CENTRO UNIVERSITÁRIO FEI
FUNDAÇÃO EDUCACIONAL INACIANA

ANTONIO VITOR GROSSI BASSI
DANILO CORREIA LIMA
LEONARDO DE ANDRADE BARBOSA
MARIANA PEREIRA CHAVES
PAULO VICTOR FRUTUOSO SILVA

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE UM SISTEMA EQUALIZADOR PARAMÉTRICO

São Bernardo do Campo
2021

ANTONIO VITOR GROSSI BASSI
DANILO CORREIA LIMA
LEONARDO DE ANDRADE BARBOSA
MARIANA PEREIRA CHAVES
PAULO VICTOR FRUTUOSO SILVA

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE UM SISTEMA EQUALIZADOR PARAMÉTRICO

Trabalho de Conclusão de Curso apresentado ao Centro Universitário FEI, como parte dos requisitos necessários para obtenção do título de Bacharel em Engenharia Elétrica. Orientado pelo Prof. Dr. Marco Antonio Assis de Melo.

São Bernardo do Campo
2021

Antonio Vitor Grossi Bassi
Danilo Correia Lima
Leonardo de Andrade Barbosa
Mariana Pereira Chaves
Paulo Victor Frutuoso Silva

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE UM SISTEMA EQUALIZADOR PARAMÉTRICO

Trabalho de Conclusão de Curso apresentado ao Centro Universitário FEI, como parte dos requisitos necessários para obtenção do título de Bacharel em Engenharia Elétrica. Orientado pelo Prof. Dr. Marco Antonio Assis de Melo.

Comissão julgadora

PROF. DR. MARCO ANTONIO ASSIS DE MELO
Orientador

PROF. DR. FABIO DELATORE
Examinador (1)

PROF. DR. MÁRCIO ANTONIO MATHIAS
Examinador (2)

São Bernardo do Campo
17/06/2021

Dedicamos este trabalho para nossos caros amigos e querida família, que nos inspiraram e motivaram a perseguir a carreira de nossos sonhos.

AGRADECIMENTOS

Ser engenheiro é um sonho perseguido por muitos que admiram as ciências naturais e exatas e toda a contribuição feita pela engenharia na história humana, entretanto não se nasce engenheiro, existe um processo gradativo que requer paciência, disciplina, esmero e acima de tudo bons mentores que estejam dispostos a despertar o intelectual interior em cada um de nós.

Nesta seção, homenageamos estes mentores. Gostaríamos de agradecer o corpo docente do Centro Universitário FEI pelo seu apoio, paciência e dedicação ao ensino acadêmico. A jornada da graduação ensinou a superarmos a nós mesmos a cada dia, buscando sempre a excelência em nossos deveres e ofícios. Agradecemos profundamente aos Professores, Mestres e Doutores por mostrar aos seus alunos que o manancial de suas capacidades não é limitado e há sempre como melhorar a cada dia.

Agradecemos também ao nosso caro orientador Prof. Dr. Marco Antonio Assis de Melo por ter nos guiado e auxiliado neste trabalho de conclusão de curso. Sua dedicação a seus alunos é de grande admiração.

“Estude com vontade aquilo que mais lhe fascina, da forma mais indisciplinada, irreverente e original possível.”

— Richard P. Feynman

RESUMO

O tráfego de informações do mundo atual conta com múltiplas tecnologias e soluções que viabilizam a troca de dados, tais como números, palavras, imagens, áudio e vídeo. O principal alicerce que possibilita todo este fenômeno são as comunicações digitais, que agora mais do que nunca, está sendo empregada em nosso cotidiano de forma abrangente. A transmissão de informações de áudio e imagem digitais via serviços de telecomunicações ou *streaming* são suscetíveis a irregularidades produzidas pelos canais de transmissão, para mitigar estes efeitos existem diversos métodos empregados como algoritmos de correção de erros, codificação de canal, codificação de fonte, filtros entre outros. Contudo, nem sempre a experiência do usuário é a ideal dada a subjetividade presente na percepção sensorial humana, então empregam-se equalizadores para a adequação. Equalizadores são sistemas de processamento de sinais empregados para corrigir ou alterar o espectro de frequência de um sinal elétrico, analógico ou digital, amplamente empregados em serviços de telefonia para mitigar efeitos de multipropagação e em serviços de transmissão de áudio e vídeo para adequação dos dados recebidos. O presente trabalho apresenta o desenvolvimento e a implementação de um equalizador paramétrico digital de oito bandas em tempo real para processamento de sinais digitais de áudio através de uma biblioteca desenvolvida em C++ e implementada no microprocessador Cortex-A72 (ARM v8) 64-bits contido na arquitetura do microcomputador *Raspberry Pi* 4, modelo B.

Palavras-chave: Equalizador, Filtros Digitais, Processamento de Sinais, C++.

ABSTRACT

Information circulation is currently sustained by several technologies and solutions that make feasible data such as words, numbers, pictures, audio and video exchangeable. The main foundation that makes this possible are digital communications, which more than ever makes itself present on our daily lives. Information transmitted through electrical means are susceptible to irregularities present on transmission channels, thus distorting and compromising the comprehension of the data contained. In order to mitigate said irregularities it is employed error detection algorithms, channel and source encoding, filters, et cetera. However, the user experience is not always the ideal given present subjectivism on human sensorial perception, consequently equalizer systems are employed to adequate the user experience. Equalizers are signal processing systems designed to correct or alter an analog or digital signal frequency spectrum. Broadly applied in telecommunications to mitigate the multi-path propagation effects and in services designed to transmit audio and video in order to correct or adequate sound and image according to the user's preference. This work presents the development and implementation of a real time eight-band digital parametric stereo equalizer for digital audio signal processing using a C++ library implemented in a Cortex-A72 (ARM v8) 64-bits microprocessor embedded on a Raspberry Pi 4 – Model B board.

Keywords: Equalizer, Digital Filters, Signal Processing, C++

LISTA DE ILUSTRAÇÕES

Figura 1 - Gráfico gerado pela função $\cos(t)$. 14
Figura 2 - Sinal não periódico 14
Figura 3 - Função *H(t)* 15
Figura 4 - Função $\delta(t-0) = \delta(t)$. 16
Figura 5 - Gráfico de um trem de impulsos $\delta_s(t)$ 17
Figura 6 - Gráfico de $H[n]$ 18
Figura 7 - Gráfico de um Impulso discreto $\delta[n]$. 18
Figura 8 - Sistema de tempo contínuo genericamente representado. 19
Figura 9 - Sistema de tempo contínuo e discreto. 20
Figura 10 - Sistema com recursividade (realimentação). 21
Figura 11 - Diagrama em blocos exemplificando o processo de amostragem. 29
Figura 12 - Sinal produzido por um sistema em tempo contínuo. 30
Figura 13 - Quantização de um sinal discreto. 30
Figura 14 - Trem de pulsos aplicados ao sinal de maneira unitária e finita 31
Figura 15 - Sinal recuperado com a utilização de um filtro passa baixas 32
Figura 16 - Ciruito RC. 36
Figura 17 - Regiões no Plano s e seus mapeamentos no Plano z. 39
Figura 18 - Blocos que compões os filtros digitais. 41
Figura 19 - Filtro FIR genérico. 42
Figura 20 - Resposta ao impulso de um filtro FIR passa faixa. 42
Figura 21 – Possível estrutura de um filtro IIR. 43
Figura 22 - Comportamento gráfico do empenamento em frequência 46
Figura 23 - Vista superior da placa de desenvolvimento. 48
Figura 24 - Interface de sincronização de áudio PCM 49
Figura 25 - Diagrama de blocos da interface de áudio PCM. 50
Figura 26 - Logotipo JUCE 51
Figura 27 - Ambiente de desenvolvimento do Projucer. 52
Figura 28 - Respostas em frequência de equalizadores ressonantes. 54
Figura 29 - Respostas em frequência do equalizador *peak* com fator Q variável. 55
Figura 30 - Comparação entre a aplicação do filtro passa-todas. 57
Figura 31 – Resposta em frequência do filtro passa-todas. 58

Figura 32 - Diagrama de fluxo de um filtro biquadrático de segunda ordem. 59
Figura 33 - Filtros biquadráticos de segunda ordem em cascata. 60
Figura 34 - Filtro biquadrático de segunda ordem na forma direta I. 61
Figura 35 - Filtro biquadrático de segunda ordem na forma direta II. 62
Figura 36 - Filtro biquadrático de segunda ordem na forma direta I transposta. 63
Figura 37 - Filtro biquadrático de segunda ordem na forma direta II transposta. 63
Figura 38 - Diagrama de fluxo da forma direta II transposta. 64
Figura 39 - Forma direta II transposta com coeficientes normalizados por $a0$ 65
Figura 40 - Processo de áudio em um sistema embarcado. 73
Figura 41 - Terminal Linux. 76
Figura 42 - Gráfico da resposta ao impulso do filtro projetado. 80
Figura 43 - Diagrama em blocos da transmissão de áudio. 82
Figura 44 - Estrutura de bandas do equalizador paramétrico 83
Figura 45 - Filtros digitais presentes em cada banda do equalizador 84
Figura 46 - Fluxograma da função principal. 85
Figura 47 - Fluxograma do bloco *MainComponent.cpp* 86
Figura 48 - Diagrama do hardware utilizado. 88
Figura 49 – Software do sistema equalizador paramétrico 89
Figura 50 - Resposta em frequência do sistema equalizador paramétrico projetado. 90
Figura 51 - Respostas do impulso do sistema 90
Figura 52 - Resposta em frequência de cada uma das 8 bandas presentes. 91
Figura 53 - Filtragem do ruído gaussiano branco para frequências graves 92
Figura 54 - Filtragem do ruído gaussiano branco para frequências médias. 92
Figura 55 - Filtragem do ruído gaussiano branco para frequências agudas. 93
Figura 56 - Corte e Ampliação de um sinal senoidal (harmônica) de 1 kHz. 93
Figura 57 - Bancada de testes para o projeto. 94
Figura 58 - Disagnóstico do gerenciador de tarefas da placa. 95
Figura 59 - Software do sistema equalizador paramétrico, distribuição Windows. 95
Figura 60 - Respostas em frequência variadas. 117
Figura 61 - Diagrama de polos e zeros dos sistemas pertencentes a figura 54 118

LISTA DE TABELAS

Tabela 1 - Especificações *Raspberry Pi – 4B* 48
Tabela 2 – Tabela de transformações bilineares. 69
Tabela 3 - Coeficientes do equalizador digital *low-shelf* 100
Tabela 4 - Coeficientes do equalizador digital *high-shelf* 100
Tabela 5 - Coeficientes do equalizador digital *peak*. 100
Tabela 6 - Coeficientes do filtro passa-todas 100

LISTA DE CÓDIGOS

Listagem 1 – Exemplo de código de implementação 74
Listagem 2 – Sintonização do filtro instanciado. 74
Listagem 3 - Cálculo dos coeficientes através da biblioteca. 75
Listagem 4 - Declaração do *buffer* de memória e processamento do sinal de entrada. 76
Listagem 5 – Geração de um arquivo .dat para o sinal processado. 77
Listagem 6 - Obtenção da resposta ao impulso do filtro digital. 78
Listagem 7 - Código do arquivo *MainComponent.hpp* 81
Listagem 8 - Funções da componente de áudio JUCE. 82
Listagem 9 - Funções construtoras da GUI. 84
Listagem 10 - Arquivo *ResonantEQ.cpp* presente na biblioteca *dspLibFEI.hpp* 101
Listagem 11 - Arquivo *MainComponent.hpp* 106
Listagem 12 - Arquivo MainComponent.cpp 108
Listagem 13 - Código da função principal do software equalizador paramétrico digital. 115

LISTA DE SIGLAS

IIR: *Infinite Impulse Response* (Resposta infinita ao impulso).

FIR: *Finite Impulse Response* (Resposta finita ao impulso).

SOS: *Second Order Sections* (Seções de Segunda Ordem).

A/D: Analógico para Digital.

D/A: Digital para Analógico.

LIT: Linear e Invariante no Tempo.

SISO: *Single Input-Single Output* (Entrada Singular – Saída Singular).

FIFO: *first-in first-out* (primeiro-a-entrar primeiro-a-sair).

VST: *virtual studio technology* (Tecnologias de estúdio virtuais).

CODEC: codificador/decodificador.

IDE: *integrated development enviroment* (Ambiente de desenvolvimento integrado).

GUI: *graphic user interface* (Interface Gráfica de Usuário).

ALSA: *Advanced Linux Sound Architecture* (Arquitetura de Som Avançada Linux)

LISTA DE SÍMBOLOS

$\boldsymbol{x(t), y(t)}$ Funções de variável contínua

$\boldsymbol{x[n], y[n]}$ Funções de variável discreta

$\boldsymbol{H(t - t_o)}$ Função de Heaviside

$\boldsymbol{\delta(t - t_o)}$ "Função" Delta de Dirac

$\boldsymbol{\delta_{T_s}(t - t_o)}$ Trem de Impulsos

$\boldsymbol{T_S}$ Período de amostragem (também é o período do trem de impulsos)

$\boldsymbol{\omega_s}$ Frequência angular de amostragem.

$\boldsymbol{f_s}$ Frequência de amostragem

$\boldsymbol{H[n - n_o]}$ Função Degrau Unitário Discreto

$\boldsymbol{\delta[n - n_o]}$ Delta de Kronecker

$\boldsymbol{T\{x(t)\}}$ Operador que representa um sistema

$\boldsymbol{j = \sqrt{-1}}$ Unidade imaginária

$\boldsymbol{T_o}$ Período de um sinal contínuo (de variável contínua)

$\boldsymbol{\omega_o}$ Frequência angular de um sinal contínuo

$\boldsymbol{N_o}$ Frequência de um sinal discreto (de variável discreta)

$\boldsymbol{\Omega_o}$ Frequência angular de um sinal discreto

$\boldsymbol{X(\omega)\ ou\ \mathcal{F}(\omega)\{x(t)\}}$ Transformada de Fourier

$\boldsymbol{\mathcal{F}^{-1}(t)\{X(\omega)\}}$ Transformada Inversa de Fourier

$\boldsymbol{X_D(\Omega)}$ Transformada de Fourier de Tempo Discreto

$\boldsymbol{\tilde{x}(t)}$ Sinal amostrado por trem de impulsos

$\boldsymbol{\mathcal{F}(t)\{\tilde{x}(t)\}\ ou\ \tilde{X}(\omega)}$ Transformada Discreta de Fourier

$\boldsymbol{z}$ Parâmetro em frequência presente na transformada bilinear.

$\boldsymbol{\mathcal{L}(s)\{x(t)\}\ ou\ \mathcal{X}(s)}$ Transformada de Laplace

$\boldsymbol{s}$ Frequência Complexa presente na Transformada de Laplace

$\boldsymbol{\mathcal{L}^{-1}(t)\{\mathcal{X}(s)\}}$ Transformada Inversa de Laplace

$\mathcal{H}(s)$ Função de transferência

SUMÁRIO

1 INTRODUÇÃO **11**

1.1 OBJETIVOS **12**

1.2 EQUALIZADORES PARAMÉTRICOS DIGITAIS E SUA APLICAÇÃO **12**

2 REVISÃO BIBLIOGRÁFICA **13**

2.1 SINAIS **13**

2.1.1 Classificação dos sinais **14**

2.1.1.1 Sinais contínuos e sinais discretos **14**

2.1.1.2 Sinais periódicos e não periódicos **14**

2.1.1.3 Principais sinais de tempo contínuo **15**

2.1.1.4 Principais sinais de tempo discreto **17**

2.2 SISTEMAS **19**

2.2.1 Classificação dos sistemas **20**

2.2.1.1 Sistemas de tempo contínuo e tempo discreto **20**

2.2.1.2 Sistemas lineares e não lineares **20**

2.2.1.3 Sistemas invariantes e variantes no tempo **21**

2.2.1.4 Sistemas com realimentação **21**

2.2.1.5 Resposta de um sistema LIT de tempo contínuo **22**

2.2.1.6 Resposta de um sistema LIT de tempo discreto **23**

2.3 ANÁLISE DE FOURIER **24**

2.3.1 Série de Fourier **24**

2.3.2 A Transformada de Fourier **25**

2.3.3 Série de Fourier de Tempo Discreto **26**

2.3.4 A Transformada de Fourier de Tempo Discreto (TFTD) **27**

2.3.5 A Transformada Discreta de Fourier **28**

2.4 TEOREMA DA AMOSTRAGEM DE NYQUIST **29**

2.4.1 Definição do teorema da amostragem **31**

2.5 A TRANSFORMADA DE LAPLACE 33
2.5.1 Função de Transferência 34
2.5.2 Análise básica de um sistema usando a função de transferência 36
2.6 A TRANSFORMADA Z 37
2.6.1 Função de Transferência Discreta 38
2.7 FILTROS DIGITAIS 40
2.7.1 Componentes dos filtros digitais 40
2.7.2 Filtros FIR 41
2.7.3 Filtros IIR 42
2.7.3.1 Transformação Bilinear 44
2.7.3.2 Empenamento em frequência 45
2.8 PLACA DE DESENVOLVIMENTO RASPBERRY PI 47
2.8.1 Processador BCM2711 – Periférico de áudio: 49
2.9 JUCE 50
2.9.1 Projucer 52
3 METODOLOGIA 53
3.1 EQUALIZADORES RESSONANTES 53
3.1.1 O Filtro Passa-Todas e Fases Não-Lineares 57
3.2 FILTROS BIQUADRÁTICOS 59
3.2.1 Estruturas de um filtro biquadrático 60
3.3 CÁLCULO DOS COEFICIENTES E A TRANSFORMAção BILINEAR 65
3.3.1 O Filtro Passa-Todas Digital 71
3.4 IMPLEMENTAÇÃO DO EQUALIZADOR 72
3.4.1 Biblioteca de Processamento de Sinais Digitais 73
3.4.2 Processamento de Áudio 80
3.4.3 Construção da Interface Gráfica de Usuário 84
3.4.4 O Código Principal 85

3.4.5 Estrutura da Placa de Desenvolvimento 88

4 RESULTADOS E DISCUSSÃO 89

4.1 CONSIDERAÇÕES FINAIS **..... 94**

5 CONCLUSÃO 96

REFERÊNCIAS 97

APÊNDICE A 100

APÊNDICE B 101

APÊNDICE C 106

APÊNDICE D 117

1 INTRODUÇÃO

A engenharia elétrica possui um papel importante no desenvolvimento de tecnologias que proporcionam comodidades para a vida humana, com os avanços do mundo globalizado originou-se um tráfego de informações sem precedentes. Durante tempos remotos para relatar-se um fenômeno ou divulgar alguma informação de utilidade pública era necessário um tempo excessivamente grande para um volume de informação restrito quando comparado com os meios de comunicação da atualidade. A necessidade de se comunicar e trocar informações levou ao desenvolvimento de artifícios que ao longo da história se tornaram cada vez mais sofisticados. O mundo atual conta com sistemas de comunicação de longo alcance capazes de transmitir, receber e processar em tempo real informações de diversas categorias.

As informações consumidas nos dias de hoje, recebidas em forma de sinais elétricos, passam por um processo minucioso de operações matemáticas denominadas como processamento de sinais, para que seja garantida a sua transmissão com alta fidelidade do conteúdo ao consumidor. A fundamentação de processamento de sinais se estabeleceu com os avanços da engenharia elétrica, particularmente nos campos da eletrônica e telecomunicações, que se estendem desde meados de 1930, nos laboratórios Bell com os engenheiros Harry Nyquist, Claude Shannon, et al.

Os sistemas elétricos e eletrônicos (digitais ou analógicos) atuam como intermediários entre o emissor de informação e o consumidor, através da instrumentação dos fenômenos físicos relacionados com os estudos do eletromagnetismo (e.g. radiodifusão), criando sistemas capazes de propagar, interceptar e reproduzir estas informações de forma compreensível pela percepção sensorial humana.

O conteúdo deste trabalho aborda sobre os métodos numéricos envolvidos em processamento digital de sinais, utilizados para o desenvolvimento e implementação de um equalizador paramétrico de 8 bandas em tempo real. Cada banda do equalizador foi implementada como um filtro IIR de oitava ordem constituído de seções de segunda ordem obtidas pela técnica da transformação bilinear. A implementação do sistema foi feita através do desenvolvimento de uma biblioteca em C++, cuja contém as funções necessárias para a execução dos filtros no microprocessador Cortex-A72 integrado no sistema *Raspberry Pi 4*, que por sua vez atuou em conjunção com uma placa de áudio USB para a conversão A/D e D/A.

1.1 OBJETIVOS

O propósito deste trabalho é o desenvolvimento de um sistema equalizador paramétrico digital destinado ao processamento de áudio, visando sua aplicação em computadores, celulares e em tecnologias de estúdio virtuais. Possibilitando que o usuário tenha controle dos parâmetros do sinal de áudio através de uma interface gráfica, podendo alterá-los de acordo com suas preferências. Uma meta secundária é tornar este documento em uma base rica em informações para introduzir o leitor ao estudo de processamento de sinais digitais fornecendo um melhor panorama da área e apresentar uma aplicação dentre as milhares presentes no mundo atual.

1.2 EQUALIZADORES PARAMÉTRICOS DIGITAIS E SUA APLICAÇÃO

Equalizadores paramétricos são sistemas de processamento de sinais de áudio amplamente utilizados em serviços de multimídia, sistemas de comunicação, reconhecimento de fala e em aplicações onde requerem a manipulação do sinal de áudio para a síntese adequada. Os filtros IIR presentes em um equalizador paramétrico digital herdam as características da resposta em frequência de filtros analógicos, entretanto, a grande vantagem de um filtro digital é a reprodução de filtros de alta ordem sem a necessidade de circuitos físicos conectados em cascata, pois este procedimento é realizado através de algoritmos implementados em um processador digital de sinais, desta forma sendo espacialmente viável em comparação aos filtros analógicos e também tendo a possibilidade de se implementar filtros de complexidade ainda maior.

Um equalizador paramétrico atua em um sinal modificando os elementos de frequência presentes através dos filtros projetados, podendo tanto amplificar quanto atenuar dentro de sua faixa de atuação. Um exemplo é a necessidade de corrigir um sinal de voz enviado em um canal de transmissão cujo provoca uma atenuação para as baixas frequências. Um equalizador paramétrico é capaz de restaurar a faixa de frequência prejudicada e manter a magnitude das outras faixas, equalizando as bandas de frequência presentes no sinal. A atuação dos filtros depende dos parâmetros de entrada do usuário, em um equalizador paramétrico é possível controlar as características da banda de passagem e sua magnitude. A combinação de índice de mérito (denominado como fator Q ou fator de qualidade) e ganho, produz um filtro capaz de rejeitar ou amplificar faixas com bandas íngremes ou suaves.

2 REVISÃO BIBLIOGRÁFICA

Um filtro digital é um sistema que realiza uma série operações matemáticas através de pontos discretos extraídos de um sinal. A estrutura de um filtro digital consiste em um sistema de conversão analógico/digital (A/D) para a extração de pontos discretos do sinal seguido de um processador digital de sinais o qual irá executar as operações matemáticas relativas à filtragem do sinal discreto. Os pontos processados são então excursionados para a conversão digital/analógica (D/A), transformando o sinal discreto em contínuo. Conhecer um sinal e suas propriedades é uma das informações de valor para que se possa processá-lo, portanto métodos de análise numérica como a série e a Transformada de Fourier junto com a Transformada de Laplace se fazem presentes e auxiliam no desenvolvimento dos sistemas destinados a receber e processar estes sinais.

2.1 SINAIS

Um sinal, em processamento de sinais, pode ser visto como qualquer função que carregue uma tensão ou uma corrente. Em sistemas eletrônicos, os sinais são caracterizados por valores de tensão ou corrente, ou seja, qualquer função em que sua dimensão seja *volt* ou *ampère*.

Uma observação aqui é que, neste trabalho, quando se fala "função", se retrata, a rigor, da lei de correspondência de uma função. Seja a função x definida como:

$$f: Dom_f \rightarrow CD_f \supseteq Im_f$$
$$t \quad \mapsto \quad f(t)$$

Aqui, f é a função propriamente dita, Dom_f é o domínio da função f, CD_f é o contradomínio da função f. A letra t representa genericamente um elemento que pertence ao domínio de f e, aqui, quase sempre é o tempo, $f(t)$ é a lei de correspondência da função f, e, por fim, Im_f é a imagem de f. Então, fica aqui subentendido que quando se diz função $f(t)$, quer dizer lei de correspondência da função f, ou seja, a variável dependente do tempo.

Quando a função tem o domínio sendo um subconjunto dos números inteiros, denota-se a lei de correspondência por $f[n]$ para evitar confusões. Aqui, o nome da função não é importante e será genérico para alguns casos.

2.1.1 Classificação dos sinais

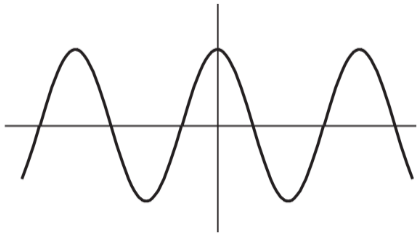
2.1.1.1 Sinais contínuos e sinais discretos

Um sinal é dito contínuo (ou, de tempo contínuo) quando depende de uma variável que pode assumir qualquer valor em um intervalo, como o tempo. A representação matemática será dada por $x(t), y(t), g(t)$, sempre seguindo a notação entre parênteses. Um sinal é dito discreto (ou, de tempo discreto) quando a variável independente pertence a um subconjunto dos números inteiros, que aqui serão representados como $x[n], y[n], g[n]$, notação entre colchetes.

2.1.1.2 Sinais periódicos e não periódicos

Os sinais periódicos são aqueles em que possuem período, ou seja, seu comportamento é cíclico podendo ser identificado em períodos. Em linguagem matemática: $x(t) = x(t + kT_o)$, onde T_o é o período de $x(t)$ e k pertence ao conjunto dos números inteiros ($\mathbb{Z}$). Um exemplo de sinal periódico é a função $\cos(t)$, mostrada na figura abaixo.

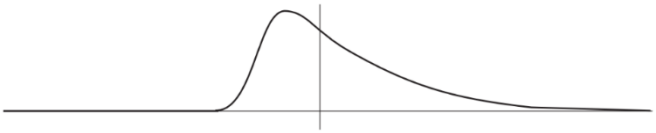
Figura 1 - Gráfico gerado pela função $\cos(t)$.



Fonte: Autor “adaptado de” Lathi, 2008.

Os sinais não periódicos são aqueles que não possuem período, assim, não mantém um padrão de repetição durante o tempo.

Figura 2 - Sinal não periódico



Fonte: Autor “adaptado de” Lathi, 2008.

As definições aqui discutidas seguem de forma análoga aos sinais discretos.

2.1.1.3 Principais sinais de tempo contínuo

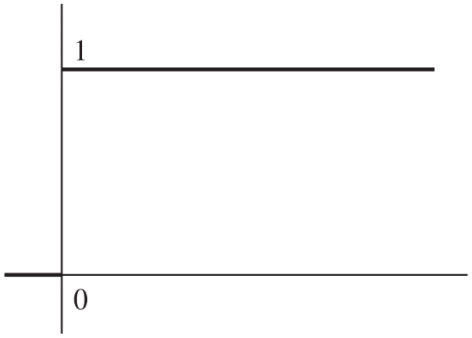
Nem sempre os sinais seguem a definição formal matemática de função, estes sinais são denominados como descritivos e seu comportamento é avaliado por um conjunto de condições. Sinais descritivos, como por exemplo o degrau unitário, são de grande importância para se avaliar o comportamento de sistemas. Os principais sinais a serem levados em consideração são introduzidos a seguir.

a) Degrau Unitário:

A função Degrau Unitário $H(t - t_o)$, ou Função de Heaviside, é definida como:

$$H(t - t_o) = \begin{cases} 1\ para\ t \geq t_o \\ 0\ para\ t < t_o \end{cases} \quad (1)$$

Figura 3 - Função *H(t)*



Fonte: Autor “adaptado de” Lathi, 2008.

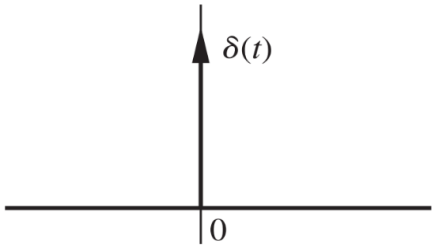
Onde t_o é um tempo qualquer. Na figura acima, $t_o = 0$.

b) Impulso:

A função Impulso $\delta(t - t_o)$, ou Delta de Dirac, é um dos exemplos das funções que não são consideradas como funções pelo formalismo matemático, pois, na verdade, é uma distribuição linear. A definição formal não será abordada, contudo, mesmo assim é possível definir essa “função” com algumas peculiaridades.

$$\delta(t - t_o) = \begin{cases} \to \infty \, , para \; t = t_o \\ 0 \, , para \; t \neq t_o \end{cases} \tag{2}$$

Figura 4 - Função $\delta(t - 0) = \delta(t)$.



Fonte: Autor “adaptado de” Lathi, 2008.

Ou seja, quanto $t = t_o$ implica que $\delta(t - t_o) \to \infty$.

Suas propriedades, ou melhor, peculiaridades são:

$$\int_{-\infty}^{+\infty} x(t) \cdot \delta(t - t_o) \cdot dt = x(t_o) \tag{3}$$

$$x(t) \cdot \delta(t - t_o) = x(t_o) \cdot \delta(t - t_o) \tag{4}$$

$$\int_{-\infty}^{t} \delta(\tau - t_o) \cdot d\tau = \begin{cases} 1 \, , se \; t \geq t_o \\ 0 \, , se \; t < t_o \end{cases} \tag{5}$$

Uma propriedade interessante da Delta de Dirac é que:

$$\int_{-\infty}^{+\infty} \delta(t - t_o) \cdot dt = 1 \tag{6}$$

Com isso, pode-se relacionar a Delta de Dirac com a Função de Heaviside como sendo a derivada desta (mesmo não existindo a rigor), ou seja:

$$\frac{d[H(t - t_o)]}{dt} = \delta(t - t_o) \tag{7}$$

Avaliando os dois lados da igualdade, temos que:

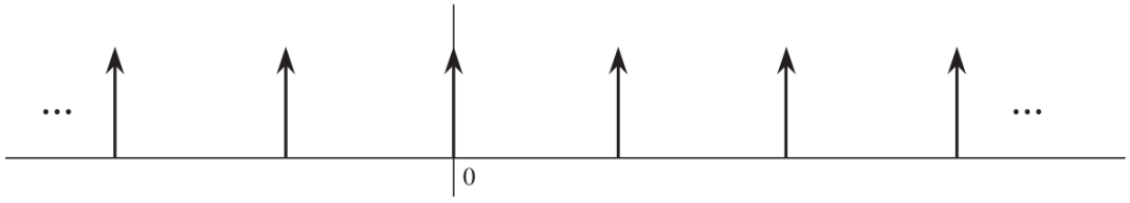
$$\int_{-\infty}^{t} \delta(t - t_o) \cdot dt = H(t - t_o) \tag{8}$$

c) Trem de impulsos:

A função Trem de impulsos pode ser vista como a repetição de Deltas de Dirac com um período T_s.

$$\delta_{T_s}(t - t_o) = \sum_{k=-\infty}^{+\infty} \delta(t - t_o - kT_s) \tag{9}$$

Figura 5 - Gráfico de um trem de impulsos $\delta_{T_s}(t)$.



Fonte: Autor “adaptado de” Lathi, 2008.

2.1.1.4 Principais sinais de tempo discreto

Um sinal de tempo discreto pode ser relacionado com um de tempo contínuo pela seguinte forma.

$$x[n] = x(nT_q)\ n \in \mathbb{Z} \tag{10}$$

O termo “n” quando referenciado da forma $x[n]$ ou $x(nT_q)$, é chamado de tempo discreto, porém, é adimensional. Pode-se entender um sinal de tempo discreto $x[n]$ como o sinal de tempo contínuo $x(t)$ nos instantes de tempo nT_q, sendo T_q um tempo qualquer.

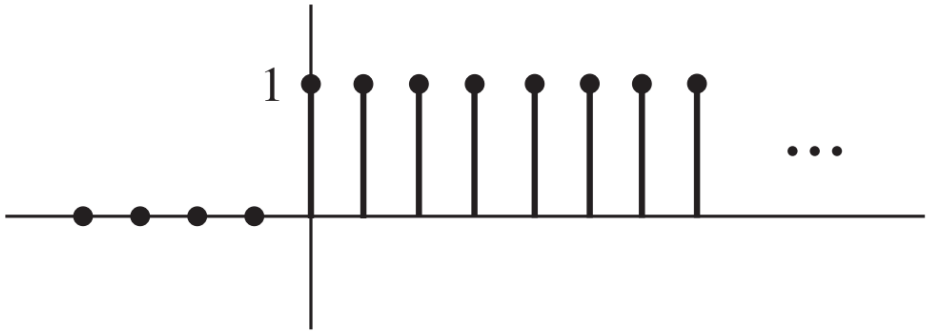
a) Degrau unitário discreto:

A definição do degrau unitário discreto $H[n-n_o]$ é dada por:

$$H[n-n_o] = \begin{cases} 1\,, n \geq n_0 \\ 0\,, n < n_o \end{cases} \quad n \in \mathbb{Z} \tag{11}$$

Onde n_o é um tempo discreto qualquer e $n_o \in \mathbb{Z}$.

Figura 6 - Gráfico de $H[n]$



Fonte: Autor "adaptado de" Lathi, 2008.

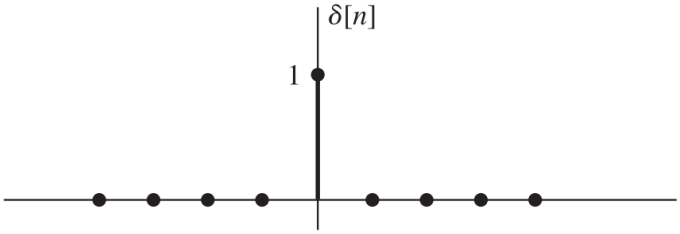
b) Impulso discreto

É a análoga a Delta de Dirac, porém, no tempo discreto. É também denominada como Dirac de Kronecker ou Impulso Unitário Discreto.

$$\delta[n-n_o] = \begin{cases} 1\,, n = n_o \\ 0\,, n \neq n_o \end{cases}, \quad n \in \mathbb{Z} \tag{12}$$

Sendo n_o também é um tempo discreto e pertence ao conjunto $\mathbb{Z}$.

Figura 7 - Gráfico de um Impulso discreto $\delta[n]$.



Fonte: Autor "adaptado de" Lathi, 2008.

c) Trem de Impulso Discreto

A função Trem de impulso discreto pode ser vista como a repetição de Deltas de Kronecker com um período T_s.

$$\delta_{T_s}[n - n_o] = \sum_{k=-\infty}^{+\infty} \delta[n - n_o - kT_s] \quad (13)$$

2.2 SISTEMAS

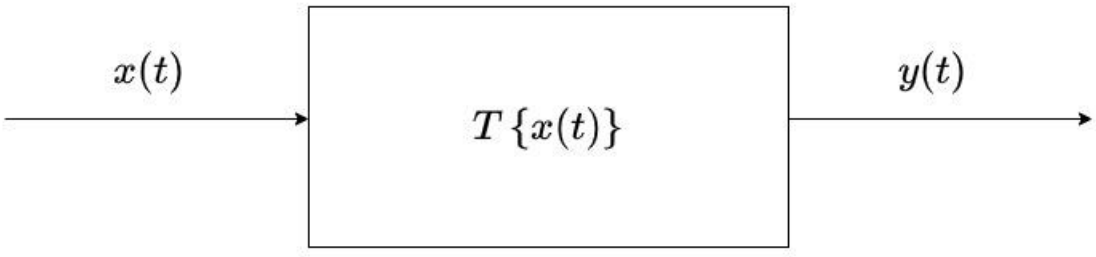
Um sistema em engenharia compreende diversos mecanismos, sendo eles mecânicos, elétricos, hidráulicos entre outros. Um sistema pode ser descrito matematicamente através da relação entre um estímulo e uma resposta, de forma mais específica, seu sinal de entrada e seu sinal de saída. Neste trabalho trataremos sobre sistemas elétricos exclusivamente, embora seja necessário enfatizar que a teoria de sistemas não se limita a apenas sistemas elétricos digitais ou analógicos.

A equação abaixo representa a relação entre os sinais de entrada e saída de forma simplificada.

$$y(t) = T\{x(t)\}\ ou\ y[n] = T\{x[n]\} \quad (14)$$

Onde $y(t)$ (ou $y[n]$) é o sinal de saída, $x(t)$ (ou $x[n]$) é o sinal de entrada e $T\{x(t)\}$ (ou $T\{x[n]\}$) é o operador que representa o sistema, logo, representa como o sistema age matematicamente sobre o sinal de entrada para gerar a saída.

Figura 8 - Sistema de tempo contínuo genericamente representado.



Fonte: Autor.

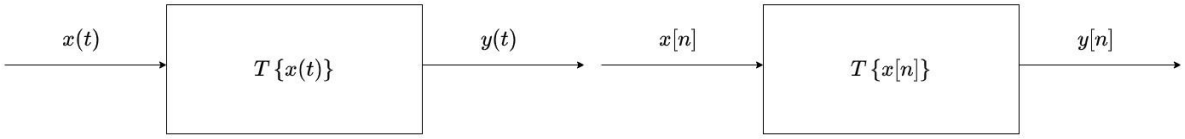
2.2.1 Classificação dos sistemas

2.2.1.1 Sistemas de tempo contínuo e tempo discreto

Um sistema pode operar tanto em sinais de tempo contínuo quanto de tempo discreto. Os sistemas que operam sobre um sinal de tempo contínuo são chamados de Sistemas de Tempo Contínuo e tratam sinais com infinitos pontos descritos por sua função. Por sua vez, os que operam em tempo discreto, tratam sinais com pontos limitados e fixos e são denominados Sistemas de Tempo Discreto. Os sistemas de tempo contínuo compreendem uma variedade de circuitos analógicos compostos por amplificadores operacionais e componentes reativas como indutores e capacitores. Sistemas de tempo discreto, como descrito por seu nome, são incapazes de computar a infinidade de pontos presentes em um sinal analógico, sendo assim, trabalhando apenas com pontos específicos no tempo, a partir desta descrição destacam-se os sistemas digitais como processadores, microcontroladores e computadores.

A figura abaixo mostra o diagrama em blocos destes sistemas.

Figura 9 - Sistema de tempo contínuo e discreto.



Fonte: Autor.

Onde $y(t)$ e $x(t)$ são sinais de tempo contínuo e $x[n]$ e $y[n]$ são sinais de tempo discreto.

2.2.1.2 Sistemas lineares e não lineares

São ditos sistemas lineares cujo operador que à representa obedece à seguinte regra:

$$T\{a \cdot x_1(t) + b \cdot x_2(t)\} = a \cdot T\{x_1(t)\} + b \cdot T\{x_2(t)\} \quad (15)$$

Onde a e b são constantes reais e $x_1(t)$ e $x_2(t)$ são entradas quaisquer.

Qualquer sistema em que seu operador não possui a propriedade da equação (15) é chamado de sistema não linear, válido da mesma forma para sistemas de tempo discreto.

2.2.1.3 Sistemas invariantes e variantes no tempo

Um sistema em que seu operador $T\{x(t-t_o)\} = y(t-t_o)$, ou seja, um descolamento no tempo faz apenas com que a saída também desloque no tempo é dito sistema invariante no tempo, caso contrário, é chamado de sistema variante no tempo.

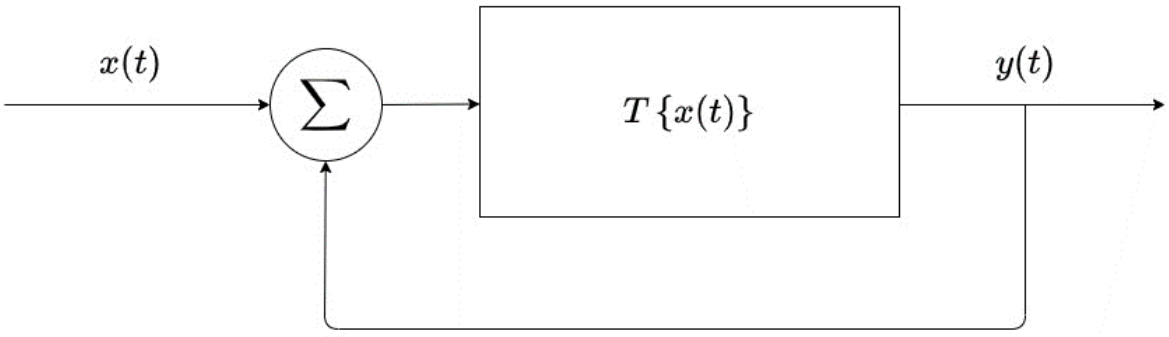
No caso de um sistema de tempo discreto: $T\{x[n-n_o]\} = y[n-n_o]$, onde n_o é um tempo discreto qualquer.

Um sistema que é linear e invariante no tempo é dito um sistema LIT.

2.2.1.4 Sistemas com realimentação

Sistemas realimentados são sistemas que a saída depende de sua entrada e, de alguma forma, de si mesma.

Figura 10 - Sistema com recursividade (realimentação).



Fonte: Autor.

As propriedades da recursividade (realimentação) em um sistema são:

a. Precisão do sistema aumentada;
b. Reduz a sensibilidade que o sistema tem a ruídos externos;
c. Diminui os efeitos não lineares do sistema;
d. Aumenta a largura de faixa que o sistema opera;
e. Tende a aumentar a instabilidade do sistema.

2.2.1.5 Resposta de um sistema LIT de tempo contínuo

O comportamento de um sistema pode ser previsto como posse da resposta ao impulso do sistema, que é a saída que este gera quando a entrada é um impulso (Delta de Dirac) com todas as condições iniciais do sistema nulas (por exemplo, carga nula nos capacitores do sistema). Assim, é definido a resposta ao impulso $h(t)$ como:

$$h(t) = T\{\delta(t)\} \tag{16}$$

Usando as propriedades do Delta de Dirac, qualquer sinal $x(t)$ pode ser escrito como:

$$x(t) = \int_{-\infty}^{+\infty} x(\tau) \cdot \delta(\tau - t) d\tau \tag{17}$$

Como $y(t) = T\{x(t)\}$, substituindo os termos, escreve-se:

$$y(t) = T\left\{ \int_{-\infty}^{+\infty} x(\tau) \cdot \delta(t - \tau) d\tau \right\} = \int_{-\infty}^{+\infty} x(\tau) \cdot T\{\delta(t - \tau)\} d\tau \tag{18}$$

A integral de convolução de dois sinais $f(t)$ e $g(t)$ é definida como de acordo com a equação abaixo.

$$f(t) * g(t) = \int_{-\infty}^{+\infty} f(\tau) \cdot g(t - \tau) d\tau \tag{19}$$

Assim, comparando a equação (18) com a equação (19), a saída $y(t)$ é a convolução de $x(t)$ com $h(t)$.

$$y(t) = x(t) * h(t) = \int_{-\infty}^{+\infty} x(\tau) \cdot h(t - \tau) d\tau = \int_{-\infty}^{+\infty} x(\tau) \cdot T\{\delta(t - \tau)\} d\tau \tag{20}$$

2.2.1.6 Resposta de um sistema LIT de tempo discreto

Analogamente, no caso de sistemas discretos, tem-se que $h[n]$ é a resposta ao delta de Kronecker.

$$h[n] = T\{\delta[n]\} \tag{21}$$

Por sua vez, reescrevendo $x[n]$ aplicando o Delta de Kronecker

$$x[n] = \sum_{k=-\infty}^{+\infty} x[k] \cdot \delta[n-k] \tag{22}$$

Como $y[n] = T\{x[n]\}$, então

$$y[n] = T\{x[n]\} = T\left\{\sum_{k=-\infty}^{+\infty} x[k] \cdot \delta[n-k]\right\} = \sum_{k=-\infty}^{+\infty} x[k] \cdot T\{\delta[n-k]\} \tag{23}$$

A soma de convolução de dois sinais $f[n]$ e $g[n]$

$$f[n] * g[n] = \sum_{k=-\infty}^{+\infty} f[k] \cdot g[n-k] \tag{24}$$

Assim, comparando as duas últimas equações:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{+\infty} x[k] \cdot h[n-k] = \sum_{k=-\infty}^{+\infty} x[k] \cdot T\{\delta[n-k]\} \tag{25}$$

A resposta ao impulso de um sistema desempenha um papel fundamental para a análise de sistemas, uma vez que, a partir do caráter desta resposta podemos averiguar a estabilidade e o caráter de sua resposta em frequência. A análise em frequência de um sistema a partir da

transformada de Fourier e sua equivalente discreta fornecem dados valiosos para engenharia de sistemas, tais como a banda de passagem, faixa de rejeição, faixa de transição e região de ressonância. Compreender a resposta em frequência de um sistema fornece um melhor panorama para como este mesmo se comporta para vários sinais de entrada com distintas frequências.

2.3 ANÁLISE DE FOURIER

É notório que a matemática dispõe uma grande variedade de métodos que podem ser úteis no âmbito da engenharia. Aqui será abordado algumas das ferramentas matemáticas mais importantes para a engenharia elétrica: as "ferramentas" de Fourier, que podem ser usadas em problemas de circuitos elétricos até equações de Maxwell. Isto se dá porque tais "ferramentas" facilitam a solução de equações integro-diferenciais transformando-as em equações algébricas, cuja resolução é mais simples.

2.3.1 Série de Fourier

Qualquer função real periódica $x(t)$, pode ser escrita em termos de soma de senos e cossenos. A Série de Fourier da função $x(t)$ é o segundo termo da equação (26), onde $\omega_o = \frac{2\pi}{T_o}$ é a frequência angular de $x(t)$, T_o o período de $x(t)$ e t_o um tempo qualquer.

$$x(t) = \sum_{n=0}^{\infty} a_n \cos(n\omega_o t) + b_n sen(n\omega_o t) \tag{26}$$

Onde,

$$a_n = \frac{2}{T_o} \int_{t_o - \frac{T_o}{2}}^{t_o + \frac{T_o}{2}} x(t) \cos(n\omega_o t)\, dt \tag{27}$$

$$b_n = \frac{2}{T_o} \int_{t_o - \frac{T_o}{2}}^{t_o + \frac{T_o}{2}} x(t) sen(n\omega_o t) dt \tag{28}$$

A Série de Fourier escrita em termos de senos e cossenos é também chamada de Série Trigonométrica de Fourier. Em suma, a Série de Fourier é um método usado para escrever qualquer função periódica em um somatório de senos e cossenos, ou só cossenos, ou só senos, uma vez que um seno é um cosseno deslocado e vice-versa.

Outro modo de escrever a Série de Fourier é a dita Série Exponencial de Fourier, sendo exatamente a mesma coisa que a série de Fourier, porém escrita de outra forma:

$$x(t) = \sum_{n=-\infty}^{+\infty} \left\{ \frac{1}{T_o} \cdot \left[\int_{-\frac{T_o}{2}}^{+\frac{T_o}{2}} x(t)\, e^{-jn\omega_o t} dt \right] \cdot e^{jn\omega_o t} \right\} \tag{29}$$

2.3.2 A Transformada de Fourier

Uma das limitações da Série de Fourier é que só podem representar sinais periódicos. Contudo, para ampliar a ideia da Série de Fourier para qualquer função (ou sinal) que pode não ser periódico é ver sinais não periódicos como tendo período infinito, ou seja, sinais não periódicos tem período $T_o \to \infty$.

Assim, para chegar numa forma que amplie o método, denota-se:

$\omega_n = n\omega_o$, $\Delta(\omega_n) = \Delta(n\omega_o) = n\omega_o - (n-1)\omega_o = \omega_o = \Delta\omega_n$.

$j = \sqrt{-1} \equiv unidade\ imaginária.$

Fazendo $T_o \to \infty$, portanto $\omega_o \to 0$, e $\omega_o \to 0 \Rightarrow \Delta\omega_n \to 0$. Definindo: $\omega \triangleq n\omega_o \to 0$ e definindo $\Delta\omega = d\omega \triangleq \Delta\omega_n \to 0$, junto que $T_o = \frac{2\pi}{\omega_o}$, pode-se reescrever a Série Exponencial de Fourier como:

$$x(t) = \sum_{n=-\infty}^{+\infty} \left\{ \frac{1}{2\pi} \cdot \left[\int_{-\infty}^{+\infty} x(t)\, e^{-j\omega t} dt \right] \cdot e^{jn\omega_o t} \cdot d\omega \right\} \tag{30}$$

Como o somatório infinito de termos infinitesimais é uma integral, então,

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \left\{ \left[\int_{-\infty}^{+\infty} x(t)\, e^{-j\omega t} dt \right] \cdot e^{j\omega t} d\omega \right\} \tag{31}$$

Define-se, então, a Transformada de Fourier como a equação (32).

$$X(\omega) = \mathcal{F}(\omega)\{x(t)\} = \int_{-\infty}^{+\infty} x(t)\, e^{-j\omega t} dt \tag{32}$$

Atentando-se ao fato de que a Transformada de Fourier é função de ω, qualquer sinal agora (periódico ou não) pode ser escrito em termos de suas frequências bastando que a integral acima exista.

A Transformada de Fourier permite ver as frequências que formam cada função ou sinal qualquer, sendo útil em incontáveis aplicações.

Além disso, a $x(t)$ pode ser reconstruída a partir de $X(\omega)$. E, para tanto, usa-se o que é chamado agora de Transformada Inversa de Fourier ($\mathcal{F}^{-1}(t)\{X(\omega)\}$), tal que

$$\mathcal{F}^{-1}(t)\{X(\omega)\} = x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) \cdot e^{j\omega t} d\omega \tag{33}$$

2.3.3 Série de Fourier de Tempo Discreto

A Série de Fourier "normal" só é aplicável a sinais de tempo contínuo, contudo, existe um análogo de tempo discreto da Série de Fourier, sendo chamada de Série de Fourier de Tempo Discreto (SFTD).

A SFTD (segundo termo da equação (34)) se aplica a sinais discretos e periódicos $x[n]$. Sendo que toda função $x[n] = x[n - N_o]$ com N_o definido como o período de $x[n]$ pode ser escrita em forma de somas de outras funções, tal que:

$$x[n] = \sum_{r=0}^{N_o-1} D_r \cdot e^{jr\Omega_o n} \tag{34}$$

Sendo $\Omega_o = \frac{2\pi}{N_o}$ a frequência de $x[n]$.

O termo D_r é descrito como

$$D_r = \frac{1}{N_o} \cdot \sum_{n=0}^{N_o-1} x[n] e^{-jr\Omega_o n} \tag{35}$$

Um fato importante é que para um sinal discreto ser periódico, a seguinte condição deve ser atendida:

$$\frac{\Omega_o}{2\pi} = \frac{k}{N_o} \quad tal\ que \quad k, N_o \in \mathbb{Z} \tag{36}$$

2.3.4 A Transformada de Fourier de Tempo Discreto (TFTD)

Assim como a SFTD tem seu análogo discreto, a Transformada de Fourier também tem um representante em tempo discreto e chama-se Transformada de Fourier de Tempo Discreto.

Também é um modo de fazer com que essa análise de sinais pela SFTD possa ser ampliada para sinais não periódicos.

Aqui o mesmo conceito é aplicado: fazendo com que $N_o \to \infty$, e alguma manipulação matemática, é possível mostrar que a definição da TFTD como $X_D(\Omega)$, sendo que

$$X_D(\Omega) = \sum_{-\infty}^{+\infty} x[n] e^{-j\Omega n} \tag{37}$$

Uma coisa importante é que $X_D(\Omega) = X_D(\omega + 2\pi)$, então, da TFTD pode-se obter $x[n]$ usando a Transformada Inversa de Fourier de Tempo Discreto, descrita como

$$x[n] = \frac{1}{2\pi}\int_{2\pi} X_D(\Omega)e^{jn\Omega} \tag{38}$$

Sendo que $\int_{2\pi}$ significa que a integral pode ser feita em quaisquer intervalos com o "tamanho" de 2π.

2.3.5 A Transformada Discreta de Fourier

Como computadores só trabalham com sinais digitais, ou seja, sinais que são discretos e quantizados (que não podem assumir qualquer valor), e, como visto, a Transformada de Fourier de Tempo Discreto é contínua, tornando inviável seu uso.

Para contornar tal fato, foi desenvolvida a Transformada Discreta de Fourier, que consegue analisar as frequências que compõe qualquer sinal discreto (periódico ou não) com uma função também discreta.

Definindo um sinal $\tilde{x}(t)$ como

$$\tilde{x}(t) = x(t)\delta_{\mathrm{T_s}}(t) = \sum_n x(nT_s)\delta(t - nT_s) \tag{39}$$

Sendo que $\sum_n$ representa um somatório genérico em n.

O sinal $\tilde{x}(t)$ é chamado de sinal amostrado, pois foi gerado a partir de multiplicações com várias Deltas de Dirac, ou seja, foi "colhido amostras" de $x(t)$.

Assim $\tilde{x}(t)$ é um sinal discreto.

A notação $\tilde{x}(t)$ também será usada para representar sinais discretos, porém, para representar sinais que foram amostrados por uma função Trem de Impulsos.

Fazendo a $\mathcal{F}(t)\{\tilde{x}(t)\}$, é obtido $\tilde{X}(\omega)$ tal que

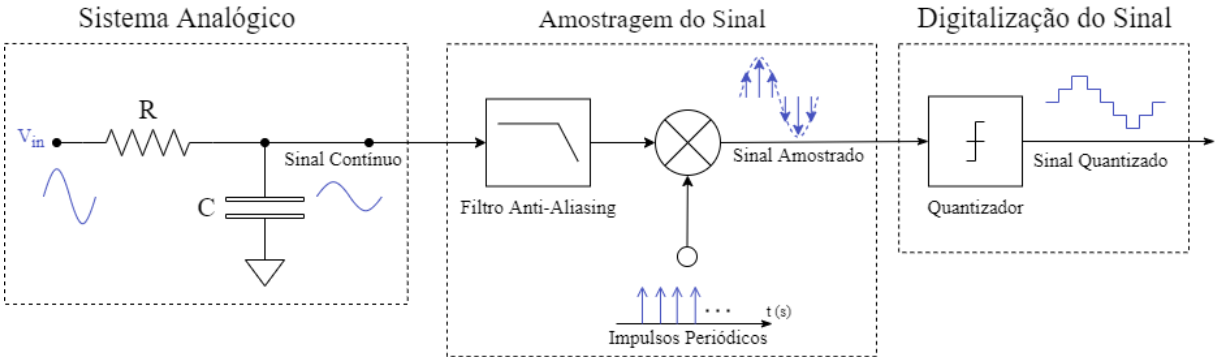
$$\mathcal{F}(t)\{\tilde{x}(t)\} = \tilde{X}(\omega) = \frac{1}{T_s}\cdot\sum_{n=-\infty}^{+\infty} X(\omega - n\omega_s) \quad onde\ \omega_s = \frac{2\pi}{T_s} \tag{40}$$

Como $\tilde{X}(\omega)$ é discreto, pode ser utilizado em algoritmos para o processamento digital de sinais.

Então, a Transformada discreta de Fourier é a própria, porém, aplicada sobre um sinal amostrado.

2.4 TEOREMA DA AMOSTRAGEM DE NYQUIST

Figura 11 - Diagrama em blocos exemplificando o processo de amostragem e digitalização do sinal.

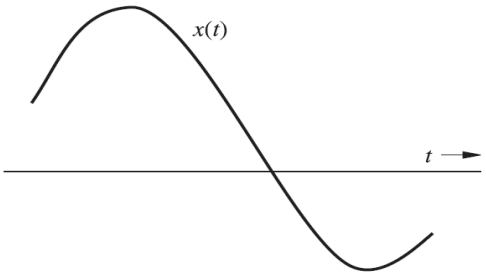


Fonte: Autor.

O processamento de sinais a partir de sistemas digitais se consolida através do processo de amostragem de um sinal contínuo. A partir do momento que o sinal está em domínio discreto, efetua-se a quantização dos valores amostrados, para que o computador esteja a apto a interpretar e armazenar os pontos discretos extraídos. O processo de quantização do sinal atribui níveis de tensão estabelecidos de acordo com a resolução em bits do sistema quantizador. Estes níveis de tensão serão então codificados para que o processador possa interpretar estes dados, e por fim o filtro digital desenvolvido estará apto para atuar. A amostragem é um processo fundamental para o processamento digital de sinais pois serve como uma ponte entre sistemas em tempo contínuo e sistemas em tempo discreto.

Para compreensão do princípio da amostragem de Nyquist, primeiramente faz-se necessário o entendimento da caracterização de sistemas contínuos e discretos. Existem sistemas amplamente conhecidos, denominados contínuos, onde há a possibilidade da execução de análises ou tratamentos de sinais provenientes de uma continuidade no tempo. Os sinais em tempo contínuo podem ser exemplificados através de senoides, cossenoides, exponenciais, ou qualquer forma de onda oriunda de um determinado sistema de prossecução de impulsos como exemplificado na figura abaixo.

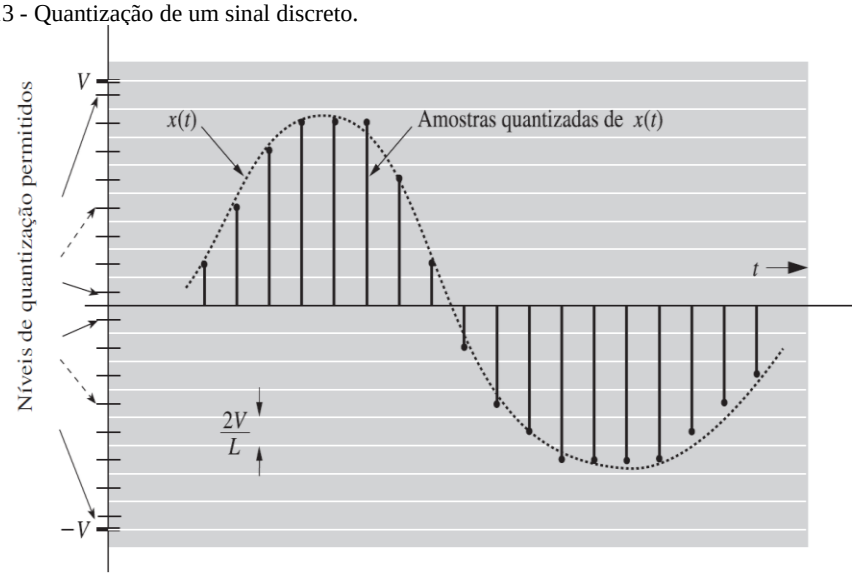
Figura 12 - Sinal produzido por um sistema em tempo contínuo.



Fonte: Autor “adaptado de” Lathi, 2008, p. 683

A análise e síntese de sinais em sistemas digitais demanda a utilização de um critério onde o tempo passa a ser discreto. Para isso, o tempo deixa de ser contínuo e passa a ser constituído de pontos específicos designados em uma subdivisão de acordo com as amostras que são colocadas em questão, como uma sequência numérica. Além da amostragem, para que esta equivalência ocorra, é utilizado o princípio de quantização, também conhecido como arredondamento, pois sinais digitais não podem assumir valores intermediários tais como os sinais analógicos, portanto os pontos específicos anteriormente mencionados, são os denominados níveis de quantização, ou valores pré-estabelecidos que podem ser assumidos por um sinal digital quando ocorre o arredondamento de um valor intermediário do sinal analógico para seu equivalente mais próximo dentro do método de quantização, conforme gráfico exemplificado abaixo.

Figura 13 - Quantização de um sinal discreto.



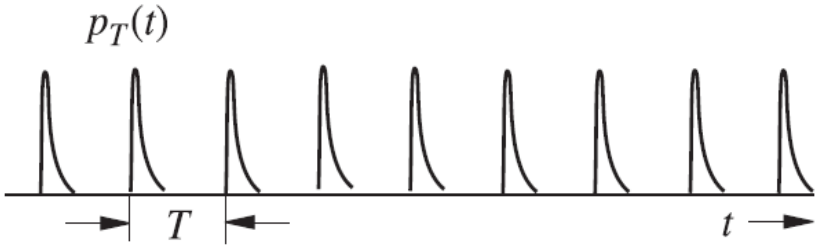
Fonte: Autor “adaptado de” Lathi, 2008, p. 698.

É importante destacar que a informação contida no sinal para ambas as designações se dá de maneira equivalente, pois é imprescindível que a transmissão dos dados seja eficiente e, portanto, uma aplicação comum para estes diferentes tempos é a filtragem digital. Através da filtragem é possível analisar uma informação de domínio contínuo transformando-o em amostras que serão interpretadas por um sistema discreto, e posteriormente filtradas e reconvertidas em domínio contínuo para serem transmitidas ao objeto final com a maior fidelidade possível, o que é viabilizado pelo acatamento da frequência de Nyquist.

2.4.1 Definição do teorema da amostragem

Considerando um sinal de espectro em frequência delimitado ou finito para uma faixa B Hz, a fidelidade de transmissão deste sinal é garantida pela menor frequência de amostragem, denominada frequência de Nyquist, cujo valor obedece à inequação que determina a taxa $f_s > 2B$ amostras por segundo, ou seja, a frequência mínima (frequência de Nyquist) para a amostragem do sinal é estabelecida em $f_s = 2B$. Para melhor fundamentação, pode-se considerar um filtro passa baixas com um trem de pulsos aplicado à entrada do sistema. Os pulsos são emitidos de maneira unitária e possuem largura finita – figura 14, determinando o intervalo – uniforme ou não – do período a ser amostrado (período de Nyquist), ou seja, podem ser representados pela Série Trigonométrica de Fourier.

Figura 14 - Trem de pulsos aplicados ao sinal de maneira unitária e finita

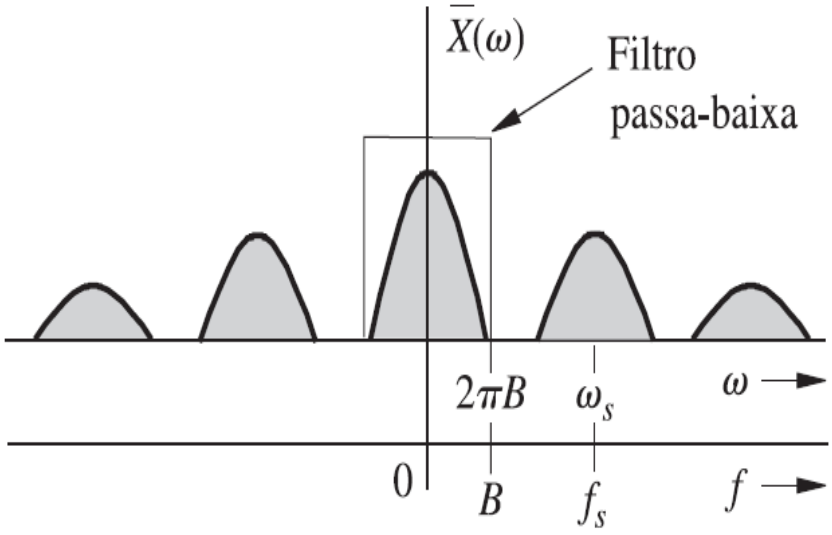


Fonte: Autor “adaptado de” Lathi, 2008, p. 683

Assim, se faz necessária a realização da Transformada de Fourier para todos os termos da série, onde é obtido um deslocamento do espectro de maneira periódica e infinita, resultando na amostragem em tempo discreto. Isso é possível pois para cada período amostrado, apenas

um sinal possuirá $f_s > 2B$ e todas as outras amostras são denominadas subamostradas, por apresentarem frequências inferiores à mínima. Para recuperação deste sinal é necessária a utilização de um filtro, para este caso como já mencionado o passa-baixa, e para isso demanda-se que os ciclos consecutivos não estejam sobrepostos, o que é obtido com a obediência da frequência de Nyquist ($f_s = 2B$). A abaixo exemplifica como o sinal será filtrado.

Figura 15 - Sinal recuperado com a utilização de um filtro passa baixas



Fonte: Autor Autor “adaptado de” Lathi, 2008, p. 683

Portanto, a frequência de Nyquist caracteriza-se por ser a menor taxa de amostragem demandada para que os dados sejam preservados, porém é importante ressaltar que este teorema somente poderá ser aplicado para o caso de o espectro não possuir nenhum dos impulsos ou componentes derivativas no índice máximo da frequência ($f = B\ Hz$).

2.5 A TRANSFORMADA DE LAPLACE

Aqui será apresentado mais uma ferramenta matemática para análise de circuitos e sistemas, sendo esta, a Transformada de Laplace (ou, transformada de Laplace Unilateral).

A Transformada de Fourier visa descrever qualquer sinal como somatório de cossenos (ou, senos), e isso é útil para a solução de problemas de circuitos elétricos onde o regime transitório do circuito é desprezível. A Transformada de Laplace visa descrever os sinais como um somatório de cossenos e exponenciais, assim, levando a parte transitória dos circuitos em análise. Em suma:

• Fourier descreve $x(t) = \sum \cos\left(\omega_G t + \phi_G\right)$

• Laplace descreve $x(t) = \sum \cos(\omega_G t + \phi_G)\, e^{\sigma_G t}$

A Transformada de Laplace de um sinal $x(t)$ é definida como $\mathcal{L}(s)\{x(t)\}$, tal que:

$$\mathcal{L}(s)\{x(t)\} = \mathcal{X}(s) = \int_0^{+\infty} x(t)e^{-st} \cdot dt \quad , s = \sigma + j\omega \tag{41}$$

A variável s é chamada de frequência complexa e é um número complexo.

Então, a vantagem da Transformada de Laplace sobre a Transformada de Fourier é que ela considera as condições iniciais dos elementos dos circuitos e seus transitórios, coisa que a Transformada de Fourier não faz, assim, sendo muito útil em, também, incontáveis aplicações na engenharia quando esses termos são consideráveis.

Com posse da Transformada de Laplace pode-se obter de volta o sinal que foi transformado com a chamada Transformada de Laplace Inversa $\mathcal{L}^{-1}(t)\{\mathcal{X}(s)\}$ definida como

$$x(t) = \mathcal{L}^{-1}(t)\{\mathcal{X}(s)\} = \frac{1}{2\pi j}\int_{\gamma - j\infty}^{\gamma + j\infty} \mathcal{X}(s)e^{st}dt \tag{42}$$

A integral acima envolve números complexos e seu cálculo não é trivial, sendo que γ é uma constante que é definida quando se leva em conta as condições para a transformada de Laplace existir e, aqui, não será abordada.

Para evitar o cálculo da Transformada de Laplace Inversa, se encontram tabelas (assim como existem para Transformadas de Fourier Inversa) com transformadas e suas equivalentes "anti-transfomadas" (Transformadas Inversas).

Como a transformada de Laplace de uma função do tempo leva a outra função que tem como variável independente valores complexos ($s = \sigma + j\omega$), o domínio da função propriamente dita é, então, um plano complexo denominado "Plano s".

2.5.1 Função de Transferência

A transformada de Laplace da resposta ao impulso $h(t) = T\{\delta(t)\}$ de um sistema LIT é chamada de Função de Transferência (FT).

Seja um sistema LIT de tempo contínuo regido pela equação $y(t) = x(t) * h(t)$, então a Transformada de Laplace de $y(t)$ é $\mathcal{L}(s)\{y(t)\} = \mathcal{Y}(s)$.

$$\mathcal{L}(s)\{y(t)\} = \mathcal{L}(s)\{x(t) * h(t)\} \tag{43}$$

Um teorema importante aqui deve ser mencionado: o Teorema da Convolução. Tal teorema diz que a Transformada de Laplace da convolução de dois sinais é igual a multiplicação da transformada de Laplace do primeiro sinal com a Transformada de Laplace do segundo sinal, matematicamente é possível escrever:

$$\mathcal{L}(s)\{f(t) * g(t)\} = \mathcal{L}(s)\{f(t)\} \cdot \mathcal{L}(s)\{g(t)\} \tag{44}$$

Então, a equação (43) pode ser escrita como:

$$\mathcal{L}(s)\{x(t) * h(t)\} = \mathcal{L}(s)\{x(t)\} \cdot \mathcal{L}(s)\{h(t)\} = \mathcal{X}(s) \cdot \mathcal{H}(s) = \mathcal{Y}(s) \tag{45}$$

E, portanto,

$$\mathcal{H}(s) = \frac{\mathcal{Y}(s)}{\mathcal{X}(s)} \tag{46}$$

$\mathcal{H}(s)$, a função de transferência do sistema, agora pode ser obtida de forma mais simples: através da razão das transformadas de Laplace do sinal de saída pelo sinal de entrada de um sistema LIT de tempo contínuo.

A análise de Laplace de um sistema LIT de tempo contínuo se resume não só facilitar as contas na análise de sistemas por conta de tabelas de Transformadas e Transformadas Inversas de Laplace, entretanto, ajuda, também, na visualização do comportamento do sistema, pois, uma vez que o sistema está descrito não mais na variável t, e sim no domínio da variável s, é possível visualizar o comportamento que o sistema tem de acordo com a frequência (método denominado "análise da resposta em frequência") e também verificar sua estabilidade do mesmo.

Um sistema é dito estável se sua resposta ao impulso tende a zero quando o tempo tende a infinito, matematicamente, um sistema é estável se obedece a condição descrita pela equação (47).

$$\lim_{t\to\infty} h(t) = 0 \tag{47}$$

A estabilidade do sistema pode ser vista através de sua função de transferência, bastando analisar a parte real dos polos (valor que tende a função para o infinito). Assim, se os polos da FT têm parte real positiva, implica que o sistema é instável. Os zeros de uma função de transferência não influenciam na estabilidade do sistema.

Uma observação aqui é que, quando um sistema é realimentado, a relação das transformadas de Laplace da saída pela entrada é chamada de função de transferência de malha fechada e é denotada por $\mathcal{H}_F(s)$, onde:

$$\mathcal{H}_F(s) = \frac{\mathcal{Y}(s)}{\mathcal{X}(s)} \quad , para\ sistemas\ munidos\ de\ retroação \tag{48}$$

Exemplificando: um sistema realimentado descrito pela função de transferência de malha fechada $\mathcal{H}_F(s) = \frac{3}{s^3+3s^2+2s+3}$ tem três polos, contudo, todos os três polos têm parte real negativa, assim, o sistema é estável.

2.5.2 Análise básica de um sistema usando a função de transferência

Visto que a função de transferência de um sistema é tal que leva a valores complexos, pode ser escrita na forma polar por $\mathcal{H}(s) = |\mathcal{H}(s)|\angle\theta_{\mathcal{H}(s)}$, onde $\theta_{\mathcal{H}(s)}$ é a fase de $\mathcal{H}(s)$.

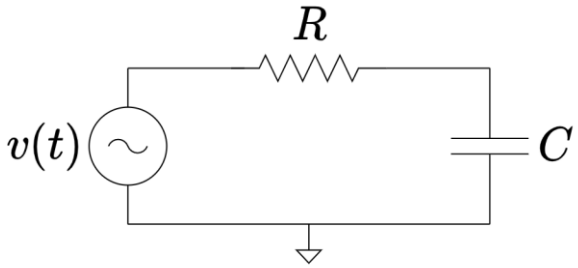
Tendo em vista que $s = \sigma + j\omega$, na prática, σ é responsável pelo termo transitório (os termos dependentes de exponenciais) da resposta do sistema e $j\omega$ é o termo responsável pelos termos senoidais.

Quando o objetivo é estudar o comportamento do sistema desprezando o transitório, é possível ignorar a parte real de s e analisar $\mathcal{H}(j\omega) = \mathcal{H}(s)|_{s=j\omega}$.

O significado mais claro disto é que, como na análise de Fourier, considera-se que todas as entradas e saídas são funções senoidais (ou cossenoidais), assim, pode-se analisar como o ganho (relação da amplitude do sinal de saída dividido pela amplitude do sinal de entrada) como variável dependente da frequência.

Uma aplicação disso é vista na análise de um circuito composto por uma fonte de tensão, um resistor e um capacitor:

Figura 16 - Ciruito RC.



Fonte: Autor.

A função de transferência que relaciona a transformada de Laplace da tensão no capacitor com a transformada de Laplace da função que descreve a fonte de tensão é $\mathcal{H}(s) = \frac{1}{1+RCs}$.

Assumindo que o circuito está em regime permanente senoidal, ou seja, todas as formas de ondas são senoides, então σ não é útil e é possível analisar o ganho que o circuito tem em em função da frequência analisando $\mathcal{H}(j\omega) = \mathcal{H}(s)|_{s=j\omega}$.

Assim, $\mathcal{H}(j\omega) = \frac{1}{1+jRC\omega} = \frac{1}{\sqrt{1+(RC\omega)^2}}\angle \tan^{-1}\frac{1}{RC\omega}$ e é possível analisar o módulo de $\mathcal{H}(j\omega)$ em função da frequência ω.

2.6 A TRANSFORMADA Z

A transformada Z é uma ferramenta matemática amplamente usada na teoria de sistemas para o tratamento de sinais e sistemas discretos. Ela surge da transformada de Laplace e possui propriedades e regras de cálculo semelhantes. A transformada Z se aplica a sinais no domínio do tempo discreto, enquanto a transformada de Laplace é usada para cálculos correspondentes no domínio do tempo contínuo.

Exemplos de aplicações da transformada Z são na teoria de controle digital e para o projeto de filtros digitais.

A transformada Z de um sinal discreto, $\mathcal{X}[z]$, é definida como

$$\mathcal{Z}[z]\{x[n]\} = \mathcal{X}[z] = \sum_{n=0}^{+\infty} x[n]z^{-n} \tag{49}$$

Onde z é uma variável que pertence ao conjunto dos números complexos ($\mathbb{C}$).

Vale ressaltar que neste trabalho quando se fala de transformada Z, se quer dizer da Transformada Z Unilateral.

O sinal pode ser recuperado a partir da transformada Z do mesmo, contudo, como na transformada de Laplace, é um tanto complicado. Assim, utilizam-se tabelas de transformadas e anti-transformadas Z para a análise com este método.

Ou seja, $\mathcal{Z}^{-1}[n]\{\mathcal{X}[z]\}$ é a transformada inversa Z

$$x[n] = \mathcal{Z}^{-1}[n]\{\mathcal{X}[z]\} \tag{50}$$

A transformada Z existe se a afirmação a seguir for satisfeita:

$$|\mathcal{X}[z]| = \sum_{n=0}^{+\infty} \frac{|x[n]|}{|z|^n} < \infty \tag{51}$$

A transformada Z, assim como a transformada de Laplace, é um operador linear, assim, a propriedade a seguir é válida, onde α e β são constantes reais:

$$\mathcal{Z}[z]\{\alpha \cdot x_1[n] + \beta \cdot x_2[n]\} = \alpha \cdot \mathcal{X}_1[z] + \beta \cdot \mathcal{X}_2[z] \quad (52)$$

A Transformada Z se relaciona com a Transformada de Laplace pela relação $z = e^{sT_s}$. Tomando um sinal amostrado $\tilde{x}(t)$, onde

$$\tilde{x}(t) = \sum_{n=0}^{+\infty} x(nT_s) \cdot \delta(t - nT_s) = \sum_{n=0}^{+\infty} x[n] \cdot \delta(t - nT_s) \quad (53)$$

A transformada de Laplace do sinal é

$$\mathcal{L}(s)\{\tilde{x}(t)\} = \int_0^{+\infty} \left(\sum_{n=0}^{+\infty} x[n] \cdot \delta(t - nT_s) \right) e^{-st} \cdot dt = \sum_{n=0}^{+\infty} x[n] \cdot e^{-nsT_s} \quad (54)$$

Contudo, como $z = e^{sT_s}$, obtém-se

$$\mathcal{L}(s)\{\tilde{x}(t)\} = \sum_{n=0}^{+\infty} x[n] \cdot z^{-n} = \mathcal{Z}[z]\{x[n]\} = \mathcal{X}[z] \quad (55)$$

É importante dar ênfase que, análogo com Laplace, a transformada Z transforma uma função de tempo discreto em outra com qual a variável dependente, z, é complexa. O plano que compreende os possíveis valores de z é chamado de "plano z"

2.6.1 Função de Transferência Discreta

A resposta ao Delta de Kronecker $h[n] = T\{\delta[n]\}$ de um sistema no domínio z, é dado por

$$\mathcal{H}[z] = \mathcal{Z}[z]\{h[n]\} = \frac{\mathcal{Z}[z]\{y[n]\}}{\mathcal{Z}[z]\{x[n]\}} = \frac{\mathcal{Y}[z]}{\mathcal{X}[z]} \quad (56)$$

$\mathcal{H}[z]$ é chamada de função de transferência discreta que, analogamente a função de transferência $\mathcal{H}(s)$ que é usada para analisar o comportamento de sistemas de tempo contínuo, é usada para analisar o comportamento de sistemas de tempo discreto. Se o sistema e realimentado, então a função de transferência discreta é chamada de função de transferência discreta de malha fechada mencionada por $\mathcal{H}_F[z]$.

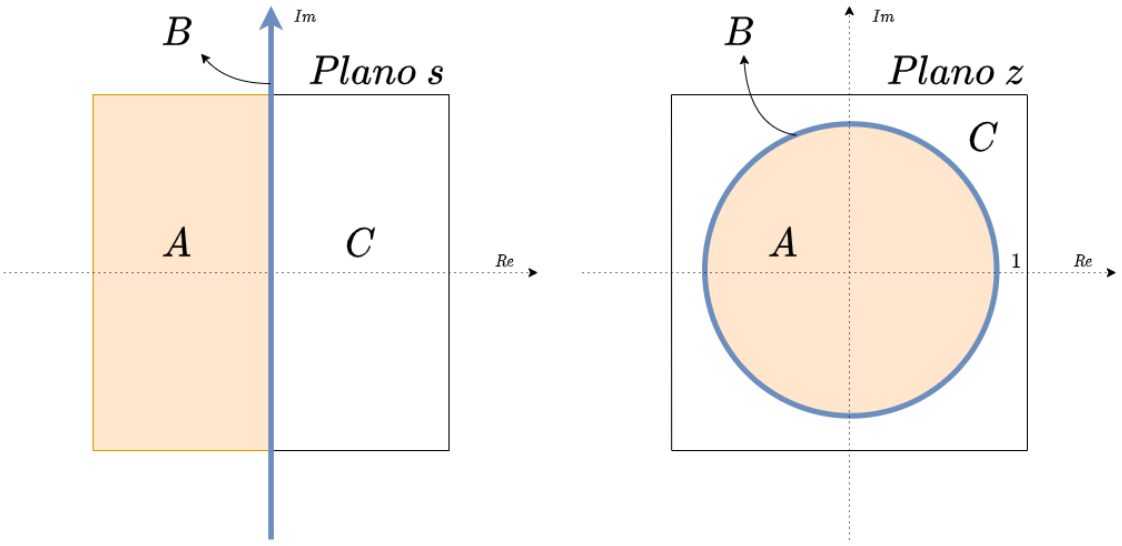
Como já visto antes nos sistemas de tempo contínuo, um sistema é estável se os polos da função de transferência $\mathcal{H}(s)$ não tem parte real positiva. Como z é relacionado com s de forma $z = e^{sT_s}$, e visto que $s = \sigma + j\omega$, pode-se reescrever z como: $z = e^{(\sigma+j\omega)T_s}$, ou, de forma trigonométrica e polar, como:

$$z = e^{\sigma T_s}(\cos \omega T_s + j \operatorname{sen} \omega T_s) = e^{\sigma T_s}\angle \omega T_s \tag{57}$$

Então, como $\sigma = \frac{1}{T_s}\ln|z|$ deve ser menor que zero para o sistema de tempo contínuo ser estável e levando em conta que T_s é sempre positivo, equivalentemente nos sistemas discretos, $|z|$ deve ser menor que 1 para que o sistema de tempo discreto seja estável.

A região em azul na figura seguinte mostra a região que os polos da função de transferência de tempo contínuo $\mathcal{H}(s)$ devem ficar para o sistema ser estável, e à direita mostra a onde os polos de $\mathcal{H}[z]$ devem estar para que o sistema de tempo discreto seja estável.

Figura 17 - Regiões no Plano s e seus mapeamentos no Plano z.



Fonte: Autor.

2.7 FILTROS DIGITAIS

A classificação dos filtros digitais é dada, geralmente, por uma característica: a duração da resposta que o filtro quando a entrada é um impulso unitário discreto, que pode ser de duração finita ou de duração infinita. Os filtros de resposta ao impulso finita são denominados de filtros FIR (*finite impulse response*) e, os filtros cuja resposta ao impulso é infinita são chamados de filtros IIR (*infinity impulse response*). O projeto e implementação destes são diferentes, pois ambos têm vantagens e desvantagens entre si. Contudo, a ideia básica do filtro digital é a mesma do analógico: agir como um seletor de frequências dos sinais.

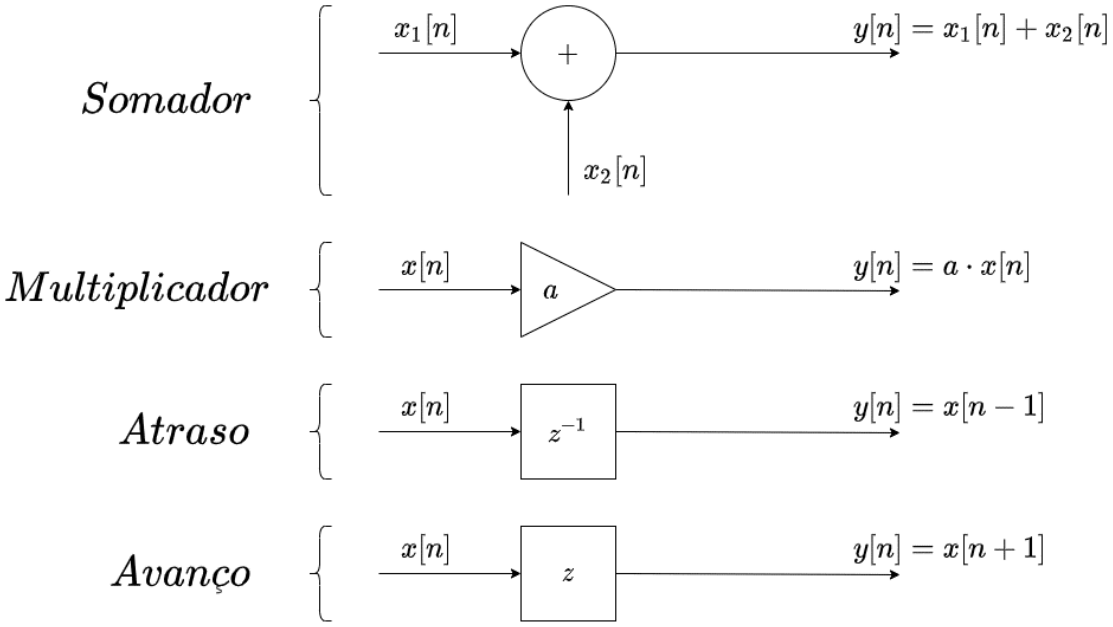
As principais vantagens que os filtros digitais têm sobre os analógicos são:

a. Os valores dos componentes que formam um filtro analógico mudam conforme a temperatura e isso não acontece com os filtros digitais, assim, estes têm estabilidade térmica;
b. São programáveis, assim, a função de transferência de um filtro digital pode ser facilmente modelada para atender outras especificações de projeto;
c. Mesmo na posse de apenas um filtro digital, este pode ser usado para processar vários sinais de entrada por meio de técnicas de multiplexação.
d. A principal desvantagem dos filtros digitais é que o desempenho destes depende da especificação do hardware, ou seja, para obter mais precisão nos filtros, os registradores do computador que armazenam os “dados” do filtro devem ser maiores.

2.7.1 Componentes dos filtros digitais

Quatro tipos de blocos interconectados formam os filtros digitais, são eles: o bloco somador, o bloco multiplicador, o bloco de atraso (ou, do inglês, *delay*) e o bloco de avanço.

Figura 18 - Blocos que compões os filtros digitais.



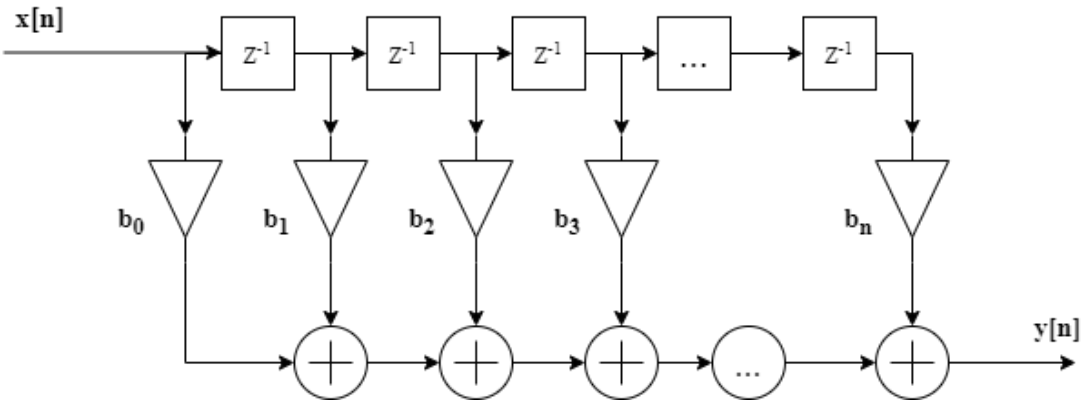
Fonte: Autor.

O bloco somador, representado com o símbolo “Σ” ou “+”, é responsável por somais dois sinais, o bloco multiplicador é responsável por multiplicar a entrada por uma constante e o bloco de atraso/avanço é responsável por atrasar ou adiantar o sinal.

2.7.2 Filtros FIR

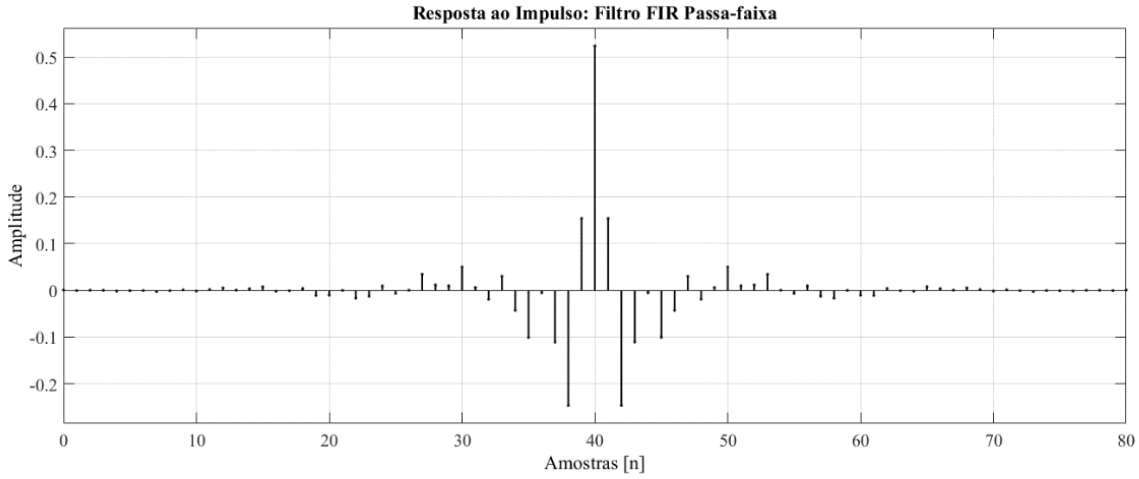
Os filtros FIR são os filtros que possuem a resposta ao impulso unitário discreto com duração finita. Assim, os filtros FIR são sempre estáveis, contudo, necessitam de alto processamento para funcionarem, devido ao alto número de coeficientes presentes em sua estrutura. Também têm a característica de não serem recursivos (realimentados), ou seja, a saída só depende da entrada e, outra característica é que não há nenhum filtro analógico análogo a eles, por serem fisicamente impraticáveis. Todos os filtros FIR possuem fase linear o que é uma qualidade desejada em desenvolvimento de filtros, pois a distorção do sinal será a mínima possível.

Figura 19 - Filtro FIR genérico.



Fonte: Autor.

Figura 20 - Resposta ao impulso de um filtro FIR passa faixa.



Fonte: Autor.

2.7.3 Filtros IIR

Filtros IIR são aqueles que apresentam a resposta ao impulso unitário discreto com duração infinita. Esses filtros são projetados com base na resposta em frequência de filtros analógicos, outra característica dos filtros IIR é a recursividade (realimentação), onde a saída do filtro depende não só do sinal de entrada, mas também da própria saída.

Entretanto, mesmo tendo a resposta ao impulso de duração infinita, esses filtros são projetados de modo a ter essa resposta tendendo a zero ao longo que o tempo passa, desse modo, fazendo com que o filtro seja estável. A saída do filtro pode ser expressa pela equação abaixo:

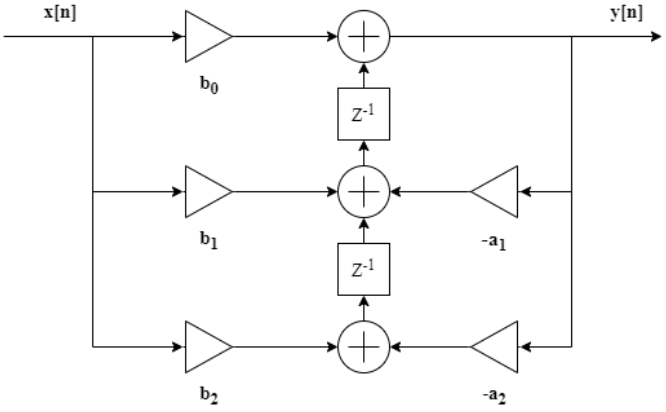
$$y[n] = \sum_{i=0}^{L-1} b_i \cdot x[n-i] \; - \; \sum_{m=1}^{M} a_m \cdot y[n-m] \tag{58}$$

Onde os coeficientes b_i e a_m são constantes e determinam a resposta do filtro. Se for aplicado a Transformada Z e reorganizada a expressão, é possível chegar na função de transferência, generalizada por:

$$\mathcal{H}[z] = \frac{\sum_{i=0}^{L-1} b_i z^{-1}}{1 + \sum_{m=1}^{M} a_m z^{-1}} \tag{59}$$

Um filtro IIR de segunda ordem pode ser representado pelo diagrama de blocos mostrado genericamente na figura a seguir:

Figura 21 – Possível estrutura de um filtro IIR.



Fonte: Autor.

Sendo filtros recursivos, filtros IIR tem a vantagem, quando comparados aos FIR, que não possuem recursividade, é que para o mesmo grau de seletividade em frequência, necessita uma ordem bem inferior e do ponto de vista aritmético ser mais simples a computação dos coeficientes em tempo real, o que resulta em um consumo de processamento menor. Outra vantagem é poder utilizar modelos analógicos, como princípio de projeto e, depois disso, transformá-los em filtros digitais. A desvantagem se dá pela característica não linear da curva de fase dos filtros recursivos, causando distorções nos sinais filtrados.

Por fim, a decisão de implementar um filtro FIR ou IIR varia muito da aplicação final proposta. O filtro FIR é naturalmente estável, transitórios iniciais indesejáveis têm duração

finita e a implementação de um filtro com fase linear é relativamente simples. Porém, esse tipo de filtro requer o cálculo de diversos coeficientes independentes, o que acaba requerendo muito tempo de processamento.

Os filtros IIR, quando bem projetados, apresentam estabilidade e boa performance, além de grande atenuação na frequência de corte utilizando filtros de baixa ordem, o que acarreta a melhora da velocidade de resposta do sistema.

Para o presente projeto foi preferido a implementação dos filtros através do modelo IIR, pois, atende bem a faixa de frequência de áudio humana e o ganho de processamento em relação ao filtro FIR é notável devido a praticidade matemática.

Para o projeto de filtros digitais, existem bastantes métodos, cada um possuindo sua metodologia que se adapta melhor conforme a demanda do projeto, dentre estes métodos, se destacam:

a. Transformação Bilinear;
b. Aproximação Retangular;
c. Técnica de invariância do impulso.

A transformação bilinear, também denominada na literatura como método de Tustin, trata o projeto do filtro digital através de um filtro analógico modelado por uma função de transferência em domínio s. A transformação bilinear preserva a estabilidade intrínseca do sistema analógico previamente modelado e contorna a distorção provocada pelo *aliasing* resultante da amostragem, ao contrário dos métodos da invariância ao impulso e aproximação retangular.

2.7.3.1 Transformação Bilinear

A Transformação Bilinear é um método utilizado em processamento de sinais para "transformar" um sistema de tempo contínuo em um equivalente discreto, ou, até mesmo, o contrário.

Com isso, na posse de um sistema com a função de transferência $\mathcal{H}(s)$ descrevendo um sistema contínuo no domínio da frequência, com a transformação bilinear é possível transformar essa função $\mathcal{H}(s)$ em $\mathcal{H}[z]$.

Na seção 2.6 foi visto que a Transformada Z é relacionada com a Transformada de Laplace de forma que $z = e^{sT_s}$, contudo, pode-se aproximar z por:

$$z \approx \frac{1 + \frac{sT_s}{2}}{1 - \frac{sT_s}{2}} \tag{60}$$

Como $z = e^{sT_s}$, então $s = \frac{1}{T_s}\ln(z)$, assim, pode-se aproximar s por:

$$s \approx \frac{2}{T_s} \cdot \frac{z-1}{z+1} \tag{61}$$

Em resumo, o método consiste em utilizar as aproximações mencionadas e fazer a aplicação destas na função de transferência de um sistema de tempo contínuo para transformá-lo em um de tempo discreto. A aplicação do método pode ser modelada matematicamente pela relação seguinte:

$$\mathcal{H}[z] = \mathcal{H}(s)|_{s=\frac{2}{T_s}\cdot\frac{z-1}{z+1}} \tag{62}$$

2.7.3.2 Empenamento em frequência

O método da transformação bilinear faz o mapeamento do semi-plano esquerdo do plano s no círculo unitário do plano z, porém, a relação entre as frequências mapeadas não se estabelece de forma linear, uma vez que a frequência do filtro analógico $\mathcal{H}(s)$ denotada por ω_a e a frequência de $\mathcal{H}[z]$ denotada por ω_d tem um mapeamento onde $-\infty < \omega_a < \infty$ deve equivaler a $-\frac{\pi}{T_s} \leq \omega_d \leq \frac{\pi}{T_s}$, e é um mapeamento de um intervalo infinito em um limitado.

Para ver isso, basta analisar a resposta em frequência dos filtros fazendo a análise desprezando os termos responsáveis pelo regime transitório. Matematicamente:

$$\mathcal{H}_d[z] = \mathcal{H}_a(s)|_{s=\frac{2}{T_s}\cdot\frac{z-1}{z+1}} \tag{63}$$

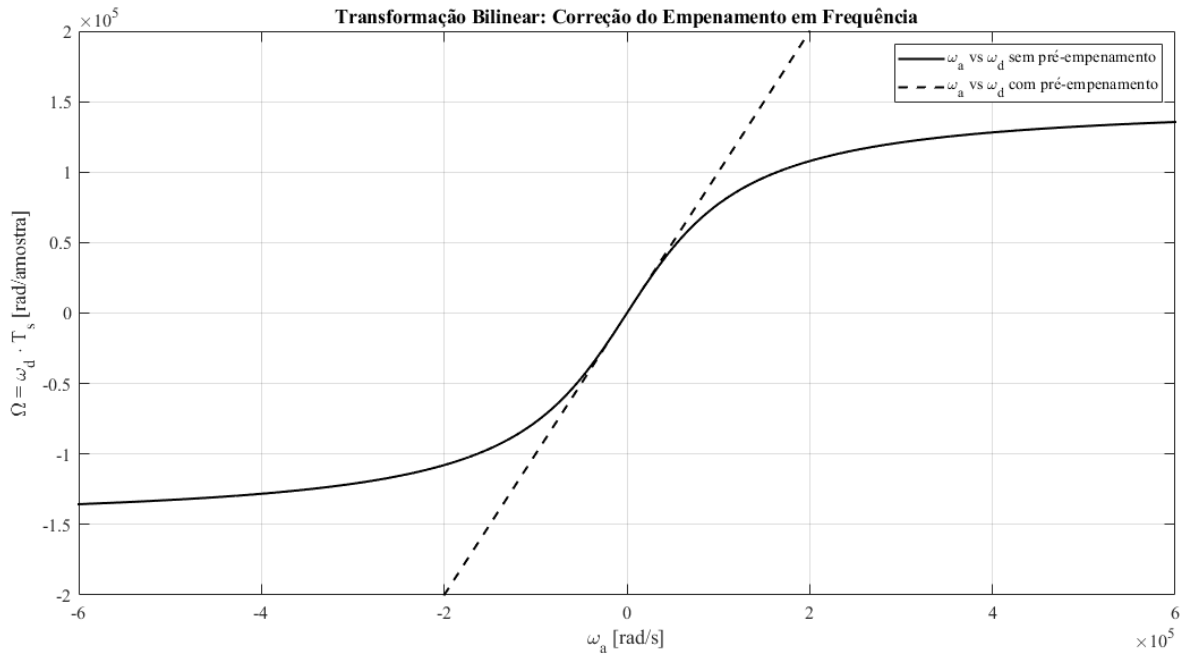
$$\mathcal{H}_d[z] = \mathcal{H}_a\left(\frac{2}{T_s}\cdot\frac{z-1}{z+1}\right) \tag{64}$$

$$\mathcal{H}_d\left[e^{j\omega_d T_s}\right] = \mathcal{H}_a\left(\frac{2}{T_s}\cdot\frac{e^{j\omega_d T_s}-1}{e^{j\omega_d T_s}+1}\right) = \mathcal{H}_a\left(j\omega_a\right) = \mathcal{H}_d\left(j\frac{2}{T_s}\tan\omega_d\frac{T_s}{2}\right) \tag{65}$$

Sendo ω_a a frequência angular em domínio analógico e ω_d a frequência angular em domínio analógico que desejamos mapear para o círculo unitário, assim, obtendo:

$$\omega_a = \frac{2}{T_s}\tan\left(\frac{\omega_d\cdot T_s}{2}\right) \Leftrightarrow \omega_d = \frac{2}{Ts}\cdot\operatorname{atan}\left(\frac{\omega_a\cdot T_s}{2}\right) \tag{66}$$

Figura 22 - Comportamento gráfico do empenamento em frequência



Fonte: Autor.

A partir do gráfico acima é possível observar que a distorção não afeta baixas frequências, pois $\theta \to 0 \Rightarrow \tan\theta \to \theta$, assim, para baixas frequências, $\omega_a \approx \omega_d$. Para contornar esse efeito em frequências elevadas, é utilizado uma técnica denominada pré-empenamento (da língua inglesa, *pre-warping*) que é útil para compensar a distorção em frequência que a transformação bilinear gera. Para tanto, é feita a seguinte definição:

$$s \approx \frac{\omega}{\tan\left(\frac{\Omega}{2}\right)} \cdot \frac{1 - z^{-1}}{1 + z^{-1}} \tag{67}$$

Sendo ω a frequência analógica que desejamos mapear para o plano z e Ω a frequência discreta definida a partir de ω como:

$$\Omega = \omega \cdot T_s = \frac{2 \cdot \pi \cdot f_c}{f_s} \tag{68}$$

Podemos observar a linearização da relação entre ω_a e ω_d provocada pela técnica do pré-empenamento através da figura 22.

2.8 PLACA DE DESENVOLVIMENTO RASPBERRY PI

O *Raspberry PI* é um microcomputador que contém em sua placa todo o hardware necessário para a comunicação com quaisquer outros dispositivos que sejam compatíveis com suas entradas/saídas.

O circuito foi desenvolvido pela instituição sem fins lucrativos denominada *Raspberry PI Foundation*, com o objetivo primário de fomentar o ensino da ciência de computação básica em escolas, mas devido ao custo-benefício do hardware, acabou se popularizando no mercado tornando-se um dos principais microcomputadores do mundo.

Existem diversos modelos do *Raspberry*, com diferenças de hardware que variam de acordo com a demanda do projeto e para o equalizador paramétrico objeto desta monografia foi utilizado o *Raspberry Pi 4 Model B* que dispõe de memória RAM de 8GB, modelo mais atual e de maior configuração possível desenvolvido e lançado ao mercado pela marca, até a publicação deste documento, cujo hardware está apresentado a seguir:

Figura 23 - Vista superior da placa de desenvolvimento.

Fonte: *Raspberry PI Foundation*, 2020.

As especificações sobre o modelo da placa utilizada são apresentadas pela tabela a seguir.

Tabela 1 - Especificações *Raspberry Pi – 4B*

Especificações	
Processador	*Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1,5 GHz*
Memória	*8 GB*
Conectividade	*5,0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0, BLE* *Gigabit Ethernet* *2x USB 3.0 ports* *2x UBS 2.0 ports*
GPIO	*40 I/O*
Vídeo e Som	*2x micro-HDMI ports (up to 4Kp60 supported);* *2-lane MIPI DSI display port* *2-lane MIPI CSI camera port* *4-pole stereo áudio and composite vídeo port.*
Alimentação	*5V DC*

Fonte: *Raspberry PI Foundation*, 2020.

O sistema operacional Linux contido na CPU do *Raspberry* pode ser encontrado na literatura como *Raspbian ou* atualmente *Raspberry Pi OS* e é um sistema *open source* baseado na distribuição do sistema operacional Linux *Debian*, leve e configurável de acordo com as preferências do usuário, ou seja, possui acesso irrestrito ao código do componente, podendo realizar alterações diretas em suas estruturas.

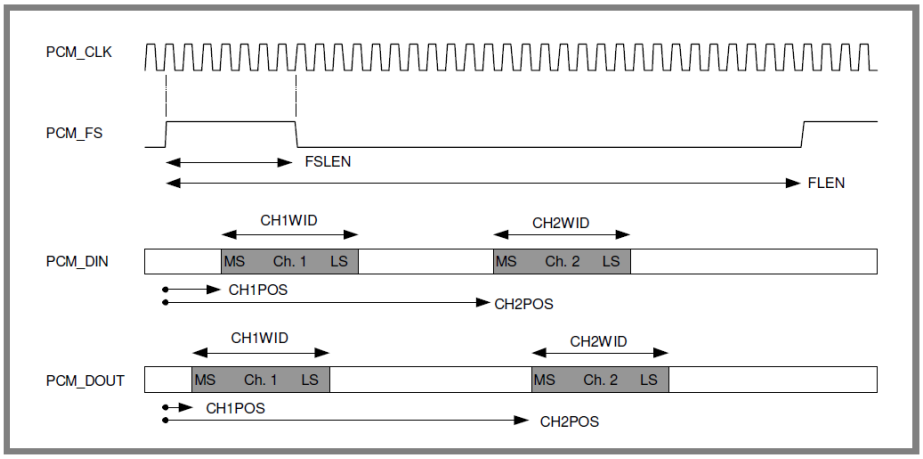
Dentro da IDE do *Raspberry*, podem ser configuradas rotinas com as entradas e saídas do hardware através de linguagens de programação, como por exemplo a linguagem de alto nível C++ utilizada neste estudo, o que viabilizou a programação desde simples filtros digitais através dos cálculos dos coeficientes adequados, até a implementação de um sistema mais complexo, como a filtragem em tempo real para áudios em plataformas de streaming com um sistema de aquisição integrado, objetivo principal deste desenvolvimento.

2.8.1 Processador BCM2711 – Periférico de áudio:

Trata-se de um periférico componente de um barramento APB (Barramento de Periférico Avançado), que proporciona ao objeto de estudo alta performance, uma vez que suporta formatos PCM compatíveis aos modulados e demodulados durante a equalização dos sinais gerados em tempo real do sistema desenvolvido ao longo deste documento.

Este periférico utiliza-se de serialização bit a bit para entradas e saídas, onde o bit mais significativo é sempre priorizado, e é utilizada a combinação de quatro sinais, com funções dispostas como na figura abaixo e explicadas logo a seguir.

Figura 24 - Interface de sincronização de áudio PCM



Fonte: *Broadcom Corporation*, 2012.

Destacando na imagem CLK como os pulsos de relógio (denominados como *clock*) que podem ser assíncronos em relação ao *clock* do barramento e logicamente invertidos a depender da necessidade do projeto, *Din* como entrada serial da mensagem, *Dout* como saída serial e *FS*

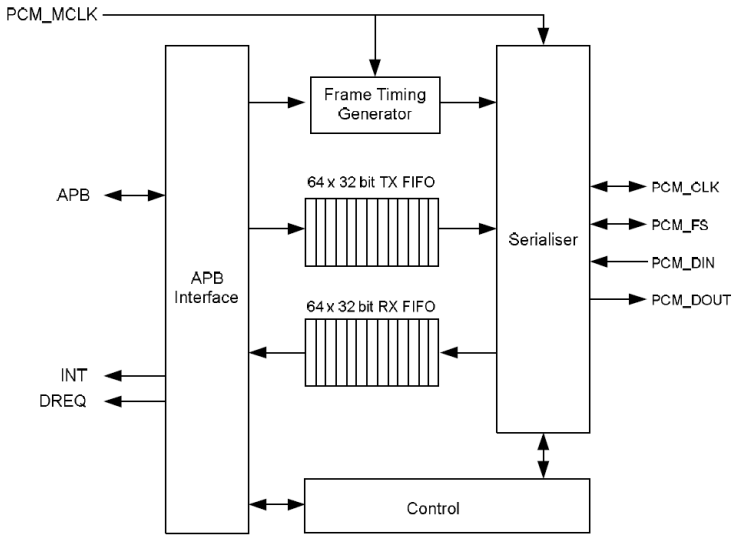
como sinal de sincronização de frames/quadros, pode-se verificar as funcionalidades que o periférico proporciona ao usuário.

O sinal de sincronização de frames merece ser enfatizado, uma vez que realiza a separação do sinal serial em frames de tamanho e posição programáveis individualmente, pois dispõe de até dois canais em cada direção, que podem ser compostos de 8 até 32 bits, estes que podem estar situados em qualquer posição do quadro, exceto sobrepostos.

Devido à possibilidade de configuração individual do sinal de sincronização de quadros e do *clock* do sistema, o periférico oferece também a versatilidade de poder operar tanto como mestre do projeto, quanto como escravo.

No diagrama de blocos abaixo, pode-se compreender ainda melhor o funcionamento deste periférico e percebe-se que transmissores e receptores operam em sistema de memória FIFO (da língua inglesa, *first-in first-out,* ou seja, primeiro bit que entra é o primeiro que sai), e que mesmo na incidência de dois canais, apenas um transmissor/receptor se faz presente, ou seja, a informação de ambos os canais é entreposta.

Figura 25 - Diagrama de blocos da interface de áudio PCM.



Fonte: *Broadcom Corporation*, 2012.

2.9 JUCE

O JUCE é uma plataforma de aplicação *framework* (Conjunto de bibliotecas e funções) na linguagem C++, sendo um software muito utilizado para o desenvolvimento de aplicações de áudio e ambientes de desenvolvimento GUI (interface gráfica de usuário). Um dos

diferenciais do JUCE é a compatibilidade com sistemas operacionais distintos, como macOS, Linux e Windows, de modo que um código escrito no Windows possa ser executado por um compilador Linux sem muita dificuldade, sendo essencial para o projeto do equalizador paramétrico, pois, permite o desenvolvimento em múltiplas plataformas.

Lançado pela primeira vez em 2004, o JUCE foi desenvolvido por Julian Storer para criar recursos gráficos e de áudio *Waveform*. Em 2014 a plataforma foi adquirida pela fábrica de hardware ROLI, sendo que em 2020 a ROLI vendeu o produto para a empresa de tecnologia de áudio PACE, o que fez o software ser ainda mais dedicado ao desenvolvimento de aplicações de áudio.

Figura 26 - Logotipo JUCE

Fonte: Repositório JUCE, 2021.

Semelhantes aos outros *frameworks* disponíveis, o JUCE contém classes que fornecem as mais variadas funções para a edição de GUI, gráficos, processamento de áudio, análise XML e JSON, networking, criptografia e muitos outros recursos. A grande gama de bibliotecas, faz com que o JUCE possa consolidar o desenvolvimento de um aplicativo sem a necessidade de precisar utilizar softwares de terceiros. Uma das inspirações que podemos encontrar nesse tipo de programação é o JDK do JAVA, sendo o JUCE o semelhante para a linguagem C++.

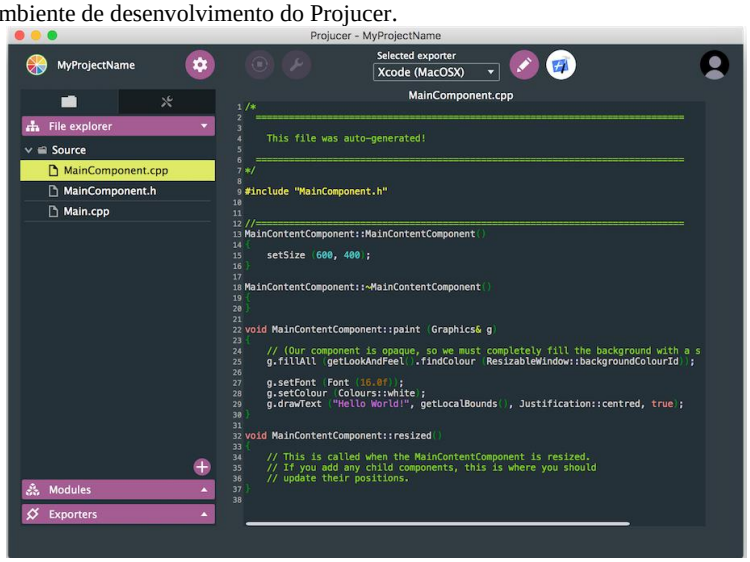
A ampla utilização do JUCE para o processamento de áudio, se deve pelo fato de o software ser concebido por Julian Storer com a estrutura de um *Tracktion* (Agora conhecido com *Waveform*, é uma estação de trabalho de áudio digital, utilizado para gravação e processamento de áudio e MIDI), antes de ser dividido em um produto independente. Além disso o amplo suporte a *drivers* de áudio (CoreAudio, ASIO, ALSA, JACK, WASAPI e DirectSound são exemplos), reprodutores MIDI – Sigla do padrão técnico que descreve o protocolo de comunicações, interface e conectores que interligam instrumentos musicais, computadores e dispositivos de áudio, sintetizadores polifônicos, leitores para os mais diversos CODECS de áudio integrados ao software, *wrappers* para a produção de plugins de áudio, efeitos e ferramentas VST (plugin de áudio que integra sintetizadores digitais e unidades de

efeito em *Tracktion's*). Tudo isso levou o JUCE ser amplamente utilizado no processamento de áudio pelos desenvolvedores.

2.9.1 Projucer

A ferramenta IDE que permite criar e principalmente gerenciar os projetos realizados através do JUCE é chamada de *Projucer*. Após a parametrização das configurações de um projeto realizado na plataforma JUCE, o projucer pode ser utilizado para gerar automaticamente uma coleção de arquivos de projetos fonte para diferentes frameworks, o que permite que o projeto seja compilado no sistema nativo do destino. Atualmente ele pode gerar projetos em Xcode, Visual Studio, Linux Makefiles, Android Ant e projetos para CodeBlocks.

Figura 27 - Ambiente de desenvolvimento do Projucer.



Fonte: JUCE *documentation*, 2021.

Outra propriedade entregue pela IDE, além de gerenciar e configurar todos os projetos em desenvolvimento, ele também possui um editor de código, gerenciador de GUI, assistente de desenvolvimento de novos projetos e um mecanismo de codificação ao vivo, muito útil na depuração do código realizado e no suporte para a criação de interfaces para o usuário.

No desenvolvimento do presente projeto, essa facilidade de transporte de projeto entre *frameworks* de sistemas operacionais diferentes foi de grande utilidade, devido a possibilidade da codificação e depuração do sistema equalizador paramétrico a distância.

3 METODOLOGIA

O desenvolvimento de filtros digitais consiste na determinação de uma equação de diferenças que realize a aproximação da resposta em frequência desejada. O projeto tem como considerações iniciais o tipo de sinal a ser processado, a frequência de amostragem utilizada e a característica da resposta em frequência desejada para os filtros. A técnica utilizada para o desenvolvimento dos filtros consiste na modelagem das funções de transferência do sistema analógico em domínio *s*, cuja resposta em frequência atende os critérios desejados, em seguida aplica-se a transformação bilinear, cuja realiza o mapeamento da função no domínio s para o domínio z de maneira a determinar a função de transferência do filtro digital correspondente a do filtro analógico. A partir das funções de transferências no domínio z, determinamos as equações de diferenças de cada filtro digital.

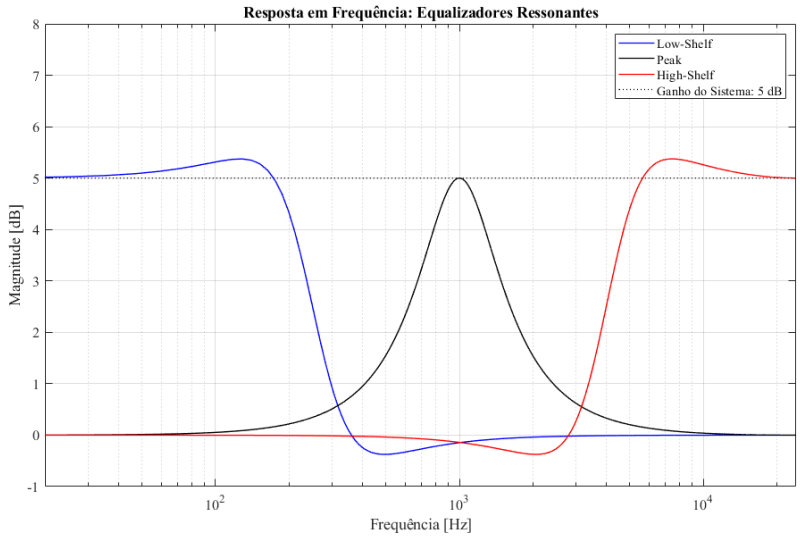
3.1 EQUALIZADORES RESSONANTES

Os filtros digitais desenvolvidos possuem sua resposta em frequência modelada a partir das respostas em frequência de equalizadores ressonantes analógicos. Equalizadores atuam fornecendo um ganho para uma determinada banda ou frequência prescrita, enquanto mantém inalterada as outras frequências fora de sua banda de atuação. A implementação do filtro digital será feita através de um filtro IIR biquadrático o qual aproxima a resposta em frequência de três equalizadores ressonantes:

a. Equalizador *Low-Shelving*.
b. Equalizador *High-Shelving*.
c. Equalizador *Peak.*

A figura a seguir ilustra o caráter dessas respostas em frequência a partir de suas funções em domínio s.

Figura 28 - Respostas em frequência de equalizadores ressonantes.



Fonte: Autor.

As curvas ilustradas pela figura acima são dadas pelas seguintes funções de transferência:

$$\mathcal{H}_L(\hat{s}) = A \cdot \frac{\hat{s}^2 + \frac{\sqrt{A}}{Q} \cdot \hat{s} + A}{A \cdot \hat{s}^2 + \frac{\sqrt{A}}{Q} \cdot \hat{s} + 1} \tag{69}$$

$$\mathcal{H}_H(\hat{s}) = A \cdot \frac{A \cdot \hat{s}^2 + \frac{\sqrt{A}}{Q} \cdot \hat{s} + 1}{\hat{s}^2 + \frac{\sqrt{A}}{Q} \cdot \hat{s} + A} \tag{70}$$

$$\mathcal{H}_P(\hat{s}) = \frac{\hat{s}^2 + \frac{A}{Q} \cdot \hat{s} + 1}{\hat{s}^2 + \frac{\hat{s}}{A \cdot Q} + 1} \tag{71}$$

As funções $\mathcal{H}_L(\hat{s})$, $\mathcal{H}_H(\hat{s})$ e $\mathcal{H}_P(\hat{s})$ são as funções de transferência normalizadas dos equalizadores *low-shelf*, *high-shelf* e *peak* respectivamente, ou seja, o termo s está normalizado pela frequência de ressonância ω, desta forma, sendo $\hat{s}$ é descrito como:

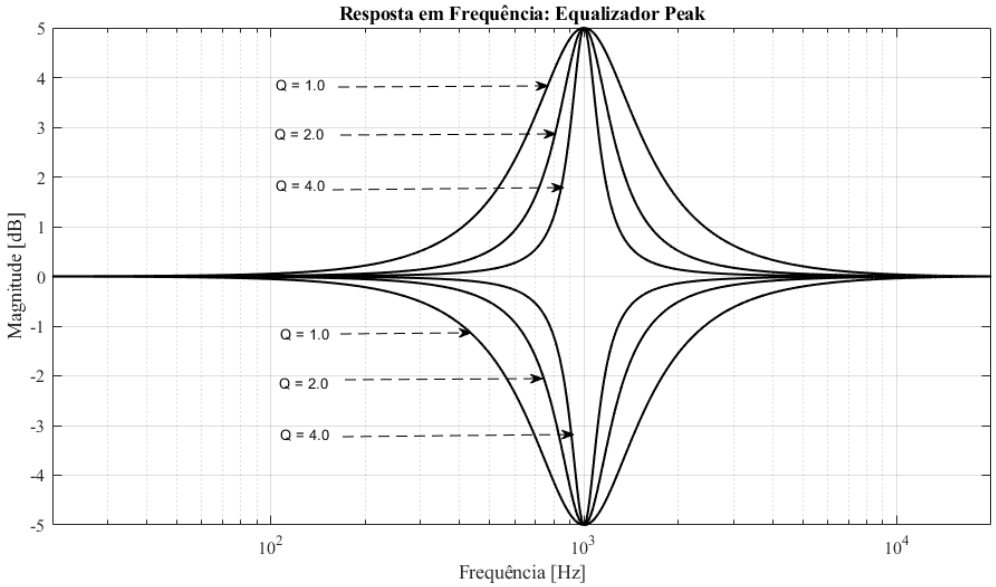
$$\hat{s} = \frac{s}{\omega} \tag{72}$$

Os termos "A" e "Q" se referem ao ganho em escala linear do equalizador e ao índice de mérito respectivamente. O índice de mérito, referido também como fator de qualidade, é dado pela relação entre a frequência de ressonância e a largura de banda do equalizador em Hertz.

$$Q = \frac{f_c}{BW} \tag{73}$$

Aumentar o fator Q, implica em reduzir a largura de banda, e por sua vez reduzi-lo implica em um aumento na largura de banda. O controle de banda é uma qualidade notável em equalizadores paramétricos, podendo remover ou ampliar frequências com alta precisão. Podemos observar o comportamento do fator de qualidade para o equalizador $\mathcal{H}_P(\hat{s})$ na figura abaixo.

Figura 29 - Respostas em frequência do equalizador *peak* com fator Q variável.



Fonte: Autor.

Ao analisar as funções de transferência podemos explicitar determinadas características de cada equalizador com relação a sinais contínuos e sinais com frequência acima da frequência de Nyquist. Ao aproximar a frequência complexa de Laplace 's' para uma frequência nula, obtemos os seguintes resultados para cada equalizador:

$$\lim_{s\to 0} \mathcal{H}_L(\hat{s}) = A^2 \tag{74}$$

$$\lim_{s\to 0} \mathcal{H}_H(\hat{s}) = 1 \tag{75}$$

$$\lim_{s\to 0} \mathcal{H}_P(\hat{s}) = 1 \tag{76}$$

Por sua vez, ao aproximar ‘s’ a uma frequência elevada, temos:

$$\lim_{s\to \infty} \mathcal{H}_L(\hat{s}) = 1 \tag{77}$$

$$\lim_{s\to \infty} \mathcal{H}_H(\hat{s}) = A^2 \tag{78}$$

$$\lim_{s\to \infty} \mathcal{H}_P(\hat{s}) = 1 \tag{79}$$

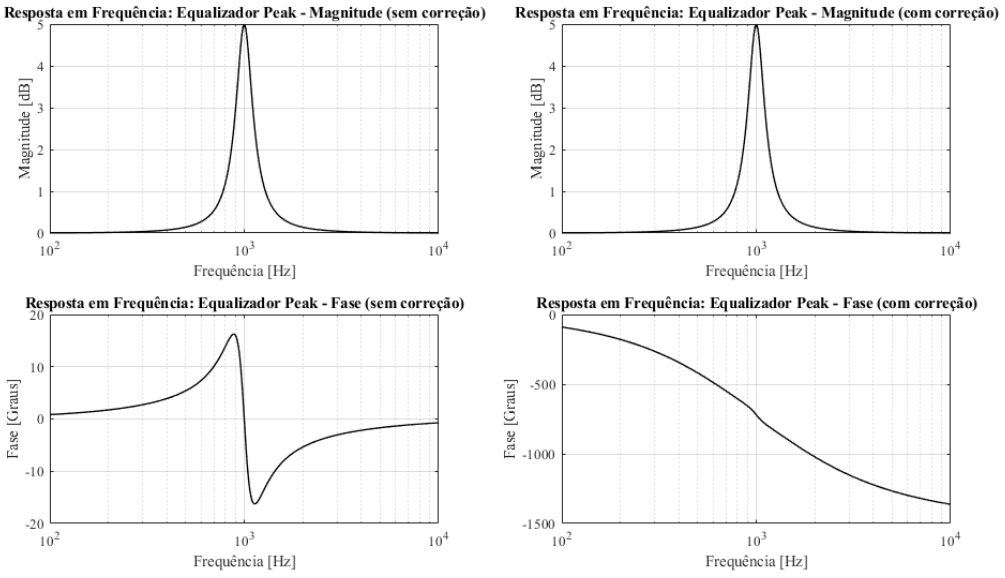
Observando atentamente, é perceptível que o equalizador $\mathcal{H}_L(\hat{s})$ possui um ganho para sinais contínuos e ganho unitário para sinais acima da frequência de Nyquist, enquanto o equalizador $\mathcal{H}_H(\hat{s})$ possui ganho unitário para frequências contínuas e para frequências elevadas possui um ganho definido. Os ganhos definidos por $\mathcal{H}_H(\hat{s})$ e $\mathcal{H}_L(\hat{s})$ são indesejáveis no projeto de equalizadores de áudio, pois níveis de tensão contínuos são indesejáveis em equipamentos de áudio, devido a possíveis danos aos transdutores presentes nos alto-falantes ou fones de ouvido, enquanto frequências acima da taxa de amostragem prescrita pela frequência de Nyquist podem produzir ruídos causados por sobre-amostragem do sinal de áudio. Os problemas de sobre-amostragem e nível contínuo são contornados através de filtros analógicos empregados antes da etapa de amostragem, como por exemplo o filtro *anti-aliasing.* Apesar destas correções estarem presentes em grande maioria de placas de áudio digitais, foi empregado ao projeto do equalizador apenas $\mathcal{H}_P(\hat{s})$, pois este possui ganho definido apenas pela frequência a qual foi sintonizado, podemos observar este comportamento com mais clareza a partir do gráfico ilustrado pela figura 28.

3.1.1 O Filtro Passa-Todas e Fases Não-Lineares

Uma característica intrínseca de filtros IIR é a não-linearidade de sua fase, onde em várias aplicações é uma característica apreciada, contudo existem situações em que a fase linear é uma métrica necessária. Embora fases lineares estejam presentes apenas em filtros FIR, existem métodos de linearização de fase para filtros IIR, que constitui em uma cascata de filtros passa-todas (do inglês *All-Pass Filter*).

Um filtro de fase não-linear produz atrasos desproporcionais para as componentes em frequência de um sinal, desta forma, produzindo distorções. A correção através de filtros passa-todas consiste em linearizar a resposta em fase do filtro, convoluindo as respostas em fase do filtro em questão com a resposta do filtro passa-todas. A figura abaixo estabelece uma comparação entre filtros com fase corrigida e filtros sem correção de fase.

Figura 30 - Comparação entre a aplicação do filtro passa-todas.



Fonte: Autor.

Examinando a figura acima verifica-se que a magnitude da resposta do equalizador *peak* se manteve inalterada, enquanto sua fase sofreu uma linearização proveniente da estrutura em cascata de quatro filtros passa-todas de segunda ordem, resultando em um filtro de oitava ordem. A função de transferência de um filtro passa-todas analógico é definida como:

$$\mathcal{H}_A(s) = \frac{\mathcal{H}_{a^*}(s)}{\mathcal{H}_a(s)} \tag{80}$$

Sendo $\mathcal{H}_a(s)$ uma função de transferência que pode ser expandida da seguinte forma:

$$\mathcal{H}_a(s) = s^N + a_1 \cdot s^{N-1} + \cdots + a_{N-1} \cdot s + a_N \tag{81}$$

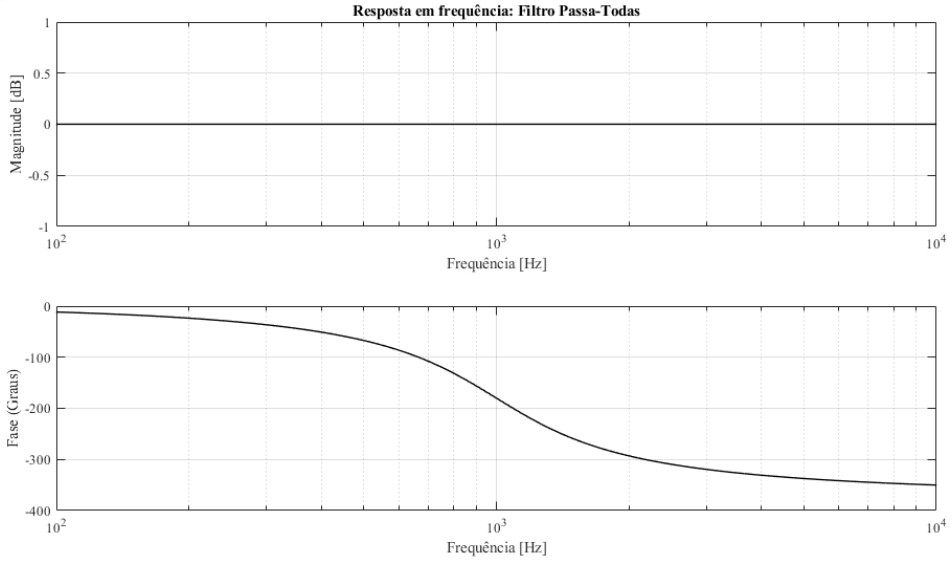
A função $\mathcal{H}_{a^*}(s)$ é o conjugado da função acima. A estrutura do filtro passa-todas permite um ganho unitário uniforme para todas as frequências enquanto sua fase apresenta comportamento o mais linear possível, isso se deve pelo comportamento de seus polos e zeros, pois se $\mathcal{H}_{a^*}(s)$ possui raiz em r_s, $\mathcal{H}_a(s)$ terá sua raiz em $-r_s$. A função de transferência normalizada do filtro passa-todas é modelada como:

$$\mathcal{H}_A(\hat{s}) = \frac{\hat{s}^2 - \frac{\hat{s}}{Q} + 1}{\hat{s}^2 + \frac{\hat{s}}{Q} + 1} \tag{82}$$

Neste caso, a frequência complexa de Laplace é normalizada pela frequência a qual o filtro passa-todas foi sintonizado, seguindo a mesma relação dada pela equação (72).

O comportamento de sua resposta em frequência é dado pelo gráfico abaixo.

Figura 31 – Resposta em frequência do filtro passa-todas.

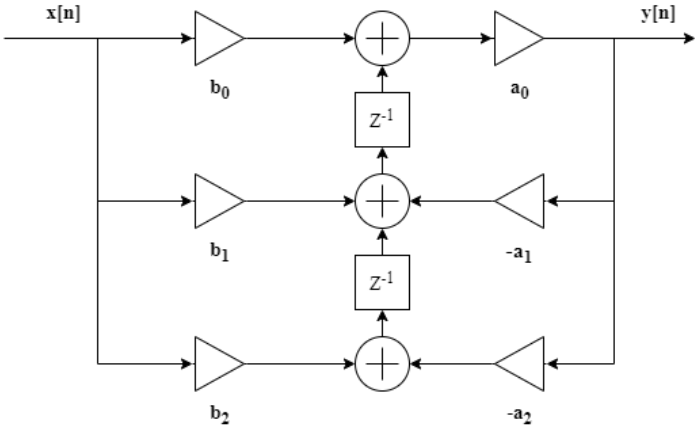


Fonte: Autor.

3.2 FILTROS BIQUADRÁTICOS

O filtro biquadrático é a estrutura principal de um equalizador paramétrico digital, caracteriza-se por ser um sistema de segunda ordem contendo dois polos e dois zeros caracterizados pelas raízes das equações quadráticas no numerador e denominador. A função de transferência para um filtro biquadrático de segunda ordem no domínio z é dada por (83):

Figura 32 - Diagrama de fluxo de um filtro biquadrático de segunda ordem.

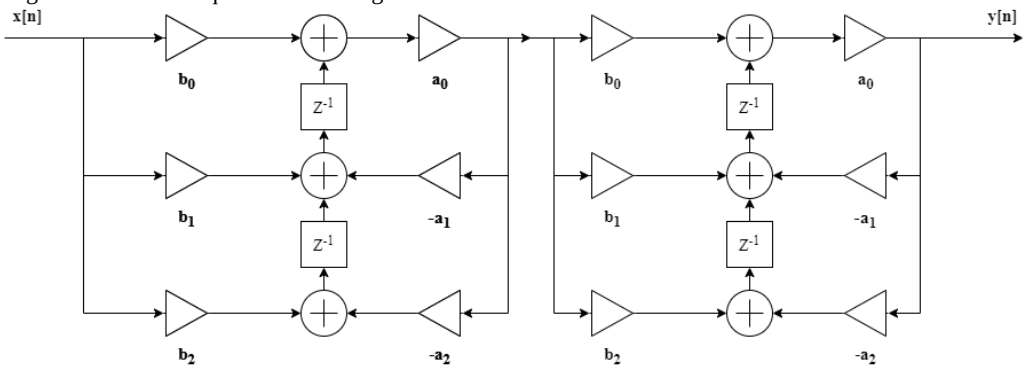


Fonte: Autor.

$$\mathcal{H}_B[z] = \frac{b_0 + b_1 \cdot z^{-1} + b_2 \cdot z^{-2}}{a_0 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2}} \tag{83}$$

Sendo "b" e "a" os coeficientes de tempo contínuo provenientes da função de transferência modelada no domínio s. O filtro biquadrático é altamente sensível à quantização do sinal e ao arredondamento de seus coeficientes, isso implica que o sistema tende a instabilidade conforme a ordem dos polinômios em seu numerador e denominador aumenta. O incremento da ordem do filtro contará com a presença de mais coeficientes sendo multiplicados entre si dentro do processamento e isso pode prejudicar à resposta em frequência por conta da precisão computacional de números decimais. É de grande importância que os coeficientes possuam a maior precisão decimal possível para que o processamento do sinal seja adequado. Para contornar este problema, impõe-se que os filtros sejam de segunda ordem, e então para aumentar a ordem do sistema constrói-se uma estrutura em cascata conforme ilustrado a seguir.

Figura 33 - Filtros biquadráticos de segunda ordem em cascata.



Fonte: Autor.

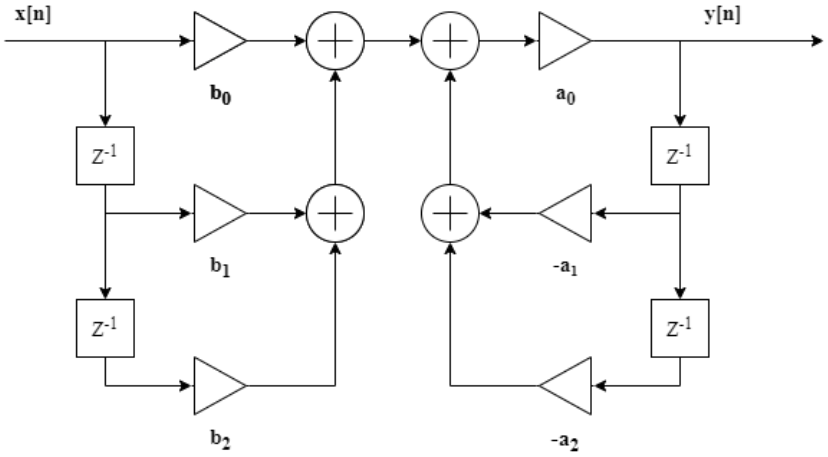
A estrutura em seções de segunda ordem (do inglês SOS, *second-order sections*), permite a construção de filtros de ordem superior, porém limitados a ordens pares. Essa técnica permite melhorar a seletividade da resposta em frequência sem comprometer os valores dos coeficientes, garantindo a estabilidade do sistema.

3.2.1 Estruturas de um filtro biquadrático

A estrutura de um filtro biquadrático pode influenciar no processamento do sinal, dependendo do posicionamento dos coeficientes dentro do fluxo operacional e o número de elementos de atraso podemos alterar a performance do filtro de maneira a melhorar a sua implementação no processador. Os elementos de um filtro digital, discutidos na seção 2.7, ao serem arranjados dentro de seu diagrama, acarretam determinados benefícios dependendo do circuito digital a ser utilizado para o processamento. Para o projeto em questão, foi definida como métrica o menor uso de memória possível, portanto a estrutura implementada neste trabalho é a forma direta I.

Existem quatro tipos de variações da forma direta: Forma Direta I, Forma Direta II e Formas Direta I e II transpostas. A Forma Direta I é caracterizada pelo diagrama de fluxo ilustrado pela figura 34.

Figura 34 - Filtro biquadrático de segunda ordem na forma direta I.



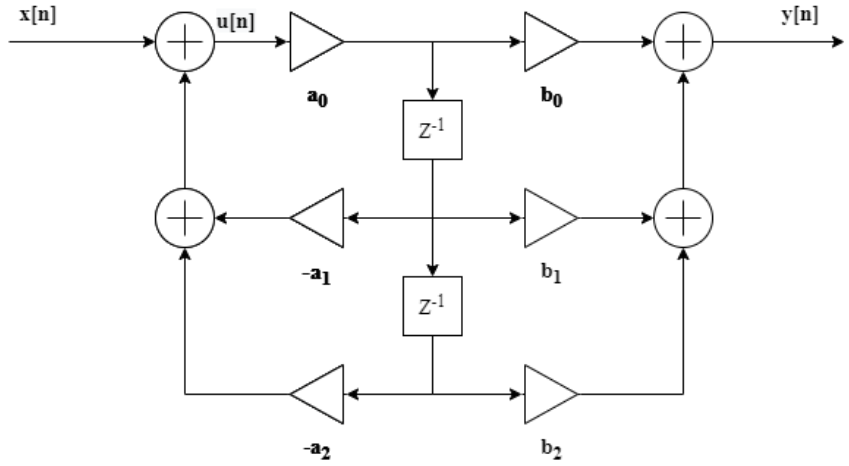
Fonte: Autor

A implementação do filtro na forma direta I é caracterizada por um filtro de apenas zeros (seção de coeficientes "b") em série com um filtro de apenas polos (seção de coeficientes "a", parte recursiva do filtro). Esta configuração, aplicada em sistemas digitais com aritmética computacional de ponto fixo em complemento de 2, minimiza as chances de transbordamento da memória, fenômeno denominado como *overflow*. Ao minimizar o risco de *overflow* a computação do sinal discreto é otimizada. Entretanto a forma direta I possui o problema ter duas vezes mais elementos de atraso que as outras configurações, o que acarreta um maior uso de memória para a implementação do filtro. A equação de diferenças de um filtro biquadrático de segunda ordem na forma direta I é dada abaixo:

$$a_0 \cdot y[n] = \sum_{i=0}^{2} b_i \cdot x[n\ -\ i] - \sum_{i=1}^{2} a_i \cdot y[n\ -\ i] \tag{84}$$

A segunda configuração é a forma direta II, caracterizada pelo diagrama de fluxo a seguir.

Figura 35 - Filtro biquadrático de segunda ordem na forma direta II.



Fonte: Autor.

A forma direta II caracteriza-se por um filtro de apenas polos em série com um filtro de apenas zeros. Esta configuração também é chamada de forma canônica, pois possui apenas dois elementos de atraso o que minimiza o uso de memória para a implementação do filtro. Porém a forma direta II possui risco de *overflow* o que pode resultar em distorções no processamento do sinal discreto. A equação de diferenças da forma direta II é descrita através das duas equações abaixo.

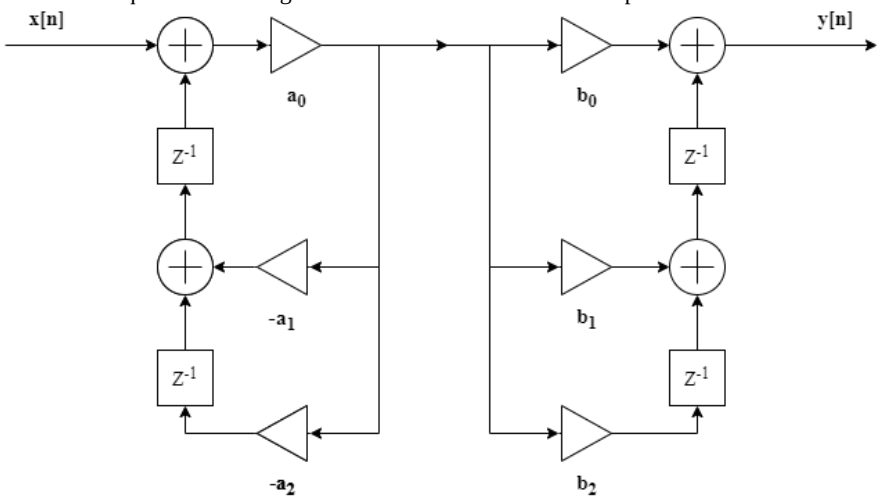
$$a_0 \cdot u[n] = x[n] - a_1 \cdot u[n-1] - a_2 \cdot u[n-2] \tag{85}$$

$$y[n] = b_0 \cdot u[n] + b_1 \cdot u[n-1] + b_2 \cdot u[n-2] \tag{86}$$

A equação (85) corresponde à seção de polos, enquanto a equação (86) corresponde à seção de zeros.

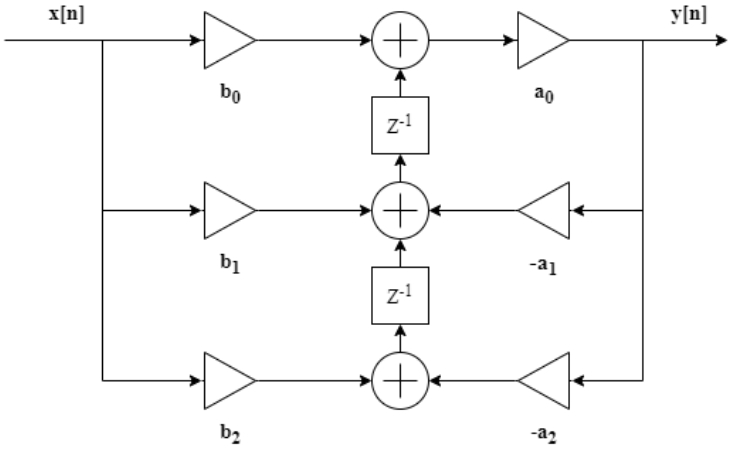
As duas últimas formas de implementação são obtidas através da transposição dos diagramas de fluxo anteriores, ou seja, revertendo as posições das seções de polos e zeros das formas diretas I e II. A transposição da estrutura de sistemas LTI do tipo SISO não altera a função de transferência, esta conclusão pode ser atingida a partir do teorema de ganho de Mason. A forma direta I transposta e a forma direta II transposta são representadas pelos diagramas de fluxo a seguir.

Figura 36 - Filtro biquadrático de segunda ordem na forma direta I transposta.



Fonte: Autor.

Figura 37 - Filtro biquadrático de segunda ordem na forma direta II transposta.



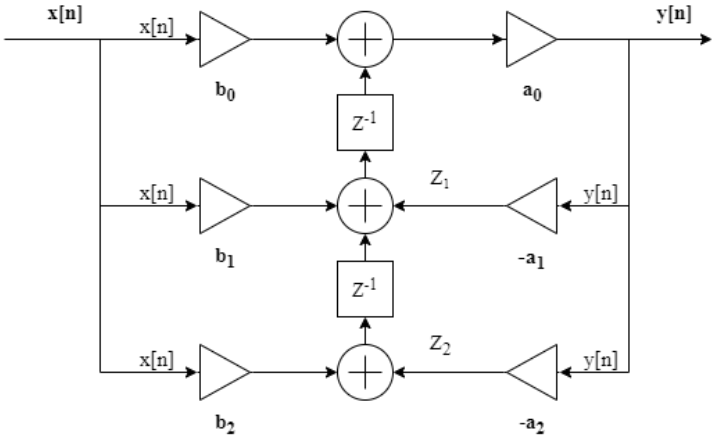
Fonte: Autor.

As formas transpostas dos diagramas de fluxo mantêm as equações de diferenças tendo em vista que sua função de transferência não é afetada pelo rearranjo das seções de polos e zeros, entretanto, este arranjo apresenta mudanças na resposta em frequência, mais especificamente nas regiões de transição do filtro. A forma direta II transposta é a utilizada para o desenvolvimento dos filtros digitais do equalizador paramétrico devido ao fato de em sua estrutura os zeros precederem os polos no fluxo do processamento, isto produz transições abruptas em determinadas frequências da banda atuante por conta da compensação atenuante produzida pelos zeros em série com o ganho elevado dos polos. Essa é uma característica presente nos filtros *low shelf*, *high shelf* e *peak*. É importante ressaltar que de todas as formas diretas discutidas nesta seção, a forma direta II transposta é a que possui o menor número de

elementos somadores e de atraso, o que é uma qualidade visando o critério de baixo uso de memória.

Observando o diagrama de fluxo da forma direta II, é possível obter sua equação de diferenças do sistema.

Figura 38 - Diagrama de fluxo da forma direta II transposta.



Fonte: Autor.

Observando a figura acima, e realizando as operações presentes no diagrama de fluxo, temos o seguinte sistema de equações:

$$\begin{cases} a_0 \cdot y[n] = b_0 \cdot x[n] + Z_1 \\ Z_1 = b_1 \cdot x[n] - a_1 \cdot y[n] + Z_2 \\ Z_2 = b_2 \cdot x[n] - a_2 \cdot y[n] \end{cases} \quad (87)$$

Aplicando as operações de atraso, expandimos o sistema de equações:

$$a_0 y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2] \quad (88)$$

$$Z_1 = b_1 \cdot x[n] - a_1 \cdot y[n] + b_2 \cdot x[n-1] - a_2 \cdot y[n-1] \quad (89)$$

$$Z_2 = b_2 \cdot x[n] - a_2 \cdot y[n] \quad (90)$$

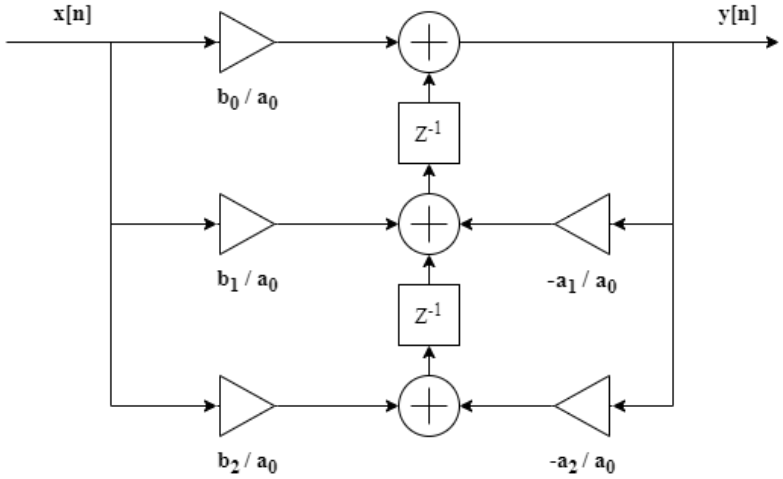
A equação de diferenças da forma direta II transposta é dada pela equação (88).

Isolando o sinal de saída $y[n]$ na equação de diferenças, divide-se ambos os lados da igualdade pelo coeficiente a_0, resultando em:

$$y[n] = \frac{b_0}{a_0}x[n] + \frac{b_1}{a_0}x[n-1] + \frac{b_2}{a_0}x[n-2] - \frac{a_1}{a_0}y[n-1] - \frac{a_2}{a_0}\,y[n-2] \tag{91}$$

Os coeficientes acima estão normalizados pelo coeficiente a_0, sendo estes efetivamente empregados para o filtro digital em forma direta II transposta.

Figura 39 - Forma direta II transposta com coeficientes normalizados por a_0.



Fonte: Autor.

3.3 CÁLCULO DOS COEFICIENTES E A TRANSFORMAÇÃO BILINEAR

Conhecendo a função de transferência em domínio s e a topologia do filtro digital a ser aplicada, a próxima etapa é computar os coeficientes do filtro digital a partir da transformação bilinear. Levando em consideração o empenamento em frequência, aproxima-se a frequência complexa normalizada '$\hat{s}$' conforme abaixo.

$$\hat{s} \approx \frac{1}{\tan\left(\frac{\Omega}{2}\right)} \cdot \frac{1 - z^{-1}}{1 + z^{-1}} \tag{92}$$

Fazendo uso da seguinte identidade trigonométrica:

$$\tan\left(\frac{\theta}{2}\right) = \frac{\text{sen}\ (\theta)}{1 + \cos\ (\theta)} \tag{93}$$

Resulta na seguinte substituição:

$$\hat{s} \approx \frac{1 + \cos\ (\Omega)}{\text{sen}\ (\Omega)} \cdot \frac{1 - z^{-1}}{1 + z^{-1}} \tag{94}$$

Multiplicando o numerador e denominador pelo termo $1 + z^{-1}$, podemos exprimir a aproximação da seguinte forma:

$$\hat{s} \approx \frac{1 + \cos\ (\Omega)}{\text{sen}\ (\Omega)} \cdot \frac{1 - z^{-2}}{1 + 2 \cdot z^{-1} + z^{-2}} \tag{95}$$

A aproximação acima representa transformação bilinear da variável '$\hat{s}$' para o domínio z. Utilizando as aproximações (95) e (92) podemos encontrar as respectivas transformações para os termos "$\hat{s}^2$" e "$\hat{s}^0 = 1$". Utilizando a seguinte identidade trigonométrica:

$$\tan^2\left(\frac{\theta}{2}\right) = \frac{1 - \cos\ (\theta)}{1 + \cos\ (\theta)} \tag{96}$$

Elevando os dois lados da aproximação dada em (92) ao quadrado, obtém-se

$$\hat{s}^2 \approx \left(\frac{1}{\tan\left(\frac{\Omega}{2}\right)}\right)^2 \cdot \frac{1 - 2 \cdot z^{-1} + z^{-2}}{1 + 2 \cdot z^{-1} + z^{-2}} \tag{97}$$

A partir da relação trigonométrica apresentada em (96):

$$\left(\frac{1}{\tan\left(\frac{\Omega}{2}\right)}\right)^2 = \frac{1}{\tan^2\left(\frac{\Omega}{2}\right)} = \frac{1 + \cos\ (\Omega)}{1 - cos(\Omega)} \tag{98}$$

A transformação bilinear para '$\hat{s}^2$' é dada como:

$$\hat{s}^2 \approx \frac{1+\cos(\Omega)}{1-cos(\Omega)} \cdot \frac{1-2\cdot z^{-1}+z^{-2}}{1+2\cdot z^{-1}+z^{-2}} \tag{99}$$

Dividindo os dois lados da aproximação dada em (94) por '$\hat{s}$'

$$1 = \hat{s}^0 \approx \frac{1+\cos(\Omega)}{\text{sen}(\Omega)} \cdot \frac{1-z^{-2}}{1+2\cdot z^{-1}+z^{-2}} \cdot \left(\frac{1}{\hat{s}}\right) \tag{100}$$

Expandindo o termo $\left(\frac{1}{\hat{s}}\right)$, resulta em:

$$\hat{s}^0 \approx \frac{1+\cos(\Omega)}{\text{sen}(\Omega)} \cdot \frac{1-z^{-2}}{1+2\cdot z^{-1}+z^{-2}} \cdot \frac{\text{sen}(\Omega)}{1+\cos(\Omega)} \cdot \frac{1+2\cdot z^{-1}+z^{-2}}{1-z^{-2}} \tag{101}$$

Simplificando os termos $1-z^{-2}$ e $\text{sen}(\omega_N)$ temos que a transformação do termo '1' é dada por:

$$\hat{s}^0 \approx \frac{1+\cos(\Omega)}{1+\cos(\Omega)} \cdot \frac{1+2\cdot z^{-1}+z^{-2}}{1+2\cdot z^{-1}+z^{-2}} \tag{102}$$

Ao examinar as transformações obtidas em (95), (99) e (102), verificamos que o termo $\frac{1+\cos(\Omega)}{1+2\cdot z^{-1}+z^{-2}}$ é comum em todas as aproximações, o que significa que ele será fatorado ao realizarmos as substituições na função de transferência, portanto simplifica-se o termo, em seguida multiplicamos os numeradores e denominadores presentes nos dois lados da aproximação por $\text{sen}^2(\Omega)$, portanto podemos exprimir as aproximações da seguinte forma.

$$\hat{s} \approx \frac{1-z^{-2}}{\text{sen}(\Omega)} \cdot \frac{\text{sen}^2(\Omega)}{\text{sen}^2(\Omega)} \tag{103}$$

$$\hat{s}^2 \approx \frac{1-2\cdot z^{-1}+z^{-2}}{1-\cos(\Omega)} \cdot \frac{\text{sen}^2(\Omega)}{\text{sen}^2(\Omega)} \tag{104}$$

$$\hat{s}^0 \approx \frac{1 + 2 \cdot z^{-1} + z^{-2}}{1 + \cos(\Omega)} \cdot \frac{\text{sen}^2(\Omega)}{\text{sen}^2(\Omega)} \tag{105}$$

A partir da identidade trigonométrica fundamental:

$$\text{sen}^2(x) + \cos^2(x) = 1 \tag{106}$$

Isolando o termo $\text{sen}^2(x)$, efetuamos a substituição nos numeradores dos termos em (104) e (105), enquanto simplificamos o termo (103), obtendo os resultados a seguir.

$$\hat{s} \approx \frac{1 - z^{-2}}{\text{sen}^2(\Omega)} \cdot \text{sen}\,(\Omega) \tag{107}$$

$$\hat{s}^2 \approx \frac{1 - 2 \cdot z^{-1} + z^{-2}}{1 - \cos(\Omega)} \cdot \frac{1 - \cos^2(\Omega)}{\text{sen}^2(\Omega)} \tag{108}$$

$$\hat{s}^0 \approx \frac{1 + 2 \cdot z^{-1} + z^{-2}}{1 + \cos(\Omega)} \cdot \frac{1 - \cos^2(\Omega)}{\text{sen}^2(\Omega)} \tag{109}$$

Expressando $1 - \cos^2(\Omega)$ como $\big(1 - \cos(\Omega)\big) \cdot (1 + \cos(\Omega)\,)$, resultam nas seguintes simplificações:

$$\hat{s} \approx \frac{1 - z^{-2}}{\text{sen}^2(\Omega)} \cdot \text{sen}\,(\Omega) \tag{110}$$

$$\hat{s}^2 \approx \frac{1 - 2 \cdot z^{-1} + z^{-2}}{\text{sen}^2(\Omega)} \cdot (1 + \cos(\Omega)\,) \tag{111}$$

$$\hat{s}^0 \approx \frac{1 + 2 \cdot z^{-1} + z^{-2}}{\text{sen}^2(\Omega)} \cdot \big(1 - \cos(\Omega)\big) \tag{112}$$

Observando atentamente, é perceptível que o termo $\text{sen}^2(\Omega)$ é comum a todos os denominadores, portanto ele será fatorado, resultando em:

$$\hat{s} \approx (1 - z^{-2}) \cdot \text{sen}\,(\Omega) \tag{113}$$

$$\hat{s}^2 \approx (1 - 2 \cdot z^{-1} + z^{-2}) \cdot (1 + \cos(\Omega)\,) \tag{114}$$

$$\hat{s}^0 \approx (\,1 + 2 \cdot z^{-1} + z^{-2}) \cdot \left(1 - \cos(\Omega)\right) \tag{115}$$

Os termos acima são essenciais para a determinação da função de transferência em domínio *z* a partir da função em domínio *s*, efetuando-se então a transformação bilinear.

A seguir encontra-se a tabela com todas as aproximações simplificadas necessárias para a construção da função de transferência do filtro digital.

Tabela 2 – Tabela de transformações bilineares.

Domínio s	**Domínio z**
$\hat{\boldsymbol{s}}^{\boldsymbol{0}} = \mathbf{1}$	$(1 + 2 \cdot z^{-1} + z^{-2}) \cdot (1 - \cos(\Omega))$
$\hat{\boldsymbol{s}}$	$(1 - z^{-2}) \cdot \text{sen}(\Omega)$
$\hat{\boldsymbol{s}}^{\boldsymbol{2}}$	$(1 + 2 \cdot z^{-1} + z^{-2}) \cdot (1 + \cos(\Omega))$
$\hat{\boldsymbol{s}}^{\boldsymbol{2}} + \mathbf{1}$	$2 \cdot (1 - 2 \cdot \cos(\Omega) \cdot z^{-1} + z^{-2})$

Fonte: "Adaptado de" Texas Instruments Application Report (2010, p. 2)

A partir da tabela acima, podemos efetuar a substituição dos termos no domínio *s* pelos termos em domínio z correspondentes. A função de transferência digital a ser determinada é a do equalizador *peak*, correspondente à equação (71) da seção 3.1. Efetuando as substituições necessárias com auxílio das aproximações presentes na tabela acima, obtemos a seguinte equação.

$$\mathcal{H}_P(\hat{s})|_{s \to z} = \frac{2 \cdot (1 - 2 \cdot \cos(\Omega) \cdot z^{-1} + z^{-2}) + \frac{A}{Q} \cdot (1 - z^{-2}) \cdot \text{sen}(\Omega)}{2 \cdot (1 - 2 \cdot \cos(\Omega) \cdot z^{-1} + z^{-2}) + \frac{(1 - z^{-2}) \cdot \text{sen}(\Omega)}{A \cdot Q}} \tag{116}$$

Efetuando as operações distributivas no numerador e denominador e reagrupando os termos colocando em evidência z^{-1} e z^{-2}, encontra-se:

$$\mathcal{H}_P(\hat{s})|_{s\to z} = \frac{2 + \frac{A}{Q}\text{sen}(\Omega) - 4\cdot\cos(\Omega)\cdot z^{-1} + \left(2 - \frac{A}{Q}\cdot\text{sen}(\Omega)\right)\cdot z^{-2}}{2 + \frac{\text{sen}(\Omega)}{A\cdot Q} - 4\cdot\cos(\Omega)\cdot z^{-1} + \left(2 - \frac{\text{sen}(\Omega)}{A\cdot Q}\right)\cdot z^{-2}} \tag{117}$$

Dividindo o numerador e o denominador por 2, podemos exprimir a função como:

$$\mathcal{H}_P(\hat{s})|_{s\to z} = \frac{1 + A\cdot\frac{\text{sen}\,(\Omega)}{2\cdot Q} - 2\cos(\Omega)\cdot z^{-1} + \left(1 - A\cdot\frac{\text{sen}\,(\Omega)}{2\cdot Q}\right)\cdot z^{-2}}{1 + \frac{1}{A}\cdot\frac{\text{sen}\,(\Omega)}{2\cdot Q} - 2\cos(\Omega)\cdot z^{-1} + \left(1 - \frac{1}{A}\cdot\frac{\text{sen}\,(\Omega)}{2\cdot Q}\right)\cdot z^{-2}} \tag{118}$$

Para fins de simplicidade, atribui-se a letra grega α (*alfa*) para representar o termo $\frac{\text{sen}\,(\Omega)}{2\cdot Q}$, portanto, escrevemos a função conforme abaixo

$$\mathcal{H}_P(\hat{s})|_{s\to z} = \frac{1 + A\cdot\alpha - 2\cdot\cos(\Omega)\cdot z^{-1} + (1 - A\cdot\alpha)\cdot z^{-2}}{1 + \frac{\alpha}{A} - 2\cdot\cos(\Omega)\cdot z^{-1} + \left(1 - \frac{\alpha}{A}\right)\cdot z^{-2}} \tag{119}$$

Realizando uma breve comparação com a função de transferência do filtro biquadrático digital, explicitado pela equação (83), conclui-se:

$$\begin{aligned} b_0 &= 1 + A\cdot\alpha \\ b_1 &= -2\cdot\cos(\Omega) \\ b_2 &= 1 - A\cdot\alpha \\ a_0 &= 1 + \frac{\alpha}{A} \\ a_1 &= b_1 \\ a_2 &= 1 - \frac{\alpha}{A} \end{aligned} \tag{120}$$

Conforme o esperado, podemos concluir que a função de transferência em domínio z do equalizador $\mathcal{H}_P(\hat{s})$ corresponde a:

$$\mathcal{H}_P[z] = \frac{1 + A \cdot \alpha - 2 \cdot \cos(\Omega) \cdot z^{-1} + (1 - A \cdot \alpha) \cdot z^{-2}}{1 + \frac{\alpha}{A} - 2 \cdot \cos(\Omega) \cdot z^{-1} + \left(1 - \frac{\alpha}{A}\right) \cdot z^{-2}} \tag{121}$$

As funções de transferência dos equalizadores *high-shelf* e *low-shelf* podem ser obtidas utilizando os mesmos métodos discutidos nesta seção. O apêndice A apresenta as tabelas contendo todos os coeficientes dos equalizadores mencionados.

3.3.1 O Filtro Passa-Todas Digital

O filtro digital equivalente ao filtro passa-todas pode ser obtido através da técnica da transformação bilinear, descrita no tópico anterior. Com base na equação (82) da seção 3.1.1, efetuamos as substituições presentes na tabela 1.

$$\mathcal{H}_A(\hat{s})|_{s \to z} = \frac{-\frac{\text{sen}(\Omega)}{Q} \cdot (1 - z^{-2}) + 2 \cdot (1 - 2 \cdot \cos(\Omega) \cdot z^{-1} + z^{-2})}{\frac{\text{sen}(\Omega)}{Q} \cdot (1 - z^{-2}) + 2 \cdot (1 - 2 \cdot \cos(\Omega) \cdot z^{-1} + z^{-2})} \tag{122}$$

Dividindo o numerador e o denominador por 2, obtém-se:

$$\mathcal{H}_A(\hat{s})|_{s \to z} = \frac{-\frac{\text{sen}(\Omega)}{2 \cdot Q} \cdot (1 - z^{-2}) + (1 - 2 \cdot \cos(\Omega) \cdot z^{-1} + z^{-2})}{\frac{\text{sen}(\Omega)}{2 \cdot Q} \cdot (1 - z^{-2}) + 2 \cdot (1 - 2 \cdot \cos(\Omega) \cdot z^{-1} + z^{-2})} \tag{123}$$

Efetuando a distributiva e reagrupando os termos em função de z^{-2} e z^{-1}, podemos exprimir a função do filtro passa-todas digital como:

$$\mathcal{H}_A[z] = \frac{1 - \alpha - 2 \cdot \cos(\Omega) \cdot z^{-1} + (1 + \alpha) \cdot z^{-2})}{1 + \alpha - 2 \cdot \cos(\Omega) \cdot z^{-1} + (1 - \alpha) \cdot z^{-2})} \tag{124}$$

Sendo seus coeficientes definidos por:

$$
\begin{aligned}
b_0 &= 1 - \alpha \\
b_1 &= -2 \cdot \cos(\Omega) \\
b_2 &= 1 + \alpha \\
a_0 &= b_2 \\
a_1 &= b_1 \\
a_2 &= b_0
\end{aligned}
\tag{125}
$$

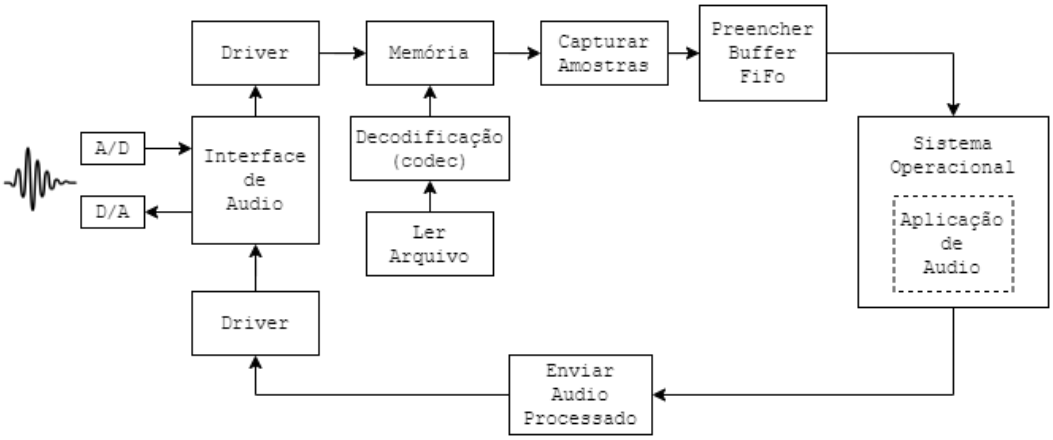
A questão da implementação de um filtro passa-todas varia da aplicação do equalizador e, portanto, deve ser ponderado sobre seu uso. O filtro passa-todas atua como um equalizador de fase, distribuindo uniformemente os atrasos para as amostras do sinal digital presentes em uma banda de frequência delimitada, esta distribuição remove artefatos de áudio produzido pelo equalizador de fase não-linear. Embora o filtro passa-todas ajude a linearizar a fase, um filtro IIR nunca terá seu comportamento em fase inteiramente linear, devido a necessidade de seus polos se encontrarem dentro do círculo unitário. Para um filtro IIR possuir fase linear seria necessário conjugar os polos de sua função de transferência, dessa forma produzindo polos fora do círculo unitário, tornando-o um sistema instável. Uma possibilidade abordada é a implementação de um filtro IIR não-causal, o que é impraticável para sistemas em tempo real, como é o caso deste trabalho.

Existem casos em que os artefatos de áudio são conteúdos produzidos intencionalmente pelo usuário do equalizador com finalidade de atingir um determinado tom ou efeito sonoro, removê-los pela equalização de fase pode ser uma funcionalidade indesejável, entretanto tendo como métrica a menor distorção possível, emprega-se o filtro passa-todas para a equalização de fase do equalizador paramétrico desenvolvido.

3.4 IMPLEMENTAÇÃO DO EQUALIZADOR

A implementação do equalizador foi realizada em conjunção com o *framework* JUCE, unindo uma biblioteca em C++ desenvolvida neste trabalho com as bibliotecas presentes no JUCE que viabilizam o acesso à placa de áudio presente no sistema descartando o uso de softwares proprietários. A biblioteca desenvolvida para este projeto conta com um total de treze funções que realizam a implementação do equalizador paramétrico. O apêndice B contém os códigos da biblioteca desenvolvida para consulta.

Figura 40 - Processo de áudio em um sistema embarcado.



Fonte: Autor “adaptado de” *juce documentation*.

A figura acima retrata o fluxo de processamento de áudio proveniente da conversão A/D ou de um arquivo codificado na memória do sistema. O acesso a processos de áudio em sistemas embarcados se consolida através programas proprietários da fabricante do hardware, denominados como *drivers*. Um *driver* de áudio é responsável por realizar o acesso ao hardware, resgatar os dados de áudio contidos e repassá-los para outros programas que se comunicam através de um processo de áudio. A princípio, não é possível acessar os *drivers* de áudio através da linguagem C++ padrão, a ponte entre a aplicação do código de processamento de áudio e o acesso aos *drivers* é realizada pelo conjunto de bibliotecas fornecidas pelo JUCE.

Através do uso dos conceitos abordados pela programação orientada a objeto, o JUCE fornece ao usuário bibliotecas capazes de interagir com os *drivers* através da criação de processos de áudio, denominados como *threads*, utilizando bibliotecas em C++ que eliminam a necessidade do uso de softwares de terceiros para o desenvolvimento de aplicações de áudio em sistemas operacionais embarcados como Windows, MacOS e Linux.

3.4.1 Biblioteca de Processamento de Sinais Digitais

O código de processamento digital de sinais desenvolvido em C++ constitui a implementação dos equalizadores caracterizados na seção 3.1. O apêndice B contém o código principal da biblioteca, cujo viabiliza a implementação de um equalizador através de instâncias declaradas e configuradas pelo usuário. Considere as linhas de código abaixo.

Listagem 1 – Exemplo de código de implementação

```
#include "dspLibFEI.hpp"

int main()
{
    float fs = 44100;   // frequência de amostragem
    float fc = 1000;    // frequência de corte ou ressonância
    float G = 5.0;      // ganho em decibéis
    float Q = 2.0;      // fator de qualidade
    int N = 2;           // Ordem do Filtro

    resonantEQ P;           // Declaração de instância
    P.configFilter(fs, fc, N); // Configuração do Filtro
    P.tuneFilter(Q, G);     // Ajuste do Filtro: Ganho e Banda

}
```

Fonte: Autor.

Temos acima um exemplo de código de implementação de um filtro digital, o arquivo "*dspLibFEI.h*" contém todas as dependências necessárias para o uso das funções contidas na biblioteca. O código descrito acima define as quatro variáveis necessárias para a construção de um filtro, enquanto o trecho de declaração de instância realiza a criação de um objeto filtro nomeado com a letra P. Embora as informações destinadas ao filtro estejam declaradas é necessário atribui-las ao filtro através das seguintes linhas de código.

Listagem 2 – Sintonização do filtro instanciado.

```
#include "dspLibFEI.hpp"

int main()
{
    float fs = 44100;   // Frequência de amostragem
    float fc = 1000;    // Frequência de corte ou ressonância
    float G = 5.0;      // Ganho em decibéis
    float Q = 2.0;      // Fator de qualidade
    int N = 2;          // Ordem do Filtro
```

```
    resonantEQ P;           // Declaração de instância
    P.configFilter(fs, fc, N); // Configuração do Filtro
    P.tuneFilter(Q, G);     // Ajuste do Filtro: Ganho e Banda
}
```

Fonte: Autor.

Ao adicionarmos as linhas de configuração e ajuste do filtro, foram fornecidos todos os dados necessários para o cálculo dos coeficientes, para isso deve ser definido o tipo de filtro a ser implementado, o que varia dependendo da aplicação desejada. A biblioteca é capaz de computar os coeficientes para os equalizadores *low-shelf*, *high-shelf*, *peak* e filtro passa-todas (*all-pass filter*), assumindo a necessidade de um equalizador *low-shelf* podemos computar seus coeficientes da seguinte forma.

Listagem 3 - Cálculo dos coeficientes através da biblioteca.

```
#include "dspLibFEI.hpp"

int main()
{
    float fs = 44100;   // Frequência de amostragem
    float fc = 1000;    // Frequência de corte ou ressonância
    float G = 5.0;      // Ganho em decibéis
    float Q = 2.0;      // Fator de qualidade
    int N = 2;          // Ordem do Filtro

    resonantEQ P;           // Declaração de instância
    P.configFilter(fs, fc, N); // Configuração do Filtro
    P.tuneFilter(Q, G);     // Ajuste do Filtro: Ganho e Banda

    P.computeLowShelfCoeffs();  // [1]
    P.getFilterInfo();          // [2]
    P.printTransferFunction();  // [3]
}
```

Fonte: Autor.

A função descrita por [1] calcula os coeficientes para um equalizador *low-shelf* com base no que foi configurado nas funções anteriores, o filtro agora está pronto para executar sua

equação de diferenças. As funções [2] e [3] são destinadas apenas para a depuração do código de modo a verificar que a instancia foi construída corretamente, compila-se o código utilizando um compilador C++, para o projeto em questão foi utilizado o compilador GNU GCC. Após a compilação o terminal do sistema operacional deve retornar a saída do código processado contendo as informações do tipo de filtro projetado, os coeficientes do numerador e denominador calculados e os ajustes informados anteriormente no código, conforme ilustrado pela figura a seguir.

Figura 41 - Terminal Linux.

```
Filter Type is: Low Shelf.
Filter Order: 2
Numerator: 1.016749      -1.896318       0.905499
Denominator: 1.000000    -1.901993       0.916574
Fixed Integer Numerator: 0       0       0
Fixed Integer Denominator: 0     0       0
Gain: 5.000000,          Q factor: 2.000000,     sampling Frequency: 44100.000000,       frequency of interest: 1000.00


       1.016749  +  -1.896318 * z-1  +  0.905499 * z-2
H(z) = ------------------------------------------------
       1.000000  +  -1.901993 * z-1  +  0.916574 * z-2
```

Fonte: Autor.

O filtro digital configurado acima caracteriza-se como um equalizador *low-shelf* com ganho de cinco decibéis e sintonizado em 1 kHz. Antes de executar a equação de diferenças do filtro é necessário informar o ponteiro referente à posição de memória que contém a primeira amostra do sinal digitalizado, dessa forma declara-se um *buffer* de memória. Um *buffer* de memória constitui um conjunto de endereços sequenciais referentes às posições da memória física do sistema, as quais armazenam cada amostra do sinal de entrada, desta forma escreve-se o seguinte código.

Listagem 4 - Declaração do *buffer* de memória e processamento do sinal de entrada.

```
#include "dspLibFEI.hpp"

#define NUM_OF_SAMPLES 1024
double sourceBuffer[NUM_OF_SAMPLES];

int main()
{
    float fs = 44100;   // Frequência de amostragem
    float fc = 1000;    // Frequência de corte ou ressonância
```

```
    float G = 5.0;      // Ganho em decibéis
    float Q = 2.0;      // Fator de qualidade
    int N = 2;          // Ordem do Filtro

    resonantEQ P;           // Declaração de instância
    P.configFilter(fs, fc, N); // Configuração do Filtro
    P.tuneFilter(Q, G);     // Ajuste do Filtro: Ganho e Banda

    P.computeLowShelfCoeffs();  // [1]
    P.getFilterInfo();          // [2]
    P.printTransferFunction();  // [3]

    /* [4] */
    // processInput(buffer fonte, buffer de destino, número de amostras);
    P.processInput(&sourceBuffer[0], &sourceBuffer[0], NUM_OF_SAMPLES);
}
```

Fonte: Autor.

A função [4] aplica a equação de diferenças na forma direta II transposta ao sinal de entrada armazenado nas posições de memória do *buffer* informado. Evitando o consumo excessivo de memória informa-se o *buffer* de destino sendo o próprio *buffer* de entrada, entretanto pode ser informado um *buffer* diferente destinado à escrita do sinal processado. O conteúdo do *buffer* de memória pode ser escrito em arquivos "*.dat*", o qual armazena os dados numéricos das amostras do sinal processado, bastando adicionar a seguinte função ao código.

Listagem 5 – Geração de um arquivo .dat para o sinal processado.

```
#include "dspLibFEI.hpp"

#define NUM_OF_SAMPLES 1024
double sourceBuffer[NUM_OF_SAMPLES];

int main()
{
    FILE* writePtr;
    char* filename = "processedSignal.dat";

    float fs = 44100;   // Frequência de amostragem
```

```
    float fc = 1000;    // Frequência de corte ou ressonância
    float G = 5.0;      // Ganho em decibéis
    float Q = 2.0;      // Fator de qualidade
    int N = 2;          // Ordem do Filtro

    resonantEQ P;           // Declaração de instância
    P.configFilter(fs, fc, N); // Configuração do Filtro
    P.tuneFilter(Q, G);     // Ajuste do Filtro: Ganho e Banda

    P.computeLowShelfCoeffs();  // [1]
    P.getFilterInfo();          // [2]
    P.printTransferFunction();  // [3]

    /* [4] */
    // processInput(buffer fonte, buffer de destino, número de amostras);
    P.processInput(&sourceBuffer[0], &sourceBuffer[0], NUM_OF_SAMPLES);
    /* [5] */
    P.genDatFile(writePtr, filename, &sourceBuffer[0], NUM_OF_SAMPLES);
}
```

Fonte: Autor.

Ao informar o ponteiro de arquivo, o nome designado ao arquivo, o *buffer* contendo o sinal armazenado e o número de amostras presentes a função é capaz de escrever cada amostra presente nas posições de memória em um arquivo que posteriormente pode ser utilizado para análise. Uma forma de averiguar a performance de um filtro projetado pelas funções posteriores é através de sua resposta impulsiva, considere o código a seguir.

Listagem 6 - Obtenção da resposta ao impulso do filtro digital.

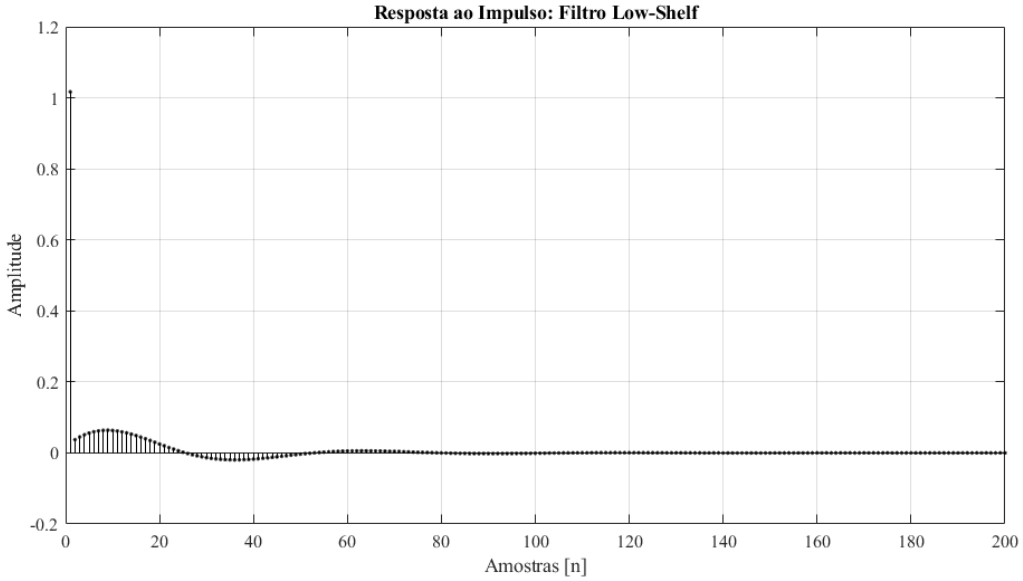
```
#include "dspLibFEI.hpp"

#define NUM_OF_SAMPLES 1024
double sourceBuffer[NUM_OF_SAMPLES];
double sourceBuffer_1[NUM_OF_SAMPLES];

int main()
{
```

```
    FILE* writePtr;

    char* filename = "processedSignal.dat";
    char* filename_1 = "impulseResponse.dat";

    float fs = 44100;   // Frequência de amostragem
    float fc = 1000;    // Frequência de corte ou ressonância
    float G = 5.0;      // Ganho em decibéis
    float Q = 2.0;      // Fator de qualidade
    int N = 2;          // Ordem do Filtro


    resonantEQ P;           // Declaração de instância
    P.configFilter(fs, fc, N); // Configuração do Filtro
    P.tuneFilter(Q, G);     // Ajuste do Filtro: Ganho e Banda

    P.computeLowShelfCoeffs();  // [1]
    P.getFilterInfo();          // [2]
    P.printTransferFunction();  // [3]

    /* [4] */
    // processInput(buffer fonte, buffer de destino, número de amostras);
    P.processInput(&sourceBuffer[0], &sourceBuffer[0], NUM_OF_SAMPLES);
    /* [5] */
    P.genDatFile(writePtr, filename, &sourceBuffer[0], NUM_OF_SAMPLES);
    /* [6] */
    P.getImpulseResponse(&sourceBuffer_1[0], NUM_OF_SAMPLES);
    P.genDatFile(writePtr, filename_1, &sourceBuffer_1[0],NUM_OF_SAMPLES);
}
```

Fonte: Autor.

Podemos exportar o arquivo produzido pelo código [6] ao software MATLAB para visualizar com mais clareza a resposta ao impulso do filtro digital projetado.

Figura 42 - Gráfico da resposta ao impulso do filtro projetado.



Fonte: Autor.

A função que reproduz a resposta ao impulso do filtro digital constitui a convolução de dois sinais descrita pela equação (24) na seção 2.2.1, sendo $f[n]$ a função caracterizada pelo delta de Kronecker e $g[n]$ a equação de diferenças do filtro dada pela equação (90) na seção 3.2.1. A implementação do equalizador paramétrico foi feita através das funções discutidas até então, enquanto o acesso aos *buffers* de memória contendo as amostras dos sinais de áudio será realizado pelo *framework* JUCE que fornece as bibliotecas necessárias para o desenvolvimento do software.

3.4.2 Processamento de Áudio

O *framework* JUCE providencia bibliotecas para o desenvolvimento de interfaces gráficas de usuário, estruturas de dados, aplicações de processamento gráfico e processamento de áudio entre muitas outras funcionalidades disponíveis em seu repositório. A biblioteca de processamento de áudio possui flexibilidade a ponto de permitir o usuário desenvolver uma aplicação de tecnologia de estúdio virtual (*Virtual Studio Technology plug-in*) para softwares de áudio já existentes ou criar um software de áudio isolado e definido pelo próprio usuário. Para o projeto do equalizador paramétrico definiu-se a criação de um software de processamento de áudio isolado que implemente o equalizador com controle em tempo real.

O arquivo "*JuceHeader.h*" contém todas as dependências necessárias para acesso às bibliotecas contidas no repositório fornecido. A construção do projeto se inicia definindo a classe componente principal que irá conter as principais funções e variáveis necessárias para a execução do programa. Cria-se um arquivo nomeado como "*MainComponent.hpp*", responsável pelo código descritivo da classe componente principal.

Listagem 7 - Código do arquivo *MainComponent.hpp*

```
class MainComponent :
    public juce::AudioAppComponent,  // [1]
    private juce::Timer         // [2]
{
public:
    MainComponent();
    ~MainComponent() override;

    // Código Completo Disponível no Apêndice C.

private:

    // Código Completo Disponível no Apêndice C.
```

Fonte: Autor.

As linhas [1] e [2] iniciam a chamada a outras componentes de códigos presentes na biblioteca, neste caso, atribui-se a classe componente principal as funções pertencentes a componentes de áudio e a temporizadores. As funções contidas na classe componente de áudio realizam a comunicação com os *drivers* do hardware através da criação de um *thread* de áudio. As funções necessárias para o acesso, processamento e reprodução de áudio são dadas pelo código a seguir.

Listagem 8 - Funções da componente de áudio JUCE.

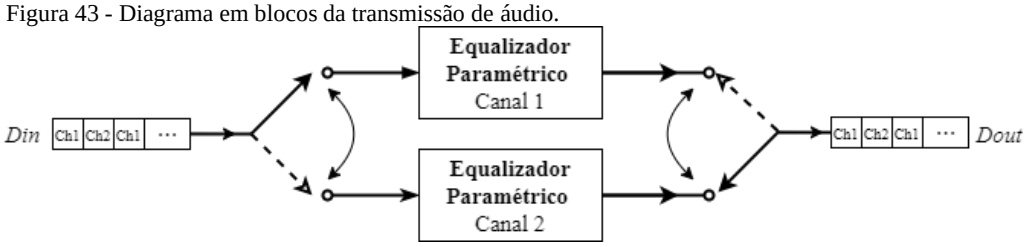
```
/* [1] Prepara dispositivo de áudio para acesso das amostras */
void prepareToPlay(int samplesPerBlockExpected, double sampleRate);

/* [2] Acessa Buffer FiFo e retorna ponteiro para escrita e leitura,
efetua o código de processamento de áudio*/
void getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill);

/* [3] Escreve amostras ao buffer da placa de áudio e libera dispositivo para reprodução*/
void releaseResources();
```

Fonte: Autor.

As funções [1], [2] e [3] acima realizam respectivamente o acesso ao *buffer* FIFO pertencente a arquitetura PCM do dispositivo de áudio, aplicação do código de processamento de sinais realizado pela biblioteca desenvolvida neste trabalho e a reprodução do áudio processado. Essas funções constituem o *thread* de áudio que se comunica com o dispositivo de áudio presente no sistema embarcado. Resgatando os conceitos explorados pela seção 2.9 observa-se que o canal PCM presente na placa *raspberry pi 4* realiza o envio dos dados processados através dos barramentos nomeados como *Din* e *Dout.* É importante ressaltar que o equalizador é projetado para trabalhar com sistemas *stereo*, ou seja, a informação de áudio é constituída por dois canais a serem processados: canal 1 e canal 2.

Através dos canais alocados pelo *thread* de áudio criado pelo JUCE a implementação do equalizador se consolida através do seguinte diagrama.

Figura 43 - Diagrama em blocos da transmissão de áudio.



Fonte: Autor.

Os canais 1 e 2 representam respectivamente os canais esquerda e direita em sistemas de reprodução de áudio *stereo*, Dentro da estrutura PCM da *raspberry* pi os canais 1 e 2 compartilham os mesmos *buffers* de transmissão e recepção, portanto, o envio das amostras é realizado de forma intercalada, em torrentes separadas e delimitadas por pulsos de

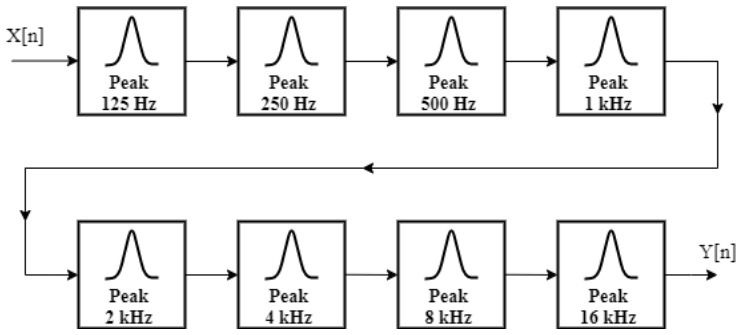
sincronismo dentro da arquitetura PCM. Cada envio possui um cabeçalho em bits que permite com que o sistema realize a distinção de cada canal recebido.

As amostras de áudio podem ser obtidas através do dispositivo de áudio instalado na placa de desenvolvimento ou através de arquivos armazenados na memória do sistema através do CODEC correspondente.

O processamento do áudio se dá pela intercalação dos canais alocados, ou seja, em um momento se processam as amostras recebidas pelo canal 1, em seguida se processam as amostras recebidas pelo canal 2 e assim por diante alternando entre os canais de acordo com cada canal recebido. O processo de envio de amostras por alternância de canais dentro de um único meio de transmissão é denominado como *interleaving*, cada canal recebido contém um número específico de amostras o qual é informado para o *thread* de áudio junto com os ponteiros de leitura e escrita do registrador FIFO de entrada ilustrado na figura 25. Após o processamento das amostras pelo equalizador, estas são enviadas ao registrador FIFO de saída, que então excursionará cada canal, através do *interleaving*, ao conversor D/A que então irá reproduzir o áudio processado ao sistema conectado.

Os equalizadores paramétricos implementados para cada canal constituem de uma cascata de filtros de oitava ordem projetados pelo arquivo *resonantEQ.cpp* contido na biblioteca *dspLibFEI.hpp* desenvolvida neste trabalho. O diagrama de blocos ilustra a estrutura de filtros digitais projetada.

Figura 44 - Estrutura de bandas do equalizador paramétrico

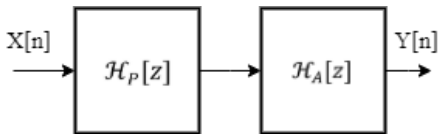


Fonte: Autor.

O equalizador conta com controle em 8 bandas distribuídas em oitavas de frequência, cada filtro projetado pode ter seu ganho e fator de qualidade controlados em tempo real através da interface gráfica de usuário, o controle ajustado pelo usuário em cada banda afeta os dois canais de áudio simultaneamente. Cada banda descrita pelo diagrama acima pode ser

decomposta em dois filtros digitais em série: o equalizador *peak* de oitava-ordem e um filtro passa-todas de oitava-ordem para linearização de fase, ambos implementados através de uma cascata seções de segunda-ordem.

Figura 45 - Filtros digitais presentes em cada banda do equalizador



Fonte: Autor.

3.4.3 Construção da Interface Gráfica de Usuário

A interface gráfica de usuário é construída através das seguintes funções.

Listagem 9 - Funções construtoras da GUI.

```
void paint (juce::Graphics& g) override;     // [1]
void drawFrame(juce::Graphics& g);           // [2]
void drawNextFrameOfSpectrum();              // [3]
void resized() override;                     // [4]
void timerCallback() override;               // [5]
void pushNextSampleIntoFifo(float Sample);   // [6]
```

Fonte: Autor.

A função [1] é responsável por atribuir dentro da janela do software cada componente de GUI declarado no arquivo *MainComponent.hpp*, tais como barras deslizantes para controle de ganho e fator de qualidade e rótulos para identificação de cada componente presente. As funções [2] e [3] são responsáveis por ilustrar o espectro de Fourier do sinal e por fim a função [4] realiza o ajuste necessário caso o tamanho da janela seja alterado.

A GUI conta com 16 barras deslizantes, 8 destinadas ao controle de ganho e 8 destinadas ao controle do fator de qualidade. As barras de ganho encontram-se limitadas entre 10 e -10 decibéis, magnitude ideal para processar o áudio, por sua vez o fator de qualidade pode ser alterado de 0.1 até 100, produzindo tanto bandas largas quanto estreitas. O espectro de Fourier é atualizado periodicamente e é gerenciado pelas funções [5] e [6], essas funções realizam a temporização e a transformada de Fourier respectivamente.

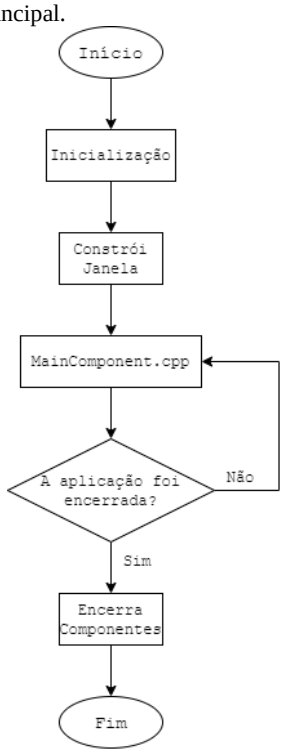
A função [6] acessa o registrador FIFO de saída contendo as amostras processadas, aplicando a transformada de Fourier e enviando-as para a função [2] que aplica a janela espectral de *Hanning* para em seguida desenhar o espectro. A função [5] se encarrega de atualizar o espectro com um temporizador a uma taxa de 30 Hz, ou seja, trinta vezes por segundo.

3.4.4 O Código Principal

Os arquivos *MainComponent.hpp* e *MainComponent.cpp* constituem toda a funcionalidade do sistema equalizador paramétrico implementado, e se encontram declarados no cabeçalho do código *main.cpp* que constitui o código principal contendo as funções das bibliotecas fornecidas pelo JUCE necessárias para produzir o software, ao ser compilado os binários resultantes da operação produzem um arquivo executável contendo o equalizador.

O fluxo das funções dentro do código principal se dá através do seguinte fluxograma.

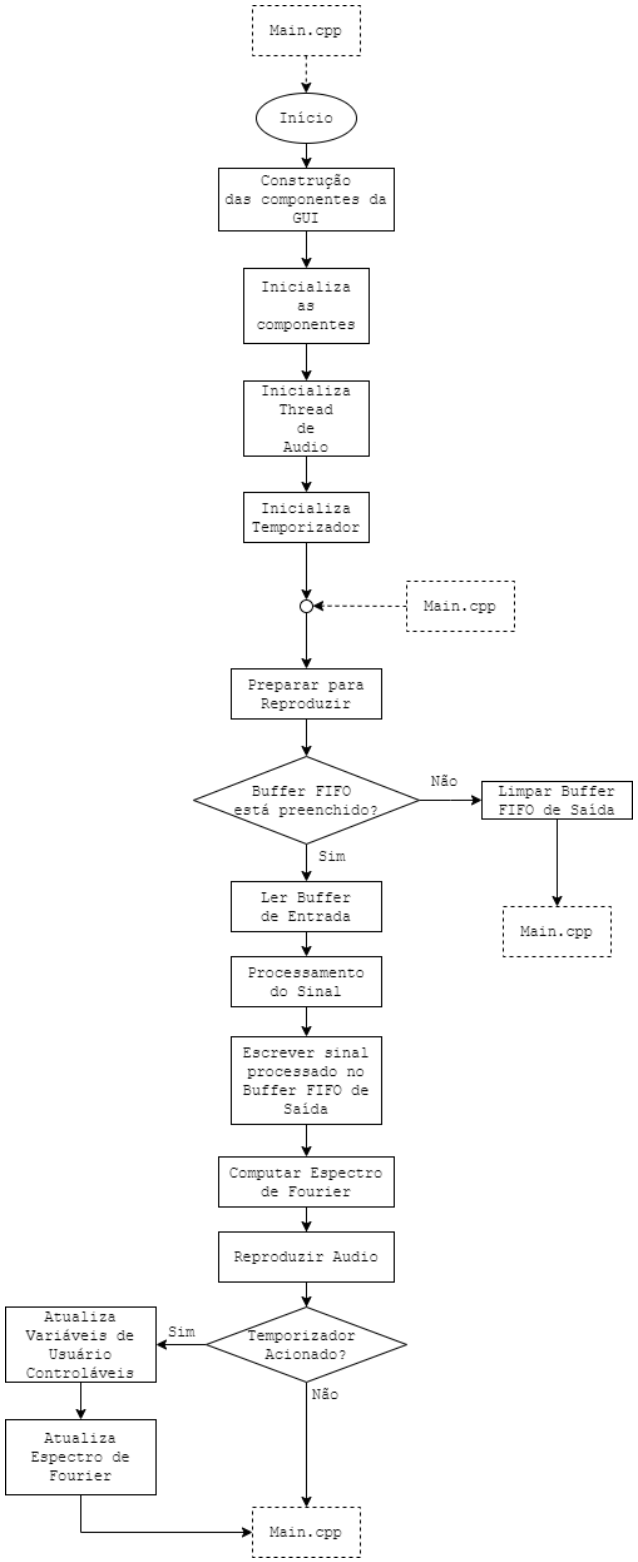
Figura 46 - Fluxograma da função principal.



Fonte: Autor.

A componente principal executa o processamento de áudio e gerenciamento da GUI projetada, seu fluxograma é caracterizado na página seguinte.

Figura 47 - Fluxograma do bloco *MainComponent.cpp*



Fonte: Autor.

O código principal contém os códigos contidos nos arquivos *MainComponent.hpp* e *MainComponent.cpp* e realiza o gerenciamento da janela do software durante a execução. As variáveis controladas são sempre atualizadas quando o bloco “Preparar para Reprodzir” é executado dentro do fluxo. As funções do equalizador abordadas na seção 3.4.2 são executadas para configurar os filtros a frequência de amostragem e frequência de ressonância de cada equalizador nos canais 1 e 2. Embora a frequência de ressonância é um parâmetro e, portanto, inalterável no software, a taxa de amostragem é um parâmetro variável. Caso o usuário altere o dispositivo de áudio conectado para outro com taxa de amostragem superior ou inferior, esta mudança refletirá nos coeficientes a serem computados, portanto, atualiza-se a taxa de amostragem junto com a frequência de ressonância.

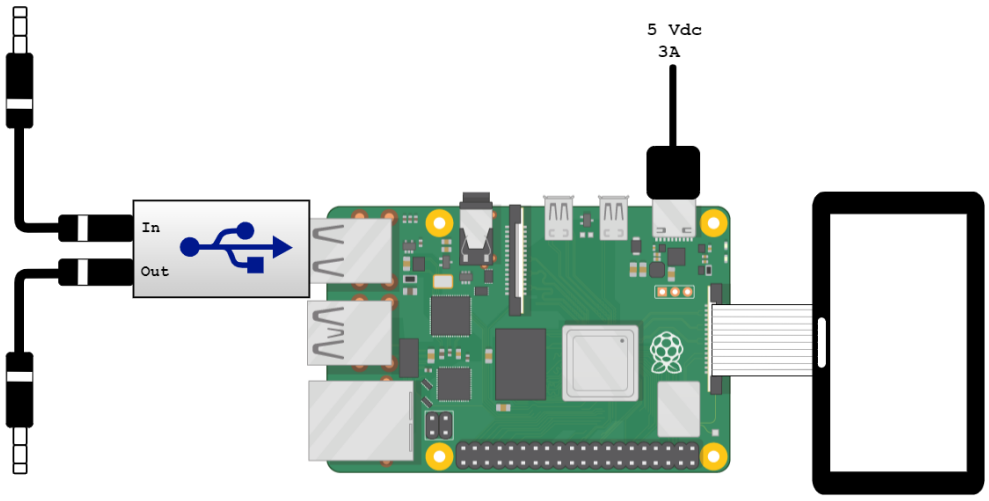
As variáveis controladas são sempre atualizadas após o *buffer* FIFO de entrada, cada mudança realizada nas barras deslizantes será efetivamente aplicada quando o bloco de áudio estiver pronto para ser lido e processado. A atualização das variáveis de usuário e do espectro de Fourier computado são realizadas a uma taxa definida pelo temporizador.

O apêndice C contém o código do software equalizador paramétrico produzido neste trabalho. Os códigos foram compilados através do compilador GNU GCC utilizando um arquivo *MakeFile* gerado para a distribuição *Linux*. Foi produzida também uma distribuição para Windows 10 utilizando o compilador *MingW64* para averiguar a performance do programa em sistemas operacionais distintos e sua performance em hardwares mais sofisticados.

3.4.5 Estrutura da Placa de Desenvolvimento

A placa *raspberry Pi 4* possui uma estrutura PCM para processamento de sinais de áudio, entretanto a placa não possui um dispositivo de áudio para realizar a conversão A/D, para isso emprega-se um adaptador de áudio USB que realiza as conversões A/D e D/A e para controle com a interface gráfica do equalizador utiliza-se uma tela com sensibilidade ao toque.

Figura 48 - Diagrama do hardware utilizado.



Fonte: Autor.

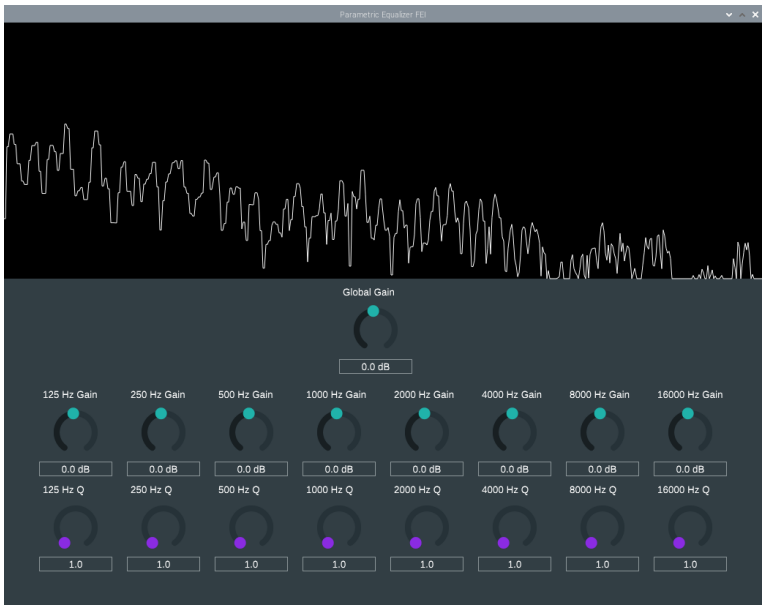
A placa opera com frequências de amostragem de até 96 kHz, acima dessa taxa deverá ser feito o procedimento de re-amostragem do sinal, contudo o adaptador de áudio USB instalado na placa opera com uma frequência 48 kHz, logo não há necessidade. A tela sensível a toque é conectada ao plugue DSI presente na placa, enquanto dois cabos de áudio são conectados ao adaptador USB, um cabo destinado à entrada de áudio e outro à saída do áudio processado. A alimentação da placa é feita através de uma fonte chaveada de 5 Volts e 3 Ampères.

O controle do equalizador será feito através da tela onde o usuário pode interagir com as barras deslizantes, ajustando o ganho e o fator de qualidade de acordo com sua preferência.

4 RESULTADOS E DISCUSSÃO

A implementação do equalizador paramétrico na placa de desenvolvimento *raspberry pi* 4, modelo B obteve sucesso, o projeto foi compilado utilizando um arquivo Linux *makefile* produzindo o software ilustrado pela figura abaixo.

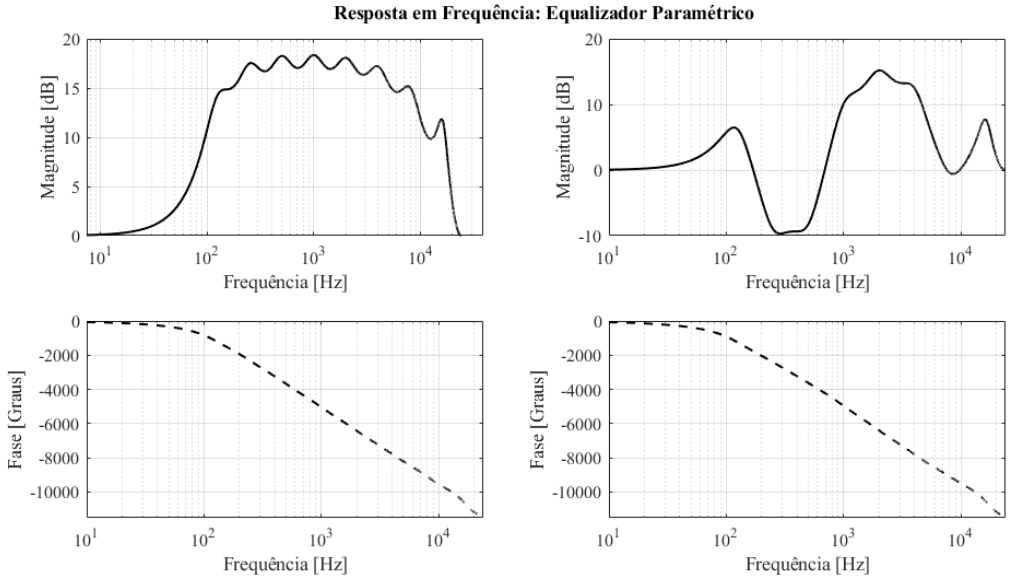
Figura 49 – Software do sistema equalizador paramétrico.



Fonte: Autor.

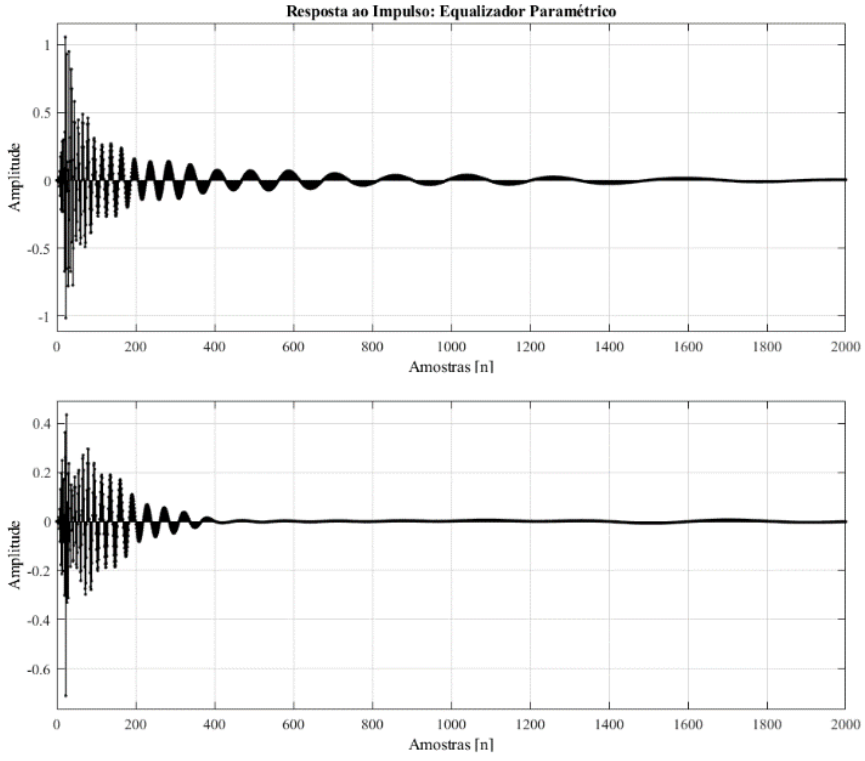
O equalizador entrega ao usuário uma resposta em frequência com fase linearizada, minimizando as distorções provocadas pelos filtros IIR. A resposta em frequência do equalizador foi obtida com auxílio do software MATLAB através da ferramenta *filter visualizer* que permite a geração dos gráficos de magnitude e fase a partir dos coeficientes computados. O equalizador conta com 8 bandas de frequência controláveis, podendo ampliar e atenuar frequências entre 125 Hz e 16 kHz. Cada banda conta com um ajuste para linearização de fase sincronizado com o fator de qualidade de cada banda, ou seja, conforme o usuário ajusta a largura de banda as mudanças realizadas refletem no filtro passa-todas, que atua linearizando a fase pertencente aquele filtro. O equalizador apresenta comportamento estável para todas as frequências dentro da sua faixa de operação (de 1 Hz até a frequência a Nyquist), podemos avaliar seu comportamento a partir de sua resposta ao impulso e resposta em frequência.

Figura 50 - Resposta em frequência do sistema equalizador paramétrico projetado.



Fonte: Autor.

Figura 51 - Respostas do impulso do sistema.



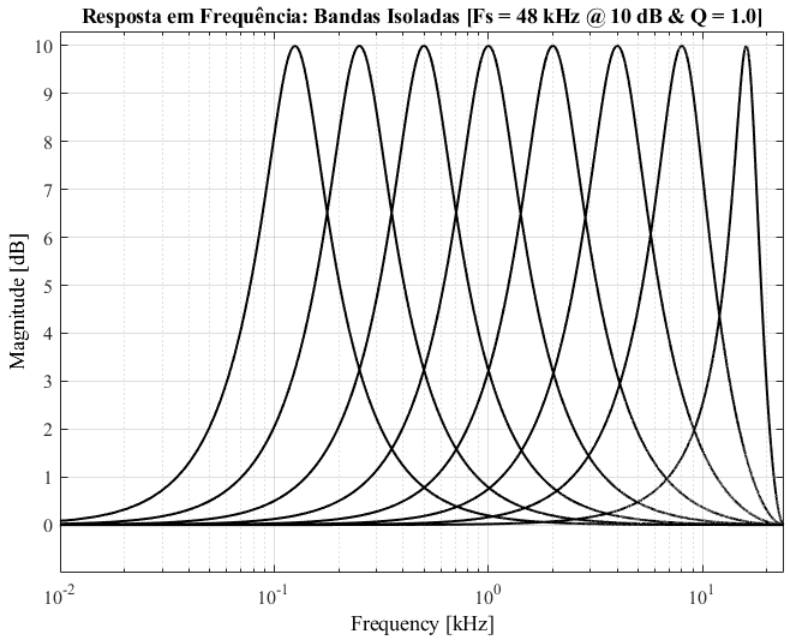
Fonte: Autor.

O apêndice D apresenta mais curvas de resposta em frequência para diferentes comportamentos junto com os diagramas de polos e zeros.

A presença dos filtros passa-todas induz um comportamento oscilatório para as respostas ao impulso, isso é provocado pelo baixo fator de atenuação, característico do processo de equalização de fase. O sistema conta com um ganho global ajustável, responsável pelo volume de saída do sistema, limitado entre -10 e 10 decibéis.

Avaliando os gráficos de resposta em frequência, é perceptível o vazamento espectral entre bandas. Apesar de todos os ganhos implementados estarem limitados ao máximo 10 decibéis, a curva com todos os ganhos em sua capacidade máxima atinge até 18 decibéis de ganho, isso se deve pela influência de cada banda configurada pelo seu fator de qualidade. Quanto menor o fator de qualidade maior será o vazamento espectral induzido, podemos verificar esse fenômeno a partir da equação (72) na seção 3.1. A figura abaixo explicita a influência que cada banda do equalizador possui em bandas vizinhas.

Figura 52 - Resposta em frequência de cada uma das 8 bandas presentes.

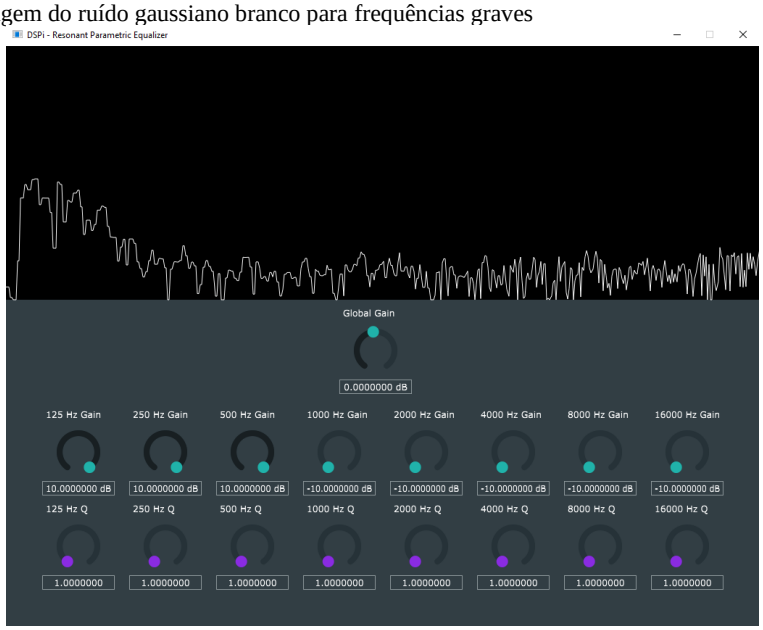


Fonte: Autor.

A partir da figura acima fica evidente o vazamento espectral proveniente da intersecção entre bandas, o efeito é amenizado ao ajustar o fator de qualidade a uma oitava de frequência o que equivale a um valor Q igual a $\sqrt{2}$. Podemos observar que intersecção da banda em 16 kHz com a banda vizinha está em um nível ligeiramente inferior quando comparado às outras, isso se deve pela característica do equalizador utilizado que possui ganho unitário para a frequência de Nyquist, em 24 kHz. Os testes de bancada indicam o caráter de funcionamento do

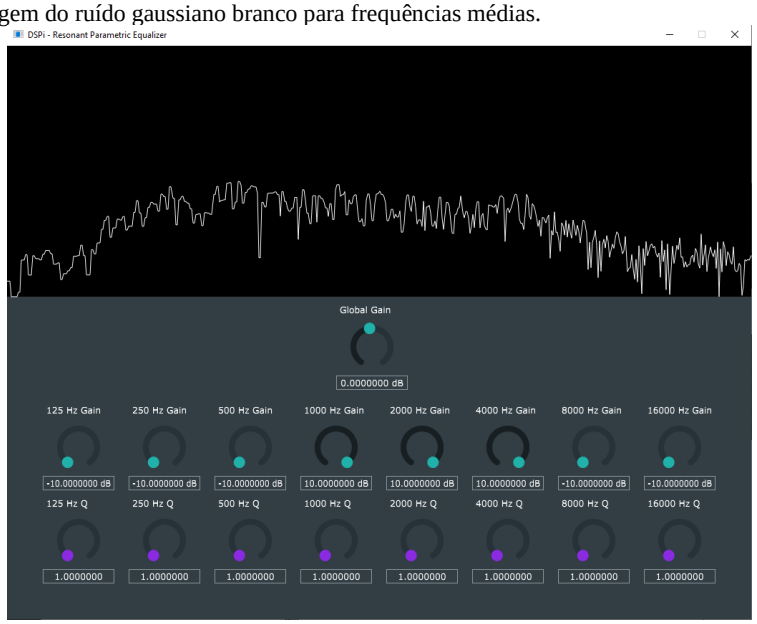
equalizador com relação aos sinais injetados. Durante os testes foram utilizados uma harmônica de 1 kHz e um ruído gaussiano branco para averiguar o comportamento dos filtros digitais projetados.

Figura 53 - Filtragem do ruído gaussiano branco para frequências graves



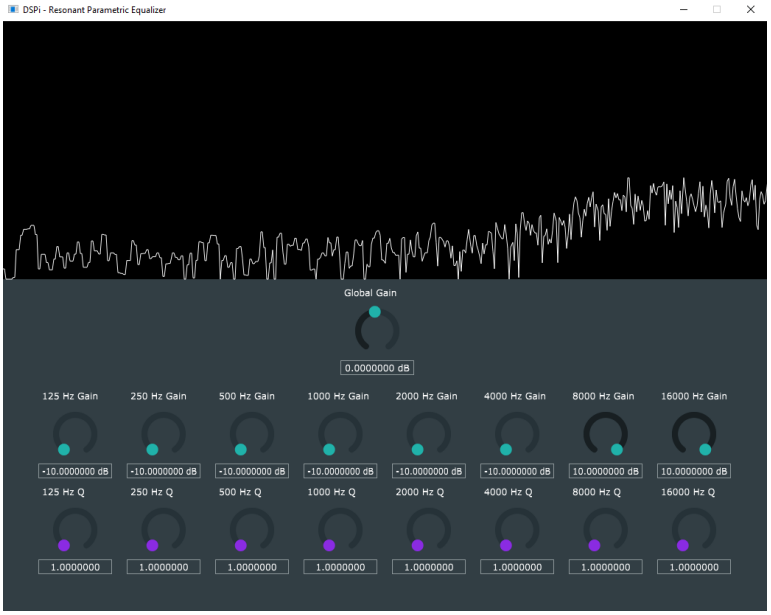
Fonte: autor.

Figura 54 - Filtragem do ruído gaussiano branco para frequências médias.



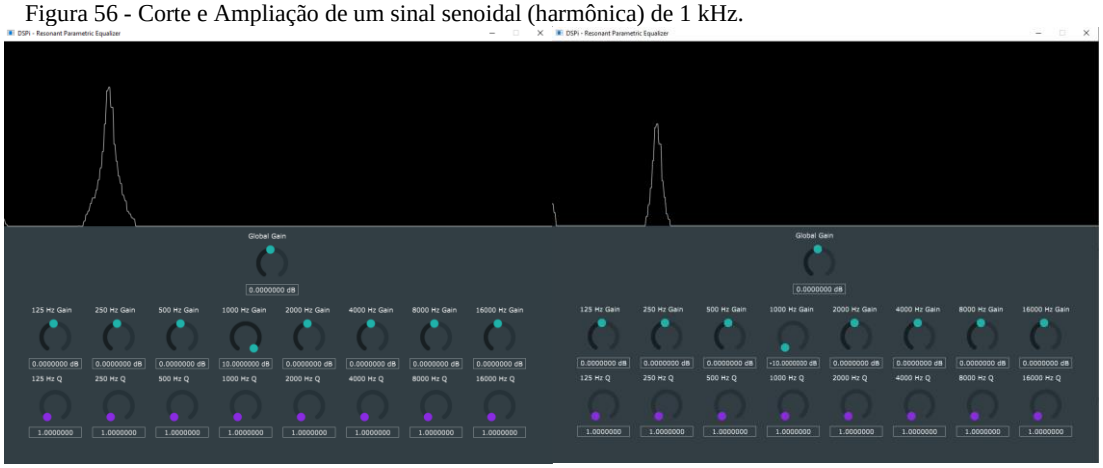
Fonte: autor.

Figura 55 - Filtragem do ruído gaussiano branco para frequências agudas.



Fonte: autor.

Figura 56 - Corte e Ampliação de um sinal senoidal (harmônica) de 1 kHz.



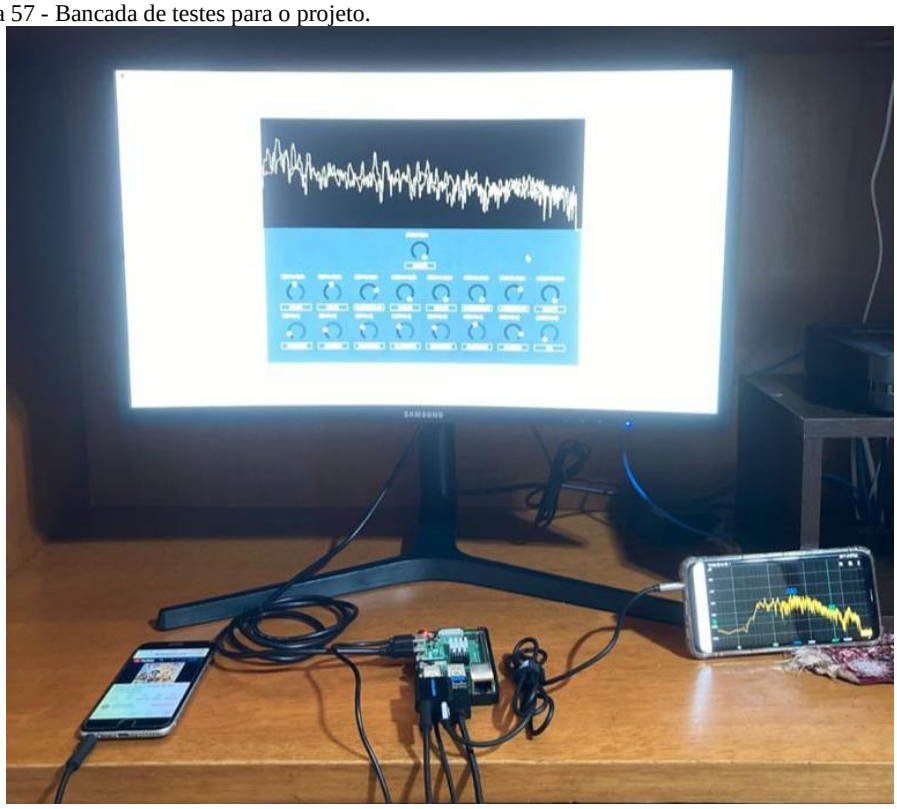
Fonte: autor.

Para o visualizador de espectro contido no software foi utilizado uma janela de *Hanning* para análise e síntese de Fourier do sinal processado. As bibliotecas contidas no *framework* JUCE contém as rotinas e sub-rotinas necessárias para realizar o processamento espectral do sinal.

4.1 CONSIDERAÇÕES FINAIS

Apesar do sucesso da implementação do equalizador paramétrico para a placa de desenvolvimento, é necessário salientar a performance da placa quanto ao software produzido.

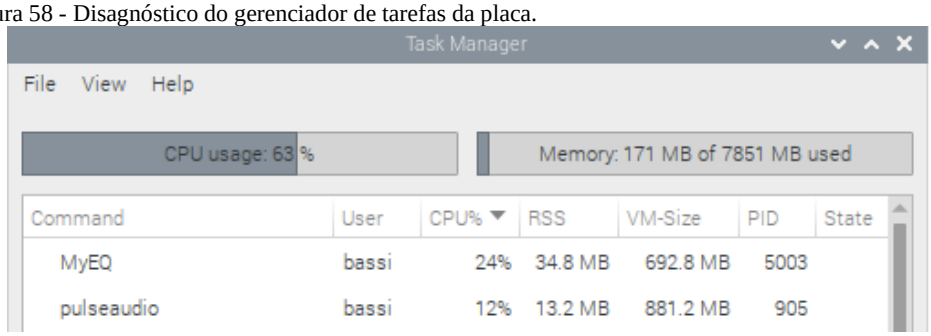
Figura 57 - Bancada de testes para o projeto.



Fonte: Autor.

A performance da distribuição Linux para *raspberry pi* do equalizador paramétrico produzido não foi a ideal para o projeto considerado, a saída de áudio, embora processada de acordo o planejado, apresentava artefatos de áudio indesejáveis. Os artefatos produzidos eram audíveis e deterioravam a qualidade de áudio esperada para o projeto, a principal causa destes artefatos ocorre devido a comunicação entre o *thread* de áudio e os drivers ALSA (*Advanced Linux Sound Architecture*). A má comunicação entre os drivers ALSA e o *thread* de áudio JUCE ocasionam em um alto uso do processador ARM, o que acarreta mal processamento das amostras pela placa de desenvolvimento.

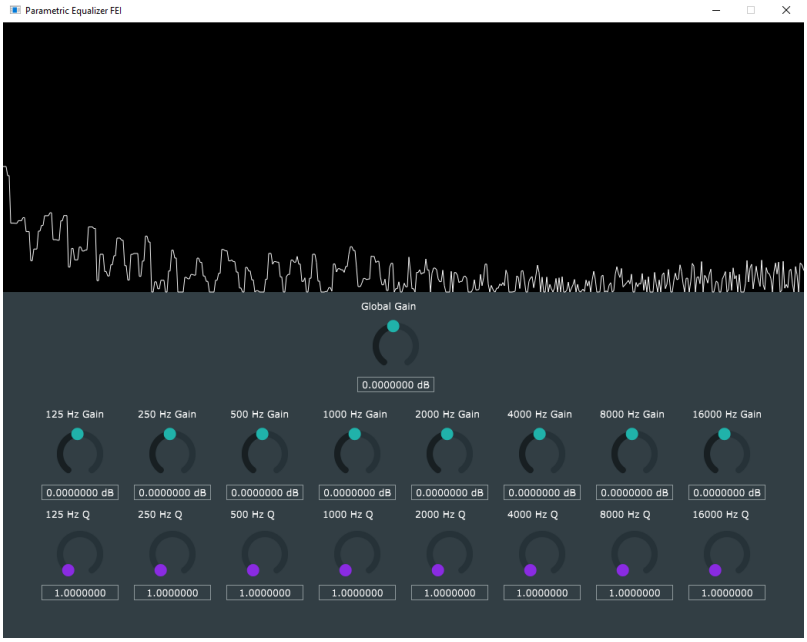
Figura 58 - Disagnóstico do gerenciador de tarefas da placa.



Fonte: autor.

Em contrapartida, a distribuição Windows obteve um ótimo resultado, sendo capaz de reproduzir áudio sem distorções ou artefatos. O projeto foi compilado utilizando o compilador *Mingw64*, cujo é um compilador GNU GCC ambientado ao sistema operacional da Microsoft.

Figura 59 - Software do sistema equalizador paramétrico, distribuição Windows.



Fonte: Autor.

5 CONCLUSÃO

A implementação de filtros digitais é regida por uma série de formalismos matemáticos essenciais para atingir os resultados adequados. O progresso dos sistemas digitais ao longo tempo possibilitou o processamento de sinais analógicos, que antes eram processados por uma extensa cadeia de circuitos físicos, serem processados por sistemas espacialmente limitados e com maior controle e precisão fornecidos pelo ambiente digital. A replicação de filtros analógicos seguindo o método da transformada bilinear se mostrou engenhoso evitando a necessidade de extensas análises para o projeto, bastando apenas definir o filtro analógico a ser implementado e computar sua função de transferência digital análoga realizando-se as aproximações necessárias. Independentemente do método a ser utilizado, vale ressaltar que os métodos numéricos aqui estudados são de grande importância e se fazem presentes em qualquer outra prática de projeto de filtros digitais

A linguagem C++ desempenhou um papel importante para o desenvolvimento do código pertencente a biblioteca de processamento digital de sinais utilizada e a construção do código da componente de áudio providenciado pelo *framework*, pois é uma linguagem extensamente utilizada em programação de sistemas embarcados e de sintaxe bem estruturada.

O processamento de sinais digitais se faz presente na atualidade e engloba todos os meios de comunicação e consumo de mídias, constituindo dessa forma, um dos principais alicerces processamento de informações constituídas por sinais analógicos e digitais.

REFERÊNCIAS

1. AZAMBUJA, Haroldo Fróes de. **A Função Delta: propriedades e aplicações.** 1964. 163 f. Tese (Doutorado) - Curso de Engenharia, Engenharia, Universidade do Rio Grande do Sul, Porto Alegre, 1964.

2. CHAKRAVORTY, Pragnan.; **IEEE Signal Processing Magazine**, v.35, n.5, p. 175-177, Sept. 2018.

3. CORDARO, Paulo Domingos; KAWANO, Alexandre. **O Delta de Dirac: uma introdução à teoria das distribuições para engenharia.** São Paulo: Livraria da Física, 2002.

4. DISTEFANO, Joseph J.; STUBBERUD, Allen R.; WILLIANS, Ivan J.; **Sistemas de Retroação e Controle: com aplicações para engenharia, física e biologia.** São Paulo: McGraw-Hill, 1972.

5. HAYKIN, Simon; MOHER, Michael. **Introdução aos Sistemas de Comunicação.** 2ª ed. Porto Alegre: Bookman, 2008.

6. HSU, Hwei P. **Sinais e Sistemas.** Porto Alegre: Bookman, 2004.

7. IRWIN, J. David. **Análise de Circuitos em Engenharia.** 4ª ed. São Paulo: Pearson, 2004.

8. JUCE framework repository, 2021. Disponível em: https://github.com/juce-framework/JUCE, Acesso em: 03 de Março, 2021.

9. KREYSZIG, Erwin. **Matemática Superior para Engenharia.** 9ª ed. São Paulo: Ltc, 2008. Vol 9.

10. KUC, Roman. **Introduction to Digital Signal Processing.** Hyderabad: Bs Publications, 1982.

11. LATHI, B. P. **Sinais e Sistemas Lineares.** 2ª ed. Porto Alegre: Bookman, 2008.

12. LATHI, B. P.; Zhi Ding. **Sistemas de Comunicações Analógicos e Digitais.** 4ª. ed. Porto Alegre: LTC, 2012.

13. LIMA, Thiago de. **Funções: uma introdução.** Santo André: UFABC, 2017.

14. MAES, Jan; VERCAMMEN, Marc, eds.; **Digital Audio Technology: A guide to CD, MiniDisc, SACD, DVD(A), MP3 and DAT**. 4ª Edição. Focal Press. ISBN 0240516540.

15. NALON, José A.; **Introdução ao Processamento Digital de Sinais**. Rio de Janeiro: LTC, 2009.

16. SMITH J. O.; **Introduction to Digital Filters with Audio Applications**, 2007, *Web*, (Stanford University Center for Computer Research in Music and Acoustics – CCRMA). Disponível em: http://ccrma.stanford.edu/~jos/filters/. Acesso em: 02 fev. 2021

17. OPPENHEIM, Alan V.; SCHAFER, Ronald W. **Processamento em tempo discreto de sinais.** [S.l.]: Pearson Education do Brasil, 2013.

18. OPPENHEIM, Alan V.; WILLSKY, Alan S.; **Sinais e Sistemas.** [S.l.]: Pearson Education do Brasil, 2010.

19. QUEVEDO, Carlos Peres. **Cálculo Avançado.** Rio de Janeiro: Editora Interciência, 2000.

20. QUEVEDO, Carlos Peres. **Circuitos Elétricos.** Rio de Janeiro: Editora Guanabara Dois, 1983.

21. Raspberry Pi 4 Computer Model B Product Brief, 2020. Disponível em: https://static.raspberrypi.org/files/product-briefs/200521+Raspberry+Pi+4+Product+Brief.pdf Acesso em: 13 de Nov. 2020.

22. RILEY, K. F.; HOBSON, M. P.; BENCE, S. J.; **Mathematical Methods for Physics and Engineering.** 3. ed. Nova York: Cambridge University Press, 2006.

23. ROBINSON, Martin. **Getting Started with JUCE**. 1ed. London - UK: Packt Publisher, 2013.

24. RORABAUGH, C. Britton. **Digital Filter Designer's Handbook**. Blue Ridge Summit: McGraw-Hill, 1993.

25. SACHIN B. JADHAV.; NIKHIL N. Mane.; **A Novel High-Speed FPGA Architecture for FIR Filter Design**. International Journal of Reconfigurable and Embedded Systems (IJRES), 2012.

26. LI, Hui. **Configure the Coefficients for Digital Biquad Filters.** Texas Instruments Application Report. SLAA447, April 2010.

27. CAPPER, Derek M.**; Introducing C++ for Scientists, Engineers and Mathematicians.** 2. Ed. Springer-Verlag London, 2001.

28. BRISTOW-JOHNSON, Robert. (2001). **The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers**.

APÊNDICE A

Tabela 3 - Coeficientes do equalizador digital *low-shelf*

	Low-Shelf
$\boldsymbol{b_0}$	$A \cdot \left((A+1) - \cos(\omega_N) \cdot (A-1) + 2 \cdot \sqrt{A} \cdot \alpha\right)$
$\boldsymbol{b_1}$	$2 \cdot A \cdot \left((A-1) - \cos(\omega_N) \cdot (A+1)\right)$
$\boldsymbol{b_2}$	$A \cdot \left((A+1) - \cos(\omega_N) \cdot (A-1) - 2 \cdot \sqrt{A} \cdot \alpha\right)$
$\boldsymbol{a_0}$	$(A+1) + \cos(\omega_N) \cdot (A-1) + 2 \cdot \sqrt{A} \cdot \alpha$
$\boldsymbol{a_1}$	$-2 \cdot \left((A-1) + \cos(\omega_N) \cdot (A+1)\right)$
$\boldsymbol{a_2}$	$(A+1) + \cos(\omega_N) \cdot (A-1) - 2 \cdot \sqrt{A} \cdot \alpha$

Fonte: Bristow-Johnson, Robert, 2001

Tabela 4 - Coeficientes do equalizador digital *high-shelf*

	High-Shelf
$\boldsymbol{b_0}$	$A \cdot \left((A+1) + \cos(\omega_N) \cdot (A-1) + 2 \cdot \sqrt{A} \cdot \alpha\right)$
$\boldsymbol{b_1}$	$-2 \cdot A \cdot \left((A-1) + \cos(\omega_N) \cdot (A+1)\right)$
$\boldsymbol{b_2}$	$A \cdot \left((A+1) + \cos(\omega_N) \cdot (A-1) - 2 \cdot \sqrt{A} \cdot \alpha\right)$
$\boldsymbol{a_0}$	$(A+1) - \cos(\omega_N) \cdot (A-1) + 2 \cdot \sqrt{A} \cdot \alpha$
$\boldsymbol{a_1}$	$2 \cdot \left((A-1) - \cos(\omega_N) \cdot (A+1)\right)$
$\boldsymbol{a_2}$	$(A+1) - \cos(\omega_N) \cdot (A-1) - 2 \cdot \sqrt{A} \cdot \alpha$

Fonte: Bristow-Johnson, Robert, 2001

Tabela 5 - Coeficientes do equalizador digital *peak.*

	Peak
$\boldsymbol{b_0}$	$1 + A \cdot \alpha$
$\boldsymbol{b_1}$	$-2 \cdot \cos(\omega_N)$
$\boldsymbol{b_2}$	$1 - A \cdot \alpha$
$\boldsymbol{a_0}$	$1 + \frac{\alpha}{A}$
$\boldsymbol{a_1}$	$-2 \cdot \cos(\omega_N)$
$\boldsymbol{a_2}$	$1 - \frac{\alpha}{A}$

Fonte: Bristow-Johnson, Robert, 2001

Tabela 6 - Coeficientes do filtro passa-todas

	Passa-Todas
$\boldsymbol{b_0}$	$1 - \alpha$
$\boldsymbol{b_1}$	$-2 \cdot \cos(\omega_N)$
$\boldsymbol{b_2}$	$1 + \alpha$
$\boldsymbol{a_0}$	$\boldsymbol{b_2}$
$\boldsymbol{a_1}$	$\boldsymbol{b_1}$
$\boldsymbol{a_2}$	$\boldsymbol{b_0}$

Fonte: Bristow-Johnson, Robert, 2001.

APÊNDICE B

Listagem 10 - Arquivo *ResonantEQ.cpp* presente na biblioteca *dspLibFEI.hpp*

```
//-----------------------------------------------------------------------------------------------------------
// Centro Universitário FEI - Campus São Bernardo do Campo/SP - 25/01/2021
// Trabalho de Conclusão de Curso: Título de Engenheiro Eletricista com ênfase em telecomunicações
//-----------------------------------------------------------------------------------------------------------
// FEI University Centre - Campus São Bernardo do Campo/SP - 25/01/2021
// Graduate Thesis: Title of Electric Engineer with telecommunications emphasis
//-----------------------------------------------------------------------------------------------------------
//                                                  ^ |G(f)| [dB]
//                CENTRO  UNIVERSITÁRIO             |                      *
//          __________   ___________    ____        |                    *   *
//         /    _____/   /    _______/   /   /|      |                   *     *        DIGITAL SIGNAL PROCESSING IS FUN!
//        /    /|____/  /   /|_______/  /   / /      |                  *       *
//       /    /_____   /   /_______    /   / /       |                 *         *
//      /     _____/  /    _______/   /   / /        |                *           *
//     /    /|_____/ /   /|_______/  /   / /         |          *     *            *     *
//    /    / /      /   /_______    /   / /          |    *    * *   *              *   * *    *
//   /____/ /      /____________/  /___/ /           |  *   * *   * *        fo       * *   * *   *
//   |____|/       |____________/  |___|/         0.0 | *_____*_____*________|________*_____*_____*______> f[Hz]
//
//-----------------------------------------------------------------------------------------------------------
// Filter Types:
//   Peak Filter or Resonant Filter: for boost/cut bands, regulating Q to high values may produce a notch response filter for
//   cut gains (negative gains)
//   All-Pass Filter for phase correction and distortion control
//   High Shelf Filter
//   Low Shelf Filter
// User input variables:
//   Gain
//   Q factor
// DISCLAIMER: This Library works in conjuction with the cmath header and other standard C++ libraries
//-----------------------------------------------------------------------------------------------------------------------

#include <iostream>
#include <cstdio>
#include <cmath>
#include <cstdint>

class resonantEQ
{
public:
    void getFilterInfo();                              // returns filter Type, Coefficients, configuration and tuning parameters
    void printTransferFunction();                      // Prints Z domain transfer function
    void configFilter(int fs, double fc, int order);   // Configures your filter
    void tuneFilter(double q, double g);               // Tunes the user parameters: Q - quality factor and G - filter gain
    void computePeakCoeffs();                          // Computes coefficients for a Peak Filter
    void computeHighShelfCoeffs();                     // Computes coefficients for a High Shelf Filter
    void computeLowShelfCoeffs();                      // Computes coefficients for a Low Shelf Filter
    void computeAllPassCoeffs();                       // Computes coefficients for a All-Pass (Delay Line) Filter
    void resetFilterType();                            // Clears coefficients and reset filter type
    void flo2Fix();                                    // Converts filter coefficients to fixed point coefficients

    void genDatFile(FILE* filePtr, char* fileName, double* sourceArray, int numOfSamples);  // Generates .dat file for gnu plot
    double processInput(double* inputBuffer, double* outputBuffer, int numOfSamples);    // processes input and returns output
    double getImpulseResponse(double* sourceBuffer, int numOfSamples);         // Computes resonant equalizer impulse response

private:
    /* Filter Type Bitmasks */
    static constexpr std::uint_fast8_t defBitMask{ 0b1111'1111 };    // Default Bitmask
    std::uint_fast8_t filtBitMask = defBitMask;                     // Filter Bitmask, initialized with default value

    /* Default Values */
    const unsigned int defaultN = 2;                                // Lowest order computable
    const double defaultFs = 44100;                                 // CD quality
    const double defaultFc = 20.0;                                  // Lowest frequency in human audible range
    const double defaultQ = sqrt(2);                                // Set to be a one octave bandwidth
    const double defaultG = 0.0;            // Neutral gain, we don't want to destroy the user's ears with high gain as default

    /* Tuning Variables */
    unsigned int N = defaultN;                                      // Filter Order
    double Fs = defaultFs;                                          // Sampling Frequency or Sampling Rate
    double Fc = defaultFc;                                          // Frequency of Interest (Cutoff, Boost, Cut and so on...)

    double Q = defaultQ;                                            // Q factor (Appliable only to Peak Filters)
    double G = defaultG;                                            // Filter Gain [dB]

    /* Intermediary Variables */
    const double Pi = 4 * atan(1);                                  // Pi = 3.14159...
    double k = 1.0;                                                 // Linear Gain
    double kExp = 1.0f / N;                                         // Gain Distribution Exponent
    double w = 2 * Pi * (Fc / Fs);                                  // Normalized Angular Frequency [rad / sample]
    double Co = cos(w);                                             // Cosine
    double Si = sin(w);                                             // Sine
    double alpha = Si / (2 * Q);                                    // Filter Damping Factor

    /* Filter Coefficients */
    double b[3] = { 0.0 };                                            // numerator
    double a[3] = { 0.0 };                                            // denominator

    int n[3] = { 0 };                                                 // fixed-integer point numerator
    int d[3] = { 0 };                                                 // fixed-integer point denominator

    int bitNum = 16;                                                // Number of bits for bitRange
    int bitRange = pow(2, bitNum - 1) - 1;                          // largest integer for given bitNum

    /* Filter Buffers */
    double Z[16] = { 0.0 };                // delay buffer
    double X[4096] = { 0.0 };              // Filter Input Samples Buffer
    double Y[4096] = { 0.0 };              // Filter Output Samples Buffer
};
```

```
inline void resonantEQ::getFilterInfo()
{
    printf("\n\n");

    if (filtBitMask == 0x01) printf("Filter Type is: Peak.\n");
    else if (filtBitMask == 0x02) printf("Filter Type is: High Shelf.\n");
    else if (filtBitMask == 0x03) printf("Filter Type is: Low Shelf.\n");
    else if (filtBitMask == 0x04) printf("Filter Type is: All-Pass.\n");
    else printf("A Filter is yet to be defined or it is a unknown type.\n");

    printf("Filter Order: %d \n", N);
    printf("Numerator: %f \t %f \t %f \n", b[0], b[1], b[2]);
    printf("Denominator: %f \t %f \t %f \n", a[0], a[1], a[2]);
    printf("Fixed Integer Numerator: %d \t %d \t %d \n", n[0], n[1], n[2]);
    printf("Fixed Integer Denominator: %d \t %d \t %d \n", d[0], d[1], d[2]);
    printf("Gain: %f, \t Q factor: %f, \t sampling Frequency: %f, \t frequency of interest: %.2f", G, Q, Fs, Fc);

    printf("\n\n");

    return;
}

inline void resonantEQ::printTransferFunction()
{
    printf("\n\n");

    printf("       %f  +  %f * z-1  +  %f * z-2  \n", b[0], b[1], b[2]);
    printf("H(z) = ------------------------------------------------\n");
    printf("       %f  +  %f * z-1  +  %f * z-2  \n", a[0], a[1], a[2]);

    printf("\n\n");
}

inline void resonantEQ::configFilter(int fs, double fc, int order)
{
    if (fs < 8000 || fc > fs / 2 || fc < 0.0 || order % 2 != 0)
    {
        if (fs < 8000)
        {
            printf("\n\n Specified sampling Rate is less than the required for capturing information \n Default Values Will be k
ept for Fs. \n\n");
            /* Apply Correction */
            fs = defaultFs;
            /* Assign Values */
            Fs = fs;
            Fc = fc;
            N = order;
        }

        if (fc > fs / 2 || fc <= 0.0 || fs < 8000)
        {
            printf("\n\n Specified frequency of Interest fc was outside the acceptable bounds: 0 < fc < fs/2 \n Default values w
ill be kept for Fc. \n\n");
            /* Apply Correction */
            fc = defaultFc;
            /* Assign Values */
            Fs = fs;
            Fc = fc;
            N = order;
        }

        if (order % 2 != 0)
        {
            printf("\n\n Specified filter order is not even, therefore SOS cascading is not possible. \n Default Values Will be
kept for N \n\n");
            /* Apply Correction */
            order = defaultN;
            /* Assign Values */
            Fs = fs;
            Fc = fc;
            N = order;
        }

        else
        {
            Fc = fc;
            Fs = fs;
            N = order;
        }
    }

    else
    {
        Fc = fc;
        Fs = fs;
        N = order;
    }
}
```

```
inline void resonantEQ::tuneFilter(double q, double g)
{
    if (q < 0.1 || q > 100.0 || g < -30.0 || g > 30.0)
    {
        if (q < 0.1 || q > 100.0)
        {
            printf("\n\n specified Q factor was outside bounds \n Default values will be kept for Q. \n\n");
            /* Apply Correction */
            q = defaultQ;
            /* Assign Values */
            Q = q;
            G = g;
        }

        if (g < -30.0 || g > 30.0)
        {
            printf("\n\n specified gain was outside bounds \n Default values will be kept for G. \n\n");
            /* Apply Correction */
            g = defaultG;
            /* Assign Values */
            Q = q;
            G = g;
        }

        else
        {
            Q = q;
            G = g;
        }
    }

    else
    {
        Q = q;
        G = g;
    }
}

inline void resonantEQ::computePeakCoeffs()
{
    filtBitMask = defBitMask & 0b0000'0001;

    kExp = 1.0f / N;
    k = pow(10, 0.05 * G); k = pow(k, kExp); //dB to linear gain conversion
    w = 2 * Pi * (Fc / Fs);
    Co = cos(w);
    Si = sin(w);
    alpha = Si / (2 * Q);

    b[0] = (1 + (alpha * k)) / (1 + (alpha / k));
    b[1] = (-2 * Co) / (1 + (alpha / k));
    b[2] = (1 - (alpha * k)) / (1 + (alpha / k));

    a[0] = 1.0f;
    a[1] = b[1];
    a[2] = (1 - (alpha / k)) / (1 + (alpha / k));

    return;
}

inline void resonantEQ::computeHighShelfCoeffs()
{
    filtBitMask = defBitMask & 0b0000'0010;

    kExp = 1.0f / N;
    k = pow(10, 0.05 * G); k = pow(k, kExp); //dB to linear gain conversion
    w = 2 * Pi * (Fc / Fs);
    Co = cos(w);
    Si = sin(w);
    alpha = Si / (2 * sqrt(2));

    b[0] = (k * ((k + 1) + (k - 1) * Co + 2 * sqrt(k) * alpha)) / ((k + 1) - (k - 1) * Co + 2 * sqrt(k) * alpha);
    b[1] = (-2 * k * ((k - 1) + (k + 1) * Co)) / ((k + 1) - (k - 1) * Co + 2 * sqrt(k) * alpha);
    b[2] = (k * ((k + 1) + (k - 1) * Co - 2 * sqrt(k) * alpha)) / ((k + 1) - (k - 1) * Co + 2 * sqrt(k) * alpha);

    a[0] = 1.0f;
    a[1] = (2 * ((k - 1) - (k + 1) * Co)) / ((k + 1) - (k - 1) * Co + 2 * sqrt(k) * alpha);
    a[2] = ((k + 1) - (k - 1) * Co - 2 * sqrt(k) * alpha) / ((k + 1) - (k - 1) * Co + 2 * sqrt(k) * alpha);

    return;
}
```

```
inline void resonantEQ::computeLowShelfCoeffs()
{
    filtBitMask = defBitMask & 0b0000'0011;

    kExp = 1.0f / N;
    k = pow(10, 0.05 * G); k = pow(k, kExp); //dB to linear gain conversion
    w = 2 * Pi * (Fc / Fs);
    Co = cos(w);
    Si = sin(w);
    alpha = Si / (2 * sqrt(2));

    b[0] = (k * ((k + 1) - (k - 1) * Co + 2 * sqrt(k) * alpha)) / ((k + 1) + (k - 1) * Co + 2 * sqrt(k) * alpha);
    b[1] = (2 * k * ((k - 1) - (k + 1) * Co)) / ((k + 1) + (k - 1) * Co + 2 * sqrt(k) * alpha);
    b[2] = (k * ((k + 1) - (k - 1) * Co - 2 * sqrt(k) * alpha)) / ((k + 1) + (k - 1) * Co + 2 * sqrt(k) * alpha);

    a[0] = 1.0f;
    a[1] = (-2 * ((k - 1) + (k + 1) * Co)) / ((k + 1) + (k - 1) * Co + 2 * sqrt(k) * alpha);
    a[2] = ((k + 1) + (k - 1) * Co - 2 * sqrt(k) * alpha) / ((k + 1) + (k - 1) * Co + 2 * sqrt(k) * alpha);

    return;
}

inline void resonantEQ::computeAllPassCoeffs()
{
    filtBitMask = defBitMask & 0b0000'0100;

    w = 2 * Pi * (Fc / Fs);
    Co = cos(w);
    Si = sin(w);
    alpha = Si / (2 * Q);

    b[0] = 1 - alpha;
    b[1] = -2 * Co;
    b[2] = 1 + alpha;

    a[0] = b[2];
    a[1] = b[1];
    a[2] = b[0];

    return;
}

inline void resonantEQ::resetFilterType()
{
    filtBitMask = defBitMask;

    N = defaultN;
    Fs = defaultFs;
    Fc = defaultFc;
    Q = defaultQ;
    G = defaultG;

    b[0] = 0.0;
    b[1] = 0.0;
    b[2] = 0.0;

    a[0] = 0.0;
    a[1] = 0.0;
    a[2] = 0.0;
}

inline void resonantEQ::clearCoeffs()
{
    n[0] = 0;
    n[1] = 0;
    n[2] = 0;

    d[0] = 0;
    d[1] = 0;
    d[2] = 0;

    printf("\nFilter Coefficients have been cleared.\n");
}

inline void resonantEQ::flo2Fix()
{
    n[0] = (int)(b[0] * bitRange);
    n[1] = (int)(b[1] * bitRange);
    n[2] = (int)(b[2] * bitRange);

    d[0] = bitRange + 1;
    d[1] = (int)(a[1] * bitRange);
    d[2] = (int)(a[2] * bitRange);

    return;
}

inline void resonantEQ::genDatFile(FILE* filePtr, char* fileName, double* sourceArray, int numOfSamples)
{
    filePtr = fopen(fileName, "w");

    for (int i = 0; i < numOfSamples; i++)
    {
        fprintf(filePtr, "\n %f", sourceArray[i]);
    }

    fclose(filePtr);
}
```

```
inline double resonantEQ::processInput(double* inputBuffer, double* outputBuffer, int numOfSamples)
{
    if (numOfSamples > 4096) numOfSamples = 4096;
    else numOfSamples = numOfSamples;

    for (int n = 0; n < numOfSamples; n++)
    {
        X[n] = inputBuffer[n]; // Write input into filter sample Buffer
        for (int k = 0; k < N; k = k + 2)
        {
            Y[n] = b[0] * X[n] + Z[k];
            Z[k] = b[1] * X[n] - a[1] * Y[n] + Z[k + 1];
            Z[k + 1] = b[2] * X[n] - a[2] * Y[n];

            X[n] = Y[n]; // Cascade filter output back into the input, thus performing a SOS cascade (if N > 2)
        }
        outputBuffer[n] = Y[n]; // Write result into Output Buffer
    }

    return 0;
}

inline double resonantEQ::getImpulseResponse(double* sourceBuffer, int numOfSamples)
{
    if (numOfSamples > 4096) numOfSamples = 4096;
    else numOfSamples = numOfSamples;

    X[0] = 1.0;
    for (int n = 0; n < numOfSamples; n++)
    {
        for (int k = 0; k < N; k = k + 2)
        {
            Y[n] = b[0] * X[n] + Z[k];
            Z[k] = b[1] * X[n] - a[1] * Y[n] + Z[k + 1];
            Z[k + 1] = b[2] * X[n] - a[2] * Y[n];

            X[n] = Y[n];
        }
        sourceBuffer[n] = Y[n];
    }

    return 0;
}
```

APÊNDICE C

Listagem 11 - Arquivo *MainComponent.hpp*

```
#pragma once

#include <JuceHeader.h>
#include "dspLibFEI.cpp"

/*
    This component lives inside our window, and this is where you should put
       all your controls and content.
*/
class MainComponent :
    public juce::AudioAppComponent,
    private juce::Timer
{
public:
    MainComponent();
    ~MainComponent() override;
    //==========================================================================
    void prepareToPlay (int samplesPerBlockExpected, double sampleRate) override;
    void getNextAudioBlock (const juce::AudioSourceChannelInfo& bufferToFill
override;
    void releaseResources() override;
    //==========================================================================
    void paint (juce::Graphics& g) override;
    void drawFrame(juce::Graphics& g);
    void drawNextFrameOfSpectrum();
    void resized() override;
    //==========================================================================
    void timerCallback() override;
    void pushNextSampleIntoFifo(float Sample);
    //==========================================================================
    enum
    {
        fftOrder = 11, // 2^11
        fftSize = 1 << fftOrder,
        scopeSize = 512
    };

private:
    // Your private member variables go here...
    juce::dsp::FFT fowardFFT;
    juce::dsp::WindowingFunction<float> window;

    float FiFo[fftSize] = {0.0f};
    float fftData[2 * fftSize] = {0.0f};
    float scopeData[scopeSize] = {0.0f};
    int bufferIndex = 0;
    bool readyToLoadNextBlock = false;

    // Global Gain Slider and Label
    juce::Slider globalGainSlider;
    juce::Label globalGainLabel;

    //First Filter Sliders and Labels
    juce::Label  gainOneLabel;
    juce::Label qOneLabel;

    juce::Slider gainOneSlider;
    juce::Slider qOneSlider;

    //Second Filter Sliders and Labels
    juce::Label gainTwoLabel;
    juce::Label qTwoLabel;

    juce::Slider gainTwoSlider;
    juce::Slider qTwoSlider;
```

```
    //Third Filter Sliders and Labels
    juce::Label gainThreeLabel;
    juce::Label qThreeLabel;

    juce::Slider gainThreeSlider;
    juce::Slider qThreeSlider;

    //Fourth Filter Sliders and Labels
    juce::Label gainFourLabel;
    juce::Label qFourLabel;

    juce::Slider gainFourSlider;
    juce::Slider qFourSlider;

    //Fifth Filter Sliders and Labels
    juce::Label gainFiveLabel;
    juce::Label qFiveLabel;

    juce::Slider gainFiveSlider;
    juce::Slider qFiveSlider;

    //Sixth Filter Sliders and Labels
    juce::Label gainSixLabel;
    juce::Label qSixLabel;

    juce::Slider gainSixSlider;
    juce::Slider qSixSlider;

    //Seventh Filter Sliders and Labels
    juce::Label gainSevenLabel;
    juce::Label qSevenLabel;

    juce::Slider gainSevenSlider;
    juce::Slider qSevenSlider;

    //Eighth Filter Sliders and Labels
    juce::Label gainEightLabel;
    juce::Label qEightLabel;

    juce::Slider gainEightSlider;
    juce::Slider qEightSlider;
    //==============================================================================

    // Filters (implemented via dspLibFEI)
    resonantEQ H1L;
    resonantEQ H2L;
    resonantEQ H3L;
    resonantEQ H4L;
    resonantEQ H5L;
    resonantEQ H6L;
    resonantEQ H7L;
    resonantEQ H8L;

    resonantEQ H1R;
    resonantEQ H2R;
    resonantEQ H3R;
    resonantEQ H4R;
    resonantEQ H5R;
    resonantEQ H6R;
    resonantEQ H7R;
    resonantEQ H8R;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
};
```

Listagem 12 - Arquivo MainComponent.cpp

```
#include "MainComponent.h"
#include <stdio.h>

//==============================================================================================================
MainComponent::MainComponent():
    fowardFFT(fftOrder),
    window (fftSize, juce::dsp::WindowingFunction<float>::hann)
{
    // Make sure you set the size of the component after
    // you add any child components.
    setSize (1040, 800);

    // Global Gain ================================================================================================
    globalGainSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
    globalGainSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
    globalGainSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

    globalGainSlider.setRange(-10.0f, 10.0f);
    globalGainSlider.setTextValueSuffix(" dB ");
    globalGainSlider.setValue(0.0f);   // Sets GainSlider Initial Value to 0.0 dB
    globalGainSlider.setSkewFactorFromMidPoint(0.5); // Setting Slider Track to be logarithmic

    globalGainLabel.setText(" Global Gain ", juce::dontSendNotification);
    globalGainLabel.attachToComponent(&globalGainSlider, false);

    addAndMakeVisible(globalGainLabel);
    addAndMakeVisible(globalGainSlider);

    // 125 Hz Q and Gain ==========================================================================================
    gainOneSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
    gainOneSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
    gainOneSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

    gainOneSlider.setRange(-10.0f, 10.0f);
    gainOneSlider.setTextValueSuffix(" dB ");
    gainOneSlider.setValue(0.0f);
    gainOneSlider.setSkewFactorFromMidPoint(0.5);

    gainOneLabel.setText(" 125 Hz Gain ", juce::dontSendNotification);
    gainOneLabel.attachToComponent(&gainOneSlider, false);

    qOneSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
    qOneSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
    qOneSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

    qOneSlider.setValue(1.0f);
    qOneSlider.setRange(1.0f, 100.0f);

    qOneLabel.setText(" 125 Hz Q ", juce::dontSendNotification);
    qOneLabel.attachToComponent(&qOneSlider, false);

    addAndMakeVisible(gainOneLabel);
    addAndMakeVisible(gainOneSlider);
    addAndMakeVisible(qOneLabel);
    addAndMakeVisible(qOneSlider);

    // 250 Hz Q and Gain  =========================================================================================
    gainTwoSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
    gainTwoSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
    gainTwoSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

    gainTwoSlider.setRange(-10.0f, 10.0f);
    gainTwoSlider.setTextValueSuffix(" dB ");
    gainTwoSlider.setValue(0.0f);
    gainTwoSlider.setSkewFactorFromMidPoint(0.5);

    gainTwoLabel.setText(" 250 Hz Gain ", juce::dontSendNotification);
    gainTwoLabel.attachToComponent(&gainTwoSlider, false);

    qTwoSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
    qTwoSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
    qTwoSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

    qTwoSlider.setValue(1.0f);
    qTwoSlider.setRange(1.0f, 100.0f);

    qTwoLabel.setText(" 250 Hz Q ", juce::dontSendNotification);
    qTwoLabel.attachToComponent(&qTwoSlider, false);

    addAndMakeVisible(gainTwoLabel);
    addAndMakeVisible(gainTwoSlider);
    addAndMakeVisible(qTwoLabel);
    addAndMakeVisible(qTwoSlider);
```

```
// 500 Hz Q and Gain  ==========================================================================
gainThreeSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
gainThreeSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
gainThreeSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

gainThreeSlider.setRange(-10.0f, 10.0f);
gainThreeSlider.setTextValueSuffix(" dB ");
gainThreeSlider.setValue(0.0f);
gainThreeSlider.setSkewFactorFromMidPoint(0.5);

gainThreeLabel.setText(" 500 Hz Gain ", juce::dontSendNotification);
gainThreeLabel.attachToComponent(&gainThreeSlider, false);

qThreeSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
qThreeSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
qThreeSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

qThreeSlider.setValue(1.0f);
qThreeSlider.setRange(1.0f, 100.0f);

qThreeLabel.setText(" 500 Hz Q ", juce::dontSendNotification);
qThreeLabel.attachToComponent(&qThreeSlider, false);

addAndMakeVisible(gainThreeLabel);
addAndMakeVisible(gainThreeSlider);
addAndMakeVisible(qThreeLabel);
addAndMakeVisible(qThreeSlider);

// 1000 Hz Q and Gain  ==========================================================================
gainFourSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
gainFourSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
gainFourSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

gainFourSlider.setRange(-10.0f, 10.0f);
gainFourSlider.setTextValueSuffix(" dB ");
gainFourSlider.setValue(0.0f);
gainFourSlider.setSkewFactorFromMidPoint(0.5);

gainFourLabel.setText(" 1000 Hz Gain ", juce::dontSendNotification);
gainFourLabel.attachToComponent(&gainFourSlider, false);

qFourSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
qFourSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
qFourSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

qFourSlider.setValue(1.0f);
qFourSlider.setRange(1.0f, 100.0f);

qFourLabel.setText(" 1000 Hz Q ", juce::dontSendNotification);
qFourLabel.attachToComponent(&qFourSlider, false);

addAndMakeVisible(gainFourLabel);
addAndMakeVisible(gainFourSlider);
addAndMakeVisible(qFourLabel);
addAndMakeVisible(qFourSlider);

// 2000 Hz Q and Gain  ==========================================================================
gainFiveSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
gainFiveSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
gainFiveSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

gainFiveSlider.setRange(-10.0f, 10.0f);
gainFiveSlider.setTextValueSuffix(" dB ");
gainFiveSlider.setValue(0.0f);
gainFiveSlider.setSkewFactorFromMidPoint(0.5);

gainFiveLabel.setText(" 2000 Hz Gain ", juce::dontSendNotification);
gainFiveLabel.attachToComponent(&gainFiveSlider, false);

qFiveSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
qFiveSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
qFiveSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

qFiveSlider.setValue(1.0f);
qFiveSlider.setRange(1.0f, 100.0f);

qFiveLabel.setText(" 2000 Hz Q ", juce::dontSendNotification);
qFiveLabel.attachToComponent(&qFiveSlider, false);

addAndMakeVisible(gainFiveLabel);
addAndMakeVisible(gainFiveSlider);
addAndMakeVisible(qFiveLabel);
addAndMakeVisible(qFiveSlider);
```

```
// 4000 Hz Q and Gain  ==========================================================================
gainSixSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
gainSixSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
gainSixSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

gainSixSlider.setRange(-10.0f, 10.0f);
gainSixSlider.setTextValueSuffix(" dB ");
gainSixSlider.setValue(0.0f);
gainSixSlider.setSkewFactorFromMidPoint(0.5);

gainSixLabel.setText(" 4000 Hz Gain ", juce::dontSendNotification);
gainSixLabel.attachToComponent(&gainSixSlider, false);

qSixSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
qSixSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
qSixSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

qSixSlider.setValue(1.0f);
qSixSlider.setRange(1.0f, 100.0f);

qSixLabel.setText(" 4000 Hz Q ", juce::dontSendNotification);
qSixLabel.attachToComponent(&qSixSlider, false);

addAndMakeVisible(gainSixLabel);
addAndMakeVisible(gainSixSlider);
addAndMakeVisible(qSixLabel);
addAndMakeVisible(qSixSlider);

// 8000 Hz Q and Gain  ==========================================================================
gainSevenSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
gainSevenSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
gainSevenSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

gainSevenSlider.setRange(-10.0f, 10.0f);
gainSevenSlider.setTextValueSuffix(" dB ");
gainSevenSlider.setValue(0.0f);
gainSevenSlider.setSkewFactorFromMidPoint(0.5);

gainSevenLabel.setText(" 8000 Hz Gain ", juce::dontSendNotification);
gainSevenLabel.attachToComponent(&gainSevenSlider, false);

qSevenSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
qSevenSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
qSevenSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

qSevenSlider.setValue(1.0f);
qSevenSlider.setRange(1.0f, 100.0f);

qSevenLabel.setText(" 8000 Hz Q ", juce::dontSendNotification);
qSevenLabel.attachToComponent(&qSevenSlider, false);

addAndMakeVisible(gainSevenLabel);
addAndMakeVisible(gainSevenSlider);
addAndMakeVisible(qSevenLabel);
addAndMakeVisible(qSevenSlider);

// 16000 Hz Q and Gain  =========================================================================
gainEightSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
gainEightSlider.setColour(juce::Slider::thumbColourId, juce::Colours::lightseagreen);
gainEightSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

gainEightSlider.setRange(-10.0f, 10.0f);
gainEightSlider.setTextValueSuffix(" dB ");
gainEightSlider.setValue(0.0f);
gainEightSlider.setSkewFactorFromMidPoint(0.5);

gainEightLabel.setText(" 16000 Hz Gain ", juce::dontSendNotification);
gainEightLabel.attachToComponent(&gainEightSlider, false);

qEightSlider.setSliderStyle(juce::Slider::SliderStyle::Rotary);
qEightSlider.setColour(juce::Slider::thumbColourId, juce::Colours::blueviolet);
qEightSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 120, 20);

qEightSlider.setValue(1.0f);
qEightSlider.setRange(1.0f, 100.0f);

qEightLabel.setText(" 16000 Hz Q ", juce::dontSendNotification);
qEightLabel.attachToComponent(&qEightSlider, false);

addAndMakeVisible(gainEightLabel);
addAndMakeVisible(gainEightSlider);
addAndMakeVisible(qEightLabel);
addAndMakeVisible(qEightSlider);
```

```
    // Some platforms require permissions to open input channels so request that here
    if (juce::RuntimePermissions::isRequired (juce::RuntimePermissions::recordAudio)
        && ! juce::RuntimePermissions::isGranted (juce::RuntimePermissions::recordAudio))
    {
        juce::RuntimePermissions::request (juce::RuntimePermissions::recordAudio,
                                           [&] (bool granted) { setAudioChannels (granted ? 2 : 0, 2); });
    }
    else
    {
        // Specify the number of input and output channels that we want to open
        setAudioChannels (2, 2);    // 2 Input Channels, 2 Output Channels
    }
    startTimerHz(30);
}

MainComponent::~MainComponent()
{
    // This shuts down the audio device and clears the audio source.
    shutdownAudio();
}

void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    // This function will be called when the audio device is started, or when
    // its settings (i.e. sample rate, block size, etc) are changed.

    // You can use this function to initialise any resources you might need,
    // but be careful - it will be called on the audio thread, not the GUI thread.

    // For more details, see the help for AudioProcessor::prepareToPlay()

    //------------------------------------------------------------------------------------------------------------------
    /* MY CODE BEGINS FROM HERE */
    /* Tutorials Provided by JUCE made this code possible */
    //------------------------------------------------------------------------------------------------------------------

    /* Filter Bank Setup */
    H1L.configFilter(sampleRate, 125, 8);
    H2L.configFilter(sampleRate, 250, 8);
    H3L.configFilter(sampleRate, 500, 8);
    H4L.configFilter(sampleRate, 1000, 8);
    H5L.configFilter(sampleRate, 2000, 8);
    H6L.configFilter(sampleRate, 4000, 8);
    H7L.configFilter(sampleRate, 8000, 8);
    H8L.configFilter(sampleRate, 16000, 8);

    H1R.configFilter(sampleRate, 125, 8);
    H2R.configFilter(sampleRate, 250, 8);
    H3R.configFilter(sampleRate, 500, 8);
    H4R.configFilter(sampleRate, 1000, 8);
    H5R.configFilter(sampleRate, 2000, 8);
    H6R.configFilter(sampleRate, 4000, 8);
    H7R.configFilter(sampleRate, 8000, 8);
    H8R.configFilter(sampleRate, 16000, 8);

}

void MainComponent::getNextAudioBlock (const juce::AudioSourceChannelInfo& bufferToFill)
{
    // Your audio-processing code goes here!
    // For more details, see the help for AudioProcessor::getNextAudioBlock()

    //------------------------------------------------------------------------------------------------------------------
    /* MY CODE BEGINS FROM HERE */
    /* Tutorials Provided by JUCE made this code possible */
    //------------------------------------------------------------------------------------------------------------------

    /* Set up Audio Devices */
    auto* device = deviceManager.getCurrentAudioDevice();
    double sampleRate = device->getCurrentSampleRate();
    int numOfSamples = bufferToFill.numSamples;
    prepareToPlay(numOfSamples, sampleRate);

    /* Tune left side */
    H1L.tuneFilter(qOneSlider.getValue(), gainOneSlider.getValue()); H1L.computePeakCoeffs();
    H2L.tuneFilter(qTwoSlider.getValue(), gainTwoSlider.getValue()); H2L.computePeakCoeffs();
    H3L.tuneFilter(qThreeSlider.getValue(), gainThreeSlider.getValue()); H3L.computePeakCoeffs();
    H4L.tuneFilter(qFourSlider.getValue(), gainFourSlider.getValue()); H4L.computePeakCoeffs();
    H5L.tuneFilter(qFiveSlider.getValue(), gainFiveSlider.getValue()); H5L.computePeakCoeffs();
    H6L.tuneFilter(qSixSlider.getValue(), gainSixSlider.getValue()); H6L.computePeakCoeffs();
    H7L.tuneFilter(qSevenSlider.getValue(), gainSevenSlider.getValue()); H7L.computePeakCoeffs();
    H8L.tuneFilter(qEightSlider.getValue(), gainEightSlider.getValue()); H8L.computePeakCoeffs();
```

```
    /* Tune right side */
    H1R.tuneFilter(qOneSlider.getValue(), gainOneSlider.getValue()); H1R.computePeakCoeffs();
    H2R.tuneFilter(qTwoSlider.getValue(), gainTwoSlider.getValue()); H2R.computePeakCoeffs();
    H3R.tuneFilter(qThreeSlider.getValue(), gainThreeSlider.getValue()); H3R.computePeakCoeffs();
    H4R.tuneFilter(qFourSlider.getValue(), gainFourSlider.getValue()); H4R.computePeakCoeffs();
    H5R.tuneFilter(qFiveSlider.getValue(), gainFiveSlider.getValue()); H5R.computePeakCoeffs();
    H6R.tuneFilter(qSixSlider.getValue(), gainSixSlider.getValue()); H6R.computePeakCoeffs();
    H7R.tuneFilter(qSevenSlider.getValue(), gainSevenSlider.getValue()); H7R.computePeakCoeffs();
    H8R.tuneFilter(qEightSlider.getValue(), gainEightSlider.getValue()); H8R.computePeakCoeffs();

    // Input Buffers
    auto* leftBufferIn = bufferToFill.buffer->getReadPointer(0, bufferToFill.startSample);
    auto* rightBufferIn = bufferToFill.buffer->getReadPointer(1, bufferToFill.startSample);

    //Output Buffers
    auto* leftBufferOut = bufferToFill.buffer->getWritePointer(0, bufferToFill.startSample);
    auto* rightBufferOut = bufferToFill.buffer->getWritePointer(1, bufferToFill.startSample);

     double leftChannel[4096] = { 0.0 };
     double rightChannel[4096] = { 0.0 };

     // Fill the buffers
     for(int i = 0; i < numOfSamples; i++)
     {
        leftChannel[i] = leftBufferIn[i];
        rightChannel[i] = rightBufferIn[i];
     }

     auto Gain = (float)globalGainSlider.getValue(); // Global Gain
     auto K = pow(10, 0.05 * Gain);                  // Global Gain conversion to Linear gain

     H1L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H1R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     H2L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H2R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     H3L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H3R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     H4L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H4R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     H5L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H5R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     H6L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H6R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     H7L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H7R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     H8L.processInput(&leftChannel[0], &leftChannel[0], numOfSamples);
     H8R.processInput(&rightChannel[0], &rightChannel[0], numOfSamples);

     // Release the buffers
     for(int j = 0; j < numOfSamples; j++)
     {

        pushNextSampleIntoFifo(K * leftChannel[j]);

        leftBufferOut[j] = K * leftChannel[j];
        rightBufferOut[j] = K * rightChannel[j];
     }
}

void MainComponent::releaseResources()
{
    // This will be called when the audio device stops, or when it is being
    // restarted due to a setting change.

    // For more details, see the help for AudioProcessor::releaseResources()
}

void MainComponent::pushNextSampleIntoFifo(float Sample)
{
    if(bufferIndex == fftSize)
    {
        if (!readyToLoadNextBlock)
        {
            juce::zeromem(fftData, sizeof(fftData));
            memcpy(fftData, FiFo, sizeof(FiFo));
            readyToLoadNextBlock = true;
        }
```

```
        bufferIndex = 0;
    }

    FiFo[bufferIndex++] = Sample;
}

void MainComponent::drawNextFrameOfSpectrum()
{
    window.multiplyWithWindowingTable(fftData, fftSize);

    fowardFFT.performFrequencyOnlyForwardTransform(fftData);

    auto mindB = -100.0f;
    auto maxdB = 0.0f;

    for(int i = 0; i < scopeSize; i++)
    {
        auto skewedProportionX = 1.0f - std::exp(std::log(1.0f - (float)i / (float)scopeSize) * 0.2f);
        auto fftDataIndex = juce::jlimit(0, fftSize / 2, (int)(skewedProportionX * (float)fftSize * 0.5f));
        auto level = juce::jmap(juce::jlimit(mindB, maxdB, juce::Decibels::gainToDecibels(fftData[fftDataIndex
]) - juce::Decibels::gainToDecibels((float)fftSize)),
            mindB, maxdB, 0.0f, 1.0f);

        scopeData[i] = level;
    }
}

void MainComponent::timerCallback()
{
    if (readyToLoadNextBlock)
    {
        drawNextFrameOfSpectrum();
        readyToLoadNextBlock = false;
        repaint();
    }
}

void MainComponent::drawFrame(juce::Graphics& s)
{
    for (int i = 1; i < scopeSize; ++i)
    {

        auto width = getLocalBounds().getWidth();
        auto height = 350;

        s.drawLine({ (float)juce::jmap(i - 1, 0, scopeSize - 1, 0, width),
            juce::jmap(scopeData[i - 1], 0.0f, 1.0f, (float)height, 0.0f),
            (float)juce::jmap(i,     0, scopeSize - 1, 0, width),
            juce::jmap(scopeData[i],     0.0f, 1.0f, (float)height, 0.0f) });
    }
}

void MainComponent::paint(juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));
    g.setColour(juce::Colours::black);
    g.fillRect(0.0, 0.0, getLocalBounds().getWidth(), 350.0);
    // You can add your drawing code here!

    g.setColour(juce::Colours::white);
    drawFrame(g);
}

void MainComponent::resized()
{
    // This is called when the MainContentComponent is resized.
    // If you add any child components, this is where you should
    // update their positions.

    /*Rectangle Positioning*/
    juce::Rectangle<int> totalArea = getLocalBounds();
    juce::Rectangle<int>SpectrumArea = totalArea.removeFromTop(totalArea.getHeight() / 2);

    int offsetX = 50;

    globalGainSlider.setBounds(460,380,100,100);

    gainOneSlider.setBounds(offsetX, getHeight() - 280, 100, 100);
    qOneSlider.setBounds(offsetX, getHeight() - 150, 100, 100);

    gainTwoSlider.setBounds(offsetX + 120, getHeight() - 280, 100, 100);
    qTwoSlider.setBounds(offsetX + 120, getHeight() - 150, 100, 100);
```

```
    gainThreeSlider.setBounds(offsetX + 240, getHeight() - 280, 100, 100);
    qThreeSlider.setBounds(offsetX + 240, getHeight() - 150, 100, 100);

    gainFourSlider.setBounds(offsetX + 360, getHeight() - 280, 100, 100);
    qFourSlider.setBounds(offsetX + 360, getHeight() - 150, 100, 100);

    gainFiveSlider.setBounds(offsetX + 480, getHeight() - 280, 100, 100);
    qFiveSlider.setBounds(offsetX + 480, getHeight() - 150, 100, 100);

    gainSixSlider.setBounds(offsetX + 600, getHeight() - 280, 100, 100);
    qSixSlider.setBounds(offsetX + 600, getHeight() - 150, 100, 100);

    gainSevenSlider.setBounds(offsetX + 720, getHeight() - 280, 100, 100);
    qSevenSlider.setBounds(offsetX + 720, getHeight() - 150, 100, 100);

    gainEightSlider.setBounds(offsetX + 840, getHeight() - 280, 100, 100);
    qEightSlider.setBounds(offsetX + 840, getHeight() - 150, 100, 100);
}
```

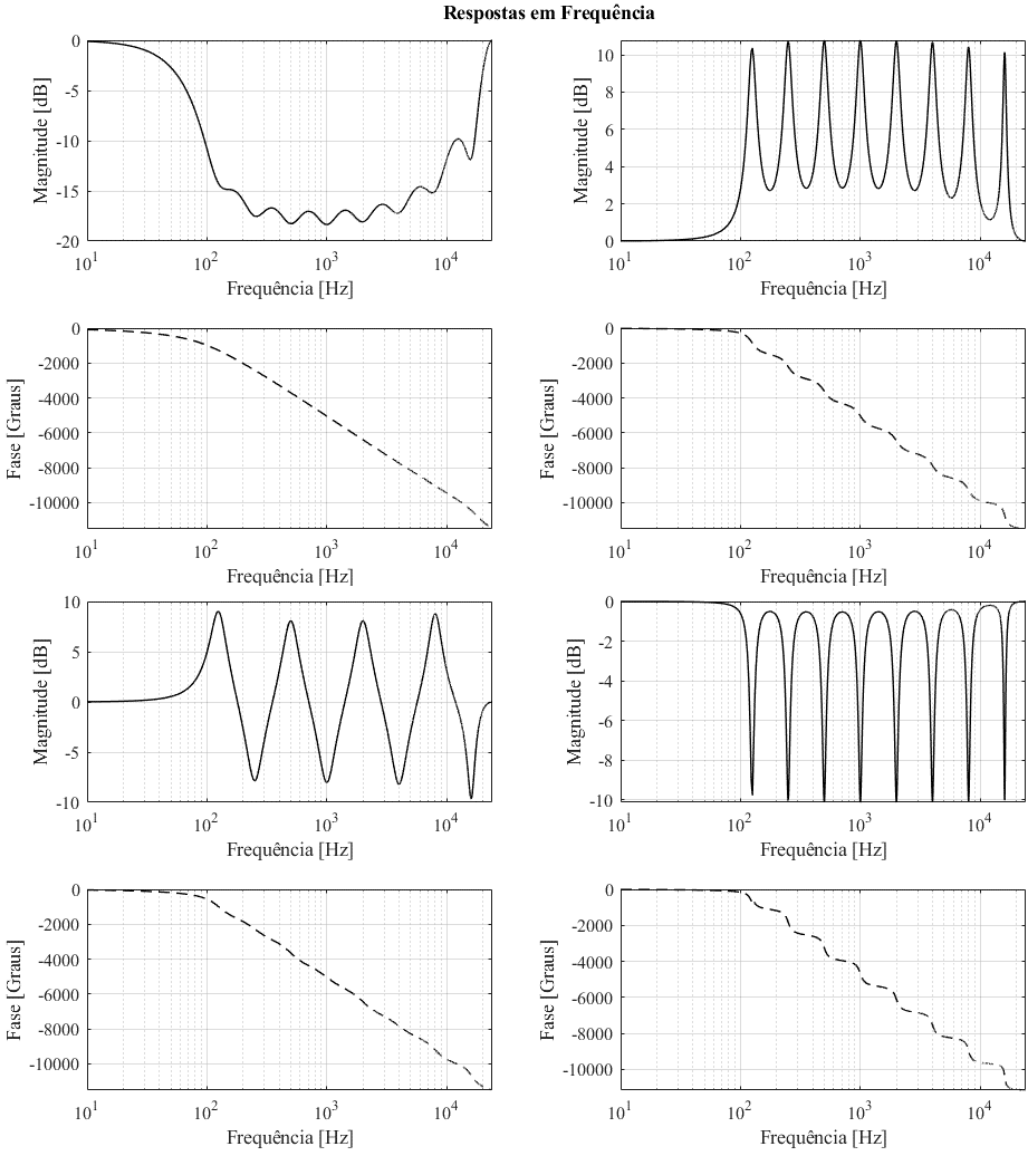
Listagem 13 - Código da função principal do software equalizador paramétrico digital.

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

    This file contains the basic startup code for a JUCE application.

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

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

//==============================================================================
class ParametricEqualizerApplication  : public juce::JUCEApplication
{
public:
    //==============================================================================
    ParametricEqualizerApplication() {}

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

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

        mainWindow.reset (new MainWindow ("Parametric Equalizer FEI"));
    }

    void shutdown() override
    {
        // Add your application's shutdown code here..

        mainWindow = nullptr; // (deletes our window)
    }

    //==============================================================================
    void systemRequestedQuit() override
    {
        // This is called when the app is being asked to quit: you can ignore this
        // request and let the app carry on running, or call quit() to allow the app to close.
        quit();
    }

    void anotherInstanceStarted (const juce::String& commandLine) override
    {
        // When another instance of the app is launched while this one is running,
        // this method is invoked, and the commandLine parameter tells you what
        // the other instance's command-line arguments were.
    }

    //==============================================================================
    /*
        This class implements the desktop window that contains an instance of
        our MainComponent class.
    */
    class MainWindow : public juce::DocumentWindow
    {
    public:
        MainWindow(juce::String name) :
            DocumentWindow(name,
                juce::Desktop::getInstance().getDefaultLookAndFeel().findColour(juce::ResizableWindow::backgro
undColourId),
                DocumentWindow::allButtons)
        {
            setTitleBarButtonsRequired(5, false);
            setUsingNativeTitleBar (true);
            setContentOwned (new MainComponent(), true);

           #if JUCE_IOS || JUCE_ANDROID
            setFullScreen (true);
           #else
            setResizable (false, false);
            centreWithSize (getWidth(), getHeight());
           #endif

            setVisible (true);
        }

        void closeButtonPressed() override
        {
            // This is called when the user tries to close this window. Here, we'll just
            // ask the app to quit when this happens, but you can change this to do
            // whatever you need.
```

```
            JUCEApplication::getInstance()->systemRequestedQuit();
        }

        /* Note: Be careful if you override any DocumentWindow methods - the base
           class uses a lot of them, so by overriding you might break its functionality.
           It's best to do all your work in your content component instead, but if
           you really have to override any DocumentWindow methods, make sure your
           subclass also calls the superclass's method.
        */

    private:
        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainWindow)
    };

private:
    std::unique_ptr<MainWindow> mainWindow;
};

//==============================================================================
// This macro generates the main() routine that launches the app.
START_JUCE_APPLICATION (ParametricEqualizerApplication)
```

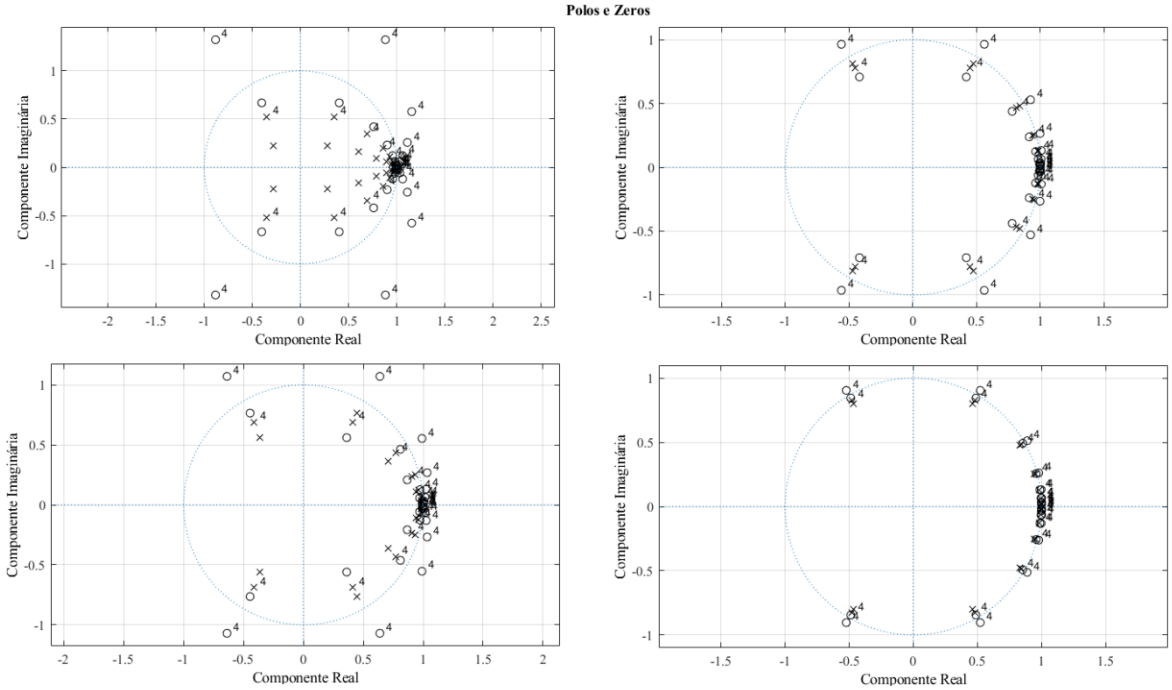
APÊNDICE D

Figura 60 - Respostas em frequência variadas.



Fonte: Autor.

Figura 61 - Diagrama de polos e zeros dos sistemas pertencentes a figura 54.



Fonte: Autor.