Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Simulações Digitais de Amplificadores de Guitarra

Pedro Henrique Medeiros Leal

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Márcio da Costa Pereira Brandão

Brasília
2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof. Dr. Márcio da Costa Pereira Brandão (Orientador) — CIC/UnB
Prof. Dr. Camilo Chang Dórea — CIC/UnB
Prof. Dr. Alexandre Zaghetto — CIC/UnB

CIP — Catalogação Internacional na Publicação

Leal, Pedro Henrique Medeiros.

Simulações Digitais de Amplificadores de Guitarra / Pedro Henrique Medeiros Leal. Brasília : UnB, 2015.
187 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. amplificador, 2. simulação, 3. WDF

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Simulações Digitais de Amplificadores de Guitarra

Pedro Henrique Medeiros Leal

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Márcio da Costa Pereira Brandão (Orientador)
CIC/UnB

Prof. Dr. Camilo Chang Dórea
CIC/UnB

Prof. Dr. Alexandre Zaghetto
CIC/UnB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, 09 de dezembro de 2015

Dedicatória

A todos aqueles apaixonados pela música e pela tecnologia.

Agradecimentos

Agradeço principalmente a Deus pela força que me deu ao longo de mais uma etapa da vida, bem como todos aqueles que me apoiaram e ajudaram ao longo do curso. A minha família, que me deu o suporte necessário, e além do necessário, e que sempre esteve perto em todos os momentos. Amigos e colegas de curso, e especialmente à Keith, pelo amor e paciência que teve durante os anos. Finalmente, e com grande carinho, ao Prof. Dr. Márcio Brandão, que com grande sabedoria me orientou ao longo deste trabalho.

Resumo

Este trabalho trata de métodos de simulação digital de circuitos analógicos aplicados à amplificadores de guitarras valvulados. São utilizados os métodos de *Wave Digital Filters* (WDF) para a simulação dos circuitos não-lineares, e o método da análise nodal para a simulação dos circuitos lineares de um amplificador valvulado. A análise comparativa das simulações é apresentada após as implementações dos métodos citados.

Palavras-chave: amplificador, simulação, WDF

Abstract

This work deals with methods for the digital simulation of analog circuits applied to vacuum-tube guitar amplifiers. The wave Digital Filter (WDF) method is used for the non-linear circuits simulation, and the nodal analysis method is used for the simulation of the linear circuits of a vacuum-tube amplifier. The comparative analysis of the simulations is presented after the implementation of the cited methods.

Keywords: amplifier, simulation, WDF

Sumário

Lista de Figuras

Capítulo 1

Introdução

Amplificadores de guitarras elétricas têm um papel importantíssimo na geração de timbres. Além de seu som ser muito agradável ao ouvido humano, cada um deles possui caracteríscas sonoras próprias, de forma que podemos associá-los a músicos específicos.

Infelizmente, esses amplificadores são caros, além de possuir grande porte, uma vez que os mais desejados pelos guitarristas utilizam válvulas (*vacuum-tubes*, ou "tubos de vácuo"). Apesar de várias tentativas de simulação desses amplificadores terem sido realizadas, visando reduzir o custo necessário para obtenção desses timbres, há uma grande discussão no mundo musical sobre a fidelidade dessas simulações, de forma que o mercado ainda prefere a utilização dos amplificadores físicos.

Assim, faz-se necessário um estudo sobre as possibilidades, na busca pela qualidade de simulações de amplificadores, ou seja, quão fiéis elas podem ser em relação aos amplificadores originais.

1.1 Problema

Para fazer uma comparação correta entre um amplificador de guitarra valvulado, e uma simulação gerada a partir da análise dos seus circuitos internos, é necessário realizar uma análise matemática das respostas de frequência de ambos os objetos, possibilitando assim, o melhor entendimento sobre dispositivos de som digitais e analógicos.

1.2 Hipótese

Será assumido a partir daqui, que é possível gerar uma simulação digital muito próxima ao amplificador original, utilizando as técnicas de discretização modernas.

1.3 Objetivos

Objetivo Geral

Realizar uma análise comparativa entre um amplificador de guitarra valvulado, e o seu equivalente simulado.

Objetivos Específicos

- Realizar a análise matemática do circuito de um ou mais amplificadores de guitarra valvulados.
- A partir das equações geradas, construir uma simulação digital equivalente a análise.
- Gerar amostras de som a partir da simulação gerada.
- Fazer a análise matemática das respostas de frequência das amostras.
- Obter um resultado a partir da comparação entre as duas análises.

1.4 Organização do Trabalho

- Capítulo 2: Uma breve descrição das fases dos amplificadores de guitarra valvulados.
- Capítulo 3: A teoria da análise dos circuitos analógicos, e análise nodal.
- Capítulo 4: Descrição dos sinais e sistemas digitais, e diversos cálculos é métodos necessários para o desenvolvimento deste trabalho.
- Capítulo 5: Uma apresentação do funcionamento físico do diodo e do triodo, acompanhado por uma revisão de diversos métodos para a simulação de não-linearidades. Também são apresentados exemplos de resultados de simulações no Matlab, do diodo e do triodo.
- Capítulo 6: Uma revisão sobre as ferramentas utilizadas para o desenvolvimento do trabalho.
- Capítulo 7: A implementação da fase do *tone stack* de um amplificador de guitarra.
- Capítulo 8: A implementação da fase não-linear de um amplificador de guitarra utilizando o método WDF.
- Capítulo 9: Conclusões sobre os resultados obtidos.

Capítulo 2

Visão geral dos amplificadores valvulados

2.1 Organização

Segundo *Jaromir Macak* e *Jiri Schimmel* [11], um amplificador geral, mostrado na figura 2.1, consiste de uma fase de pré-amplificação, composta por diversas válvulas eletrônicas de dois elementos (diodos) e de três elementos (triodos), com valores de circuitos diferentes, um seguidor de cátodo, e um *tone stack* [22]. A fase de amplificação é composta por um amplificador *push-pull*, um transformador de saída e um mecanismo de realimentação global. As fases mais perceptíveis em um amplificador de guitarra são as

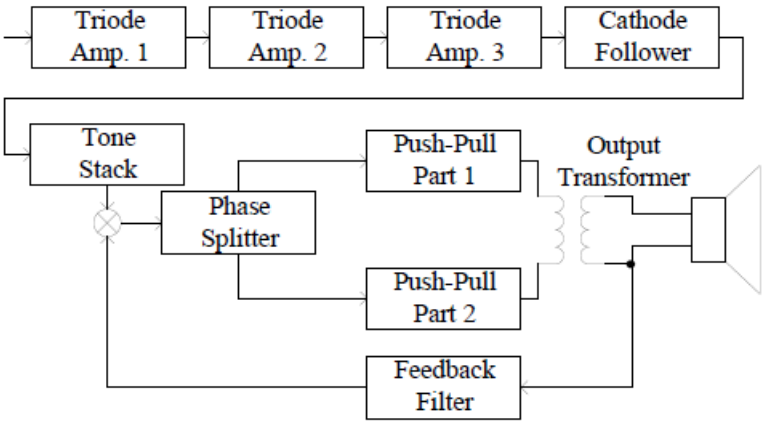

Figura 2.1: A estrutura de um amplificador valvulado [11]

fases de pré-amplificação e a fase do *tone stack*. Elas serão descritas mais a frente neste capítulo, nas seções de amplificação não-linear e *tone stack*.

2.2 Amplificação não-linear

As válvulas de três eletrodos, representadas na figura 2.2, abordadas por Jyri Pakarinen and David T. Yeh, no artigo *A Review of Digital Techniques for Modeling Vacuum-Tube Guitar Amplifiers* [15] foram criadas no início do século XX visando amplificar sinais de baixa voltagem. O primeiro, chamado de cátodo, é aquecido, e dependendo da altura relativa da placa (ou ânodo), que é controlada pela voltagem aplicada ao sistema, uma corrente elétrica flui constantemente entre o cátodo e a placa. O triodo é um tipo de válvula que introduz um terceiro eletrodo, chamado de tela. Ela funciona como uma barreira que limita o fluxo de elétrons para a placa. Se esta barreira é grande o suficiente, ela pode parar o fluxo de elétrons completamente.

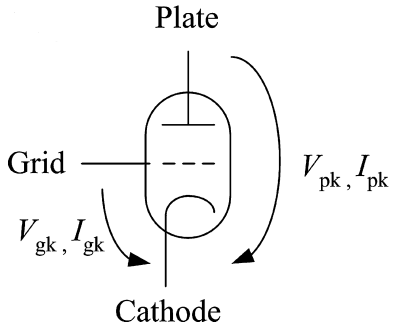

Figura 2.2: Representação de uma válvula eletrônica [15].

A corrente gerada entre o cátodo e a placa é uma função não-linear, sendo alterada pela variação de tensão aplicada à tela. Quando esta tensão é pequena, então a corrente passa sem cortes. Mas em tensões extremas, ela é bastante limitada, resultando em uma não-linearidade, saturando a corrente.

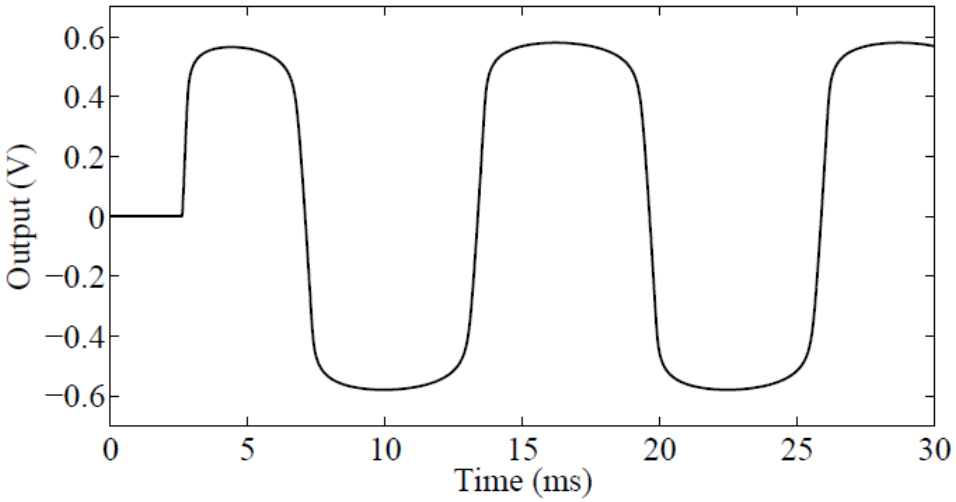

Figura 2.3: Senóide saturada devido a passagem por um diodo.

O efeito perceptível gerado pela saturação da corrente é chamado de distorção, e é procurado por guitarristas, além de estar presente na maioria das músicas modernas. Ele pode ser visualizado na Figura 2.3. A simulação digital de uma válvula é amplamente estudada, devido a sua natureza de grande complexidade, e existem diversos métodos para realizar a tarefa, de forma que os mais recentes visam inserir realismo ao som produzido.

2.3 *Tone Stack*

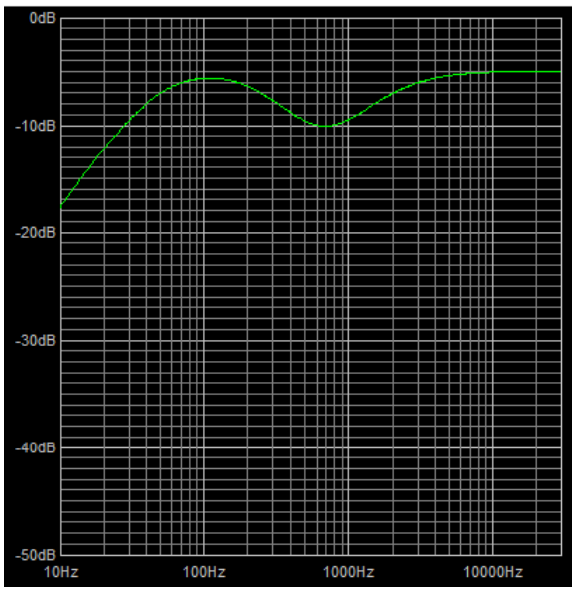

Figura 2.4: Resposta de frequência de um amplificador Marshall, quando os potenciômetros estão posicionados em 0.5 [1].

Após o sinal passar pelas válvulas, a distorção aumenta o número de componentes de frequência do sinal original. produzido pela guitarra. Essas frequências podem ser controladas através de um mecanismo inserido logo após a saída das válvulas, conhecido como *tone stack* [22]. Ele é um filtro que possui controles para alterar ganhos em algumas bandas de frequência, comumente frequências agudas (*treble*, geralmente entre 1000-10000hz), frequências médias (*middle*, geralmente entre 100 e 1000hz), e frequências graves (*low*, geralmente entre 10-100hz).

O filtro digital simula sistemas lineares com grande eficiência e acurácia. Como os circuitos para esse sistema geralmente são lineares, é possível utilizar funções de transferência lineares digitais para descrevê-los. Para obter essas funções, é possível aplicar a transformada de Laplace para análise do circuito, obtendo-se assim a função de transferência linear no domínio analógico, que pode ser convertida ao domínio digital utilizando a transformação bilinear.

O aplicativo *Tone Stack Calculator* [1] mostra bem as variações entre frequências e a não-ortogonalidade entre os potenciômetros para várias marcas de amplificadores, mediante a alteração desses controles, como por exemplo, no gráfico da Figura 2.4.

O efeito perceptível gerado por este filtro dá-se pela ênfase em algumas frequências do som gerado, tornando-o assim, mais grave ou agudo, dependendo das entradas de controle

definidas pelo operador do amplificador.

Capítulo 3

Análise de circuitos

3.1 Introdução

Um circuito é um arranjo de elementos, que possui nós que são conectados a terminais. Existem sete elementos básicos formantes de qualquer circuito. São eles: (1) fonte de voltagem, (2) fonte de corrente, (3) resistor, (4) capacitor, (5) indutor, (6) diodo e (7) transistor. Os elementos possuem terminais de quantidades variadas. Entre cada par de terminal existe uma voltagem e uma corrente passante. A teoria dos circuitos fundamentalmente assume que essas voltagens e correntes seguem duas leis físicas, conhecidas como leis de Kirchoff. A partir dessas leis e das equações que definem os elementos, é possível realizar uma análise matemática completa de todos os circuitos.

Leis de Kirchoff

A soma das voltagens em um caminho fechado é zero (Lei de Kirchoff das Voltagens, LKV), e a soma das correntes que saem de um nó também é zero (Lei de Kirchoff das Correntes, LKC)[19].

Definição dos elementos

Cada elemento possui uma definição matemática que o descreve, e que serão apresentadas a seguir. Tem-se o elemento seguido de sua equação.

Resistor:

$$v = Ri, \tag{3.1}$$

Capacitor:

$$i = C\frac{dv}{dt}, \tag{3.2}$$

Indutor:

$$v = L\frac{di}{dt}, \tag{3.3}$$

Fonte de Voltagem:

$$v(t) = e(t), \tag{3.4}$$

Fonte de Corrente:

$$i(t) = j(t), \tag{3.5}$$

Diodo:

$$I = I_s(e^{\frac{V}{V_t}} - 1). \tag{3.6}$$

Em que v é a tensão em um terminal, e i é a corrente passante no terminal.

3.2 Circuitos lineares e não-lineares

Um elemento é classificado como linear se e somente se a equação que o define é linear, ou seja, a voltagem v e a corrente i, juntas, satisfazem as propriedades de superposição e aditividade na equação, caso contrário, o elemento é não-linear. São elementos lineares: resistor, capacitor, indutor e fonte de voltagem independente. O diodo e o transistor são elementos não-lineares.

Um circuito formado apenas por elementos lineares será necessariamente linear, mas circuitos contendo elementos não-lineares pode ser linear ou não linear.

Equações LKV

Um circuito RLC, que é formado apenas por resistores, capacitores e indutores, possui para cada terminal dos elementos uma voltagem e uma corrente passante. A voltagem é determinada escolhendo-se um nó para ser o terra, e então é atribuída uma voltaem para todos os nós em respeito ao nó terra. A este último, não é atribuída uma voltagem, conforme a figura 3.1.

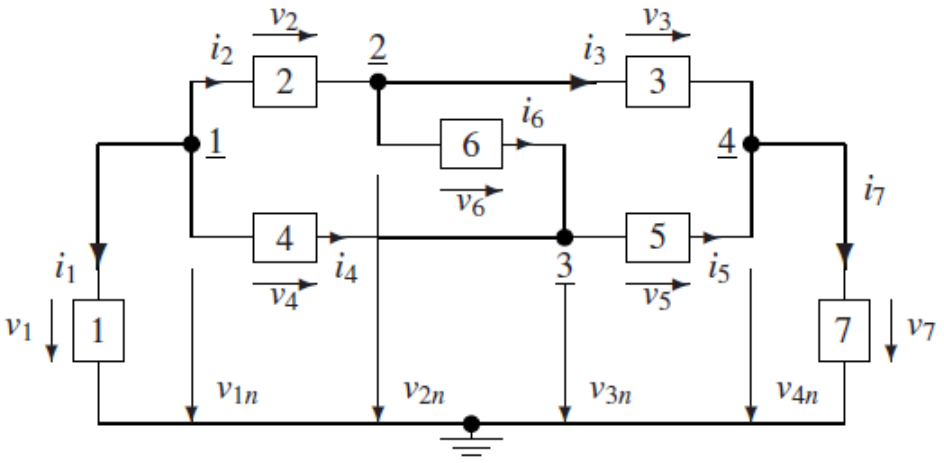

Figura 3.1: Designação das correntes e voltagens em um circuito RLC.[19]

Cada elemento no circuito é conectado a exatamente dois nós. Cada terminal unido aos dois nós conectados ao elemento forma um laço. É possível escrever uma equação para cada laço, de forma que as voltagens nos terminais são descritas em termos dos nós do circuito, exceto o terra, da seguinte forma:

$$v_1 = v_{1n} \tag{3.7}$$

$$v_2 = v_{1n} - v_{2n} \tag{3.8}$$

$$v_3 = v_{2n} - v_{4n} \tag{3.9}$$

$$v_4 = v_{1n} - v_{3n} \tag{3.10}$$

$$v_5 = v_{3n} - v_{4n} \tag{3.11}$$

$$v_6 = v_{2n} - v_{3n} \tag{3.12}$$

$$v_7 = v_{4n} \tag{3.13}$$

Essas equações são suficientes para descrever todo o circuito, uma vez que outras equações podem ser representadas a partir de uma combinação linear delas.

Equações LKC

A segunda lei de Kirchoff assume que a soma das correntes saindo e entrando em um nó é zero, portanto, é possível escrever as equações LKC que descrevem o circuito da Figura 3.1. Admitindo-se que a corrente que sai do terminal tem sinal positivo, e vice-versa, chega-se as seguintes equações:

$$i_1 + i_2 + i_4 = 0 \tag{3.14}$$

$$-i_2 + i_3 + i_6 = 0 \tag{3.15}$$

$$-i_3 - i_5 + i_7 = 0 \tag{3.16}$$

$$-i_4 + i_5 - i_6 = 0 \tag{3.17}$$

$$-i_1 - i_7 = 0 \tag{3.18}$$

É fácil perceber que cada variável de corrente aparece nas equações uma vez com sinal positivo e outra vez com sinal negativo[19]. Isso acontece porque a corrente em cada terminal deve sair de um nó e entrar em outro, de forma que a soma de todas as equações LKC é zero.

3.3 Análise nodal nodificada

A análise nodal modificada surge com o intuito de se encontrar um conjunto de equações que sejam compactas, e aplicáveis a todos os circuitos. Seja o circuito da figura 3.2

As equações geradas pelos nós são as mesmas geradas pelo circuito da figura 3.1, pois

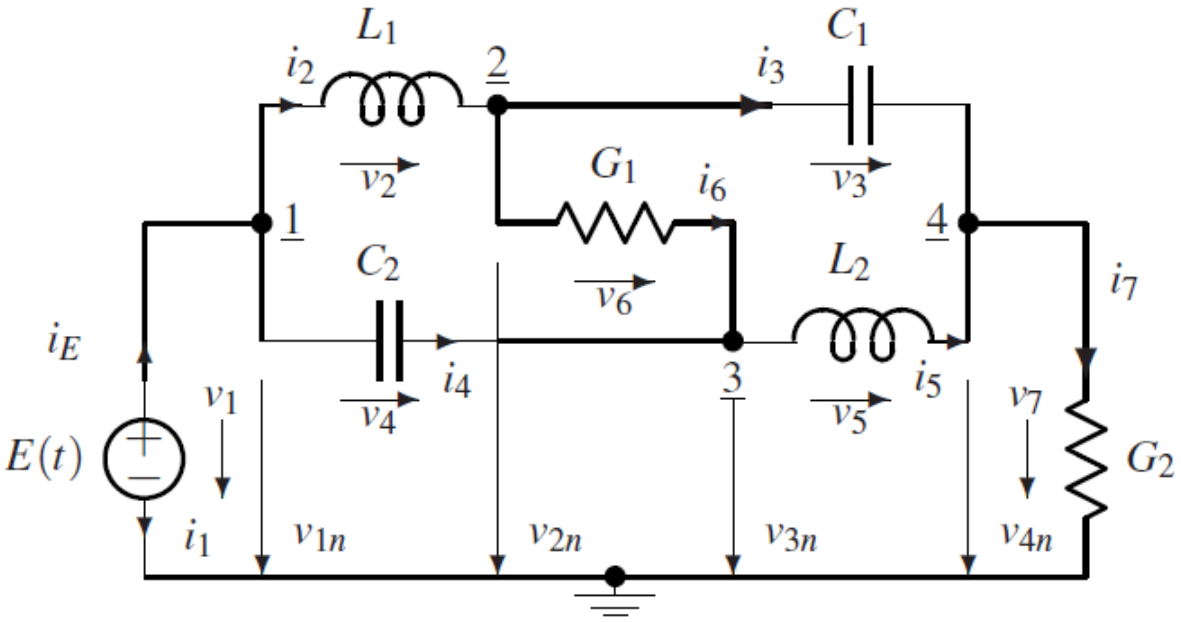

Figura 3.2: Circuito para descrever a análise nodal modificada[19].

ambos possuem a mesma topologia. Substituindo as definições dos elementos nas equações LKC, e substituindo a voltagem do elemento pela sua definição LKV, quando a função do elemento é do tipo $i = f(v)$, e nas equações LKV quando a definição do elemento é uma função da forma $v = f(i)$,

$$v_1 = E(t) \tag{3.19}$$

$$v_2 = L_1 \frac{di_2}{dt} \tag{3.20}$$

$$i_3 = C_1 \frac{dv_3}{d_t} \tag{3.21}$$

$$i_4 = C_2 \frac{dv_4}{d_t} \tag{3.22}$$

$$v_5 = L_2 \frac{di_5}{dt} \tag{3.23}$$

$$i_6 = G_1 v_6 \tag{3.24}$$

$$i_7 = G_2 v_7 \tag{3.25}$$

é possível gerar um sistema de equações, que pode ser escrito da seguinte maneira:

$$\begin{pmatrix} C_2 & 0 & -C_2 & 0 & 0 & 0 & 0 \\ 0 & C_1 & 0 & -C_1 & 0 & 0 & 0 \\ -C_2 & 0 & C_2 & 0 & 0 & 0 & 0 \\ 0 & -C_1 & 0 & C_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & L_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & L_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\frac{d}{dt}\begin{pmatrix} v_{1n} \\ v_{2n} \\ v_{3n} \\ v_{4n} \\ i_{L1} \\ i_{L2} \\ i_E \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & G_1 & -G_1 & 0 & -1 & 0 & 0 \\ 0 & -G_1 & G_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & G_2 & 0 & -1 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} v_{1n} \\ v_{2n} \\ v_{3n} \\ v_{4n} \\ i_{L1} \\ i_{L2} \\ i_E \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ E(t) \end{pmatrix}$$

Essas equações são chamadas equações nodais modificadas, e possuem a seguinte estrutura básica

$$M\frac{dx}{dt} + Nx = u(t)$$

3.4 Função de transferência

O Plano-S

É possível fazer a descrição de circuitos no domínio da frequência, na forma $Y(s) = X(s)$, em que $s = \sigma + j\omega$, é uma frequência complexa. A tranformação da equação do circuito do domínio do tempo para o domínio da frequência é feito através da transformada de laplace, que é definida por

$$X(s) = \int_0^\infty x(t)e^{-st}dt$$

Por exemplo, seja o capacitor, no qual a equação de voltagem é dada por $v(t) = \int_{t_0}^{\infty} i(t)dt$. Aplicando a transformação de Laplace, obtém-se as seguintes impedâncias:

Capacitor:

$$z_c = \frac{1}{sC}, \tag{3.26}$$

onde C é a capacitância. A equação do resistor é dada por

$$z_r = R, \tag{3.27}$$

onde R é a resistência. A equação do indutor é dada por:

$$z_i = sL, \tag{3.28}$$

onde L é a indutância e C é a capacitância, do indutor e do capacitor.

Relação entrada-saída

As funções de transferência são uma representação da resposta do circuito em relação a uma entrada. Essa relação é descrita por

$$Y(s) = H(s)X(s),$$

em que $H(s)$ é a função de transferência. Obtém-se, então:

$$H(s) = \frac{Y(s)}{X(s)}.$$

Essa representação possui diversas utilidades, como a propriedade da resposta de frequência de um circuito. Pode-se extrair a função de transferência de um circuito realizando uma análise nodal modificada, utilizando as descrições dos elementos no plano-s, e em seguida, isolando as equações para estarem em função da entrada e da saída, para poder, então, gerar a relação $H(s) = \frac{Y(s)}{X(s)}$.

Por exemplo seja o circuito representado pela figura 3.3.

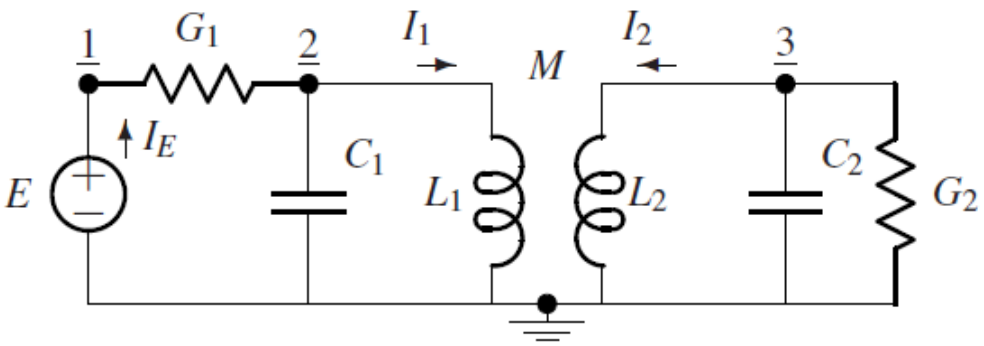

Figura 3.3: Circuito passa banda.[19]

Obtêm-se da análise nodal modificada no plano-s do circuito as seguintes equações:

$$\begin{pmatrix} G_1 & -G_1 & 0 & 0 & 0 & -1 \\ -G_1 & G_1 + sC_1 & 0 & 1 & 0 & 0 \\ 0 & 0 & G_2 + sC_2 & 0 & 1 & 0 \\ 0 & -1 & 0 & sL_1 & sM & 0 \\ 0 & 0 & -1 & sM & sL_2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} V_{1n} \\ V_{2n} \\ V_{3n} \\ I_1 \\ I_2 \\ I_e \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ E \end{pmatrix}$$

Sejam os valores dos elementos dados: $G_1 = G_2 = \frac{1}{10S}, C_1 = C_2 = 1F, L_1 = L_2 = 1H, M = \frac{1}{10}H$, então, a função de transferência é:

$$F(s) = \frac{V_{3n}}{E} = \frac{100s}{9900s^4 + 1980s^3 + 20099s^2 + 2000s + 10000}.$$

Capítulo 4

Processamento de sinais digitais

4.1 Sinal de áudio e amostragem

Um sinal é a variação de uma quantidade, como a pressão do ar em uma onda sonora, carregando uma informação a ser transmitida. Um sinal pode ser uma função de uma ou mais dimensões, ou seja, uma função de uma ou mais variáveis[18].

Um sinal $x(t)$ pode ser contínuo ou discreto. Seja um sinal contínuo $x(t) = sen(2\pi ft)$, em que a variável t representa o tempo. Ele pode ser discretizado na forma $x(n) = sen(2\pi fnt_s)$, em que t_s é o período de amostragem, no qual $t_s = \frac{1}{f_s}$, e n é o índice no tempo discreto, conforme ilustrado na Figura 4.4.

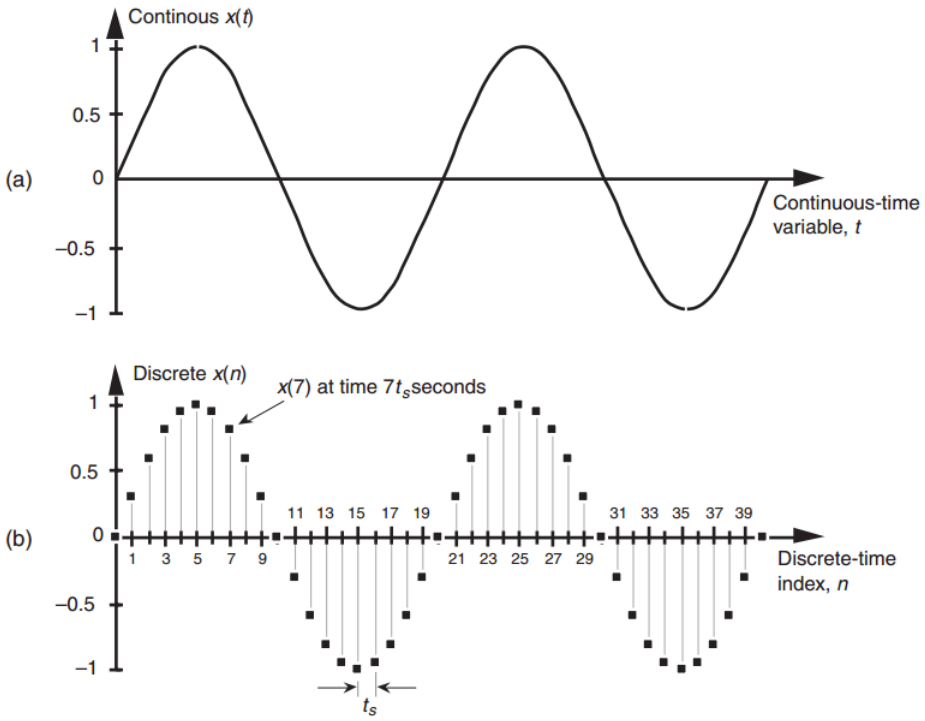

Figura 4.1: Sinal contínuo (a) e sinal discretizado (b)[10].

Um sinal pode ser representado nos domínios do tempo e da frequência, em que um sinal é mapeado, através da transformada de Fourier discreta ou contínua, para o domínio

da frequência como uma soma de ondas senóidais com frequências distintas, conforme a Figura 4.2.

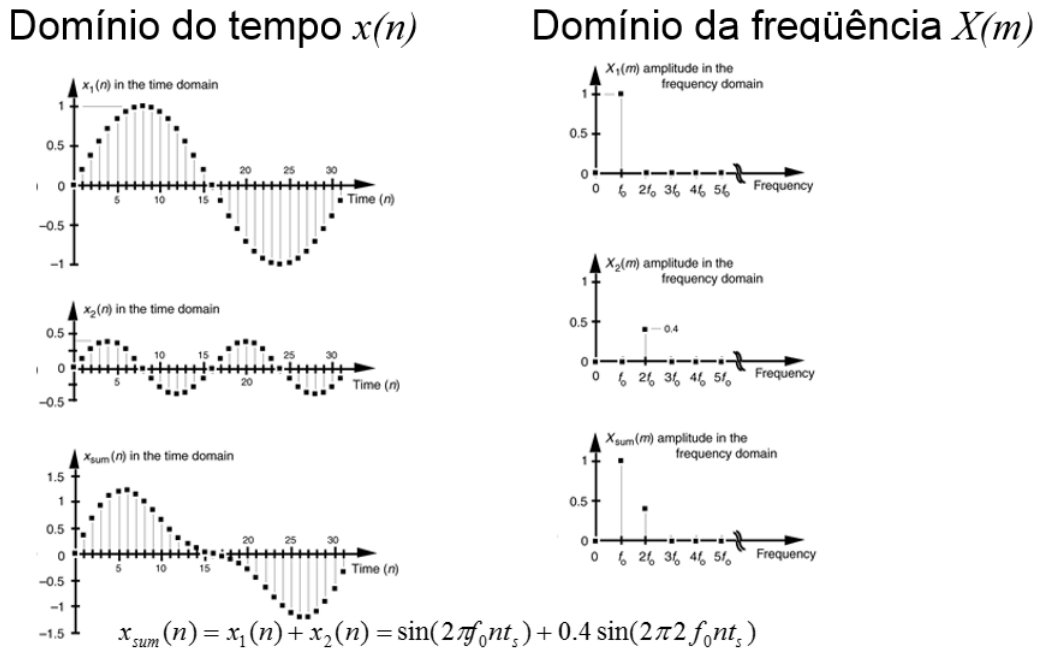

Figura 4.2: Mapeamento de sinais para o domínio da frequência.[10]

4.1.1 Teorema de Nyquist

Segundo o teorema de Nyquist, um sinal de áudio pode ser reconstruído a partir de suas amostras tomadas a um intervalo $T \leq \frac{1}{2*f_{max}}$. Caso o sinal possua uma frequência máxima superior à metade da frequência de amostragem, ele não pode ser devidamente reconstruído, gerando o fenômeno conhecido como *aliasing*, ou seja, o surgimento de frequências que não estão no espectro original, conforme ilustrado na figura 4.3.

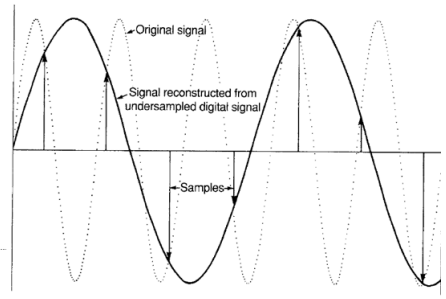

Figura 4.3: *Aliasing* gerado devido à baixa frequência de amostragem. [10]

4.1.2 Recuperação do sinal original

Quando um sinal é amostrado a uma frequência f_s, são geradas infinitas cópias espaçadas por f_s do seu espectro de frequência. Essas réplicas do espectro original podem

ser eliminadas utilizando-se um filtro passa baixa, que remove as frequências mais altas, recuperando-se assim, a cópia inicial, e por consequência o sinal inicial. Essas cópias de espectro ocorrem porque a forma assumida por um sinal discretizado é a da função degrau, que possui infinitas frequências.

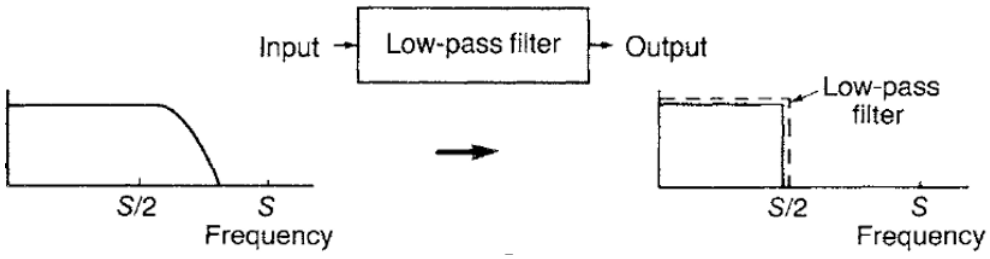

Figura 4.4: Recuperação do sinal digital através do filtro passa-baixa[10].

4.2 A transformada de Fourier Discreta

A série de Fourier foi introduzida como uma forma de expansão de uma função, em termos de senos e cossenos, que são as funções de base do método. Na análise de Fourier, um sinal é decomposto nas vibrações senoidais que o constituem. O inverso (ou seja, o sinal a ser ressintetizado) pode ser obtido somando as frequências que constituem este sinal.

A transformação de Fourier tem uma grande importância, dado que grande parte dos sinais encontrados na vida diária são gerados por algumas formas de vibrações que possuem uma estrutura periódica.

As equações de análise e síntese de Fourier discretas são as seguintes:

Equação de análise(decomposição):

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-j2\pi\ kn}{N}}$$

Equação de síntese:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{j2\pi\ kn}{N}}$$

4.2.1 Função Delta ou Função de Dirac

A função de Dirac é uma função teórica com um grande uso prático. Considere um pulso de área unitária, como mostrado na Figura 4.5. Conforme Δ tende à zero, esse pulso tende a um impulso. A função do impulso pode ser definida como a função do pulso de área unitária com uma largura infinitesimal, conforme a equação:

$$\delta(t) = \lim_{\Delta \to 0} p(t) = \begin{cases} \frac{1}{\Delta} & |t| \leq \frac{\Delta}{2} \\ 0 & |t| > \frac{\Delta}{2} \end{cases}$$

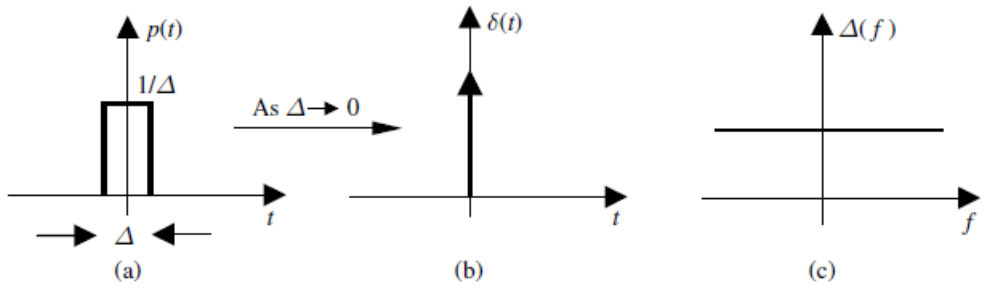

Figura 4.5: Pulso de área unitária, que tende a um impulso a medida que Δ tende à zero. [18]

É fácil de perceber que a integral da função de Dirac

$$\int_{-\infty}^{\infty} \delta(t)dt = 1$$

dado, que o pulso possui área unitária, ou seja, $\Delta \times \frac{1}{\Delta}$.

4.2.2 Exemplo - o espectro de frequência de um trem de impulsos

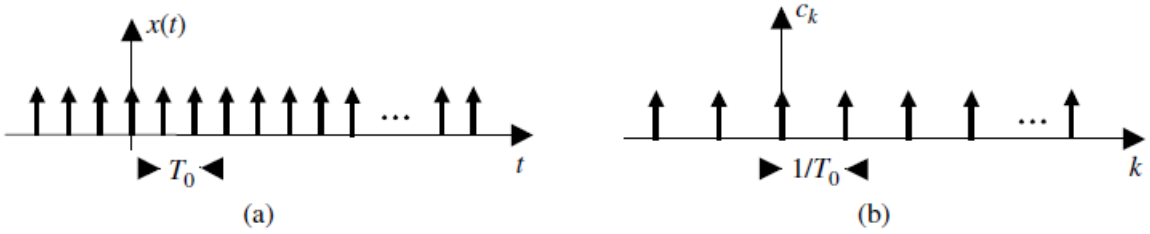

Figura 4.6: Um trem de pulsos de área unitária(a) e sua representação de no domínio da frequência [18]

Seja o trem de impulsos $x(t)$ com período e 10kHz, e amplitude A = 1mV, conforme a Figura 4.6, em que $x(t)$ é representado por:

$$x(t) = \sum_{m=-\infty}^{\infty} A \times p(t - mT_0) \times \Delta$$

Em que $p(t)$ é um pulso de área unitária de comprimento Δ. Para representar o impulso, pode-se utilizar

$$\lim_{\Delta \to 0} p(t - mT_0) \times \Delta = \int_{-\infty}^{\infty} \delta(t - mT_0)dt$$

Sabe-se que, para o intervalo de um período $-\frac{T_0}{2} < t < \frac{T_0}{2}$, $x(t) = A\delta(t)$, então, a série de Fourier para este intervalo é:

$$c_k = \frac{1}{T_0}\int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} x(t)e^{-jk\omega_0 t}dt = \frac{1}{T_0}\int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} A\delta(t)e^{-jk\omega_0 t}dt = \frac{Ae^0}{T_0}\int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} \delta(t)dt = \frac{A}{T_0} = 10V$$

O espectro de frequência de um trem de impulsos periódico é um trem de impulsos de período $1/T_0$.

4.3 A transformada-z

A transformada-z é um método no domínio da frequência, para análise e síntese de sinais e sistemas.

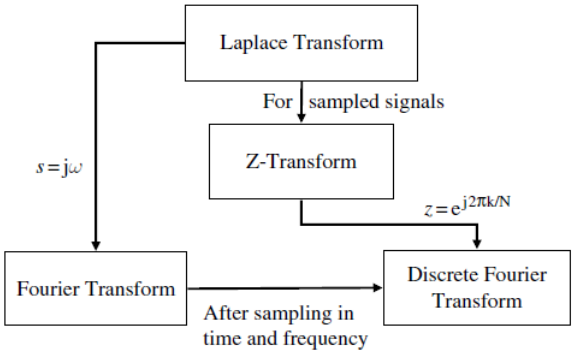

Figura 4.7: A relação entre as transformadas de *Laplace*, z e de Fourier. Para sinais discretos, a transformada de *Laplace* se torna a transformada-z[18]

A transformada-z é definida da seguinte forma:

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

em que m é o índice do sinal discreto. A variável z pode ser interpretada como um operador de *delay*, ou um alterador de fase, ou seja

$$x(m) -> z^{-k} -> x(m-k)$$

Esta propriedade pode ser provada realizando-se a transformada-z de x(m-k).

$$X(z) = \sum_{-\infty}^{\infty} x(m-k)z^{-m} = \sum_{-\infty}^{\infty} x(n)z^{-(n+k)} = z^{-k}\sum_{-\infty}^{\infty} x(n)z^{-n} = z^{-k}X(z)$$

Ela é importante para o entendimento e construção de filtros, que serão peças fundamentais para a construção dos sinais ao longo deste trabalho.

4.4 Filtros digitais

4.4.1 Introdução

Os filtros são um componente básico de processamento digital em qualquer contexto. Sua função é atenuar ou explicitar uma frequência, ou uma certa banda de frequências. Segundo *Saeed V. Vaseghi* [18], são quatro as funções principais de um filtro: confinar um sinal a uma banda de frequência pré-determinada, decompor um sinal em duas ou mais sub-bandas, modificar o espectro de frequência de um sinal e finalmente, modelar a relação entrada-saída de um sistema, como sintetizadores musicais, função essa, que será utilizada ao longo deste documento.

4.4.2 Filtros básicos

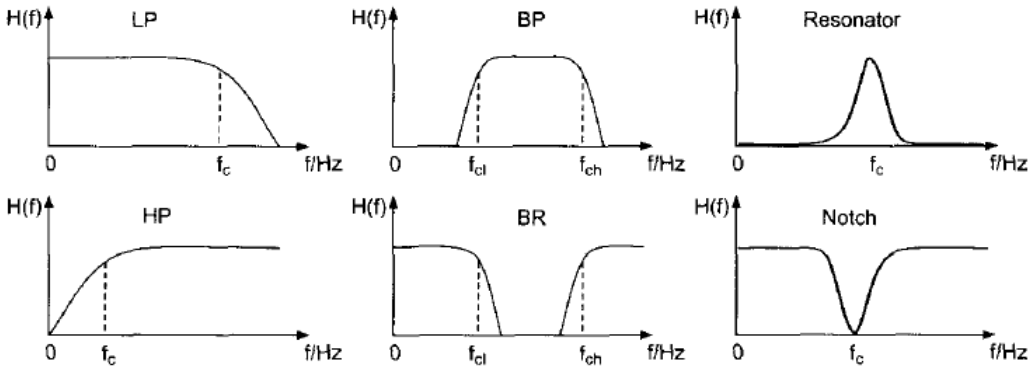

Figura 4.8: Descrição das respostas dos filtros básicos. Da esquerda para a direita: filtro passa-baixa/passsa-alta, filtro passa-banda/rejeita-banda, filtro passa-banda/rejeita-banda. *DAFx* [5])

.

Existem diversos tipos de filtros, mas alguns mais básicos são implementados em quase todos os sistemas de som, e são esses: Filtro passa baixa, no qual atenua frequências mais altas, e mantém a amplitude das frequências graves; filtro passa-alta, faz a função inversa do filtro passa-baixa; filtro passa-banda/rejeita-banda, no qual preserva a amplitude de certa quantidade de frequências definidas, mas atenua aquelas que estão fora dessa banda de frequências, conforme a Figura 4.8.

4.4.3 Descrições dos filtros

Serão abordadas duas formas de descrição de filtros, sendo elas as seguintes:

Relação entrada e saída no tempo

É utilizada uma equação de diferença, como uma combinação entre as amostras de entrada passadas e a atual. Por exemplo:

$$y(m) = ay(m-1) + x(m) \tag{4.1}$$

onde $y(m)$ é a saída do filtro $x(m)$ é a entrada, e a é um coeficiente.

Função de transferência

A função de transferência de um filtro representa a relação entre a entrada e a saída do sistema, onde em sistemas analógicos,

$$H(s) = \frac{Y(s)}{X(s)} \tag{4.2}$$

onde s representa a entrada no plano s, ou contínuo. No plano z, usado para representar sistemas digitais, temos $H(z) = \frac{Y(z)}{X(z)}$. A função de transferência de um filtro digital representa a relação entre as transformadas-z da entrada e da saída do sistema, que pode ser obtida substituindo-se $s = c\frac{1-z^{-1}}{1+z^{-1}}$ em 4.2. A resposta de frequência da saída de um sistema digital pode ser obtida a partir da sua função de transferência no plano z.[18]. Existem ferramentas que podem analisar essas funções, gerando suas respostas graficamente. Por exemplo, a ferramenta *Matlab*, da empresa *MathWorks* [12]. A função de transferência da equação 4.1, seria:

$$H(z) = \frac{1}{1 - az^{-1}} \tag{4.3}$$

4.4.4 Filtros recursivos e não-recursivos

Os filtros não-recursivos(ou filtro de resposta finita de impulso, FIR), não tem nenhum *feedback* ao seu circuito, portanto, ele se estabiliza em zero em tempo finito, após o impulso terminar. Sua equação é dada por:

$$y(m) = \sum_{k=0}^{M} b_k x(m-k)$$

Ao analisar a equação, percebe-se que a saída do filtro é constituída apenas pelas amostras anteriores da entrada, combinada com a amostra atual, portanto a saída depende da entrada, de forma que este filtro é também conhecido como sistema de *feed-forward*. Sua função de transferência possui 1 como numerador.

Já os filtros recursivos(ou filtro de resposta infinita de impulso, IIR), possuem *feedback* da sua saída para a entrada, dessa forma ele não se estabiliza em zero em tempo finito, e saída persiste indefinidamente(em teoria, infinitamente). Sua equação é dada por

$$y(m) = \sum_{k=1}^{N} a_k y(m-k) + \sum_{k=0}^{M} b_k x(m-k)$$

Podemos perceber que a equação depende também de amostras passadas da saída, desta forma, sua função de transferência possui o polinômio correspondente à primeira soma da sua equação no numerador, e no denominador possui o polinômio correspondente à segunda soma da sua equação.

4.5 Soluções numéricas para resolução de equações

Aqui será apresentado métodos para resolução de equações não-lineares e equações diferenciais ordinárias. [4]

4.5.1 Equações Não-Lineares

Método de Newton-Raphson

Método utilizado para estimar as raízes de uma equação. Ele é dado por

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Onde n é a iteração atual.

No caso do domínio digital, esta equação pode ser dada da seguinte forma

$$X[n+1] = X[n] - \frac{f(X[n])}{f'(X[n])}$$

Onde n representa o índice do passo de iteração. Conhecendo então, f(X[0]) e f(X'[0]) é possível calcular a próxima raiz, mais precisa em relação a anterior.

4.5.2 Equações diferenciais ordinárias (EDO)

Os métodos básicos para resolução de equações na forma [20]

$$\frac{dv}{dt} = \dot{v} = f(t, v),$$

onde o estado v é em geral um vetor, e $f(v, t)$ é uma função não-linear, e t, é o tempo, a variável de integração independente da EDO que descreve o circuito.

Aqui serão utilizados colchetes para representar o índice de tempo, cujo tamanho do passo é predefinido, e T representará o tamanho do passo, ou, no domínio digital, a taxa de amostragem.

Euler direto

$$v[n] = v[n-1] + T\dot{v}[n-1]$$

Em que, conhecendo o estado inicial $v[0]$, e, conhecendo também $\dot{v}[0]$, ou seja, conhecendo o valor inicial e a tangente neste ponto, é possível aproximar o próximo estado do sistema, utilizando o valor passado, e assim por diante.

Quanto menor é o passo da iteração, ou seja, a taxa de amostragem, mais precisa será a aproximação.

Euler indireto

$$v[n] = v[n-1] + T\dot{v}[n]$$

Semelhante ao método Euler direto, exceto por ser necessário a derivada do estado atual.

Regra trapezoidal

$$v[n] = v[n-1] + \frac{T}{2}(\dot{v}[n] + \dot{v}[n-1]),$$

Em que, de forma semelhante ao Euler Direto, conhecendo o valor do estado anterior, e a sua tangente, é possível aproximar o próximo estado do sistema. Como esse método depende também da derivada do estado atual, então ele passa a ser de segunda ordem. A regra trapezoidal tem o menor erro por truncamento entre todos os outros métodos de segunda ordem, e é equivalente a discretização de uma função de transferência no domínio analógico através da transformação bilinear.

Wave Digital Filter (WDF)

Os filtros digitais de onda, são uma implementação alternativa a regra trapezoidal, onde as variáveis de Kirchhoff (tensão e corrente)são substituídas pelas variáveis a e b, nas entradas e saídas das portas . Este método, será estudado com mais detalhes no capítulo 5.

4.6 Transformação bilinear

A transformação bilinear [23] é um método para discretizar uma função de transferência a partir da sua equivalente no domínio analógico. A transformação é dada por:

$$s = c\frac{1 - z^{-}1}{1 + z^{-}1}$$

O sistema contínuo:

$$H(s) = \frac{b_n s^n + ... + b_1 s + b_0}{a_n s^n + ... + a_1 s + a_0}$$

resulta em

$$H(z) = \frac{B_n z^{-n} + ... + B_1 z^{-1} + B_0}{A_n z^{-n} + ... + A_1 z^{-1} + A_0}$$

Onde $c = 2/T$, em que T é o período de amostragem do sistema. Para sistemas de primeira ordem

Para um sistema de primeira ordem, os coeficientes são

$$B_0 = b_0 + b_1 c$$

$$B_1 = b_0 - b_1 c$$

$$A_0 = a_0 + a_1 c$$

$$A_1 = a_0 + a_1 c$$

Para um sistema de segunda ordem, os coeficientes são

$$B_0 = b_0 + b_1 c + b_2 c^2$$

$$B_1 = 2b_0 - 2b_2 c^2$$

$$B_2 = b_0 - b_1 c + b_2 c^2$$

$$A_0 = a_0 + a_1 c + a_2 c^2$$

$$A_1 = 2a_0 - 2a_2 c^2$$

$$A_2 = a_0 - a_1 c + a_2 c^2$$

4.6.1 Sistema de terceira ordem

Como este trabalho utilizará o sistema de terceira ordem para descrever o *tone stack*, será apresentado o exemplo direto para um sistema desta ordem.

$$H(z) = \frac{B_0 + B_1 z^1 + B_2 z^2 + B_3 z^3}{A_0 + A_1 z^1 + A_2 z^2 + A_3 z^3}$$

Os coeficientes são

$$B_0 = -b_0 - b_1 c - b_2 c^2 - b_3 c^3,$$

$$B_1 = -3b_0 - b_1 c + b_2 c^2 + 3b_3 c^3,$$

$$B_2 = -3b_0 + b_1 c + b_2 c^2 - 3b_3 c^3,$$

$$B_3 = -b_0 + b_1 c - b_2 c^2 + b_3 c^3,$$

$$A_0 = -a_0 - a_1 c - a_2 c^2 - a_3 c^3,$$

$$A_1 = -3a_0 - a_1 c + a_2 c^2 + 3a_3 c^3,$$

$$A_2 = -3a_0 + a_1 c + a_2 c^2 - 3a_3 c^3,$$

$$A_3 = -a_0 + a_1 c - a_2 c^2 + a_3 c^3.$$

Capítulo 5

Métodos Não-Lineares

5.1 Introdução

Existem algumas características que podem ser observadas e percebidas ao se tratar de distorção. O artigo *A perceptual approach on clipping and saturation* [3] aborda algumas delas, de forma que uma distorção pode ser classificada como suave ou intensa, bem como estática ou dinâmica. Algumas técnicas de manipulação de sinal podem ser aplicadas a elas, como o limitador, que visa tornar a onda a mais limpa possível, e o compressor, que funciona como o oposto do limitador.

Os algoritmos para simulação de amplificadores propostos são diversos, como o waveshapping estático, abordado por *Davit T. Yeh* [21], que foi amplamente utilizado, e apresenta bons resultados para sinais estacionários.

Os métodos citados a seguir tentam simular circuitos musicais baseando-se na resolução de equações diferenciais ordinárias. WDF (*wave digial filters*), ou filtros digitais de onda em um tradução literal, método aplicado apresentado por *Matti Karjalainen* e *Jyri Pakarinen* em [14] simula circuitos dinâmicos, e é eficiente para caso o circuito apresente um ou duas funções não lineares, mas um amplificador de guitarra pode conter mais funções não lineares, neste caso, a eficiência do método é reduzida.

Existem também métodos numéricos para a simulação de diodos limitadores. Eles são embasados na resolução de equações diferenciais ordinárias (EDOs) não lineares. Esse método é utilizado em simuladores de circuitos esquemáticos, e é descrito por *David T. Yeh* [20], também em [23].

Finalmente, [11] apresenta um método no qual é introduzida uma aproximação na EDO que descreve a fase do triodo do amplificador digital, utilizando o método de Euler.

$$y_{n+1} = y_n + Tsf(y_{n+1})$$

5.2 Diodo

A Figura 5.1 representa o diodo limitador, e como ele pode ser aproximado para sinais pequenos. Esses circuitos são tipicamente um filtro passa baixa (um resistor ligado à saída e um capacitor ligado à terra), com um diodo limitador ligado em paralelo com o

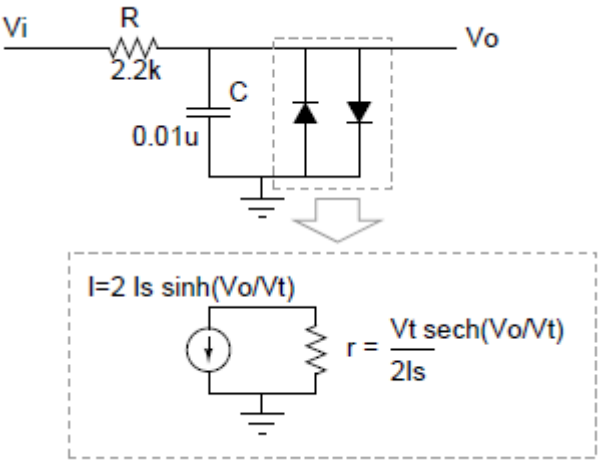

Figura 5.1: Aproximação do diodo para sinais pequenos [20].

capacitor. A equação do diodo é dada por:

$$I_d = I_s(e^{\frac{V}{V_t}} - 1)$$

onde V_t e I_s são parâmetros do modelo do diodo. A EDO correspondente ao diodo pode ser extraída a partir das leis de *Kirchhoff*, e é dada por:

$$\frac{dV_0}{dt} = \frac{V_i - V_o}{RC} - 2\frac{I_s}{C}sinh(V_o/V_t), \tag{5.1}$$

onde V_o e V_i são sinais de saída e entrada, respectivamente.

5.3 Triodo e fase de amplificação

A Figura 5.2 representa um circuito de amplificação por um triodo, cujo funcionamento está descrito no capítulo 2.

5.4 *Waveshaping* estático

O método mais direto para implementação de distorção digital, é através de um mapeamento não-linear da entrada para a saída do sistema. Este tipo de modificação é chamada de *waveshaping*. Como o mapeamento não muda com o tempo, então o método é estático.

Para estes mapeamentos, são utilizados polinômios não lineares, como por exemplo, o utilizado na Figura 5.3, primeiramente por *Araya* [2]

$$y = \frac{3x}{2}(1 - \frac{x^2}{3})$$

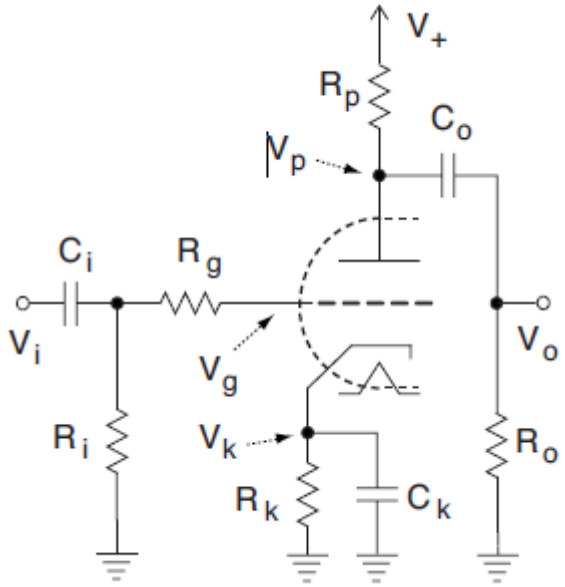

Figura 5.2: Fase de amplificação por triodo típica [14].

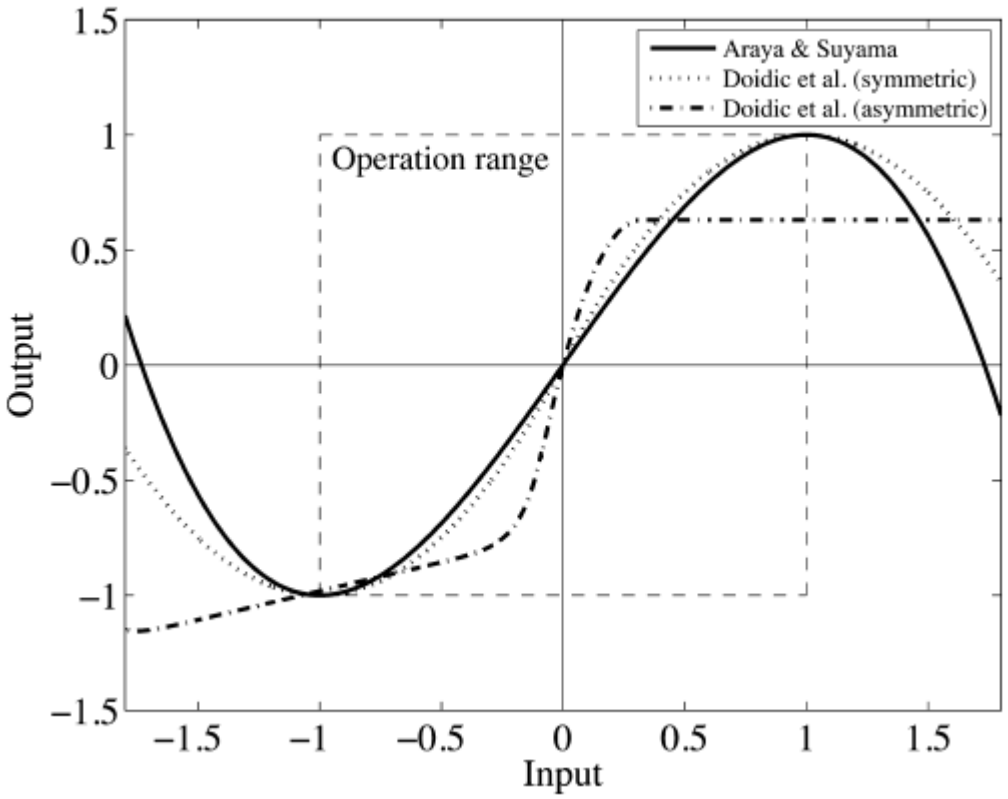

Figura 5.3: *Waveshapping* estático para uma senoide. Em pontilhado, o mapeamento estático [15].

onde x é a entrada, entre -1 e 1, e y é a saída. Outras funções não lineares são sugeridas por *Doidic* [6].

5.4.1 Tabelas de mapeamento

Além de funções não lineares, é possível gerar distorções também através de tabelas, onde a não-linearidade é salva antes do início do processamento de áudio. A vantagem do método é a possibilidade do desenho livre da não-linearidade, sendo assim, mais fácil obter a relação desejada. Em compensação, este método ocupa um espaço grande de memória, portanto, devem ser usadas interpolações, e uma tabela de baixa resolução.

5.5 Soluções numéricas

Podemos utilizar algumas estimativas numéricas para resolver EDO's de uma forma direta. Estes métodos foram explicados previamente no Capítulo 4. Podemos aplicar estes método na equação 5.1, do diodo, para estimar os próximos estados do sistema. Assumindo V_o como 0, e sendo V_i a voltagem de entrada, ou o sinal de áudio que chega ao sistema, é possível executar as iterações de aproximação da EDO.

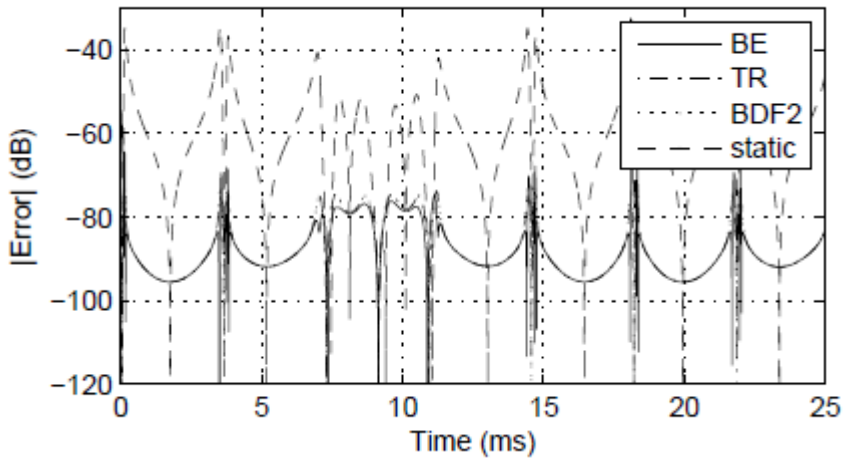

Figura 5.4: Comparação dos métodos para um diodo limitador: Euler Indireto (BE), regra trapezoidal (TR), e aproximação estática (static)[20].

5.5.1 Simulação do diodo limitador

Seja o método de Euler, dado por

$$v[n] = v[n-1] + T\dot{v}[n-1]$$

Em que $v[n]$ é o estado atual do sistema, T é a tava de amostragem e $\dot{v}$ é a Equação 5.1 de diferença do diodo. V_i é o sinal de entrada e V_o o sinal de saída anterior. Podemos inicializar o estado inicial do sistema como 0. O código Matlab[12] para a iteração de Euler, setando os devidos parâmetros físicos do diodo, e definindo um seno a 3hz como

alimentação do sistema, é dado no apêndice A deste trabalho.

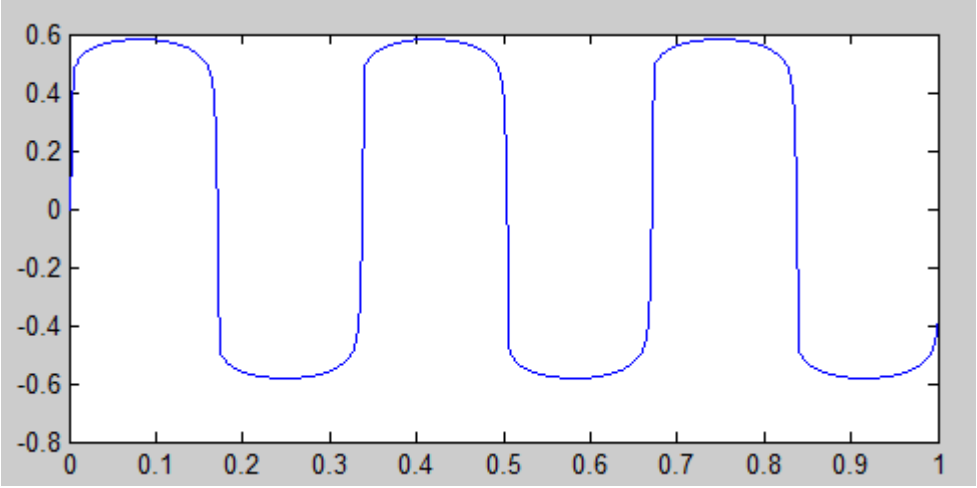

Figura 5.5: Gráfico resultante da implementação do método de Euler para uma senoide a 3khz, em uma taxa de amostragem igual a 88200hz. Observar como a senoide é clipada mais intensamente quando a entrada se aproxima do máximo.

5.5.2 Simulação da válvula(triodo)

O triodo, representado pela Figura 5.2, pode ser representado pelo modelo que pode ser encontrado no artigo [13], de *N. Noren*:

$$i_a = \frac{E_1^{E_x}}{K_{g1}}(1 + sign(E_1))$$

onde:

$$E_1 = \frac{U_{ak}}{K_p} log(1 + exp(K_p(\frac{1}{\mu} + \frac{U_{gk}}{\sqrt{K_{vb} + U_{ak}^2}}))$$

i_a é a corrente no prato do triodo, onde os parâmetros μ, E_x, K_{g1}, K_{g2}, K_p, e K_{vb} são constantes que variam de modelo a modelo de válvulas. Essas constantes estão listadas na tabela da figura 5.6.

Tube	μ	E_x	K_{g1}	K_{g2}	K_p	K_{vb}
12AX7	100	1.4	1060	-	600	300
6L6GC	8.7	1.35	1460	4500	48	12
EL34	11.0	1.35	650	4200	60	24

Figura 5.6: Constantes para os diversos modelos de válvulas.

A corrente na tela do triodo é dada por:

$$\begin{cases} g_{cf}(u_{gk} - g_{co})^{3/2}, u_{gk} \geq g_{co} & (5.2) \\ 0, u_{gk} < g_{co} & (5.3) \end{cases}$$

em que $g_{co} = -0,2$, e $g_{cf} = 10^{-5}$. u_{gk} é a diferença de potencial entre a tela e o cátodo.

O circuito do triodo pode ser encontrado em cada amplificador de guitarra. O exemplo citado pode ser descrito utilizando o sistema diferencial não-linear:

$$0 = G_2 \frac{U_i n G_1 + U_g G_2}{G_1 + G_g + G_2} - U_g G_2 - i_g$$

$$0 = U_k G_k - i_a - i_g + C_k \frac{dU_k}{dt}$$

$$0 = U_n G_a - U_a G_a - U_a G_l - i_a$$

onde i_g, a corrente na grade do triodo, é uma função que recebe $U_g - U_k$ como parâmetro, e i_a, a corrente na placa, é uma função que recebe $U_g - U_k$, e $U_a - U_k$ como parâmetros.

É possível utilizar o método de Euler para aproximar os próximos passos do sistema. Seja:

$$y_{n+1} = y_n + T_s f(y_n)$$

Chega-se ao seguinte resultado:

$$0 = G_2 \frac{U_i n G_1 + U_g G_2}{G_1 + G_g + G_2} - U_g G_2 - i_g$$

$$0 = U_k - U_{k+1} - \frac{U_k G_k - i_a - i_g}{C_k f_s}$$

$$0 = U_n G_a - U_a G_a - U_a G_l - i_a$$

Em que U_k representa o estado anterior do sistema, e pode ser inicializado em 0 na primeira amostra. Desta forma, o sistema possui duas entradas, U_k e $U_i n$. Desta forma, todas as variáveis desconhecidas do sistema podem ser calculadas a partir das diferentes combinações de entradas, e desta forma, serem aproximadas como funções de dois parâmetros.

Os resultados da simulação, encontrados no artigo [11], estão representados nas figuras 5.7, 5.8 e 5.9, em que U_{c1m} representa U_{k-1}

5.6 Wave Digital Filters

Na definição de *Yeh* [23], uma formulação alternativa para o problema das EDO's, é considerar os sinais e estados em termos de variáveis de onda, e aplicar uma discretização local, ou seja, por componente, em uma taxa de amostragem constante. WDF aplica basicamente a transformação bilinear, buscando preservar a estabilidade entre os domínios digitais e analógicos. Existem também, métodos mais precisos para este fim *Fränken* e *Ochs* [9]. Quando a transformação bilinear é utilizada para discretização, a formulação da WDF é equivalente a regra trapezoidal para resolução de EDOs e resulta numa trajetória de estados idêntica a medida que as iterações são resolvidas.

$$s = \frac{2}{T} \frac{1 - z^-1}{1 + z^-1}.$$

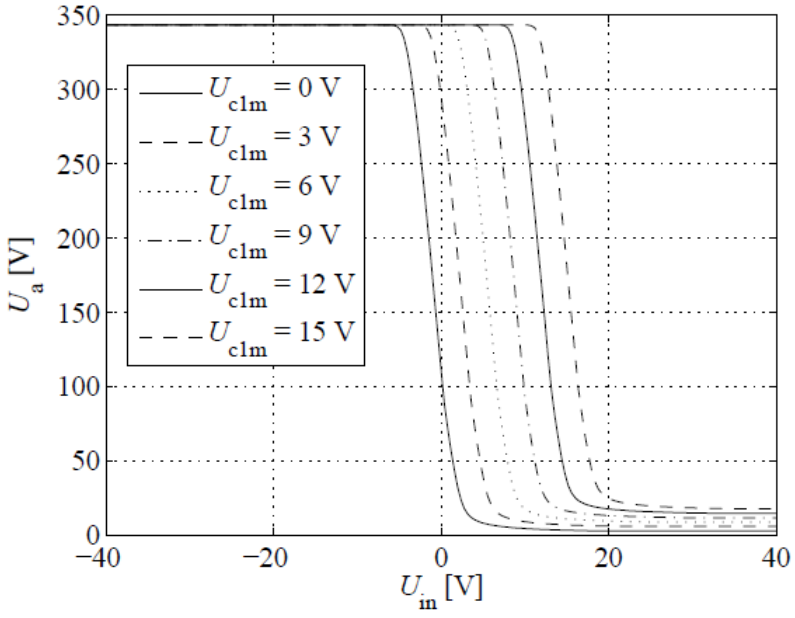

Figura 5.7: Variável Ua em função da entrada.

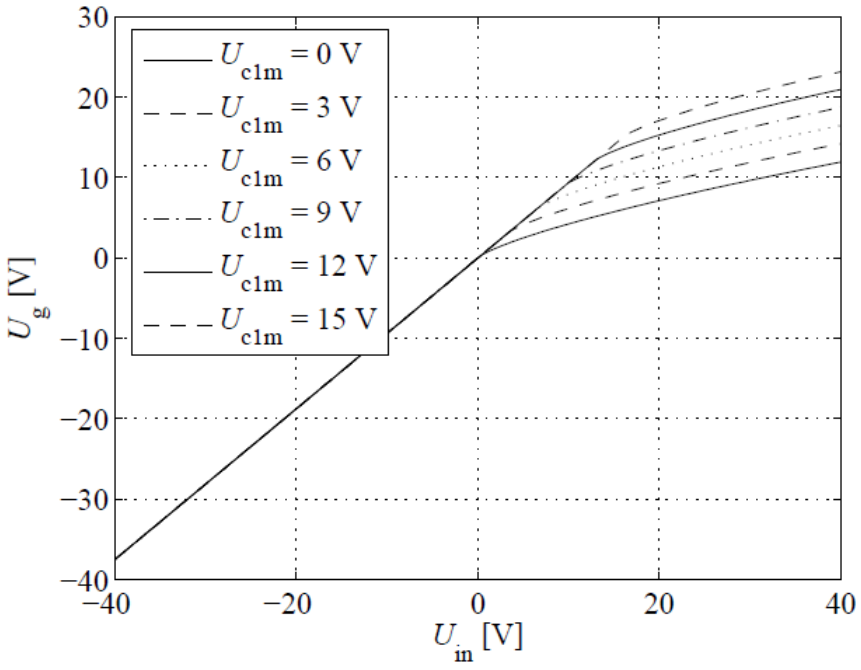

Figura 5.8: Variável Ug em função da entrada.

5.6.1 Princípios básicos

Os filtros digitais de onda veem um circuito linear de N portas como um conjunto de elementos, que são conectados um ao outro utilizando portas , em que, para cada porta, é possível definir uma voltagem V e uma corrente I. Mas, ao invés de utilizar o par de

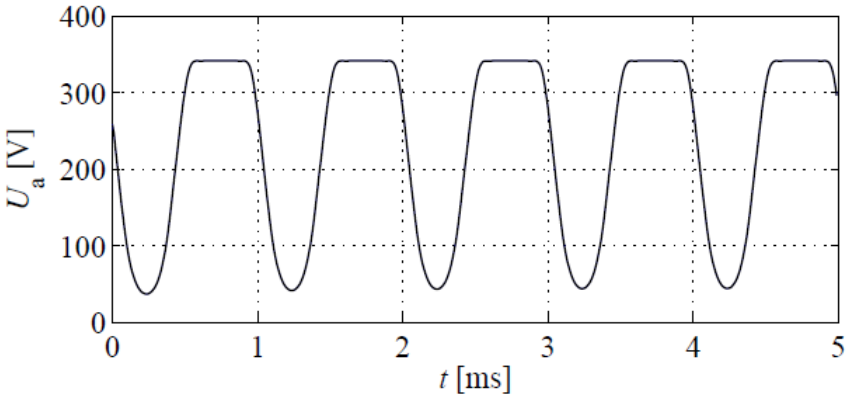

Figura 5.9: Variável Ua em função do tempo.

Kirchoff, WDF utiliza as variáveis a e b:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 & R_0 \\ 1 & -R_0 \end{bmatrix} \begin{bmatrix} V \\ I \end{bmatrix}, \tag{5.4}$$

Onde a representa a onda que chega ao elemento, e b representa a onda que sai do elemento, após ser processada. R define a impedância de porta dos elementos. Os componentes básicos de uma porta no WDF, correspondem aos componentes básicos de um circuito eletrônico: resistores, capacitores, indutores e fontes de tensão. Os elementos que contém mais de uma porta correspondem aos adaptadores em série e em paralelo. Esses adaptadores implementam as conexões em série e em paralelo, e permitem a conexão entre dois elementos de uma porta.

5.6.2 Adaptadores WDF

Os adaptadores calculam o espalhamento ao longo das portas através das suas impedâncias de porta. As relações de espalhamento entre as portas são sempre encontradas substituindo a equação 5.4 nas equações de Kirchhoff que definem o elemento ou o adaptador.

As relações de espalhamento para um adaptador de N portas em paralelo são dadas por:

$$b_v = a_p - a_v,$$

em que:

$$a_p = \gamma_1 a_1 + \gamma_2 a_2 + ... + \gamma_n a_n$$

e

$$\gamma_v = \frac{2G_v}{G1 + G2 + ... + Gn}$$

no qual γ_v são os parâmetros de cada porta v, e b_v é a onda refletida para cada porta v.

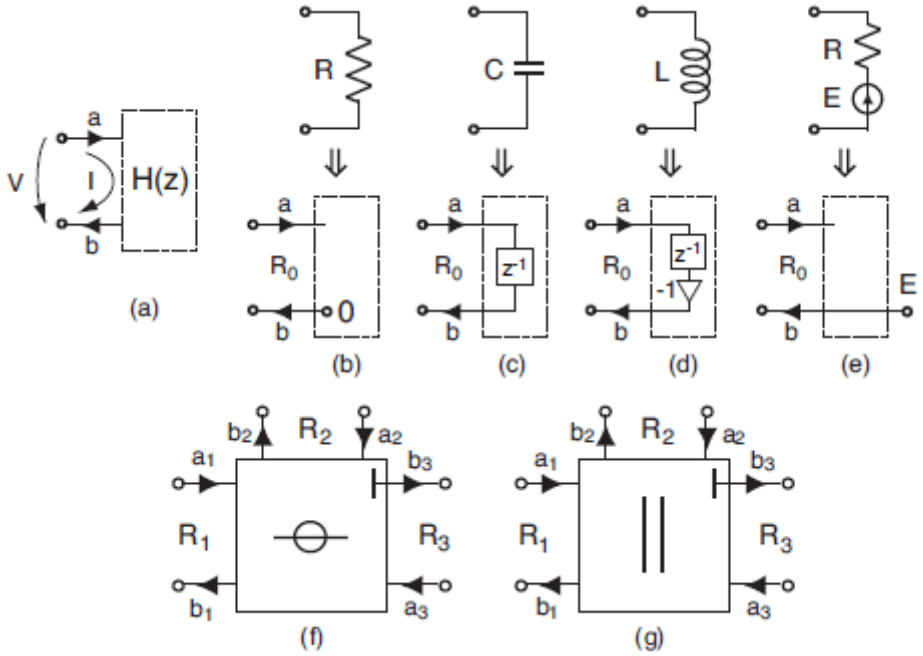

Figura 5.10: Elementos WDF, e suas ondas de saída [14].

As relações de espalhamento para um adaptador de N portas em série são dadas por:

$$b_v = a_p - \gamma_v a_s$$

em que:

$$a_p = a_1 + a_2 + ... + a_n$$

e

$$\gamma_v = \frac{2R_v}{R1 + R2 + ... + Rn}$$

onde γ_v são os parâmetros de cada porta v, e b_v é a onda refletida para cada porta v. Em ambos os casos A_v é a onda de entrada.

Após a definição dos adaptadores e elementos, já é possível construir estruturas computacionais. Normalmente, para elementos WDF não lineares, é necessário procurar uma solução numérica iterativa, para encontrar as raízes de uma equação não-linear. Essas raízes representam a onda que sai do elemento. Quando conectados, os diversos elementos WDF podem ser representados pela estrutura de árvore binária, onde os adaptadores são os nós, e os elementos são as folhas. E as variáveis de onda transitam nos dois sentidos.

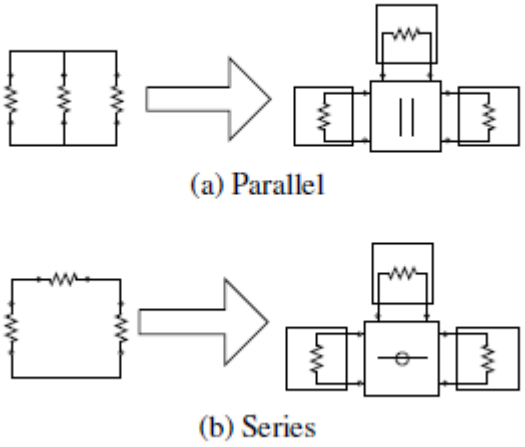

Figura 5.11: Adaptadores WDF, e seus circuitos correspondentes [23].

Capítulo 6

Ferramentas de Áudio Digital

Existem diversas ferramentas que auxiliam o desenvolvedor na construção de sínteses, e efeitos de áudio. Tanto na forma de linguagens de programação, como na forma de software. Elas facilitam a manipulação de sinais digitais, sendo possível realizar todas as transformações de maneira direta e eficiente.

6.1 Mathworks Matlab

O Matlab[12] é uma ferramenta que ajuda na simulação e criação de gráficos, bem como facilidades na manipulação de sinais, e diversas transformações úteis para esse tipo de dado. Ao longo desse trabalho ele será amplamente utilizado para realização de cálculos numéricos, simulação de funções de transferência, bem como visualização dos resultados simulados.

6.2 Steinberg VST - Virtual Studio Technology

O VST[17] é uma interface C/C++ que permite a captação de som em tempo real, bem como a síntese e uso de instrumentos em boa qualidade. É um padrão aberto, e existem diversos desenvolvedores de áudio que utilizam o VST. Os VSTs são executados a partir de um host, que chama esta interface, permitindo assim, o uso de múltiplos VSTs em um único host. A principal idéia que levou a criação do padrão, era a possibilidade da criação de um estúdio em um computador pessoal.

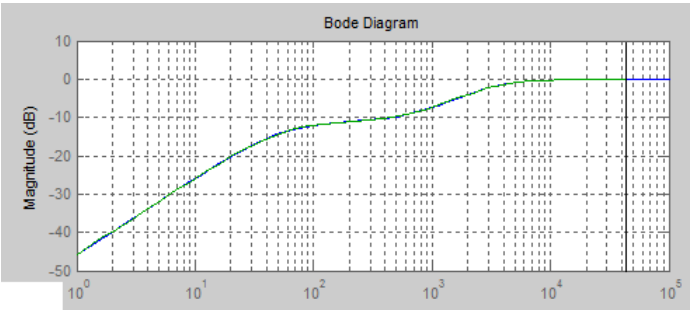

Figura 6.1: Plotagem de gráfico da ferramenta matlab[12].

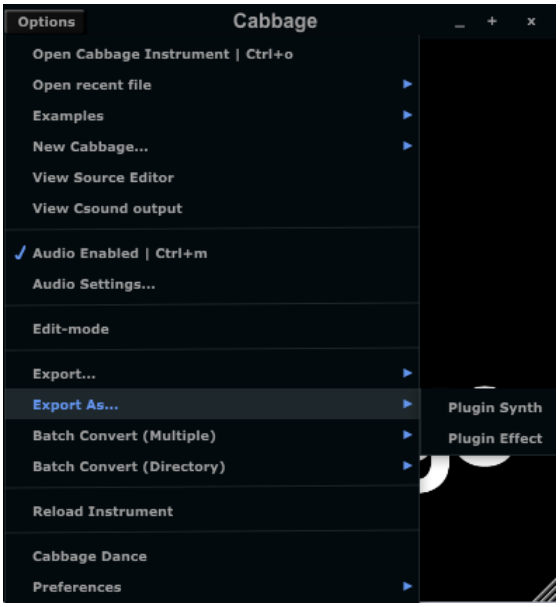

Figura 6.2: O Cabbage pode exportar arquivos CSound para plugins VST.

6.3 DAW - Digital Audio Workstaion

Os DAWs são softwares que gravam, editam, executam e geram arquivos de áudio. A grande maioria possui a funcionalidade de executar plugins externos, normalmente no formato VST. Um DAW gratuito utilizado no meio acadêmico é o audacity[7]. Podemos utilizar o Audacity para executar plugins VST, que processem o áudio gravado por ele.

6.4 CSound

O CSound é uma linguagem de script bastante direta aplicada ao processamento, síntese e manipulação de áudio digital. Ela é poderosa, e funciona através da especificação de instrumentos, que podem ser usados para gerar ou manipular áudio. Possui também ferramentas gráficas para a plotagem dos sinais, e támbém para controles, como potenciômetros e sliders. Neste trabalho essa ferramenta será utilizada para especificação de funções de transferência, e também para construção de filtros digitais.

6.5 Cabbage

O Cabbage[8] é um ferramenta que utiliza a linguagem CSound, e provê uma série de controles gráficos, para ajudar no desenvolvimento de softwares de áudio que sejam multi-plataforma. Finalmente, o Cabbege provê uma forma de traduzir o CSound para o padrão VST, de forma que é possível executar softwares CSound em DAWs. O Cabbage será a ferramenta utilizada para gerar as simulações de amplificadores neste trabalho.

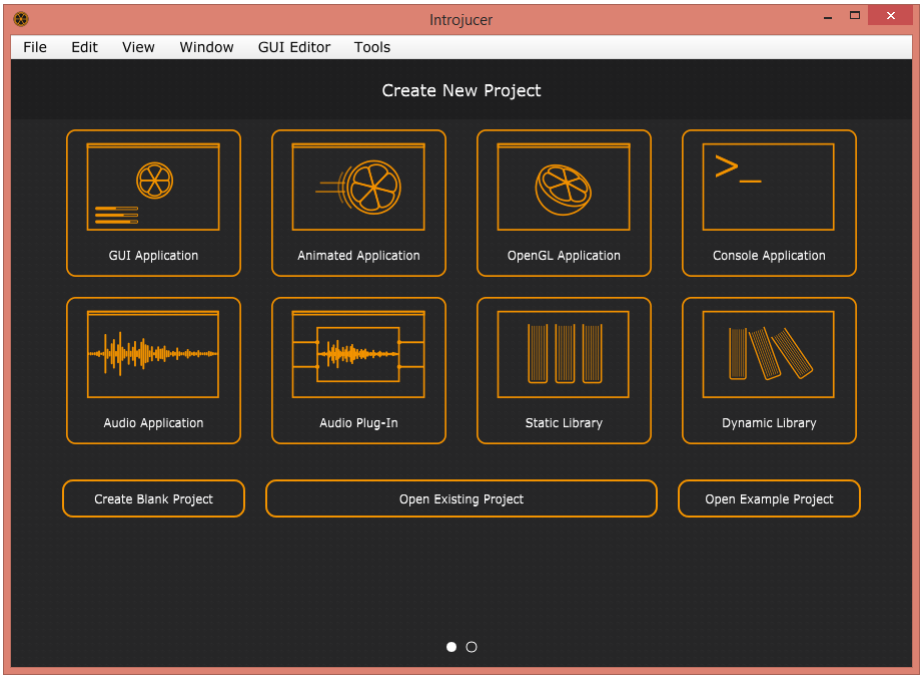

Figura 6.3: O introjucer permite a criação de plugins VST.

6.6 Juce

O Juce[16] é uma biliboteca c++ que visa criar aplicativos de áudio. O Juce provê também uma maneira simplificada de gerar VSTs, conforme a Figura 6.3 e será estudado ao longo deste trabalho. O Juce possui o *introjucer*, que gera automaticamente arquivos fonte, pronto para serem executados. Ele também possui várias ferramentas gráficas, bem como ferramentas de processamento de áudio que ajudam nos estudos dos sons gerados. Será utilizado para a construção de um dos algoritmos de simulação de amplificador neste trabalho, o WDF.

Capítulo 7

Simulação digital do *Tone Stack* e análise dos dados

7.1 Implementação

Será abordado primeiramente a implementação do *tone stack*, devido a sua simplicidade em relação à simulação das válvulas eletrônicas, utilizado-se, para isso, o circuito descrito por *David T. Yeh* e *Julius Smith* [22], do amplificador clássico *Bassman '59* da *Fender*. Ele é exposto Figura 7.1, e representa um filtro linear variante no tempo, pois possui potenciômetros que variam seus coeficientes, a saber, t (*treble*), m (*medium*) e l (*low*).

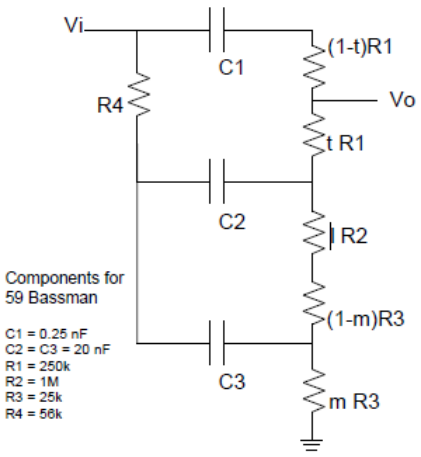

Figura 7.1: *Tone stack* do *Fender Bassman* [22]

.

É possível encontrar a função de transferência no plano s que descreve esse circuito, através da sua análise nodal. Ela é dada pela equação 7.1, e está disponível em [22]:

$$H(s) = \frac{b1s + b2s^2 + b3s^3}{a0 + a1s + a2s^2 + a3s^3} \tag{7.1}$$

Onde:

$$b1 = t*C1*R1 + m*C3*R3 + l*(C1*R2 + C2*R2) + (C1*R3 + C2*R3) \quad (7.2)$$

$$\begin{aligned} b2 = {} & t*(C1*C2*R1*R4 + C1*C3*R1*R4) - (m*m)*(C1*C3*(R3*R3) \\ & + C2*C3*(R3*R3)) \\ & + m*(C1*C3*R1*R3 + C1*C3*(R3*R3) + C2*C3*(R3*R3)) \\ & + l*(C1*C2*R1*R2 + C1*C2*R2*R4 + C1*C3*R2*R4) \\ & + l*m*(C1*C3*R2*R3 + C2*C3*R2*R3) \\ & + (C1*C2*R1*R3 + C1*C2*R3*R4 + C1*C3*R3*R4) \end{aligned} \quad (7.3)$$

$$\begin{aligned} b3 = {} & l*m*(C1*C2*C3*R1*R2*R3 + C1*C2*C3*R2*R3*R4) - (m*m) \\ & *(C1*C2*C3*R1*(R3*R3) + C1*C2*C3*(R3*R3)*R4) + m \\ & *(C1*C2*C3*R1*(R3*R3) + C1*C2*C3*(R3*R3)*R4) + t*C1*C2*C3 \\ & *R1*R3*R4 - t*m*C1*C2*C3*R1*R3*R4 + t*l*C1*C2*C3*R1*R2*R4 \end{aligned} \quad (7.4)$$

$$a0 = 1 \quad (7.5)$$

$$\begin{aligned} a1 = {} & (C1*R1 + C1*R3 + C2*R3 + C2*R4 + C3*R4) \\ & + m*C3*R3 + l*(C1*R2 + C2*R2) \end{aligned} \quad (7.6)$$

$$\begin{aligned} a2 = {} & m*(C1*C3*R1*R3 - C2*C3*R3*R4 + C1*C3*(R3*R3) + C2*C3*(R3*R3)) \\ & + l*m*(C1*C3*R2*R3 + C2*C3*R2*R3) \\ & - (m*m)*(C1*C3*(R3*R3) + C2*C3*(R3*R3)) + l \\ & *(C1*C2*R2*R4 + C1*C2*R1*R2 + C1*C3*R2*R4 + C2*C3*R2*R4) \\ & + (C1*C2*R1*R4 + C1*C3*R1*R4 + C1*C2*R3*R4 + C1*C2*R1*R3 \\ & + C1*C3*R3*R4 + C2*C3*R3*R4) \end{aligned} \quad (7.7)$$

$$\begin{aligned} a3 = {} & l*m*(C1*C2*C3*R1*R2*R3 + C1*C2*C3*R2*R3*R4) - (m*m) \\ & *(C1*C2*C3*R1*(R3*R3) + C1*C2*C3*(R3*R3)*R4) + m \\ & *(C1*C2*C3*R4*(R3*R3) + C1*C2*C3*(R3*R3)*R1 - C1*C2 \\ & *C3*R1*R3*R4) + l*C1*C2*C3*R1*R2*R4 + C1*C2*C3*R1*R3*R4 \end{aligned} \quad (7.8)$$

Realizando a análise dessa função de transferência no *Matlab*[12], obtemos a resposta de frequência, mostrada na figura 7.2,

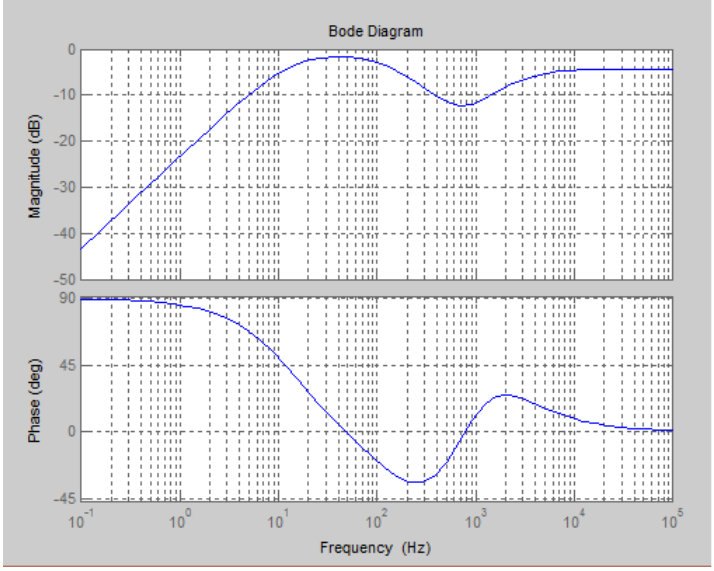

Figura 7.2: Análise da função de tranferência do *Fender Bassman* [12]

.

A equação de transferência pode ser discretizada utilizando a transformação bilinear. Substituindo $s = c\frac{1-z^{-1}}{1+z^{-1}}$ na equação 7.1, obtêm-se:

$$H(z) = \frac{B0 + B1z^{-1} + B2z^{-2} + B3z^{-3}}{A0 + A1z^{-1} + A2z^{-2} + A3z^{-3}} \tag{7.9}$$

Onde:

$$B0 = -b1c - b2c^2 - b3c^3$$
$$B1 = -b1c + b2c^2 + 3b3c^3$$
$$B2 = b1 * c + b2 * c^2 - 3 * b3 * c^3$$
$$B3 = b1 * c - b2 * c^2 + b3 * c^3$$
$$A0 = -a0 - a1 * c - a2 * c^2 - a3 * c^3$$
$$A1 = -3 * a0 - a1 * c + a2 * c^2 + 3 * a3 * c^3$$
$$A2 = -3 * a0 + a1 * c + a2 * c^2 - 3 * a3 * c^3$$
$$A3 = -a0 + a1 * c - a2 * c^2 + a3 * c^3$$

Que é a função de tranferência de um filtro de resposta infinita ao impulso, variante no tempo. Realizando a análise da função de transferência no eixo z, no *Matlab* [12], obtem-se a resposta de frequência representada na Figura 7.3, sendo notável a inversão de fase. Quanto a resposta de frequência, manteve-se igual ao plano s.

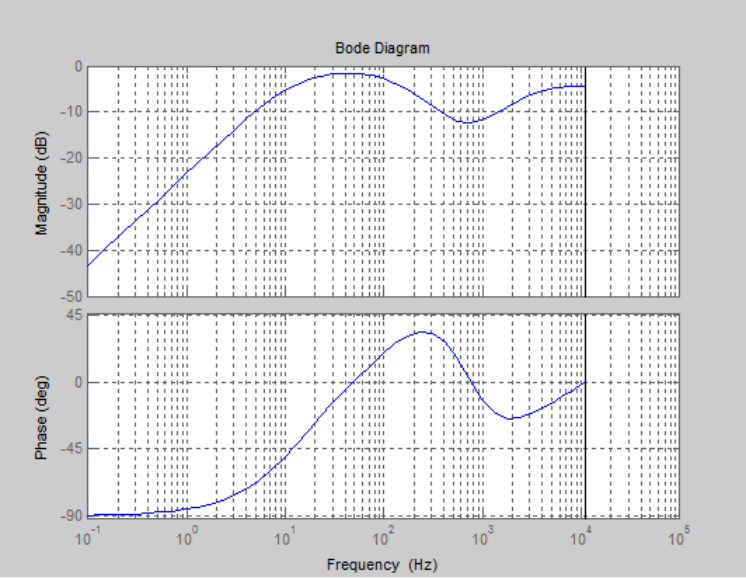

Figura 7.3: Análise da função de tranferência do *Fender Bassman* [12] discretizada

.

7.2 Análise comparativa

Fazendo a análise utilizando a ferramenta *Matlab*[12], podemos observar os seguintes resultados, para uma frequência de amostragem de 44.1KHz:

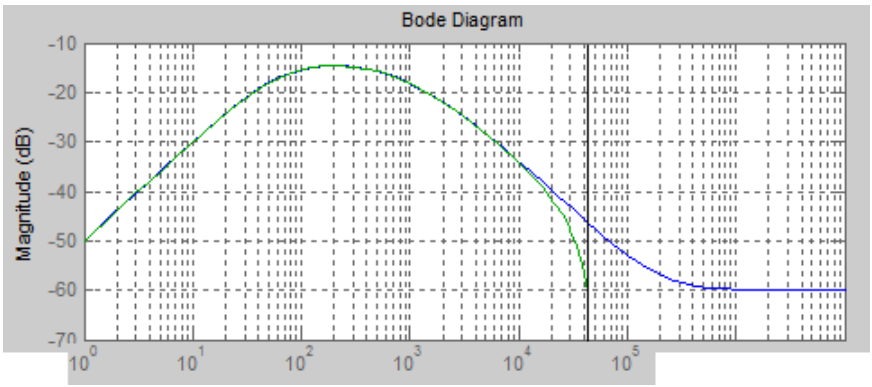

Figura 7.4: Resposta de frequência para l = 0, m = 0, t = 0.001.

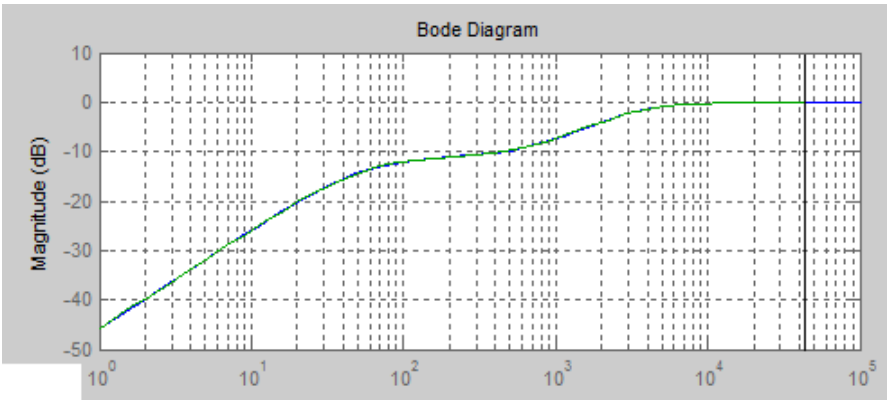

Figura 7.5: Resposta de frequência para l = 0, m = 0.5, t = 1.

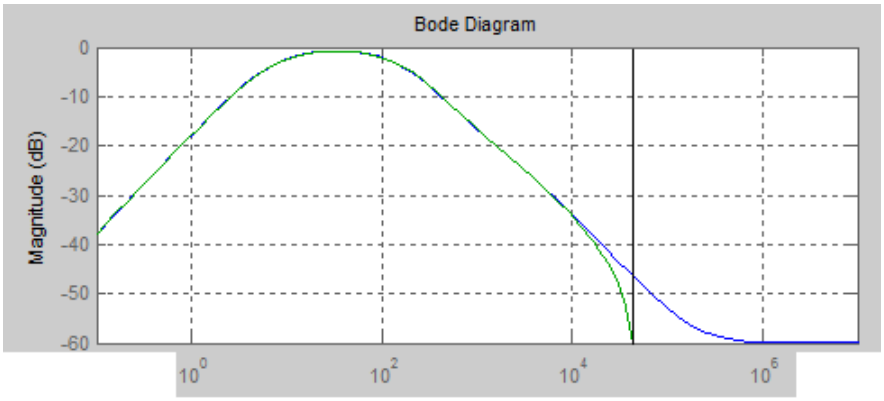

Figura 7.6: Resposta de frequência para l = 1, m = 0, t = 0.001.

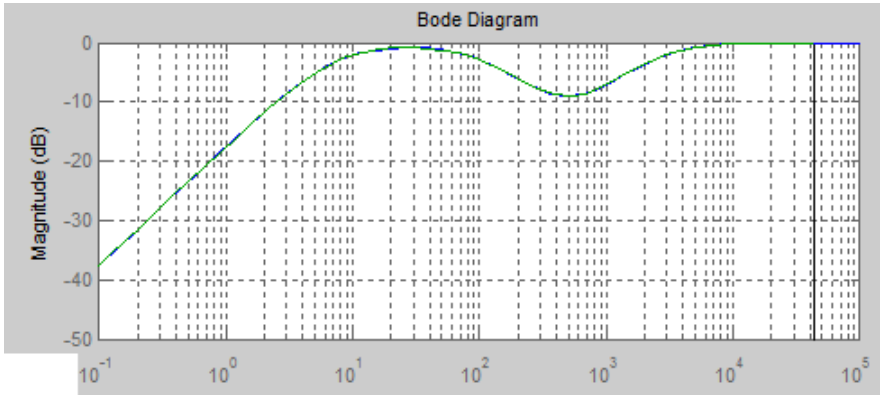

Figura 7.7: Resposta de frequência para l = 1, m = 1, t = 1.

Nos gráficos comparativos, ambas as funções de transferência (digital e analógica) foram analisadas no Matlab, em que a linha azul representa a resposta de frequência analógica, e a linha verde representa a resposta de frequência digital.

Desta forma, o *tone stack* pode ser parametrizado de forma exata, exceto em algumas frequências mais extremas, e a transformação bilinear de discretização proveu um

mapeamento muito bom de circuitos analógicos (essas variações nas frequências mais altas já eram esperadas da transformação bilinear), utilizando-se de uma frequência de amostragem relativamente fácil de se obter, 44.1KHz ou superior já é uma frequência de amostragem amplamente utilizada, em qualquer processador razoavelmente moderno. A simulação do *tone stack* no cabbage pode ser encontrada no apêndice B deste documento.

Capítulo 8

Implementação WDF da Válvula

8.1 Diodo

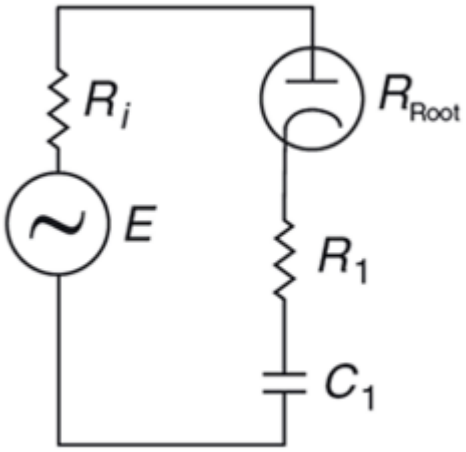

Figura 8.1: Circuito elétrico contendo um diodo. [14].

Considere o circuito contendo um diodo apresentado na Figura 8.1, em que $R_1 = 80\Omega$, $R_i = 1\Omega$, $C_1 = 35\mu F$. A equação do diodo é dada por:

$$I_d = I_s e^{(\frac{V_d}{V_t} - 1)},$$

em que, para o diodo apresentado, $V_t = 36mV$ e $I_s = 125.56A$. A representação WDF do circuito é dada pela Figura 8.2, sendo constituído apenas por adaptadores em série.

Cada elemento é ligado a um adaptador, que por sua vez, é capaz de calcular a onda que vai em direção a raiz, e a onda que se espalha até chegar aos elementos, que são as folhas da árvore WDF. A saída do circuito é a voltagem no resistor R_1. Todos os elementos são conhecidos, e fazem parte da definição dos elementos WDF, exceto pelo próprio diodo. Sua não-linearidade pode ser calculada através da resistência não-linear do diodo que é dada por

$$R_d = I_s e^{-V_t V_d},$$

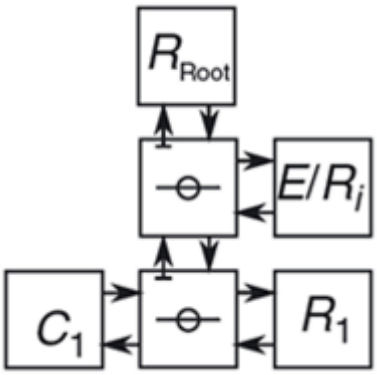

Figura 8.2: Representação WDF do circuito contendo um diodo. [14].

e então atualizando a onda incidente ao elemento, de acordo com as regras de espalhamento, em que

$$b = (\frac{R_d - R_s}{R_d + R_s})a,$$

e R_s é a resistência da porta do adaptador em série ligado ao diodo. Este elemento é então colocado no topo da árvore, de acordo com a formulação WDF. O resultado do circuito é apresentado na Figura 8.3.

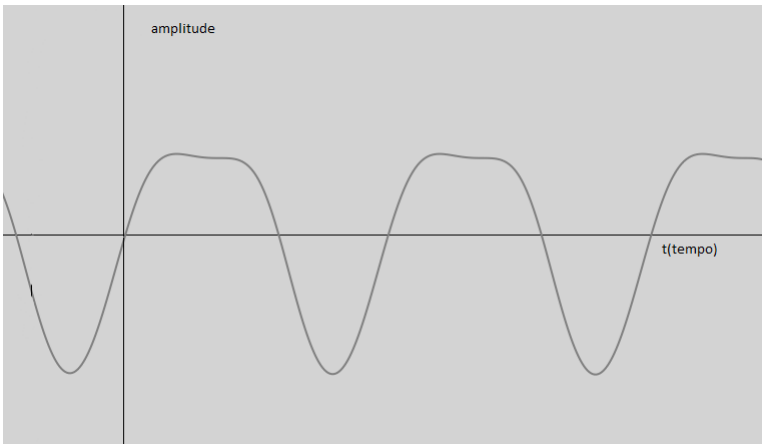

Figura 8.3: Resultado da implementação do diodo.

8.2 Triodo

O triodo apresentado neste trabalho pode ser representado pela estrutura exposta na Figura 8.4. Todos os elementos são conhecidos, exceto pelo próprio triodo, de forma deve-se modelar sua não linearidade, para que possa ser inserido corretamente na árvore WDF. Considerando o seguinte modelo já apresentado

$$i_a = \frac{E_1^{E_x}}{K_{g1}}(1 + sign(E_1)),$$

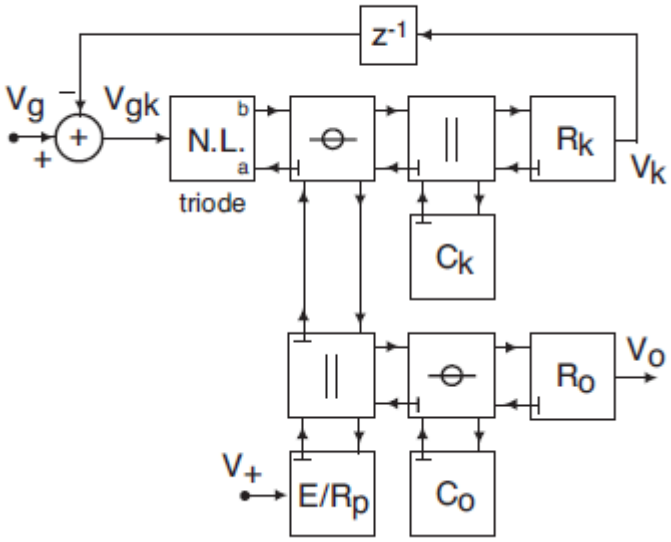

Figura 8.4: Árvore binária WDF, representando o circuito da fase de amplificação. [14].

onde:

$$E_1 = \frac{V_{pk}}{K_p} log(1 + exp(K_p(\frac{1}{\mu} + \frac{V_{gk}}{\sqrt{K_{vb} + V_{pk}^2}}))$$

em que i_a é a corrente do elemento em função do tempo, podemos aplicá-lo ao método WDF:

$$a = V_{pk} + R.ia(V_{pk}, V_{gk}),$$

ou seja,

$$V_{pk} + R.ia(V_{pk}, V_{gk}) - a = 0$$

$$b = Vpk + R.ia(V_{pk}, V_{gk})$$

Onde V_{gk} e a são conhecidos, e R é a resistência da porta do adaptador em série. Pode-se aproximar o valor de V_{pk} a partir das equações, utilizando-se o método de Newton-Raphson, que para este caso, é expresso da seguinte forma:

$$V_{pk}[n+1] = V_{pk}[n] - \frac{f(V_{pk}[n])}{f'(V_{pk}[n])}$$

em que $f(V_{pk}) = V_{pk} + R.ia(V_{pk}, V_{gk}) - a$ e $f'(V_{pk})$ é a derivada de $f(V_{pk})$. A onda refletida do triodo pode então ser calculada, a partir da nova amostra de V_{pk}, que por sua vez desce até as folhas da árvore. O resultado desse circuito é apresentado na Figura 8.5.

A implementação WDF em c++ pode ser encontrada no apêndice C deste trabalho.

8.3 Considerações Finais

O método WDF é computacionalmente ineficiente, devido a sua estrutura complexa. Segundo os estudos presentes em [14], para uma taxa de amostragem de 44100Hz, o

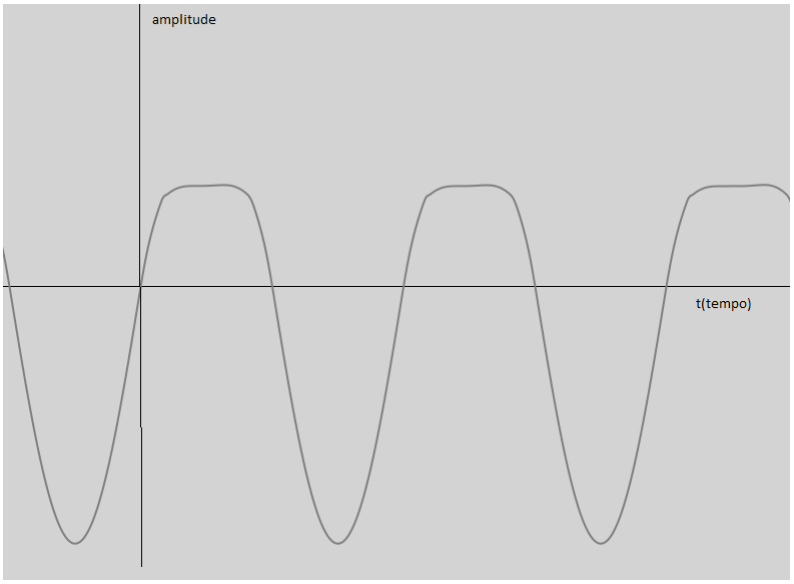

Figura 8.5: Senóide clipada pela fase do trido.

método WDF requer 3.76 vezes mais uso de CPU, em relação a uma iteração baseada em tabela.

A grande vantagem do método WDF é a sua modularidade, que permite mapear circuitos de maneira mais intuitiva em relação aos demais. De forma que pode-se utilizar os diversos elementos já criados para reproduzir um dado circuito. Uma abordagem possível para reduzir a sua grande demanda computacional, seria a geração tabelas em tempo de compilação, para as diversas possibilidades de voltagem na tela, e de ganho.

Capítulo 9

Conclusão

Ao longo deste trabalho foram desenvolvidos e analisados os métodos WDF, e de análise nodal. Ambos os algoritmos possuem complexidade $O(n)$, em que para o WDF, n é o número de elementos de circuito que fazem parte da árvore. E na análise nodal, n é o número da ordem do filtro IIR.

Mesmo assim, devido a parte não-linear do circuito, o WDF pode se tornar lento, pois ela geralmente é implementada utilizando métodos iterativos para achar raízes de uma equação. Portanto, seria ideal que o circuito contesse poucos elementos não-lineares, para permitir uma simulação em tempo real.

Quanto à fidelidade, ambos os métodos foram satisfatórios. O WDF gerou formas de onda comparáveis às válvulas, formando uma distorção mais suave em ralação as distorções digitais convencionais. O *tone stack* obteve resultados iguais ao circuito analógico, exceto em frequências próximas a frequência de amostragem.

Para trabalhos futuros, deve-se simular o circuito de um amplificador completo. A análise do *Fender Bassman 59* pode ser encontrado online, facilitando a implementação WDF. O *tone stack* dele já foi implementado neste trabalho.

Apêndice A

Simulação Matlab do Diodo Utilizando o Método de Euler

```
taxaamost = 1/88200;
freq = 3;
x = 0:(taxaamost):1;
Vi = sin(2*pi*freq*x)*10e12;
Is = 2.52e 9;
Vt = 45.3e 3;
R = 2.2e3;
C = 0.46e 6;
Vo = cell(1, 88201);
i = 1;
Vo(1) = {0};
while i <= 88200
    dv = (Vi(i)    cell2mat(Vo(i)))/R*C    2*(Is/C)*sinh(cell2mat(Vo(i))/Vt);
  Vo(i+1) = {cell2mat(Vo(i)) + taxaamost*dv};
   i = i + 1;
end
plot(x, cell2mat(Vo));
```

Apêndice B

Implementação CSound no Cabbage do ToneStack

```
<Cabbage>
form size(400, 300), caption("Untitled"), pluginID("plu1")
checkbox bounds(0, 0, 60, 20), channel("liga"), text("liga"), value(1)
rslider bounds(60, 0, 60, 60), channel("vol"), text("vol"), range(0, 1, 1)
rslider bounds(120, 0, 60, 60), channel("t"), text("treble"), range(0, 1,
    0)
rslider bounds(180, 0, 60, 60), channel("m"), text("medium"), range(0, 1,
    0)
rslider bounds(240, 0, 60, 60), channel("l"), text("low"), range(0, 1, 1)

</Cabbage>
<CsoundSynthesizer>
<CsOptions>
 n  d
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs=1

instr 1
  kliga chnget "liga"
  kt chnget "t"
  km chnget "m"
  kl chnget "l"
  ainL, ainR ins


inicializacao:
  ic = 2*sr
  iC1 = 0.25e 9
  iC2 = 20e 9
  iC3 = iC2
  iR1 = 250e3
  iR2 = 1e6
  iR3 = 25e3
```

```
36  iR4 = 56e3
37
38  it chnget "t"
39  im chnget "m"
40  il chnget "l"
41
42  ib1 = it*iC1*iR1 + im*iC3*iR3 + il*(iC1*iR2 + iC2*iR2) + (iC1*iR3 + iC2*
        iR3)
43
44  ib2 = it*(iC1*iC2*iR1*iR4 + iC1*iC3*iR1*iR4)   (im*im)*(iC1*iC3*(iR3*iR3)
        + iC2*iC3*(iR3*iR3)) + im*(iC1*iC3*iR1*iR3 + iC1*iC3*(iR3*iR3) + iC2*
        iC3*(iR3*iR3)) + il*(iC1*iC2*iR1*iR2 + iC1*iC2*iR2*iR4 + iC1*iC3*iR2*
        iR4) + il*im*(iC1*iC3*iR2*iR3 + iC2*iC3*iR2*iR3) + (iC1*iC2*iR1*iR3 +
        iC1*iC2*iR3*iR4 + iC1*iC3*iR3*iR4)
45
46  ib3 = il*im*(iC1*iC2*iC3*iR1*iR2*iR3 + iC1*iC2*iC3*iR2*iR3*iR4)   (im*im)
        *(iC1*iC2*iC3*iR1*(iR3*iR3) + iC1*iC2*iC3*(iR3*iR3)*iR4) + im*(iC1*iC2
        *iC3*iR1*(iR3*iR3) + iC1*iC2*iC3*(iR3*iR3)*iR4)  + it*iC1*iC2*iC3*iR1*
        iR3*iR4   it*im*iC1*iC2*iC3*iR1*iR3*iR4 + it*il*iC1*iC2*iC3*iR1*iR2*
        iR4
47
48  ia0 = 1
49
50  ia1 = (iC1*iR1 + iC1*iR3 + iC2*iR3 + iC2*iR4 + iC3*iR4) + im*iC3*iR3 + il
        *(iC1*iR2 + iC2*iR2)
51
52  ia2 = im*(iC1*iC3*iR1*iR3   iC2*iC3*iR3*iR4 + iC1*iC3*(iR3*iR3) + iC2*iC3
        *(iR3*iR3)) + il*im*(iC1*iC3*iR2*iR3 + iC2*iC3*iR2*iR3)   (im*im)*(iC1
        *iC3*(iR3*iR3) + iC2*iC3*(iR3*iR3)) + il*(iC1*iC2*iR2*iR4 + iC1*iC2*
        iR1*iR2 + iC1*iC3*iR2*iR4 + iC2*iC3*iR2*iR4) + (iC1*iC2*iR1*iR4 + iC1*
        iC3*iR1*iR4 + iC1*iC2*iR3*iR4 + iC1*iC2*iR1*iR3 + iC1*iC3*iR3*iR4 +
        iC2*iC3*iR3*iR4)
53
54  ia3 = il*im*(iC1*iC2*iC3*iR1*iR2*iR3 + iC1*iC2*iC3*iR2*iR3*iR4)   (im*im)
        *(iC1*iC2*iC3*iR1*(iR3*iR3) + iC1*iC2*iC3*(iR3*iR3)*iR4) + im*(iC1*iC2
        *iC3*iR4*(iR3*iR3) + iC1*iC2*iC3*(iR3*iR3)*iR1   iC1*iC2*iC3*iR1*iR3*
        iR4) + il*iC1*iC2*iC3*iR1*iR2*iR4 + iC1*iC2*iC3*iR1*iR3*iR4
55  /*                                                          */
56  iB0 =  ib1*ic   ib2*(ic^2)   ib3*(ic^3)
57
58  iB1 =  ib1*ic + ib2*(ic^2) + 3*ib3*(ic^3)
59
60  iB2 = ib1*ic + ib2*(ic^2)   3*ib3*(ic^3)
61
62  iB3 = ib1*ic   ib2*(ic^2) + ib3*(ic^3)
63
64  iA0 =  ia0   ia1*ic   ia2*(ic^2)   ia3*(ic^3)
65
66  iA1 =  3*ia0   ia1*ic + ia2*(ic^2) + 3*ia3*(ic^3)
67
68  iA2 =  3*ia0 + ia1*ic + ia2*(ic^2)   3*ia3*(ic^3)
69
70  iA3 =  ia0 + ia1*ic   ia2*(ic^2) + ia3*(ic^3)
71
72  aoutL filter2 ainL, 4, 3, iB0/iA0, iB1/iA0, iB2/iA0, iB3/iA0, iA1/iA0,
        iA2/iA0, iA3/iA0
```

```
  aoutR filter2 ainR, 4, 3, iB0/iA0, iB1/iA0, iB2/iA0, iB3/iA0, iA1/iA0,
      iA2/iA0, iA3/iA0

  rireturn
  /*ktrigv metro 200
  kval max_k aoutL, ktrigv, 1
  printk 1, kval*/

  ktrig changed kt
  ktrig1 changed km
  ktrig2 changed kl
  if ktrig == 1 || ktrig1 == 1 || ktrig2 == 1then
    reinit inicializacao
  endif
  if kliga == 1 then
    outs aoutL*6, aoutR*6
  elseif kliga == 0 then
    outs ainL, ainR
  endif


endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1
f0 3600

i 1 0 3600
</CsScore>
</CsoundSynthesizer>
```

Apêndice C

Implementação do Método WDF do Diodo e Triodo

C.1 Elementos WDF

```
#include "WDFElement.h"

WDFElement::WDFElement(std::string id){
  a = 0;
  b = 0;
  r = 0;
  g = 0;
}

WDFElement::~WDFElement(){
}

double WDFElement::getB(){
  return b;
}

double WDFElement::getA(){
  return a;
}

double WDFElement::getVoltage(){
  return (a + b) / 2.0;
}

double WDFElement::getCurrent(){
  return (a   b) / 2.0 * r;
}

void WDFElement::reset(){
}

double WDFElement::getResistance(){
  return r;
}

```

```
36 double WDFElement::getImpedance(){
37   return g;
38 }
39
40 std::string WDFElement::getId(){
41   return id;
42 }
```

```
1 #ifndef WDFELEMENT_H
2 #define WDFELEMENT_H
3 #include <string>
4
5 class WDFElement {
6 public:
7   WDFElement(std::string id);
8   virtual ~WDFElement();
9
10  double getB();
11  double getA();
12  double getVoltage();
13  double getCurrent();
14
15  virtual void reset();
16  virtual void calculateA(double a) = 0;
17  virtual void calculateB() = 0;
18  double getResistance();
19  double getImpedance();
20  std::string getId();
21
22 protected:
23  double r, g;
24  double b, a;
25  std::string id;
26 };
27 #endif
```

```
1 #include "Adaptor.h"
2
3 Adaptor::Adaptor(std::string id, WDFElement* left, WDFElement* right,
      double r) : WDFElement(id){
4   this->left = left;
5   this->right = right;
6   this->r = r;
7   this->g = 1 / r;
8 }
9
10 WDFElement* Adaptor::getLeft(){
11   return left;
12 }
13 WDFElement* Adaptor::getRight(){
14   return right;
15 }
16
17 void Adaptor::reset(){
18   left->reset();
```

```
  right->reset();
}

Adaptor::~Adaptor()
{
  delete left;
  delete right;
}
```

```
#pragma once
#include "WDFElement.h"
class Adaptor :
  public WDFElement
{
public:
  Adaptor(std::string id, WDFElement* left, WDFElement* right, double r);
  WDFElement* getLeft();
  WDFElement* getRight();
  void reset() override final;
  virtual ~Adaptor();
protected:
  WDFElement *left, *right;
};
```

```
#include "ParallelAdaptor.h"

void ParallelAdaptor::calculateA(double a){
  this->a = a;
  left->calculateA(a + b   left->getB());
  right->calculateA(a + b   right->getB());
}

ParallelAdaptor::ParallelAdaptor(std::string id, WDFElement* left,
    WDFElement* right)
  : Adaptor(id, left, right,
    1/(left->getImpedance() + right->getImpedance())){
}

void ParallelAdaptor::calculateB(){
  left->calculateB();
  right->calculateB();
  double leftv = (left->getImpedance() / g) * left->getB();
  double rightv = (right->getImpedance() / g) * right->getB();
  this->b = leftv + rightv;
}
```

```
#pragma once
#include "Adaptor.h"
class ParallelAdaptor : public Adaptor{
public:
  void calculateA(double a) override;
  ParallelAdaptor(std::string id, WDFElement* left, WDFElement* right);
  void calculateB() override;
private:
  double an;
```

```
};
```

```
#include "SeriesAdaptor.h"


SeriesAdaptor::SeriesAdaptor(std::string id, WDFElement* left, WDFElement*
    right) :
  Adaptor(id, left, right, left->getResistance() + right->getResistance())
     {}

void SeriesAdaptor::calculateA(double a){
  double leftB = left->getB();
  double rightB = right->getB();
  double leftA = leftB   (left->getResistance() / r) * (a   b);
  double rightA = rightB   (right->getResistance() / r) * (a   b);
  left->calculateA(leftA);
  right->calculateA(rightA);
  this->a = a;
}

void SeriesAdaptor::calculateB(){
  left->calculateB();
  right->calculateB();
  double lb = left->getB();
  double rb = right->getB();
  this->b =  (lb + rb);
}
```

```
#pragma once
#include "Adaptor.h"

class SeriesAdaptor : public Adaptor{
public:
  SeriesAdaptor(std::string id, WDFElement* left, WDFElement* right);
  void calculateA(double a) override;
  void calculateB() override;
};
```

```
#include "Capacitor.h"

Capacitor::Capacitor(std::string id, double C, double sampleRate) :
    WDFElement(id){
  this->r = sampleRate / 2.0 * C;
  this->g = 1 / r;
  z = 0;
  this->sampleRate = sampleRate;
  this->C = C;
}

void Capacitor::calculateA(double a){
  z = a;
  this->a = a;
}

void Capacitor::calculateB(){
```

```
  b = z;
}

void Capacitor::reset(){
  this->z = 0;
}
```

```
#pragma once
#include "WDFElement.h"

class Capacitor : public WDFElement{

public:
  Capacitor(std::string id, double C, double sampleRate);
  void calculateA(double a) override;
  void calculateB() override;
  void reset() override;
private:
  double z;
  double sampleRate;
  double C;

};
```

```
#include "Resistor.h"

Resistor::Resistor(std::string id, double R) : WDFElement(id){
  this->r = R;
  this->g = 1 / r;
}

void Resistor::calculateA(double a){
  this->a = a;
}

void Resistor::calculateB(){
  this->b = 0;
}
```

```
#pragma once
#include "WDFElement.h"

class Resistor : public WDFElement{

public:
  Resistor(std::string id, double R);
  void calculateA(double a) override;
  void calculateB() override;
};
```

```
#include "VoltageSource.h"

VoltageSource::VoltageSource(std::string id, double R) : WDFElement(id){
  this->r = R;
  this->g = 1 / r;
```

```
  inputVoltage = 0;
  a = 0;
  b = 0;
}

void VoltageSource::calculateA(double a){
  this->a = a;
}

void VoltageSource::setInput(double i){
  this->inputVoltage = i;
}

double VoltageSource::getInput(){
  return inputVoltage;
}

void VoltageSource::calculateB(){
  b = inputVoltage;
}
```

```
#pragma once
#include "WDFElement.h"

class VoltageSource : public WDFElement{
public:
  VoltageSource(std::string id, double R);
  void calculateA(double a) override;
  void calculateB() override;
  void setInput(double i);
  double getInput();
private:
  double inputVoltage;
};
```

```
#include "WDFRoot.h"


WDFElement* WDFRoot::getRoot(){
  return root;
}

WDFRoot::WDFRoot(std::string id, WDFElement* root) : WDFElement(id){
  this->root = root;
}


WDFRoot::~WDFRoot(){
}
```

```
#pragma once
#include "DSP.h"
#include "WDFElement.h"

class WDFRoot :
```

```
  public WDFElement
{
public:
  WDFRoot(std::string id, WDFElement* root);
  WDFElement* getRoot();
  virtual ~WDFRoot();
protected:
  WDFElement* root;
};
```

```
#include "DiodeS.h"

DiodeS::DiodeS(std::string id, WDFElement* root, double sampleRate) :
    WDFRoot(id, root){
  vt = 0.036;
  Is = 125.56;

  lastv = 0;
  da = 0;
  b = 0;
}

void DiodeS::calculateA(double a){
  root->calculateA(r * b);

  lastv = root->getVoltage();
}

void DiodeS::calculateB(){
  root->calculateB();
  b = root->getB();
  rp = Is * exp( vt * lastv);
  r = (rp   root->getResistance())
    / (rp + root->getResistance());
}

DiodeS::~DiodeS()
{
}
```

```
#pragma once
#include "WDFRoot.h"
#include "ParallelAdaptor.h"
#include "SeriesAdaptor.h"
#include "Capacitor.h"
#include "Resistor.h"
#include "VoltageSource.h"

class DiodeS :
  public WDFRoot
{
public:
  DiodeS(std::string id, WDFElement *root, double sampleRate);
  void calculateA(double a) override;
  void calculateB() override;
```

```
  virtual ~DiodeS();
private:
  double da, lastv, newv, vt, rp, Is, b, r;
};
```

```
#include "TriodeValve.h"

void TriodeValve::calculateA(double a){
  root>calculateA(b);
}

double TriodeValve::Id(double vpk){
  double func = f(vpk);
  if (func > 0){
    sgn = 1;
  }
  else{
    sgn = 0;
  }
  double val = vpk + root>getResistance()*func   a;
  return val;
}

double TriodeValve::did(double vpk){
  if (sgn == 1){
    return (2 * kx*root>getResistance()*(log(exp(kp*(1 / mi + vgk / pow((
        vpk*vpk + kvb), (1 / 2)))) + 1) / kp   (pow(vgk*vpk, 2) * exp(kp*(1
        / mi + vgk / pow((vpk * vpk + kvb), (1 / 2))))) / ((exp(kp*(1 / mi +
         vgk / pow((vpk * vpk + kvb), (1 / 2)))) + 1)*pow((vpk * vpk + kvb),
         (3 / 2))))*pow(((vpk*log(exp(kp*(1 / mi + vgk / pow((vpk * vpk +
        kvb), (1 / 2)))) + 1)) / kp), (kx   1))) / kg1 + 1;
  }
  return 1;
}

double TriodeValve::f(double vpk){
  double e2 = kp*(1 / mi + vgk / sqrt(kvb + vpk*vpk));
  double expr = log(1 + exp(e2));
  double e1 = (vpk / kp)*expr;
  if (e1 >= 0){
    return 2 * (pow(e1, kx) / kg1);
  }
  return 0;
}

double TriodeValve::newton(double x, int interacoes, double epsilon){
  double dx = 0;
  double xx = 0;
  double vdid = 0, vid = 0;
  int k = 0;
  double nx = x;
  while ((abs(xx   nx) > epsilon) &&
    (k < interacoes) &&
    (Id(nx) != 0))
  {
    vid = Id(nx);
```

```
47      vdid = did(nx);
48      dx = vid / vdid;
49      xx = nx;
50      nx = nx   dx;
51      k++;
52    }
53    return nx;
54  }
55
56  void TriodeValve::calculateB(){
57    root->calculateB();
58    a = root->getB();
59
60    vpk = newton(vpk, 20, 1e-5);
61    b = vpk   root->getResistance()*f(vpk);
62  }
63
64  TriodeValve::TriodeValve(std::string id, WDFElement* root, double
        sampleRate, int samplesPerBlock):
65  WDFRoot(id, root){
66    mi = 100;
67    kx = 1.4;
68    kg1 = 1060;
69    kp = 600;
70    kvb = 300;
71    b = 0;
72    a = 0;
73    vgk = -0.5;
74    vpk = 250.0;
75    vg = 0;
76    flag = false;
77  }
78
79  void TriodeValve::setVGK(double vgk){
80    this->vgk = vgk;
81  }
82
83  TriodeValve::~TriodeValve()
84  {
85  }
```

```
1  #pragma once
2  #include "WDFRoot.h"
3  class TriodeValve :
4    public WDFRoot
5  {
6  public:
7    void calculateA(double a) override;
8    void calculateB() override;
9    double Id(double vpk);
10   double did(double vpk);
11   double f(double vpk);
12   double newton(double vpk, int interacoes, double epsilon);
13   TriodeValve(std::string id, WDFElement* root, double sampleRate, int
         samplesPerBlock);
14   void setVGK(double vgk);
```

```
  virtual ~TriodeValve();
protected:
  double nvgk, vgk, nvk, nvpk, vpk, vg;
  double mi, kx, kg1, kp, kvb;
  double sgn;
  double state, v, out;
  bool flag;
};
```

C.2 Circuitos WDF

```
#include "Triode.h"

void Triode::mouseDown(int w, int h, Component* c){

}

Triode::Triode(std::string id, double sampleRate, int samplesPerBlock) :
    DSP(id, sampleRate, samplesPerBlock){
  r0 = new Resistor("r0", 1e6);
  rk = new Resistor("rk", 1e3);
  vs = new VoltageSource("vs",100e3);
  c0 = new Capacitor("c0", 10e 9, sampleRate);
  ck = new Capacitor("ck", 10e 6, sampleRate);
  s2 = new SeriesAdaptor("s2", c0, r0);
  p2 = new ParallelAdaptor("p2", vs, s2);
  p1 = new ParallelAdaptor("p1", ck, rk);
  s1 = new SeriesAdaptor("s1", p2, p1);
  //pt = new ParallelAdaptor("pt", p2, p1);
  triodeV = new TriodeValve("triodo", s1, sampleRate, samplesPerBlock);

  state = 0;
}

double Triode::calculate(double input, double sampleRate){
  vs->setInput(250);

  double vrk = rk->getVoltage();
  double vgk = input   vrk;
  triodeV->setVGK(vgk);

  triodeV->calculateB();
  triodeV->calculateA(triodeV->getB());

  v = r0->getVoltage();
  //v = (triodeV->getA() + triodeV->getB()) / 2;
  return v / 250 + 0.7;
}

Triode::~Triode(){
}
```

```
#pragma once
```

```
#include "ParallelAdaptor.h"
#include "SeriesAdaptor.h"
#include "Capacitor.h"
#include "VoltageSource.h"
#include "Resistor.h"
#include "Events.h"
#include "DSP.h"
#include "TriodeValve.h"

class Triode :
  public DSP, Events
{
public:
  void mouseDown(int w, int h, Component* c) override;
  Triode(std::string id, double sampleRate, int samplesPerBlock);
  ~Triode();
  double calculate(double input, double sampleRate) override;
private:

  ParallelAdaptor *p1, *p2, *pt;
  SeriesAdaptor *s1, *s2;
  Capacitor *c0, *ck;
  VoltageSource *vs;
  Resistor *r0, *rk;
  TriodeValve *triodeV;
  double nvgk, vgk, nvk, nvpk, vpk, vg;
  double state, v,out;
  double gain;
};
```

```
#include "DiodeClipper.h"
#include <stdio.h>
#include <cmath>

DiodeClipper::DiodeClipper(std::string id, double sampleRate, int
    samplesPerBlock) : DSP(id, sampleRate, samplesPerBlock){
  vs = new VoltageSource("vs", 2.2e3);
  c = new Capacitor("c", 0.01e 6, sampleRate);
  c2 = new Capacitor("c2", 0.47e 6, sampleRate);
  s = new SeriesAdaptor("s", c2, vs);
  p = new ParallelAdaptor("p", s, c);
  root = p;

  vt = 45.3e 3;
  Is = 2.52e 9;
  lastb = 0.1;
  lastv = 0;
  gain = 0;
}


DiodeClipper::~DiodeClipper(){
  delete p;
}

double DiodeClipper::calculate(double input, double sampleRate){
```

```
  double di, ddi;
  double incoming, reflected;
  double voltage, r;

  vs->setInput(4e6*input);

  root->calculateB();
  incoming = root->getB();

  rp = root->getResistance();
  lastb = newton(incoming, lastb, 20, 1e-5);

  root->calculateA(lastb);
  lastv = (incoming + lastb) /2;

  return lastv/5;
}

double DiodeClipper::Id(double a, double b){
  double sin = sinh((a + b) / (2 * vt));
  double dw = 2 * Is*sin   ((a   b) / (2 * rp));
  return dw;
}

double DiodeClipper::dId(double a, double b){
  double cossine = cosh((a + b) / (2 * vt));
  double ddw =  1 / (2*rp) + (Is * cossine) / vt;
  return ddw;
}

double DiodeClipper::newton(double a, double lastb, int interacoes, double
    epsilon){
  double b, newb, div, id;
  int i = 0;
  b = 0;
  if (lastb > epsilon)
    newb = lastb;
  else{
    newb = 0.1;
  }
  while (abs(newb   b) > epsilon
      && i < interacoes
      && Id(a, newb) != 0){
    b = newb;
    div = Id(a, b) / dId(a, b);
    newb = b   div;
    i++;
  }
  return newb;
}
```

```
#ifndef DIODECLIPPER_H
#define DIODECLIPPER_H

#include "ParallelAdaptor.h"
#include "SeriesAdaptor.h"
```

```
#include "Capacitor.h"
#include "VoltageSource.h"
#include "Resistor.h"
#include "DSP.h"

class DiodeClipper :
  public DSP
{
public:
  DiodeClipper(std::string id, double sampleRate, int samplesPerBlock);
  virtual ~DiodeClipper();
  double calculate(double input, double sampleRate) override;
private:
  WDFElement *root;
  VoltageSource *vs;
  ParallelAdaptor *p;
  SeriesAdaptor *s;
  Capacitor *c, *c2;
  Resistor *r1;
  double out;
  double vt, Is, rp;
  double lastb, lastv;
  double Id(double a, double b);
  double dId(double a, double b);
  double newton(double a, double lastb, int interacoes, double epsilon);
  double gain;
};

#endif
```

```
#include "SeriesClipper.h"
#include <math.h>


void SeriesClipper::mouseDown(int w, int h, Component* c){
  float height = c->getHeight();
  gain = h / (height);
  gain = gain*60;
}

SeriesClipper::SeriesClipper(std::string id, double sampleRate, int
    samplesPerBlock) : DSP(id, sampleRate, samplesPerBlock){
  vs = new VoltageSource("vs", 1.0);
  r1 = new Resistor("r1", 80.0);
  c1 = new Capacitor("c1", 3.5e-5, sampleRate);

  s2 = new SeriesAdaptor("s2", r1, c1);
  s1 = new SeriesAdaptor("s1", vs, s2);
  diode = new DiodeS("root", s1, sampleRate);

  gain = 0;
}

double SeriesClipper::calculate(double input, double sampleRate){
  vs->setInput(60 * input);

```

```
  diode->calculateB();
  diode->calculateA(diode->getB());

  return r1->getVoltage()/25;
}

SeriesClipper::~SeriesClipper(){
  delete vs;
}
```

```
#pragma once
#include "VoltageSource.h"
#include "Resistor.h"
#include "Capacitor.h"
#include "SeriesAdaptor.h"
#include "ParallelAdaptor.h"
#include "Events.h"
#include "DSP.h"
#include "DiodeS.h"

class SeriesClipper : public DSP, Events
{
public:
  void mouseDown(int w, int h, Component* c) override;
  SeriesClipper(std::string id, double sampleRate, int samplesPerBlock);
  ~SeriesClipper();
  double calculate(double input, double sampleRate) override;

private:
  VoltageSource *vs;
  Resistor *r1;
  Capacitor *c1, *c2;

  SeriesAdaptor *s1;
  SeriesAdaptor *s2;
  ParallelAdaptor *p;
  DiodeS *diode;
  double gain;
};
```

C.3 Implementação Tone Stack

```
#pragma once
class IToneStack
{
public:
  virtual int getAsize() = 0;
  virtual int getBsize() = 0;
  virtual void calculateCoefs() = 0;
  virtual void getACoefs(double *aCoefs) = 0;
  virtual void getBCoefs(double *bCoefs) = 0;
};
```

```cpp
#include "ToneStackFilter.h"

ToneStackFilter::ToneStackFilter(std::string id, double sampleRate, int
    numSamples, IToneStack *tone) : DSP(id, sampleRate, numSamples){
  setToneStack(tone);
}

double ToneStackFilter::calculate(double input, double sampleRate){
  toneStack->getACoefs(a);
  toneStack->getBCoefs(b);

  double suma = 0;
  double sumb = (b[0]/a[0])*input;
  for (int i = 1; i < sizea; i++){
    suma += (a[i]/a[0]) * (delaya->getDelayed(i - 1));
  }
  for (int i = 1; i < sizeb; i++){
    sumb += (b[i]/a[0]) * (delayb->getDelayed(i - 1));
  }
  float v0 = (sumb - suma);
  delayb->updateDelay(input);
  delaya->updateDelay(sumb - suma);

  return v0;
}

void ToneStackFilter::setToneStack(IToneStack* tone){
  sizea = tone->getAsize();
  sizeb = tone->getBsize();
  delaya = new Delay(sizea);
  delayb = new Delay(sizeb);
  a = (double*)calloc(sizea, sizeof(double));
  b = (double*)calloc(sizeb, sizeof(double));
  memset(a, 0, sizea*sizeof(double));
  memset(b, 0, sizeb*sizeof(double));
  tone->calculateCoefs();
  tone->getACoefs(a);
  tone->getBCoefs(b);

  toneStack = tone;
}

ToneStackFilter::~ToneStackFilter(){
}
```

```cpp
#pragma once
#include "DSP.h"
#include "IToneStack.h"
#include "Delay.h"
class ToneStackFilter :
  public DSP
{
protected:
  double *a, *b;
  Delay* delaya, *delayb;
```

```
  int sizea, sizeb;
  IToneStack *toneStack;
public:
  ToneStackFilter(std::string id, double sampleRate, int numSamples,
      IToneStack *tone);
  double calculate(double input, double sampleRate) override;
  void setToneStack(IToneStack *tone);
  virtual ~ToneStackFilter();
};
```

```
#include "Bassman.h"


Bassman::Bassman(double sampleRate){
  sr = sampleRate;
  ic = 2 * sr;
  iC1 = 0.25e 9;
  iC2 = 20e 9;
  iC3 = iC2;
  iR1 = 250e3;
  iR2 = 1e6;
  iR3 = 25e3;
  iR4 = 56e3;
  it = 0;
  im = 0;
  il = 0;
}

int Bassman::getAsize(){
  return 4;
}

int Bassman::getBsize(){
  return 4;
}

void Bassman::getACoefs(double* aCoefs){
  aCoefs[0] = iA0;
  aCoefs[1] = iA1;
  aCoefs[2] = iA2;
  aCoefs[3] = iA3;
}

void Bassman::getBCoefs(double* bCoefs){
  bCoefs[0] = iB0;
  bCoefs[1] = iB1;
  bCoefs[2] = iB2;
  bCoefs[3] = iB3;
}

void Bassman::calculateCoefs(){
  ib1 = it*iC1*iR1 + im*iC3*iR3 + il*(iC1*iR2 + iC2*iR2) + (iC1*iR3 + iC2*
      iR3);

  ib2 = it*(iC1*iC2*iR1*iR4 + iC1*iC3*iR1*iR4)   (im*im)*(iC1*iC3*(iR3*iR3)
       + iC2*iC3*(iR3*iR3))
```

```
45      + im*(iC1*iC3*iR1*iR3 + iC1*iC3*(iR3*iR3) + iC2*iC3*(iR3*iR3))
46      + il*(iC1*iC2*iR1*iR2 + iC1*iC2*iR2*iR4 + iC1*iC3*iR2*iR4) + il*im*(iC1
            *iC3*iR2*iR3
47      + iC2*iC3*iR2*iR3) + (iC1*iC2*iR1*iR3 + iC1*iC2*iR3*iR4 + iC1*iC3*iR3*
            iR4);
48
49    ib3 = il*im*(iC1*iC2*iC3*iR1*iR2*iR3 + iC1*iC2*iC3*iR2*iR3*iR4)   (im*im)
          *(iC1*iC2*iC3*iR1*(iR3*iR3)
50      + iC1*iC2*iC3*(iR3*iR3)*iR4) + im*(iC1*iC2*iC3*iR1*(iR3*iR3) + iC1*iC2*
            iC3*(iR3*iR3)*iR4)
51      + it*iC1*iC2*iC3*iR1*iR3*iR4   it*im*iC1*iC2*iC3*iR1*iR3*iR4 + it*il*
            iC1*iC2*iC3*iR1*iR2*iR4;
52
53    ia0 = 1;
54
55    ia1 = (iC1*iR1 + iC1*iR3 + iC2*iR3 + iC2*iR4 + iC3*iR4) + im*iC3*iR3 + il
          *(iC1*iR2 + iC2*iR2);
56
57    ia2 = im*(iC1*iC3*iR1*iR3   iC2*iC3*iR3*iR4 + iC1*iC3*(iR3*iR3) + iC2*iC3
          *(iR3*iR3))
58      + il*im*(iC1*iC3*iR2*iR3 + iC2*iC3*iR2*iR3)   (im*im)*(iC1*iC3*(iR3*iR3
            ) + iC2*iC3*(iR3*iR3))
59      + il*(iC1*iC2*iR2*iR4 + iC1*iC2*iR1*iR2 + iC1*iC3*iR2*iR4 + iC2*iC3*iR2
            *iR4)
60      + (iC1*iC2*iR1*iR4 + iC1*iC3*iR1*iR4 + iC1*iC2*iR3*iR4 + iC1*iC2*iR1*
            iR3
61      + iC1*iC3*iR3*iR4 + iC2*iC3*iR3*iR4);
62
63    ia3 = il*im*(iC1*iC2*iC3*iR1*iR2*iR3 + iC1*iC2*iC3*iR2*iR3*iR4)   (im*im)
          *(iC1*iC2*iC3*iR1*(iR3*iR3)
64      + iC1*iC2*iC3*(iR3*iR3)*iR4) + im*(iC1*iC2*iC3*iR4*(iR3*iR3) + iC1*iC2*
            iC3*(iR3*iR3)*iR1
65        iC1*iC2*iC3*iR1*iR3*iR4) + il*iC1*iC2*iC3*iR1*iR2*iR4 + iC1*iC2*iC3*
            iR1*iR3*iR4;
66    /*                                                    */
67    iB0 =  ib1*ic   ib2*(ic*ic)   ib3*(pow(ic, 3));
68
69    iB1 =  ib1*ic + ib2*(ic*ic) + 3 * ib3*(pow(ic, 3));
70
71    iB2 = ib1*ic + ib2*(ic*ic)   3 * ib3*(pow(ic, 3));
72
73    iB3 = ib1*ic   ib2*(ic*ic) + ib3*(pow(ic, 3));
74
75    iA0 =  ia0   ia1*ic   ia2*(ic*ic)   ia3*(pow(ic, 3));
76
77    iA1 =  3 * ia0   ia1*ic + ia2*(ic*ic) + 3 * ia3*(pow(ic, 3));
78
79    iA2 =  3 * ia0 + ia1*ic + ia2*(ic*ic)   3 * ia3*(pow(ic, 3));
80
81    iA3 =  ia0 + ia1*ic   ia2*(ic*ic) + ia3*(pow(ic, 3));
82  }
83
84  void Bassman::setMiddle(double middle){
85    this >im = middle;
86    calculateCoefs();
87  }
```

```
88
89 void Bassman::setTreble(double treble){
90   this->it = treble;
91   calculateCoefs();
92 }
93
94 void Bassman::setBass(double bass){
95   il = bass;
96   calculateCoefs();
97 }
98
99 Bassman::~Bassman()
100 {
101 }
```

```
1  #pragma once
2  #include "IToneStack.h"
3  #include <math.h>
4  class Bassman :
5    public IToneStack
6  {
7  public:
8    Bassman(double sampleRate);
9    int getAsize() override;
10   int getBsize() override;
11   void getACoefs(double* aCoefs) override;
12   void getBCoefs(double* bCoefs) override;
13   void calculateCoefs() override;
14   void setBass(double bass);
15   void setMiddle(double middle);
16   void setTreble(double treble);
17   virtual ~Bassman();
18 private:
19   double ic;
20   double iC1;
21   double iC2;
22   double iC3;
23   double iR1;
24   double iR2;
25   double iR3;
26   double iR4;
27   double sr;
28   double it, im, il;
29   double ib1, ib2, ib3, ia0, ia1, ia2, ia3;
30   double iB0, iB1, iB2, iB3, iA0, iA1, iA2, iA3;
31 };
```

C.4 Implementação Interface Gráfica

```
1 #include "Plot.h"
2
3
4 Plot::Plot(Component* c, int bufferSize){
```

```
  this->c = c;
  this->bufferSize = bufferSize;
}

void Plot::draw(Graphics& g, float* buffer){
  float centerX = c->getWidth() / 2;
  float centerY = c->getHeight() / 2;
  // draw a representative sinewave
  g.drawHorizontalLine(centerY, 0, c->getWidth());
  Path wavePath;
  wavePath.startNewSubPath(0, centerY);

  float x = 1;
  int ind = 0;
  while (ind < bufferSize){
    wavePath.lineTo(x, centerY   600 * buffer[ind]);
    x += (c->getWidth() / (bufferSize))*2;
    ind++;
  }

  g.setColour(Colours::grey);
  g.strokePath(wavePath, PathStrokeType(2.0f));
}

Plot::~Plot(){
}
```

```
#pragma once
#include "../JuceLibraryCode/JuceHeader.h"
#include "juce_graphics/geometry/juce_PathStrokeType.h"

class Plot
{
public:
  Plot(Component* c, int bufferSize);
  void draw(Graphics& g, float* buffer);
  virtual ~Plot();
private:
  Component* c;
  int bufferSize;
};
```

```
#include "ToneSlider.h"

ToneSlider::ToneSlider(Component* c){
  this->c = c;
  x = c->getWidth()    WIDTH;
  y = 0;
}

void ToneSlider::addSlider(Slider* s){
  s->setBounds(x, y, WIDTH, HEIGHT);
  y += HEIGHT;
  c->addAndMakeVisible(s);
}
```

```
#pragma once
#include "../JuceLibraryCode/JuceHeader.h"

#define WIDTH 200
#define HEIGHT 20
#include <list>

class ToneSlider : public Slider::Listener{
public:
  ToneSlider(Component* c);
  void addSlider(Slider* s);
private:
  int x, y;
  std::list<Slider*> sliders;
protected:
  Component* c;
};
```

```
#include "BassmanStack.h"


BassmanStack::BassmanStack(Component *c, Bassman *bassman) : ToneSlider(c){
  this->c = c;
  this->bassman = bassman;
  addSlider(&b);
  addSlider(&m);
  addSlider(&t);
  b.setRange(0, 1);
  b.addListener(this);
  b.setValue(0.5);
  m.setRange(0, 1);
  m.addListener(this);
  m.setValue(0.5);
  t.setRange(0, 1);
  t.addListener(this);
  t.setValue(0.5);
}

void BassmanStack::sliderValueChanged(Slider* slider){
  if (slider == &b)
    bassman->setBass(b.getValue());
  if (slider == &m)
    bassman->setMiddle(m.getValue());
  if (slider == &t)
    bassman->setTreble(t.getValue());
  c->repaint();
}

BassmanStack::~BassmanStack()
{
}
```

```
#pragma once
#include "ToneSlider.h"
#include "Bassman.h"
```

```
class BassmanStack :
  public ToneSlider
{
public:
  void sliderValueChanged(Slider* slider) override;
  BassmanStack(Component* c, Bassman* bassman);
  virtual ~BassmanStack();
private:
  Slider b, m, t;
  Bassman *bassman;
};
```

C.5 Entradas

```
#include "Sin.h"

void Sin::setInput(float* buffer){
  for (int i = 0; i < numSample; i++){
    buffer[i] = sin(2 * M_PI*frequency*t);
    t += 1 / sampleRate;
  }
  if (t > sampleRate){
    t = 0;
  }
}

Sin::Sin(double freq, double sampleRate, int numSamples){
  frequency = freq;
  t = 0;
  gain = 0;
  this>numSample = numSamples;
  this>sampleRate = sampleRate;
}

void Sin::reset(){
  t = 0;
}

Sin::~Sin()
{
}
```

```
#pragma once
#include "Input.h"
#define _USE_MATH_DEFINES
#include <math.h>

class Sin {
public:
  void setInput(float* buffer);
  Sin(double freq, double sampleRate, int numSamples);
  void reset();
  virtual ~Sin();
```

```
private:
  double frequency;
  double t;
  double gain;
  double sampleRate;
  int numSample;
};
```

```
#include "FileReader.h"


void FileReader::setInput(double* buffer){
}

void FileReader::mouseDown(int w, int h, Component* c){
}

void FileReader::changeListenerCallback(ChangeBroadcaster* source){
  if (!transportSource.isPlaying()){
    transportSource.setPosition(0.0);
    transportSource.start();
  }
}

void FileReader::setInput(float* const* bufferToUse){
  transportSource.getNextAudioBlock(*source);
}
void FileReader::init(){
  FileChooser chooser("Select a Wave file to play...",
    File::nonexistent,
    "*.wav");                                          // [7]

  if (chooser.browseForFileToOpen())                                          //
     [8]
  {
    File file(chooser.getResult());                                          // [9]
    AudioFormatReader* reader = formatManager.createReaderFor(file); //
       [10]

    if (reader != nullptr)
    {
      readerSource = new AudioFormatReaderSource(reader, true);
                        // [11]
      transportSource.setSource(readerSource, 0, nullptr, reader>
         sampleRate); // [12]
      transportSource.start();
    }
  }
}

void FileReader::prepareToPlay(double sampleRate, int numSample){
  this>sampleRate = sampleRate;
  this>numSample = numSample;
}

```

```
44 FileReader::FileReader(float* const* bufferToUse, double sampleRate, int
       numSamples) : Input(sampleRate ,numSamples){
45   formatManager.registerBasicFormats();
46   transportSource.addChangeListener(this);
47   this->sampleRate = sampleRate;
48   this->numSample = numSamples;
49   ai = new AudioInterpreter(sampleRate, numSamples);
50   buffer = new juce::AudioSampleBuffer(bufferToUse, 1, numSamples);
51   source = new juce::AudioSourceChannelInfo(buffer, 0, numSamples);
52   transportSource.prepareToPlay(numSample, sampleRate);
53 }
54
55
56
57
58 FileReader::~FileReader(){
59   delete &transportSource;
60   delete &readerSource;
61   delete &formatManager;
62   delete buffer;
63   delete source;
64 }
```

```
1 #pragma once
2 #include "Input.h"
3 #include "AudioInterpreter.h"
4 #include "Events.h"
5
6 class FileReader :
7   public Input, ChangeListener, Events
8 {
9 public:
10   void setInput(double* buffer) override;
11   void mouseDown(int w, int h, Component* c) override;
12   void changeListenerCallback(ChangeBroadcaster* source) override;
13   void setInput(float* const* bufferToUse);
14   void init() override;
15   void prepareToPlay(double sampleRate, int numSamples);
16   FileReader(float* const* bufferToUse, double sampleRate, int numSamples);
17   virtual ~FileReader();
18 private:
19   AudioInterpreter *ai;
20   AudioFormatManager formatManager;
21   ScopedPointer<AudioFormatReaderSource> readerSource;
22   AudioTransportSource transportSource;
23   juce::AudioSampleBuffer *buffer;
24   juce::AudioSourceChannelInfo *source;
25 };
```

C.6 JUCE e Tratamento de *Buffers*

```
1 #include "AudioInterpreter.h"
2
```

```
3
4  AudioInterpreter::AudioInterpreter(double sampleRate, int samplePerBlock){
5    this->sampleRate = sampleRate;
6    this->numSamples = samplePerBlock;
7    gain = 1;
8    volume = 1;
9  }
10
11 void AudioInterpreter::processAudioBlock(float* input, const
       AudioSourceChannelInfo& bufferToFill, float* ret){
12   float **chanInfo = (float**)calloc(bufferToFill.buffer->getNumChannels(),
         sizeof(float*));
13   for (int i = 0; i < bufferToFill.buffer->getNumChannels(); i++){
14     chanInfo[i] = bufferToFill.buffer->getWritePointer(i, bufferToFill.
           startSample);
15   }
16   for (int i = 0; i < bufferToFill.numSamples; i++){
17     float result = calculateEffect(input[i]*gain, sampleRate);
18     for (int j = 0; j < bufferToFill.buffer->getNumChannels(); j++){
19       chanInfo[j][i] = result*volume;
20       ret[i] = result*volume;
21     }
22   }
23 }
24
25 void AudioInterpreter::addeffect(DSP* effect){
26   effects.push_back(effect);
27 }
28
29 void AudioInterpreter::getAudioBlock(const AudioSourceChannelInfo&
       bufferToFill, double* ret){
30   const float *chanInfo;
31   chanInfo = bufferToFill.buffer->getReadPointer(0, bufferToFill.
         startSample);
32   for (int i = 0; i < bufferToFill.numSamples; i++){
33       ret[i] = chanInfo[i];
34   }
35 }
36
37 void AudioInterpreter::setGain(double gain){
38   this->gain = gain;
39 }
40
41 void AudioInterpreter::setVolume(double volume){
42   this->volume = volume;
43 }
44
45 void AudioInterpreter::setToneStack(DSP* tone){
46   this->tone = tone;
47 }
48
49 void AudioInterpreter::setWDF(DSP* wdf){
50   this->wdf = wdf;
51 }
52
53 double AudioInterpreter::calculateEffect(double input, double sampleRate){
```

```
54    float result = input;
55    /*for (std::list<DSP*>::iterator it = effects.begin(); it != effects.end
          (); ++it){
56      result = (*it)->calculate(result, sampleRate);
57    }*/
58    result = wdf->calculate(result, sampleRate);
59    result = tone->calculate(result, sampleRate);
60
61    return result;
62  }
63
64  AudioInterpreter::~AudioInterpreter()
65  {
66  }
```

```
1  #pragma once
2  #include "DSP.h"
3  #include <list>
4  #include "IToneStack.h"
5
6  class AudioInterpreter
7  {
8  public:
9    AudioInterpreter(double sampleRate, int samplePerBlock);
10   void processAudioBlock(float *input, const AudioSourceChannelInfo&
         bufferToFill, float* ret);
11   void addeffect(DSP *effect);
12   void getAudioBlock(const AudioSourceChannelInfo& bufferToFill, double*
         ret);
13   void setGain(double gain);
14   void setVolume(double volume);
15   void setToneStack(DSP *tone);
16   void setWDF(DSP* wdf);
17   virtual ~AudioInterpreter();
18 private:
19   std::list<DSP*> effects;
20   double calculateEffect(double input, double sampleRate);
21   double sampleRate, numSamples;
22   double gain, volume;
23   DSP *tone;
24   DSP *wdf;
25 };
```

```
1  #include "DSP.h"
2
3  DSP::~DSP(){
4  }
5
6  DSP::DSP(std::string id, double sampleRate, int samplesPerBlock){
7    this->sampleRate = sampleRate;
8    this->samplesPerBlock = samplesPerBlock;
9    this->id = id;
10 }
```

```
1  #ifndef DSP_H
```

```
#define DSP_H

#include <stdlib.h>
#include "../JuceLibraryCode/JuceHeader.h"

class DSP{
public:
  DSP(std::string id, double sampleRate, int samplePerBlock);
  virtual ~DSP();
  virtual double calculate(double input, double sampleRate) = 0;

protected:
  double sampleRate;
  int samplesPerBlock;
  std::string id;

private:
  float gain;
};

#endif
```

```
#include "Delay.h"

Delay::Delay(int order){
  delay = (double*)calloc(order, sizeof(double));
  memset(delay, 0, sizeof(double)*order);
  this>order = order;
}

void Delay::updateDelay(double input){
  for (int i = order   1; i > 0; i  ){
    delay[i] = delay[i   1];
  }
  delay[0] = input;
}

double Delay::getDelayed(int d){
  return delay[d];

}

Delay::~Delay(){
}
```

```
#pragma once
#include "DSP.h"
class Delay
{
public:
  Delay(int order);
  void updateDelay(double input);
  double getDelayed(int d);
  virtual ~Delay();
private:
```

```
  double *delay;
  int order;
};
```

```
#pragma once
#include "../JuceLibraryCode/JuceHeader.h"
class Events
{
public:
  virtual ~Events(){

  }
  virtual void mouseDown(int w, int h, Component *c) = 0;
};
```

```
/*
  ==============================================================================

    This file was auto generated by the Introjucer!

    It contains the basic startup code for a Juce application.

  ==============================================================================
*/

#include "../JuceLibraryCode/JuceHeader.h"

Component* createMainContentComponent();

//==============================================================================
class WavefilterCreatorApplication  : public JUCEApplication
{
public:
    //==============================================================================
    WavefilterCreatorApplication() {}

    const String getApplicationName() override       { return ProjectInfo::
        projectName; }
    const String getApplicationVersion() override    { return ProjectInfo::
        versionString; }
    bool moreThanOneInstanceAllowed() override       { return true; }

    //==============================================================================
    void initialise (const String& commandLine) override
    {
        // This method is where you should put your application's
           initialisation code..
```

```
30
31          mainWindow = new MainWindow (getApplicationName());
32      }
33
34      void shutdown() override
35      {
36          // Add your application's shutdown code here..
37
38          mainWindow = nullptr; // (deletes our window)
39      }
40
41      //==============================================================================
42      void systemRequestedQuit() override
43      {
44          // This is called when the app is being asked to quit: you can
                ignore this
45          // request and let the app carry on running, or call quit() to
                allow the app to close.
46          quit();
47      }
48
49      void anotherInstanceStarted (const String& commandLine) override
50      {
51          // When another instance of the app is launched while this one is
                running,
52          // this method is invoked, and the commandLine parameter tells you
                what
53          // the other instance's command line arguments were.
54      }
55
56      //==============================================================================
57      /*
58          This class implements the desktop window that contains an instance
                of
59          our MainContentComponent class.
60      */
61      class MainWindow    : public DocumentWindow
62      {
63      public:
64          MainWindow (String name)  : DocumentWindow (name,
65                                                      Colours::lightgrey,
66                                                      DocumentWindow::
                                                            allButtons)
67          {
68              setUsingNativeTitleBar (true);
69              setContentOwned (createMainContentComponent(), true);
70              setResizable (true, true);
71
72              centreWithSize (getWidth(), getHeight());
73              setVisible (true);
74          }
75
```

```
76          void closeButtonPressed() override
77          {
78              // This is called when the user tries to close this window. Here, we'll just
79              // ask the app to quit when this happens, but you can change this to do
80              // whatever you need.
81              JUCEApplication::getInstance()->systemRequestedQuit();
82          }
83
84          /* Note: Be careful if you override any DocumentWindow methods the base
85             class uses a lot of them, so by overriding you might break its functionality.
86             It's best to do all your work in your content component instead, but if
87             you really have to override any DocumentWindow methods, make sure your
88             subclass also calls the superclass's method.
89          */
90
91      private:
92          JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainWindow)
93      };
94
95  private:
96      ScopedPointer<MainWindow> mainWindow;
97  };
98
99  //==============================================================================
100 // This macro generates the main() routine that launches the app.
101 START_JUCE_APPLICATION (WavefilterCreatorApplication)
```

```
1  /*
2    ==============================================================================
3
4      This file was auto generated!
5
6    ==============================================================================
7  */
8
9  #ifndef MAINCOMPONENT_H_INCLUDED
10 #define MAINCOMPONENT_H_INCLUDED
11
12 #define _USE_MATH_DEFINES
13
14 #include "../JuceLibraryCode/JuceHeader.h"
15 #include "SeriesClipper.h"
16 #include "Plot.h"
17 #include "Triode.h"
18 #include "ToneStackFilter.h"
19 #include "Bassman.h"
```

```
20 #include "SimpleLoP.h"
21 #include "AudioInterpreter.h"
22 #include "Sin.h"
23 #include "FileReader.h"
24 #include "BassmanStack.h"
25 #include "DiodeClipper.h"
26 #include "BassmanWDF.h"
27
28 //==============================================================================
29 /*
30     This component lives inside our window, and this is where you should put all
31     your controls and content.
32 */
33 class MainContentComponent   : public AudioAppComponent,
34                                public Slider::Listener,
35                                public Button::Listener
36 {
37 public:
38   void buttonClicked(Button* b) override{
39     if (b == &dio)
40       processor->setWDF(dc);
41     if (b == &series)
42       processor->setWDF(sc);
43     if (b == &trio)
44       processor->setWDF(triode);
45     if (b == &bassmanB)
46       processor->setWDF(bassmanWDF);
47     if (b == &sinI)
48       insource = false;
49     if (b == &fileI){
50       insource = true;
51       file->init();
52     }
53
54   }
55
56   void sliderValueChanged(Slider* slider) override{
57     if (slider == &g)
58       processor->setGain(g.getValue());
59     if (slider == &volume)
60       processor->setVolume(volume.getValue());
61     repaint();
62   }
63
64   //==============================================================================
65     MainContentComponent()
66     {
67         setSize (800, 600);
68         setAudioChannels (2, 2);
69     }
70
```

```
71      ~MainContentComponent()
72      {
73          shutdownAudio();
74      }
75
76    void initSliders(){
77      addAndMakeVisible(g);
78      g.setRange(0, 7);
79      g.addListener(this);
80      g.setValue(1.0);
81      addAndMakeVisible(volume);
82      volume.setRange(0, 1);
83      volume.addListener(this);
84      volume.setValue(1.0);
85      //addAndMakeVisible(dio);
86      addAndMakeVisible(series);
87      addAndMakeVisible(trio);
88      addAndMakeVisible(bassmanB);
89      addAndMakeVisible(sinI);
90      addAndMakeVisible(fileI);
91      dio.setButtonText("Diodo");
92      series.setButtonText("Diodo 2");
93      trio.setButtonText("Triodo");
94      bassmanB.setButtonText("BassMan");
95      sinI.setButtonText("input: seno");
96      fileI.setButtonText("input: arquivo");
97      dio.addListener(this);
98      series.addListener(this);
99      trio.addListener(this);
100     bassmanB.addListener(this);
101     sinI.addListener(this);
102     fileI.addListener(this);
103   }
104
105     //==============================================================================

106     void prepareToPlay (int samplesPerBlockExpected, double sampleRate)
            override
107     {
108     this->sampleRate = sampleRate;
109     this->samplePerBlock = samplesPerBlockExpected;
110
111     buffer = (float*)calloc(samplePerBlock, sizeof(float));
112     input = (float*)calloc(samplePerBlock, sizeof(float));
113
114     dc = new DiodeClipper("tr", sampleRate, samplePerBlock);
115     sc = new SeriesClipper("tr", sampleRate, samplePerBlock);
116     triode = new Triode("tr", sampleRate, samplePerBlock);
117     bassmanWDF = new BassmanWDF("bassmanwdf", sampleRate, samplePerBlock);
118
119     bassman = new Bassman(sampleRate);
120     bassmanS = new BassmanStack(this, bassman);
121     tone = new ToneStackFilter("bassman", sampleRate, samplePerBlock,
            bassman);
122
```

```
123     plot = new Plot(this, samplePerBlock);
124     processor = new AudioInterpreter(sampleRate, samplePerBlock);
125 
126     file = new FileReader(&input, sampleRate, samplePerBlock);
127     sin = new Sin(220, sampleRate, samplePerBlock);
128 
129     processor->setWDF(sc);
130     processor->setToneStack(tone);
131 
132     insource = false;
133 
134     initSliders();
135     }
136 
137     void getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
            override
138     {
139     if (insource)
140       file->setInput(&input);
141     else
142       sin->setInput(input);
143 
144     processor->processAudioBlock(input, bufferToFill, buffer);
145     }
146 
147     void releaseResources() override
148     {
149     }
150 
151     //
            ==============================================================================

152     void paint (Graphics& g) override
153     {
154     plot->draw(g, buffer);
155     }
156 
157     void resized() override   {
158     g.setBounds(0, 0, 200, 20);
159     volume.setBounds(0, 20, 200, 20);
160     dio.setBounds(200, 0, 100, 20);
161     series.setBounds(200, 20, 100, 20);
162     trio.setBounds(200, 40, 100, 20);
163     bassmanB.setBounds(200, 60, 100, 20);
164     sinl.setBounds(0, getHeight()   40, 200, 20);
165     filel.setBounds(0, getHeight()   20, 200, 20);
166 
167     }
168 
169   void mouseDown(const MouseEvent& e) override{
170     repaint();
171 
172   }
173 
174 private:
175   Bassman *bassman;
```

```
176     BassmanStack* bassmanS;
177     BassmanWDF *bassmanWDF;
178     double sampleRate;
179     int samplePerBlock;
180     DiodeClipper *dc;
181     SeriesClipper *sc;
182     Triode *triode;
183     ToneStackFilter *tone;
184     Plot *plot;
185     AudioInterpreter* processor;
186 
187     FileReader *file;
188     Sin *sin;
189 
190     float *buffer;
191     float  *input;
192 
193     bool insource;
194 
195     Slider g, volume;
196     TextButton dio, trio, series, bassmanB, sinI, fileI;
197 
198       JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainContentComponent)
199 };
200 
201 Component* createMainContentComponent()     { return new
        MainContentComponent(); }
202 
203 
204 #endif   // MAINCOMPONENT_H_INCLUDED
```

Referências

[1] Duncan Amps. Tone stack calculator. http://www.duncanamps.com/tsc/, june 2006. vii, 5

[2] Toshinori Araya and Akio Suyama. Sound effector capable of imparting plural sound effects like distortion and other effects, October 29 1996. US Patent 5,570,424. 24

[3] Stefania Barbati and Thomas Serafini. A perceptual approach on clipping and saturation. 2002. 23

[4] Mokhtar S Bazaraa, Hanif D Sherali, and Chitharanjan Marakada Shetty. *Nonlinear programming: theory and algorithms*. John Wiley & Sons, 2013. 20

[5] DAFx Conference. *DAFx Digital Audio Effects*. John Willey and Sons Ltd, 2002. vii, 18

[6] Michel Doidic, Michael Mecca, Marcus Ryle, Curtis Senffner, et al. Tube modeling programmable digital guitar amplification system, August 4 1998. US Patent 5,789,689. 26

[7] Roger Dannenberg Dominic Mazzoni. Audacity. http://audacityteam.org/, 2000. 34

[8] The Cabbage Foundation. Cabbage. https://github.com/rorywalsh/cabbage/releases, 2014. 34

[9] Dietrich Fränken and Karlheinz Ochs. Synthesis and design of passive runge-kutta methods. *AEU-International Journal of Electronics and Communications*, 55(6):417–425, 2001. 28

[10] Richard G Lyons. *Understanding digital signal processing*. Pearson Education, 2010. vii, 13, 14, 15

[11] Jaromir Macak and Jiri Schimmel. Real time guitar tube amplifier simulation using approximation of differential equations. *DAFx*, (10):1–8, 2010. vii, 3, 23, 28

[12] Mathworks. Matlab. http://www.mathworks.com/products/matlab/index-b.html. viii, 19, 26, 33, 37, 38, 39

[13] N. Noren. Improved vacuum tube models for spice simulations. http://www.normankoren.com/Audio/Tubemodspice_article.html, june 2003. 27

[14] J. Pakarinen and M. Karjalainen. Wave digital simulation of a vacuum-tube amplifier.

Intl. Conf. on Acoustics, Speech and Signal Proc., 2006. vii, viii, 23, 25, 31, 42, 43, 44

[15] Jyri Pakarinen and David T. Yeh. A review of digital techniques for modeling vacuum-tube guitar amplifiers. *Computer Music Journal*, 33(2):85–100, 2009. vii, 4, 25

[16] Martin Robinson. *Getting Started with JUCE*. Packt Publishing Ltd, 2013. 35

[17] Steinberg. Vst - virtual studio technology. http://www.steinberg.net/en/company/technologies.htm 1996. 33

[18] Saeed V. Vaseghi. *Multimedia Signal processing*. John Wiley and Sons Ltd, 2007. vii, 13, 16, 17, 18, 19

[19] Omar Wing. *Classical circuit theory*, volume 773. Springer Science & Business Media, 2008. vii, 7, 8, 9, 10, 12

[20] David T Yeh, Jonathan Abel, and Julius O Smith. Simulation of the diode limiter in guitar distortion circuits by numerical solution of ordinary differential equations. *Proceedings of the Digital Audio Effects (DAFx'07)*, pages 197–204, 2007. vii, 20, 23, 24, 26

[21] David T Yeh, Jonathan S Abel, and Julius O Smith. Simplified, physically-informed models of distortion and overdrive guitar effects pedals. In *Proc. of the Int. Conf. on Digital Audio Effects (DAFx-07)*, pages 10–14. Citeseer, 2007. 23

[22] David T Yeh and Julius O Smith. Discretization of the '59 fender bassman tone stack. *DAFx*, 9(6):18–20, 2006. viii, 3, 5, 36

[23] David Te-Mao Yeh. *Digital implementation of musical distortion circuits by analysis and simulation*. PhD thesis, Stanford University, 2009. viii, 21, 23, 28, 32