica não-linear, que cria componentes em frequência que não estão presentes no sinal de entrada, fique mais evidente quando analisamos os espectrogramas gerados.

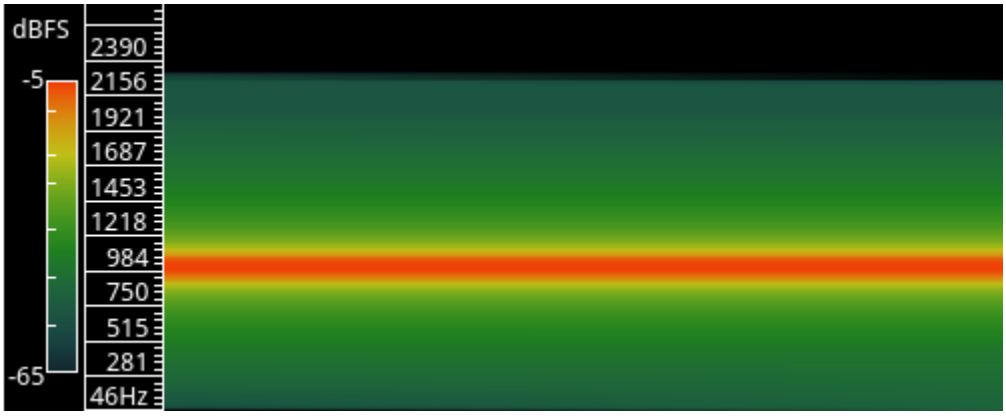

Figura 3.1: Espectrograma da senoide utilizada para os experimentos

Overdrive sem parâmetros

Na figura 3.2 temos a comparação entre as formas de onda da entrada e da saída do *plugin*, com a entrada em preto e a saída em laranja. Podemos observar que o sinal gerado tem amplitude maior em relação à entrada, além de apresentar o corte simétrico desejado, sendo observável a saída de valor constante para entradas cujo módulo é maior que 0.6. O espectrograma do sinal de saída é dado pela figura 3.3, notamos que o efeito insere frequências da forma $f(k) = 1 + 2k$ kHz para $k = 0, 1, ..., 10$, o que está de acordo com o espectrograma para *symmetrical soft clipping* apresentado na figura 2.15.

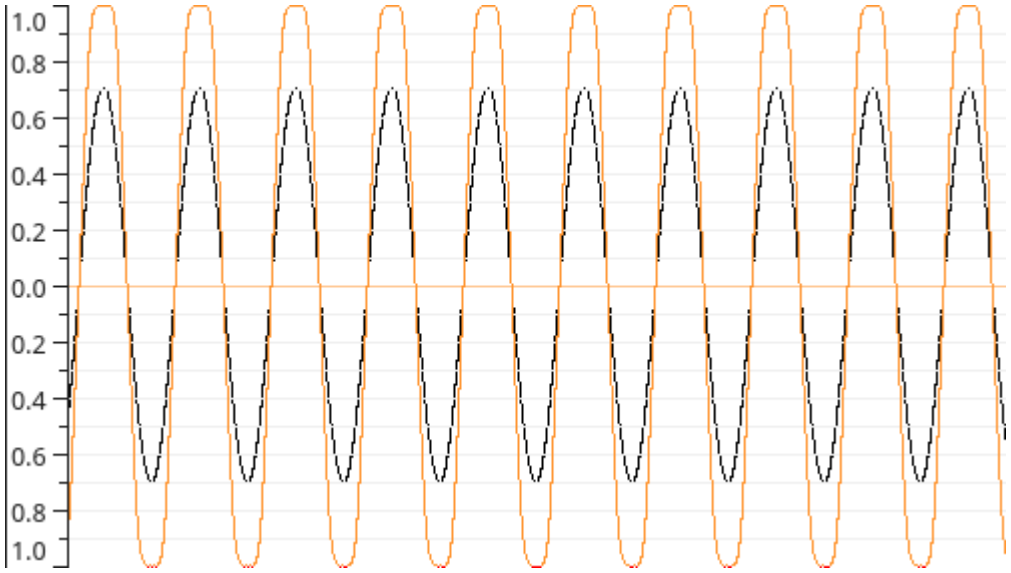

Figura 3.2: Forma de onda para o *overdrive* sem parâmetros

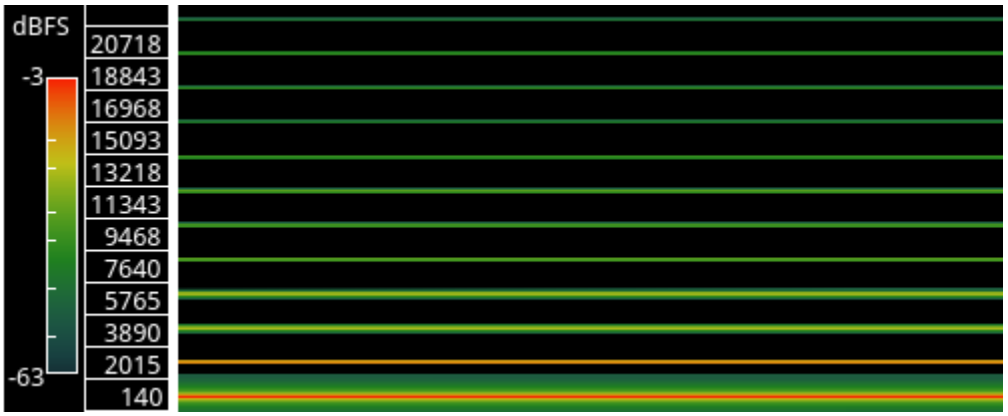

Figura 3.3: Espectrograma para o *overdrive* sem parâmetros

Fuzz

Na figura 3.4 temos a comparação entre as formas de onda da entrada e da saída do *plugin* para $gain = 20$. O espectrograma do sinal de saída é dado pela figura 3.5, notamos que ocorre a criação de componentes de frequência não presentes no sinal original de uma forma mais intensa que para o efeito *overdrive*, o que é induzido pela forma de onda aproximadamente quadrada.

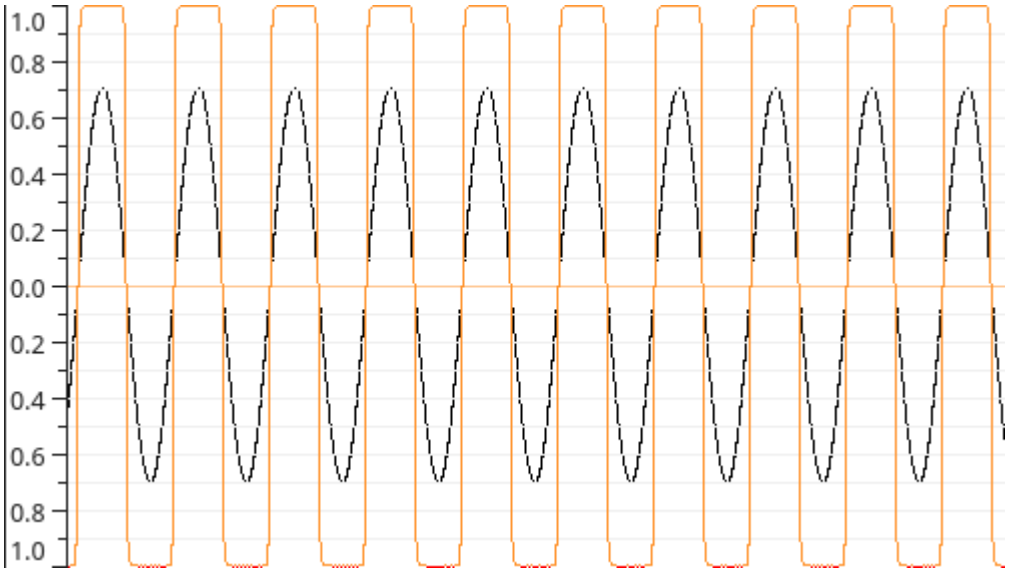

Figura 3.4: Forma de onda para o *fuzz*

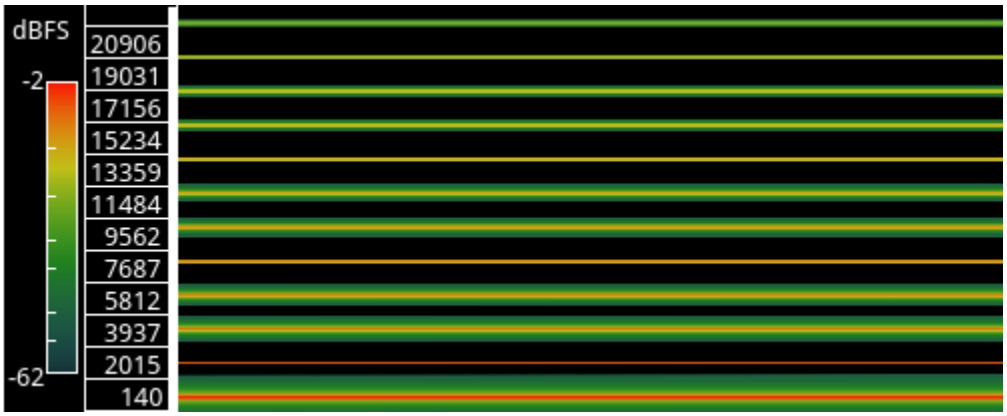

Figura 3.5: Espectrograma para o *fuzz*

Overdrive com dois parâmetros

Na figura 3.6 temos a comparação entre as formas de onda da entrada e da saída do *plugin* para $\theta_1 = 0.5$ e $\theta_2 = 0.75$. Podemos observar que o sinal gerado tem amplitude maior em relação à entrada, chegando a valores próximos a ± 1. O espectrograma do sinal é dado pela figura 3.7, notamos que o efeito insere componentes de frequência de forma similar ao *overdrive* sem parâmetros.

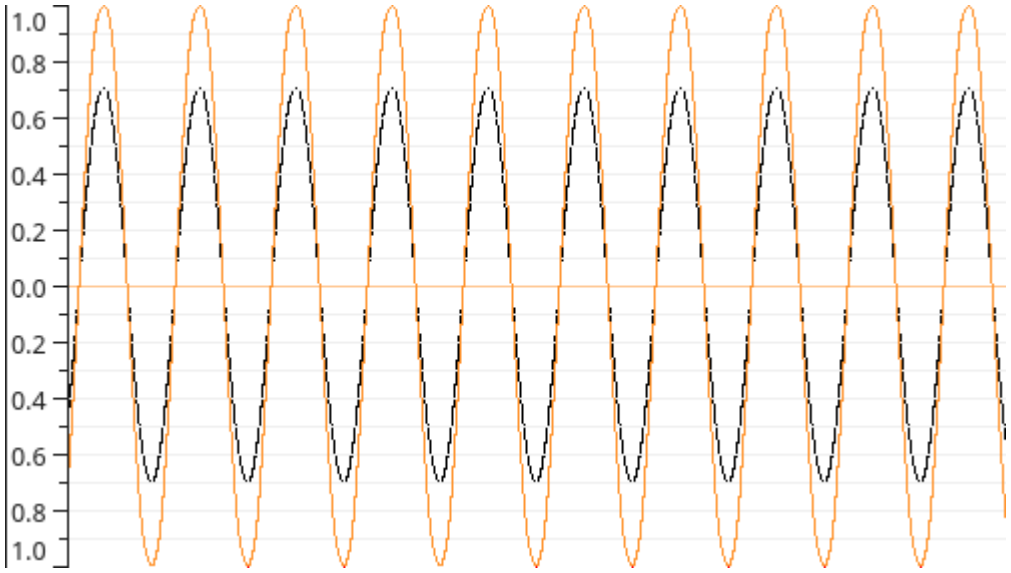

Figura 3.6: Forma de onda para o *overdrive* com parâmetros $\theta_1 = 0.5$ e $\theta_2 = 0.75$

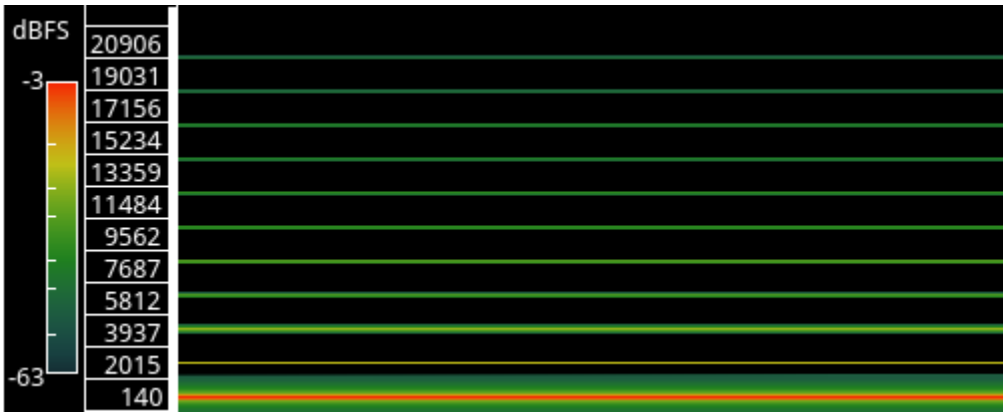

Figura 3.7: Espectrograma para o *overdrive* com parâmetros $\theta_1 = 0.5$ e $\theta_2 = 0.75$

Na figura 3.8 temos a comparação entre as formas de onda da entrada e da saída do *plugin* para $\theta_1 = 0.01$ e $\theta_2 = 0.02$. Podemos observar que o sinal gerado tem forma de onda aproximadamente quadrada. O espectrograma do sinal é dado pela figura 3.9, notamos que o efeito insere componentes de frequência de forma similar ao *fuzz*.

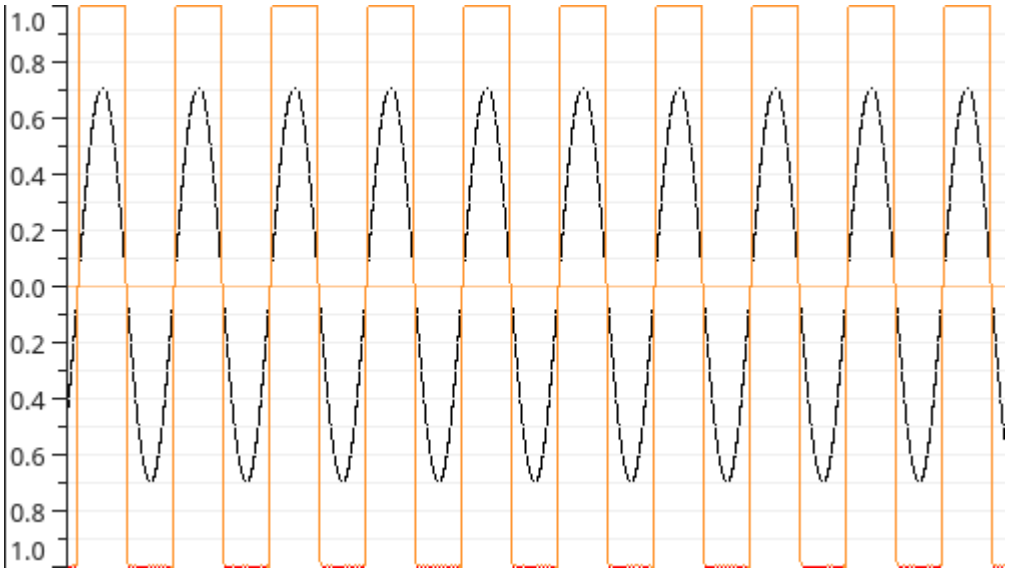

Figura 3.8: Forma de onda para o *overdrive* com parâmetros $\theta_1 = 0.01$ e $\theta_2 = 0.02$

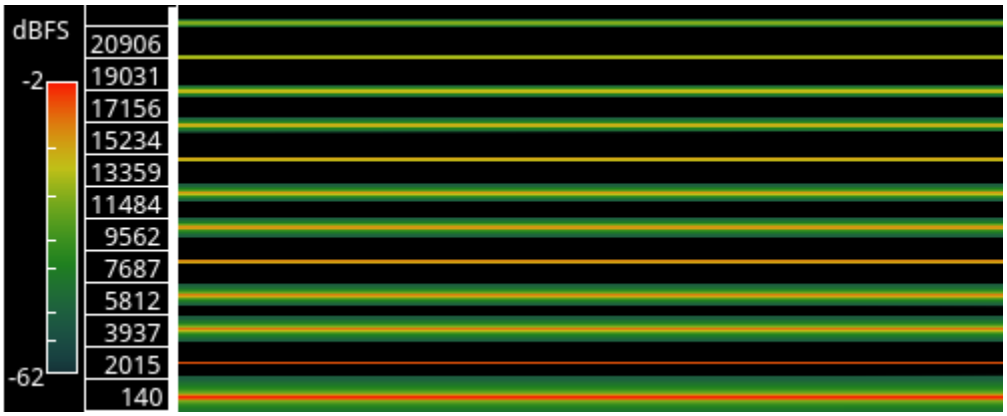

Figura 3.9: Espectrograma para o *overdrive* com parâmetros $\theta_1 = 0.01$ e $\theta_2 = 0.02$

Simulador de tubo

Na figura 3.10 temos a forma de onda da saída do *plugin* para uma senoide de 1 kHz, com os parâmetros $gain = 4$, $q = 0.8$ e $dist = 3$. Nesse caso escondemos a forma de onda da entrada para não poluir o gráfico. Podemos observar que o sinal gerado apresenta diferenças quando comparamos suas parcelas positivas e negativas, como esperado de uma implementação de corte assimétrico. O espectrograma do sinal é dado pela figura 3.11, notamos que o efeito insere praticamente todas componentes de frequência múltiplas de 1 kHz que estão abaixo de 17 kHz e cujas intensidades diminuem com a frequência.

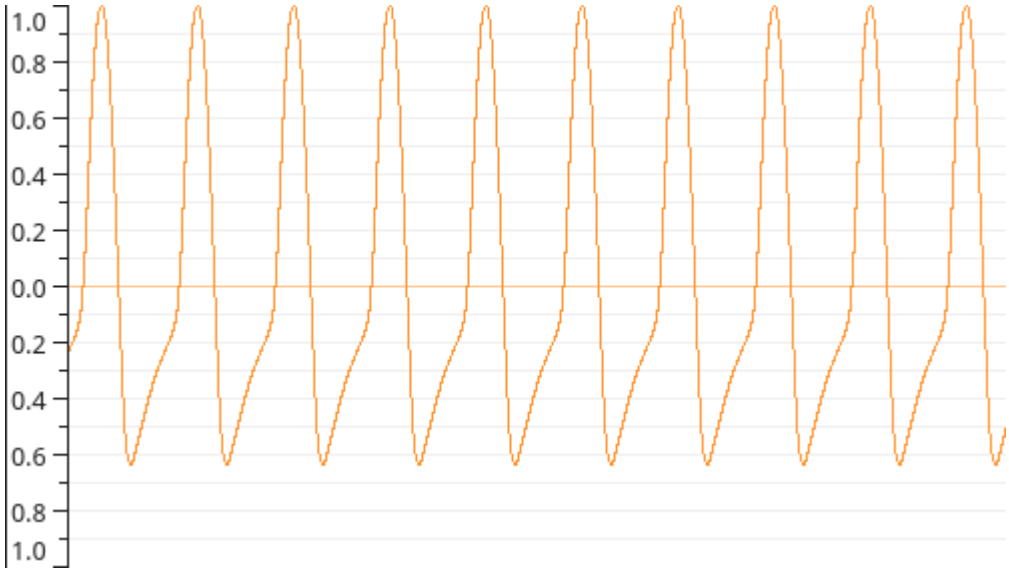

Figura 3.10: Forma de onda para o simulador de tubo com parâmetros $gain = 4$, $q = 0.8$ e $dist = 3$

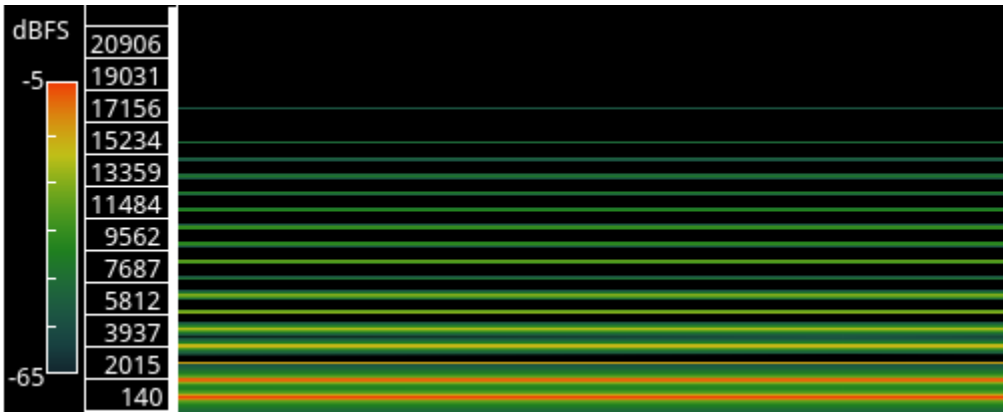

Figura 3.11: Espectrograma para o simulador de tubo com parâmetros $gain = 4$, $q = 0.8$ e $dist = 3$

Pitch shifting

Na figura 3.12 temos o espectrograma da saída do *plugin* para $\alpha = 0.5$ e podemos observar que as componentes de frequência mais intensas foram as que estão próximas da frequência 500 Hz. Já na figura 3.13 temos o espectrograma da saída do *plugin* para $\alpha = 2.0$ e podemos observar que as componentes de frequência mais intensas foram as que estão próximas da frequência 2000 Hz.

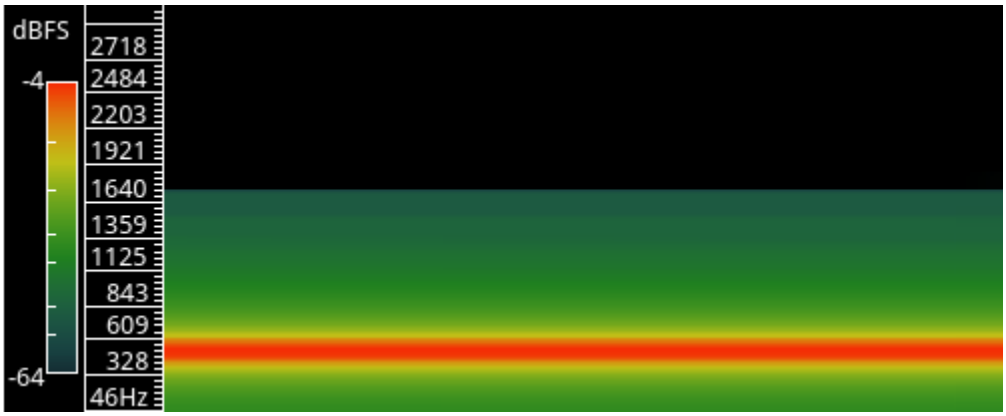

Figura 3.12: Espectrograma para o *pitch shifting* com $\alpha = 0.5$

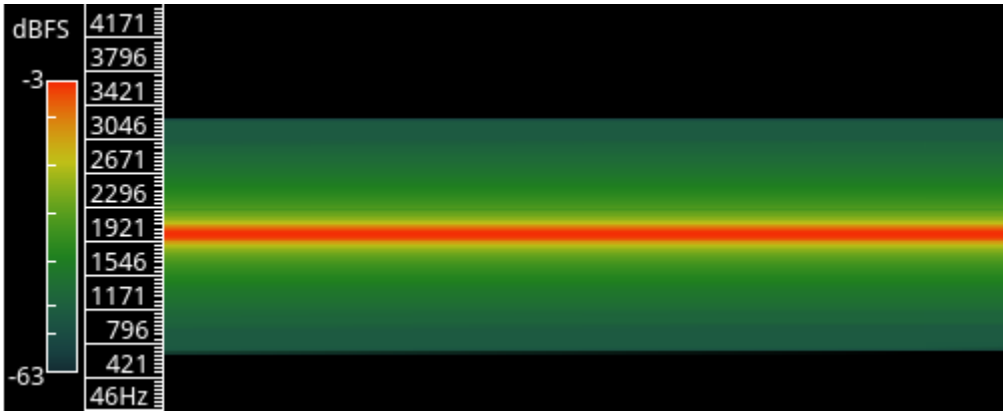

Figura 3.13: Espectrograma para o *pitch shifting* com $\alpha = 2.0$

3.3.2 Efeitos lineares

Para efeitos lineares os experimentos foram realizados com o que chamamos de ruído branco, que possui espectrograma dado pela figura 3.14. O ruído branco é uma boa forma de verificar a saída de filtros lineares porque possui componentes em todas as frequências de seu espectro, como podemos visualizar na figura.

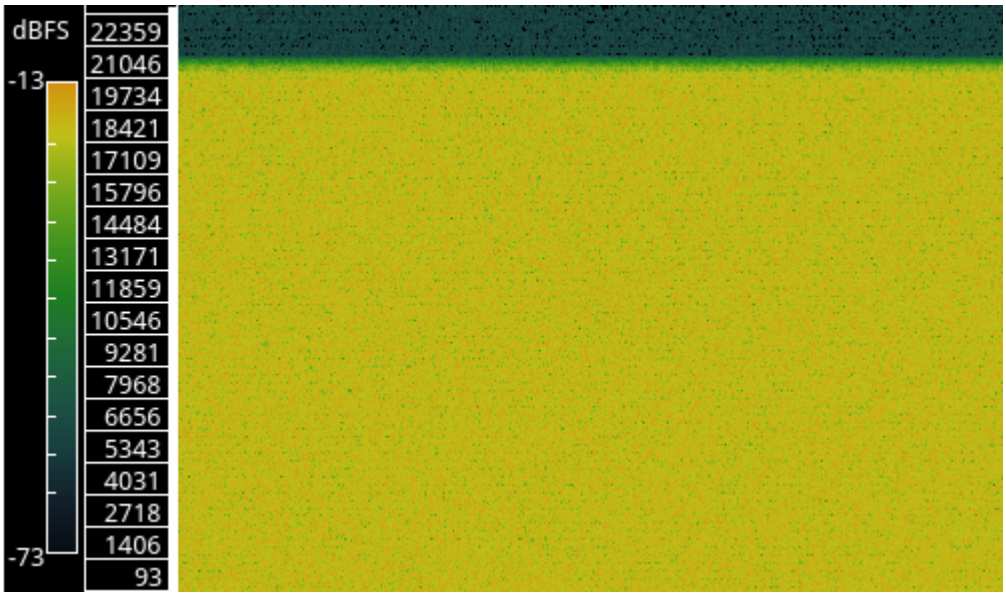

Figura 3.14: Espectrograma do ruído branco

Wah-wah

Na figura 3.15 temos o espectrograma da saída do *plugin* para frequência central e largura de banda iguais a 480 Hz. Podemos observar que as componentes em frequências mais próximas de 480 Hz foram enfatizadas, enquanto as restantes se enfraqueceram. Já na figura 3.16 temos o espectrograma da saída do *plugin* para frequência central igual a 9600 Hz e largura de banda igual a 480 Hz. Podemos observar o mesmo efeito para as componentes em frequências próximas de 9600 Hz.

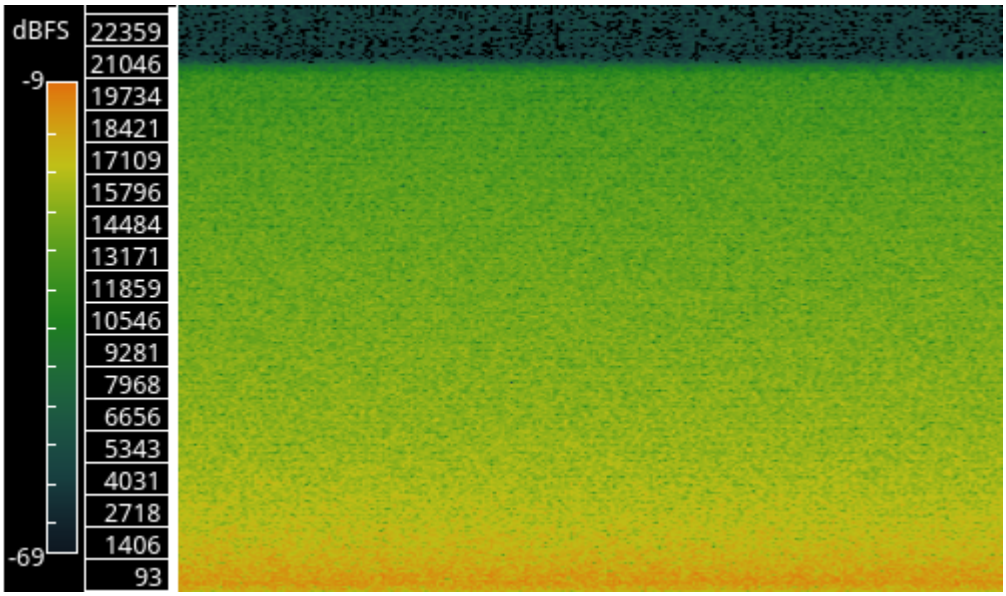

Figura 3.15: Espectrograma para o *wah-wah* com $f_c = 480$ e $f_b = 480$

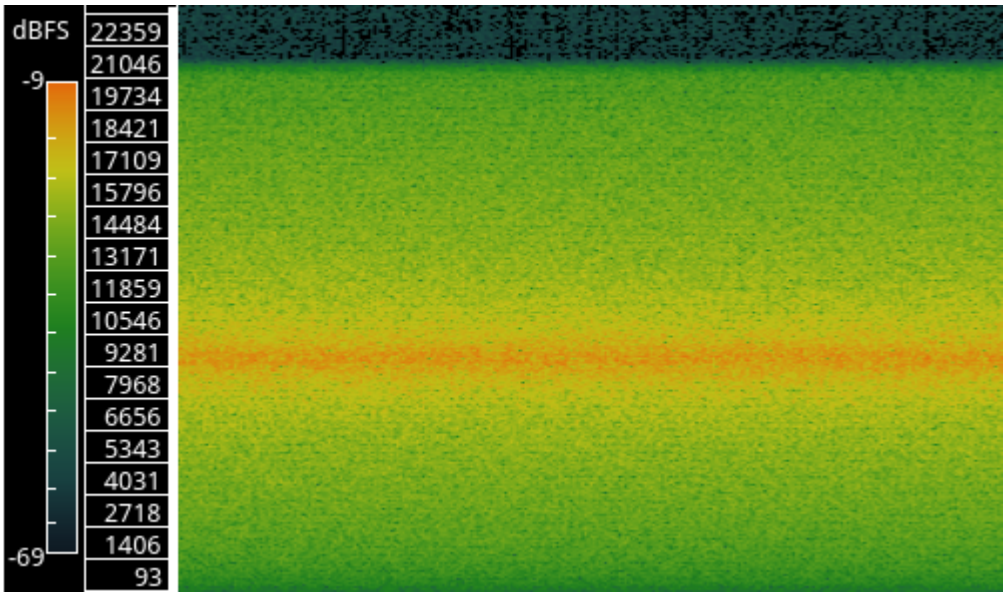

Figura 3.16: Espectrograma para o *wah-wah* com $f_c = 9600$ e $f_b = 480$

Capítulo 4

Conclusão

O trabalho consistiu do estudo de conceitos de computação musical direcionados à concepção de famílias de efeitos clássicos de áudio, bem como suas implementações para ambientes em tempo real. Dentro do conteúdo abordado, foi possível adquirir maiores conhecimentos sobre representações de áudio discretizadas, concepções de filtros digitais e de técnicas não-lineares, assim como seus reflexos nas representações consideradas, em especial o espectro.

As representações de áudio no computador são dadas através de sinais digitais, gerados a partir de um sistema elétrico chamado ADC (*Analog to Digital Converter*) que leva em conta o Teorema de Nyquist, com seus respectivos espectros podendo ser gerados pela transformada rápida de Fourier.

Em relação a filtros lineares, vimos que existe uma categoria de filtros que podem ser descritos através de fórmulas matemáticas. Essas fórmulas consistem de uma fração de polinômios cujas raízes determinam as respostas de magnitude e fase do filtro, isso nos diz o comportamento da saída de todas as componentes de frequência para qualquer sinal de entrada que possua alguma das componentes. Essas considerações foram importantes para a implementação dos efeitos *wah-wah* e *auto-wah*.

Para técnicas não-lineares, focamos na implementação de efeitos por meio das técnicas de *waveshaping* e *pitch shifting*. No *waveshaping* temos um processo de mapeamento da amplitude do sinal de entrada a uma nova amplitude para o sinal de saída dada por uma função não-linear que pode ser definida por uma ou mais expressões analíticas. Por meio dessa técnica foi possível implementar os efeitos *overdrive*, *fuzz* e um simulador de tubo, além da proposição de um sistema linear que, dados dois parâmetros de entrada, podemos ter um único efeito que transita do *overdrive* ao *fuzz* de acordo com os valores dos parâmetros. No *pitch shifting* temos um processo que nos permite alterar a frequência do sinal de entrada sem que seja alterada a sua duração. Vimos que sua implementação para tempo real necessitou de modificações em relação ao algoritmo estudado na bibliografia, pois em ambientes de tempo real não temos acesso a todo o arquivo de áudio a ser processado na etapa de *time stretching*.

Além do conteúdo teórico, foi necessário um estudo mais técnico para a implementação de

plugins no padrão LV2. Vimos que é necessário definir uma interface entre o *plugin* e o *host*, contendo os dados compartilhados e os métodos chamados pelo *host*. Na etapa experimental validamos os *plugins* por meio de comparações entre os gráficos gerados por suas saídas e os gráficos que foram expostos no capítulo de fundamentação teórica.

Portanto, podemos concluir que boa parte dos objetivos iniciais do trabalho foram atingidos. Pôde-se aprender alguns dos fundamentos de computação musical, tendo uma introdução em técnicas de processamento de sinais. Esse estudo também gerou uma compreensão maior sobre quais mudanças ou características são inseridas no áudio quando consideramos a utilização de alguns dos efeitos implementados durante o trabalho, e as diferenças entre efeitos lineares e não-lineares. Pôde-se aprender quais são as limitações introduzidas pelo processamento de áudio em blocos (ou em tempo real), e como é feita a implementação de *plugins* em padrão LV2.

Referências Bibliográficas

[Ard()] **Ard()** Ardour. https://ardour.org/. Citado na pág. 2, 27

[Aud()] **Aud()** Audacity. http://www.audacityteam.org/. Citado na pág. 27

[CAL()] **CAL()** CALF. http://calf-studio-gear.org/. Citado na pág. 2

[Cub()] **Cub()** Cubase. https://www.steinberg.net/en/products/cubase/start.html. Citado na pág. 27

[LV2()] **LV2()** LV2. http://lv2plug.in/book/. Citado na pág. 2, 27

[Pro()] **Pro()** ProTools. http://www.avid.com/pro-tools. Citado na pág. 27

[REA()] **REA()** REAPER. http://www.reaper.fm/. Citado na pág. 27

[Son()] **Son()** SonicVisualiser. http://sonicvisualiser.org/. Citado na pág. 2, 37

[Tur()] **Tur()** Turtle. http://www.w3.org/TeamSubmission/turtle/. Citado na pág. 2, 27

[Moore(1990)] **Moore(1990)** F. Richard Moore. *Elements of Computer Music*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-252552-6. Citado na pág. 1, 3, 4, 5, 7, 8, 9, 12

[Puckette(2007)] **Puckette(2007)** Miller Puckette. *The Theory and Technique of Electronic Music*. World Scientific Publishing Company. ISBN 9789812700773. Citado na pág. 1, 19

[Zölzer(2011)] **Zölzer(2011)** Udo Zölzer. *DAFX: Digital Audio Effects: Second Edition*. John Wiley & Sons, Ltd, Chichester, UK. ISBN 9780470665992. doi: 10.1002/9781119991298. URL http://doi.wiley.com/10.1002/9781119991298. Citado na pág. 1, 6, 11, 14, 15, 16, 17, 18, 20, 22, 23, 24, 25, 31, 32