Processamento flexível de sinais de áudio em tempo real

Thilo Koch

Tese apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Doutor em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Marcelo Gomes de Queiroz

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, maio de 2022

Processamento flexível de sinais de áudio em tempo real

Esta versão da dissertação/tese contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 29/08/2022. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Marcelo Gomes de Queiroz (orientador) - IME-USP
- Prof. Dr. Rodrigo Schramm - UFRGS
- Prof. Dr. Marcelo Mortensen Wanderley - MCGILL CANADA
- Prof. Dr. Flávio Luiz Schiavoni - UFSJ
- Prof. Dr. Jônatas Manzolli - UNICAMP

Resumo

KOCH, T. **Processamento flexível de sinais de áudio em tempo real**. 2022. 164 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Programas de processamento de áudio em tempo real são comuns há décadas, e correspondem a aplicações que possuem muitas formas e rodam em plataformas computacionais diversas. Todavia, todas as plataformas têm em comum o fato de possuírem um limite de capacidade computacional, o que se torna crítico especialmente em situação de sobrecarga, acarretando resultados indesejáveis como interrupções intermitentes, artefatos sonoros ou mesmo a parada completa da execução do sistema.

Este trabalho apresenta uma metodologia que permite a realização de um *trade-off* flexível entre os custos computacionais do processamento e a qualidade percebida do resultado desse processamento. Assim, certas situações de sobrecarga podem ser evitadas sacrificando-se parcialmente a qualidade. Essa flexibilização, permite a adaptação dos custos do sistema de forma dinâmica, em tempo de execução, controlando a qualidade do resultado para evitar tais situações de sobrecarga. Oferece, portanto, uma abordagem para analisar e parametrizar os elementos do processamento de áudio de forma coordenada e atrelada tanto aos custos computacionais quanto às medidas de qualidade correspondentes. Com otimizações, esse método pode ser usado também para adaptar estaticamente algoritmos e parâmetros a outras plataformas de hardware com recursos computacionais diferentes.

Palavras-chave: *trade-off* , percepção de áudio, custos computacionais flexíveis, processamento de sinais de áudio em tempo real, afinação automática de aplicações de processamento de sinais de áudio.

Abstract

KOCH, T. **Flexible audio signal processing in real-time**. 2022. 162 p. Thesis (Doctorate) - Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Real time audio processing applications common for decades and running on a wide range of hardware in many diferent forms. However, all these platforms have in common that have a limited computational ressources which turns critical in overload situations. This results in unwanted sound events: strong distortions, clicks and interruptions.

This work presents a methodolgy which allows for the realization of a trade-off between computing costs and the perceived quality of the resulting audio. With this approach the computational costs of the elements of the audio processing chain can be analyzed and parameterized to control them dynamically thus avoiding overload situations of the system sacrificing quality partially. This flexibilization is realized by means of a trade-off between computational costs and the perceived quality of the audio output supposing the existence of perceptual space of representation in which the signal can be modified without much loss of quality.

Keywords: *trade-off* , audio perception, flexible computing costs, real-time audio signal processing, audio application autotuning.

Sumário

Lista de Abreviaturas

AES *Audio Engineering Society* (Sociedade de engenharia de áudio)

ALSA *Advanced Linux Sound Architecture* (Arquitetura avançada de som em Linux)

API *Application Programming Interface* (Interface de programação de aplicações)

BSD *Berkeley Software Distribution* (Distribuição de software da Universidade Berkley)

CD *Compact Disc* (Disco compacto a laser)

CISC *Complex Instruction Set Computer* (Computador com um conjunto de instruções complexas)

CPU *Central Processing Unit* (Unidade central de processamento)

DAW *Digital Audio Workstation* (Estação de trabalho de áudio digital)

DI *Distortion Index* (Índice de distorção)

DIX *Disturbance Index* (Índice de perturbação)

DPCM *Differential Pulse-Code modulation* (Modulação por código de pulso diferencial)

EBU *European Broadcasting Union* (União Europeia de Radiodifusão)

FFT *Fast Fourier Transform* (Transformada rápida de Fourier)

FIR *Finite impulse response* (Filtro de resposta ao impulso finita)

GPL *GNU General Public License* (Licença Pública Geral GNU)

GPU *Graphics Processing Unit* (Unidade de processamento gráfico)

HT *Hyper Threading Technology*

IIR *Infinite impulse response* (Filtro de resposta ao impulso infinita)

ITU *International Telecommunication Union* (União Internacional de Telecomunicações)

ITU-R *ITU Radiocommunication Sector* (União Internacional de Telecomunicações, Seção de Rádio)

Jack *JACK Audio Connection Kit* (Arcabouço de programação de áudio Jack)

LOD *Level Of Detail* (Nível de detalhe)

LTI *linear time-invariant system* (Sistema linear e invariante no tempo)

MOVs *Model Output Values* (Valores de saída do modelo)

NMR *Noise-to-Mask Ratio* (Relação ruído-mascaramento)

OASE *Objective Audio Signal Evaluation* (Avaliação objetiva de sinais de áudio)

ODG *Objective Difference Grade* (Grau de diferença objetivo)

PARSEC *Princeton Application Repository for Shared-Memory Computers* (Repositório de aplicações para computadores com memória compartilhada)

PCM *Pulse-Code Modulation* (Modulação por código de pulso)

PEAQ *Perceptual Evaluation of Audio Quality* (Avaliação perceptual da qualidade de áudio)

PERCEVAL *Perceptual Evaluation of the Quality of Audio Signals* (Avaliação perceptual de qualidade de sinais de áudio)

POSIX *Portable Operating System Interface* (Interface portável entre sistemas operacionais)

QoS *Quality of Service* (Qualidade de serviço)

RTP *Real-time Transport Protocol* (Protocolo de transmissão em rede em tempo real)

SDG *Subjective Difference Grade* (Grau de diferença subjetivo)

SIMD *Single Instruction Multiple Data* (Arquiteturas de instrução única sob dados múltiplos)

SoC *System on a Chip* (Sistema-em-um-chip)

SQNR *Signal-to-Quantization-error-Noise ratio* (Relação sinal-erro de quantização)

WFS *Wave Field Synthesis* (Síntese do campo sonoro)

WSD *Weighted Spectral Distortion* (Distorção espectral ponderada)

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

> [...] as análises [...] mostram que a história de um conceito não é, de forma alguma, a de seu refinamento progressivo, de sua racionalidade continuamente crescente, de seu gradiente de abstração, mas a de seus diversos campos de constituição e de validade, a de suas regras sucessivas de uso, a dos meios teóricos múltiplos em que foi realizada e concluída sua elaboração.
>
> (Foucault, 2012, p.5)

1.1 Problematização

Em sistemas de processamento de áudio podem acontecer – e acontecem – situações de sobrecarga computacional. Tais situações resultam em eventos sonoros insatisfatórios como interrupções intermitentes, artefatos sonoros ou mesmo em algo totalmente indesejável, como por exemplo a parada completa da execução do sistema.

O problema fundamental é que todos os sistemas computacionais possuem um limite de recursos de processamento, e assim tal situação de sobrecarga pode acontecer em quaisquer sistemas e plataformas. Embora a maioria dos sistemas considerados nesse trabalho não sejam sistemas de tempo real *stricto sensu*, ou seja, eles não possuem mecanismos que garantam a conclusão do processamento no tempo devido, o problema fundamental é a própria carga computacional do processamento de áudio quando esta excede a capacidade computacional disponível. A priorização de certas tarefas consideradas mais críticas poderia resolver o problema em relação a estas tarefas priorizadas, porém não garante a execução de todas as tarefas necessárias dentro dos prazos pressupostos pelo processamento de áudio em tempo real.

Uma opção para aliviar esse problema poderia ser o aumento dos recursos computacionais. Entretanto, por várias razões práticas, isso não é sempre possível, pois, por um lado, essa opção exige a disponibilidade de recursos financeiros, e, por outro lado, em muitos casos, ela exige também uma adaptação do ambiente de processamento e/ou até mesmo mudanças nos programas usados. Isso explica o interesse no estudo de sistemas que sejam capazes de administrar de forma mais eficiente os recursos computacionais que já estão disponíveis, através do controle e gerenciamento da carga computacional. Ainda que a abordagem proposta neste trabalho também exija modificações de software, ela não requer investimento em infra-estrutura para aumentar a capacidade

computacional.

Adicionalmente, no contexto do processamento de áudio coloca-se também a questão da qualidade do resultado sonoro, uma vez que essa qualidade está relacionada a fenômenos da percepção humana, exigindo avaliações complexas e frequentemente subjetivas. A opção de limitar o uso do sistema restringindo seu potencial para evitar as situações de sobrecarga, ainda que seja a opção escolhida pela maioria dos usuários, obviamente não resolve o problema porque reduz a utilidade do sistema e impacta a qualidade perceptual do resultado sonoro produzido. Nesse caso, a aplicação da metodologia aqui proposta representa um aumento do potencial de uso do sistema.

1.2 Problemas relacionados

O problema de recursos computacionais limitados e da necessidade de algum *trade-off* não se aplica apenas à área de processamento de áudio, mas também a muitas outras áreas da computação aplicada, e se acentua quando o processamento é executado sob condições de tempo real. Exemplos de sistemas de tempo real encontram-se principalmente na área de controle, quando um sistema computacional monitora o estado de um sistema controlado, como, por exemplo, um processo de produção industrial, e intervém para estabelecer o estado desejado desse processo. Sistemas de tempo real planejam o escalonamento de tarefas a executar, além de possuírem mecanismos para garantir os tempos de entrega dos resultados dessas tarefas. Para lidar com situações em que a demanda por recursos computacionais excede os recursos disponíveis, foram desenvolvidas técnicas de computação flexível (*flexible computation*) (Feng e Liu, 1997; Lin *et al.*, 1987).

Na transmissão de dados em rede ocorrem situações semelhantes. Redes de computadores são limitadas quanto à quantidade de dados que pode ser transmitida em um certo intervalo de tempo. Consequentemente, podem acontecer situações de sobrecarga, isto é, a demanda pelo recurso excede a capacidade da rede de transmitir os dados. O conceito de qualidade de serviço (*quality of service*) lida com essa situação, sendo usado no planejamento e na administração da rede, possibilitando a negociação entre pares (*peers*) para determinar uma qualidade de serviço em concordância com os recursos de rede disponíveis (Steinmetz e Nahrstedt, 1995). Além disso, existem vários protocolos de rede para administrar o recurso de forma mais eficiente e evitar situações de sobrecarga, como por exemplo o *Resource reservation protocol* (RSVP), para reservar recursos de rede, ou o *Internet group managment protocol* (IGMP), para configurar a distribuição de pacotes de dados.

Outro domínio onde se realiza um *trade-off* entre tempo de execução e espaço na memória é a compressão de dados de áudio. No esquema de compressão denominado *codificação perceptual*, tenta-se alcançar uma máxima compactação dos dados de áudio com o mínimo de perda de qualidade do ponto de vista perceptual (Painter e Spanias, 2000). Na fase de codificação, os dados são compactados (consumindo tempo de execução no processador) a fim de serem ou armazenados em menor espaço na memória ou transmitidos por uma rede de computadores economizando tanto largura de banda quanto tempo de transmissão. Na fase de decodificação, geralmente exigida a cada uso ou manipulação dos dados, se consome novamente tempo de processamento para a descompactação.

Não raro, estas situações aparecem combinadas em aplicações que permitem controlar o *trade-off* entre tempo de processamento e qualidade percebida, ou entre outras medidas relacionadas (por exemplo espaço de armazenamento e tempo de transmissão em rede). Através do uso de compressão de dados, recursos de rede são usados de forma mais eficiente, permitindo em certos casos aliviar a carga da rede e evitar situações de sobrecarga. Isso se aplica por exemplo à comunicação bidirecional por áudio e vídeo em tempo real, que tem diferentes graus de exigência em relação à latência ou à qualidade de áudio conforme a situação (videoconferência, concertos distribuídos, etc.). Sistemas operacionais interativos tratam este problema através de estratégias heurísticas de escalonamento das tarefas concorrentes, priorizando tarefas importantes em detrimento de outras.

Na área de espacialização existem sistemas de gerenciamento de recursos que controlam o *trade-off* entre a precisão da síntese e os custos computacionais (Herder, 1999a; Tsingos, 2005). Todavia, essas abordagens são limitadas às aplicações de espacialização, limitação essa que se pretende

superar, considerando que a proposta aqui desenvolvida é aplicável a uma grande variedade de software que inclui espacialização.

A pesquisa desenvolvida nesta tese possui caráter multidisciplinar, interconectando diversas áreas:

- processamento de sinais digitais
- acústica e psicoacústica
- sistemas operacionais
- estrutura de dados e análise de algoritmos
- otimização matemática
- otimização de software

A importância das três primeiras áreas já foi estabelecida nesta introdução. A consideração de estruturas de dados e da análise de algoritmos permite a modificação de algoritmos de processamento de áudio existentes para introduzir um controle do *trade-off* entre custos computacionais e qualidade perceptual. As duas últimas áreas inserem-se especificamente na flexibilização de algoritmos para processamento de áudio, através da otimização de coeficientes de filtros e da técnica de *autotuning*.

1.3 Metodologia e Objetivos

O objetivo principal deste trabalho consiste em investigar, discutir e propôr métodos capazes de controlar o *trade-off* entre os custos computacionais e a qualidade de áudio em diversas aplicações específicas de processamento de áudio em tempo real (flexibilização). Questões de pesquisa mais específicas são delineadas abaixo.

A metodologia desenvolvida para alcançar este objetivo parte da análise teórica dos algoritmos associados a tais aplicações, investigando a exequibilidade dos mecanismos de flexibilização para controle do *trade-off* , a fim de adaptar implementações e elementos de processamento (*plugins*) existentes da forma menos invasiva possível. Tais mecanismos estão associados à introdução de *parâmetros de flexibilização* específicos a cada implementação, sendo que os detalhes da metodologia proposta estão no Capítulo 3.

O contexto em que a metodologia proposta se insere é o das plataformas e aplicações de código aberto, por uma dupla razão. Por um lado, a disponibilidade dos códigos permite a intervenção necessária à flexibilização dos algoritmos que permite o controle do *trade-off* entre custos e qualidade. Por outro lado, as licenças abertas facilitam a difusão e ampla utilização da proposta, possibilitando eventuais readequações e desenvolvimentos futuros por outros pesquisadores e usuários.

As questões que orientam esta pesquisa são:

- É possível propôr um procedimento generalizado de flexibilização para algoritmos de processamento de áudio em tempo real?
- De que maneiras um sistema de processamento de áudio pode controlar o *trade-off* entre custos computacionais e qualidade percebida (do resultado do processamento) de forma flexível?
- Como escolher dentre diferentes métodos de gerenciamento do *trade-off* em função dos objetivos da aplicação?
- Como se pode quantificar a qualidade percebida em função dos parâmetros de flexibilização?
- Como mapear os custos computacionais de forma consistente em função dos parâmetros de flexibilização?

- Quais são as aplicações onde a abordagem proposta é mais eficaz? Em que casos um sistema desse tipo é capaz de resolver ou aliviar o problema de sobrecarga computacional? Até que ponto é possível minimizar a perda de qualidade percebida adaptando-se dinamicamente os custos computacionais aos recursos disponíveis?

Validação e limitações da Proposta Para a avaliação da metodologia proposta nessa pesquisa, procuramos determinar matematicamente e experimentalmente a diferença entre os resultados auditivos em cenários específicos de aplicação.

Para tanto, as etapas percorridas para a validação da metodologia são:

- estabelecimento de critérios de avaliação por revisão bibliográfica psicoacústica,
- análise e parametrização de elementos de processamento de áudio e desenvolvimento de métodos de controle dinâmico desses parâmetros,
- implementação de protótipos do sistema em plataformas comumente utilizadas em processamento de áudio, e
- avaliação experimental para determinar a qualidade percebida em configurações específicas.

A metodologia aqui proposta mostra-se adequada tanto para sistemas de processamento de áudio de pequeno porte, como por exemplo sistemas embarcados em aparelhos auditivos ou sistemas móveis, quanto também para computadores pessoais e aplicações como *digital audio workstations (DAW)* e sistemas análogos. O principal critério de aplicabilidade da proposta desta pesquisa é, no entanto, se a perda de qualidade percebida é admissível pelos usuários. Sistemas de processamento de áudio que dependam da exatidão da computação, como aplicações de medição, não são alvo da presente pesquisa, bem como estão excluídos sistemas que exijam parâmetros de processamento fixos. As limitações de aplicabilidade deste estudo estão detalhadas na Seção 3.4.

Adicionalmente, deve-se considerar: a) que os resultados de uma avaliação experimental da qualidade percebida dependem, entre outros fatores, da escolha das amostras sonoras utilizadas, o que limita uma caracterização generalizada, e b) certos casos de uso não permitem facilmente uma avaliação experimental por razões práticas, por exemplo, quando certos equipamentos não estão disponíveis para a pesquisa (como é o caso de sistemas de espacialização de grande porte para síntese de campo sonoro – *Wave Field Synthesis* (Síntese do campo sonoro) (WFS)). Apesar destas considerações restringirem a universalidade dos mecanismos desenvolvidos, inúmeras situações de interesse prático podem ser identificadas e abordadas pela metodologia proposta, como detalhado no Capítulo 3. Assim, considera-se possível avaliar e determinar a exequibilidade e a eficácia da solução proposta para casos de uso e formas de processamento específicos.

Por fim, é válido destacar que a complexidade da abordagem e da metodologia aqui propostas possui diferentes dimensões. Em primeiro lugar, deve ser entendido que as modificações propostas no software, correspondentes à introdução de parâmetros de flexibilização, se aplicam principalmente a unidades elementares de processamento. Isso significa que o problema principal – as situações de sobrecarga – não é resolvido globalmente de uma só vez, como ocorreria se uma solução fosse aplicada no nível mais baixo do sistema de software, de modo que todos os processamentos se beneficiassem automaticamente da modificação. Uma outra dimensão de complexidade resulta do caráter dinâmico do sistema de controle com retroalimentação, ou seja, do fato de que as medições de desempenho determinam as decisões sobre a modificação dos parâmetros do processamento flexível, que por sua vez mudam o desempenho da aplicação.

Reprodutibilidade Este trabalho almeja uma pesquisa científica aberta e reprodutível por qualquer pessoa interessada, o que torna necessário o uso de fontes abertas e de ferramentas livremente disponíveis; por isso esta pesquisa utiliza exclusivamente software livre e fontes sonoras públicas com licenças que admitem seu uso em pesquisas. Para certas tarefas, por exemplo a avaliação da qualidade percebida, esta determinação restringe a utilização de software comercial que poderia

ter vantagens em comparação com o software livre, como é o caso de certas implementações do método PEAQ (Seção 4.3.2). Por outro lado, esta decisão metodológica garante que os experimentos apresentados podem ser replicados independentemente, e a pesquisa facilmente continuada e estendida.

1.4 Contribuições

Os principais fundamentos da abordagem deste trabalho são aqueles que permitem um *trade-off* , pressupondo que: (a) os elementos, que formam em seu conjunto o processamento de áudio da aplicação, são parametrizáveis, de forma que existem parâmetros nos algoritmos usados que podem ser manipulados em tempo de execução, e que controlam o próprio tempo de execução do algoritmo; (b) existe um espaço perceptual de representação no qual o sinal pode ser modificado sem prejuízo perceptual para o usuário.

As principais contribuições deste trabalho são as seguintes:

- uma metodologia para o processamento flexível de áudio em tempo real que inclui o controle do *trade-off* entre os custos computacionais e a qualidade percebida, reduzindo ou evitando, significativamente, situações de sobrecarga, com a menor perda possível de qualidade do ponto de vista da percepção do usuário.
- a parametrização de diversos elementos de processamento de sinais de áudio visando o processamento flexível, incluindo a investigação da relação entre parâmetros de flexibilização, custos computacionais decorrentes e qualidade percebida do resultado.
- o desenvolvimento de métodos de controle dinâmico que coordenam em tempo real a manipulação dos parâmetros dos elementos de processamento de áudio, viabilizando a computação no prazo exigido com a mínima perda de qualidade;
- um método baseado em otimização linear para a parametrização e flexibilização de filtros de resposta finita ao impulso (*Finite impulse response* (Filtro de resposta ao impulso finita) (FIR)), bem como um método de avaliação dos resultados da flexibilização;
- um método de afinação automática (*autotuning*[1]) dos parâmetros de flexibilização em função dos custos computacionais e da qualidade percebida;
- implementação de um sistema de processamento de áudio flexibilizado (a aplicação `flexmix`), que demonstra a viabilidade da abordagem.

1.5 Organização do Trabalho

No Capítulo 2 são apresentados vários conceitos ligados a ambientes de processamento de sinais, sistemas de processamento em tempo real, percepção de qualidade e parametrização do processamento de sinais.

No Capítulo 3 é apresentada a metodologia proposta para abordar a flexibilização de aplicações de processamento de áudio em tempo real.

A aplicação experimental da metodologia em uma aplicação específica é apresentada no Capítulo 4.

No Capítulo 5 os resultados da pesquisa são confrontados com os objetivos inicialmente traçados, e caminhos de pesquisa futura são delineados.

O detalhamento das ferramentas utilizadas nesta pesquisa se encontra no Apêndice B, visando facilitar a reprodutibilidade da pesquisa. Além deste, o Apêndice A traz os detalhes matemáticos relacionados à otimização linear dos coeficientes de filtros de resposta finita ao impulso.

[1] Não deve ser confundido com o efeito de áudio que tem o mesmo nome para a correção de *pitch*; veja www.antarestech.com/product/auto-tune-unlimited

Capítulo 2

Conceitos

Neste capítulo serão revisados conceitos básicos quanto à representação de sinais de áudio em sistemas digitais e quanto à codificação perceptual, para mostrar como se introduz a ideia de erro, sua computação ou estimação e os conceitos de qualidade relacionados. No decorrer do capítulo será apresentada uma abordagem de como definir a qualidade de áudio a partir de testes de escuta, bem como da captura desse processo em algoritmos de previsão de qualidade. As aplicações de processamento de áudio investigados nesta pesquisa têm que produzir o sinal de saída sem interrupções, elas são submetidas ao regime de tempo real; decorrentemente, será apresentado o conceito teórico básico de tempo real na computação e, à luz de *trade-offs* entre o tempo de computação e a qualidade do resultado, o modelo da computação imprecisa. Os trabalhos revisados neste capítulo se relacionam com síntese ou processamento de áudio que lidam com tais *trade-offs* ou, ao menos, podem viabilizá-los por demonstrar potenciais flexibilizações, sejam no lado computacional ou perceptual da troca. Esta lista não exaustiva visa exemplificar a aplicação do nosso ponto de vista a trabalhos que podem ser adaptados ao contexto desta pesquisa. O capítulo é concluído com uma visão geral dos ambientes mais relevantes de processamento de áudio em tempo real que existem para sistemas Linux, uma vez que esse sistema foi usado como estudo de caso na parte experimental, e também porque as características dos ambientes determinam as escolhas disponíveis à abordagem prática, principalmente quanto ao gerenciamento de tarefas e do tempo.

2.1 Representação e codificação de áudio

Para o processamento de sinais de áudio, os sinais têm que ser capturados e transformados em uma representação digital. Normalmente esse processo de digitalização tem a seguinte forma:

1. captura (obtenção) do som, por exemplo, com um microfone que transforma a pressão das ondas sonoras no ar em um sinal elétrico;

2. filtragem do sinal por um filtro passa-baixas para remover partes do sinal incompatíveis com o próximo passo;

3. conversão analógico-digital do sinal.

O resultado desse processo é uma sequência de valores na memória do computador implicando em uma discretização no tempo e uma quantização dos valores medidos.

Por um lado, o sinal é discretizado no tempo, isto é, o conversor mede o sinal em intervalos constantes, como pode ser visto na Equação 2.1 (Oppenheim *et al.*, 1999):

$$\mathbf{x}[n] = x_c(nT), \tag{2.1}$$

onde x_c é o sinal de entrada contínuo, $\mathbf{x}$ é o vetor de valores das amostras e T é o intervalo em segundos entre amostras do sinal de entrada; o valor $f = 1/T$ é chamado taxa de amostragem, e é

medido em Hertz (Hz). Com isso, todos os valores de $x_c(t)$ entre os pontos amostrados (ou seja, para $t \in (nT, (n+1)\,T)$) são perdidos. Essa perda restringe a faixa de frequências representáveis pelo sinal discretizado; isso é expresso pelo teorema de Nyquist, o qual constata que para representar um sinal que contém componentes de frequência até x Hz é necessária uma taxa de amostragem de pelo menos $2x$ Hz (Moore, 1990). Por isso, no processo descrito acima, o filtro passa-baixas tem que eliminar componentes de frequências mais altas que pelo menos metade da taxa de amostragem.

Por outro lado, os valores medidos pelo conversor têm que ser convertidos em valores numéricos de precisão finita, ou seja, os valores medidos são mapeados em um conjunto fixo de valores de amplitude em um processo denominado quantização, sendo que cada amostra é quantizada para o valor mais próximo dentro do conjunto. Esse processo de quantização introduz um erro de amplitude (o erro de quantização) que é definido pela diferença entre os valores de entrada do quantizador e, os valores quantizados e depois dequantizados.

Uma medida objetiva de qualidade que quantifica o erro da digitalização de sinais de áudio é o *Signal-to-Quantization-error-Noise ratio* (Relação sinal-erro de quantização) (SQNR) definido pela relação 2.2 (Oppenheim *et al.*, 1999, pág. 122), em que *sinal* representa a amplitude pré-quantização, e *ruído* a diferença de amplitude do sinal decorrente da quantização:

$$\text{SQNR} \sim \frac{\text{variância do sinal } (\sigma_s^2)}{\text{variância do ruído } (\sigma_r^2)}. \tag{2.2}$$

Para, por exemplo, uma amplitude máxima de um sinal de U_{max} volts, que corresponde a uma faixa de valores de $-U_{max}$ até $+U_{max}$, e para uma quantização linear em valores de N bits, teremos faixas de quantização de largura igual, dada por

$$\frac{2U_{max}}{2^N} = \frac{U_{max}}{2^{N-1}}; \tag{2.3}$$

considerando uma distribuição uniforme do ruído, teríamos uma variância associada de

$$\sigma_r^2 = \frac{1}{12}\left(\frac{U_{max}}{2^{N-1}}\right)^2 = \frac{U_{max}^2}{3 \cdot 2^{2N}}, \tag{2.4}$$

e assim:

$$\begin{aligned} \text{SQNR} &= 10\log_{10}\left(\frac{\sigma_s^2}{\sigma_r^2}\right) \\ &= 10\log_{10}\left(\frac{3 \cdot 2^{2N}\sigma_s^2}{U_{max}^2}\right) \\ &= 6.02N + 4.77 - 20\log_{10}\left(\frac{U_{max}}{\sigma_s}\right), \end{aligned} \tag{2.5}$$

sendo σ_s o desvio padrão do sinal. A expressão acima permite observar que a relação U_{max}/σ_s pode ser vista como um *trade-off* entre o valor da amplitude máxima e o desvio padrão do sinal. Usando, por exemplo, um valor $\sigma_s = U_{max}/4$, que é razoável para sinais analógicos de música e fala (Oppenheim *et al.*, 1999), obtém-se:

$$\text{SQNR} \approx 6N - 7.25[\text{dB}]. \tag{2.6}$$

Assim, para uma gravação de um sinal analógico em um *Compact Disc* (Disco compacto a laser) (CD) de áudio, que é codificado em amostras de $N = 16$ bits, tem-se um SQNR de aproximadamente 90 dB. Porém, isso se aplica apenas nos casos em que a entrada de sinal é mapeada à faixa *inteira* de valores de amplitude disponíveis: sinais de amplitude menor terão SQNR menor, conforme a Equação 2.2.

Na quantização linear exemplificada acima, os valores quantizados produzidos pelo conversor analógico-digital são uniformemente espaçados de acordo com faixas de amplitude de largura igual.

Geralmente, as amostras quantizadas são codificadas em uma representação binária. Essa codificação é chamada modulação por código de pulsos (*Pulse-Code Modulation* (Modulação por código de pulso) (PCM)) pois cada amostra no sinal elétrico corresponderá a um padrão de pulsos retangulares. Normalmente a codificação PCM é utilizada em conjunto com a quantização linear. Uma das desvantagens desse processo é que a representação digital de amplitudes baixas é pobre, reflexo do baixo SQNR (veja a Equação 2.2 (Moore, 1990)).

Uma alternativa à quantização linear é a quantização em valores de ponto flutuante ou ainda em escala logarítmica (por exemplo, μ-law). No caso da codificação em ponto flutuante tem-se um SQNR igual em todas as faixas de amplitude correspondentes a um mesmo expoente. Além disso, a quantização em ponto flutuante permite uma faixa dinâmica[1] muito maior que a quantização linear.

Outra opção de representação das amostras é a modulação por código de pulsos diferencial (*Differential Pulse-Code modulation* (Modulação por código de pulso diferencial) (DPCM)). Considerando-se que a diferença entre a amplitude de duas amostras consecutivas seja pequena, a DPCM representa apenas o valor da diferença entre elas. Desta forma, elimina-se a redundância de valores próximos, permitindo uma codificação usando menos bits.

Ainda que os tipos de codificação discutidos acima sejam comumente chamados de codificações *sem perda*, o processo de captura-codificação implica necessariamente na inclusão de erros, que dependem (a) dos parâmetros do processo (resolução da quantização, frequência da captura), (b) das limitações e tolerância dos elementos eletrônicos envolvidos na amostragem.

Representação no domínio da frequência

Uma forma equivalente de representação de áudio, no caso analógico, é a transformada de Fourier do sinal. No caso do sinal digital, a transformada de Fourier discreta do sinal digital leva a uma representação equivalente no domínio da frequência. Nesse domínio, o sinal é representado por coleções (janelas) de valores chamadas *espectros*, onde cada valor ou *bin* representa a amplitude ou energia para uma certa frequência. A resolução espectral máxima é limitada pelo número de amostras do sinal incluídas em cada janela. Portanto, é necessário considerar que o janelamento dos valores de entrada introduz uma distorção e que, normalmente, esse janelamento é feito com algum tipo de sobreposição de janelas consecutivas.

A representação no domínio da frequência tem um papel importante na análise psicoacústica, já que certos efeitos psicoacústicos são eficientemente identificados nessa representação (veja a Seção 2.2). Decorrentemente, vários padrões de codificação utilizam esta representação.

2.2 Codificação perceptual

Nas representações de sinal previamente apresentadas, um sinal arbitrário, após codificado, não perde nenhuma parte de seu conteúdo. Contrariamente, a codificação perceptual pretende reduzir o número de informações contidas na representação ao mínimo necessário sob a condição que o sinal ressintetizado seja perceptualmente indistinguível do original. Esse tipo de compressão explora irrelevâncias perceptuais e redundâncias estatísticas (Painter e Spanias, 2000). Neste caso - diferentemente das semelhanças matemáticas acima descritas - o relevante é a semelhança do ponto de vista da percepção humana.

Usa-se a codificação perceptual (particularmente o formato *mp3*[2]) ubiquamente para o armazenamento e a transmissão de dados de áudio uma vez que a quantidade de dados necessária para a representação do sinal nesse formato é normalmente muito menor que a quantidade de dados necessária para codificações sem perda.

O processo genérico de codificação (Figura 2.1) contém uma parte de análise do sinal para estimar componentes temporais e espectrais, cujos resultados são usados na codificação e quantização

[1] Aqui, a faixa dinâmica pode ser entendida como a diferença entre a máxima e a mínima amplitudes representáveis.
[2] A denominação mais precisa é MPEG-2 Audio Layer III, definida na norma ISO/IEC 13818-3:1998

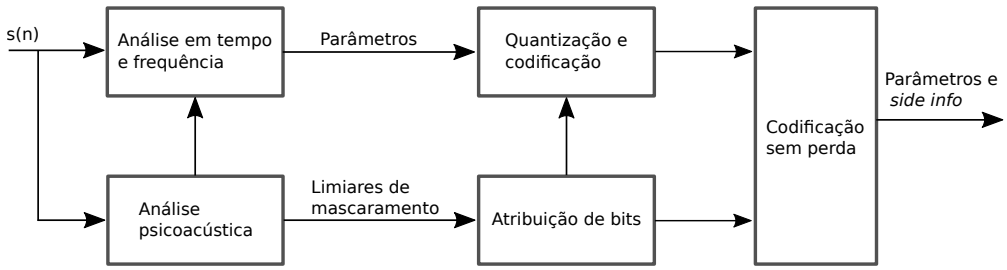

Figura 2.1: *Processo genérico de um codificador perceptual de áudio; fonte: (Painter e Spanias, 2000, p. 452).*

do sinal. Para identificar informações irrelevantes e redundâncias, vários princípios psicoacústicos são usados. A seguir, são descritos os mais importantes:

Limiar absoluto de audibilidade O limiar absoluto da audibilidade carateriza a energia mínima que um tom puro precisa ter para ser percebido por um ouvinte em um ambiente silencioso, e depende da frequência do tom. Portanto este limiar pode ser interpretado como a energia máxima das distorções, introduzidas pelo processo de codificação, que são admissíveis e toleráveis pela percepção. Todavia, os codificadores devem considerar que essas distorções aparecem em diversas faixas de frequência, e que o limiar da audibilidade varia em função da frequência.

Bandas críticas e mascaramento Um tom puro apresentado ao ouvido é capaz de mascarar ou tornar inaudível um ruído de faixa estreita, cuja amplitude é menor que a amplitude do tom puro. Esse fenômeno é chamado de mascaramento. O requisito para que o fenômeno aconteça é que a frequência do ruído esteja próxima à frequência do tom puro. Essa medida de proximidade, está associada a uma faixa de frequência chamada banda crítica, e sua largura depende da frequência do tom puro (Fletcher, 1940; Pohlmann, 2000). É importante ressaltar que o mascaramento interfere na definição do limiar absoluto de audibilidade.

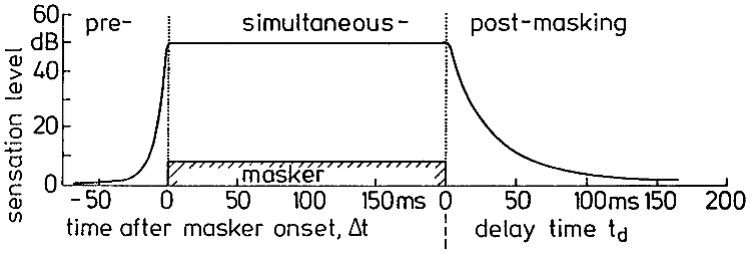

Figura 2.2: *Regiões temporais do pré-mascaramento, do mascaramento simultâneo e do pós-mascaramento; reproduzido de (Zwicker e Fastl, 2008, pág. 78).*

Mascaramento simultâneo Um mascaramento, portanto, ocorre quando mais de um estímulo é apresentado ao ouvido simultaneamente, resultando em algumas componentes inaudíveis. Convencionalmente, três tipos de mascaramento simultâneo são considerados: *noise masking tone*, quando um ruído de faixa estreita mascara um tom na mesma banda crítica (quando o volume desse é menor que um certo valor); *tone masking noise*, quando um tom mascara um ruído; e *noise masking noise*, quando um ruído mascara um outro ruído na mesma banda crítica. Além disso existem efeitos entre bandas críticas, ou seja, sinais mascaradores podem influenciar os limiares de audibilidade em outras bandas críticas próximas.

Mascaramento temporal Além do mascaramento simultâneo, outra modalidade de mascaramento foi observada, por exemplo, por Zwicker (Zwicker e Fastl, 2008), na qual os limiares de audibilidade para sons mascarados também crescem antes e depois da presença do sinal mascarador,

ou seja, podem ocorrer tanto pré-mascaramentos curtos e/ou pós-mascaramentos decrescentes com o tempo, como pode ser visto na Figura 2.2.

Entropia perceptual A entropia perceptual (*perceptual entropy*, medida em bits/amostra) representa um limiar teórico para compressibilidade de um sinal, e é usada para determinar o número de bits usados no codificador (veja a Figura 2.1) para quantizar o espectro do sinal. A entropia perceptual é estimada a partir do limiar de mascaramento (resultado da aplicação das regras de mascaramento ao espectro), e descreve a intensidade do ruído de quantização que pode ser introduzido sem ser percebido pelo ouvinte.

O resultado da codificação perceptual é uma representação do sinal mais compacta que a representação obtida por codificação sem perdas. Todavia, os limiares e mascaramentos resultantes de experimentos psicoacústicos podem ser variados, no intuito de obter representações com taxas de compressão diferentes, e portanto com qualidades diferentes.

2.3 Qualidade de áudio

Esta seção apresenta um revisão de métricas quantitativas da qualidade de áudio. O ser humano é exposto (e se expõe) a sons de fontes muito diversas, de ruídos ambientais de fontes naturais, de fontes técnicas ou da música e da arte. Apesar dessa diversidade, em muitos casos, ainda é possível formalizar a ideia de qualidade *boa* ou *ruim*. Essa formalização leva em consideração processos cognitivos e começaram a ser pesquisados quando a psicoacústica se formava no começo do século 20. Alguns exemplos a citar são o trabalho de Rayleigh sobre a localização de fontes (Rayleigh, 1907) e as pesquisas de Fletcher junto aos Laboratórios da Bell (*Bell Telephone Laboratories*) (Fletcher, 1929). Historicamente, a pesquisa em escuta (problemas, materiais e métodos) foi condicionada principalmente pelos interesses e necessidades da pesquisa telefônica, da terapia de fala e de áreas relacionadas (Sterne, 2012). Decorrentemente, problemas de avaliação de qualidade são abordados por conceitos como incômodo (*annoyance*), agradabilidade (*pleasantness*) e inteligibilidade no âmbito da aplicação da psicoacústica (Zwicker e Fastl, 2008).

Zwicker e Fastl afirmam que, no caso da avaliação de qualidade de som, efeitos estéticos e cognitivos podem ter um papel essencial e, por isso, nem todos fatores que contribuem para as medições psicoacústicas podem ser estimados (Zwicker e Fastl, 2008, pág. 327). Mesmo que a psicoacústica consiga descrever, por exemplo, avaliações de incômodo obtidas em experimentos com um descritor *incômodo psicoacústico* (*psychoacoustic annoyance*), em casos de sons nos quais efeitos estéticos e cognitivos predominam, como por exemplo na música, os descritores psicoacústicos perdem, ao menos parcialmente, sua força de descrição.

Enquanto na indústria de áudio e entretenimento, tradicionalmente, existe um conceito de uma *qualidade de som de produto* bom ou ruim (*product sound quality*), é relativamente nova a ideia da *quantificação* da qualidade percebida de processamento de áudio (Bech e Zacharov, 2006). A padronização de métricas de qualidade é desejável para vários fins, entre outros: a orientação em decisões empresariais, o reconhecimento e a comparabilidade de resultados e produtos, bem como a redução de custos. Consequentemente, órgãos relevantes, como, por exemplo, a *Audio Engineering Society* (Sociedade de engenharia de áudio) (AES), a *European Broadcasting Union* (União Europeia de Radiodifusão) (EBU) e a *International Telecommunication Union* (União Internacional de Telecomunicações) (ITU) desenvolveram, respectativamente, as recomendações e normas apropriadas, que definem métodos e modelos[3].

[3]por exemplo, o conjunto de recomendações *ITU Radiocommunication Sector* (União Internacional de Telecomunicações, Seção de Rádio) (ITU-R) relacionadas a avaliação perceptual de áudio para ouvintes *experts* para a banda larga. A ITU-R define, entre outros, um método para a mensuração de qualidade percebida de áudio (BS1387-1).

2.3.1 Teste de escuta

Os testes e métricas aqui descritos são derivados sem alteração do *teste objetivo* descrito na recomendação ITU-R.

Quantificação da impressão O teste de escuta segue o método científico experimental no qual se tenta descobrir influências de variáveis independentes sobre variáveis dependentes através de experimentos. No experimento da quantificação da impressão, o participante (*variável independente*) é exposto a um estímulo (*variável independente*) sob certas condições (*variáveis independentes*). Em seguida, o participante dá respostas (*variáveis dependentes*) às questões sobre sua percepção.

Essa transformação do universo analógico das percepções do participante em valores discerníveis e discretos na resposta é o fundamento da hipótese que certas propriedades de áudio sejam quantificáveis.

O atributo da resposta (*response attribute*) representa o tipo de resposta que se relaciona a uma propriedade percebida do áudio que será quantificada, como por exemplo, a *loudness*, o *pitch* ou a posição espacial. O formato da resposta (*response format*) inclui uma escala e um método que o participante deve usar para relatar a impressão.

Por ser objetivo, quantificável, generalizável e facilmente comunicável, um método padrão é *atribuir números a impressões*. No dimensionamento direto, o participante converte a sensação diretamente em uma magnitude e relata-a. Em contraste, o dimensionamento indireto pede que o participante tente discriminar entre estímulos.

Em geral, os participantes devem conhecer o atributo e o formato da resposta, isto é, devem conhecer o vocabulário consensual para falar sobre os estímulos. Se o mesmo não existir - no idioma em questão - tem que ser desenvolvido, normalmente em processos de grupo. Esse processo de levantamento inclui a identificação das propriedades a partir de estímulos representativos, a definição e descrição das propriedades e suas escalas, e um teste de consistência. O vocabulário consensual tem que ter as seguintes propriedades: exatidão e não ambiguidade das descrições, bem como a capacidade de gerar consenso entre participantes do teste.

Dentro desse padrão pode ser esperado que testes de escuta, quanto à qualidade percebida de estímulos auditivos, sejam capazes de:

- identificar que dois estímulos são perceptualmente iguais;
- identificar que um estímulo é igual, superior ou inferior a um outro estímulo;
- determinar o grau de diferença entre estímulos;
- determinar preferências entre sistemas de áudio.

A partir desse tipo e espectro de respostas podem ser estabelecidas escalas correspondentes, como, por exemplo, as escalas de mensuração de Steven (Nunnally e Bernstein, 1994):

- *nominal* permite identificação e contagem;
- *ordinal* permite estabelecer uma ordem monotônica;
- *intervalada* permite uso de distâncias e médias;
- *racional* permite uso de quocientes de valores.

Porém, testes de escuta não permitem obter respostas diretas a questões como:

- localizar parâmetros problemáticos em algoritmos de processamento, ou
- identificar métodos para melhorar a qualidade de áudio percebida.

contração	participantes subestimam grandes diferenças e sobrestimam pequenas diferenças
contração sequencial	o estímulo anterior tem influência na percepção do estímulo seguinte
magnitudes de diferença pouco familiares	comparar, por exemplo, o incômodo auditivo de um rato com o de um avião
escalas	por não usar a escala inteira, respostas são aglomeradas numa parte da escala
esperança	esperança pela influência de outras modalidades envolvidas (*cross modality*), por exemplo, influências visuais ou táteis

Tabela 2.1: *Principais vieses para testes de escuta.*

Configuração do experimento No experimento usa-se participantes de teste para a avaliação objetiva de qualidade de um sistema de (re)produção. A configuração do experimento deve garantir que a variação das variáveis dependentes seja causada pelas variáveis independentes e não por variáveis desconhecidas, que não são controláveis. Quanto às variáveis conhecidas, destacam-se: (a) o sistema de reprodução (o sistema em teste), (b) os participantes de teste, (c) a situação de escuta (inclusive o procedimento do experimento) e, (d) o sinal.

O participante de teste é o instrumento de medição. A seleção dos participantes de teste se baseia nas características dos mesmos para possibilitar a generalidade das conclusões do experimento. Uma seleção randomizada tem que ser planejada e restringida para garantir a representatividade. A ITU recomenda as seguintes categorias de participantes de teste para a classificação dos resultados: participante não treinado, participante experiente e *expert*, que se diferenciam pelo nível de familiarização e treinamento dos participantes em relação a métricas e formalizações relacionadas ao som.

Como os participantes do teste possuem certos vieses, a situação de escuta e, especialmente, o procedimento devem ser construídos de maneira que os efeitos desses vieses sejam minimizados ou excluídos. A tabela 2.1 apresenta os principais vieses neste contexto (veja, por exemplo, (Bech e Zacharov, 2006, pág. 86) para uma discussão de testes de escuta ou (Bonneel *et al.*, 2010) sobre a percepção bimodal de material áudio-visual, que demonstra a forte influência de propriedades visuais sobre a percepção auditiva).

O sinal usado nos testes pode ser qualquer sinal, bastando que excite as diferenças perceptuais em questão, com base no atributo da resposta. Além disso, o sinal tem que ser apropriado em suas características temporais, espectrais e espaciais, para evitar outros possíveis vieses.

2.3.2 Modelos preditivos de percepção

Modelos preditivos de percepção podem ser usados para prever uma característica particular, como por exemplo, o *loudness* ou um desempenho geral (como a qualidade percebida), porém somente aplicáveis em domínios definidos tais como a telefonia (padronizado na ITU-T P.862), ou a localização espacial em uma dimensão (como permite, por exemplo, a Auditory Modeling Toolbox[4] (Egger, 2013; Piotr *et al.*, 2014)). A grande vantagem desses modelos é que podem simplificar, bem como acelerar testes de escuta, pois, em certos casos, podem substituir participantes de teste por programas de computador. Porém, os modelos preditivos de percepção se baseiam em resultados obtidos por experimentos anteriores e, por isso, têm validade apenas dentro dos mesmos parâmetros que definiram estes experimentos. Apresentamos aqui um algoritmo baseado em um modelo preditivo que foi padronizado e será usado na fase de experimentação desta pesquisa.

[4] amtoolbox.sourceforge.net

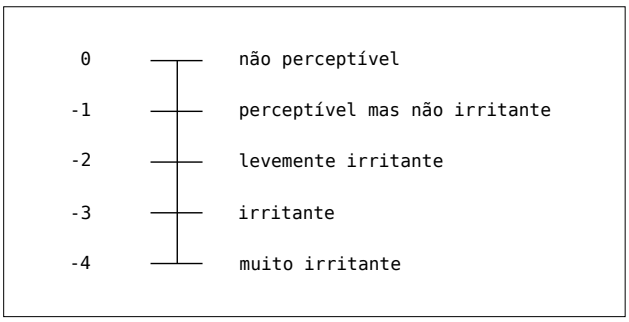

Figura 2.3: *A escala de avaliação ITU-R do Objective Difference Grade (Grau de diferença objetivo) (ODG).*

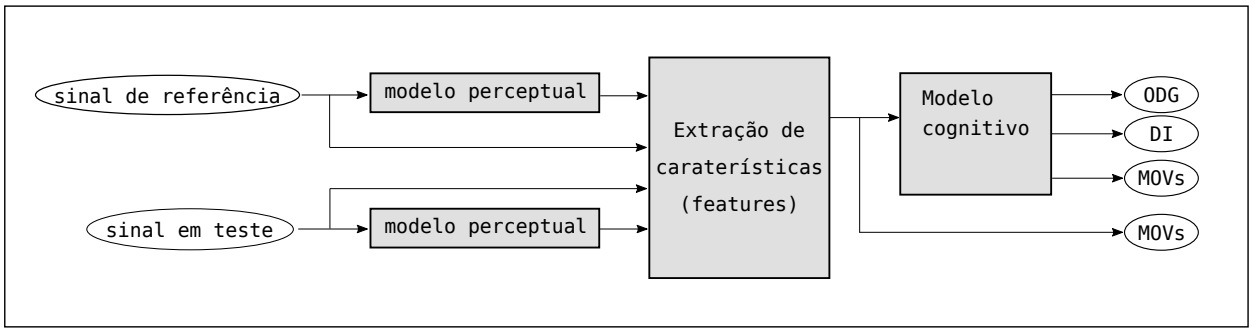

Figura 2.4: *Estrutura principal do método de medição PEAQ.*

Algoritmo *Perceptual Evaluation of Audio Quality* (Avaliação perceptual da qualidade de áudio) (PEAQ) Como método para a mensuração objetiva de percepção de qualidade de áudio, o PEAQ foi definido na recomendação ITU-R BS.1387-1 tornando viável a mensuração automática ou computacional. Como entrada, o método recebe o sinal original (sinal de referência) e uma outra versão do sinal (sinal de teste) e produz como saída: (a) o grau objetivo de diferença (ODG), (b) o índice de distorção (*Distortion Index* (Índice de distorção) (DI)), (c) os valores de saída do modelo (*Model Output Values* (Valores de saída do modelo) (MOVs)). A escala do ODG, como definida pela ITU-R, tem valores de -4 para diferenças muito irritantes a 0 indicando que a diferença não é perceptível (veja a Figura 2.3). Como outros modelos preditivos, o Algoritmo PEAQ contém um modelo do sistema auditivo acoplado a um modelo cognitivo para representar a percepção. Contrariamente a outros algoritmos que usam, por exemplo, métricas de distorção (harmônica), o PEAQ não compara sinais diretamente. O algoritmo de PEAQ estima os padrões de excitação no ouvido, para depois considerar os efeitos psicoacústicos (veja a Seção 2.2), e usa esses resultados para extrair vários parâmetros de qualidade. Esses parâmetros são combinados para produzir o grau de diferença (ODG) e o índice de distorção (DI) usando uma rede neural. Os parâmetros internos de qualidade se baseiam em parâmetros de qualidade como *Disturbance Index* (Índice de perturbação) (DIX), *Noise-to-Mask Ratio* (Relação ruído-mascaramento) (NMR), *Perceptual Evaluation of the Quality of Audio Signals* (Avaliação perceptual de qualidade de sinais de áudio) (PERCEVAL) e *Objective Audio Signal Evaluation* (Avaliação objetiva de sinais de áudio) (OASE) (Bitto *et al.*, 1999). Essa estrutura pode ser esquematizada como na Figura 2.4.

A recomendação (ITU-R) previa duas versões, uma versão básica com custos computacionais menores e uma versão avançada mais precisa, porém requerendo mais tempo de processamento. Além da análise de Fourier (*Fast Fourier Transform* (Transformada rápida de Fourier) (FFT)) que a versão básica utiliza exclusivamente, a versão avançada do algoritmo usa um modelo perceptual baseado em filtros. Uma descrição detalhada desses filtros encontra-se em (Bitto *et al.*, 1999).

O ODG deve corresponder ao grau de diferença subjetivo (*Subjective Difference Grade* (Grau de diferença subjetivo) (SDG)) obtido pelo teste de escuta (definido na ITU-R BS.1116), porém tal correspondência não é possível em função do espalhamento dos resultados de testes subjetivos (Bitto *et al.*, 1999, pág. 33). Todavia, testes que compararam resultados do PEAQ com

resultados de testes de escuta, por exemplo Bitto et al. (Bitto *et al.*, 1999) ou Zaunschirn et al. (Zaunschirn *et al.*, 2014), mostram que as predições do PEAQ são úteis, ao menos para o estado de desenvolvimento de aplicações ou *codecs*, e que os resultados do PEAQ são superiores aos resultados obtidos por métodos mais antigos. Zaunschirn et al. afirmam ainda que as predições do PEAQ devem ser interpretados em relação a outras predições da mesma amostra de áudio de referência (Zaunschirn *et al.*, 2014). Por estes motivos, o PEAQ será utilizado com ferramenta para comparações da qualidade percebida entre diferentes versões de processamento de áudio com os mesmos dados de entrada (sinal, parâmetros etc.).

2.4 Tempo real e Computação imprecisa

O sinal de saída de uma rotina computacional de processamento de áudio - sob a condição de tempo real (principalmente em aplicações *ao vivo*) - corresponde a uma sequência de amostras que, afinal, são enviadas para um conversor digital-analógico, que por sua vez tem como saída um sinal analógico que pode ser amplificado e reproduzido através de alto-falantes. Para que esse procedimento acoplado do processamento e da reprodução funcione, o sistema deve gerar, de forma ininterrupta, uma certa quantidade de amostras por segundo. Em caso de falha, o conversor digital-analógico iria gerar valores incorretos na conversão, causando um efeito audível normalmente indesejado. Dentro dessa perspectiva, um sistema de processamento de áudio é considerado um sistema de tempo-real. Neste trabalho não consideraremos sistemas que trabalham no paradigma de processamento em lote (*batch*), ou outros sistemas considerados *offline*, porque o resultado do processamento de áudio é armazenado para reprodução posterior e a condição de tempo real não é necessária.

Um sistema de tempo real é um sistema que garante que os resultados são produzidos até um certo instante limite: o prazo da entrega (*deadline*). Os sistemas de processamento de áudio podem ser descritos com o modelo de tarefa periódica (*periodic task model*) (Liu, 2000); a tarefa de produzir amostras é executada periodicamente para obter uma sequência com uma taxa de amostras por segundo definida. Taxas comuns são, por exemplo, a de 44100 amostras por segundo, usada para CDs, e a de 48000 amostras por segundo, encontrada em equipamentos de vídeo digital. Em estúdios usa-se também taxas maiores, como por exemplo a de 96000 amostras por segundo. Quase todos os sistemas trabalham com blocos de amostras referentes a períodos menores do que 1 segundo, mas isso não muda a descrição global do modelo de tarefa periódica. Majoritariamente, o software de processamento de áudio é composto por módulos, sendo que cada um computa um resultado parcial que depois é processado por outros módulos até chegar ao resultado final. O sistema tem que escalonar as subtarefas na ordem correta, considerando o fluxo de dados, e cumprir os prazos de entrega. Um esquema desse processo pode ser visto na Figura 2.5, onde t_1, t_2, t_3 são as subtarefas executadas periodicamente que produzem um resultado final (a_1, a_2, a_3) até o prazo de entrega. Para evitar falhas, a soma dos tempos de execução (incluindo a transferência de dados entre módulos) não pode ser maior que o intervalo entre os prazos de entrega. A inequação 2.7 mostra esse requisito:

$$t_{block} > \sum_{i=1}^{M} t_i, \tag{2.7}$$

onde t_{block} é a duração dos blocos de amostras[5] a serem computados, M o número de módulos e t_i o tempo de execução do i-ésimo módulo para processar um bloco de dados. Enquanto essa condição for satisfeita, um plano de execução em tempo real é factível.

Para o escalonamento em sistemas de tempo real existem diferentes abordagens, como a abordagem orientada ao relógio (*clock-driven approach*) ou a abordagem orientada a prioridades (*priority-driven approach*). Na abordagem orientada ao relógio, todos os tempos de iniciação das tarefas são previamente planejados a partir do conhecimento dos tempos de execução e das condições de pre-

[5]Nesse caso, a *duração do bloco* significa a duração da peça de áudio que um bloco de amostras representa; tipicamente $t = n/r$ onde n é o número de amostras e r é a taxa de amostragem.

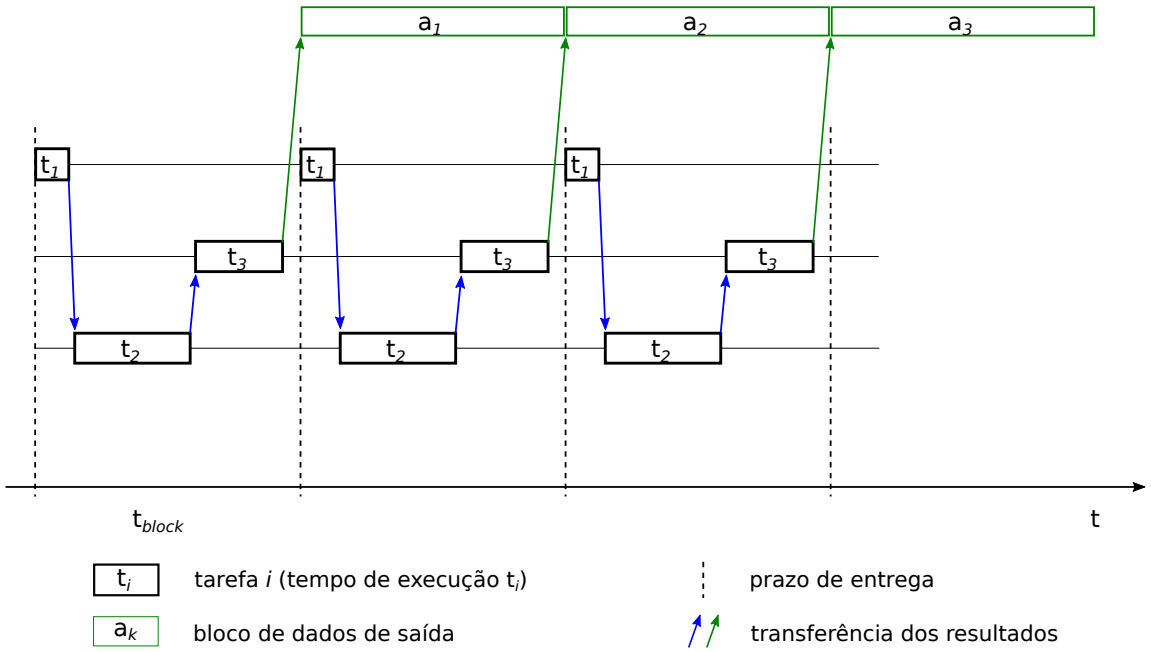

Figura 2.5: *Modelo de tarefa periódica.*

cedência. Isso não se aplica ao caso dos sistemas interativos porque, nesse caso, os tempos de execução, o número de tarefas e as dependências entre as mesmas mudam dinamicamente. Além disso, deve ser considerado que muitos sistemas de processamento de áudio, de interesse na presente pesquisa, rodam sobre sistemas operacionais que não são propriamente sistemas de tempo real.

A estratégia desses sistemas, como Linux, Windows NT e OS X, para emular propriedades de sistemas de tempo real é tentar escalonar os devidos processos o mais rapidamente possível. O escalonador nesses sistemas usa filas de prioridade para decidir, em certos intervalos, a sequência de processos ou tarefas a serem executados (Tanenbaum, 2009, pág. 748, 871). A alta prioridade designada aos processos de áudio de tempo real garante apenas que esses processos serão executados no momento mais próximo possível, porém o sistema operacional não pode garantir os tempos de entrega e, consequentemente, não pode prevenir situações de sobrecarga quando a condição da inequação 2.7 é violada (Koch, 2008). Estes sistemas operacionais tentam mitigar o problema pela implementação da norma POSIX.1b[6] que requer a introdução de novas políticas de escalonamento permitindo uma assim chamada *soft realtime* (Kerrisk, 2010). Contudo, esta estratégia oferece apenas uma *esperança* de sucesso no processamento correto a tempo.

Para processos críticos, a interferência de operadores é necessária para reconciliar (balancear) os recursos requisitados com os recursos disponíveis. Isso não se aplica apenas ao caso de tempo real; esta conciliação já começa na aquisição do hardware e na seleção dos componentes de software necessários. Dado o hardware e software, o tamanho do problema tem que ser ajustado para garantir o processamento suficientemente correto no contexto da produção, sendo essa garantia o respeito aos tempos de entrega das tarefas (saída de som ininterrupta) na grande maioria dos casos.

Computação imprecisa O termo computação flexível, ou computação imprecisa, refere-se a uma classe de aplicações que são concebidas e implementadas para realizar um *trade-off* entre o tempo de execução e a qualidade dos resultados (Lin *et al.*, 1987). Essa abordagem de computação flexível em problemas de processamento em tempo real parte da observação que, em certos casos, um resultado com uma qualidade menor - no momento devido - é melhor que o resultado exato, porém atrasado.

Os métodos de implementação que Liu (Liu, 2000) apresenta têm em comum a ideia de que

[6] Versão atual IEEE Std 1003.1-2017 contém as funções para o tratamento de *soft realtime* pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_08

cada tarefa crítica é dividida em uma componente obrigatória e uma componente opcional. Quanto às propriedades das componentes e à estratégia de escalonamento, Liu diferencia três métodos:

1. Quando o resultado do processamento da componente opcional pode ser desconsiderado, essa componente é chamada de "peneira" (*sieve*). Exemplos desse tipo de computação encontram-se, por exemplo, na área de codificação de vídeo em *MPEG*, onde a componente obrigatória produz I-frames e a componente opcional computa *B-* e *P-frames*, que são descartáveis. Quando a situação de carga do sistema é crítica, os *B-* e *P-frames* não são computados, o que leva a uma menor qualidade do vídeo decodificado.

2. No caso em que a componente opcional melhora monotonicamente a qualidade do resultado e pode ser interrompida a cada momento, caracteriza-se o método em etapas (*milestone*). Essa estratégia é utilizada por exemplo em esquemas de codificação progressiva de imagens, onde componentes de frequências baixas são renderizados antes daqueles correspondentes de frequências altas. Esse método pressupõe que a qualidade do resultado é melhorada gradualmente e converge, com o tempo, ao resultado desejado. Dessa maneira, o resultado obtido é sempre melhor quando a componente opcional é tardiamente interrompida e o tempo disponível pode ser usado de forma máxima.

3. No método de versões múltiplas (*multiple version*), usa-se implementações diferentes para a mesma tarefa. Cada versão corresponde a imprecisões diferentes tendo tempos de execução diferentes. Uma versão primária produz o resultado preciso mas está relacionada a um tempo de execução maior que as demais versões, que introduzem resultados inexatos requerendo tempos de execução menores. O escalonador toma a decisão sobre qual versão será usada a partir da disponibilidade dos recursos computacionais visando seu uso de forma ótima.

Decorrentemente, define-se o critério de otimalidade como (Liu, 2000): (a) completar todas as tarefas obrigatórias em tempo, e (b) maximizar a qualidade do resultado. O erro do resultado é definido a partir da distância entre o resultado impreciso e o resultado correto. O erro médio se calcula como a soma dos erros médios de cada tarefa que dependem do tempo de execução concedido a uma tarefa:

$$E_m = \sum_{i=1}^{M} E_i(x_i), \tag{2.8}$$

onde E_i é a média da soma dos erros por ciclo da tarefa i em função do tempo de processamento x_i concedido à tarefa. Supondo linearidade das funções de erro, é possível obter, de modo *off-line*, um cronograma estático ótimo. Nos casos em que as funções de erro sejam não-lineares, o problema a se resolver pode se tornar NP-difícil. Na Figura 2.6 pode se ver um cronograma do modelo de computação imprecisa, com componentes opcionais. A tarefa 2 contem uma parte obrigatória (t_2) e uma parte opcional (o_2) do tipo *milestone* cuja execução pode ser interrupta; a tarefa 3 contém uma parte obrigatória (t_3) e uma parte opcional do tipo peneira (o_3) que pode ser pulada. O cronograma não resulta na qualidade melhor possível da saída, uma vez que se a componente o_3 não for executada, restaria um tempo sem computação que poderia ser usado para a tarefa o_2. Com mais tempo de computação, a tarefa o_2 pode produzir uma saída melhor. Uma outra opção de otimização possível é obtida quando o tempo de computação da tarefa o_2 for encurtado, deixando tempo para a execução da tarefa opcional o_3. A decisão entre as opções depende do grau de imprecisão da tarefa o_2, do grau de imprecisão da não-execução da tarefa o_3 e da imprecisão introduzida pela propagação de erro da tarefa 2 à tarefa 3.

Para a estimação dos erros, Liu et al. (Liu *et al.*, 1987) propõem uma função que considera o quociente entre o tempo de execução concedido (t_i) e o tempo da execução completa da tarefa (T_i):

$$E_i = 1 - \frac{t_i}{T_i}. \tag{2.9}$$

Fouad et al. (Fouad *et al.*, 2001) usam um fator de ponderação (w_i) para expressar a importância

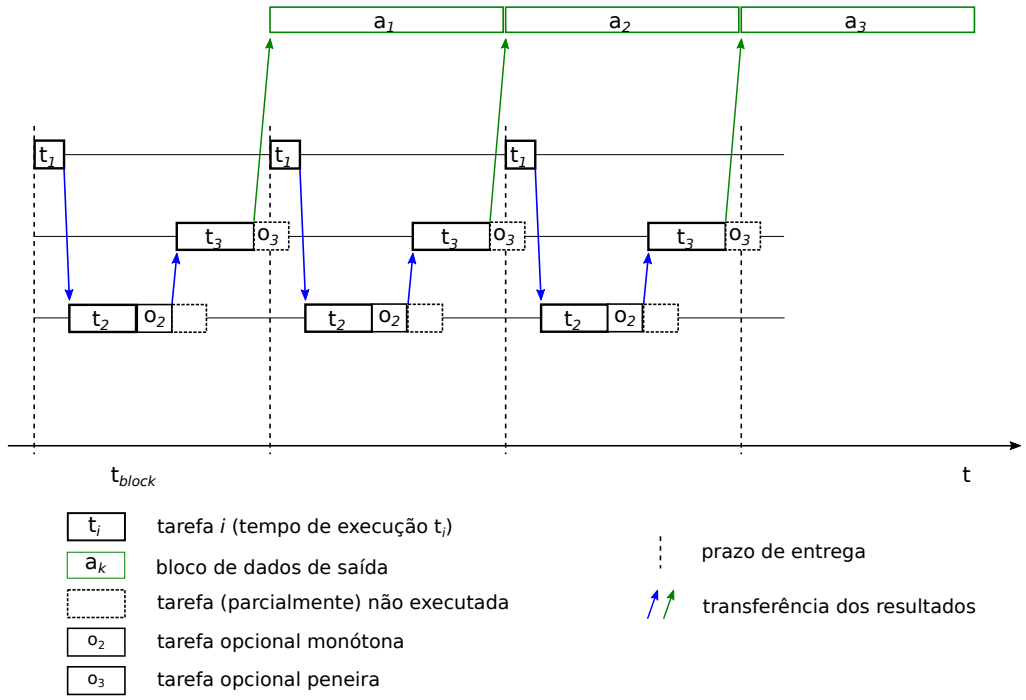

Figura 2.6: *Computação imprecisa: modelo de tarefa periódica com tarefas opcionais.*

de certas tarefas e forçar sua prioridade em uma aplicação de áudio:

$$E_m = \sum_{i=1}^{M} w_i E_i(x_i). \tag{2.10}$$

Uma abordagem que considera a propagação de erro em cadeias de tarefas dependentes é proposta por Feng et. al. (Feng e Liu, 1997). Em casos nos quais a imprecisão produzida por uma tarefa não depende apenas do tempo de execução concedido a ela, mas também da qualidade dos dados de entrada, a decisão sobre a atribuição de tempos de execução não pode considerar cada tarefa isolada.

Os autores introduzem um novo tipo de tarefas obrigatórias destinadas a corrigir o erro produzido pela tarefa anterior. Essas tarefas são executadas antes da tarefa obrigatória e seu tempo de execução depende do tamanho do erro dos dados de entrada. Em experimentos, os autores mostram que, nos casos em que os custos de recuperação são altos, as soluções melhores (melhor qualidade) contêm predominantemente componentes opcionais ao invés de componentes de correção, demonstrando que é mais eficiente priorizar tarefas opcionais do que tarefas de recuperação.

A abordagem é baseada na observação que o erro de uma tarefa depende da imprecisão do sinal da entrada e da imprecisão introduzido pela própria tarefa, como expressado na seguinte fórmula:

$$E_i = f_i(\phi_i, E_{i-1}), \tag{2.11}$$

onde E_i é o erro de computação da i-ésima tarefa e f_i é a função de erro e ϕ_i é o tempo de execução concedido para o elemento. Os autores mostram também que dentro destas soluções, as heurísticas de escalonamento que têm melhor desempenho são aquelas que consideram o *error-scaling factor* das componentes obrigatórias e das componentes opcionais da tarefa. Esse erro pode ser entendido como uma medida que quantifica a ampliação do erro de entrada pela tarefa e depende to tempo de execução concedido à tarefa. Os autores estimam funções de escala de erro *error-scaling factor* para cada tarefa pela influência ao resultado final deixando as outras tarefas inalterados. Assim transformam Equação 2.11 obtendo uma forma simplificada:

$$E_i = e_i(\phi_i)E_{i-1}, \tag{2.12}$$

onde e_i é a função dos fatores de escala de erro. Os autores mostram que a linearização das funções dos fatores de escala de erro (e_i) permitem a transformação e a simplificação do problema de planejamento. Os propostos algoritmos de planejamento conseguem minimizar o erro de cada elemento em tempo linear ($\mathcal{O}(n)$).

2.5 Trabalhos relacionados

A fim de melhor descrever o problema de flexibilização é relevante revisar não apenas trabalhos diretamente relacionados à solução, mas também pesquisas referentes a algoritmos de síntese e processamento de áudio. Em particular, apresentamos uma revisão não exaustiva de pesquisas na área de otimização que possam contribuir a uma solução flexível para o processamento de áudio em tempo real.

2.5.1 Processamento em dispositivos de capacidade computacional baixa

Este tema foi investigado por Bianchi (Bianchi, 2013), que sistematizou as limitações e possibilidades dos referidos dispositivos. A partir disso, o autor identifica um subespaço de parâmetros que sejam passíveis de modificação. Uma vez que estes parâmetros são identificados, uma análise teórica permite estabelecer o intervalo de valores viável para cada um destes parâmetros.

O autor implementa os algoritmos: (a) FFT, (b) convolução no domínio do tempo e da frequência, (c) síntese aditiva, e (d) *Phase Vocoder*[7], e identifica, correspondentemente, os seguintes parâmetros comuns a todos os algoritmos:

- o tamanho do bloco processado em cada passo (o número de amostras N);
- o formato dos dados processados[8] que acelera ou desacelera cada operação por um fator constante;
- implementações simplificadas da multiplicação (usando multiplicação e divisão com inteiros ou *bit-shifting*) com resultados menos precisos que acelera cada multiplicação por um fator constante.

A tabela 2.2 apresenta os algoritmos, junto a seus respectivos parâmetros específicos e seus custos computacionais na notação $\mathcal{O}$, quando aplicável, onde N representa o número de amostras, M o número de coeficientes e A o número de osciladores usados na síntese aditiva.

FFT	número de amostras	$\mathcal{O}(N \log N)$
convolução (domínio do tempo)	número de amostras, número de coeficientes	$\mathcal{O}(NM)$
convolução (domínio da frequência)	número de amostras	$\mathcal{O}(N \log N)$
número de amostras, síntese aditiva	número de osciladores, versões diferentes da implementação da interpolação na consulta de tabela	$\mathcal{O}(AN)$
Phase vocoder	parâmetros da FFT e da síntese aditiva	

Tabela 2.2: *Algoritmos pesquisados por Bianchi e possíveis parametrizações.*

Mesmo que o autor não inclua uma analise teórica mais detalhada dos custos computacionais, nos resultados dos testes ele demonstra a influência de cada um destes parâmetros ao desempenho e estima valores viáveis para cada dispositivo investigado. Além disso, o trabalho mostra como a

[7] O *Phase Vocoder* é uma técnica para representar um sinal através de um conjunto de osciladores cujas amplitudes e frequências instantâneas variam com o tempo (Dolson, 1986).

[8] Neste contexto, como formato de dados entende-se o espaço ocupado por uma amostra do sinal em bits e o tipo de codificação como, por exemplo, inteiro ou ponto flutuante.

parametrização pode ser induzida por uma investigação mais aprofundada em operações básicas como a multiplicação.

2.5.2 Perfuração de código

Sidiroglou et al. propuseram um método generalizado de perfuração de código (*code perforation*) para possibilitar a troca entre precisão (qualidade) e desempenho de um sistema de processamento (Sidiroglou *et al.*, 2011). A ideia principal é pular iterações de laços contidos no código produzindo resultados aproximados. Dado um laço adequado, o número de iterações é modificado para omitir computações usando três opções possíveis: (a) módulo: omitir cada n-ésima iteração; (b) truncar: omitir iterações no começo ou fim do laço; (c) aleatório: omitir iterações de forma aleatória a uma taxa média pré-estabelecida.

Os autores identificaram os seguintes padrões globais de laços que podem ser perfurados com perdas pequenas:

- enumeração do espaço de busca (pulos na busca);
- métrica de buscas;
- simulações do tipo Monte Carlo (diminuir o número de amostras);
- melhoria gradual;
- atualização de estrutura de dados (pular algumas atualizações).

O método é destinado a ser usado para: (a) melhorar o desempenho do sistema ou da aplicação; (b) economizar energia; (c) adaptar a outros contextos ou plataformas computacionais, (d) adaptar o desempenho de uma aplicação dinamicamente durante a execução; (e) melhorar a compreensão da aplicação pelos desenvolvedores, para melhoramentos futuros.

Os autores apresentam dois algoritmos para modificar código-fonte automaticamente produzindo sistematicamente versões perfuradas com o objetivo de explorar o espaço do *trade-off* entre desempenho e precisão. Em relatório técnico (Hoffmann *et al.*, 2009), o mesmo grupo de pesquisadores apresenta um compilador *perfurativo* usado para otimizar estaticamente o código de uma aplicação no tempo de compilação. Para isso, o compilador usa dados de treino para explorar o *trade-off* entre desempenho e acurácia do código perfurado. A métrica de precisão proposta computa primeiro uma abstração (descritor) do resultado original e do resultado da aplicação com perfuração, e, em um segundo passo, compara esses resultados através de uma diferença atenuada (cálculo da precisão / distorção). Esta métrica depende do resultado original e do resultado da execução perfurada que faz necessário executar o algoritmo tanto na forma original quanto na forma perfurada. Isso impede a computação da métrica em tempo real.

Adicionalmente, implementam um sistema (*runtime system*) que opera ao mesmo tempo que a aplicação principal e adapta a execução da aplicação principal no tempo de execução. Esse sistema usa os perfis de desempenho e acurácia gerados pelo compilador para gerenciar o grau de perfuração e assim a qualidade do resultado e o desempenho da aplicação. Os autores mostram em experimentos que o sistema facilita a adaptação do conceito em ambientes reais com falhas e escassez de recursos em tempo real.

Os cenários de aplicação pesquisados e apresentados pelos autores foram: codificação de vídeo, rastreamento de pessoas em vídeo, análise financeira, busca de imagens por similaridade e algoritmos do conjunto de *benchmarks* PARSEC[9]. Os autores mostram que é factível melhorar o desempenho dos algoritmos testados por fatores de 2 até 7, enquanto a precisão reduz menos que 10%.

[9]Repositório *Princeton Application Repository for Shared-Memory Computers* (Repositório de aplicações para computadores com memória compartilhada) (PARSEC): parsec.cs.princeton.edu

2.5.3 Nível de detalhe

Nível de detalhe (*Level Of Detail* (Nível de detalhe) (LOD)) é uma disciplina cujo objetivo é conciliar a complexidade e o desempenho em aplicações de computação gráfica interativas. O processo se dá através do controle da quantidade/montante de detalhes que representam o mundo virtual (Luebke *et al.*, 2003). O conceito foi concebido primariamente por Clark ao identificar a necessidade de melhorar o desempenho dos algoritmos que geram imagens de cenas em 3 dimensões (Clark, 1976).

A ideia fundamental para a redução dos custos computacionais baseia-se no uso de representações menos detalhadas para partes (a) pequenas, (b) distantes e, (c) menos importantes da cena. O conceito de nível de detalhe enfoca a redução da complexidade das redes polinomiais (*polygonal mesh*) que representam a cena. Outros parâmetros da cena como, por exemplo, a quantização e a discretização das cores podem ser vistos também como propriedades de detalhes, mas não são considerados neste conceito.

O nível de detalhe de um objeto para uma certa cena, que induz sua concretização na renderização, pode ser obtido por modelos diferentes. O modelo *discreto* contém o objeto em várias versões com níveis de detalhe variados, pré-calculados e guardados na memória. Além disso, o modelo *contínuo* consiste em uma representação do objeto a qual se gera um espectro de níveis de detalhe quase contínuo. Além da memória para a estrutura de dados do objeto, essa abordagem requer recursos na *Central Processing Unit* (Unidade central de processamento) (CPU) e/ou na *Graphics Processing Unit* (Unidade de processamento gráfico) (GPU). Um terceiro modelo de nível de detalhes (*view-dependent LOD*) ajusta ainda os níveis de detalhe em dependência do ponto de vista na cena atual e diferenciado para partes diferentes do objeto; porém isso corresponde a uma demanda aos recursos computacionais ainda maior que com os outros modelos. O autor apresenta ainda outras ideias avançadas de simplificação das cenas como, por exemplo,

- exclusão de objetos muito distantes da renderização (*view-frustum culling*),
- uso de processos paralelos para a preparação de objetos e cenas,
- utilização de versões diferentemente detalhadas de objetos pré-renderizados e
- uso de métricas perceptuais.

A avaliação das cenas e dos objetos simplificados pode ser feita *à mão* pelos desenvolvedores. Várias métricas computáveis foram desenvolvidas para cenas complexas com muitos objetos visíveis ou para situações nas quais a avaliação *à mão* não é viável. Essas métricas foram desenvolvidas tanto para o uso em tempo real quanto *off-line*.

Como métricas mais algébricas ou técnicas, usa-se, por exemplo, diferentes medidas/definições de distância entre as versões diferentes do objeto ou da cena, considerando também as diferenças de seus atributos como cor e textura. Por outro lado, critérios perceptuais foram introduzidos para guiar as simplificações, como, por exemplo,

- excentricidade do objeto na cena,
- velocidade do objeto e
- profundidade de campo (*depth-of-field*) da cena.

O critério de excentricidade considera o efeito de que a periferia da visão humana seja percebida com menos detalhes, o que permite o uso de menos detalhes para a renderização do objeto na cena. Baseado na observação de que o ser humano percebe menos detalhes em objetos com velocidades altas, o critério da velocidade do objeto relaciona o nível de detalhe à velocidade. A visão humana entendida pela ótica geométrica normalmente tem uma distância em foco da visão, fora dessa distância a visão é mais borrada quanto mais longe dessa distância (fora do foco). Decorrentemente, o critério da profundidade de campo atribui níveis de detalhe a um objeto correspondentemente à sua posição em relação ao foco ótico da cena.

Os conceitos desse domínio visual não podem, obviamente, ser aplicados diretamente no domínio aural, uma vez que não existem correspondências entre as áreas e para os parâmetros / propriedades espaciais como, por exemplo, distância, tamanho, foco e velocidade. Mesmo assim, pode-se aprender com as abordagens por comparação das metodologias, lembrando que o objetivo do controle do nível de detalhe se alinha com o interesse da presente pesquisa em controlar e gerenciar o *trade-off* entre qualidade e desempenho em tempo real. Correspondentemente, os conceitos desta seção por um lado informam a metodologia desenvolvida nesta pesquisa e, por outro lado, confirmam sua utilidade e viabilidade.

2.5.4 Simulação de salas

A ciência da acústica de salas foi estabelecida por Sabine há um século (Sabine, 1923), porém a simulação de salas tornou-se possível somente com o desenvolvimento dos computadores nos anos 1960 (Schroeder *et al.*, 1962), enquanto a auralização em tempo real popularizou-se apenas nos anos 1990, em função da evolução dos sistemas digitais e do crescimento exponencial da capacidade computacional.

Atualmente existem programas para o planejamento e a simulação de edificações, como salas de concerto ou instalações de grande porte para eventos ou pesquisa, assim como para a auralização de ambientes virtuais em jogos (Moldrzyk *et al.*, 2007). Existem também implementações voltadas a caixas de som, estúdios musicais ou ambientes físicos menores que prezam por definição sonora. A maioria desses programas baseiam-se na acústica geométrica, na qual o campo sonoro é reduzido à energia, tempo de propagação e direção de raios de partículas sonoras (Vorländer, 2008).

Nesta seção apresentaremos vários aspectos do campo de pesquisa relacionado com a simulação de salas. Discutiremos métodos e modelos centrais mostrando os pontos nos quais seria possível introduzir o mecanismo de *trade-off*, seus custos computacionais e suas respectivas consequências em termos perceptuais.

Resposta Impulsiva A resposta impulsiva é a resposta de um sistema a um sinal impulsivo (delta de Dirac). No nosso caso estamos considerando a sala como um sistema linear e por isso a resposta impulsiva vem a ser a caraterística central na simulação de salas. A partir da resposta impulsiva podem ser derivadas várias propriedades, como por exemplo a inteligibilidade, a transparência e a impressão espacial. Para essas propriedades, diferentes faixas da resposta impulsiva são relevantes: enquanto a parte inicial da resposta impulsiva é significativa tanto para a localização de fontes sonoras como para a inteligibilidade da fala, a parte tardia da resposta impulsiva é responsável pela impressão de reverberação e também para a imersão do ouvinte. Neste sentido temos que a simulação deve focar-se nas faixas da resposta impulsiva relevantes para o objetivo da aplicação.

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)\mathrm{d}\tau = (x * h)(t) \tag{2.13}$$

A equação 2.13 descreve uma sala quando vista como um sistema linear invariante no tempo (*linear time-invariant system* (Sistema linear e invariante no tempo) (LTI)) onde $x(t)$ é o sinal de entrada, $y(t)$ o sinal de saída e $h(t)$ a resposta impulsiva do sistema.

Assim fica óbvio que a resposta impulsiva pode ser usada para impor a caraterística da sala a um sinal de entrada por convolução.

De fato, aqui considera-se a resposta impulsiva apenas para uma configuração da sala, isto é, para posições fixas das fontes e do receptor. Em simulações dinâmicas a resposta impulsiva tem que ser recalculada cada vez que uma das posições muda. Em certos casos de uso, apenas uma média sobre um número pequeno de posições de recepção é necessária para formar uma resposta impulsiva generalizada *da sala*.

A seguir, vamos apresentar algoritmos fundamentais que computam a resposta impulsiva de uma sala a partir de um modelo apropriado, discutir seus limites e algumas abordagens de flexibilização dos mesmos. Implementações desses algoritmos são encontradas em uma multitude de software

sobretudo na área do planejamento acústico e de simulação, como por exemplo CATT-Acoustic[10], Enhanced Acoustic Simulator (EASE)[11], bem como diversos *plugins* para *digital audio workstations* (DAW) e aplicações CAD (Pelzer *et al.*, 2014).

2.5.4.1 Modelo de fontes virtuais (*Image source model*)

O modelo de fontes virtuais advém da acústica geométrica e visa a construção de fontes virtuais espelhadas. Ele parte da observação que reflexões de fontes sonoras são percebidas como sinais originados além dos limites físicos da sala. O histograma e a resposta impulsiva energética podem ser construídos a partir das distâncias entre as fontes virtuais e o receptor e, ainda, do número de reflexões de cada raio sonoro. O processo de construção das fontes virtuais pode ser descrito da seguinte forma:

1. criar fontes virtuais espelhando a fonte original em todas as paredes;
2. testar a audibilidade das fontes virtuais a partir do ponto do ouvinte;
3. criar fontes virtuais de ordem maior a partir das fontes virtuais audíveis já criadas;
4. repetir os passos 2 e 3 até que a distância entre as fontes virtuais e o ouvinte seja maior que $c\ t_{max}$, onde c é a velocidade do som e t_{max} o raio da simulação, que define o comprimento da resposta impulsiva calculada.

A resposta impulsiva pode ser construída a partir das distâncias que se relacionam a tempos de chegada ($t_{chegada} = d/c$) e da energia das partículas que sofreram atenuação ($E = E_0 \prod_{i=1}^{N} R_i$, onde N é o número de reflexões e R_i é o fator de reflexão da i-ésima parede no caminho do raio).

Conforme Cramer (Cremer, 1948) os custos computacionais crescem com ordem $\mathcal{O}(t_{max}^3)$ quando as fontes são espelhadas em três dimensões. O passo 2, que é repetido para cada fonte virtual construída, testa se as reflexões seriam percebidas dentro dos limites da sala através do teste *point-in-polygon*. Allen e Berkley (Allen e Berkley, 1979) mostraram que o teste de audibilidade não é necessário para salas retangulares, porque todas as fontes virtuais são audíveis nesse caso. Nos últimos anos foram desenvolvidas muitas modificações desse algoritmo básico. De um lado, para acelerar o processamento e, por outro lado, para superar as deficiências do modelo, tais como a desconsideração dos fenômenos de difração e espalhamento.

2.5.4.2 Traçado estocástico de raios (*Ray tracing*)

O traçado estocástico de raios é um método para computar a resposta impulsiva de uma sala a partir da ideia de fontes sonoras que emitem partículas sonoras (impulsos) em várias direções, cuja propagação é simulada geometricamente. Krokstad e coautores implementaram um dos primeiros algoritmos desse tipo (Krokstad, 1968). A Figura 2.7 mostra o processo geral dessa simulação.

No começo uma partícula é criada com uma direção aleatória seguindo uma distribuição predefinida. Depois - e em cada iteração - é determinado se a partícula bate num limite da sala (parede), nesse caso a direção e a energia da partícula são ajustadas conforme a lei de reflexão, e se a partícula chega no detector. O teste de absorção elimina a partícula caso ela exceda um limiar chamado de tempo máximo de vida. Numa outra versão do algoritmo usa-se um limiar de energia mínima para determinar a eliminação da partícula na simulação. Ambos os limiares são definidos anteriormente. Ao bater no detector, a energia restante e o tempo de vida da partícula são registrados no histograma da resposta impulsiva energética. O número total de partículas geradas depende de vários fatores, mas principalmente da desejada resolução da resposta impulsiva. Por causa da dependência da reflexão da frequência do sinal, a simulação deve ser repetida para as faixas de frequência de interesse.

[10] www.catt.se
[11] ease.afmg.eu

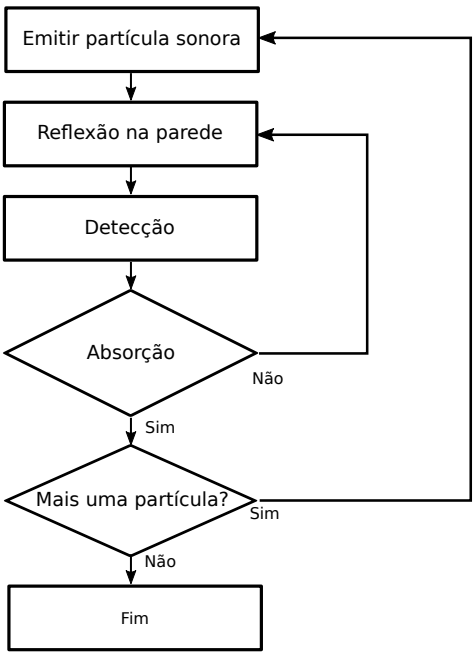

Figura 2.7: *Diagrama de fluxo do traçado estocástico de raios (Vorländer, 2008, p. 184).*

2.5.4.3 Modelos de salas

Qualquer software de simulação de salas tem que ter um modelo matemático do objeto a simular, incluindo fontes sonoras e receptoras. No detalhamento do modelo há de se considerar o objetivo da simulação, assim como o fato de que cada detalhe aumentará a complexidade da simulação e o tempo de execução da mesma. Dependendo dos fins para os quais a simulação é usada, o detalhamento pode ser ajustado, por exemplo, quando a inteligibilidade é mais importante ou quando o sinal de entrada tem uma faixa larga de frequências. Uma opção é incluir apenas objetos que têm um tamanho similar ou maior do que o comprimento das ondas sonoras em questão, tendo em vista que objetos pequenos tem pouca influência nas ondas sonoras de frequências baixas.

2.5.5 Espacialização

Dentro da área de auralização (*auralization*)[12], a espacialização pode requerer altos níveis de recursos computacionais, dependendo do tipo da simulação, dos algoritmos envolvidos e da precisão desejada e, por isso, geralmente se caracteriza por ser um problema de desempenho mais complexo a resolver, quando comparado a outras áreas. Por outro lado, os sistemas de espacialização trabalham não só com conteúdos sonoros, mas também com um modelo geométrico de fontes sonoras no espaço, para projetar/construir a cena acústica, permitindo usar essas informações para otimizar o processamento de dados.

2.5.5.1 Síntese de campo sonoro

O conceito da síntese de campo sonoro foi formulado por Berkhout (Berkhout, 1988) em 1988 e foi desde então desenvolvido e implementado para vários ambientes. A técnica tenta a ressíntese de um campo sonoro expandido, por meio da interferência de sinais de um conjunto de alto-falantes próximos. Matematicamente a síntese baseia-se na descrição do campo sonoro em função da pressão e velocidade na fronteira do ambiente de simulação pela Integral de Kirchhoff-Helmholtz. Em vários passos de simplificação, como a redução do modelo a duas dimensões ou a redução da superfície fechada por alto-falantes a distâncias discretas, é possível designar uma função *driver* para os sinais

[12] Aqui entendida como "técnica de criar arquivos de sons audíveis de dados numéricos de simulações, mensurações ou sintetizados" (Vorländer, 2008).

de cada alto-falante. Assim, os alto-falantes emitem o sinal calculado pelo *driver* que considera as propriedades geométricas da situação para sintetizar o campo sonoro desejado (Weinzierl, 2008). Para, por exemplo, representar uma fonte pontual numa certa posição do ambiente, a função do *driver* tem a seguinte forma generalizada:

$$Y_m(x,\omega) = S_{FP}\ F(\omega)\ A(x)\ D(x,\omega), \tag{2.14}$$

onde S_{FP} representa o sinal da fonte primária, $F(\omega)$ é um termo de filtragem, x contém os parâmetros geométricos e $A(x)$ representa a atenuação devida à distância da fonte primária a uma linha de referência que representa a posição ideal de um ouvinte. O termo de atraso $D(x,\omega)$ explica-se também por essa distância que a onda sonora tem que percorrer.

Para sintetizar um campo sonoro que contém mais fontes virtuais, cada sinal de saída dos alto-falantes se dá pela sobreposição de cada sinal de uma fonte primária processados pela função do *driver*.

2.5.5.2 Síntese binaural

A síntese binaural baseia-se na ideia de sintetizar um sinal a ser reproduzido diretamente no ouvido de um ouvinte. Para a síntese de uma cena com várias fontes, os sinais das fontes devem ser convolvidos com as respectivas respostas impulsivas e sobrepostos no sinal final. As respostas impulsivas devem descrever todo o caminho de transmissão do ponto onde a fonte se localiza até o ouvido. A equação 2.15 mostra essa relação para um ouvido e um número P de fontes:

$$p(t) = \sum_{i=0}^{P-1} s_i(t) * IR_i(t) \tag{2.15}$$

onde $p(t)$ é o sinal de saída, $s_i(t)$ o sinal da fonte i e $IR_i(t)$ a resposta impulsiva. A transformada de Fourier da resposta impulsiva - a função de transferência - compõe-se, para fontes esféricas num campo livre, da seguinte forma (Vorländer, 2008):

$$H = \frac{\mathrm{e}^{-j\omega t}}{ct}\ H_{ar}\ HRTF(\theta,\phi), \tag{2.16}$$

onde o primeiro termo é responsável pelo atraso do sinal devido à distância e H_{ar} representa a atenuação devida ao ar. As *head-related transfer functions* (HRTFs) descrevem os efeitos de filtragem do sinal, causados pelo torso, cabeça e orelha do ouvinte, e dependem do ângulo de incidência.

2.5.5.3 Otimização por agrupamento

Uma aplicação comum dentro desta área é a projeção espacial de fontes sonoras. Para a reprodução da cena espacializada existem várias técnicas como, por exemplo, a WFS, a síntese binaural e Dolby Atmos[13]. Em cada uma dessas técnicas, cada fonte sonora (ou 'objeto de áudio') tem que ser processada individualmente em função da sua posição espacial.

Partindo da noção da resolução angular do conceito de nível de detalhe (veja a Seção 2.5.3), Herder (Herder, 1998, 1999a,b) desenvolve a ideia de agrupamento de fontes sonoras e correlaciona a resolução angular na visão com o mesmo efeito psicoacústico: a resolução da localização sonora percebida. Ele propôs e implementou a ideia de juntar fontes virtuais distantes em grupos, reduzindo assim o número de fontes virtuais que devem ser processadas pelo sistema de espacialização, o que, por sua vez, reduz os custos computacionais do sistema inteiro. O algoritmo proposto pode ser separado nas seguintes etapas:

1. desconsiderar fontes não audíveis, não relevantes e silenciosas,

[13] www.dolby.com/technologies/dolby-atmos

2. formar grupos de fontes virtuais e calcular a posição, velocidade e direção representativa do grupo,

3. mixar/misturar as fontes virtuais em cada grupo para obter fontes representantes do grupo e

4. processar as fontes representantes (de grupos) no sistema de espacialização.

Quando os pontos 3 e 4 têm que ser executados continuamente para produzir a saída audível, os pontos 1 e 2 podem ter uma frequência de avaliação menor, dependendo das velocidades de movimentação espacial das fontes virtuais e da velocidade das mudanças na cena. A redução do número de fontes (ponto 1) segue a seguinte estratégia:

- as fontes com um volume[14] menor que um certo limiar são desconsideradas e tomadas como fontes silenciosas;
- excluir as fontes que se localizam fora do espaço de representação;
- atribuir todas as fontes cujo espectro contém apenas frequências baixas a uma fonte única de som ambiental, uma vez que a percepção da localização dessas fontes é espremida (não se percebe uma localização precisa).

A formação dos grupos segue um algoritmo que considera (a) a proximidade das fontes entre si, (b) a localização e (c) a velocidade das fontes em relação ao ouvinte e pode ser parametrizado quanto ao número de grupos desejados (que depende dos recursos computacionais disponíveis). Dentro dos grupos, as fontes são ordenadas pela sua importância[15] para facilitar o gerenciamento posterior de recursos.

Na mistura das fontes em um grupo (ponto 3) e na renderização dos grupos (ponto 4), as importâncias ou prioridades são usadas para gerenciar o número de fontes ou grupos a serem processados dependendo dos recursos disponíveis.

Para a avaliação do método, os participantes do teste de escuta estimaram o grau de dissemelhança entre as seguintes condições de renderização: 1) sem limitações no número de agrupamentos; 2) com número restrito de agrupamentos; 3) com número mínimo de agrupamentos (apenas dois mais um canal de ambiente). Os resultados mostram que os participantes julgaram as versões com e sem agrupamento como similares, discriminando a versão com dois grupos. Herder afirma que essa abordagem é capaz de diminuir os custos computacionais e ajustar a síntese às limitações de recursos computacionais. Todavia esse processo resulta em uma imagem espacial (*spatial image*) diferente da obtida sem essas limitações.

2.5.5.4 Auralização baseada na percepção

Uma outra abordagem quanto ao gerenciamento de recursos em aplicações de espacialização foi proposta por Tsingos (Tsingos, 2005). O aspecto central do conceito da auralização baseada na percepção é o uso de princípios psicoacústicos (veja a Seção 2.2) para a redução do volume dos dados que têm que ser processados pelo sistema de espacialização. A abordagem divide-se em 4 estágios:

1. **pré-processamento**: transformação dos sinais de entrada para o domínio da frequência em *frames*, computação / construção de descritores como, por exemplo, a tonalidade (*spectral flatness*)[16] e o valor eficaz (RMS) do espectro;

2. **mascaramento**: avaliação de audibilidade por análise psicoacústica para *frames* sucessivos;

[14] Nesse contexto, o volume é definido pelo ganho da fonte dividido pelo quadrado do distância da fonte ao ponto de escuta.

[15] Neste contexto, a importância é uma função do volume e da prioridade do grupo, e é usada como prioridade.

[16] Tonalidade ou *spectral flatness* representa aqui o quociente entre a média geométrica e a média aritmética do espectro.

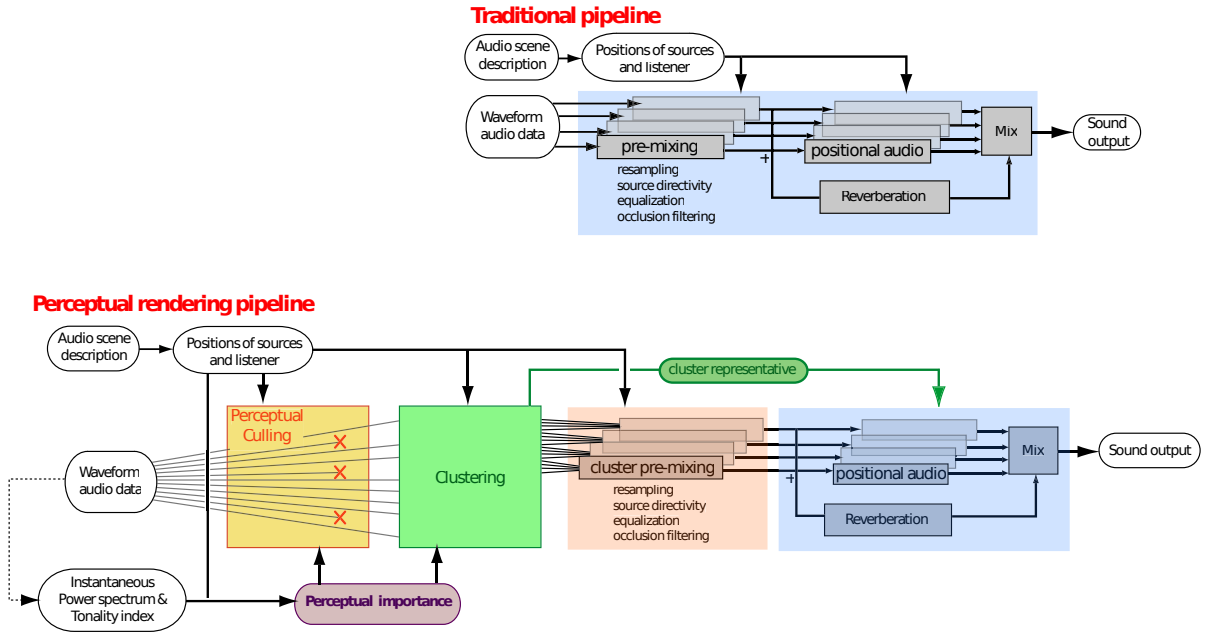

Figura 2.8: *Pipeline de espacialização tradicional e pipeline proposto por Tsingos et al.; reproduzido de (Tsingos et al., 2003).*

3. **amostragem por importância**: determinar o volume dos dados a selecionar e processar para cada sinal de entrada (atribuir cotas), dando preferência a valores de cota com mínimas degradações perceptíveis do resultado;

4. **processamento** dos dados selecionados.

As cotas determinam o número de bins por *frame* a serem processados para cada fonte e assim estão relacionadas ao número de operações básicas (multiplicações e adições) a computar. Por isso, a cota total (soma das cotas) é apropriada para controlar e gerenciar os custos computacionais da aplicação.

O autor implementa três aplicações (mixagem e equalização, filtragem FIR em blocos e espacialização) e afirma que melhorias significativas podem ser alcançadas em todos os casos, permitindo que mais fontes virtuais (objetos sonoros) possam ser usadas na entrada para a projeção.

Em uma pesquisa relacionada, Tsingos et al. (Tsingos *et al.*, 2003) enfocam um sistema para a auralização perceptual de cenas complexas. A Figura 2.8 mostra um pipeline de espacialização tradicional e o pipeline proposta e implementada por Tsingos et al. As imagens mostram a introdução de novos passos de processamento.

Abatimento perceptual (*perceptual culling*) O primeiro passo, o abatimento perceptual, é utilizado para reduzir o número de fontes (virtuais) a processar nos passos seguintes e usa descritores derivados no passo de pré-processamento, como achatamento espectral (*spectral flatness*) e, derivado disso, um índice de tonalidade (*tonality index*).

Agrupamento O agrupamento acontece em dois passos: (a) identificar representantes de agrupamento potenciais entre todas as fontes por uma abordagem heurística, (b) atribuir as fontes restantes por critérios de distância (geométrica) e importância (*loudness*) minimizando a soma dos erros angulares.

Além dessa proposta, existem vários outros algoritmos de agrupamento propostos pelos mesmos autores e em outros trabalhos, por exemplo, (Moeck *et al.*, 2007; Tsingos, 2005), que podem ser inseridos neste passo (usando/interpretando o pipeline da Figura 2.8 como esquema genérico).

O agrupamento dinâmico proposto por Moeck et al. (Moeck *et al.*, 2007) é uma estratégia recursiva; começa com apenas um agrupamento contendo todas as fontes e segue dividindo-se em

dois, e assim segue de forma recursiva até que o erro angular esteja abaixo de um certo limiar. Esta abordagem é diferente porque parte da divisão do espaço, enquanto abordagens anteriores são inicializadas com o número máximo de agrupamentos (igual ao número de fontes) para posterior associação (ou seja, redução do número de agrupamentos). Esta associação usa critérios de distância, velocidade e/ou direção (veja a Seção 2.5.5.3). Finalmente, os autores mostram em experimentos que o algoritmo recursivo tem melhor desempenho que os outros algoritmos.

2.5.5.5 Escalonamento baseado na percepção

A pesquisa de Foud et al. (Fouad *et al.*, 1997) apresenta uma técnica de gerenciamento de situações de sobrecarga em ambientes virtuais com muitas fontes sonoras. A abordagem incorpora uma estratégia de escalonamento em tempo real baseado no modelo de computação imprecisa (veja a Seção 2.4) com tarefas monótonas (método em etapas). Correspondentemente, em um primeiro passo do escalonador, as tarefas obrigatórias são planejadas de forma precisa; em seguida, o tempo de processamento restante é atribuído às tarefas opcionais monótonas. Para isso, os autores desenvolvem uma métrica justa (*fair*) que considera a priorização de certas fontes sonoras. A priorização se baseia na ideia de atenção que um ouvinte presta a um certo som e usa três fatores para avaliar os sons:

- a direção da cabeça do ouvinte, uma vez que um ouvinte tenta apoiar a percepção por correspondência visual;
- a intensidade do som (ou seja, o volume do som);
- a idade do som dentro do ambiente sonoro (*adaption response*).

A técnica, chamada alocação por prioridade (*priority allocation*), atribui o tempo de processamento restante às tarefas opcionais em proporção de suas prioridades. Em um experimento padronizado que comparou o algoritmo de alocação por prioridade (PA) com o algoritmo de *least utilization* (LU) proposto por Liu et al. (Liu *et al.*, 1987) foi mostrado que a percepção da degradação do sinal foi menor para o algoritmo PA em situação de sobrecarga forte e igual para situações de sobrecarga leve. Isso demonstra também que a métrica de atenção usada para inferir a prioridade é útil.

2.5.6 Mixagem seletiva de sons

Para desenvolver o método específico da mixagem seletiva, Kleczkowski e Pluta (Kleczkowski e Pluta, 2014) pesquisam o efeito perceptual da exclusão de certas componentes na mixagem de sinais musicais. O método proposto pelos autores consiste na execução dos seguintes passos para cada bloco de sinal de saída:

1. transformação dos blocos de entrada de todos os sinais para o domínio da frequência;
2. ordenação das componentes das fontes diferentes para cada faixa de frequências (*frequency bin*) pela magnitude local;
3. mixagem das componentes para cada faixa de frequências, excluindo as componentes com magnitudes menores que um certo limiar (veja a Figura 2.9).

O objetivo do experimento descrito no artigo foi o de estimar um limiar *favorável* em termos de métricas psicoacústicas como, por exemplo, *spaciousness*, localização, claridade, ausência de distorção e ruido. O limiar é definido como o quociente entre a energia de uma componente e a soma das energias de todas as componentes. As fontes musicais usadas foram, por exemplo, guitarra, saxofone e bateria, dentro do gênero jazz.

Os autores concluíram que, em geral, um limiar de 8 dB leva a experiências mais favoráveis, porém em alguns casos o limiar de 12 dB obteve a melhor avaliação. Mesmo que o método introduza mudanças audíveis, a maioria dos participantes dos testes preferiam o sinal manipulado

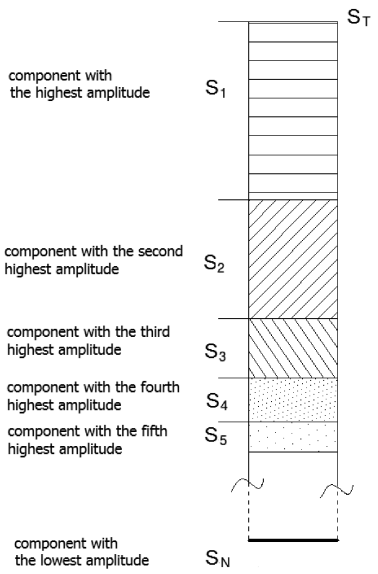

Figura 2.9: *Arranjo de fontes de entrada ordenadas pela amplitude local de cada fonte (S_i), onde S_T denota a energia total da faixa; reproduzido de (Kleczkowski e Pluta, 2014, pág. 2).*

por considerarem-no mais 'detalhado' (nos casos melhores), o que levou os autores a inferir que os resultados não são apenas altamente subjetivos, mas também que as escolhas dependem de preferências estéticas.

A proposta dos autores visto à luz desta pesquisa, oferece um potencial de *trade-off* entre o número de componentes escolhidos para a mixagem e qualidade do resultado, especialmente porque o resultado é avaliado (subjetivamente) como não irritante.

2.5.7 Linguagens de programação para tempo real

Existem várias linguagens de programação que permitem ou apoiam sistemas de tempo real como primitivos da linguagem, como, por exemplo, `Ada`[17], Real-Time Java Specification (`RTJS`)[18], `Real-Time POSIX C`[19] ou linguagens síncronas como `Lustre`[20] ou `Esterel`[21]. `Timed C`[22], por exemplo, adiciona primitivos de linguagem para definir pontos de cronometragem (*timing points*) que representam os pontos no tempo em que uma certa operação deve ser concluída, e no caso de transgressão deste limite, podem ser acionadas funções para conter o comportamento em tempo real (Natarajan e Broman, 2018). O uso destas linguagens é limitado pela capacidade de tempo real dessas sistemas e pela aplicabilidade nos arcabouços e ambientes de processamento de áudio relevantes para esta pesquisa. Porém, apresentamos aqui uma linguagem que trata o problema da imprecisão e de tarefas opcionais, indicando abordagens para o processamento flexível.

A linguagem de programação `FLEX` é uma linguagem experimental baseada em `C++` com funcionalidade para tempo real usando *performance polymorphism* (Kenny e Lin, 1991; Lin e Natarajan, 1991) com o objetivo de implementar sistemas flexíveis cujos tempos de execução podem ser ajustados para que todos os prazos de entrega sejam cumpridos, sob quaisquer circunstâncias. Como o escalonamento de tarefas com prazos de entrega e recursos limitados é NP-completo, a abordagem parte de cronogramas viáveis que contêm apenas tarefas com uma precisão mínima, e considera tempos e recursos ainda livres para melhorar a precisão dos resultados.

A definição da linguagem permite a definição de dependências (de outras tarefas ou condições) e limitações (*constraints*) para cada tarefa, estabelecendo a ordem de execução das tarefas bem como

[17] www.adacore.com/
[18] www.aicas.com/wp/products-services/technology#rtsj
[19] docs.oracle.com/cd/E19455-01/806-0632/6j9vm89ic
[20] www-verimag.imag.fr/The-Lustre-Programming-Language-and
[21] www-sop.inria.fr/meije/esterel/esterel-eng.html
[22] github.com/timed-c

a distribuição do tempo de execução e atribuição de outros recursos às tarefas. Para a previsão do tempo de execução e do uso de outros recursos, a componente dinâmica do `FLEX` (*run-time environment*) mede e atualiza esses valores no tempo de execução. Por outro lado, o arcabouço impõe a corretude temporal (*temporal correctness*) por interrupção de tarefas que transgridem os limites, chamando uma função de exceção. Tarefas do tipo *milestone* se encaixam facilmente neste modelo de execução uma vez que retornam resultados cada vez mais precisos no decorrer do tempo; uma interrupção pode acontecer a qualquer momento, retornando o resultado mais preciso computado até o momento da interrupção. A linguagem também apoia o modelo de versões múltiplas para as tarefas. No tempo de execução, o escalonador escolhe uma versão da tarefa a executar cujo uso de recursos - inclusive o tempo de execução - se encaixa com o cronograma planejado pelo escalonador. Esse mecanismo, chamado polimorfismo de desempenho, e a distribuição de tempo de execução às tarefas do tipo *milestone* dependem de uma relação estabelecida entre os recursos necessários (predominantemente: tempo de execução) e a recompensa em termos de precisão ou qualidade do resultado. No modelo de execução que `FLEX` estabelece, essa função de recompensa pode ser estimada antes da execução real ou diretamente no tempo de execução do programa em tempo real. Para esses fins, o arcabouço emprega um analisador estático no caso de medições antecipadas e um analisador dinâmico no tempo de execução do programa, que produzem arquivos com dados de desempenho para o uso em iterações futuras.

2.6 Ambientes de processamento

Nesta seção apresentaremos arcabouços e padrões para a programação de aplicações típicas de processamento de áudio relevantes para a presente pesquisa, pois representam ambientes em tempo real nos quais a flexibilização dos processos traria vantagens ao(s) usuário(s). A análise apresentada focar-se-á nas opções relevantes para a flexibilização dos custos computacionais, bem como na forma como essas ferramentas tratam situações de sobrecarga.

Concentraremos nosso escopo a aplicações para o sistema operacional Linux[23], ainda que os ambientes aqui apresentados existam em outros sistemas operacionais. A plataforma Linux é conveniente porque a maioria do software para esse sistema, incluindo o próprio Linux, é de código aberto, fato esse que facilita a pesquisa e favorece não só sua disseminação, como seu futuro desenvolvimento.

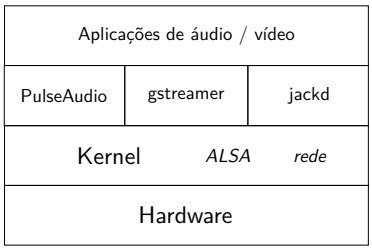

Figura 2.10: *Camadas do Linux audio stack.*

Linux audio stack As camadas típicas envolvidas no processamento de áudio em uma plataforma Linux pode ser visto na Figura 2.10. O kernel do Linux contém o arcabouço *Advanced Linux Sound Architecture* (Arquitetura avançada de som em Linux) (ALSA) que controla o hardware de áudio específico e oferece serviços/interfaces mais complexos para as camadas acima (*user space*), por exemplo, dispositivos lógicos (*devices* e *subdevices*) com interfaces de saída, de entrada, de configuração, de controle e de mixagem, entre outros.

Os arcabouços de áudio como `PulseAudio` e `Gstreamer` (veja a Figura 2.10) realizam uma abstração das interfaces de ALSA para oferecer a aplicações de usuário a semântica de fontes (*sources*),

[23] The Linux Kernel Archives, www.kernel.org

destinos (*sinks*), *plugins* e fluxos de dados (*streams*)[24]. Essa semântica é realizada/implementada pelo padrão clássico de programação de separação das aplicações em servidor e clientes. O *host* (servidor) tem a função de coordenar, gerenciar e escalonar os *plugins* (clientes), que executam o processamento de áudio a cada requisição do *host*, fornecendo os dados de áudio de entrada e guardando os dados de saída do *plugin*. Neste conceito, as entradas e as saídas de som reais são tratados como *plugins* sem *sink* ou, respectivamente, sem *source*.

2.6.1 PulseAudio

Atualmente, o PulseAudio é o servidor de som/áudio de padrão em distribuições de Linux e existe também para outros sistemas operacionais como *Berkeley Software Distribution* (Distribuição de software da Universidade Berkley) (BSD) e Mac OS X. O arcabouço segue o padrão de *host* e *plugin*[25]. Ele contém diversos plugins para o processamento dos dados (amplificação, equalização, codificação) e para a entrada e saída de som, por exemplo, para a transmissão pela rede (*Real-time Transport Protocol* (Protocolo de transmissão em rede em tempo real) (RTP)).

2.6.2 *Jack audio server*

JACK Audio Connection Kit (Arcabouço de programação de áudio Jack) (Jack)[26] é um servidor de áudio de baixa latência para sistemas seguindo o padrão *Portable Operating System Interface* (Interface portável entre sistemas operacionais) (POSIX) que conecta aplicações, os chamados clientes, a outras aplicações e dispositivos de áudio. `Jack` ganhou muita difusão porque é capaz de conectar clientes de diferentes contextos/processos (do ponto de vista do sistema operacional), em contraste aos *Digital Audio Workstation* (Estação de trabalho de áudio digital)s (DAWs) que normalmente deixam apenas importar *plugins* e permitem apenas conexões internas. Atualmente a maioria das aplicações de áudio pode conectar-se ao servidor `jackd`.

`Jack` ancora-se no modelo servidor-cliente, comunicando-se com os clientes e com o driver do dispositivo de áudio. O servidor analisa as conexões entre os clientes e forma um grafo correspondente ao fluxo de dados. Os clientes, que são os vértices nesse grafo, são chamados periodicamente para processar blocos de dados de áudio, e o servidor disponibiliza os resultados para os clientes sucessores no grafo até os dados chegarem no driver do dispositivo de saída (Letz *et al.*, 2005). Os dados trocados entre os clientes são amostras de 32-bits em ponto-flutuante, com uma taxa fixa de amostras por segundo, determinada conforme as opções do dispositivo de áudio. Essa definição estática tem a vantagem da simplicidade na implementação do servidor e dos clientes.

O escalonamento não considera os tempos de execução dos clientes, mas registra o atraso de entrega no driver do dispositivo de áudio e posteriormente notifica clientes do evento (*xrun*). Essa notificação permite, de forma simples, que os clientes possam ajustar-se à situação de sobrecarga. Todavia, o servidor `Jack` não tem influência no escalonamento do núcleo (*kernel*) do sistema operacional (além de ser executado com alta prioridade) e não conhece quais recursos computacionais estarão disponíveis no próximo ciclo.

2.6.3 O arcabouço *GStreamer*

`Gstreamer`[27] é um arcabouço para desenvolver aplicações multimídia, isto é, uma biblioteca para construir pipelines de elementos para o processamento de áudio ou vídeo. O arcabouço é muito usado para aplicações como tocadores e editores de música, conversores e servidores de transmissão (*streaming*), bem como aplicações de síntese. `Gstreamer` é muito popular na área

[24] Aplica-se aqui a mesma definição de *sink* e *source* como no contexto do arcabouço `Gstreamer` (confirme Figura 2.11): um *sink* denomina a entrada de som de um *plugin* e uma *source* sua saída.

[25] Um bom esquema para do PulseAudio em um sistema Linux pode ser visto aqui: rudd-o.com/linux-and-free-software/how-pulseaudio-works

[26] www.jackaudio.org

[27] `Gstreamer`, gstreamer.freedesktop.org

de dispositivos embarcados, por exemplo em sistemas de circuitos integrados (*System on a Chip* (Sistema-em-um-chip) (SoC)).

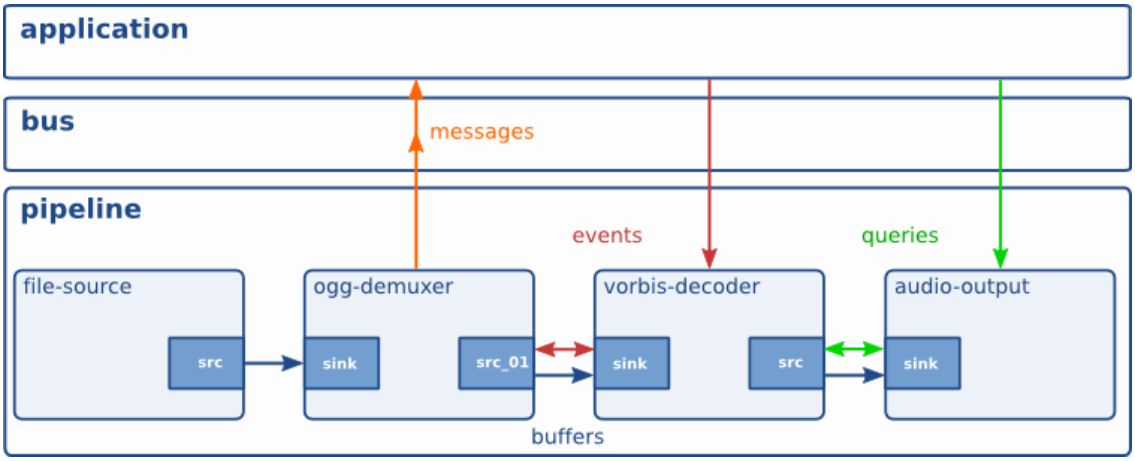

Figura 2.11: *Comunicação em uma aplicação usando* `Gstreamer` *(Taymans et al., 2015, p. 8).*

Na Figura 2.11 pode ser vista a estrutura geral de uma aplicação usando `Gstreamer`, onde os elementos de processamento formam um pipeline e comunicam-se através de mensagens de diferentes tipos. A aplicação pode utilizar essas mensagens para rearranjar conexões entre elementos, manipular parâmetros desses elementos e descobrir e recuperar erros. Isso permite a mudança de formato das amostras, bem como da taxa de amostras no tempo de execução (Taymans *et al.*, 2015). Todavia, as numerosas opções desse modelo implicam uma complexidade maior das aplicações, assim como acarretam um desempenho menor se comparado, por exemplo, ao servidor Jack. Para `Gstreamer`, existem centenas de *plugins* ou elementos disponíveis, não apenas para processamento de áudio, mas também ferramentas de otimização do fluxo de dados. Quanto ao escalonamento, o `Gstreamer` segue a mesma estratégia que o servidor Jack, e situações de sobrecarga são relatadas para a aplicação através de mensagens *Quality of Service* (Qualidade de serviço) (QoS), permitindo à aplicação reagir a essa situação.

2.6.4 Linux Audio Developer's Simple Plugin API version 2 - LV2

LV2[28] é um padrão aberto e extensível para *plugins* de processamento de áudio que podem ser usados em combinação com os assim chamados *plugin hosts*, que implementam a interface LV2. Como *plugin host* podem atuar, por exemplo, DAWs como Ardour[29] ou Qtractor[30] e também sistemas mais simples como Synthpod[31].

Em relação com os outros ambientes desta seção, o LV2 pode ser visto como um arcabouço parcial apenas para *plugins* cujo complemento é um *host*. Desta maneira, desenvolvedores criam apenas os plugins e não tem que lidar com a implementação do host.

Os *plugins* compõem-se de duas partes: o manifesto, que contém os metadados do *plugin*, e um arquivo que contém o código (em C ou C++) para o processamento dos dados de áudio. O *host* e o *plugin* comunicam-se através de mensagens, que permitem, por exemplo, mudanças dinâmicas de parâmetros. Isso incluí configurações das portas pelas quais os dados de áudio são trocados entre *host* e *plugin* que permite mudanças de formatos de dados trocados.

O comportamento de uma aplicação completa em situações de sobrecarga depende do *host* e normalmente é similar ao `Gstreamer`, ou seja, erros de atraso são registrados e notificados, mas não levam a uma reconfiguração automática do pipeline de processamento.

[28] LV2 homepage: lv2plug.in
[29] Ardour *digital audio workstation*, ardour.org
[30] Qtractor - *Audio/MIDI multi track sequencer*, qtractor.sourceforge.net
[31] Synthpod - contêiner não linear de *plugins*, openmusickontrollers.github.io/lv2/synthpod

Além dos ambientes aqui discutidos existem muitos outros, pois a área de processamento de áudio em plataformas computacionais está em desenvolvimento continuo[32]. Independente disso, a maioria destes arcabouços compartilham a mesma estrutura fundamental de *host* e *plug-in* e permitem a comunicação entre estas componentes. Consequentemente, estes arcabouços podem ser abordados com/pela metodologia proposta nesta pesquisa.

[32]Durante o tempo do desenvolvimento desta pesquisa popularizaram-se novos arcabouços de processamento para plataformas interativas, como por exemplo o arcabouço `Pipewire`

Capítulo 3

Uma metodologia para o processamento flexível

O objetivo deste trabalho é desenvolver uma metodologia que permita abordar a realização de um *trade-off* entre os custos computacionais do processamento de áudio e a qualidade percebida do resultado auditivo desse processamento. O foco desse trabalho recai sobre o processamento de áudio em tempo real e, decorrentemente, sobre a flexibilização dos custos computacionais relativos ao tempo de execução/processamento. Todavia, uma metodologia capaz de realizar um *trade-off* entre qualidade e um outro recurso, como por exemplo uso de energia, também poderia ser derivada desta proposta.

Como metodologia se entende no contexto desta pesquisa, uma série de métodos, procedimentos e técnicas que podem ser utilizados visando atingir o objetivo. A metodologia proposta compõe-se de:

- métodos de análise do desempenho de uma aplicação de áudio em tempo real de forma detalhada;
- métodos, abordagens e perspectivas relativos à parametrização dessa aplicação e de seus componentes;
- técnicas para a mensuração da percepção de qualidade;
- técnicas para conciliar o desempenho da aplicação e a qualidade da percepção;
- técnicas para controlar o desempenho da aplicação em tempo real (flexibilização).

A metodologia tem uma forma genérica, uma vez que a proposta pode ser aplicada a um número amplo de cenários e casos de uso. Sua validação é demonstrada neste capítulo de forma teórica; sua viabilidade para casos específicos demonstra-se pela implementação e pelos experimentos realizados. Por outro lado, esta prática guia a formulação da metodologia, uma vez que os componentes de software e ferramentas têm seus respectivos campos de uso, características e limitações. Assim, a parte experimental (cf. Capítulo 4) deve ser vista como complementar a este capítulo.

Como visto no Capítulo 2, existem várias soluções para casos particulares do problema do processamento de áudio flexível na literatura, por exemplo, para a flexibilização de certas aplicações de espacialização, para a avaliação de qualidade ou para o controle de recursos em tempo real. Nossa proposta parte também de soluções parciais, mas almeja uma generalização das técnicas estudadas na literatura através da metodologia aqui desenvolvida. A diferença principal entre o foco desta pesquisa e a de seus trabalhos relacionados está, por um lado, em como combinar essas técnicas para o uso em um sistema flexibilizado e, por outro lado, em identificar os potenciais parâmetros de flexibilização em algoritmos de processamento de áudio existentes.

A análise e parametrização de algoritmos de processamento de áudio tem como propósito a identificação e a introdução de parâmetros novos que afetam o *trade-off* entre desempenho e qual-

idade, considerados requisitos essenciais para a flexibilização. A flexibilização realiza-se pelo controle do comportamento da aplicação no tempo de execução utilizando esses parâmetros; em outras palavras, a parametrização possibilita a flexibilização. No contexto desta análise é introduzida a versão do modelo de computação imprecisa adotado pela metodologia aqui proposta. Mensurações do comportamento da aplicação e de suas partes completam a análise destes algoritmos; correspondentemente, serão discutidos alguns aspectos práticos relacionados, como o ambiente de mensuração.

Propõe-se também, como componente da metodologia, um método eficaz para mapear a correspondência entre a flexibilização dos custos computacionais e os efeitos desta sobre a qualidade percebida pelo usuário. Esse mapeamento é usado para otimizar o *trade-off* entre desempenho e qualidade. A viabilidade desse método é demonstrada com exemplos práticos na parte experimental desta pesquisa.

Nas seções a seguir, serão apresentados elementos e cadeias específicos de processamento de sinais e possíveis estratégias de flexibilização, estratégias para o gerenciamento dinâmico do *trade-off* entre qualidade do resultado e custos computacionais, e algumas limitações da abordagem proposta.

3.1 Análise e parametrização do processamento de áudio

Para a análise de algoritmos de processamento de áudio, supõe-se o modelo comumente utilizado para esse tipo de aplicação, que é baseado (a) em elementos de processamento, que são os parâmetros do sistema e, portanto, unidades elementares, e (b) no fluxo dos dados entre eles. Os elementos básicos são executados periodicamente, processando blocos de amostras, que são repassados para os próximos elementos na cadeia para processamento na próxima rodada. Isso é verdade não apenas para os arcabouços analisados (veja a Seção 2.6), mas também para um grande número de outros arcabouços e aplicações de processamento de áudio em tempo real.

Nas aplicações analisadas, a coordenação segue o padrão *push flow* (`Gstreamer`) ou *pull flow* (jackd, Ardour, `Gstreamer`, lv2) (Vermeulen *et al.*, 1995), e as cadeias de processamento compõem um pipeline [1]. Decorrentemente, essas aplicações de processamento de áudio podem ser conceituadas e representadas por grafos direcionados. A discussão parte dos vértices e arestas do grafo (elementos básicos de processamento de áudio e suas respectivas entradas e saídas) para, em seguida, considerar subgrafos (pipelines e aplicações inteiras).

A questão a resolver pela análise neste passo é estabelecer a função de custos computacionais e, com isso, identificar os parâmetros específicos que têm influência no desempenho da aplicação. Nesta análise, não consideramos o tempo que uma aplicação precisa para a coordenação do fluxo de dados, considerando que este tempo é geralmente pequeno e quase constante (veja a Seção 3.3.2).

[1] Este modelo também é referenciado muitas vezes como *pipes and filters*; no entanto, neste trabalho usa-se *elementos de processamento* para denominar *filters* no contexto teórico. No contexto das implementações em software utiliza-se *plugins*, mesmo que os arcabouços tenham adotado outras denominações, como *client* (jack, SuperCollider), *unit generator* (MusicN, SuperCollider), *opcodes* (MusicN, CSound) ou *nodes* (PipeWire) .

3.1.1 Elementos básicos de processamento

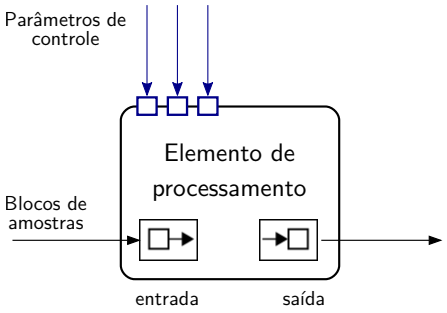

Figura 3.1: *Elemento de processamento em representação esquemática.*

Deseja-se identificar aqui as unidades elementares que compõem um algoritmo de processamento de áudio e que correspondam a manipulações específicas e recorrentes destes sinais. Por um lado, um elemento básico é caracterizado por ter uma utilidade do ponto de vista do usuário no contexto de sua produção musical; por outro lado, pode-se ver em uma análise mais detalhada que muitas unidades de processamento de áudio comuns são compostas por mais do que um processamento elementar. Exemplos de elementos básicos encontrados comumente têm funções distintas, como filtragem, compressão, amplificação, mixagem, reverberação, espacialização, etc. A literatura sobre esse tipo de elemento básico é ampla; nesta pesquisa considera-se exemplos de Collins (Collins, 2010), Moore (Moore, 1990) e Zölzer et al.(Zölzer, 2011). Uma análise destes algoritmos revela sua estrutura interna e suas componentes. Por exemplo, um equalizador pode conter vários filtros distintos e também passos de amplificação; a função de *vocoder* requer a análise de som por FFT e a ressíntese por IFFT, entre outros procedimentos. No entanto, no contexto desta pesquisa, adota-se o ponto de vista da utilidade para o usuário, segundo o qual tanto equalizador quanto vocoder podem ser considerados elementos básicos, em conformidade com a organização de software de produção musical.

Existem muitas implementações destes elementos básicos em pacotes, ou usadas individualmente, comumente como *plugins*. As análises desta pesquisa baseiam-se principalmente nas coleções ligadas aos ambientes de processamento consideradas na Seção 2.6 e na parte experimental (veja o Capítulo 4).

Um elemento de processamento recebe dados de entrada (sinal), processa-os (de acordo com os parâmetros de configuração ou controle) e produz dados de saída. Os custos do processamento dependem na maioria dos casos: (a) do tamanho do bloco de dados (amostras) a ser processado e, (b) dos custos para processar cada um desses dados (amostras). Na Figura 3.1 pode ser visto um exemplo de uma representação esquemática de um elemento de processmanto, que tem uma entrada e uma saída de sinal com três parâmetros de controle. Além disso, existem vários elementos básicos cuja função não é a produção de blocos de amostras na saída, mas a produção de metadados, como por exemplo a extração de *features* (loudness, tonalidade, conteúdo espectral, etc.). A este grupo pertencem também elementos que envolvem algoritmos mais complexos como a avaliação de características psicoacústicas[2] e derivadas, como por exemplo qualidade de áudio[3], impressão espacial ou ainda algoritmos da área de *music processing and music information retrieval*[4]. Esses elementos também devem ser incluídos na análise e adaptação.

[2]Por exemplo, *japa* é um analisador perceptual para o *Jack Audio Connection Kit*: wiki.linuxaudio.org/apps/all/japa

[3]Por exemplo, *gstpeaq* é um *plugin* do arcabouço `Gstreamer` para a avaliação da qualidade percebida conforme descrito na norma ITU-R BS.1387-1.

[4]Por exemplo, *chromaprint* é um *plugin* do arcabouço `Gstreamer`: gstreamer.freedesktop.org/documentation/chromaprint

3.1.1.1 Adaptação ao modelo da computação imprecisa

É preciso considerar os principais métodos da computação imprecisa, apresentados na Seção 2.4. No ambiente considerado nesta pesquisa[5], estes métodos não podem ser aplicados de forma direta, uma vez que os sistemas operacionais considerados não conseguem garantir o tempo exato de uma interrupção. Consequentemente, no nível do ambiente de processamento (arcabouço de áudio) não existem recursos para realizar mecanismos como os exigidos nos modelos da computação imprecisa.

No entanto, certos mecanismos para os métodos peneira e de etapas poderiam ser implementados à base de *soft realtime*. Porém, isso implicaria em profundas mudanças nos arcabouços[6], contradizendo o critério de mudanças mínimas nos códigos existentes. Haveria também a opção da implementação dos mecanismos dentro dos elementos básicos (sem modificar o arcabouço), por exemplo, o próprio elemento poderia observar os tempos de execução para estimar custos futuros, e usar isso para sua adaptação e auto-interrupção. Todavia, essa opção é desconsiderada pela alta complexidade de implementação estimada.

A proposta desta pesquisa se adapta ao método das versões múltiplas. Este método difere dos outros porque nele o escalonador arbitra sobre os recursos concedidos/atribuídos ao elemento, decidindo qual versão do elemento é chamada, sendo que a versão do elemento escolhida executa sua tarefa sem interrupção. Este método pode ser adaptado como segue: ao invés de cada versão indicar um bloco de código distinto, as versões referem-se ao mesmo bloco de código, porém com parâmetros de configuração distintos. Esta adaptação é viável para os sistemas e arcabouços considerados, uma vez que não exige mudanças nos próprios arcabouços, já que este método não requer a interrupção dos elementos durante sua execução.

Para os elementos, isso implicaria apenas na parametrização do desempenho dos mesmos. Na lógica de programação, a parametrização teria sua expressão em parâmetros adicionais nas chamadas do *plugin* em tempo de execução. A opção de executar o elemento com valores diferentes para esses parâmetros pode ser entendida como uma flexibilização do elemento.

O objetivo da análise e parametrização dos elementos básicos se divide nos seguintes passos:

- estabelecer uma forma teórica da função de custos computacionais;
- introduzir novos parâmetros de acordo com os critérios de utilidade e viabilidade;
- mensurar o desempenho do elemento para diversos valores dos parâmetros;
- estabelecer uma função de custos parametrizada, a partir da forma teórica e dos dados obtidos pelas mensurações.

Os primeiros dois pontos estão interligados, pois a introdução de novos parâmetros requer um entendimento detalhado do funcionamento do elemento, obtido pela análise de suas características, o que inclui a análise dos custos. Porém, a adaptação do algoritmo, que reestrutura o elemento, incita nova análise de custos e modifica as características do elemento.

3.1.1.2 Síntese de uma função de custos

Executa-se uma análise dos custos computacionais dos elementos básicos a partir do código-fonte, considerando-se, também, o algoritmo em pseudocódigo.

A análise segue os métodos descritos, por exemplo, por (Cormen *et al.*, 2000; Knuth, 1998; Sedgewick e Wayne, 2011). Contrariamente às análises de custos computacionais mais comuns, que se interessam principalmente pelos custos de pior caso, parece ser útil nesse caso incluir na análise um mapeamento mais diversificado de casos, principalmente como um tentativa de incluir outros fatores que contribuam para os custos computacionais, modelando, assim, o desempenho

[5] no nível de sistema operacional: POSIX.1-2001 e principalmente POSIX.1j a padronização das extensões para tempo real (*soft*)

[6] Os arcabouços investigados têm apenas uma semântica, a saber, a de chamar a execução de um *plugin* e esperar sua conclusão, porque foram concebidos para sistemas de *soft realtime*.

do elemento de forma mais precisa e detalhada. Com isso, neste caso, a análise do algoritmo tem como resultado não apenas uma função de custos, mas também estabelece um mapeamento entre componentes da função de custos e as estruturas específicas no algoritmo que produzem tais custos.

Mesmo que o objetivo final exija a contagem de custos em unidades de tempo (s), conta-se as operações no pseudocódigo ou no código-fonte em unidades abstratas, considerando-se que cada uma delas tem custo (simbólico) unitário, ou seja, constante. Cada um dos parâmetros do elemento que tenham influência nos custos, entendidos aqui como potenciais parâmetros de flexibilização, deve estar representado na função. Todavia, em muitos casos, existem parâmetros que são indispensáveis para o funcionamento intencionado pelo usuário, como por exemplo a frequência de corte ou o tipo de transformação (passa-baixa ou passa-alta) de um filtro, os quais, neste caso, devem ser desconsiderados como parâmetros de flexibilização, pois sua alteração modificaria a própria utilidade do elemento para o usuário.

```
1  // x[]: bloco das amostras de entrada
2  // y[]: bloco das amostras de saída
3  // para cada faixa de frequências:
4  // a0[], a1[], a2[], b1[], b2[]: coeficientes dos filtros
5  // hx1[], hx2[]: valores das amostras de entrada passadas
6  // hy1[], hy2[]: valores das amostras de saídas passadas
7
8  for each sample index i:
9    for each frequency band index f:
10     y[i] =  a0[f]*x[i] + a1[f]*hx1[f] + a2[f]*hx2[f] + b1[f]*hy1[f] + b2[f]*hy2
           [f]
11
12     // update history
13     hx2[f] = hx1[f]
14     hx1[f] = x[i]
15     hy2[f] = hy1[f]
16     hy1[f] = y
17     x[i] = y[i]
```

Algoritmo 3.1: *Resumo do algoritmo de equalizador.*

A análise do código-fonte exemplar do Algoritmo 3.1 indica custos computacionais de $z = nFc_1$, onde n representa o número de amostras, F é o número de faixas de frequências e c_1 representa a soma de todos os custos computacionais constantes das linhas 10 até 17.

Obtém-se desta maneira uma função (teórica) de custos da forma:

$$Z_{elemento} = z(n, p_1, \ldots p_P), \tag{3.1}$$

onde n é o número de amostras e $p_1, \ldots, p_P$ são os parâmetros adicionais. Complementarmente, é necessário definir as faixas de valores úteis para cada um dos parâmetros, que podem ter tipos diferentes, como:

- números inteiros ou ponto flutuante,
- tipos binários representando decisões, como por exemplo a de usar uma dentre duas funções alternativas,
- tipos enumerados para permitir seleções, por exemplo, para selecionar um método de interpolação.

Em muitos casos existe um parâmetro representando o tipo de dados das amostras a processar, tais como `F32LE`[7] e `F64LE`[8].

[7] Denomina valores do tipo ponto flutuante com 32 bits na ordem *little-endian*.
[8] Como `F32LE`, mas usando 64 bits.

3.1.1.3 Introdução de novos parâmetros

A metodologia de introdução de novos parâmetros é feita através de métodos heurísticos, como descrito a seguir. É importante ressaltar que a sistematização exaustiva de possibilidades de flexibilização em algoritmos de processamento de áudio não é o foco deste trabalho. Isso significa que a metodologia tem a forma de ideias, indícios e exemplos.

Nessa seção, parametrização significa a introdução de parâmetros novos. Parâmetros são qualificados pelos seguintes critérios:

- o algoritmo contém métodos ou mecanismos (correspondentes ao parâmetro) que o adaptam a valores diferentes do parâmetro;
- o parâmetro participa da função de custos computacionais (tendo uma influência relevante nela).

A questão da influência da parametrização na qualidade do resultado é discutida na Seção 3.2. Ela coloca-se aqui como uma questão sobre casos extremos: Quais são os valores (ou faixas de valores) úteis, fora dos quais o elemento não cumpre mais sua função? Funcionalidade, neste contexto, equivale a cumprir as intenções do usuário.

Decorrentemente, os problemas que se colocam são:

- quais são os parâmetros de interesse para a parametrização,
- como se realiza a parametrização no nível do algoritmo,
- como se quantifica o parâmetro na função dos custos, e
- quais são valores ou faixas de valores úteis?

Como pontos de partida usa-se resultados de trabalhos apresentados na Seção 2.5. Dentre estes, a dissertação de Bianchi (Bianchi, 2013) descreve três parâmetros fundamentais e típicos para a parametrização dos elementos. Em sua pesquisa, Bianchi identifica parâmetros comuns a todos os algoritmos pesquisados por ele que têm impacto no desempenho mas não inibem a função do algoritmo, descrevendo inclusive valores úteis para tais parâmetros. Esses parâmetros são:

- o número de amostras a processar,
- o tipo numérico das amostras,
- um seletor de algoritmos dentre um conjunto de algoritmos de complexidades e custos computacionais diferentes.

O primeiro parâmetro é o mais comum nos algoritmos considerados e, portanto, deve participar da função de custos. No contexto de processamento periódico de blocos de amostras, o número de amostras a processar por um elemento é controlado pelo tamanho do bloco e pelo período de execução do elemento.

Os demais parâmetros indicados pelo trabalho de Bianchi poderiam ser investigados como possíveis parâmetros para o elemento em questão, desde que ainda não participassem da função de custos (e do algoritmo do elemento). Neste contexto, para cada parâmetro é necessário projetar: (a) como ajustar o elemento para funcionar com valores distintos desse parâmetro, (b) qual é a função de custos adaptada, e (c) quais são os valores úteis para esse parâmetro?

O tipo numérico de amostras, segundo parâmetro indicado por Bianchi, pode ter uma influência nos custos computacionais quando, por exemplo, se usa processadores (CPUs) superescalares[9]. Ainda pode haver diferenças de desempenho entre tipos baseados em números inteiros ou ponto flutuante, uma vez que os processadores tratam esses dois tipos de formas diferentes e usando

[9]Processadores que processam certas operações em paralelo mesmo tendo apenas um núcleo. Por exemplo, o comando ADDPS (de uma CPU x86-64), adiciona paralelamente 4, 8 ou 16 valores do tipo ponto flutuante.

comandos diferentes. O tipo numérico das amostras pode entrar na função de custos como condição ou diretamente como fator junto às operações. Vale ressaltar que a precisão dos valores é diferente entre os tipos de dados e que isso impacta o resultado e sua precisão.

O terceiro parâmetro, enfatizado por Bianchi, indica a seleção entre algoritmos diferentes entre si. Essa abordagem é congruente com o modelo das versões múltiplas da computação imprecisa. Todavia, uma investigação em diferentes implementações de operações básicas[10] pode ser mais complexa e requer uma análise não apenas do algoritmo mas também dos dados de entrada esperados. Se for aplicável, poder-se-ia investigar como implementar distintas versões de operações como aproximação, estimação ou interpolação. Para a operação de interpolação existem, por exemplo, a interpolação linear, cúbica ou por repetição. Cada tipo de interpolação possui propriedades particulares, distintas áreas de aplicação e diferentes implementações, implicando em custos distintos.

Na Seção 3.5 são apresentadas algumas investigações de elementos particulares, juntamente com propostas de parametrizações.

Na introdução de certos parâmetros, mostra-se que, em alguns casos, tem-se que introduzir mecanismos adicionais para ajustar o elemento aos padrões de entrada e saída. Por exemplo, o tipo numérico das amostras tem que ser adequado na entrada e saída. O processamento com diferentes tipos de dados internamente ao elemento implica na conversão dos dados após a entrada e antes da saída. Em muitos casos, os custos adicionais pelas conversões não se justificam em relação aos custos totais do elemento. No entanto, em um outro nível de análise, considerando cadeias de elementos, os custos relativos são menores. Para a parametrização de uma cadeia de elementos, isso significa que tais mecanismos de conversão ou adaptação não teriam que ser repetidos em cada elemento interno, como discutido na Seção 3.1.2.

3.1.1.4 Mensuração do desempenho do elemento

Executa-se mensurações dos tempos de execução do elemento básico. As mensurações dos tempos de execução do elemento em consideração têm como objetivo a concretização da função de custos obtida no passo anterior. A função de custos é contabilizada em unidades abstratas, cada unidade representando uma operação. Contudo, essa função pode ser transformada e mapear o tempo de execução em segundos e aproximada aos valores mensurados para obter uma função de custos útil ao processo de flexibilização.

A mensuração deve explorar um subconjunto representativo do espaço de todas as possíveis combinações de parâmetros, para viabilizar o mapeamento entre os parâmetros da função e os custos em tempo de execução ligados a ela.

Ambiente de mensuração Para este passo é importante definir e estabelecer um ambiente de mensuração. Esse ambiente deve ser configurado para que as medições aconteçam sob as mesmas condições, de forma reprodutível. Por exemplo, a velocidade e a carga do sistema são estáveis ou reprodutíveis? É útil que o ambiente de medição seja tão semelhante quanto possível ao caso de uso, mas isso tem que ser conciliado com a necessidade de mensurar o tempo de execução do elemento de processamento de forma menos instável possível, uma vez que apenas desta forma espera-se obter uma correlação com a função teórica de custos. Além disso, tem-se que considerar a disponibilidade de ferramentas de mensuração e suas características. Quais ferramentas são adequadas? Quais quantidades podem ser mensuradas por qual ferramenta? Qual é a influência do estado da máquina e do ambiente às quantidades observadas? Em particular, algumas métricas podem ser distorcidas quando, por exemplo, a carga do sistema for alterada.

Devido à vasta complexidade deste processo, é necessário estar atento a efeitos colaterais inesperados em certas configurações e, se for necessário ou útil, explorar diferenças no comportamento para excluir estes efeitos ou, pelos menos, descrevê-los e quantificá-los. O número de repetições no

[10] implementações simples tendo custos computacionais pequenos com resultados imprecisos e/ou implementações mais complexas tendo custos maiores mas com resultados precisos

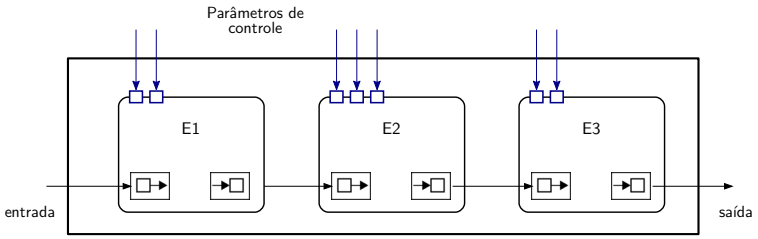

Figura 3.2: *Exemplo de uma cadeia de elementos.*

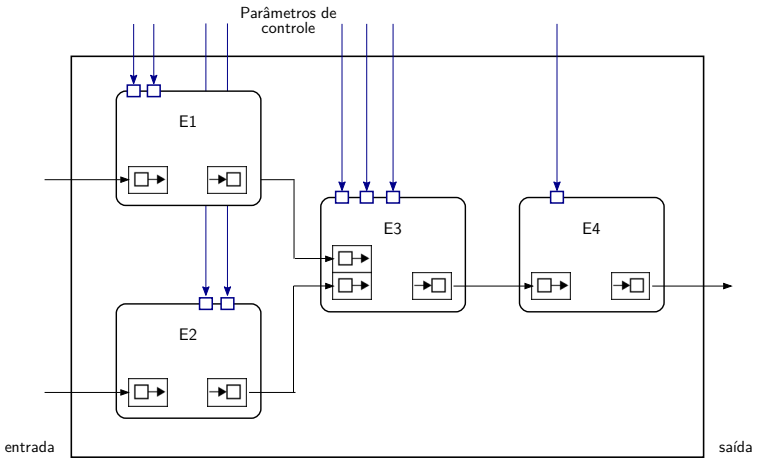

Figura 3.3: *Exemplo de uma cadeia de elementos com duas entradas de sinal.*

experimento deve ser planejado para minimizar a influência de efeitos temporários da plataforma e de erros aleatórios de medição.

No caso demonstrado no Capítulo 4, observa-se que a contagem de operações executadas pelo elemento tem uma forte correlação com o tempo de execução, quando a carga total do sistema for baixa e a aplicação for executada com a maior velocidade possível (com alta prioridade). Além disso, foi concluído que os tempos de execução assim mensurados correspondem aos tempos de execução dos mesmos elementos mensurados no próprio caso de uso.

3.1.1.5 Função limiar de custos computacionais

Computa-se a função dos custos por aproximação da função teórica aos resultados das mensurações.

As consequências de uma eventual sobrecarga em um sistema 'ao vivo' (artefatos audíveis indesejados) parecem piores do que as consequências de uma subutilização de recursos (possibilidade de uma pior qualidade percebida) do ponto de vista desta pesquisa. Consequentemente, a função de custos aproximada ($\hat{z}$) não deveria subestimar os custos reais (aqui representados pelos valores mensurados) e, decorrentemente, deve ser verdadeira a seguinte relação:

$$\forall \tilde{p} : \hat{z}\left(\tilde{p}\right) \geq \tilde{z}\left(\tilde{p}\right), \tag{3.2}$$

sendo $\tilde{p}$ uma combinação de parâmetros com a qual foi executada uma mensuração de custos e $\tilde{z}(\tilde{p})$ um valor mensurado com a combinação de parâmetros $\tilde{p}$.

No caso demonstrado no Capítulo 4, usa-se um método da otimização linear para este passo de aproximação.

3.1.2 Cadeias e grafos de processamento

Em uma aplicação de processamento de áudio, certos elementos do processamento são combinados para produzir o resultado desejado, utilizando o resultado intermediário do processamento

de um elemento como entrada para outro(s) elemento(s), e assim por diante. Dessa maneira, aplicações de áudio podem ser descritas como grafos de processamento, onde os elementos básicos de processamento são os vértices enquanto as arestas representam o fluxo de dados de áudio digital. Considerando a aplicação inteira como grafo, cadeias correspondem a subgrafos que podem conter vários elementos, e cadeias podem também aparecem como elementos combinados em cadeias maiores. No extremo, contendo todos os elementos, a cadeia pode se igualar à aplicação inteira. Nas Figuras 3.2 e 3.3 podem ser vistos dois exemplos de cadeias de elementos de processamento com uma e duas entradas com os parâmetros de controle de cada elemento.

Esta seção traz uma discussão dessa perspectiva, bem como as relações de *trade-off* que tornam-se factíveis em função dessa perspectiva.

Cadeias podem ser identificadas a partir do ponto de vista da produção musical, como por exemplo, faixas que contêm todos os elementos que processam os dados de um instrumento musical (veja a aplicação no Capítulo 4). Em aplicações de espacialização, os passos de processamento de cada fonte virtual (pipeline) constituem uma cadeia, como também as fontes agregadas no caso do agrupamento. Além disso, pode-se executar algoritmos em grafos para identificar subgrafos que possam ser tratados como cadeias.

Os custos computacionais de uma cadeia de elementos correspondem à soma dos custos de seus elementos com os custos de transferência de dados entre esses elementos, além dos custos de coordenação (escalonamento). Esses custos de transferência e de coordenação são constantes para cada elemento ($\mathcal{O}(1)$), portanto crescem linearmente com o número de elementos ($\mathcal{O}(n)$, sendo n o número de elementos de processamento). Porém, esses custos são muito pequenos em comparação com os custos internos aos elementos. Essas suposições são verdadeiras para os arcabouços considerados nesta pesquisa (Bagnoli, 2010; Letz *et al.*, 2005; Taymans *et al.*, 2015). Correspondentemente, calculam-se os custos computacionais da cadeia (Z_{cadeia}) como

$$Z_{cadeia} = z_c(E) + \sum_{i=1}^{E} Z_i, \tag{3.3}$$

onde z_c representam os custos computacionais de transferência e de coordenação em função do número de elementos E, e Z_i são os custos computacionais do i-ésimo elemento. Considerando a Equação 3.1, que expressa os custos de cada elemento em função do número de amostras e de seus parâmetros internos, calculam-se os custos da cadeia como

$$Z_{cadeia} = z_c(E) + \overbrace{\sum_{i=1}^{E} z_i \left(n_i, p_1^{(i)}, \dots p_{m_i}^{(i)}\right)}^{Z_E}, \tag{3.4}$$

onde z_i representa a função de custos do i-ésimo elemento, n_i o número de amostras do bloco e $p_j^{(i)},\ j = 1, \dots, m_i$ os parâmetros do i-ésimo elemento. O termo destacado Z_E representa a soma das funções de custos dos elementos, que depende de todos os parâmetros $n_i, p_1^{(i)}, \dots p_{m_i}^{(i)}$ dos elementos individuais.

A partir das fórmulas acima é possível estimar custos máximos e mínimos para cada elemento em função das faixas de valores de seus parâmetros. Em relação aos custos de transferência e coordenação $Z_c(E)$, em muitos casos pode parecer que flexibilização do número de elementos não faça muito sentido do ponto de vista do usuário. Além disso, a flexibilização deste parâmetro deve contribuir pouco ao potencial de exploração do *trade-off* , considerando seus baixos custos. Decorrentemente, é possível tratar esse termo como constante.

3.1.2.1 Introdução de novos parâmetros

Na análise e parametrização dos elementos, encontraram-se parâmetros cujo controle não faria sentido individualmente para cada elemento, pois exigiriam adaptações na interface entre elementos.

Entretanto, considerando cadeias, é possível introduzir tais adaptações antes do primeiro e após o último elemento de uma cadeia, ou seja, na interface da cadeia com o restante do grafo, uma vez que os custos computacionais dessas adaptações têm menos peso relativamente aos custos da cadeia do que em relação aos custos de elementos individuais. No entanto, a parametrização da cadeia tem como requisito que todos os elementos contidos na cadeia compartilhem o mesmo parâmetro. A faixa de valores válidos para este parâmetro pode ser obtida como conjunção das faixas de valores válidos do elementos.

A seguir, serão apresentadas algumas parametrizações possíveis em aplicações comuns, ressaltando que esta listagem não poderia ser completa já que a identificação das cadeias e parametrizações dependem muito da aplicação em questão.

Parametrização da taxa de amostragem A maioria dos elementos de processamento no âmbito musical executam laços sobre todas as amostras. Do ponto de vista de cadeias de elementos é possível introduzir uma forma de perfuração de código, por exemplo, seguindo Sidiroglou et al. (veja a Seção 2.5.2), através do controle do número de amostras de entrada na cadeia. Uma variação do número de amostras a processar causa uma variação do número de laços dentro dos elementos, permitindo então uma parametrização dos custos computacionais da cadeia em questão. Tal método envolve dois processos: a manipulação dos dados na entrada da cadeia e uma forma de reconstrução dos dados de saída para reestabelecer a taxa de amostras.

Das opções propostas por Sidiroglou et al. investiga-se aqui o método da omissão modular, por se considerar a mais adequada ao contexto deste trabalho. Com esta opção visa-se eliminar cada n-ésima amostra na entrada da cadeia, omitindo assim as respectivas iterações dos laços. Isso pode ser realizado neste contexto pelo método de reamostragem, que muda o número de amostras por segundo (ou por bloco) de forma sistemática. Dos outros métodos propostos por Sidiroglou et al., o método da omissão de amostras no começo e/ou no fim não tem correspondência útil neste contexto. O método de omissão aleatória gera resultados menos previsíveis que no caso da omissão modular, e ainda traz o problema da reconstrução da sequência, pois seria necessário saber quais amostras foram omitidas e em qual ponto da sequência.

Por outro lado, o método da reamostragem é bem conhecido e utilizado em processamento de sinais, e particularmente em processamento de dados digitais de áudio (Collins, 2010; Crochiere e Rabiner, 1983a; Moore, 1990). O método tem como entrada uma sequência de amostras, com uma certa taxa de amostragem, e como saída uma sequência correspondente, porém representada com uma outra taxa de amostragem. Em uma reamostragem de uma taxa maior para uma taxa menor, perde-se informação que não pode ser reconstruída em uma reamostragem inversa, pois tem-se que filtrar (por um filtro passa-baixo) o sinal antes da reamostragem a fim de evitar o rebatimento (*aliasing*) do conteúdo espectral acima da frequência de Nyquist. Consequentemente a reamostragem pode ter forte influência na qualidade percebida, dependendo do conteúdo espectral dos dados de áudio.

Parametrização do formato das amostras (tipo de dados) Como discutido na Seção 3.1.1, o tipo numérico das amostras pode ter impacto nos custos computacionais dependendo das implementações dos elementos e da plataforma computacional (em particular, a CPU). Seguindo a mesma lógica que com a parametrização da taxa de amostragem, pode-se investigar a utilidade do uso de tipos numéricos distintos junto com a inclusão de elementos de adaptação de tipo.

Todavia, é importante considerar os erros introduzidos, pois as adaptações de tipo necessárias requerem passos de quantização. Considerando a discussão da quantização na Seção 2.2, sabe-se que o mapeamento entre tipos numéricos distintos impacta no SQNR e na faixa dinâmica do som representado. Esse tipo de erro é individual para cada par de tipos numéricos considerados na conversão. A perda de qualidade pode ser grave em certos casos.

É importante considerar, também, que o número de tipos numéricos úteis é pequeno em comparação com a faixa de valores do parâmetro da taxa de amostragem, e que por isso, a transição entre os tipos não é gradual. Por exemplo, não existem tipos intermediários entre inteiros de 16 bits e inteiros de 8 bits: essa transformação deve causar uma mudança mais drástica na qualidade

percebida do que se fosse possível realizar por exemplo uma transformação de 16 bits para 14 bits; porém, implementações que usem inteiros de 14 bits não são usuais nesta área[11]. Isso ocorre porque os tipos numéricos existentes em software estão enraizados nos tipos numéricos definidos e usados pela CPU. Por isso, não é possível tirar aproveito de um ganho teórico no desempenho com uma representação de amostras com 14 bits em comparação com uma representação com 16 bits, uma vez que a CPU oferece registros e operações com 16 bits mas não com 14 bits.

Parametrização do tamanho dos blocos processados Também é possível considerar a parametrização do tamanho dos blocos processados. Supondo que o número de amostras a processar não mude, a única diferença no comportamento seria uma mudança da frequência de chamadas dos elementos, o que tem potencial de economia computacional uma vez que cada chamada implica em custos de troca de contexto. Na prática, isso pode implicar em custos distintos uma vez que tais chamadas têm custos computacionais próprios. Adicionalmente, o tamanho do bloco é vinculado ao atraso final do resultado, sendo que a latência associada a blocos maiores (menos chamadas, mais economia) poderia incomodar em ambientes interativos.

Apesar de um possível ganho teórico, não se propõe nesse trabalho considerar uma tal parametrização, pois a complexidade de eventuais implementações seria alta e requeria mudanças nos próprios arcabouços, por ausência de ferramentas suficientes. Por exemplo, o Jackd trabalha com um tamanho de bloco fixo para todos os *plugins*, e o mesmo é válido para Ardour usando plugins do tipo LV2. Em aplicações de `Gstreamer` o tamanho dos blocos é variável, porém não existem ferramentas para apoiar sua flexibilização em tempo de execução.

Na Seção 3.5 serão apresentados (e citados) algumas outras abordagens de parametrização e flexibilização, especialmente em contextos específicos (métodos de simulação e espacialização).

3.1.2.2 Mensuração do desempenho do elemento

Para estimar os custos computacionais de transferência e de coordenação (c_z, veja a Equação 3.4), executa-se mensurações com configurações de parâmetros distintos. Nesse caso, teoricamente, tem-se que fazer apenas uma série de mensurações com apenas uma configuração de parâmetros, pois os custos de transferência e de coordenação devem ser independentes dos parâmetros da função z_E. Todavia, convém que se desenvolva, através das mensurações, uma intuição quanto ao comportamento da cadeia, uma vez que esta pesquisa, neste ponto, tem caráter heurístico. Por exemplo, não se pode excluir a possibilidade de que aconteçam efeitos colaterais (resultado da interferência/interação entre os elementos) que deveriam ser investigados com o objetivo de ajustar ou corrigir a função de custos da cadeia Z_{cadeia}.

No caso de parametrizações introduzidas no nível da cadeia, adicionam-se os custos das transformações introduzidas,

$$Z_{cadeia} = z_c\left(E\right) + z_a(p_{a,1} \ldots p_{a,P_a}) + z_E\left(n_1, \ldots n_E, p_{1,1}, \ldots p_{1,P_1}, p_{2,1} \ldots p_{E,P_P}\right). \tag{3.5}$$

Correspondentemente, executam-se mensurações com um subconjunto representativo do espaço de todas as possíveis combinações dos parâmetros $p_{a,1} \ldots p_{a,P_a}$. Também é possível a análise e mensuração dos elementos de adaptação de forma isolada, incluindo suas funções de custos na soma da Equação 3.5 para depois estimar os custos de transferência e de coordenação (z_c) com a cadeia inteira.

A função de limiar de custos deve ser obtida por aproximação da função teórica com os resultados das mensurações, da mesma forma como foi feito para os elementos, por exemplo, por otimização linear; o método deve respeitar a condição 3.2.

A análise e parametrização do processamento prossegue com cadeias cada vez maiores, que englobam elementos e cadeias já investigados, até que a cadeia seja igual à aplicação inteira. Dependendo da escolha do tipo de gerenciamento de recursos para a aplicação, pode ser útil estabelecer uma representação hierárquica dos elementos.

[11] Veja, por exemplo, a biblioteca `libsndfile` ou o arcabouço `Gstreamer`

Paralelização Em sistemas com mais do que um núcleo, a aplicação de áudio pode ser executada de forma paralela, considerando que ela seja capaz de tirar vantagens da paralelização. Os arcabouços de processamento de áudio investigados nesta pesquisa permitem a sua execução de forma paralela pelas seguintes características:

- os sinais de entrada são segmentados em blocos e o processamento dos blocos é realizado através de interrupções assíncronas;
- cada função de processamento (elemento) depende exclusivamente do bloco correspondente à sua entrada no momento de cada interrupção, e portanto não depende dos resultados dos demais elementos processados na mesma interrupção.

A execução paralelizada leva a tempos de execução menores, mas tem um limite que está na própria estrutura da aplicação e de seus dados de entrada e nos recursos de hardware disponíveis. Por exemplo, a aceleração computacional de um código arbitrário em função de um certo número de núcleos de execução é limitada teoricamente pelas lei de Amdahl e de Gustafson-Barsis (Gebali, 2011). Entretanto, a obtenção dos limitantes superiores teóricos para o fator de aceleração, além de complexa no contexto desta pesquisa, não ajudaria na construção de versões paralelas dos algoritmos considerados.

Por outro lado, a execução paralela dos elementos no contexto das interrupções do processamento segmentado em blocos é realizada trivialmente dentro dos arcabouços e permite a obtenção de estimativas experimentais para os fatores de aceleração desejados. O fator de aceleração da execução de uma aplicação pode ser mensurado como proporção entre o tempo de *processamento* da aplicação (a soma dos tempos de execução de cada instrução[12]), e o tempo real que a aplicação necessita[13] para sua execução. A mensuração destes tempos é possível em tempo real através do sistema operacional ou do próprio arcabouço, e representa uma forma viável de estimação do fator de aceleração. Esse fator de aceleração pode ser usado juntamente com as funções de custos teóricos estabelecidas para estimar o tempo de execução da aplicação.

3.2 Gerenciamento de recursos

Os elementos e cadeias de processamento compõem a parte que processa os dados de áudio em uma ordem pré-definida. O ponto discutido nesta seção é como gerenciar o processo por inteiro visando a flexibilização. O gerenciamento flexível, neste contexto, inclui tarefas de decisão e execução de ações de flexibilização na aplicação, em referência ao *trade-off* . As decisões devem ser tomadas usando o conhecimento dos estados do processamento da aplicação e do ambiente computacional. Adicionalmente, tais decisões dependem do conhecimento sobre a aplicação quanto à previsão de custos e de qualidades percebidas. O pressuposto desse mecanismo de controle é sua eficácia e que o futuro comportamento do sistema seja controlável.

A seguir, esta seção discorre sobre o escalonamento de tarefas, sensores da plataforma computacional e a qualidade perceptual. Ao final, a seção aborda o gerenciamento da flexibilização interrelacionando tais aspectos.

3.2.1 Escalonamento

No contexto desta pesquisa, pressupõe-se que a aplicação de áudio é executada em um sistema operacional interativo, preferencialmente do padrão POSIX.1b que oferece opções de *soft real time* (veja a Seção 2.4). Além disso, enfoca-se apenas o gerenciamento do recurso tempo e a qualidade percebida. A aplicação de áudio é executada usando os recursos e serviços oferecidos pelo sistema operacional, tendo a forma de um servidor de áudio com *plugins* ou de uma aplicação integrada.

Essa configuração consiste em duas 'camadas' de escalonamento:

[12] *process time*
[13] *wall-clock time*

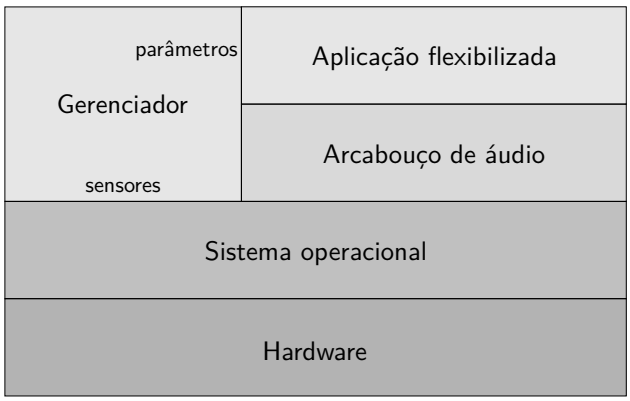

Figura 3.4: *Camadas principais da plataforma computacional e uma aplicação flexibilizada.*

1. No primeiro nível, o sistema operacional é executado com seu próprio gerenciamento de recursos. O sistema não pode ser de tempo real uma vez que é interativo. A opção de *soft real time* se traduz apenas na promessa de execução prioritária de certas tarefas (Kerrisk (Kerrisk, 2010, pág. 737)), já que o sistema não consegue garantir a entrega de resultados em um tempo limite arbitrário.

2. No segundo nível, e dependendo do primeiro, atua o escalonador do arcabouço em forma de servidor de áudio, ou de forma embutida na aplicação. Uma vez que não pode contar com tempos limite, a única opção da aplicação é detectar problemas posteriormente e reagir apropriadamente. O pressuposto implícito do arcabouço é que: 'o que funcionou no passado vai funcionar no futuro'; ou seja, que os recursos computacionais disponíveis no próximo ciclo serão os mesmos. Os arcabouços pesquisados oferecem apenas mecanismos rudimentares para resolver problemas de sobrecarga, normalmente apenas uma sinalização do problema a certas componentes. A função principal desses escalonadores consiste no planejamento da ordem de execução dos elementos, na coordenação dos fluxos de dados e na transmissão de mensagens entre as componentes da aplicação.

A composição do sistema em camadas permite que o escalonador do arcabouço e o usuário possam abstrair os processos no nível do sistema operacional. No nível da aplicação, as unidades principais tratadas são os elementos e cadeias de processamento de áudio.

Neste trabalho, adiciona-se uma nova camada de controle de processamento (gerenciador de flexibilização) que permite abstrair o escalonamento dos elementos, pressupondo que os elementos seriam escalonados de maneira eficiente pelo mecanismo de escalonamento do arcabouço. O gerenciador de flexibilização planeja apenas a configuração da aplicação através dos parâmetros de flexibilização, considerando o *trade-off* entre o tempo de execução e a qualidade percebida. O controle de desempenho da aplicação será realizado no nível (de abstração) do arcabouço por mensagens aos elementos e cadeias. Na Figura 3.4 pode ser vista uma representação desta estrutura proposta. Contudo, essa abordagem descarta outras soluções existentes, na medida em que as soluções não podem ser integradas nesta camada adicional.

As simplificações decorrentes da abstração permitem uma estrutura clara das funções do gerenciador de flexibilização:

1. apuração do estado de sistema (inclusive análise das mensagens do arcabouço);

2. decisão sobre a configuração de parâmetros dos elementos para o próximo ciclo;

3. comunicação dos parâmetros aos elementos de processamento.

Para tanto, é necessária a parametrização prévia dos elementos, encaixando-os no modelo das versões múltiplas. Isto permite o controle do tempo de execução necessário à aplicação de áudio.

Destaca-se o problema de escalonamento, quando for reduzido apenas ao recurso tempo e visto no contexto de tempo real (veja a Seção 2.4): o tempo de execução da aplicação não pode ser maior que o tempo disponível. A seguinte desigualdade (baseada na Desigualdade 2.7) expressa essa relação para o caso de processamento de áudio em tempo real:

$$T_n \geq t_a(n), \tag{3.6}$$

onde T_n é o tempo máximo disponível para o processamento de um bloco de n amostras e t_a é o tempo de processamento necessário para a aplicação produzir um bloco de n amostras. O valor de T_n é determinado pela taxa de amostras usada na conversão dos dados pelo hardware (saída de áudio) e tem que ser visto como constante. Essa desigualdade pode ser restringida pelos tempos de execução necessitados pelo sistema operacional e por outras aplicações em execução, com o resultado:

$$T_n \geq t_a(n) + t_s + t_o, \tag{3.7}$$

onde t_s é o tempo de execução de processos do sistema operacional no intervalo T_n e t_o é o tempo de execução de processos de outras aplicações no mesmo intervalo.

Nota-se que essa desigualdade é válida apenas para sistemas com uma CPU[14]. Em sistemas com mais de uma CPU, o tempo disponível para a execução de processos (*process time*)[15] é maior que o tempo de parede (*walltime*)[16]. A aceleração, que aplicações e o sistema operacional usufruem, é expressada por um fator de aceleração, que depende da arquitetura do hardware e do próprio software com sua capacidade de execução em paralelo. Assim, obtém-se:

$$T_n \geq f_a \hat{t}_a(n) + t'_s + t'_o, \tag{3.8}$$

onde $\hat{t}_a$ é o tempo de execução da aplicação de áudio contado em tempo de processo (*process time*. que corresponde ao tempo de execução em uma CPU da Equação 3.6), f_a representa o fator de aceleração, e t'_s e t'_o representam os tempos de execução dos processos do sistema operacional e das outras aplicações em execução em tempo de parede.

O fator de aceleração da aplicação investigada é estimado por experimento discutido na Seção 4.5.3.1. Uma vez que o número de núcleos da CPU comumente não muda durante a execução dessa aplicação, o fator pode ser visto como um fator constante de conversão entre o tempo de processo (*cputime*) e o tempo real (*wall-clock time*) para a aplicação. O fator de aceleração pode ser mensurado durante a execução da aplicação pelo gerenciador de flexibilização.

Destaca-se, que o gerenciador de flexibilização deve assegurar a observância da Desigualdade 3.8 através do controle do tempo de execução da aplicação de áudio. Para seu planejamento do próximo ciclo, o gerenciador pressupõe que os tempos de execução do sistema operacional e das outras aplicações sejam os mesmos que no ciclo passado. Essa suposição, também necessária para o escalonamento, tem sua origem na imprevisibilidade de plataformas interativas e consequente incapacidade de previsão do futuro pelo sistema operacional, mas na prática as abordagens baseadas nessa premissa funcionam na maioria dos casos.

Para o gerenciador de flexibilização, propõe-se, adicionalmente, o uso de sensores que avaliam propriedades específicas do sistema na tentativa de estimar a futura carga do sistema, tornando-o mais proativo se comparado com abordagens que apenas reagem a falhas passadas. Correspondentemente, a Equação 3.8 pode ser expandida da seguinte forma:

$$T_n \geq f_a \hat{t}_a(n) + t'_r + p, \tag{3.9}$$

onde t'_r é a soma dos tempos consumidos por todos os outros processos no ciclo anterior ($t'_s + t'_o$) e p é a mudança prevista do tempo de execução dos processos, bem como processos adicionais no

[14] No intuito de simplificar a nomenclatura, usa-se no texto o termo CPU como sinônimo de core, denominando uma unidade que executa um processo por vez, sem paralelismo.

[15] Por tempo de processamento entende-se o tempo que o processo leva em uma CPU.

[16] O tempo de parede é o tempo necessário para execução de uma tarefa do ponto de vista do usuário.

próximo ciclo de avaliação. O valor de p também pode ser negativo se a estimação, derivada dos sensores, prever um uso menor do recurso tempo.

3.2.1.1 Sensores de estado do sistema

Para o escalonador, sensores relevantes são aqueles que detectam o estado do sistema (no passado) e, melhor ainda, aqueles que permitem uma prognose do estado no futuro. O caso em que o gerenciador de flexibilização tem que prevenir situações de sobrecarga, e decorrentes interrupções de áudio, parece mais crítico do ponto de vista da qualidade percebida do que o caso de subutilização de recursos, uma vez que em caso de subutilização perde-se apenas uma oportunidade de obter uma qualidade gradualmente melhor, associada a uma reação tardia do gerenciador. Por isso, esta pesquisa optou por enfocar os casos em que o tempo disponível para a execução da aplicação de áudio diminui.

Um preceito comum para a análise de desempenho de sistemas computacionais é o método *USE* (*utilization, saturation, errors*), que propõe a investigação de métricas de utilização, de saturação e de número de erros ocorridos. O método aponta que a utilização e a saturação de certos recursos podem indicar o uso de recursos futuros.

Como a carga do sistema[17] pode ser usada para a estimação da carga futura, o crescimento da carga do sistema pode indicar um crescimento futuro. Isto fica claro quando da inicialização de aplicações que, muitas vezes, utiliza recursos diferentemente dos utilizados em fases seguintes. Por exemplo, a inicialização de um processo no nível do sistema operacional envolve muitas vezes operações como requisição de memória. No nível da aplicação observa-se, muitas vezes, operações como a inicialização de variáveis e a abertura de arquivos, sendo que, nas fases seguintes da execução, muda-se o perfil de requisições de recursos computacionais. Precisamente por isso, esse comportamento típico pode ser usado pelo gerenciador de flexibilização na estimação da mudança da carga do sistema quanto ao recurso tempo (parâmetro p da Desigualdade 3.9).

Processos, no nível do sistema operacional, são executados por certos intervalos de tempo (jiffies), depois são interrompidos para deixar outros processos serem executados na CPU. O processo continua sua execução só depois de outros processos, dependendo da decisão do escalonador. Um processo esperando sua execução na fila indica um requisito futuro de tempo de execução. Por isso, a soma de todos os tempos de espera na fila, e o crescimento dessa podem ser usados também na estimação de p.

Como foi argumentado anteriormente, um crescimento de uso de memória por um processo pode indicar um aumento de tempo de execução no futuro, por isso, tanto o crescimento no uso de memória quanto o tempo de espera pelo requerimento de memória também podem ser considerados para a estimação dos tempos que os processos utilizarão no próximo ciclo.

Para a previsão do uso do recurso tempo no futuro, p, propõe-se para esta metodologia as seguintes métricas:

- a carga do sistema (utilização);
- o uso de memória (utilização);
- a soma dos tempos de espera na fila de cada processo esperando por sua execução (saturação);
- a soma dos tempos que cada processo espera para o requerimento de memória (saturação).

O peso de cada métrica para previsões úteis tem que ser determinado experimentalmente. Na parte experimental desta pesquisa, foram encontrados configurações e valores úteis para os casos investigados.

[17] A carga do sistema é a medida que relaciona o uso de um recurso ao seu uso máximo possível (potencial) e representa a utilização do recurso.

3.2.2 Avaliação de qualidade

Conceptual aspect	Example of Issues	Suitable Measuring Methods
Auditive Quality Classical Psychoacoustics	Perceptual properties such as loudness, roughness, sharpness, pitch, timbre, spaciousness	*Indirect scaling*: thresholds, difference limens, points of subjective equality *Direct scaling*: category scaling, ratio scaling, direct magnitude estimation
Aural-scene Quality Perceptual Psychology	Identification and localization of sounds in a mixture, speech intelligibility, audio perspective incl. distance cues, scenic arrangement, tonal balance, aural transparency	*Discretic*: semantic differential, multidimensional scaling. *Syncretic*: scaling of preference, suitability, and/or appropriateness, benchmarking against target sounds
Acoustic Quality Physics	Sound-pressure level, impulse response, transmissions function, reverberation time, sound-source position, lateral-energy fraction, inter-aural cross correlation	*Instrumental measurements* with physical equipment for the measurement of elasto-dynamic vibrations and waves, including appropriate signal processing
Aural-communication Quality Communication Sciences	Product-sound quality, comprehensibility, usability, content quality, immersion, assignment of meaning, dialogue quality	*Psychological (cognitive) tests*, particularly in realistic use cases, e.g., the product in use, the audience in concert, etc., questionnaires, dialogue tests, comprehension test, usability tests, market surveys

Tabela 3.1: *Sinopse dos quatro níveis conceituais de qualidade de som, reproduzido de Blauert et al. (Blauert e Jekosch, 2010).*

Na sua publicação sobre um modelo de qualidade em níveis (Blauert e Jekosch, 2010), Blauert et al. enfatizam: "ao selecionar métodos de avaliação quantitativa da qualidade sonora ... é de extrema importância estarmos conscientes da quantidade de abstração envolvida em cada tarefa específica de avaliação"[18]. Na Tabela 3.1, proposta por Blauert et al., pode-se ver a classificação dos métodos de mensuração pelos aspectos conceituais envolvidos na avaliação de qualidade de som.

Para esta pesquisa é imprescindível encontrar um método de avaliação quantitativa de qualidade que seja automatizável, como forma de avaliar o sistema em função de diferentes configurações da aplicação. Como descrito no Capítulo 2, existem vários métodos de extração de propriedades psicoacústicas para a avaliação da qualidade auditiva. Entretanto, o método PEAQ foi introduzido para a avaliação objetiva de qualidade percebida na tentativa de desenvolver um método automatizável que envolva um modelo cognitivo. Por envolver um modelo cognitivo, o método PEAQ pode ser classificado no segundo nível da Tabela 3.1 como método sincrético de avaliação de qualidade de uma cena aural. Neste nível existem outros métodos automatizáveis, tais como métodos de localização de som e de mensuração de inteligibilidade e equilibro tonal, como vistos na Seção 2.3. Em comparação com os demais métodos, o PEAQ engloba um maior número de aspectos da qualidade auditiva e da cena aural. Por exemplo, um método de estimação da qualidade na reprodução da localização de fontes sonoras[19] não registra outras qualidades sonoras.

Algumas limitações na aplicação do método PEAQ no contexto desta pesquisa são delineadas a seguir, a fim de explicitar as hipóteses simplificadoras adotadas nesta pesquisa, segundo as quais o PEAQ é considerado um método adequado para a estimação da qualidade de áudio.

Um dos objetivos principais da norma que estabelece o método PEAQ é a comparabilidade de resultados de equipamentos técnicos, buscando afastar-se de uma avaliação subjetiva individual-

[18] "having to select methods for quantitative assessment of sound quality ... it is of paramount importance to be aware of the amount of abstraction involved in every specific assessment task."

[19] veja `amtoolbox`

izada. A norma limita sua aplicação, na medida em que define suas sub-métricas e seus instrumentos de mensuração, por exemplo, (a) o ouvinte *expert* formado pelo treino de escuta, de comunicação (vocabulários consensuais) e de exclusão de vieses; (b) o teste de escuta em condições que excluam influências não controláveis; (c) a afinação do algoritmo pelo uso de valores médios. Esses elementos da norma garantem a generalidade dos resultados científicos nos quais se apoia, o que, ao mesmo tempo, diminui a relevância da norma para o uso individual em contextos específicos. Por exemplo, um ouvinte poderia estar mais interessado nas variações de sua percepção em função dos seus diversos contextos cotidianos de escuta do que em uma média de suas avaliações perceptuais, quanto mais de uma média de avaliações de diversos ouvintes.

Especificamente, a avaliação da qualidade percebida por um usuário não corresponde necessariamente com a avaliação do algoritmo PEAQ. Como visto na Seção 2.3, certos descritores psicoacústicos objetivos (como por exemplo, o incômodo psicoacústico (*psychoacoustic annoyance*)) perdem sua capacidade de descrição quanto se trata de efeitos estéticos. Decorrentemente, o PEAQ, que incorpora índices similares (por exemplo, o índice de distúrbio (*disturbance index*)) tem sua relevância diminuída naqueles contextos. Mais ainda, o usuário pode possuir predileções pessoais na sua avaliação individual, enquanto possíveis vieses de sujeitos de teste são excluídos na construção da norma que define o PEAQ.

Sterne descreve no seu livro (Sterne, 2012) a interdependência entre a percepção de qualidade de música e vários fatores sociais. No seu texto, o autor explora, entre outras, a questão da popularidade de formatos de áudio inferiores (de baixa definição) diante da disponibilidade de formatos de alta definição. Uma razão apurada é que certos usuários tendem a preferir sons imperfeitos em função de sua memória afetiva.

Consideradas todas estas limitações, para esta metodologia propõe-se o uso do algoritmo PEAQ porque ele, dentre os métodos automatizáveis, é o mais abstrato na classificação de Blauert. Além disso, o método PEAQ é padronizado em uma norma, o que permite uma maior reprodutibilidade e comparabilidade dos resultados em relação a outros algoritmos de estimação de qualidade. Considerando que a metodologia proposta também possui caráter experimental, outros métodos automatizados podem ser considerados para a avaliação de qualidade, principalmente em domínios específicos de aplicações de áudio, como a espacialização. Na parte experimental desta pesquisa (veja a Seção 4.3.2), discute-se mais detalhadamente implementações de métodos de avaliação de qualidade, sua disponibilidade e escolhas de software para a experimentação.

3.3 Gerenciamento do *trade-off* entre qualidade e custos

A aplicabilidade do algoritmo PEAQ na metodologia desenvolvida é determinada principalmente pelas seguintes propriedades:

- o método PEAQ compara duas versões de sinal de áudio e computa o grau de diferença objetivo (ODG) de qualidade percebida;
- a qualidade do resultado final do processamento de áudio não é facilmente modelável a partir de combinações das qualidades de resultados parciais (locais).

A primeira propriedade restringe a aplicabilidade do algoritmo sob condições de tempo real, pois duas versões teriam que ser produzidas simultaneamente em situação de quase sobrecarga computacional. A segunda propriedade tem sua razão na componente do algoritmo denominada 'cognitiva', que reflete a dificuldade em se estimar o impacto perceptual de um elemento de processamento local no resultado global, e como esse impacto é percebido em conjunto com outras fontes sonoras.

Feng et al. (Feng e Liu, 1997), como descrito na Seção 2.4, estimam funções de escala de erro (*error-scaling factor*) lineares para cada tarefa[20] e resolvem um problema de otimização da qualidade do resultado global. Porém, essa transferência de modelos de qualidade local em global naquela

[20] O autores consideram tarefas em grafos lineares da área de compressão de imagens, reconhecimento de fala e rastreamento de radar.

aplicação em imagens não parece viável no contexto desta pesquisa, uma vez que a percepção de um erro ou distorção sonoros produzido em um elemento de uma cadeia de processamento do sinal de áudio depende da interação psicoacústica com as demais componentes audíveis no sinal, especialmente em casos típicos onde várias fontes sonoras são misturadas.

Em outras abordagens de controle da qualidade percebida, vistos no Capítulo 2, usa-se conhecimento sobre o fluxo de dados da aplicação de áudio, como no caso da mixagem seletiva de som (veja a Seção 2.5.6), ou sobre a importância de cada elemento no contexto da aplicação inteira, como é o caso das aplicações de espacialização (veja a Seção 2.5.5). Nesses casos, a qualidade (global) percebida é estimada por testes subjetivos de escuta para certas configurações dos parâmetros de processamento. As relações assim obtidas poderiam ser úteis para certas aplicações no contexto deste trabalho, todavia, o fato de não usarem testes de qualidade automatizados impede o seu uso na abordagem experimental desta pesquisa.

Por estas razões, nesta metodologia opta-se pelo mapeamento do espaço de parâmetros em relação à qualidade do resultado final. Este mapeamento deve ser realizado antes de seu uso em uma aplicação flexibilizada, uma vez que o método PEAQ compara uma versão ideal (e precisa), resultado do processamento sem restrições de tempo de computação, com uma versão computada em tempo real, possivelmente imprecisa. Combina-se esse mapeamento com a função de custos computacionais (veja as Seções 3.1.2.2 e 4.5.3) para se obter um mapeamento direto dos custos computacionais nas qualidades estimadas pelo PEAQ.

3.3.1 Busca por *autotuning*

Um mapeamento completo do espaço de configurações em relação às qualidades estimadas pelo algoritmo PEAQ é inviável devido ao número de configurações possíveis, mesmo com poucos parâmetros de flexibilização, como é o caso da aplicação `flexmix` utilizada na parte experimental desta pesquisa (que possui $> 10^{65}$ configurações distintas, veja o Capítulo 4).

O método proposto para otimizar as configurações de parâmetros que resultam em melhores qualidades percebidas é a afinação automática (*autotuning*) Ansel *et al.* (2013). O `OpenTuner` é um arcabouço criado para sistematizar e popularizar este método de afinação automática, cuja ideia principal é estabelecer experimentalmente, através de simulações, a relação entre os parâmetros do algoritmo e seu desempenho, a fim de construir um modelo de função a ser otimizada.

A necessidade de um tal método experimental aparece por exemplo na compilação de aplicações de alto desempenho em plataformas de hardware distintas, onde a escolha da configuração dos parâmetros de compilação para cada plataforma tem um impacto significativo no desempenho do código executável, e onde a modelagem matemática deste desempenho em função dos parâmetros de compilação é inviável na prática. Ansel et al. (Ansel *et al.*, 2013) mostram que o `OpenTuner` é capaz de melhorar o desempenho destas e outras aplicações (por exemplo em pipelines de processamento de imagens) mais eficientemente do que outros arcabouços de afinação automática. O arcabouço `OpenTuner` permite usar várias estratégias para a busca de configurações, como, por exemplo, a evolução diferencial, buscas gulosas ou *Torczon hillclimbers*. Com a ajuda de meta-técnicas como o *one-armed bandit*, o *OpenTuner* escolhe e executa diversas estratégias de busca, selecionando e priorizando estratégias que encontrem configurações com melhor desempenho (tempo de execução) da aplicação em teste.

Este arcabouço pode ser adaptado ao contexto deste trabalho para que otimize a qualidade de áudio do resultado final, ao invés do desempenho medido através do tempo de execução. Em ambos os casos, não é trivial escrever funções matemáticas explícitas que descrevam a relação entre os parâmetros de configuração e a métrica que se deseja otimizar. Mais especificamente, nesta pesquisa deseja-se estabelecer, através de afinação automática, a relação entre os parâmetros de flexibilização dos elementos e cadeias de processamento de áudio e a qualidade de áudio percebida (PEAQ). A abordagem adaptada para a metodologia de flexibilização consiste na modelagem de uma função objetivo para a otimização que combina a qualidade de áudio e também uma limitação do tempo de execução, que visa permitir o processamento sonoro em tempo real (veja o Capítulo 4).

3.3.2 Gerenciador de flexibilização

O gerenciador de flexibilização pode ser implementado conforme a estrutura descrita na Seção 3.2.1. Na sua implementação, deve-se considerar que o gerenciador

- seja executado em intervalos (ciclicamente),
- tenha acesso aos sensores e
- consiga comunicar as mudanças de configurações aos elementos e cadeias de processamento.

Adicionalmente, é desejável que os custos computacionais do próprio gerenciador sejam baixos, para não comprometer a abordagem da flexibilização.

Em cada execução, o gerenciador deve estimar o tempo de processamento disponível no próximo ciclo a partir do estado do sistema (carga computacional) e encontrar uma configuração da aplicação que, por um lado, respeite o limite de custos e, por outro lado, leve à melhor qualidade percebida possível, utilizando o mapeamento produzido pela auto-afinação, como discutido na seção anterior. O procedimento para a aplicação da configuração selecionada depende principalmente das funcionalidades do arcabouço de áudio utilizado.

3.4 Limitações

A ideia da flexibilização do processamento de áudio visando manipular o *trade-off* entre custo computacional e qualidade tem várias limitações, alguns aspectos das quais serão discutidos nessa seção. Por um lado, existe um limite da própria aplicabilidade dessa ideia em programas e sistemas de processamento de áudio, além de um limite na exequibilidade da implementação. Por outro lado, a ideia encontra seu limite maior quando o *trade-off* entre a qualidade e os custos não é mais útil.

Aplicabilidade Analisando aplicações de processamento de áudio, observa-se que vários cenários de uso não são compatíveis com a abordagem desenvolvida neste trabalho. Trata-se, nesse caso, de aplicações que não permitem uma qualidade variável do resultado, pois o cenário de uso especifica uma qualidade predefinida e fixa. Esse é o caso, por exemplo, de aplicações em produção de áudio tais como gravações para a produção de CDs, além de sistemas de alta fidelidade (*high fidelity*) ou sistemas de medição na área de acústica de edificações.

Exequibilidade da implementação A estratégia de implementação da abordagem proposta é adaptar aplicações e *plugins* existentes da forma menos invasiva possível para facilitar a difusão da proposta. Ao mesmo tempo, esta estratégia pode limitar certas opções de flexibilização, por exemplo, quando a taxa de amostragem e/ou a quantização são fixas dentro da aplicação ou o arcabouço usado não permite a variação desses parâmetros. Outro limite de exequibilidade refere-se à possibilidade de implementar pipelines de processamento no domínio da frequência dentro da aplicação ou do arcabouço, bem como a viabilidade de adaptar o escalonamento das aplicações.

Limites do *trade-off* entre qualidade e custos Através da manipulação de um *trade-off* entre qualidade e custo computacional, a qualidade do resultado pode degradar para além de um limite perceptualmente aceitável, ao mesmo tempo em que não seja possível evitar a situação de sobrecarga. Essa possibilidade é inevitável e intrínseca à abordagem proposta neste trabalho. Para elementos de processamento cujo potencial de ganho em desempenho é pequeno em relação à perda de qualidade, a probabilidade de deparar com esse caso limítrofe é maior.

3.5 Elementos e cadeias específicos

Nesta seção são discutidos alguns exemplos de como abordar a flexibilização de elementos de processamento específicos. Para cada elemento, realiza-se (a) a análise da função no contexto de

processamento de áudio, (b) a análise de seu funcionamento para estabelecer a função dos custos computacionais e (c) propostas de parâmetros de flexibilização e avaliação dos efeitos decorrentes.

Enquanto a flexibilização do elemento de filtragem (filtro FIR) é implementada e testada na parte experimental deste trabalho, os demais elementos, sobretudo os da área de espacialização, incluindo sua análises e propostas, são discutidos de forma teórica, incluindo e contextualizando abordagens encontradas na literatura e suas adaptações para um ambiente de processamento flexibilizado. Esses elementos não são incluídos na parte experimental deste trabalho, por um lado por ainda não existirem métricas automatizáveis (objetivas) para a qualidade percebida e, por outro lado, pela ausência de implementações adequadas, com código-fonte aberto, que incluam a permissão para publicar alterações, necessária para este estudo. Apesar disso, considera-se que a discussão teórica destes elementos constitui uma contribuição deste trabalho, além de apontar o caminho para possíveis trabalhos futuros no contexto de processamento flexível de elementos de espacialização.

Destaca-se que, além do filtro FIR discutido a seguir, na parte experimental vários outros elementos mais simples que fazem parte da aplicação de teste (`flexmix`) também são discutidos quanto à sua flexibilização, como por exemplo o elemento de reamostragem (veja a Seção 4.4.1 e seguintes).

3.5.1 Filtro de resposta ao impulso finita

Filtros digitais são muito usados em várias fases do processamento de áudio. Tipos comuns são filtros passa-baixas, passa-altas e passa-faixa usados para ajustar sinais na produção musical. Particularmente, filtros de resposta ao impulso finita são ligados à área da auralização e são usados para imprimir efeitos acústicos em sinais, incluindo propriedades espaciais (Vorländer, 2008).

Especialmente, a convolução de um sinal de áudio com uma resposta ao impulso é uma técnica central no contexto da simulação de salas e pode ser interpretada como a aplicação de um filtro FIR. Como visto nas Seções 2.5.4 e 2.5.5, a aplicação do filtro transfere para o sinal de áudio características de um ambiente (visto como um sistema LTI) ligadas a efeitos cognitivos que ajudam na orientação espacial de um ouvinte, e que têm influência em métricas psicoacústicas (inteligibilidade da fala, imersão, etc.).

A seguir, são discutidos filtros de resposta ao impulso finita (FIR), suas implementações principais e a flexibilização proposta por este trabalho.

Os filtros FIR são definidos por

$$y_n = \sum_{i=-\infty}^{\infty} a_i x_{n-i}, \tag{3.10}$$

onde x é o sinal de entrada, y é o sinal de saída e a é o vetor dos coeficientes que caracterizam o filtro, sendo que $a_i \neq 0$ apenas para uma coleção finita de índices. Considerando-se uma coleção de M índices e que $a_i \neq 0$ para $i = 0, \ldots, M-1$, tem-se a equação finita

$$y_n = \sum_{i=0}^{M-1} a_i x_{n-i}; \tag{3.11}$$

na interpretação da equação acima como uma *convolução linear* com M coeficientes; valores de x_{n-i} inexistentes (por exemplo, anteriores ao início efetivo do sinal) são tratados como 0; outra interpretação possível é a da *convolução circular*

$$y_n = \sum_{i=0}^{M-1} a_i x_{(n-i) \bmod M}, \tag{3.12}$$

onde todos os índices são interpretados em aritmética módulo M, correspondendo à interpretação de que todos os sinais são periódicos.

No domínio da frequência, as respectivas Transformadas de Fourier X, Y e A estão relacionadas pelo teorema da convolução, que no contexto dos sinais de tempo infinito assume a forma

$$Y(f) = A(f)X(f), \ \forall f, \tag{3.13}$$

associada à convolução linear (Equação 3.10), onde f representa a frequência em uma unidade arbitrária (Hz, rad/amostra, etc.), ou a forma

$$Y_k = A_k X_k, \ k = 0, \ldots, M-1, \tag{3.14}$$

no caso dos sinais finitos, associada à convolução circular no domínio do tempo (Proakis e Manolakis, 1996) (Equação 3.11), porque a discretização no domínio da frequência tem como consequência um rebatimento temporal (*temporal aliasing*).

Filtros desse tipo podem ser implementados em duas formas principais: diretamente no domínio do tempo, ou no domínio da frequência. A escolha da implementação depende de considerações sobre custos computacionais e outros fatores como, por exemplo, limitações por causa da arquitetura do hardware.

3.5.1.1 Convolução no domínio do tempo

```
1  // x[]: bloco das amostras de entrada
2  // y[]: bloco das amostras de saída
3  // ak[]: valores dos coeficientes
4
5  for each sample index i:
6    add_to_history (x[i])
7    for each coef index k:
8      y[i] += ak[k] * get_history_at (k)
```

Algoritmo 3.2: *Aplicação de um filtro FIR a um sinal: convolução linear.*

A implementação no domínio do tempo decorre diretamente da Equação 3.11 e pode ser vista no Algoritmo 3.2 que mostra a convolução calculada para um bloco de amostras. A função `add_to_history` adiciona o valor da amostra atual ao vetor das entradas passadas (*history*, os valores x_{n-i} na Equação 3.11); a função `get_history_at` devolve o valor da amostra com índice k no vetor das entradas passadas. Ambas as funções têm custos computacionais constantes ($\mathcal{O}(1)$). Os custos computacionais são dominados pelos custos da multiplicação e adição na linha 8 do Algoritmo 3.2. Para um filtro com M coeficientes, essa linha será executada M vezes. Logo tem-se, para um sinal de n amostras, custos computacionais de:

$$Z_{\text{fir}} = c_0 + c_1 n + c_2 nM, \tag{3.15}$$

onde c_0, c_1 e c_2 são constantes que dependem dos tempos de execução de trechos do código e que podem ser estimadas experimentalmente.

3.5.1.2 Convolução no domínio da frequência

A implementação no domínio da frequência baseia-se nas Equações 3.13 e 3.14 que correspondem à multiplicação frequência a frequência dos espectros do sinal e do filtro. As diferenças de interpretação entre as respectivas equações no domínio do tempo (Equações 3.10 e 3.11) podem ser compatibilizadas através de uma técnica de extensão com zeros, como descrita a seguir. A convolução linear de dois vetores x e a de comprimentos L e M, respectivamente, produz um vetor de saída y de comprimento $L+M-1$, que é maior do que o sinal de entrada x. Assim, podemos estender ambos os vetores com zeros até o comprimento final $L+M-1$ e depois aplicar a convolução circular através da multiplicação finita no domínio da frequência, obtendo um espectro Y de comprimento $L+M-1$ que pode ser transformado de volta para o sinal y no domínio do tempo.

No caso de um sinal de entrada de comprimento arbitrário (por exemplo, entrada em tempo real), deve-se segmentá-lo em blocos de tamanho L e combinar os vetores resultantes para obter o resultado correto da convolução linear. Uma forma possível de combinar estes vetores é sobrepor e somar os últimos $M-1$ valores do resultado de um bloco aos primeiros $M-1$ valores do próximo bloco, numa estratégia conhecida como *overlap-add* (Oppenheim *et al.*, 1999).

```
1  // x[]: bloco de valores das amostras de entrada (inclusive entradas passadas)
2  // y[]: bloco de valores das amostras de saída
3  // a[]: valores dos coeficientes do filtro
4
5  para cada bloco de amostras x[]:
6    x_ext[] = estender x[] com zeros éat o comprimento M + N - 1
7    a_ext[] = estender a[] com zeros éat o comprimento M + N - 1
8    X[] = DFT de x_ext[]
9    A[] = DFT de a_ext[]
10   para cada elemento X[i]:
11     Y[i] = X[i] * A[i]
12   y_ext[] = IDFT de Y[]
13   overlap-add y[] e y_ext[]
```

Algoritmo 3.3: *Convolução linear a partir da convolução circular no domínio da frequência.*

O algoritmo 3.3 mostra uma possível implementação para a convolução linear de um bloco usando a convolução circular no domínio da frequência, onde DFT significa Transformada de Fourier Discreta (*Discrete Fourier Transform*) e IDFT a inversa da DFT. Porém, existem implementações mais eficazes como mostra, por exemplo, Bianchi (Bianchi, 2013) que usam a redundância e simetria presentes em passos intermediários do cálculo. Para filtros FIR constantes, cujos coeficientes não mudam com o tempo, os custos computacionais calculam-se pela fórmula

$$Z_{\text{fftfir}} = c_0 + c_1 n + c_2 n \log_2 (c_3 n)\,, \tag{3.16}$$

onde n é o número de amostras e $c_i,\ i = 0, \ldots, 3$ são constantes cujos valores são desconhecidos. Os custos da transformação do filtro para o domínio da frequência estão encaixados na constante c_0, uma vez que os coeficientes não mudam.

Na aplicação prática mostra-se que a convolução no domínio do tempo é menos eficiente que a implementação no domínio da frequência quando o número de coeficientes do filtro é grande. Moore (Moore, 1990, pág. 144), por exemplo, afirma que a convolução rápida (FFT) é mais eficiente para filtros com mais que 30 coeficientes[21]. Porém, os custos computacionais reais dependem da arquitetura do hardware e de outros fatores, e deveriam ser estimados experimentalmente para cada plataforma e casos de uso.

Respostas impulsivas associadas a ambientes normalmente têm muitas amostras, o que significa que o filtro respectivo também tem muitos coeficientes. Consequentemente, esses filtros normalmente são computados no domínio da frequência. Todavia, a técnica introduz um atraso no sinal do tamanho do bloco usado na transformação. Para aplicações nas quais um atraso menor é desejado, algoritmos mistos foram desenvolvidos, por exemplo por Gardner (Gardner, 1995) e outros. Os algoritmos propostos executam uma convolução no domínio do tempo para computar um primeiro segmento de amostras sem atraso, enquanto para o restante da computação usa-se a convolução no domínio do tempo.

3.5.1.3 Flexibilização

Esta pesquisa enfoca a convolução no domínio do tempo para demonstrar o processo de flexibilização de um elemento de processamento de áudio. Como visto na Equação 3.15, os custos computacionais do algoritmo são dominados pelo número de amostras e pelo número de coeficientes do filtro. Porém, como discutido na Seção 3.1.1, o número de amostras não é considerado como

[21] Na implementação do `Gstreamer` encontra-se um valor similar, de 32 coeficientes para a troca entre a convolução direta e a rápida.

parâmetro potencial de flexibilização de elementos de processamento isolados, e sim das cadeias de processamento (através da reamostragem).

Considerando o número de coeficientes do filtro como flexível, coloca-se a questão de como produzir uma família de filtros de tamanhos diferentes mas que representem aproximadamente o mesmo processamento do ponto de vista perceptual. Alguns trabalhos anteriores sobre a redução de custos computacionais em processamento de áudio, ainda que fora do contexto de flexibilização aqui proposto, foram bastante úteis para esta pesquisa e são discutidos a seguir.

Existem várias abordagens para reduzir os custos computacionais. Por exemplo, Schafer e Rabiner (Schafer e Rabiner, 1973) discutiram técnicas de interpolação do filtro no domínio de frequência. Boudreaux e Parks (Boudreaux e Parks, 1983) propuseram filtros com poucos valores não-nulos (*sparse filters*). Essa abordagem é expandida em Baran, Wei e Oppenheim (Baran *et al.*, 2010) com métodos de programação linear para a computação dos coeficientes.

A seguir, discutiremos três abordagens de flexibilização do número de coeficientes mostrando suas vantagens e desvantagens. Na parte experimental deste trabalho, essas abordagens são comparadas quanto aos custos computacionais e também do ponto de vista da percepção de qualidade.

Flexibilização dos últimos coeficientes A primeira abordagem examinada consiste em eliminar coeficientes no fim do filtro. Essa estratégia é razoável em casos onde a resposta do filtro ao impulso tem magnitude tendendo a zero, como acontece, por exemplo, em respostas impulsivas de salas. A vantagem desta abordagem é a implementação simples, e como se pode ver na parte experimental, o melhor desempenho em comparação com os outros métodos. Como discutido na Seção 3.2.2, no contexto da avaliação de qualidade pela métrica PEAQ não basta considerar perdas exclusivamente do ponto de vista espectral, porém a função de transferência do filtro é útil para estimar perdas de qualidade em filtros similares.

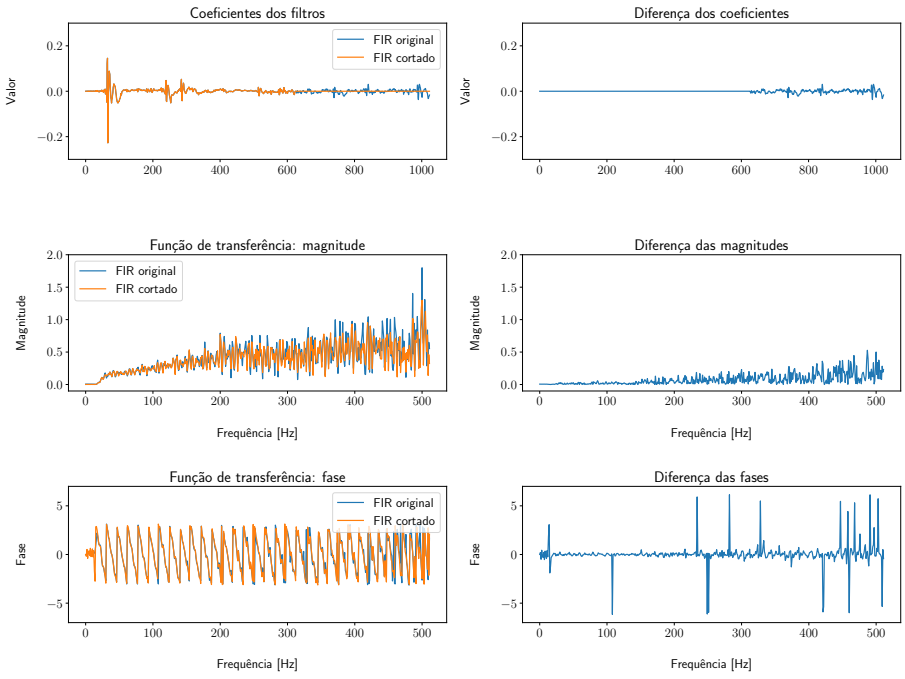

Figura 3.5: *Comparação de dois filtros FIR: versão original e versão cortada.*

No lado esquerdo da Figura 3.5, pode-se ver uma resposta impulsiva com 1024 coeficientes e uma versão encurtada com 824 coeficientes, junto com seus respectivos espectros de magnitude e de fase; no lado direito pode-se ver as diferenças entre os filtros quanto aos coeficientes e espectros. Vale lembrar que um corte nos últimos coeficientes introduz um janelamento retangular do sinal e, consequentemente, uma mudança espectral que é tanto mais nítida quanto maior o número de coeficientes eliminados. Adicionalmente, no caso de respostas impulsivas de salas o corte de

coeficientes pode impactar a percepção de localização, pois picos na resposta impulsiva indicam a ocorrência de reflexões acústicas que são importantes para a orientação espacial de um ouvinte (veja as Seções 2.5.4 e 2.5.5). No exemplo dado na Figura 3.5, aparecem reflexões entre as amostras de índice 50 e 100 e duas menores entre 200 e 400, que neste exemplo foram preservadas pelo corte.

Flexibilização dos coeficientes com menor valor Para superar a limitação do método anterior quanto às reflexões e tentar conservar os picos da resposta impulsiva no domínio do tempo, propõe-se uma segunda estratégia para variar o número de coeficientes. Com essa abordagem, reduz-se o número de coeficientes do filtro anulando-se coeficientes de menor valor absoluto, independentemente de onde ocorram. Adicionalmente, é realizado um ajuste de escala na resposta impulsiva a fim de preservar a energia total. Na Figura 3.6 esta estratégia é aplicada ao mesmo filtro do exemplo anterior.

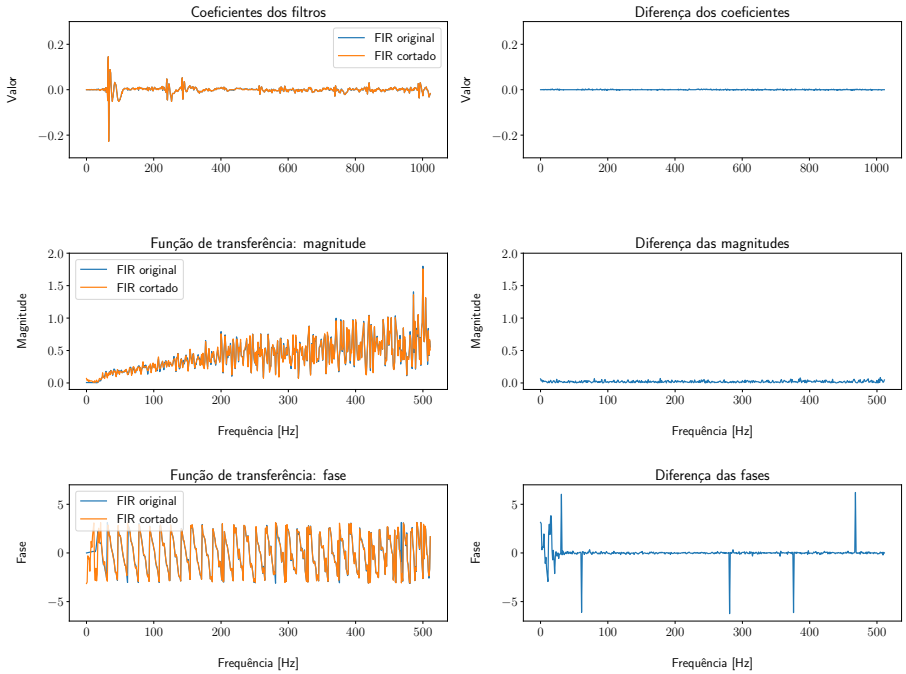

Figura 3.6: *Comparação de dois filtros FIR: versão completa e versão reduzida: menores valores absolutos zerados.*

Flexibilização usando otimização linear A terceira abordagem aqui proposta foi desenvolvida na tentativa de preservar todo o espectro do filtro, inclusive o espectro de fase. Ela é baseada na abordagem de Baran et al. (Baran *et al.*, 2010) para a geração iterativa de versões esparsas para filtros FIR, a partir dos dois passos abaixo:

1. elimina-se o coeficiente de menor valor absoluto; e
2. define-se um problema de otimização linear a fim de ajustar os coeficientes remanescentes do filtro para ajustar o espectro do filtro esparso ao espectro do filtro original.

Enquanto Baran et al. restringem seu método a filtros FIR com fase linear através do ajuste dos espectros de magnitude, aqui estende-se a técnica deles a filtros FIR em geral considerando o ajuste dos espectros complexos. No Apêndice A é apresentada a derivação do problema linear correspondente [22].

[22] Para resolver os sucessivos problemas de otimização usa-se a biblioteca `scipy.optimize` (linguagem `Python`) com o método `HiGHS simplex`. Uma descrição detalhada do método encontra-se em docs.scipy.org/doc/scipy/reference/optimize.linprog-highs.html#optimize-linprog-highs

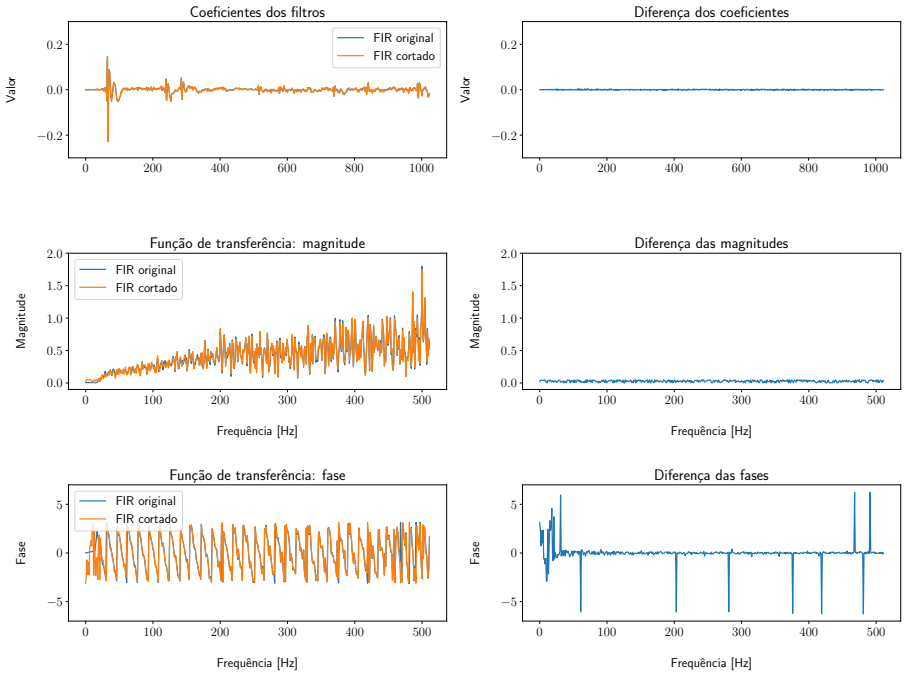

Figura 3.7: *Comparação de dois filtros FIR: versão completa e versão otimizada com menos coeficientes.*

Devido aos altos custos computacionais, não é possível usar o método de otimização em tempo real durante a execução da aplicação de áudio. Consequentemente, as versões do filtro precisam ser pré-computadas para toda a faixa de valores dos parâmetros de flexibilização (número de coeficientes e taxa de amostragem). Na implementação apresentada na Seção 4.4.8, todas as variantes do filtro são carregadas na memória durante a inicialização do elemento de filtragem, para garantir um acesso rápido em tempo de processamento. Na Figura 3.7 pode ser visto o resultado desta estratégia, aplicada ao mesmo filtro do exemplos anteriores.

As três abordagens apresentadas permitem que o número de coeficientes possa ser usado como parâmetro de entrada para o algoritmo flexibilizado de convolução, que trata as respostas impulsivas eficientemente como vetores esparsos. A função de custos da Equação 3.16 descreve a relação entre o número de coeficientes remanescentes, M, e o tempo necessário para a execução do algoritmo em cada uma das três abordagens.

Na parte experimental deste trabalho é apresentada uma comparação entre os três métodos, do ponto de vista da qualidade percebida (PEAQ), a partir de uma aplicação com um único elemento correspondente a uma convolução com um filtro FIR. Porém, é relevante lembrar que a estimação da qualidade pelo PEAQ se refere ao resultado final de uma cadeia de elementos de processamento e não se aplica à avaliação de elementos isolados. Uma avaliação de qualidade pelo PEAQ em uma cadeia incluindo filtros FIR flexibilizados é apresentada no Capítulo 4 no contexto da aplicação experimental `flexmix`, onde várias fontes sonoras são filtradas por filtros FIR e mixadas.

3.5.2 Simulação de salas

Nesta seção discute-se como abordar a flexibilização de algoritmos de simulação de sala. Devido aos custos computacionais relativamente altos dessas simulações, vários métodos de redução desses custos foram propostos e desenvolvidos por muitos autores, como visto nas Seções 2.5.4 e 2.5.5.

3.5.2.1 Parametrização e Flexibilização no modelo das salas

Para apropriar o modelo geométrico da sala aos algoritmos, as superfícies são decompostas em planos. Superfícies de ordem maior, como cilindros, são decompostas em pequenos planos, e a preocupação é não gerar detalhes demais, de modo que o modelo seja simplificado, em concordância

com os objetivos da aplicação e com a capacidade computacional. Certos algoritmos exigem que os modelos geométricos sejam preprocessados para transformá-los em outras representações, ou requerem bancos de dados com valores pré-calculados, diminuindo assim o tempo do processamento no momento da simulação em tempo real (por exemplo *mashs* (De Moor *et al.*, 2003) ou *binary space partitioning - BSP* (Schröder, 2011)).

A estratégia para a parametrização deve ser, consequentemente, a de trabalhar com vários modelos que tenham diferentes graus de detalhamento e custos computacionais. Embora existam poucas estratégias de simplificação automática de modelos de salas (Thakur *et al.*, 2009), é sempre possível gerar esses modelos manualmente, e alternar dinamicamente os diferentes modelos em tempo de execução, dependendo da situação de sobrecarga do sistema.

Uma relação universal (em forma de equação) entre a complexidade do modelo geométrico da sala e os custos computacionais para o processamento desse modelo em geral não está disponível, devido à complexidade da interação entre o modelo e os algoritmos de processamento. Todavia, é possível estabelecer uma relação empírica para casos específicos.

3.5.2.2 Parametrização e Flexibilização na simulação de salas

Além do uso de modelos geométricos aproximados (com complexidades diferentes) de salas, uma outra estratégia de flexibilização pode ser controlar a duração da resposta impulsiva calculada. Essa duração da resposta impulsiva corresponde ao parâmetro $t_{\max}$ na Seção 2.5.4. Essa possibilidade depende da estrutura concreta da aplicação, porque muitas aplicações usam o modelo de fontes virtuais apenas para uma primeira parte da resposta impulsiva, e usam outros algoritmos, como por exemplo *ray tracing*, para a parte tardia da resposta impulsiva. Nesse caso, a flexibilização deve adaptar também os outros algoritmos usados na construção da resposta impulsiva.

3.5.2.3 Parametrização e Flexibilização no traçado estocástico de raios

Vorländer (Vorländer, 2008) determina o tempo de execução do algoritmo como:

$$Z = N t_{\max} \overline{n} \tau, \tag{3.17}$$

onde N é o número total de partículas, t_{max} é o comprimento da resposta impulsiva desejada, $\overline{n}$ é o número médio de reflexões que a partícula sofre e τ são os custos dos testes por reflexão ou por detecção, que podem ser descritos com a seguinte equação:

$$\tau = n_w t_w + n_d t_d + t_c, \tag{3.18}$$

onde n_w é o número de planos a serem testados por reflexão, t_w é o tempo necessário para executar um teste de reflexão (*point-in-poligon test*), n_d é o número de detectores, t_d é o tempo de execução de um teste de colisão de uma partícula num detector e t_c são os custos de gerenciamento de dados, independentemente do número de paredes e detectores.

Como visto na Seção 3.5.2.1 sobre modelos de salas, existem pré-processamentos que transformam o modelo de sala – um conjunto de planos – em estruturas mais adequadas para o processamento rápido da simulação. Por exemplo, o particionamento de espaço binário (*binary space partitioning*) divide os espaço recursivamente para estabelecer uma representação desse espaço em forma de árvore (*tree*) que permite reduzir o número de testes por reflexão (n_w) e o número de testes por detecção (n_d). Consequentemente, os custos computacionais dos testes são modificados, como pode ser visto na seguinte equação:

$$\tau = \log_2 n_w \; t_w + \log_2 n_d \; t_d + t_c. \tag{3.19}$$

Para os custos computacionais totais do algoritmo do traçado de raios estocástico obtém-se:

$$Z = N t_{\max} \overline{n} (\log_2 n_w \; t_w + \log_2 n_d \; t_d + t_c). \tag{3.20}$$

A partir desta função de custos, podem ser considerados três parâmetros úteis para a flexibilização:

1. O parâmetro n_w, que depende da complexidade do modelo de sala, pode ser variado como discutido na Seção 3.5.2.1.

2. O parâmetro $t_{\max}$, que define o comprimento da resposta impulsiva, pode ser flexibilizado; vale lembrar que a maioria das aplicações que computam a resposta impulsiva no contexto de tempo real implementam um algoritmo híbrido que usa o modelo de fontes virtuais (2.5.4.1) para a parte inicial da resposta impulsiva e o traçado de raios estocástico apenas para a parte tardia[23]. A flexibilização do comprimento da resposta impulsiva influencia vários parâmetros psicoacústicos que dependem especialmente da parte tardia como, por exemplo, a clareza de fala e música (*speech and music clarity*) e o envolvimento (*envelopment*) (Vorländer, 2008). Correspondentemente, esses parâmetros têm que ser avaliados quando o comprimento da resposta impulsiva é flexibilizado.

3. O número de partículas (N), que controla a resolução da resposta impulsiva computada, pode ser variado, considerando que um aumento do número de partículas reduz a flutuação estocástica (por cálculo da média) e aumenta a reprodutibilidade. No entanto, um número grande de partículas não reduz os erros inerentes do modelo (geometria da sala simplificada, modelo da acústica geométrica, etc.), ou seja, a flexibilização deste parâmetro deve ser vista sempre em conjunto com os demais parâmetros flexibilizados.

3.5.3 Espacialização

3.5.3.1 Parametrização e Flexibilização na síntese de campo sonoro

Os custos computacionais da síntese de campo sonoro (veja a Seção 2.5.5) podem ser calculados, considerando-se a Equação 2.14 (função do *driver*). Para um alto falante e um bloco de amostras de tamanho n, obtém-se:

$$Z_{\text{speaker}}(n) = P\ (c_F + c_A + c_S)\ n, \tag{3.21}$$

onde P é o número de fontes sonoras (objetos sonoros), c_F representa os custos da aplicação do filtro $F(\omega)$, c_A os custos da computação da atenuação $A(x)$ e c_S os custos da sobreposição dos sinais das fontes virtuais. Enquanto os custos c_A e c_S são constantes por amostra (uma multiplicação e uma adição), os custos do filtro c_F (um filtro passa-baixas) dependem do número de coeficientes, se o filtro for aplicado no domínio do tempo, ou da taxa de amostragem, caso o filtro seja aplicado no domínio da frequência (veja as considerações da Seção 3.5.1 sobre os filtros FIR e as considerações de Baalman e coautores (Baalman *et al.*, 2007) sobre a implementação de um sistema de WFS).

Os custos totais para um sistema de WFS com A alto-falantes podem ser expressos pela seguinte função:

$$Z_{\text{WFS}}(n) = A\ P\ (c_F + c_A + c_S)\ n. \tag{3.22}$$

Os parâmetros utilizados em uma estratégia de flexibilização podem ser o número de alto-falantes A e o número de fontes virtuais (objetos sonoros) P.

Relativamente à flexibilização do número de alto-falantes A, deve-se considerar que este número está relacionado à distância entre os alto-falantes e, consequentemente, ao rebatimento espacial. O sistema não é capaz de reproduzir corretamente as fontes virtuais que contêm faixas espectrais acima de uma frequência limiar, pois sinais com frequências acima deste valor aparecem como artefatos sonoros que têm uma orientação espacial incorreta. Esta frequência é na média $f_{\text{nyquist}} = c/2\delta x$ onde c é a velocidade do som e δx representa a distância entre os alto-falantes (Berkhout *et al.*, 1993). Isso significa que o número mínimo de alto-falantes utilizados deve respeitar a condição de reprodução das frequências relevantes contidas no sinal a ser espacializado. Uma flexibilização deste parâmetro deve considerar, adicionalmente, os artefatos produzidos

[23] Nesse caso o parâmetro $t_{\max}$ define o comprimento da resposta impulsiva inteira.

nos momentos de transição entre configurações, e respectivas estratégias para a minimização de seus efeitos perceptuais.

A flexibilização do número P de fontes virtuais que são reproduzidas pelo sistema pode adaptar as estratégias seguidas para sistemas de síntese binaural discutidas na Seção 2.5.5.3 e na próxima seção (3.5.3.2).

3.5.3.2 Parametrização e Flexibilização na síntese binaural

Os custos computacionais da síntese binaural podem ser calculados, considerando-se as equações 2.15 e 2.16:

$$Z_{\text{binaural}}(n) = P\ (c_{\text{ar}} + c_{\text{HRTF}} + c_s)\ n, \tag{3.23}$$

onde c_{ar} são os custos da aplicação da atenuação H_{ar}, e c_s os custos da sobreposição das fontes sonoras. Os custos da aplicação da HRTF, c_{HRTF}, variam dependendo de qual procedimento for usado. Esse procedimento pode ser uma convolução ou um outro método como, por exemplo, o uso de filtros IIR como aproximações às HRTFs. Dependendo da implementação tem-se custos lineares (veja a Seção 3.5.1.1) para a implementação da síntese usando a convolução no domínio do tempo, ou custos logarítmicos (veja a Seção 3.5.1.2) para a implementação no domínio da frequência; de todo modo é possível parametrizar o algoritmo usado seguindo as estratégias da Seção 3.5.1.

É necessário lembrar que a equação 2.16 é válida apenas para posições fixas da fonte sonora e da orelha do ouvinte, e para um ângulo fixo de incidência; se as posições ou o ângulo mudarem, uma nova HRTF deve ser computada. Para isso, as aplicações que implementam a síntese binaural normalmente incluem um banco de dados com HRTFs para vários ângulos de incidência diferentes. Todavia, não é factível armazenar HRTFs para todos os ângulos possíveis; os bancos de dados publicamente disponíveis têm uma resolução angular maior que o mínimo ângulo audível (Montesião de Sousa, 2010). Por essa razão torna-se necessário um método de interpolação que gera HRTFs aproximadas para ângulos não considerados no banco de dados, a partir de HRFTs existentes no banco de dados. Nos últimos anos, foram desenvolvidos vários métodos de aproximação ou interpolação dependendo do formato usado para representar as HRTFs na aplicação. Uma boa visão geral sobre HRTFs, suas propriedades, medição e emprego são apresentados por Hammershøi e Møller (Hammershøi e Møller, 2005), enquanto Souza (Montesião de Sousa, 2010) apresenta técnicas de interpolação no domínio do tempo e no domínio da frequência, incluindo a aproximação de HRTFs com filtros de resposta infinita (*infinite impulse response* (IIR)). Os custos dessas interpolações e das transições entre as HRTFs (mixagem) crescem com resoluções angulares altas e também quando fontes se movem rapidamente na cena. Nessa relação existe um potencial de folga que pode ser explorado mudando-se a resolução angular da síntese, o que causa, no entanto, mudanças do ponto de vista perceptual.

Número de fontes virtuais Uma flexibilização do número de fontes virtuais (P) pode ser realizada através de duas estratégias.

A primeira opção é desconsiderar algumas fontes a partir de critérios perceptuais como Tsingos e coautores propuseram ((Tsingos *et al.*, 2003) e (Tsingos, 2005)). Esta estratégia avalia quais fontes iriam contribuir significativamente para o sinal de saída usando critérios de mascaramento (audibilidade) e volume (*loudness*), como visto na Seção 2.5.5.4. O *limiar de mascaramento* pode, portanto, ser um candidato à flexibilização, pois com esse limiar seria possível controlar o número de fontes excluídas, dependendo da carga computacional, assim como também são candidatas à flexibilização as cotas concedidas a cada *cluster* (conjunto de fontes sonoras, veja a Seção 2.5.5.4 e Figura 2.8).

A segunda opção para tornar flexível o número de fontes baseia-se na ideia de agrupar fontes que serão tratadas como uma única fonte (*clustering*). Desta forma, os custos computacionais podem ser controlados forçando a formação de *clusters*. Herder (Herder, 1999a), por exemplo, propôs e implementou a ideia de juntar fontes virtuais distantes em *clusters* dependendo da proximidade das fontes entre si, da direção e da velocidade das mesmas. Posteriormente, os sinais são somados

obtendo-se um sinal representante do *cluster*, cuja posição é calculada como média das posições das fontes incluídas no *cluster*. Herder afirma que essa abordagem é capaz de ajustar a síntese às limitações de recursos computacionais, todavia resulta em uma imagem espacial (*spatial image*) diferente da obtida sem essas limitações. Nessa abordagem, o número de *clusters* que são construídos pode ser utilizado como parâmetro de flexibilização.

Capítulo 4

Avaliação experimental

Os objetivos da fase experimental são, por um lado, demonstrar a viabilidade das propostas da presente pesquisa e, por outro, estabelecer um ambiente de experimentação em si (*test bed*) que permita aprofundar a pesquisa usando outras aplicações de processamento de áudio, outras métricas de qualidade ou outros elementos de processamento.

O ambiente de experimentação é composto pelos seguintes componentes principais: (a) a aplicação de processamento a ser testada, (b) ferramentas que estimam métricas de qualidade e (c) ferramentas de otimização.

Na Seção 4.4 são apresentados experimentos que objetivam a exploração dos elementos de processamento quanto aos custos computacionais, seus potenciais de flexibilização e outras características tais como a interpretação e conciliação dos resultados com a análise teórica.

O código fonte desenvolvido para experimentos está disponível junto com a tese no banco das teses da biblioteca da Universidade de São Paulo (teses.usp.br/[..]).

4.1 Escolha das ferramentas

A disponibilidade das ferramentas a serem usadas é de importância fundamental, não apenas para a reprodutibilidade da pesquisa, mas também para sua utilização tanto em contextos práticos quanto teóricos. Porém, a disponibilidade de software depende da plataforma computacional usada, do sistema operacional e, até mesmo, da disponibilidade de bibliotecas. Isso pode vir a ser um problema no decorrer do tempo, uma vez que as versões de cada componente de software mudam; adicionalmente, mudam também as interfaces de programação (*Application Programming Interface* (Interface de programação de aplicações)s (APIs)). Uma configuração experimental que pode parecer padrão em um momento pode deixar de ser padrão em pouco tempo e levar a problemas de compatibilidade entre os componentes; um exemplo é a mudança de arquitetura do processador dos computadores da Apple[1] em conjunto com mudanças profundas do sistema operacional. No caso de software distribuído apenas na forma compilada, estes problemas se agravam, uma vez que nesse formato todas as dependências estão *congeladas* em certas versões de sistemas operacionais e bibliotecas, sobretudo porque que na forma compilada o software roda apenas em uma arquitetura de hardware, como por exemplo, em uma CPU da Intel ou com arquitetura ARM. Além disso, a maioria do software está disponível apenas para poucos sistemas operacionais. No decorrer desta pesquisa, muito do software encontrado só existe para versões antigas e/ou obsoletas de sistemas operacionais ou de plataformas de hardware; em outros casos, o software depende de bibliotecas não mais disponíveis.

Ressalta-se ainda que, nos artigos científicos, muitas vezes, os algoritmos são pouco documentados, fato que pode prejudicar ou inviabilizar a reprodutibilidade das pesquisas; de qualquer forma, isso obriga pesquisadores a reprogramar algoritmos para problemas já resolvidos ou abordados, o que representa um desperdício de recursos. Por outro lado, software cujo código-fonte está disponível

[1] Apple II: MOS 6502 → Apple Macintosh: Motorola 68k → Power Macintosh: AIM PowerPC → IntelMac: x86/x64 → Apple M1

livremente tem a vantagem de permitir sua modificação e recompilação, e assim, sua adaptação a novas configurações tanto de hardware quanto de software.

Adicionalmente, Pinto (Pinto, 2006), que investigou o uso e a produção de software livre em universidades públicas, mostra as vantagens do acesso aberto a resultados científicos e destaca "[sic] o caráter cooperativo da pesquisa científica, a importância da transparência e neutralidade no acesso ao *commons* da Ciência e a natureza anti-rival da informação científica."

Na tentativa de aliviar os problemas acima mencionados e com o intuito de tentar garantir a disponibilidade da configuração experimental no futuro, nesse trabalho foi utilizado apenas software de código aberto ou software livre, ainda que isso tenha limitado a escolha em alguns casos. O software produzido no processo desta pesquisa, igualmente, foi licenciado como software livre. Os resultados numéricos e as ferramentas de medição desenvolvidas também estão incluídos no pacote publicado.

Linux foi o sistema operacional usado como plataforma na fase experimental em todas as implementações. Essa escolha não tem sua razão na disponibilidade do código-fonte do sistema mas sim nos termos da licença de software (*GNU General Public License* (Licença Pública Geral GNU) (GPL)) do sistema operacional e dos ambientes de desenvolvimento que permitem seu uso livre e ágil. A disponibilidade desses componentes para diversos tipos de hardware viabiliza a transferência e aplicação do conhecimento para novas plataformas. Por isso, o software desenvolvido nesta pesquisa deve ser facilmente portado para outros sistemas operacionais, desde que as bibliotecas necessárias estejam disponíveis.

As fontes sonoras (arquivos de áudio) usadas para a experimentação foram escolhidas por sua relevância neste trabalho para o caso de uso investigado (apresentado na Seção 4.5) e pela possibilidade de acesso livre por outros pesquisadores. O site *Cambridge Music Technology*[2] do engenheiro de som Mike Senior oferece arquivos de som para o treinamento de pessoas na área de mixagem de som e conta com uma grande e ativa comunidade de discussão. Os sons musicais e as respostas impulsivas oferecidos não podem ser usados para fins comerciais, mas admitem o uso educacional[3].

4.2 Escolha do arcabouço de áudio

A principal função do arcabouço de áudio dentro desta pesquisa é ser a base das implementações e experimentações, ou seja, a base da prova de conceito. Decorrentemente, o arcabouço precisa ser adequado e relevante para a área de processamento de áudio no contexto da produção musical, uma vez que a metodologia desenvolvida nesta pesquisa enfoca essa área.

Aqui, o critério de software livre coloca-se como uma necessidade, uma vez que o método desta pesquisa consiste em adaptar códigos-fonte existentes ao conceito de flexibilização proposto, ao invés de reimplementar algoritmos de processamento de áudio a partir de suas descrições teóricas encontradas na literatura.

Além dos supracitados critérios gerais, foram considerados aspectos técnicos ligados ao processamento flexível de áudio. Um arcabouço de áudio viabiliza ou inviabiliza a modificação de certos parâmetros fundamentais do processamento como, por exemplo, a taxa de amostragem ou o tamanho e o formato das amostras, o que restringe as opções disponíveis para aplicação da metodologia do processamento flexível.

Complexidade da implementação Do ponto de vista desta pesquisa, os arcabouços diferem quanto ao grau de complexidade que requerem na implementação de certos algoritmos, que são necessários para demonstrar a viabilidade da metodologia, ou seja, um arcabouço pode apoiar ou dificultar as implementações. Por exemplo, a princípio é possível implementar funções para a comunicação dos elementos entre si ou dos elementos com a aplicação. No entanto, um arcabouço

[2] www.cambridge-mt.com/ms/mtk
[3] FAQ: www.cambridge-mt.com/ms/mtk-faq

que já ofereça essa funcionalidade parece mais adequado, especialmente quando os *plugins* existentes já possuem essa capacidade de comunicação.

Magnitude do *trade-off* alcançável Da análise dos elementos e cadeias de processamento surgem as opções de parametrização. Entretanto, a implementação de mecanismos para o controle dos parâmetros depende da disponibilidade de recursos do arcabouço. O arcabouço pode facilitar ou restringir o controle do fluxo de dados, o uso de formatos dos dados ou mesmo mudanças dinâmicas no processamento ao vivo. A magnitude do *trade-off* alcançável, ou seja, a relação entre o ganho de desempenho e a perda de qualidade ou vice-versa, é menor no caso de parâmetros que não possam ser facilmente controlados através do arcabouço. Isso é válido especialmente para parâmetros com maior potencial de contribuição ao *trade-off* , como por exemplo a taxa de amostragem.

Adequação do arcabouço Um arcabouço pode ser mais adequado que outro em função de sua própria estrutura, de seu funcionamento geral e de sua API, devido à menor complexidade que evoca na implementação, ou devido à possibilidade de alcançar um melhor *trade-off* .

Portanto, foram considerados os seguintes arcabouços e/ou ambientes:

- o *Jack audio connection kit*,
- a combinação de um servidor de áudio (por exemplo uma DAW) com *plugins* do padrão LV2 e
- o arcabouço `Gstreamer`.

Como descrito na Seção 2.6, o servidor de som do *Jack* não permite a mudança da taxa de amostragem em tempo de execução, e o formato dos dados é restrito. Mesmo que, a princípio, a introdução de outros formatos fosse possível (o servidor *jackd* é agnóstico quanto aos formatos ou tamanho dos dados), a parametrização do formato e da taxa de amostragem, no sentido proposto neste trabalho, parece inviável.

A combinação de uma DAW com *plugins* do padrão LV2 pode ser um ambiente a investigar, pois o padrão LV2 permite flexibilizações dos parâmetros da taxa de amostragem e do tipo de dados, assim como possibilita a comunicação das componentes da aplicação. Todavia, a implementação para 'flexibilizar' a taxa de amostragem representa um abuso da API, de acordo com o manual que sugere o uso de uma mesma taxa de amostragem em todas as componentes[4].

Diferentemente dos demais, o arcabouço `Gstreamer` possui uma infraestrutura rica para a adaptação dinâmica. O `Gstreamer` permite a negociação dinâmica (em tempo de execução) de formatos de dados, de taxas de amostragem, de número de canais entre os *plugins*. Além disso, oferece mecanismos de comunicação entre as componentes e destas com a aplicação. Os autores da implementação do algoritmo PEAQ, que foi usada ao longo desta pesquisa, afirmam que a flexibilidade do arcabouço quanto à configuração de cadeias de processamento foi a razão de sua implementação com `Gstreamer` (Holters e Zölzer, 2015).

Optou-se pelo uso do arcabouço `Gstreamer` para as implementações experimentais pois permite um *trade-off* maior que os outros arcabouços, bem como a maior capacidade de reconfiguração dinâmica. Ademais, a complexidade da implementação da metodologia proposta é menor, uma vez que o arcabouço facilita a abordagem seguida neste trabalho quanto às funcionalidades de comunicação e à capacidade de controle dinâmico dos parâmetros de flexibilização.

Mostra-se que as linguagens de programação discutidas na Seção 2.5.7 não se encaixam facilmente nos arcabouços de processamento de áudio porque requerem componentes incompatíveis com o funcionamento dos arcabouços de processamento de áudio ou necessitam adaptações complexas. Por exemplo, a linguagem `FLEX` requer um *run-time environment* adicional para estimar os

[4] A taxa de amostragem é definida durante a instanciação de um *plugin* e não pode ser modificada durante a execução (veja a documentação lv2plug.in/book). Para a mudança da taxa de amostragem, um mecanismo de substituição do *plugin* tem que ser implementado, que deve tratar a parada de processamento no momento de reconfiguração, as de- e reconexões do elemento com outros elementos e vários outros aspectos.

tempos de execução que deveria ser integrado com um arcabouço de áudio. A linguagem `Timed C` é relacionada com as estratégias *sieve* ou *milestone* da computação imprecisa enquanto a estratégia seguida neste trabalho é a das versões múltiplas.

4.3 Métricas utilizadas

4.3.1 Modelo de custos computacionais

Na aplicação do método de processamento flexível, um modelo de custos computacionais da aplicação em teste faz-se necessário, tanto para prever tempos de execução, quanto para viabilizar seu controle efetivo. Destaca-se que a análise dos algoritmos envolvidos não deve ser apenas qualitativa, mas também quantitativa, no sentido que o modelo (função de custos) deve mapear os parâmetros específicos dos elementos de processamento aos tempos de execução previstos, ou outras métricas equivalentes observáveis.

A análise dos algoritmos deve considerar (a) os tempos de execução das operações executadas e (b) a frequência de execução das operações (Knuth, 1998). Knuth, por exemplo, usa uma linguagem de programação (MIX) para escrever os algoritmos e analisá-los. Essa linguagem foi desenhada para preservar as características e estruturas do hardware nos tipos de dados e operações da linguagem. Para a análise, os custos computacionais são contabilizados em unidades inteiras, com contadores associados a cada tipo de operação elementar. O tempo de execução de cada uma destas operações é visto como fixo, mas depende do computador no qual o algoritmo for implementado, e por isso deve ser estimado experimentalmente.

No contexto contemporâneo, usando hardware recente[5], observam-se alguns problemas quanto à adaptação do método acima. O número de instruções disponíveis e a complexidade das instruções no nível da CPU aumentaram, bem como a complexidade do micro-código, especialmente nas arquiteturas *Complex Instruction Set Computer* (Computador com um conjunto de instruções complexas) (CISC). Desta forma, mesmo o código de máquina nessas arquiteturas representa uma abstração maior do hardware do que aquela considerada por Knuth, por exemplo nas instruções elementares que operam simultaneamente sobre linhas ou colunas de uma matriz (instruções *Single Instruction Multiple Data* (Arquiteturas de instrução única sob dados múltiplos) (SIMD)[6]. Apesar de eventuais otimizações realizadas nas instruções SIMD, pode-se considerar que mudam o conjunto de operações e os custos correspondentes a cada operação, porém a estrutura da função de custos não muda.

Outro problema observado na adaptação do método de análise de custos de Knuth é que o tempo de execução de uma mesma operação elementar não deveria ser visto como constante pelas seguintes razões: (a) a frequência na qual a CPU opera varia dependendo de fatores externos (*CPU frequency scaling*), e (b) o tempo de carregamento dos dados de entrada e salvamento dos resultados depende do estado do *cache* (L1, L2, etc.), inviabilizando a previsão exata desses tempos.

Sem desprezar a relevância destas observações, os resultados dos experimentos executados no contexto desta pesquisa torna evidente a possibilidade de estimar valores médios para os tempos de execução das operações. Essas estimativas permitem a previsão dos custos computacionais dos algoritmos de processamento de áudio com precisão suficiente para o controle do processo de flexibilização. Por essas razões, o modelo de análise de custos computacionais de Knuth será aplicado aos algoritmos de processamento de áudio no contexto desta pesquisa.

As análises de algoritmos apresentadas nas próximas seções dividem-se em três passos:

1. análise do código-fonte e construção de uma função de custos;

2. mensuração de tempos de execução em função dos parâmetros de entrada; e

[5] Foca-se aqui principalmente CPUs para servidores e PCs da Intel e da AMD, em correspondência com os objetivos deste trabalho.

[6] SIMD denomina um conjunto de operações de uma CPU que permitem a execução paralela em um núcleo; com estas operações executa-se uma operação com vários dados simultaneamente.

3. mapeamento e ajuste da função teórica aos resultados.

Essas análises são simplificadas, no contexto desta pesquisa, pelo fato de que o arcabouço `Gstreamer` é escrito em `C`. A principal vantagem da linguagem `C` nesse contexto é utilizar comandos próximos das operações básicas da linguagem da máquina, permitindo uma contabilidade direta de operações. Isso contrasta com aplicações escritas em linguagens que são executadas em máquinas virtuais como, por exemplo, Java, Python e Octave/Matlab, que exigem análises de todos os componentes do ambiente de execução para uma correta estimação de custos.

Na próxima seção é apresentada uma investigação das métricas disponíveis, de seus comportamentos sob diferentes condições do sistema computacional, bem como da utilidade destas métricas para a estimação dos custos computacionais.

4.3.1.1 Mensuração de tempo

A seleção de um sistema operacional baseado no *time-sharing*, que permite o uso interativo, requer uma investigação do comportamento das métricas sob diferentes estados do sistema em teste, para subsequentemente definir o uso dessas métricas no contexto desta pesquisa. Com este objetivo, considera-se três experimentos preliminares que serão apresentados e discutidos a seguir. Os experimentos abordam:

- a relação das métricas com a frequência de trabalho da CPU e a carga computacional do sistema,
- a relação das métricas com o número de núcleos (ou de CPUs) e
- a influência das políticas de escalonamento do sistema operacional sobre os tempos de execução.

O primeiro experimento preliminar consiste em consecutivas execuções de uma aplicação de processamento de áudio, cujos tempos de execução, entre outras métricas relacionadas aos custos computacionais, são registrados. A aplicação usada no experimento realiza um pipeline de processamento de áudio típica, semelhante aos pipelines das Seções 4.4 e 4.5. As tarefas do processamento da aplicação de teste são: (a) leitura de um arquivo de áudio, (b) decodificação dos dados, (c) amplificação do sinal, (d) reamostragem do sinal, (e) codificação em formato wav, e (f) gravação dos dados em um arquivo. A fim de excluir a latência imprevisível do acesso aos discos (rígidos ou tipo SSD), a leitura e gravação de arquivos da aplicação será restrita à memória RAM (utilizando um RAM *filesystem*)[7].

A aplicação de teste é executada 200 vezes, sendo que a cada 50 execuções são alteradas as configurações da CPU e a demanda por recursos computacionais (estresse). As configurações aplicadas podem ser vistas na Tabela 4.1. A ferramenta `stress-ng`[8] é usada para controlar a demanda adicional aos recursos computacionais, especialmente o tempo de execução na CPU e o acesso à memória.

[7] No entanto, os experimentos da Seção 4.4.5 mostram que os arquivos são armazenados na memória cache (RAM) após a primeira leitura direta do disco SSD (mediado pelo sistema de arquivos), sendo que o tempo desta primeira leitura é descartado para efeito de mensuração. No caso da gravação, um efeito cache também pode ser observado, uma vez que o sistema operacional armazena um arquivo na memória RAM antes de ser gravado no disco em uma operação de sincronização postergada.

[8] github.com/ColinIanKing/stress-ng

Número do experimento	Frequência da CPU	Carga computacional adicional do sistema
0 - 49	fixa	mínima
50 - 99	fixa	alta
100 - 149	variável	alta
150 - 199	variável	mínima

Tabela 4.1: *Estado do ambiente durante a execução do experimento para a avaliação de métricas.*

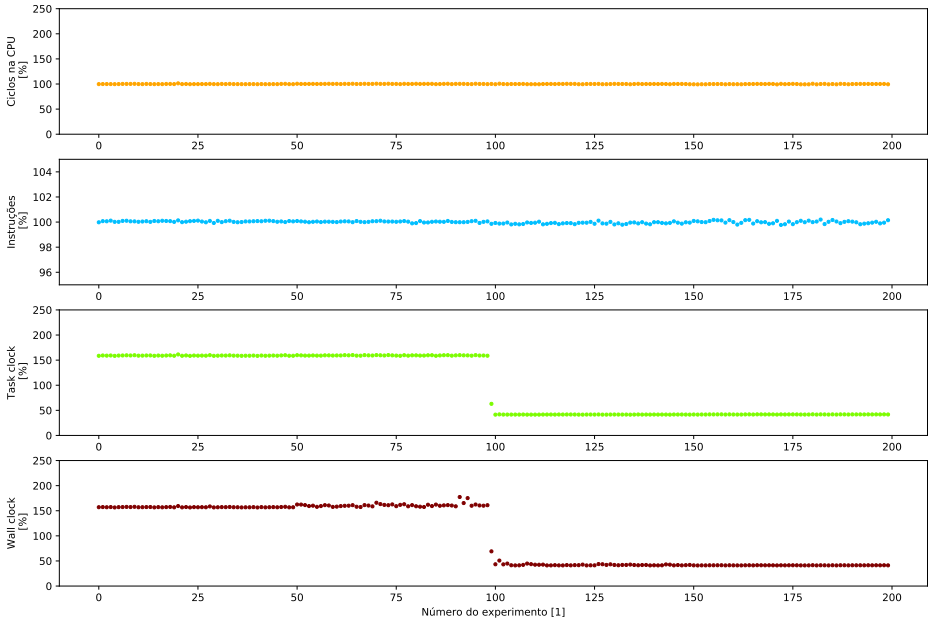

Figura 4.1: *Relação das métricas com os estados do sistema computacional (veja a Tabela 4.1) - sem HT.*

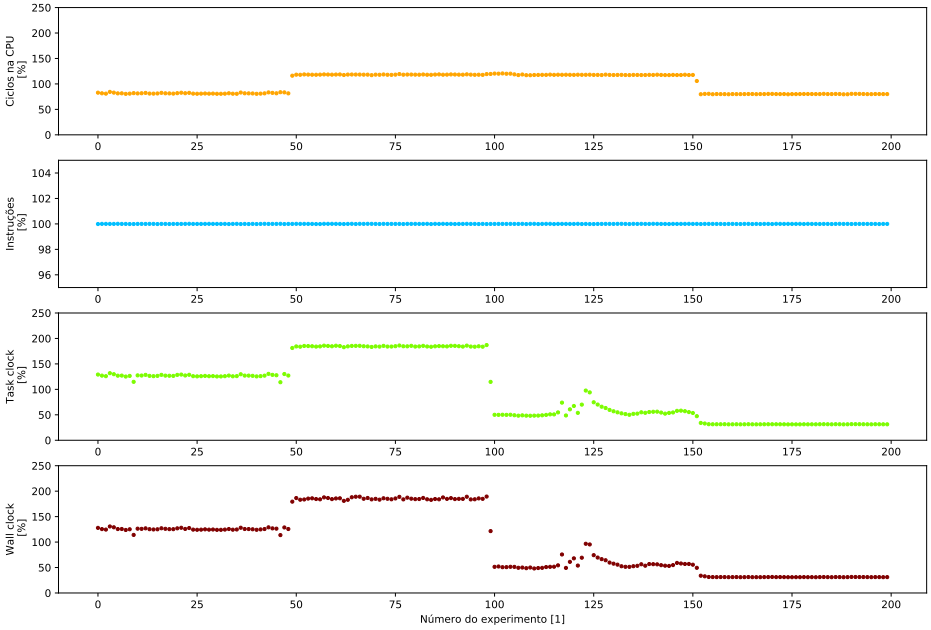

Figura 4.2: *Relação das métricas com os estados do sistema computacional (veja a Tabela 4.1) - com HT.*

As Figuras 4.1 e 4.2 mostram os resultados do experimento executado em um sistema com uma CPU com *Hyper Threading Technology* (HT)[9] e em um sistema com uma CPU sem HT. Para cada execução da aplicação, foram obtidos:

- o número de ciclos nos quais a aplicação ocupou a CPU;
- o número de instruções executadas (*retired*) no contexto da aplicação;
- o tempo de execução da aplicação do ponto de vista da CPU (*task clock*);
- o tempo de execução do ponto de vista do usuário (*wall clock*).

Os números de ciclos e de instruções acima incluem aqueles associados ao código do experimento, bem como tudo o que é executado pelo sistema operacional em função da aplicação.

A apresentação dos valores mensurados (e normalizados) em função da rodada do experimento permite visualizar efeitos intermitentes (localizados temporalmente), considerando que a aplicação de teste é executada consecutivamente com intervalo pequeno entre execuções. Observam-se variações significativas no comportamento das métricas, tanto de diferentes métricas no mesmo ambiente de computação, quanto das mesmas métricas em ambientes diferentes (com ou sem HT).

O número de ciclos que o programa ocupa na CPU (o primeiro diagrama nas Figuras 4.1 e 4.2) se apresenta aproximadamente constante em todos os experimentos, quando a aplicação de teste é executada em um sistema sem HT. Isso ocorre porque o número de ciclos é proporcional ao número (fixo) de instruções da aplicação em execução, sendo que pequenas flutuações podem ser causadas pelos tempos de espera por requerimentos a recursos externos à CPU, como por exemplo a memória RAM. No caso do ambiente computacional com HT, o número de ciclos aumenta enquanto o sistema está sob estresse (experimentos 50-149), e isso pode ser explicado através do funcionamento

[9]HT é uma arquitetura de CPUs; definição: www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading; apresentações mais detalhadas podem ser encontradas em Marr et al. (Marr *et al.*, 2002) e Tanenbaum (Tanenbaum, 2009, pág. 20).

do mecanismo de *hyperthreading*[10]. No experimento, a carga computacional adicional aplicada (estresse), diminui o acesso da aplicação de teste aos recursos de execução, aumentando assim o número de ciclos que a aplicação ocupa na CPU (parte dos quais em espera).

O número de instruções executadas (veja o segundo diagrama nas Figuras 4.1 e 4.2) também permanece aproximadamente constante sob todas as condições testadas, porque a tarefa computacional (código e dados de entrada do experimento) não é alterada. Entretanto, pequenas flutuações de valores podem ser observadas devido à reação do programa às mudanças no estado do sistema operacional, por exemplo, o estado do sistema de arquivos. Nesse sentido, o estado do sistema operacional pode ser entendido como uma parte flutuante dos dados de entrada do experimento.

Enquanto o *task clock* (o terceiro diagrama nas Figuras 4.1 e 4.2) representa o tempo de execução da aplicação em uma CPU, o *wall clock* (o quarto diagrama nas Figuras 4.1 e 4.2) registra o tempo de espera do usuário pela conclusão da aplicação; esse tempo é sempre maior que o *task clock* porque inclui o tempo de espera causada por outros processos que competem pelos mesmos recursos computacionais (Kerrisk, 2010). Por isso, observa-se uma variação do tempo *wall clock* maior que aquela do *task clock* enquanto o sistema computacional está sob estresse de outros processos.

Quando a opção de controle dinâmico da frequência da CPU (*frequency scaling*)[11] está ativa (experimentos 100-199), ambas as métricas de tempo de execução mostram valores menores em comparação com o controle dinâmico de frequência inativo, devido ao aumento do valor da frequência da CPU permitido pelo *frequency scaling*. Em sistemas com HT observam-se maiores variações dos tempos de execução, em função do controle dinâmico de frequência da CPU, em comparação com sistemas sem HT, devido à amplificação dos efeitos da frequência variável com HT.

No segundo experimento preliminar investiga-se a relação das métricas com o número de núcleos da CPU[12] disponíveis para o processamento da aplicação de teste, considerando que, atualmente, grande parte das aplicações de processamento de áudio são usadas em sistemas com mais de um núcleo.

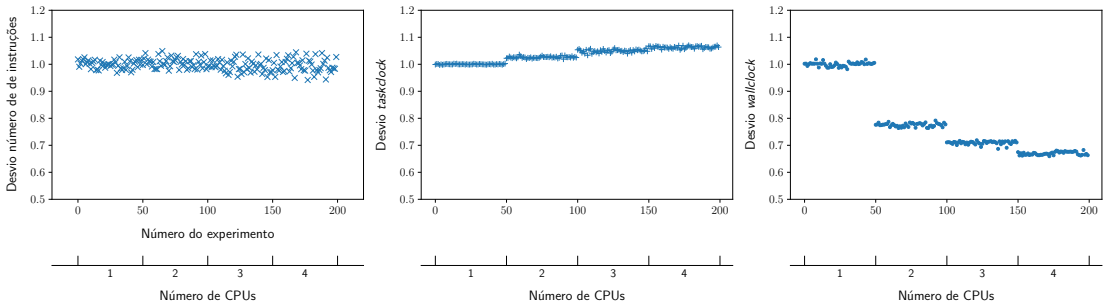

Figura 4.3: *Relação das métricas com o número de núcleos. (Todas as medidas são relativas à média dos valores obtidos com 1 núcleo.)*

Analogamente ao primeiro experimento, a aplicação de teste foi executada 200 vezes e a cada 50 rodadas o número de núcleos foi aumentado. Na Figura 4.3 pode-se ver o resultado para as diferentes métricas. Vê-se que o número de instruções executadas é quase constante, no entanto observa-se maiores variações nas execuções com mais núcleos. Isso ocorre porque em cada execução

[10]Com esse mecanismo podem ser executadas duas aplicações (*threads*) 'paralelamente' em uma CPU, uma vez que uma CPU pode memorizar duas cópias de estados de aplicações (*architectural state*), apesar de continuar executando apenas uma aplicação a cada vez, visto que possui apenas um conjunto de recursos de execução (*physical execution resources*). A CPU consegue alternar a execução entre os *threads*, por exemplo, para explorar o tempo de espera por uma leitura da memória de um *thread* enquanto avança o outro (Marr *et al.*, 2002). Esse mecanismo leva ao uso mais eficaz das estruturas físicas da CPU.

[11]Por exemplo, a 'Tecnologia Avançada de Passorrápido da Intel' (*Enhanced Intel SpeedStep Technology*).

[12]Nesta pesquisa usa-se o termo núcleo independentemente de sua realização em hardware, que pode ser com vários núcleos por chip de CPU ou com vários chips. Essa diferença não é tão relevante para essa pesquisa, visto que as diferenças arquitetônicas levam apenas a diferenças na eficácia da paralelização, mas não na complexidade computacional resultante da paralelização.

o processamento é distribuído entre os núcleos, e portanto depende de muitos fatores dinâmicos considerados pelo sistema operacional, tais como a existência de outros processos, suas prioridades, etc. Isso leva aos mais variados caminhos de execução[13] quando há mais núcleos, portanto a custos computacionais variáveis.

Quanto aos tempos de execução em CPU (*taskclock*), observa-se um pequeno aumento quando a aplicação de teste é executada com mais núcleos, devido aos custos computacionais dos mecanismos necessários de coordenação e sincronização entre os núcleos, em função da distribuição do processamento. Isso ocorre porque no caso *multicore* essa métrica reflete a soma dos tempos de execução nos núcleos e dos mecanismos de paralelização.

O tempo de execução do ponto de vista do usuário (*wallclock*) é o tempo que ele espera até a conclusão da aplicação; esse tempo é menor quando mais núcleos são usados, fato esperado, uma vez que mais recursos computacionais estão disponíveis. Conforme a Lei de Amdahl (Rodgers, 1985), o grau de diminuição desse tempo em função do número de núcleos depende não apenas do hardware mas também do grau de paralelizabilidade da aplicação em teste.

Considerando os resultados dos primeiros dois experimentos, propõe-se para esta pesquisa o uso das seguintes métricas:

- o número de instruções executadas para confirmar as funções teóricas de custos;
- o *taskclock* para a estimação dos custos dos elementos de processamento; e
- o *wallclock* para a avaliação dos custos da aplicação, considerando a paralelização.

Mesmo que a mensuração do número de instruções executadas se correlacione fortemente com as previsões teóricas e, neste experimento, se mostre estável sob condições diferentes, não é possível estimar um tempo de execução sem conhecer o tempo de execução de cada instrução.

O número de ciclos em que a aplicação reside na CPU poderia ser usado da mesma forma que o número de instruções executadas, porém somente em sistemas *sem* HT. Poder-se-ia calcular os tempos de execução (*task clock*) a partir do número de ciclos, porém seria necessário conhecer a duração (variável) de cada ciclo, o que diminui a utilidade dessa métrica.

Ressalta-se que a mensuração do tempo de execução na CPU (*task clock*) pode produzir dados úteis para o planejamento das tarefas porque este tempo expressa os custos reais dos algoritmos. No entanto, para englobar a execução paralelizada em sistemas com mais de um núcleo, deve-se considerar o tempo de execução do ponto de vista do usuário (*wallclock*).

Quanto à configuração do sistema computacional, os resultados do primeiro experimento sugerem o uso de um sistema *sem* HT, com a opção de escalonamento da frequência da CPU *desligada*, ou seja, com uma frequência da CPU fixa para a mensuração dos tempos de execução. Quanto à seleção de uma frequência fixa, vale lembrar que há mecanismos do sistema computacional para modificar a frequência em função da temperatura, no intuito de evitar o superaquecimento da CPU. Esses mecanismos são controlados pelo BIOS e/ou pelo sistema operacional. Por isso, sugere-se o uso de uma frequência baixa da CPU visando um menor aquecimento e, consequentemente, evitando que tais mecanismos sejam acionados, o que levaria a distorções dos resultados.

O primeiro experimento também demonstra a necessidade de evitar qualquer carga computacional adicional durante as mensurações dos tempos de execução, o que entretanto não é necessário para a contagem de instruções executadas.

O terceiro experimento preliminar explora características de políticas de escalonamento e sua influência nos tempos de execução mensurados. A aplicação de teste é executada 100 vezes sucessivas para cada política de escalonamento. Destas, as primeiras 50 execuções do experimento ocorrem em um ambiente computacional configurado com poucos processos adicionais em execução, enquanto as 50 execuções subsequentes, no mesmo ambiente computacional, foram expostas a estresse (`stress-ng` com prioridade padrão).

[13] Aqui entendidos como realizações de execução (*execution paths*).

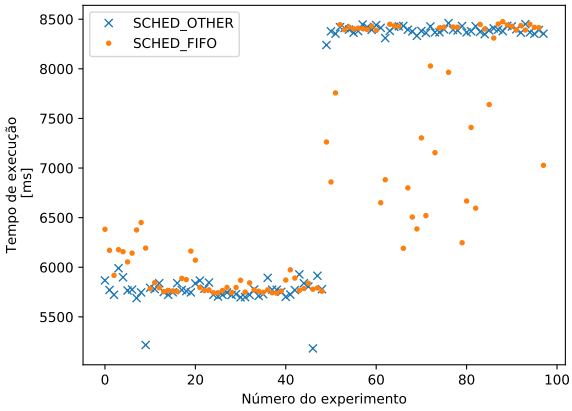

Figura 4.4: *Tempos de execução sob diferentes políticas de escalonamento de processos.*

A Figura 4.4 apresenta os resultados desse experimento cujo diagrama mostra os tempos de execução da aplicação de teste usando políticas de escalonamento diferentes. Em Linux, o sistema operacional usado nesta pesquisa, cada processo tem uma política de escalonamento associada que é usada pelo escalonador para a priorização dos processos. Há várias políticas de escalonamento, dentre elas, `SCHED_OTHER` que é a política padrão, e `SCHED_FIFO` e `SCHED_RR` para processos com alta prioridade. A execução de um processo sob a política `SCHED_OTHER` o submete a um escalonamento do tipo *round robin* com tempo de CPU compartilhado (*time-sharing*) e prioridades dinâmicas, política essa usada para a maioria dos processos, principalmente para processos contendo interações com o usuário. `SCHED_FIFO` e `SCHED_RR` são políticas que permitem a execução em tempo real na definição da norma POSIX.1b (*soft realtime*).

Um processo sob a política `SCHED_FIFO` só pode ser interrompido para a execução de processos com uma prioridade (de tempo real) ainda maior, enquanto a política `SCHED_RR` permite/introduz *time-sharing* entre processos com a mesma prioridade (*round robin*).

Observa-se nos resultados que, no sistema sob estresse, os tempos de execução aumentam significativamente, no caso de `SCHED_OTHER` para todas as execuções, enquanto no caso `SCHED_FIFO` os tempos de execução aumentam em graus diferentes levando a um maior espalhamento dos valores. Entretanto, em um sistema de tempo real, esperar-se-ia que os tempos de execução não aumentassem. Isso demonstra que o sistema operacional usado não concede prioridade absoluta, nem mesmo para processos que usem as opções (*facilities*) de tempo real da norma POSIX.1b, confirmando assim a inabilidade deste sistema para garantir os tempos-limite de conclusão de tarefas (como discutido na Seção 2.4). Para esta pesquisa, extraem-se duas conclusões:

- para as mensurações, deve ser sempre considerada a carga computacional adicional ocorrente no sistema de medição visando diminuí-la;
- reitera-se a necessidade da conciliação entre os requerimentos de recursos dos processos ocorrentes no sistema, dado que o sistema operacional não consegue evitar influências adversas por outros processos no processamento de áudio.

Sob a política de escalonamento `SCHED_FIFO`, um processo só pode ser interrompido por processos com maior prioridade, portanto esse processo pode causar um bloqueio do sistema, principalmente, porque não tem que ceder a CPU depois de um intervalo de tempo limite, como é o caso sob outras políticas. Esse problema nota-se em especial quando um processo chama funções do sistema operacional que acessem o hardware periférico, pois essas chamadas podem bloquear a execução desse processo devido à espera pela resposta de tal hardware periférico e, com isso, bloquear a execução dos demais processos do sistema.

Por esta razão, indica-se o uso da política `SCHED_RR` que permite as mesmas prioridades que `SCHED_FIFO`, porém evitando-se que situações de bloqueio ocorram, pois o processo é interrompido

e removido da CPU depois de um certo tempo (quantum) para permitir a execução de processos com a mesma ou maior prioridade[14].

4.3.2 Avaliação da qualidade de áudio percebida

Como discutido na Seção 2.3, existem vários conceitos de avaliação da qualidade percebida para aplicação em diversos contextos e com objetivos bem distintos. Para a parte experimental desta pesquisa, optou-se por adaptar o método PEAQ (veja a Seção 2.3.2) pelas seguintes razões:

- a área de aplicação do PEAQ é a avaliação da degradação de sinais de áudio de alta definição, a mesma desta pesquisa;
- a avaliação por PEAQ é automatizável, ao contrário de experimentos envolvendo avaliadores humanos;
- a existência de implementações do PEAQ livres de patentes, publicadas com licenças de software livres.

Por causa das incertezas quanto às definições de certos parâmetros contidos no algoritmo PEAQ (Holters e Zölzer, 2015; Kabal, 2003), suas implementações não estão completamente em conformidade com a norma, como discutido adiante. Ainda assim, predomina em favor da escolha do PEAQ a inexistência de outros métodos padronizados automatizáveis aplicáveis a sinais musicais de alta definição, tornando-o uma pré-condição fundamental desta pesquisa.

Atualmente existem várias implementações comerciais e de software livre do PEAQ, tais como:

- PQevalAudio de Kabal (Kabal, 2003) em `Matlab` e `C`[15],
- GstPEAQ[16] de Holters e Zölzer (Holters e Zölzer, 2015),
- `peaqb`, de Gottardi e Akinori[17], implementando apenas a versão básica do método,
- versão comercial da empresa Opticom[18].

A implementação da Opticom oferece conformidade com a norma, mas foi desconsiderada nesta pesquisa por possuir uma licença comercial e código fechado (veja as considerações na Seção 1.3).

Em seu artigo, Holters et al. (Holters e Zölzer, 2015) mostram que nenhuma implementação de software livre está em conformidade com a norma ITU-R BS.1387, o que motivou os autores a escrever uma versão própria (`GstPEAQ`). Essa é a melhor implementação em software livre de acordo com a conformidade com a norma e com os tempos de execução, ainda que ela também não possa ser considerada em plena conformidade com a norma. Além disso, essa implementação oferece duas versões com desempenhos distintos, uma básica e outra avançada, e encaixa-se facilmente no arcabouço usado nesta pesquisa que é o `Gstreamer`. Por todas essas razões, a implementação `GstPEAQ` foi utilizada nesta pesquisa para todas as avaliações da qualidade de áudio percebida.

Além da avaliação da qualidade de áudio percebida com o PEAQ, usa-se nesta pesquisa uma medida objetiva de distorção ponderada (ponderação A), para complementar os experimentos com um método simples e conhecido na área da psicoacústica. Por sua simplicidade e familiaridade, essa medida evidencia relações simples que permitem distinguir versões similares de sinais de áudio.

Os demais algoritmos pesquisados, capazes de realizar a avaliação de certas propriedades perceptuais, não parecem viáveis para esta pesquisa pelas razões a seguir (por exemplo: (Piotr *et al.*, 2014) ou (Egger, 2013)). Embora sejam desejáveis a aplicação e o uso dos métodos para estimar efeitos espaciais, tais como os disponibilizados, por exemplo, na *Auditory Modeling Toolbox*, não

[14] Uma descrição aprofundada apresenta Kerrisk (Kerrisk, 2010, pág. 737). Ver também no manual do sistema com: `man sched`.

[15] www-mmsp.ece.mcgill.ca/Documents/Software

[16] github.com/HSU-ANT/gstpeaq

[17] github.com/akinori-ito/peaqb-fast, continuação do projeto: sourceforge.net/projects/peaqb

[18] www.opticom.de/technology/peaq.php

é claro como uma diferença na percepção espacial se traduz em uma diferença de percepção de *qualidade*. Além disso, a *Auditory Modeling Toolbox* tem seu foco na pesquisa e no entendimento do processo de audição, assim como em oferecer métodos para estimar posições de fontes sonoras únicas, e não para a avaliação de percepções espaciais de várias fontes sonoras, o que seria um requisito para a aplicação nesta pesquisa.

4.3.3 Ambiente de experimentação

Os dados técnicos do hardware para a mensuração de desempenho (custos computacionais) são:

- computador pessoal (PC) / workstation;
- Intel Xeon CPU L5420 (arquitetura x86_64), 2,50 GHz tendo 4 *cores*, sem *Turbo Boost*[19];
- Placa mãe: IPM41-D3 (Itautec ST4254) RAM: 8 GiB DDR2 (1066 MHz);
- Discos rígidos: Samsung SSD 860 (SATA3 / 1,5 GB).

O software principal usado na fase de experimentação é caracterizado a seguir:

- sistema operacional: Linux (kernel 5.11);
- distribuição: Fedora GNU/Linux release 32;
- compilador: gcc versão 10.2, glibc versão 2.31;
- ALSA 1.2.3.2, gstreamer versão 1.16.

Tanto para a simplificação da análise quanto devido às observações feitas na discussão das métricas (Seção 4.3.1.1), a CPU foi configurada sem a opção de *frequency scaling*, com a frequência fixada em 2,0 GHz. À luz da discussão na Seção 4.3.1.1, as aplicações de teste foram executadas com alta prioridade, sob a política `SCHED_RR` e em um ambiente com carga mínima, para melhorar a reprodutibilidade dos experimentos bem como para diminuir influências não controláveis nos resultados.

A ferramenta `perf` A ferramenta `perf` [20] usa o subsistema dos contadores de desempenho (*performance counters*) do Linux kernel para gerar valores estatísticos globais sobre a execução de aplicações.

Dentre as estatísticas disponíveis, as seguintes são usadas nesta pesquisa:

- o número de instruções executadas,
- o tempo de execução do processo (relógio do processo *process time*)[21],
- o tempo de execução em tempo real (*wall-clock time*)[22].

As ferramentas específicas para `Gstreamer` Para a investigação de aplicações baseadas no arcabouço `Gstreamer` há duas opções principais. Por um lado, pode-se inserir elementos como `cpureport` ou `progressreport` na aplicação de teste, para obter informações no tempo de execução da aplicação. Por outro lado, pode-se utilizar as ferramentas da coleção *gst-instruments*, que gravam as informações no tempo de execução para uma análise posterior.

As principais informações usadas nesta pesquisa são:

[19] *Turbo Boost* refere se a um método (tecnologia implementada pela Intel) em que a velocidade de um ou mais cores de uma CPU pode ser aumentada além dos limites permitidos para o longo prazo, porém isto é possível apenas por um tempo curto evitando assim a danificação do hardware por excesso de calor. Esse método ajuda com plataformas interativas que apresentam/manifestam/dispõem picos curtos de computação.

[20] A ferramenta pertence ao software ferramental do sistema operacional (*kernel tools*: veja www.kernel.org

[21] Usa-se um relógio do tipo `CLOCK_PROCESS_CPUTIME`, que contabiliza o tempo que um processo ocupa a CPU, incluindo os tempos de execução das chamadas de sistema.

[22] Com este tipo de relógio (`CLOCK_MONOTONIC`) conta-se o tempo total transcorrido.

- os tempos totais de execução da aplicação[23];
- os tempos de execução dos elementos[24];
- o grafo do processamento que é usado para gerar as figuras dos pipelines neste capítulo.

Linguagens de programação O arcabouço `Gstreamer` é implementado principalmente na linguagem C, e disponibiliza APIs para C++ e Python. Todas as implementações de *plugins* e aplicações investigados neste trabalho foram escritas principalmente em C, apesar de alguns dos algoritmos analisados conterem código em linguagens de programação diferentes, como por exemplo a linguagem ORC[25] e código de máquina (SIMD extensions).

A execução dos testes e experimentos foi controlada por *scripts* na linguagem Python. Usou-se também *Scripts* em Python junto com as bibliotecas de programação linear para a avaliação dos resultados, bem como para os passos de otimização do filtro FIR (Seção 4.4.8). O arcabouço OpenTuner, usado na otimização da Seção 4.5, também é baseado em Python.

4.4 Custos computacionais de elementos de processamento de áudio

Nesta seção investiga-se certos elementos de processamento de áudio para detalhar suas características e seus comportamentos quanto ao desempenho em função dos parâmetros de processamento. Adicionalmente, discute-se o uso das diferentes métricas para a avaliação de desempenho, e a relação entre estas métricas e as equações teóricas. Os elementos de processamento escolhidos são aqueles elementos usados para compor a *aplicação* flexibilizada (`flexmix`), apresentada e analisada na Seção 4.5.

Os elementos considerados nesta seção são:

- elemento de amplificação (`GstAudioAmplify`[26]),
- elementos de equalização (`GstIirEqualizer3Bands`, `GstIirEqualizer10Bands`),
- elemento de compressão dinâmica[27] (`GstAudioDynamic`),
- elementos de leitura de arquivos (`GstFileSrc`, `GstFlacParse`, `GstFlacDec`),
- elemento de reamostragem (`GstAudioResample`),
- elemento de mixagem (`GstAudioMixer`) e
- elemento de filtragem do tipo FIR (`GstFlexfir`)

A apuração individual do desempenho de cada elemento não seria estritamente necessária para uma análise detalhada da aplicação `flexmix`, uma vez que existem outras ferramentas que permitem esta análise. No entanto, a investigação individual simplifica e acelera a análise final, bem como facilita a organização dos experimentos e sua apresentação, especialmente à luz das ferramentas de mensuração disponíveis e suas métricas. A investigação individual permite também que a experimentação seja mais flexível e possa focar as propriedades específicas dos elementos.

[23] *wall-clock time.*

[24] *taskclock* em *process time*, esse relógio tem o tipo `CLOCK_THREAD_CPUTIME` similar ao tipo `CLOCK_PROCESS_CPUTIME`, entretanto considerando apenas um *thread*. Isso permite a mensuração de elementos de processamento (*plugins*) isolados em vez de uma mensuração global do tempo do processo.

[25] Linguagem de programação com apoio à programação distribuída e paralela; orc.csres.utexas.edu

[26] O código-fonte destes elementos encontra-se nos pacotes `gst-plugins-base` ou `gst-plugins-good` dentro da coleção do `Gstreamer`: github.com/GStreamer.

[27] *compressão* refere-se aqui à manipulação da dinâmica do áudio, não tendo relação com compactação.

Desta forma, a análise inicia-se com algoritmos simples e elementares, e prossegue para aplicações compostas por elementos de processamento interligados.

A análise e os experimentos são apresentados para cada elemento de acordo com a estrutura a seguir, que aborda os seguintes aspectos:

- semântica do elemento: o que o elemento faz do ponto de vista musical, ou sua finalidade;
- análise do código-fonte e sua representação através de uma função de custos;
- experimentos: estrutura do pipeline mensurado e resultados numéricos dos experimentos;
- discussão dos resultados considerando a função teórica de custos;
- discussão de possíveis parametrizações (flexibilizações) e o papel dos respectivos parâmetros na função de custos e os efeitos perceptuais decorrentes.

Nos experimentos fica evidente que, em alguns casos excepcionais, os comportamentos de certos elementos não podem ser entendidos ou explicados dentro do arcabouço de análise adotado nesta pesquisa (veja a Seção 3.1.1.2). Nestes casos, sintetiza-se funções de custos limiares a partir dos resultados das mensurações, considerando o contexto da aplicação `flexmix`.

4.4.1 Elemento de amplificação: `GstAudioAmplify`

Utiliza-se o elemento `GstAudioAmplify` para a amplificação de som considerando um certo fator (de ganho); o elemento possui uma entrada e uma saída de som, e recebe como parâmetro o fator de amplificação e o tipo de dados.

```
1  // x[]: cadeia das amostras de entrada
2  // y[]: cadeia das amostras de saída
3  // f: fator de amplificação
4
5  if f <> 1.0:
6    for each sample index i:
7      y[i] = f * x[i]
```

Algoritmo 4.1: *Resumo do algoritmo de amplificação* `GstAudioAmplify`.

O pseudocódigo do elemento `GstAudioAmplify` pode ser visto no Algoritmo 4.1, que representa uma versão simplificada do código-fonte original. Essa simplificação é conveniente para a maioria dos elementos de processamento analisados nesta seção, pois o código-fonte *real* geralmente contém trechos irrelevantes para a análise de desempenho, e pode estar distribuído em mais de um arquivo da biblioteca `gstreamer`.

Vale lembrar que o `Gstreamer` divide os fluxos de amostras (*streams*) em blocos e, em laço, executa os elementos tendo como entradas estes blocos de amostras, que têm tamanho fixo e são indexados a partir do índice 0. Uma vez que esta segmentação não é significativa do ponto de vista da análise de desempenho, os algoritmos apresentados nesta seção omitem o código relativo à segmentação para simplificar a apresentação. Além disso, considera-se, onde for apropriado, que todos os vetores de saída são inicializados com 0.

No pseudocódigo do elemento `GstAudioAmplify` pode-se ver que o elemento, além das operações para a realização do laço, executa uma multiplicação para cada amostra de entrada. No caso em que o fator de amplificação for igual a 1, o elemento não executa o laço e tem custo computacional próximo de zero.

Seguindo a metodologia proposta no Capítulo 3, obtém-se a seguinte função dos custos computacionais do elemento:

$$z_{amp} = c_0 + c_1 n, \tag{4.1}$$

onde n é o número de amostras a processar, e c_0, c_1 são constantes que dependem do software e do hardware concreto (como discutido nas Seções 3.1.1 e 3.5). Observa-se que o código executado não

depende da taxa de amostragem, porém o custo do laço por unidade de tempo depende do número de amostras e da taxa de amostragem. Os custos computacionais do elemento de amplificação podem ser expressos em função da duração em segundos do sinal produzido (t) e da taxa de amostragem em Hz (r) através da equação

$$z_{amp}(t, r) = c_0 + c_1 tr. \tag{4.2}$$

4.4.1.1 Experimento

Nome do *plugin*	Intervalo da taxa de amostragem	Tipos de dados permitidos[28]	Outros parâmetros
`GstAudioAmplify`	[1, 2147483647]	S8, S16LE, S32LE, F32LE, F64LE	fator de amplificação

Tabela 4.2: *Parâmetros do elemento `GstAudioAmplify`.*

Os parâmetros de configuração do elemento `GstAudioAmplify` que influenciam o desempenho podem ser vistos na Tabela 4.2. O elemento aceita taxas de amostragem no intervalo de 1 Hz a 2147483647 Hz e funciona com vários tipos de dados. Para cada tipo de dados, é necessário avaliar as constantes da função de custos (Equação 4.2) separadamente.

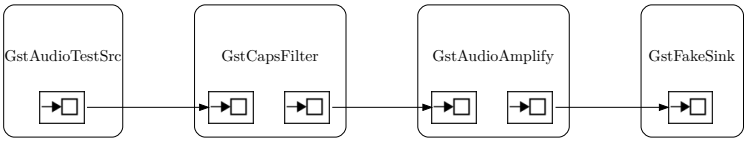

Figura 4.5: *Pipeline para a mensuração do desempenho do elemento `GstAudioAmplify`.*

Para a mensuração do desempenho do elemento de amplificação (`GstAudioAmplify`), constrói-se o pipeline como na Figura 4.5, na qual os elementos de processamento de áudio são representados pelos retângulos, enquanto as setas representam o fluxo de dados de áudio (blocos de amostras). O elemento `GstAudioTestSrc` produz fluxos de amostras de áudio em blocos com uma certa taxa de amostragem. O elemento `GstFakeSink` simula uma saída (*sink*) de áudio que apenas descarta as amostras que chegam na sua entrada, sem realmente produzir qualquer saída. Os elementos `GstAudioTestSrc` e `GstFakeSink` são necessários, pois no arcabouço `Gstreamer` cada pipeline precisa ter sempre elementos iniciais e terminais, e são utilizados nas mensurações por terem propriedades de desempenho simples e custos computacionais previsíveis.

O elemento `GstCapsFilter` não tem custos computacionais próprios e não participa no processamento das amostras. Esse elemento é incluído no pipeline porque permite impôr os parâmetros do experimento, tais como a taxa de amostragem e o tipo de dados das amostras, à cadeia de elementos de processamento. Outros elementos terminais, como aqueles que leem arquivos ou aqueles de saída de som real, são menos previsíveis em seu comportamento, pois esses elementos acoplam o pipeline a mecanismos mais complexos do sistema operacional (por exemplo, ao driver do hardware de áudio) e, com isso, poderiam introduzir distorções nas mensurações.

Considerando que os experimentos desta seção têm o objetivo de avaliar os custos computacionais de pipelines de processamento, deve-se utilizar pipelines que não contenham uma saída de som real, caso contrário, o tempo de execução do pipeline sempre seria igual ou maior que o duração do som produzido.

[28] Usa-se a seguinte forma para denominar o tipo de dados: ponto flutuante (F) ou inteiros com sinal (S), o número de bits por amostra (8-64) e a ordem dos bytes de dados *endianness* (LE=*little-endian* ou BE=*big-endian*)

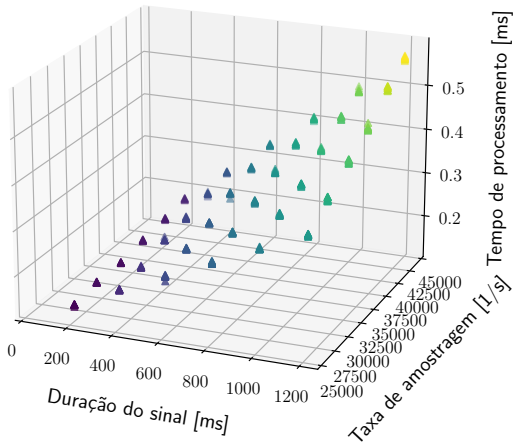

Figura 4.6: *Desempenho do elemento* `GstAudioAmplify`*: Tempo de processamento do elemento em função da taxa de amostragem e da duração do sinal para o tipo de dados ponto flutuante com 32 bits.*

No experimento, o pipeline é executado com diferentes configurações para os parâmetros, sendo gravado a cada vez o tempo de execução do elemento[29]. Os parâmetros são variados para varrer o espaço das configurações; esse espaço é estabelecido pelas faixas úteis dos parâmetros duração de som, taxa de amostragem, e tipo de dados.

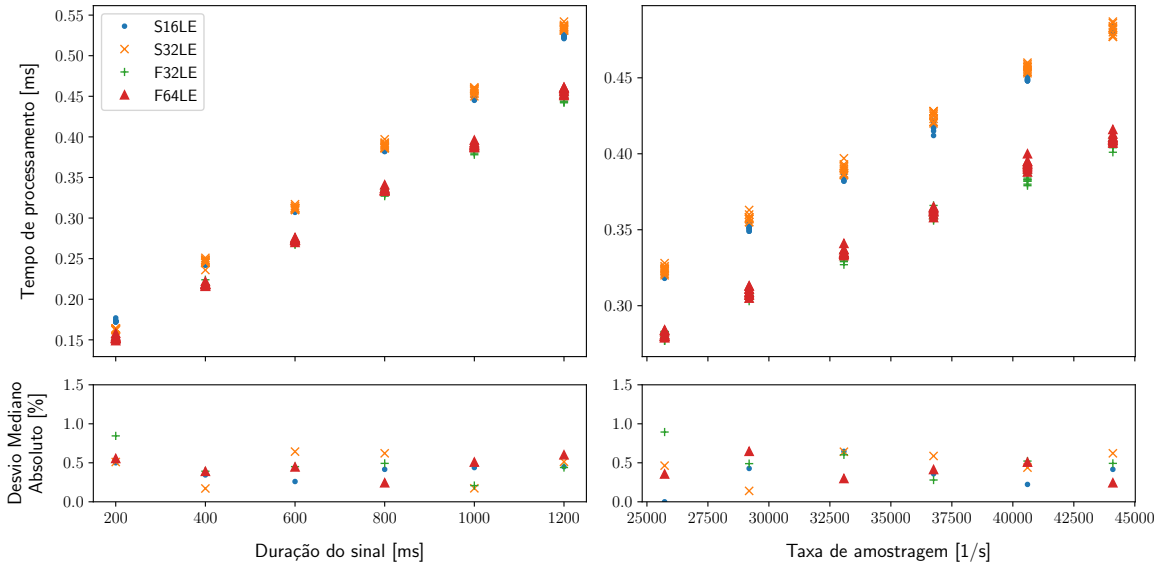

Figura 4.7: *Desempenho do elemento* `GstAudioAmplify` *para tipos de dados diferentes: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do som; b) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem.*

Nas Figuras 4.6 e 4.7 podem ser vistos os resultados do experimento: os tempos de execução

[29]Para todos os experimentos desta seção utiliza-se a ferramenta `gst-launch` que oferece grande flexibilidade na construção de pipelines na linha de comando. Por exemplo, executa-se o pipeline deste experimento com o seguinte comando: `gst-launch audiotestsrc wave=0 volume=1.0 freq=2000 num-buffers=22 samplesperbuffer=2048 ! audio/x-raw, rate=44100, format=F32LE ! audioamplify amplification=0.9 ! fakesink blocksize=2048`.

do elemento em *process time*[30]. Fica evidente que os tempos de processamento mensurados são compatíveis com a Equação 4.2. Os valores dos coeficientes c_0 e c_1 podem ser obtidos por ajuste da função aos valores mensurados.

Vale lembrar que a função de custos tem por objetivo oferecer um limiar sempre maior que os custos efetivamente mensurados, porque o controle dos custos está sujeito à capacidade computacional máxima, um limiar absoluto que não pode ser superado. Por isso, uma simples regressão linear não seria suficiente para o ajuste desta função. Um método de ajuste para o elemento `GstAudioAmplify` pode ser obtido por programação linear. Por exemplo, para uma função de custos com 4 coeficientes e 2 parâmetros

$$z(p, q) = c_0 + c_1 p + c_2 q + c_3 q^2,$$

pode-se construir o seguinte problema de otimização:

$$\min_{\{c_0,c_1,c_2,c_3\}} \sum_{k=1}^{K} \Big(c_0 + c_1 p_k + c_2 q_k + c_3 q_k^2 - \tilde{z}(p_k, q_k) \Big) \left[= c_0 K + c_1 \sum_{k=1}^{K} p_k + c_2 \sum_{k=1}^{K} q_k + c_3 \sum_{k=1}^{K} q_k^2 \right]$$

sujeito a

$$\begin{aligned} & c_0 + c_1 p_k + c_2 q_k + c_3 q_k^2 \geq \tilde{z}(p_k, q_k) \\ & k = 1, \ldots, K \\ & c_0, c_1, c_2, c_3 \in \mathbb{R} \end{aligned} \tag{4.3}$$

onde K é o número de medições e $\tilde{z}_k$ são tempos de execução mensurados com os parâmetros p_k e q_k. Na versão alternativa da função objetivo, as variáveis $c_{0,1,2,3}$ do modelo estão colocadas em evidência, e o termo constante $\sum_{k=1}^{K} \tilde{z}(p_k, q_k)$, desnecessário para o modelo de minimização, foi ignorado. No caso do elemento `GstAudioAmplify`, o modelo acima teria apenas duas variáveis, e a função $\tilde{z}$ teria apenas 1 parâmetro (n).

Para resolver os problemas de otimização usa-se o pacote `scipy.optimize` (linguagem `Python`) que oferece vários métodos de programação linear (*solvers*), como por exemplo, `HiGHS simplex`, `interior-point` ou `revised simplex`. Uma descrição desses métodos encontra-se na documentação do pacote `scipy.optimize`[31].

Os valores calculados para c_0 e c_1 para cada caso pode-se ver na Tabela 4.3. Os erros relativos foram obtidos pela estatística robusta do Desvio Mediano Absoluto (*Median Absolute Deviation* ou *MAD*. Adicionalmente, na última coluna da tabela, são apresentado os custos computacionais (em ms) do algoritmo para um sinal de 1 s sendo a taxa de amostragem 44100 Hz. Esses valores podem ser úteis para comparar, de forma simplificada) os custos computacionais dos diversos elementos investigados nesta seção.

[30] A mensuração do tempo de execução em *wall-clock time*, que é aplicável apenas para pipelines como um todo, não é útil nesse caso porque inclui os custos computacionais dos outros elementos.

[31] docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html

Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
S16LE	0.1119	1.056e-05	2.3	0.58
S32LE	0.1010	1.106e-05	1.3	0.59
F32LE	0.1011	8.890e-06	1.4	0.49
F64LE	0.0998	9.100e-06	1.7	0.50

Tabela 4.3: *Coeficientes da função de custos do elemento* `GstAudioAmplify`.

4.4.1.2 Flexibilização

O experimento mostra que o tipo de dados tem influência nos custos computacionais. O pipeline tem um desempenho melhor quando se usa tipos de ponto flutuante (F32LE, F64LE) em comparação com os tipos inteiros (S16LE, S32LE), mas a diferença de desempenho entre precisões diferentes (16, 32 ou 64 bits) do mesmo tipo de dados é pequena. O desempenho é melhor quando se processa dados do tipo ponto flutuante tem sua explicação na arquitetura da CPU[32]. O fato de não haver diferença de desempenho quando ao número de bits indica que as operações não são vetorizados na CPU (execução paralela, SIMD).

Considerando o elemento de forma isolada, não parece útil usar o tipo de dados como parâmetro de flexibilização, principalmente porque os formatos com números inteiros não apresentam uma melhor precisão e levam a custos computacionais maiores. Em conjunto com outros elementos, que tenham diferenças entre os custos de tipos distintos, a utilização do tipo de dados como parâmetro de flexibilização pode ser sensata (veja as Seções 3.1.2 e 4.4.7 sobre cadeias de elementos).

Como alternativa ao modelo de flexibilização proposto, seria possível utilizar o parâmetro de amplificação para manipular os custos computacionais, por exemplo, arredondando valores de amplificação próximos de 1, o que equivale a desligar o processamento do elemento. Essa estratégia não é seguida nesta pesquisa, porque o impacto perceptual dessa intervenção no processo musical não se justifica pelo pequeno ganho computacional da flexibilização no contexto do `flexmix`, considerando que os custos computacionais do elemento de amplificação são desprezíveis se comparados com os outros elementos contidos no `flexmix`.

Em uma outra abordagem alternativa, poder-se-ia implementar algoritmos simplificados de multiplicação como os de Bianchi (Bianchi, 2013), introduzidos na Seção 2.5; aqueles algoritmos possuem custos computacionais bem menores que a multiplicação comum, mas admitem apenas certos fatores de amplificação[33]. Em uma estratégia de flexibilização poder-se-ia aproveitar desta vantagem, considerando-se como alternativa o uso de valores de amplificação arrendondados para diminuir os custos computacionais, sem prejuízo da opção de usar a configuração original, que possui custos relativos maiores porém maior precisão do resultado. Esta abordagem à flexibilização pode ser considerada para todos os elementos de processamento que envolvem multiplicações ou divisões com tipos inteiros, inclusive para os elementos discutidos nesta seção.

4.4.2 Elementos de equalização: `GstIirEqualizer3Bands`, `GstIirEqualizer10Bands`

Os elementos `GstIirEqualizer3Bands` e `GstIirEqualizer10Bands` são utilizados para controlar certas faixas do espectro de um sinal de áudio, aplicando no sinal de entrada ganhos em 3 ou 10 faixas de frequências.

Os filtros são obtidos pela combinação de 3 ou 10 filtros de resposta ao impulso infinita (*Infinite*

[32] As operações aritméticas com dados do tipo ponto flutuante são executadas usando unidades aritméticas da CPU diferentes daquelas usadas nas operações com dados do tipo inteiro.

[33] tipicamente fatores da forma 2^n, com n positivo ou negativo, para os quais a multiplicação não produza *underflow* ou *overflow* das representações nos sistemas S16LE ou S32LE.

impulse response (Filtro de resposta ao impulso infinita) (IIR)[34]) do tipo passa-faixa, que podem ser caracterizados pela seguinte equação (Moore, 1990):

$$y_n = \sum_{i=0}^{M_a} a_i x_{n-i} - \sum_{i=1}^{M_b} b_i y_{n-i}, \tag{4.4}$$

onde y e x são os sinais de entrada e saída, e a e b são os vetores (não-nulos) dos coeficientes do filtro com tamanhos M_a M_b respectivamente. Esta equação define um filtro com dois polos e dois zeros, que são configurados para localizar a faixa de passagem do filtro em função da frequência central.

```
1  // x[]: bloco das amostras de entrada
2  // y[]: bloco das amostras de saída
3  // para cada faixa de frequências:
4  // a0[], a1[], a2[], b1[], b2[]: coeficientes dos filtros
5  // hx1[], hx2[]: valores das amostras de entrada passadas
6  // hy1[], hy2[]: valores das amostras de saídas passadas
7
8  for each sample index i:
9    for each frequency band index f:
10     y[i] =  a0[f]*x[i] + a1[f]*hx1[f] + a2[f]*hx2[f] + b1[f]*hy1[f] + b2[f]*hy2
           [f]
11
12     // update history
13     hx2[f] = hx1[f]
14     hx1[f] = x[i]
15     hy2[f] = hy1[f]
16     hy1[f] = y
17     x[i] = y[i]
```

Algoritmo 4.2: *Resumo do algoritmo do elemento* `GstIirEqualizer3Bands`

O Algoritmo 4.2 representa um resumo do algoritmo de processamento implementado nos elementos `GstIirEqualizer3Bands` e `GstIirEqualizer10Bands`. Esta implementação baseia-se na aplicação de um filtro do tipo IIR em cada faixa de frequências; cada filtro possui coeficientes diferentes, calculados na configuração do elemento[35].

A função de custos deriva diretamente do Algoritmo 4.2, com o qual se obtém a seguinte equação:

$$z_{eq} = c_0 + c_1' mn \tag{4.5}$$

onde n é o número de amostras, m o número de faixas de frequências do filtro, e c_0 e c_1' são as constantes de custos. Neste exemplo, fica evidente que a constante c_1', que representa os custos computacionais das adições e multiplicações na linha 7 do Algoritmo 4.2, pode ter um valor diferente dependendo se tais operações são executadas paralelamente ou sequencialmente. Ainda assim, a função de custos computacionais expressa custos lineares no número de amostras n. Sendo m uma constante ($m = 3$ no caso do elemento `GstIirEqualizer3Bands` e $m = 10$ no caso do elemento `GstIirEqualizer10Bands`), pode-se combinar m e c_1' para formar uma nova constante $c_1 = mc_1'$. Expressando os custos computacionais em função da duração do sinal produzido (t) e da taxa de amostragem em Hz (r) obtém-se:

$$z_{eq}(t, r) = c_0 + c_1 tr. \tag{4.6}$$

[34] A resposta do filtro ao impulso é geralmente infinita em função da retroalimentação com as saídas anteriores (termos $b_i y_{n-i}$ na Equação 4.4).

[35] Interessantemente, no código-fonte, é usada a adição com os coeficientes (b1, b2) mesmo que correspondam à subtração do segundo termo da Equação 4.4. Isso é compensado no cálculo dos coeficientes (na configuração do elemento) por multiplicação dos coeficientes por -1. Provavelmente, espera-se um melhor desempenho (do algoritmo) à luz das operações SIMD disponíveis, visto que muitas arquiteturas de CPU possuem uma operação de multiplicação seguida de adição (*multiply-add*) e, apenas arquiteturas mais novas oferecem uma operação de multiplicação seguida de subtração. Normalmente, compiladores não são capazes de reconhecer a opção de usar a operação *multiply-add* com o sinal dos coeficientes trocados, pois isso requer uma reescrita do código-fonte.

A função de custos do elemento `GstIirEqualizer10Bands` é a mesma do elemento `GstIirEqualizer3Bands`, porém com outra constante c_1.

4.4.2.1 Experimento

Nome do *plugin*	Intervalo da taxa de amostragem	Tipos de dados permitidos	Outros parâmetros
`GstIirEqualizer 3Bands`	[1000, 2147483647]	S16LE, F32LE, F64LE	ganhos (3) nas faixas de frequência
`GstIirEqualizer 10Bands`	[1000, 2147483647]	S16LE, F32LE, F64LE	ganhos (10) nas faixas de frequência

Tabela 4.4: *Parâmetros dos elementos `GstIirEqualizer3Bands` e `GstIirEqualizer10Bands`.*

Na Tabela 4.4 podem ser vistos os parâmetros de configuração dos elementos de equalização que influenciam o desempenho. Os elementos aceitam taxas de amostragem na faixa de 1000 Hz a 2147483647 Hz e funcionam com vários tipos de dados. Para cada tipo de dados, é necessário avaliar as constantes da função de custos (Equação 4.2) separadamente.

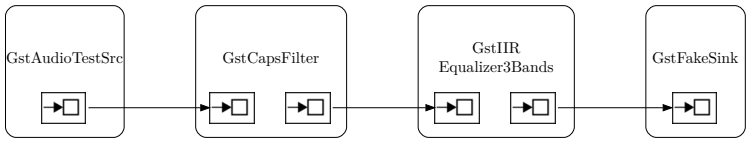

Figura 4.8: *Pipeline para a mensuração do desempenho do elemento `GstIirEqualizer3Bands`.*

Para a mensuração do desempenho dos elementos de equalização (`GstIirEqualizer3Bands`, `GstIirEqualizer10Bands`), constrói-se o pipeline como visto na Figura 4.8 usando o mesmo padrão que no caso do elemento de amplificação (`GstAudioAmplify`). Tal como o pipeline, a execução do experimento também segue esse padrão (veja a Seção 4.4.1.1).

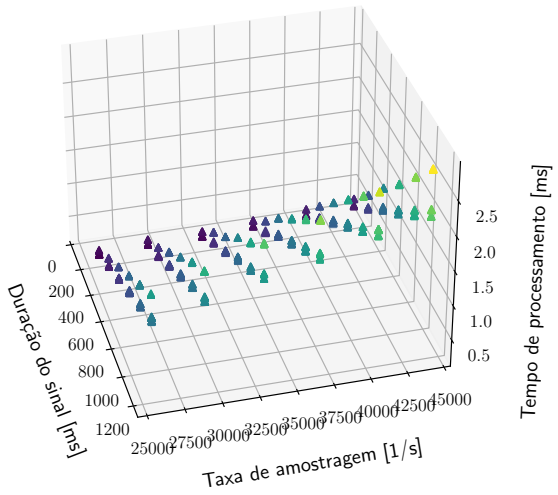

Figura 4.9: *Desempenho do elemento* `GstIirEqualizer3Bands`*: Tempo de processamento do elemento em função da taxa de amostragem e da duração do sinal para o tipo de dados ponto flutuante com 32 bit.*

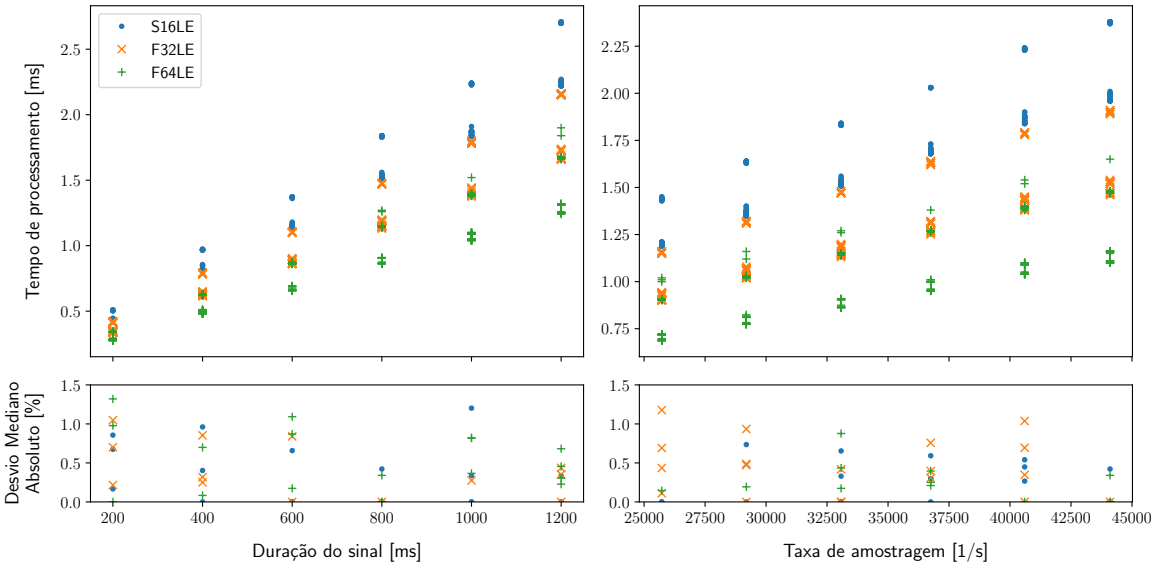

Figura 4.10: *Desempenho do elemento* `GstIirEqualizer3Bands` *para tipos de dados diferentes: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do som; b) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem.*

Os resultados do experimento, apresentados nas Figuras 4.9 e 4.10, confirmam a função de custos da Equação 4.5. No entanto, observa-se que (a) o desempenho é pior quando se usa números inteiros como tipo de dados do que quando se usa números de ponto flutuante e (b) o desempenho de números de 32 bits é pior do que o desempenho de números de 64 bits. O primeiro fato pode ser explicado, como no caso do elemento `GstAudioAmplify` (veja a Seção 4.4.1), pelo fato de que as operações com tipos de dados distintos são executadas em distintas unidades aritméticas da CPU. A diferença de desempenho entre os tipos ponto flutuante de 32 bits e de 64 bits pode ser explicada pela arquitetura da CPU (`x86_64`) que é otimizada para tipos de 64 bits.

Uma outra observação é que os tempos de execução do mesmo experimento (mesmos parâmetros) têm uma distribuição bimodal (dois picos). Para investigar este fato, que não pode ser explicado

com o algoritmo, foram executadas mensurações do desempenho do pipeline com números distintos de núcleos da CPU; além do tempo de execução, foram registrados o número de falhas de página[36], o número de falhas do *cache*[37] e o número de falhas do *cache L1*[38] usando a ferramenta `perf`. Vale lembrar que esta ferramenta com seus contadores engloba todo o tempo de execução de uma aplicação, mas não os tempos de execução de elementos específicos (contidos em uma aplicação `Gstreamer`).

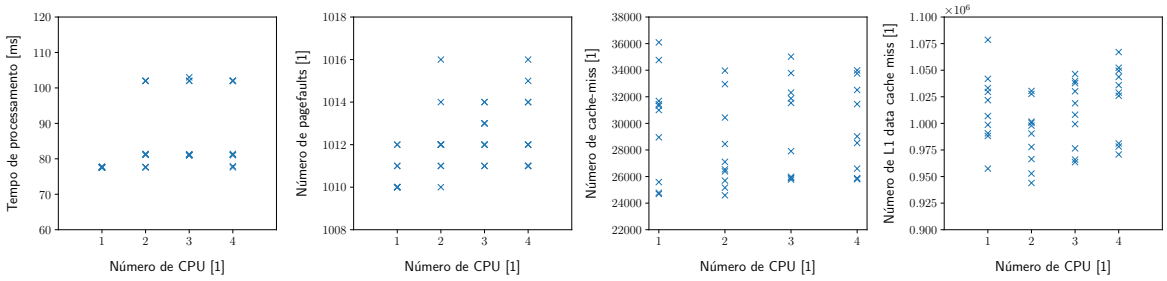

Figura 4.11: *Desempenho do elemento* `GstIirEqualizer3Bands` *em função do número de núcleos da CPU (da esquerda para a direita, respectivamente): a) tempo de processamento para um bloco de amostras; b) número de* pagefaults*; c) número de* cache-misses*; d) número de L1* cache-misses*.*

Os resultados deste experimento, que podem ser vistos na Figura 4.11, mostram que o efeito de dois tempos de execução distintos (distribuição bimodal) desaparece quando se utiliza um único núcleo e que o número de falhas de página é menor do que com mais núcleos. Observa-se que o tempo de execução com somente um núcleo é menor ou igual ao tempo de execução com mais núcleos, o que significa que o algoritmo não é executado de forma paralela quando se utiliza mais de um núcleo. Quanto às falhas de *cache*, observa-se diferentes distribuições de valores mensurados em função do número de núcleos; com mais que um núcleo, há distribuições bi- ou multi-modais.

Estes resultados experimentais mostram que a distribuição bimodal dos tempos de processamento do elemento aparece quando se executa o pipeline com mais que um núcleo da CPU. Os resultados quanto às falhas de página e de *cache* não sugerem uma explicação conclusiva.

Vale lembrar que o arcabouço de análise do desempenho de algoritmos adotado nesta pesquisa (veja a Seção 3.1.1.2) não requer o entendimento de todos os efeitos quanto ao desempenho de um elemento; é suficiente que (a) a função dos custos computacionais tenha uma correlação boa com os custos médios mensurados, e (b) a função ajustada dos custos seja capaz de reproduzir o desempenho mensurado em seu limite superior. No contexto de uma aplicação flexibilizada, no entanto, é necessário considerar que a imprecisão da função dos custos de um elemento tem um impacto na imprecisão da aplicação (flexibilizada) proporcional aos seus custos computacionais em relação aos custos totais da aplicação.

Conclui-se também que o resultado destes experimentos amplia a noção de que os custos computacionais mensurados nesta pesquisa dependem fortemente da configuração do hardware (neste caso da CPU). Observa-se também que diferentes configurações não mudam a forma linear da função de custos, porém levam a constantes c_0 e c_1 distintas para cada configuração de hardware.

[36] Dados requisitados pela CPU que não foram encontrados na memória virtual do processo; o respectivo bloco de memória tem que ser tornado acessível pela CPU (mapeamento). Este mecanismo pode levar a atrasos quando se acessa os dados pela primeira vez.

[37] Dados requisitados pela CPU que não foram encontrados na memória de acesso rápido (*cache*) e são lidos da memória principal, entretanto, com maior tempo de leitura.

[38] Dados requisitados pela CPU que não foram encontrados no *cache L1*, memória de acesso mais rápido que é associada diretamente a um núcleo da CPU. Os dados têm que ser lidos do *cache* (geral) ou da memória principal.

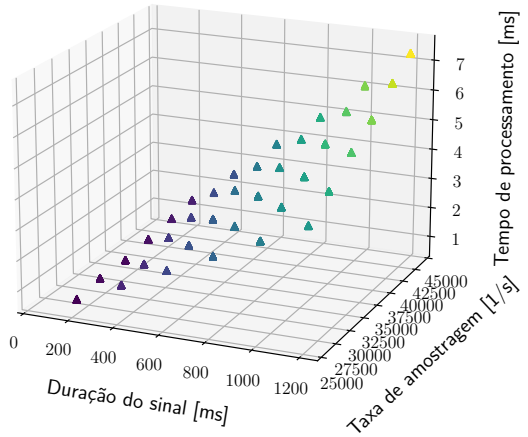

Figura 4.12: *Desempenho do elemento* `GstIirEqualizer10Bands`*: Tempo de processamento do elemento em função da taxa de amostragem e da duração do sinal para o tipo de dados ponto flutuante com 32 bit).*

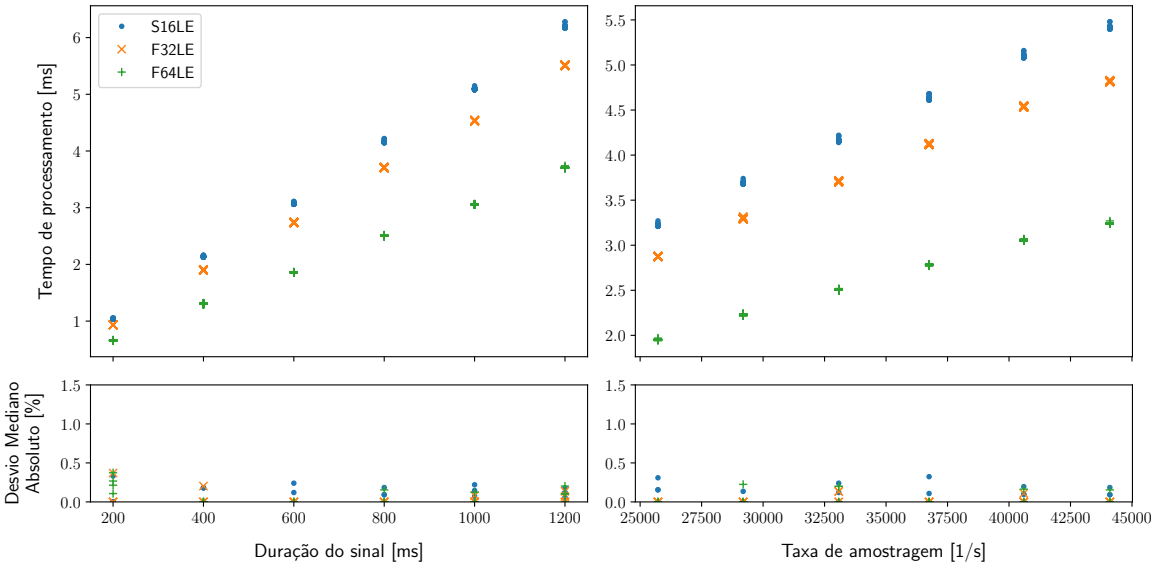

Figura 4.13: *Desempenho do elemento* `GstIirEqualizer10Bands` *para tipos de dados diferentes: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do sinal; b) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem.*

As Figuras 4.12 e 4.13 mostram os resultados do experimento com o elemento `GstIirEqualizer10Bands` que confirmam a função de custos 4.6. Em comparação com os resultados do elemento `GstIirEqualizer3Bands` (veja a Figura 4.10) observa-se que não há distribuições bimodais dos tempos de execução. Os tempos de execução do elemento `GstIirEqualizer10Bands` são maiores que os do elemento `GstIirEqualizer3Bands` quando executados com os mesmos parâmetros, conforme a Função 4.5.

Os valores calculados para c_0 e c_1, para cada caso, podem ser vistos nas Tabelas 4.5 e 4.6 desta Seção.

Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
S16LE	0.1317	6.523e-05	20.2	3.01
F32LE	0.1117	5.201e-05	24.3	2.41
F64LE	0.1146	3.952e-05	18.9	1.86

Tabela 4.5: *Coeficientes da função de custos do elemento* `GstIirEqualizer3Bands`.

Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
S16LE	0.1547	15.37e-5	2.0	6.93
F32LE	0.1594	13.59e-5	2.1	6.15
F64LE	0.1365	9.059e-05	1.8	4.13

Tabela 4.6: *Coeficientes da função de custos do elemento* `GstIirEqualizer10Bands`.

4.4.2.2 Flexibilização

O experimento mostra a influência dos parâmetros taxa de amostragem (r), duração de sinal produzido (t) e tipo de dados nos custos computacionais dos elementos `GstIirEqualizer3Bands` e `GstIirEqualizer10Bands`. No entanto, como no caso do elemento `GstAudioAmplify`, o desempenho relativo entre os tipos de dados indica que o tipo de dados das amostras não parece ser um parâmetro de flexibilização útil.

Analogamente ao elemento `GstAudioAmplify`, os elementos de equalização também poderiam ser desligados usando-se ganho zero para cada faixa[39]. A possibilidade de desligar o processamento destes elementos poderia ser usada em uma estratégia de flexibilização, por exemplo, para aplicações com muitas faixas de som ou para faixas de som com pouca nitidez. Porém, uma estratégia neste sentido não é desenvolvida nesta pesquisa por corresponder a uma perturbação significativa da especificação musical da cadeia de elementos DSP.

Considerando que a coleção do `Gstreamer` possui um elemento de equalização com o número de faixas de frequências como parâmetro de configuração (`GstIirEqualizerNBands`), poder-se-ia desenvolver uma estratégia de flexibilização usando este parâmetro considerando a diferença de desempenho indicado pelo parâmetro m na função de custos (veja a Equação 4.5), bem com a diferença entre os desempenhos mensurados dos elementos `GstIirEqualizer3Bands` e `GstIirEqualizer10Bands` (veja as Figuras 4.10 e 4.13). Entretanto, o uso do número de faixas de frequências como parâmetro de flexibilização requereria a implementação de funções de transformação entre as versões dos filtros com números de faixas diferentes que respeitasse a especificação musical do usuário. Vale lembrar que a utilidade da flexibilização de um elemento depende da aplicação musical na qual o elemento é empregado, de sua função dentro da aplicação e sua relação com outros elementos, e também se a variação de desempenho daquele elemento tem impacto relevante nos custos computacionais da aplicação como um todo.

4.4.3 Elemento de controle da dinâmica: `GstAudioDynamic`

O elemento `GstAudioDynamic` pode exercer duas funções distintas: a de compressão dinâmica (diminuindo amplitudes altas) ou a de expansão dinâmica (aumentando amplitudes baixas); nesta pesquisa considera-se apenas a primeira função.

[39] Utiliza-se a escala dB para os ganhos na configuração dos elementos de equalização.

O elemento `GstAudioDynamic` aplica um remapeamento da amplitude de forma não-linear usando funções de transferência de amplitude, por exemplo,

$$y = \begin{cases} \text{sgn}(x) * (t + r(|x| - t)), & \text{se } |x| > t \\ x, & \text{c.c.,} \end{cases} \tag{4.7}$$

onde y representa a amostra atual de saída, x representa a amostra atual de entrada, e r (ratio) e t (*threshold*) são coeficientes fixos escolhidos em função das predileções musicais. Esta função de transferência é utilizada no elemento `GstAudioDynamic` quando se usa a configuração *hard-knee*. Além disso, o elemento possui uma outra configuração, *soft-knee*, que usa a seguinte função de transferência:

$$y = \begin{cases} \text{sgn}(x) * (ax^2 + b|x| + c), \text{se } |x| < 1 \\ \text{“hard-knee”} \qquad \text{c.c.,} \end{cases} \tag{4.8}$$

onde y representa a amostra atual de saída, x representa a amostra atual de entrada, e a, b e c são coeficientes fixos.

```
1  // x[]: cadeia das amostras de entrada
2  // y[]: cadeia das amostras de saída
3  // t: limiar (threshold)
4  // r: ratio
5
6  for each sample index i:
7    if x[i] > t:
8      y[i] = t + r * (x[i] - t)
9    else if x[i] < -t:
10     y[i] = -t + r * (x[i] + t)
11   else:
12       y[i] = x[i]
```

Algoritmo 4.3: *Resumo do algoritmo do elemento* `GstAudioDynamic` *no modo* hard-knee.

O pseudocódigo do elemento `GstAudioDynamic`, no modo *hard-knee*, pode ser visto no Algoritmo 4.3; as constantes `t` (*threshold*) e `r` (ratio) são parâmetros de configuração do elemento.

```
1  // x[]: cadeia das amostras de entrada
2  // y[]: cadeia das amostras de saída
3  // t: limiar (threshold)
4  // r: ratio
5  // ap, bp, cp, an, bn, cn: constantes
6
7  for each sample index i:
8    if (x[i] > 1.0)
9      y[i] = 1.0 + (x[i] - 1.0) * r
10   else if (x[i] > t)
11     y[i] = ap * x[i] * x[i] + bp * x[i] + cp
12   else if (x[i] < -1.0)
13       y[i] = -1.0 + (x[i] + 1.0) * r;
14   else if  (x[i] < -t)
15     y[i] = an * x[i] * x[i] + bn * x[i] + cn;
16   else:
17       y[i] = x[i]
```

Algoritmo 4.4: *Resumo do algoritmo do elemento* `GstAudioDynamic` *no modo* soft-knee.

O pseudocódigo do elemento `GstAudioDynamic` pode ser visto no Algoritmo 4.4; os coeficientes `ap`, `bp`, `cp`, `an`, `bn` e `cn` são computados a partir dos parâmetros de configuração `threshold` e `ratio`. Em comparação com o algoritmo de *hard-knee*, nota-se que se utiliza dois limiares (1.0 et) e que o número de multiplicações nos casos $1.0 > x > t$ e $-1.0 < x < -t$ é maior que no caso *hard-knee*. Além disso, pode ser visto no código-fonte do elemento, no modo *soft-knee*, que as constantes (`ap`,

`bp`, `cp`, `an`, `bn` e `cn`) são computadas para cada bloco de amostras[40]. Decorrentemente espera-se um desempenho pior quando se utiliza o modo *soft-knee*.

Para a função de custos do Algoritmo 4.3 (*hard-knee*), obtém-se a seguinte equação :

$$z_{dynhard}(n_t) = c_0 + c_t n_t, \tag{4.9}$$

onde n_t é o número de amostras que têm valores fora do intervalo $(-t, t)$ e c_0, c_t são as constantes a se estimar experimentalmente. Analogamente, constrói-se a função de custos do Algoritmo 4.4 (*soft-knee*),

$$z_{dynsoft}(n_m, n_t) = c_0 + c_m n_m + c_t n_t, \tag{4.10}$$

onde n_m é o número de amostras com valores fora do intervalo $(-1.0, 1.0)$, n_t é o número de amostras com valores dentro do intervalo $(-1.0, 1.0)$ mas fora do intervalo $(-t, t)$ e , c_0, c_m e c_t são constantes. Espera-se que o valor de c_t seja um pouco maior que c_m pois o cálculo nas linhas 11 e 15 envolve mais multiplicações. Espera-se também que c_0 seja maior que o c_0 no modo *hard-knee* (Equação 4.9), decorrente do cálculo dos coeficientes (veja a discussão após o Algoritmo 4.4).

Observa-se que os custos computacionais dos Algoritmos 4.4 e 4.3 dependem dos valores reais de entrada, uma questão que deve ser considerada nos experimentos.

4.4.3.1 Experimento

Nome do *plugin*	Intervalo da taxa de amostragem	Tipos dos dados permitidos	Outros parâmetros
`GstAudioDynamic`	[1, 2147483647]	S16LE, F32LE	*hard-knee*, *soft-knee*, *threshold*

Tabela 4.7: *Parâmetros do elemento `GstAudioDynamic`.*

Os parâmetros de configuração do elemento `GstAudioDynamic` que influenciam o desempenho podem ser vistos na Tabela 4.7. Destaca-se que o elemento aceita apenas dois tipos de dados com faixas de valores representáveis distintas. Vale lembrar que a cardinalidade dos conjuntos dos valores permitidos (S16LE: 2^{16}, F32LE: $\approx 2^{32}$) tem grande influência na quantização do sinal e, decorrentemente, na qualidade percebida (veja a Seção 2.1).

Como discutido na análise teórica do elemento, espera-se que o elemento tenha um desempenho melhor no modo *hard-knee* que no modo *soft-knee*, e que o parâmetro *threshold* tenha influência no desempenho do elemento também.

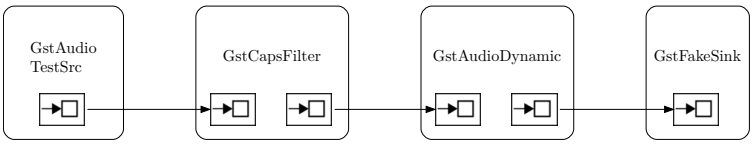

Figura 4.14: *Pipeline para a mensuração do desempenho do elemento `GstAudioDynamic`.*

Para a mensuração do desempenho do elemento `GstAudioDynamic`, constrói-se o pipeline como vista na Figura 4.14 usando o mesmo padrão dos elementos anteriores.

A execução do experimento deve seguir o mesmo padrão dos experimentos com o elemento de amplificação (veja a Seção 4.4.1.1), todavia é importante considerar-se a relação entre o limiar do compressor e os custos computacionais. Em função disso, constrói-se um experimento preliminar que utiliza sinais específicos (ruído, dente-de-serra, triangular) para estimar os custos computacionais em função do limiar.

[40](no arcabouço `Gstreamer` blocos normalmente têm tamanhos de 1024 até 8192 amostras)

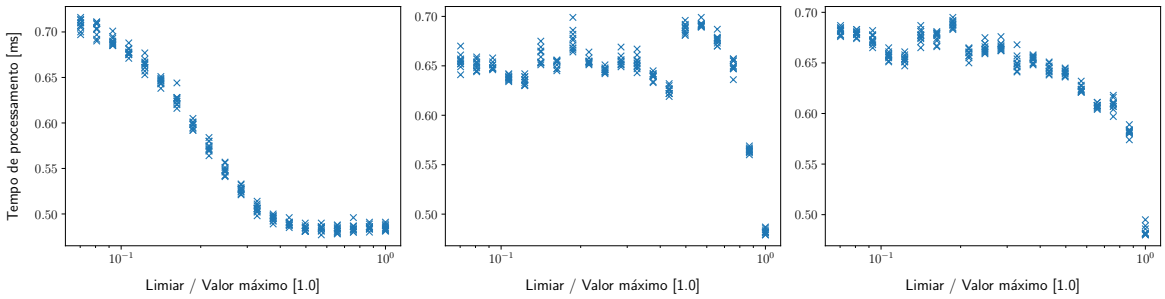

Figura 4.15: *Desempenho do elemento* `GstAudioDynamic` *tempo de processamento em função de limiares de compressão diferentes e tipos de sinal: a) (esquerda) tempo de processamento para o ruído rosa; b) (centro) tempo de processamento para o sinal dente-de-serra; c) (direita) (centro) tempo de processamento para o sinal triangular.*

Na Figura 4.15 podem ser vistos os resultados do experimento, especificamente os tempos de processamento do elemento `GstAudioDynamic` em função do quociente do limiar e do valor máximo do sinal ($q = t/x_{max}$). Nesta figura podem ser observados claramente os máximos e mínimos do tempo de processamento. Como se espera, os máximos ocorrem com valores pequenos do quociente ($q < 0.1$), ao passo que os mínimos ocorrem com valores maiores deste quociente ($q \approx 1.0$). As diferenças entre os tipos de sinais podem ser explicadas pela distribuição dos valores das amostras, que são distintos para cada um dos tipos de sinal testados. Visto que o sinal de entrada é imprevisível no caso geral, deve-se utilizar limiares pequenos em comparação com os valores absolutos das amostras (< 0.1) para avaliar o pior caso do desempenho. Os resultados mostram também que os custos computacionais no melhor caso podem ser 30% menores que no pior caso.

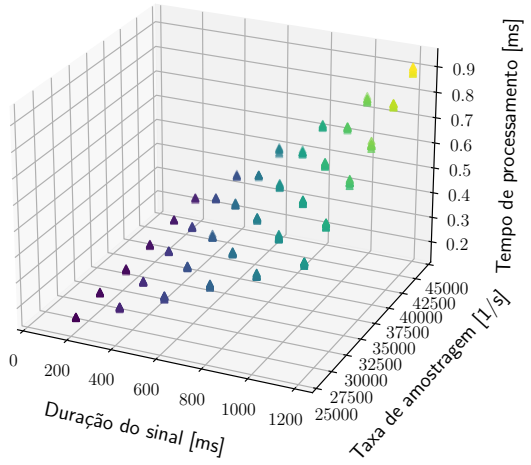

Figura 4.16: *Desempenho do elemento* `GstAudioDynamic`*: Tempo de processamento do elemento em função da taxa de amostragem e da duração do sinal para o tipo de dados ponto flutuante com 32 bits.*

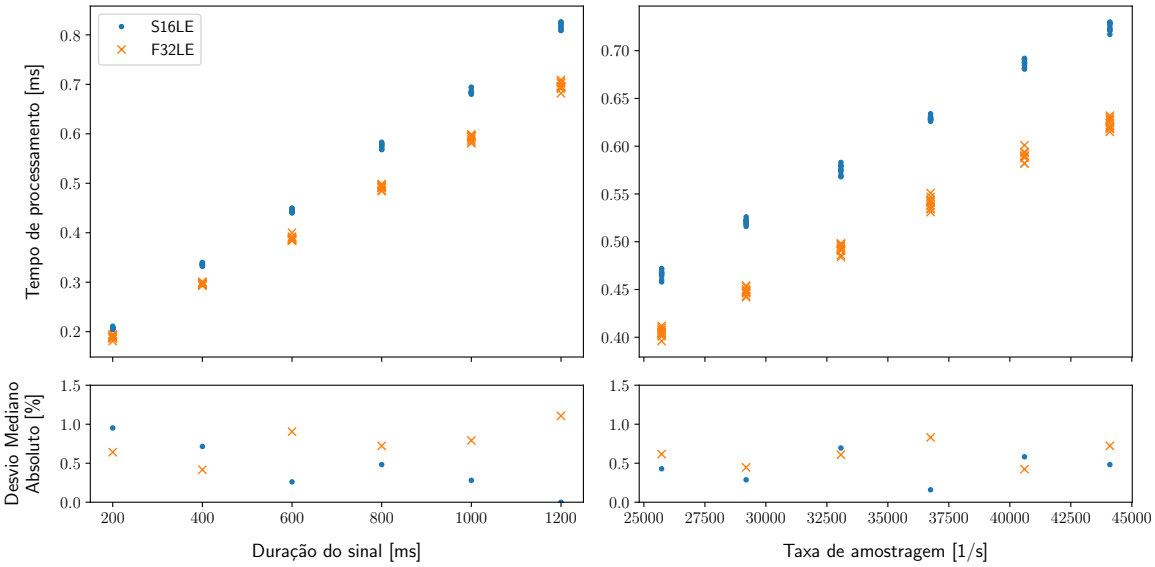

Figura 4.17: *Desempenho do elemento* `GstAudioDynamic` *para tipos de dados diferentes: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do sinal; b) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem.*

Nas Figuras 4.16 e 4.17 podem ser vistos os resultados do experimento com o elemento `GstAudioDynamic` no modo *soft-knee*. Como com os outros elementos, fica evidente o pior desempenho quando se utiliza números inteiros.

Visto que o sinal de teste (ruído rosa) apresenta uma distribuição uniforme em relação às faixas de amplitude consideradas na Equação 4.10), pode-se substituir n_m e n_t por $f_m n$ e $f_t n$ respectivamente, onde f_m e f_t representam a proporção de valores em tais faixas em função do número de amostras total. Consequentemente, pode-se atualizar a Equação 4.10 como segue:

$$\begin{aligned} z_{dynsoft}(n) &= c_0 + c_m f_m n + c_t f_t n \\ &= c_0 + (c_m f_m + c_t f_t)\, n \\ &= c_0 + c_1 n \qquad \text{sendo} \quad c_1 = c_m f_m + c_t f_t \\ z_{dynsoft}(t, r) &= c_0 + c_1 tr. \end{aligned} \tag{4.11}$$

Desta forma, a Equação 4.11 descreve mais adequadamente os custos computacionais mensurados. Aplicando o método de ajuste de função (veja o problema de otimização linear 4.3) podem ser calculados os valores de c_0 e c_1, lembrando que a função de custos representa o pior caso dos custos computacionais.

Além disso, pode-se atualizar a Equação 4.9 (*hard-knee*) quanto ao valor de n_t com o mesmo argumento usado no caso do modo *soft-knee*; desta forma pode-se substituir n_t por $f_t n$, onde f_t é a fração fixa dos valores de amostras fora do intervalo $(-t, t)$ em relação ao número total de amostras (n):

$$\begin{aligned} z_{dynhard}(n) &= c_0 + c_t f_t n \\ &= c_0 + c_1 n \qquad \text{sendo} \quad c_1 = c_t f_t \\ z_{dynhard}(t, r) &= c_0 + c_1 tr. \end{aligned} \tag{4.12}$$

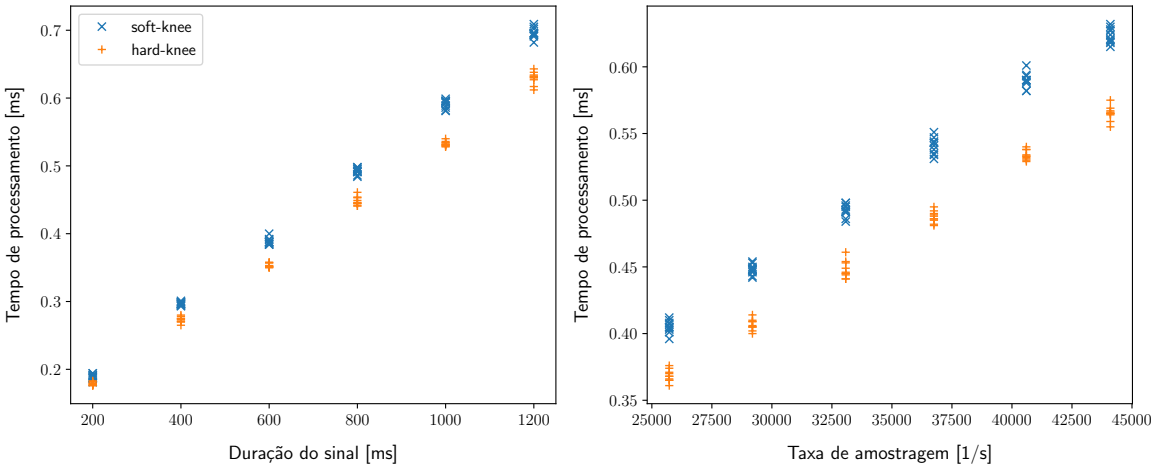

Figura 4.18: *Comparação do desempenho dos modos soft-knee e hard-knee do elemento* `GstAudioDynamic`*: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do sinal; b) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem.*

Na Figura 4.18 pode ser visto o resultado do experimento com o elemento `GstAudioDynamic` comparando os modos *soft-knee* e *hard-knee*. Esta figura comprova a conjectura acerca dos valores de c_0 e c_1 no caso do modo *soft-knee* serem maiores que os respectivos valores no caso do modo *hard-knee*.

Os valores de c_0 e c_1 obtidos pelo ajuste das funções de custos podem ser vistos na Tabela 4.8.

Opção	Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
hard-knee	S16LE	0.1028	1.685e-05	1.4	0.85
	F32LE	0.1004	1.338e-05	1.6	0.69
soft-knee	S16LE	0.1030	1.808e-05	1.5	0.90
	F32LE	0.0998	1.521e-05	1.6	0.77

Tabela 4.8: *Coeficientes da função de custos do elemento* `GstAudioDynamic`*.*

4.4.3.2 Flexibilização

Assim como os demais elementos discutidos neste capítulo, mostra-se que o desempenho depende da duração do sinal e da taxa de amostragem. Observa-se que o parâmetro *tipo de dados* não é útil para a flexibilização porque o tipo de dados `S16LE` tem um desempenho pior nos experimentos do que o desempenho do tipo `F32LE` e, além disso, espera-se que o uso do tipo `S16LE` leve a uma pior qualidade percebida do que o uso do tipo `F32LE`.

Considerando a pequena diferença de desempenho quando se compara os modos *hard-knee* e *soft-knee*, espera-se que uma flexibilização deste parâmetro não seja muito eficiente. No entanto, isso depende da aplicação na qual o elemento é utilizado e, mais especificamente, da proporção dos custos computacionais do elemento em relação aos custos computacionais totais da aplicação.

4.4.4 Elemento de mixagem: `GstAudioMixer`

O elemento `GstAudioMixer` tem como função musical mixar várias faixas de som em uma faixa única, correspondendo à função de uma mesa de mixagem.

```
1  // x[][]: bloco de amostras de entrada (por faixa)
2  // y[]: bloco de amostras de saída
```

```
3  // vol[]: fator de amplificação (por faixa)
4
5  for each channel index c:
6    for each sample index i:
7      y[i] += vol[c] * x[c][i]
```

Algoritmo 4.5: *Resumo do algoritmo do elemento* `GstAudioMixer`.

O Algoritmo 4.5 apresenta um resumo do código-fonte da função de processamento do elemento `GstAudioMixer`[41]. No código-fonte faz-se uso da linguagem de programação `Orc` (Kitchin *et al.*, 2009) que permite paralelizar a execução da adição e multiplicação de blocos de amostras por meios de vetorização (via SIMD).

Para a função dos custos computacionais do Algoritmo 4.5, obtém-se a seguinte equação:

$$z_{mix}(n) = c_0 + c_1 n_c + c_2 n_c n \tag{4.13}$$

ou, equivalentemente,

$$z_{mix}(t, r) = c_0 + c_1 n_c + c_2 n_c t r, \tag{4.14}$$

onde n é o número de amostras da entrada, n_c é o número de canais, t é a duração do sinal, r é a taxa de amostragem (de onde $n = tr$), e c_0, c_1 e $c_2$são parâmetros a serem estimados experimentalmente. Destaca-se que uma paralelização das multiplicações e adições (linha 7 do Algoritmo 4.5) reduz o número de laços (linha 6) por um fator *constante* e, por isso, não muda a forma da equação (função linear); a redução dos custos computacionais expressa-se em um valor c_1 menor do que no caso sem paralelização.

Curiosamente, o elemento não utiliza outros esquemas de paralelização possíveis. Por exemplo, para aplicações com mais de 4 faixas de entrada, poder-se-ia numa primeira etapa mixar pares de faixas de forma paralelizada e, sucessivamente, mixar em cada etapa posterior os resultados das mixagens da etapa anterior. A função de custos computacionais, nesse caso, seria $z_{mix}(n) \approx c_0 + c_1 \log_2 n_c n$, devido à realização paralela de todas as mixagens em cada uma das $\log_2 n_c$ etapas do processo.

4.4.4.1 Experimento

Nome do *plugin*	Intervalo da taxa de amostragem	Tipos de dados permitidos	Parâmetros
`equalizer-3bands`	[1, 2147483647]	S16LE, S32LE, F32LE, F64LE, ...	(Número de canais)

Tabela 4.9: *Parâmetros do elemento* `GstAudioMixer`.

Os parâmetros de configuração do elemento `GstAudioMixer` que influenciam o desempenho podem ser vistos na Tabela 4.9. Além dos tipos de dados indicados na tabela, o elemento pode processar vários outros tipos; lista-se na tabela apenas os tipos de dados comuns aos outros elementos discutidos nesta seção.

[41] O código-fonte relevante se encontra nos arquivos: `gstaudiomixer.c`, `gstaudiomixerorc-dist.c`.

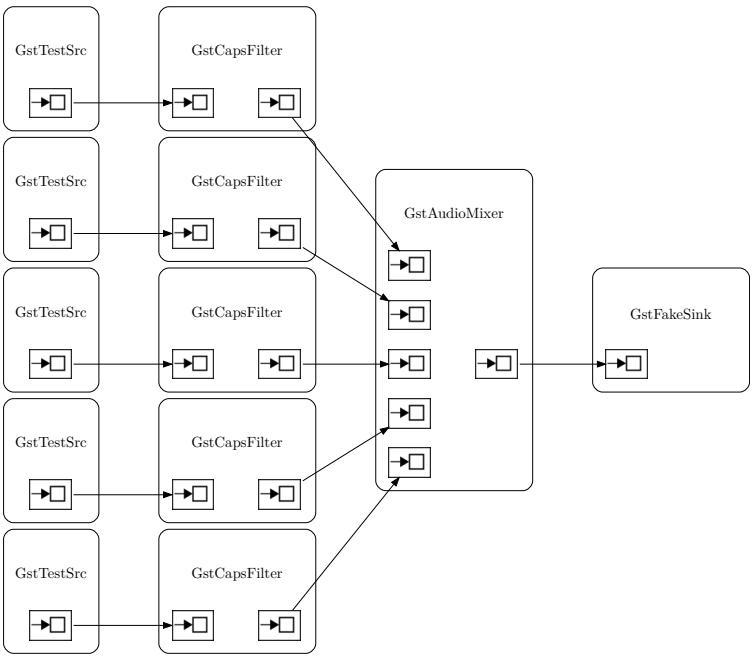

Figura 4.19: *Pipeline para a mensuração do desempenho do elemento* `GstAudioMixer`.

Para a mensuração do desempenho do elemento de mixagem (`GstAudioMixer`), constrói-se o pipeline como na Figura 4.19. Observa-se que as entradas do elemento `GstAudioMixer` são produzidas pelos elementos `GstTestSrc`, e que esses elementos podem ser executados paralelamente, uma vez que não há dependência entre eles, pois não há entradas de amostras nos elementos `GstTestSrc` e, decorrentemente, a execução de um destes elementos não precisa esperar a conclusão dos demais elementos `GstTestSrc`. No entanto, a entrada do elemento `GstAudioMixer` depende dos sinais de todas as faixas (em cada bloco de amostras), sendo executado somente depois dos elementos `GstTestSrc`.

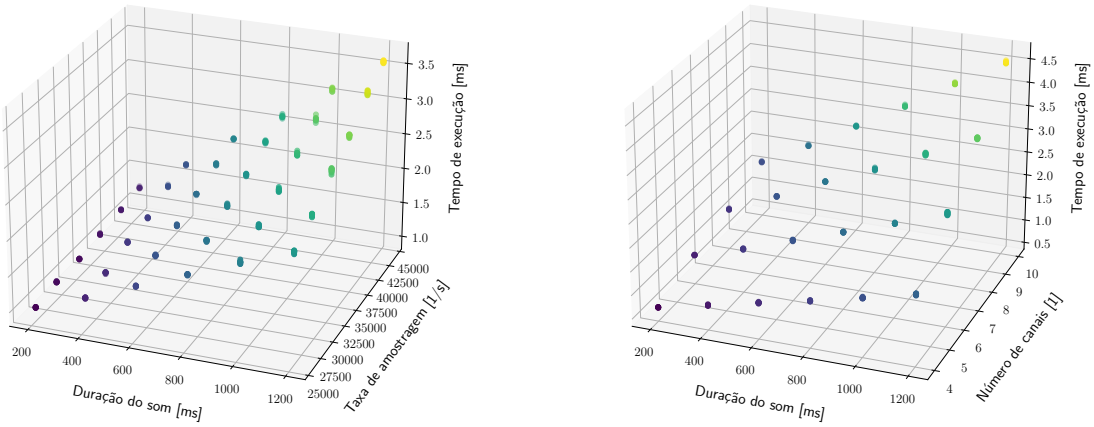

Figura 4.20: *Desempenho do elemento* `GstAudioMixer`*: a) tempo de processamento do elemento em função da duração do sinal e da taxa de amostragem; b) tempo de processamento em função da duração do sinal e do número de canais.*

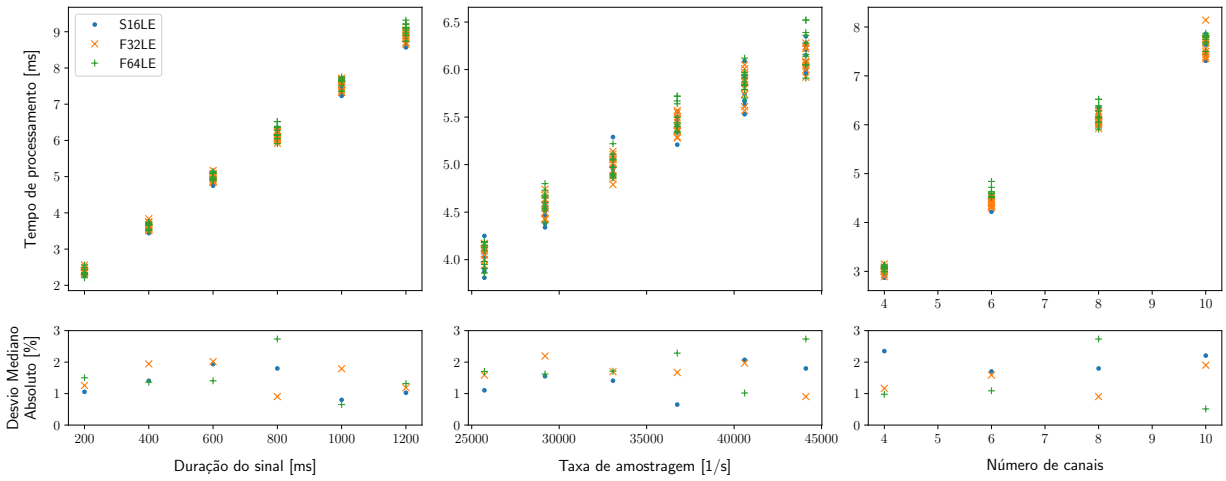

Figura 4.21: *Desempenho do elemento* `GstAudioMixer`*: a) (esquerda) tempo de processamento de um sinal com uma taxa de amostragem fixa em função da duração do sinal; b) (centro) tempo de processamento de um sinal de duração fixa com um número fixo de canais, em função da taxa de amostragem; c) (direita) tempo de processamento de um sinal de duração fixa com uma taxa de amostragem fixa, em função do número de canais.*

Os resultados do experimento com o elemento `GstAudioMixer` podem ser vistos nas Figuras 4.20 e 4.21. Fica evidente que os custos dependem de forma linear dos parâmetros duração do sinal (t), taxa de amostragem (r) e número de canais (n_c), e que os custos computacionais mensurados são compatíveis com a função de custos teórica. Consequentemente, pode-se empregar o mesmo método de ajuste utilizado com os outros elementos. Os valores estimados de c_0 e c_1 podem ser vistos na Tabela 4.10.

Tipo de dados	c_0 [ms]	c_1 [ms por faixa]	c_2 [ms por amostra e faixa]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s}, n_c = 8)$ [ms]
S16LE	0.018	0.1380	1.844e-05	4.0	7.63
S32LE	0.0	0.1423	1.862e-05	3.8	7.71
F32LE	0.0	0.1416	1.843e-05	3.5	7.64
F64LE	0.0	0.1373	1.888e-05	3.8	7.76

Tabela 4.10: *Coeficientes da função da custos* GstAudioMixer.

4.4.4.2 Flexibilização

No caso geral, o elemento `GstAudioMixer` não sugere um *trade-off* útil, uma vez que a diferença do desempenho entre tipos de dados diferentes é pequena. No entanto, poder-se-ia imaginar uma aplicação na qual certas faixas de som são incidentais ou descartáveis, pelo menos temporariamente; neste caso, a aplicação como um todo poderia ser flexibilizada quanto ao número de faixas de som incluídas na mixagem. Por exemplo, ao se descartar uma faixa da mixagem (o que economiza custos computacionais no elemento `GstAudioMixer`, visto que estes custos dependem do número de canais), poder-se-ia eliminar toda a cadeia de elementos anterior que produz essa faixa de entrada do elemento `GstAudioMixer`, o que pode ter um efeito muito relevante no desempenho da aplicação. Decorrentemente, deveriam ser considerados os custos computacionais totais da aplicação ao se introduzir este *trade-off* . Por esta razão, discute-se a flexibilização do número de canais na Seção 4.5 no contexto de uma aplicação completa, considerando também critérios para a exclusão ou inclusão de certas faixas, ou seja, de certas cadeias de elementos.

Em uma implementação que use o número de faixas para a realização de um *trade-off* deve-se sempre considerar que o processo de ligar e desligar faixas de som tem que ser controlado para evitar distorção grosseiras (descontinuidades ou *clicks*), como discutido na Seção 3.5.1. Além disso, seria possível controlar o ligar e desligar simultâneo de diversas fontes através de janelamentos cuidadosamente construídos para evitar distorções espectrais (como por exemplo *phasing*).

4.4.5 Elementos de entrada de áudio: `GstFileSrc`, `GstFlacParse`, `GstFlacDec` e `GstAudioConvert`

Em muitas aplicações de áudio usa-se funções de leitura de arquivos de áudio, como é o caso da aplicação `flexmix` (veja a Seção 4.5). A função de leitura de arquivos e a transformação dos dados lidos em blocos de amostras é realizada pelos elementos `GstFileSrc`, `GstFlacParse` e `GstFlacDec`. Para a adaptação do tipo de dados das amostras ao tipo de dados adequado à aplicação, pode ser utilizado o elemento `GstAudioConvert`. Esta seção trata principalmente do problema de converter uma representação de áudio em arquivo em um fluxo de amostras que pode ser utilizado nas cadeias de elementos de processamento. Por esta razão, não são considerados parâmetros de flexibilização, uma vez que o fluxo produzido deve representar fielmente o conteúdo do arquivo, e modificações posteriores sempre podem ser tratadas como elementos da cadeia de processamento.

Uma cadeia composta por estes elementos pode ser utilizada como uma unidade básica de construção de aplicações de áudio, visto que a funcionalidade desta cadeia representa um padrão que se repete. Decorrentemente, discute-se nesta seção os custos computacionais destes elementos em conjunto, ou seja, os custos da cadeia que contém estes elementos (*cadeia de leitura de arquivo*).

```
1  while (bytes_to_read > 0)
2    num_bytes = read (src, dst, bytes_to_read)
3    bytes_to_read -= num_bytes
4    dst += num_bytes
5    src += num_bytes
```

Algoritmo 4.6: *Resumo do algoritmo do elemento `GstFileSrc`.*

O Algoritmo 4.6 apresenta um resumo do código-fonte da função de processamento do elemento `GstFileSrc`. Na listagem, `src` representa um ponteiro aos dados do arquivo[42] e `dst` representa um ponteiro a um bloco de amostras na memória. A função executa uma leitura sequencial do arquivo. A função `read` preenche a memória (a partir do endereço `dst`) e devolve o número de bytes lidos; esse número é usado para diminuir o contador de leitura (`bytes_to_read`) e adiantar os ponteiros (`dst` e `src`). O algoritmo permite que a função `read` defina o número de bytes que são lidos em cada chamada da função, uma vez que esse número pode variar e depende do estado do hardware; essa função é construída de forma a não bloquear a execução do processo (inadmissível para processos de tempo real[43]), devolvendo a quantidade de bytes disponível para leitura imediata no hardware (até o limite superior de `bytes_to_read`).

A partir do Algoritmo 4.6 obtém-se uma função de custos da forma $z_{\text{fsrc}}(n) = c_0 + c_1 n$, ou equivalentemente $z_{\text{fsrc}}(t, r) = c_0 + c_1 tr$, onde n é o número de amostras, t é a duração do sinal, r é a taxa de amostragem e c_0 e c_1 são as constantes a estimar experimentalmente.

O elemento `GstFlacParse` tem a função de leitura do cabeçalho do arquivo e a configuração do próximo elemento (`GstFlacDec`). Visto que o cabeçalho possui um tamanho fixo (por arquivo), espera-se custos computacionais constantes ($\mathcal{O}(1)$) para o elemento `GstFlacParse`. O elemento `GstFlacDec` transforma os dados lidos pelo elemento `GstFileSrc` em cadeias de blocos de amostras (*buffers*) com seus metadados apropriados (entre outros:*timestamps*, formato das amostras, taxa de amostragem, etc.) considerando o alinhamento correto dos dados.

Para os elementos `GstFlacDec` e `GstAudioConvert` espera-se custos computacionais no máximo $\mathcal{O}(n)$, visto que possuem uma estrutura de processamento envolvendo uma quantidade fixa de operações por amostra.

[42] Com sistemas POSIX, arquivos abertos são acessados por endereços na memória virtual de um processo.
[43] Neste contexto, para processos de *soft realtime* (POSIX.1b) usando `SCHED_FIFO`; veja as Seções 2.4 e 4.3.

Uma vez que cada um dos elementos desta seção possui uma função linear de custos ($z(n) = c_0 + c_1 n$) ou uma função constante ($z = c_0$), pode-se supor que a função de custos da cadeia destes elementos é:

$$z_{\text{finput}}(n) = c_0 + c_1 n, \tag{4.15}$$

ou equivalentemente:

$$z_{\text{finput}}(t, r) = c_0 + c_1 t r, \tag{4.16}$$

onde n é o número de amostras, t é a duração do sinal, r é a taxa de amostragem e c_0 e c_1 são as constantes a estimar experimentalmente.

4.4.5.1 Experimento

Em contraste com os outros experimentos desta seção e como explicado anteriormente, o experimento com a *cadeia de leitura de arquivo* não investiga opções de flexibilização, mas apenas explora o desempenho desta cadeia em função de parâmetros operacionais, que são determinados principalmente pelo arquivo de áudio (taxa de amostragem, tipo de dados).

A execução do experimento deve seguir o mesmo padrão dos experimentos com os outros elementos desta seção; varia-se a duração do sinal, a taxa de amostragem e o tipo de dados das amostras.

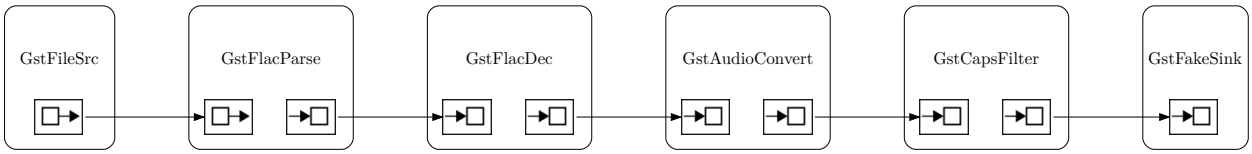

Figura 4.22: *Pipeline para a mensuração do desempenho da* cadeia de leitura de arquivo.

Para a mensuração do desempenho da *cadeia de leitura de arquivo*, constrói-se o pipeline como visto na Figura 4.22, usando arquivos de áudio pré-produzidos com tipos de dados e taxas de amostragens apropriados. Os arquivos pré-produzidos são armazenados na memória RAM (*RAM file system*) para evitar atrasos imprevisíveis no processamento, associados ao acesso ao disco rígido ou ao dispositivo SSD, o que é uma prática comum em aplicações de áudio[44].

[44] Vale lembrar que sistemas operacionais como Linux armazenam arquivos lidos na memória RAM (*caching*) para acelerar acessos subsequentes. Adicionalmente, o sistema operacional normalmente já copia para a memória RAM mais dados do arquivo do que são requeridos pela função `read`, na tentativa de antecipar acessos sucessivos para melhor eficiência da leitura de arquivos (Kerrisk, 2010, pág. 234).

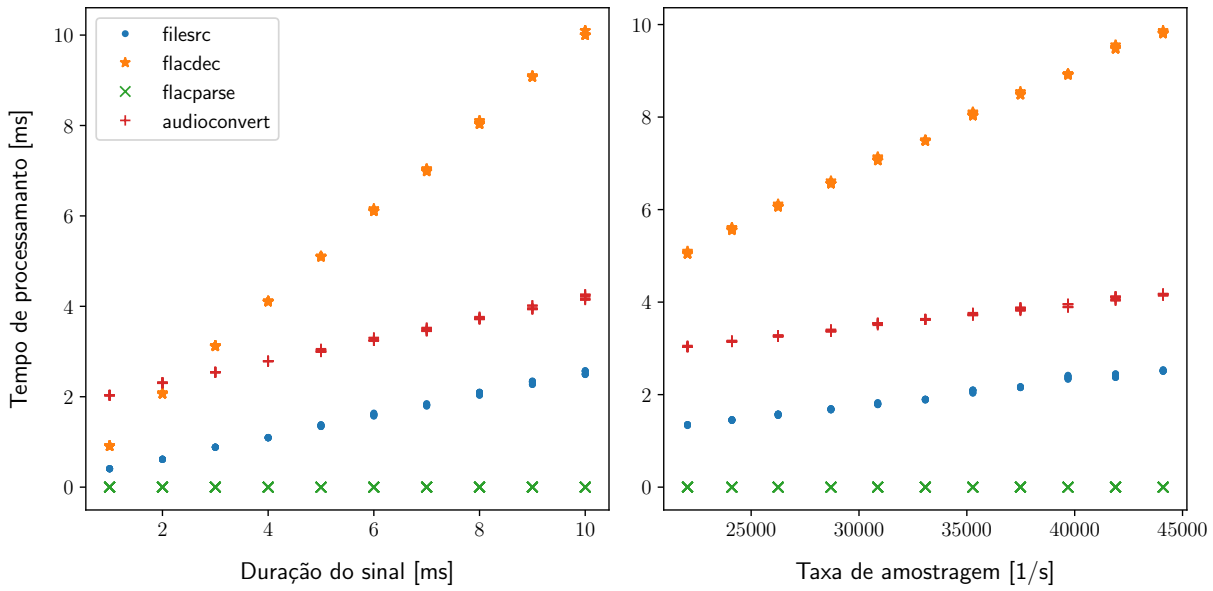

Figura 4.23: *Desempenho dos elementos da* cadeia de leitura de arquivo*: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do sinal; b) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem.*

A Figura 4.23 apresenta os resultados do experimento para cada um dos elementos, confirmando a suposição de que os elementos contidos na cadeia de leitura de arquivo possuem custos computacionais constantes (`GstFlacParse`) ou lineares (`GstFileSrc`, `GstFlacDec` e `GstAudioConvert`).

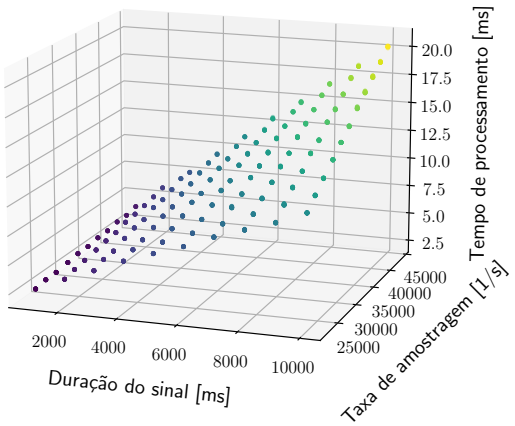

Figura 4.24: *Desempenho da* cadeia de leitura de arquivo*: tempo de processamento em função da duração do sinal e da taxa de amostragem.*

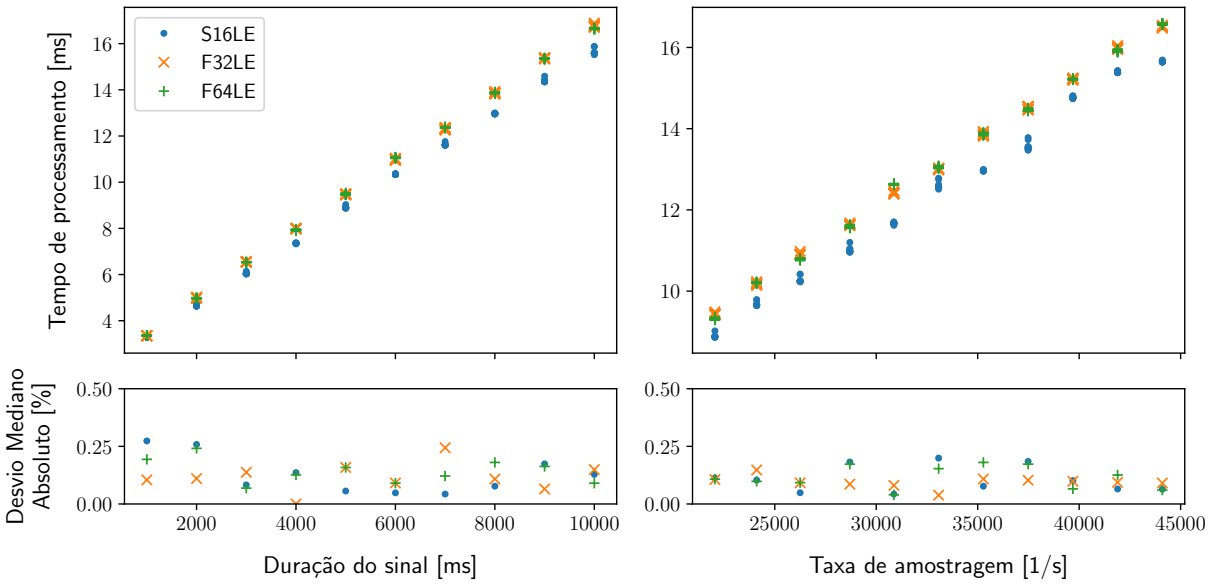

Figura 4.25: *Desempenho da* cadeia de leitura de arquivo*: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do sinal; b) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem.*

Nas Figuras 4.24 e 4.25 pode ser vista a soma dos custos dos elementos da cadeia em função da duração do sinal e da taxa de amostragem. Os resultados confirmam a função linear de custos (Equação 4.15).

Observa-se também que o uso de tipos de dados diferentes não leva a grandes variações de custos computacionais, mesmo considerando que os tamanhos dos arquivos de entrada sejam diferentes. Isto pode ser explicado pelo fato de que os tempos de acesso aos dados com tipos diferentes são similares, analogamente ao que ocorre com o elemento de amplificação (`GstAudioAmplify`, veja a Seção 4.4.1.2), pelo fato de que os arquivos de áudio são armazenados na memória RAM.

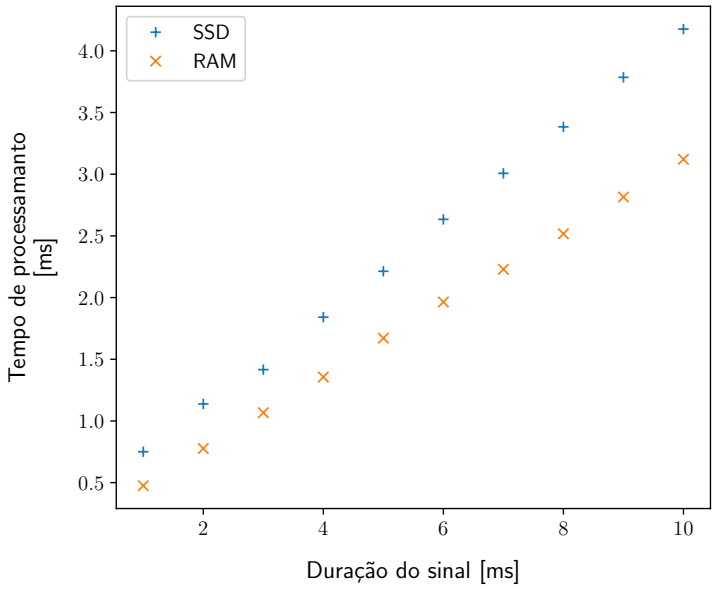

Figura 4.26: *Comparação do desempenho da* cadeia de leitura de arquivo *utilizando armazenamento em dispositivo SSD e em memória RAM.*

Na Figura 4.26 pode ser visto o desempenho do elemento `filesrc` em um experimento que usa

arquivos de áudio armazenados em um dispositivo SSD (sem *caching*) e um experimento que usa arquivos de áudio armazenados na memória RAM. Pode ser visto que a diferença entre os tempos de processamento cresce com a duração do sinal (tamanho do arquivo), o que confirma o acesso mais demorado a cada amostra quando se usa armazenamento em SSD sem *caching*.

Os valores dos coeficientes c_0 e c_1 da função de custos computacionais são obtidos pelo mesmo procedimento de otimização aplicado aos outros elementos; os valores podem ser vistos na Tabela 4.11.

Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
S16LE	2.440	3.883e-05	3.8	4.15
F32LE	2.129	4.235e-05	2.2	4.0
F64LE	2.165	4.283e-05	3.8	4.05

Tabela 4.11: *Coeficientes da função da custos* cadeia de leitura de arquivo.

4.4.6 Elemento de saída de áudio: `GstPulseSink`

`GstPulseSink` é o elemento que recebe um sinal de áudio e envia esses dados em blocos à placa de áudio pelo servidor de som `PulseAudio`[45]. O elemento envia os dados periodicamente e correspondentemente à taxa de amostragem. O experimento tem como objetivo a mensuração dos custos deste algoritmo.

Considerando que o tempo de execução de um pipeline com uma saída de som em tempo real é sempre igual à duração do sinal, não faz sentido usar uma ferramenta como `perf` que captura principalmente o tempo de execução do pipeline. Faz-se necessário utilizar uma ferramenta como `gst-top`, que é capaz de contar apenas o tempo de processamento de um elemento específico, desconsiderando os tempos de espera entre os envios de dados.

Uma vez que o processamento envolve uma quantidade fixa de operações por bloco, e consequentemente por amostra, espera-se uma função linear de custos computacionais da seguinte forma:

$$z_{\text{sndout}}(n) = c_0 + c_1 n, \tag{4.17}$$

ou equivalentemente:

$$z_{\text{sndout}}(t, r) = c_0 + c_1 t r, \tag{4.18}$$

onde n é o número de amostras, t é a duração do sinal, r é a taxa de amostragem e c_0 e c_1 são as constantes a estimar experimentalmente.

4.4.6.1 Experimento

O experimento investiga o desempenho do elemento `GstPulseSink` em função da duração do sinal, considerando apenas a taxa de amostragem de 44100 Hz e o tipo de dados 'F32LE', que são o padrão no sistema operacional utilizado nesta pesquisa (veja a Seção 4.3.3) e correspondem ao uso do elemento na aplicação `flexmix`.

Outra razão para esta escolha da taxa de amostragem igual à taxa padrão do sistema é que, no caso contrário, o elemento executa uma reamostragem para adaptar a taxa de amostragem de sua entrada à taxa do servidor de som (`PulseAudio`). Este comportamento, que leva a custos adicionais, não é desejado na aplicação `flexmix`, na qual a flexibilização é realizada com os próprios elementos de reamostragem de forma explícita (veja a Seção 4.4.7 e 4.5). O mesmo argumento é válido quanto ao parâmetro do tipo de dados ('F32LE').

[45] www.freedesktop.org/wiki/Software/PulseAudio.

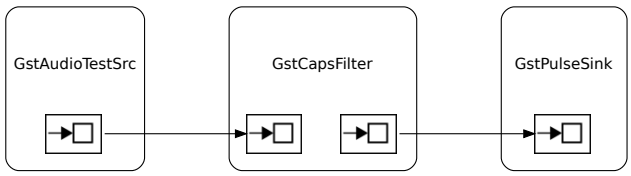

Figura 4.27: *Pipeline para a mensuração do desempenho do elemento* `GstPulseSink`.

Para a mensuração do desempenho do elemento `GstPulseSink`, constrói-se o pipeline como visto na Figura 4.27, na qual o elemento `GstAudioTestSrc` produz o número desejado de amostras com o tipo 'F32LE', e o elemento `GstCapsFilter` configura a taxa de amostragem (44100 Hz).

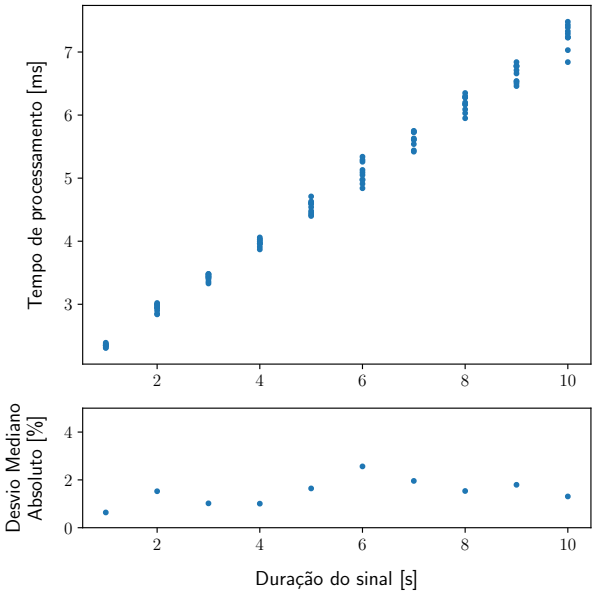

Figura 4.28: *Desempenho do elemento* `GstPulseSink`.

Na Figura 4.28 pode ser visto o resultado do experimento com o elemento `GstPulseSink`, resultado este que é compatível com a função de custos teórica (Equação 4.17). Observa-se que a variação dos valores mensurados, expressa pelo desvio mediano absoluto, é maior do que a variação para a maioria dos outros elementos investigados nesta seção. No entanto, esse fato deve ser refletido no contexto da aplicação de teste `flexmix`, considerando a proporção dos custos computacionais do elemento em relação aos custos computacionais totais da aplicação.

Os valores dos coeficientes c_0 e c_1 da função de custos computacionais são obtidos, como das outras vezes, pelo mesmo procedimento de otimização aplicado aos outros elementos; os valores podem ser vistos na Tabela 4.12.

Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
F32LE	1.863	1.224e-05	0.4	2.4

Tabela 4.12: *Coeficientes da função de custos do elemento* `GstPulseSink`.

Vale lembrar que no caso em que os blocos de amostras não cheguem na entrada do elemento `GstPulseSink` dentro do prazo para o processamento em tempo real, o elemento não consegue enviar as amostras para o servidor de som (`PulseAudio`). Em consequência, o som na saída da placa de áudio é interrompido e o valor das amostras ausentes é considerado nulo, o que leva a artefatos sonoros indesejáveis. O elemento `GstPulseSink` não oferece métodos de extrapolação do

sinal para amenizar este problema, como por exemplo a repetição do último bloco, como ocorre em outros servidores de som.

4.4.7 Elemento de reamostragem: `GstAudioResample`

A reamostragem é o processo de converter um bloco de amostras com uma dada taxa de amostragem em um outro bloco de amostras que representa o mesmo sinal de áudio, no entanto, com outra taxa de amostragem (e portanto, com um tamanho diferente). O elemento `GstAudioResample` é o elemento do `Gstreamer` que executa esta função usando um fator de reamostragem (quociente da taxa de amostragem desejada na saída em relação à taxa de amostragem do sinal da entrada).

Como mostram, por exemplo, Crochiere e Rabiner (Crochiere e Rabiner, 1983b, pág. 39), a reamostragem de um bloco de amostras por um fator racional f envolve os seguintes passos:

1. *decomposição do fator de reamostragem:* encontrar valores para $L, M \in \mathbb{N}$ de tal forma que $f = L/M$ e L seja o menor possível,
2. *upsampling:* aumentar o número de amostras pelo fator L, inserindo $L - 1$ zeros entre as amostras do sinal original,
3. *filtragem:* processar o sinal por um filtro de convolução do tipo passa-baixas para eliminar as componentes incompatíveis com a nova taxa de amostragem, e
4. *downsampling:* reduzir o número de amostras pelo fator M, preservando apenas 1 amostra a cada M amostras do sinal processado.

A implementação do elemento `GstAudioResample` segue a abordagem supracitada. Os custos computacionais do processo correspondem, em sua maior parte, aos custos da execução do filtro, uma vez que os outros passos representam apenas simples operações de transferência de memória. A partir da descrição de cada um dos passos do algoritmo, estabelece-se a seguinte função de custos:

$$z_{res} = c_0 + c_{10}n_1 + c_{11}n_1 + c_2 n_2, \tag{4.19}$$

onde n_1 é o número de amostras depois do *upsampling*, n_2 é o número de amostras de saída do algoritmo (depois do *downsampling*) e c_0, c_{10}, c_{11}, c_2 são constantes.

Pode-se combinar os termos dos passos 2 e 3, uma vez que dependem do mesmo parâmetro n_1, substituindo $c_{10} + c_{11}$ por c_1:

$$z_{res} = c_0 + c_1 n_1 + c_2 n_2. \tag{4.20}$$

Seja n_0 o número de amostras de entrada do algoritmo, então tem-se $n_1 = Ln_0$ e $n_2 = L/Mn_0$ e, decorrentemente,

$$\begin{aligned} z_{res} &= c_0 + c_1 L n_0 + c_2 \frac{L}{M} n_0 \\ &= c_0 + c_1 L n_0 + c_2 f n_0 \\ &= c_0 + (c_1 L + c_2 f)\, n_0. \end{aligned} \tag{4.21}$$

O número de amostras que devem ser filtradas depende do fator de *upsampling* L e, decorrentemente, da decomposição do fator de reamostragem em um quociente de números inteiros, $f = L/M$. Consequentemente, pode-se obter valores de L muito diferentes para fatores f próximos, dependendo do método de decomposição; esse efeito e estratégias relacionadas são discutidos na próxima seção.

4.4.7.1 Experimento

A reamostragem de sinais de áudio poderia em certos contextos ser empregada com uma função musical, porém nesta pesquisa sua utilização corresponde a um método de flexibilização de cadeias

de elementos de processamento (veja a Seção 3.1.2). Como os custos computacionais de quase todos os elementos crescem com o número de amostras, fica evidente que é possível controlar os custos computacionais do processamento de uma cadeia de elementos modificando sua taxa de amostragem.

Com essa abordagem, constrói-se cadeias de elementos que são encapsuladas, tendo elementos de reamostragem em suas entradas e saídas. Com isso, os elementos contidos na cadeia podem processar os sinais usando uma outra taxa de amostragem, independente do restante da aplicação, enquanto os elementos de reamostragem adaptam as taxas dos sinais de entrada e saída de acordo com as taxas usadas pelos elementos antecessores e sucessores da cadeia encapsulada. Em função desta abordagem, esta pesquisa enfoca apenas *pares* de elementos de reamostragem.

Uma vez que os custos computacionais de um elemento de reamostragem podem variar muito em função de fatores de reamostragem próximos, propõe-se para esta pesquisa a utilização de um número limitado de taxas de amostragem e, assim, de fatores de reamostragem. Estas taxas de amostragem são selecionadas de modo a otimizar o desempenho dos elementos de reamostragem. Por isso, o experimento é dividido em duas partes: (a) a seleção de taxas de amostragem admissíveis; (b) a modelagem dos custos computacionais em função dos parâmetros de processamento.

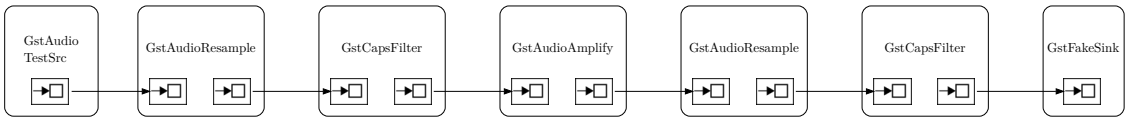

Figura 4.29: *Pipeline para a mensuração do desempenho do elemento* `GstAudioResample`.

Na Figura 4.29 pode-se ver o pipeline para a mensuração dos custos computacionais de um par de elementos de reamostragem (`GstAudioResample`).

Para separar a mensuração dos elementos de reamostragem de entrada e saída no pipeline, inclui-se um elemento intermediário de processamento de áudio (`GstAudioAmplify`), evitando assim eventuais efeitos mútuos indesejáveis que podem ocorrer entre dois elementos de reamostragem concatenados diretamente, devido ao gerenciamento de memória dentro da aplicação `Gstreamer`. Os elementos `GstCapsFilter` têm a função de configurar dos parâmetros de tamanho dos blocos, tipo de dados e taxa de amostragem, como visto nos experimentos com os outros elementos. Os elementos de reamostragem separam a cadeia de elementos em três segmentos que podem processar seus sinais com taxas de amostragem diferentes, sendo que a colocação dos elementos `GstCapsFilter` em cada segmento permite o controle da taxa de amostragem nestes segmentos.

Seleção de taxas de amostragem admissíveis Para a construção da função de custos de um par de elementos de reamostragem utiliza-se a Função 4.21, que depende do número de amostras na entrada do pipeline (n_0) e do número de amostras de saída do primeiro elemento (n_1), que é sempre igual ao número de amostras de entrada do segundo elemento de reamostragem. Vale observar que as taxas de amostragem na entrada do primeiro elemento e na saída do segundo elemento de reamostragem também são iguais, o que implica que os fatores de reamostragem destes elementos têm que ser recíprocos. Para os custos de cada elemento (z_1 e z_2) tem-se:

$$z_1 = c_0 + (c_1 L_1 + c_2 f_1)\, n_0, \tag{4.22}$$
$$z_2 = c_0 + (c_1 L_2 + c_2 f_2)\, n_1, \tag{4.23}$$

onde L_1 e L_2 são os fatores de *upsampling* dos respectivos elementos de reamostragem e f_1 e f_2 são os fatores de reamostragem. Pode-se expressar n_1 em função de n_0 como $n_1 = f_1 n_0$:

$$z_2 = c_0 + (c_1 L_2 + c_2 f_2)\, f_1 n_0,$$
$$z_2 = c_0 + (c_1 f_1 L_2 + c_2)\, n_0, \tag{4.24}$$

usando $f_2 f_1 n_0 = n_0$, já que $f_1 f_2 = 1$, uma vez que o pipeline tem o mesmo número de amostras na entrada e na saída.

Como soma dos custos dos elementos (Equações 4.22 e 4.24) obtém-se:

$$\begin{aligned} z_1 + z_2 &= 2c_0 + (c_1 L_1 + c_2 f_1 + c_1 f_1 L_2 + c_2)\, n_0 \\ &= 2c_0 + (c_1\,(L_1 + f_1 L_2) + c_2\,(f_1 + 1))\, n_0. \end{aligned} \tag{4.25}$$

Essa função dos custos computacionais possui a mesma característica que a função dos custos computacionais de um elemento de reamostragem genérico (Equação 4.21), ou seja, que os custos computacionais, dependentes de L e M, podem variar muito para fatores de reamostragem f próximos, porque os valores de L e M são calculados em função de $f = L/M$, onde L e M têm que ser necessariamente números inteiros.

Esta característica destaca a necessidade de se selecionar certas taxas de amostragem para serem usadas em um mecanismo de flexibilização. Para esta abordagem, supõe-se que a taxa de amostragem dos sinais de entrada e de saída do pipeline seja fixa, e que varie apenas a taxa de amostragem interna, ou seja, entre os elementos de reamostragem. O experimento consiste então na execução do pipeline (Figura 4.29) varrendo a faixa de valores úteis para tal taxa de amostragem com os outros parâmetros de processamento fixos.

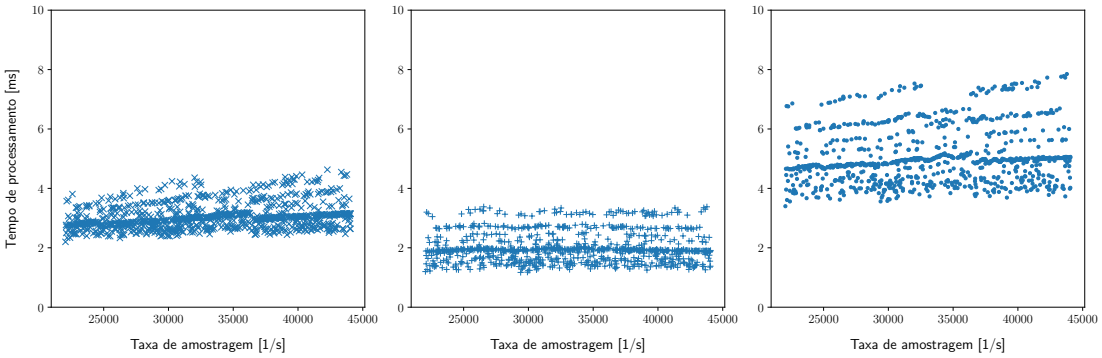

Figura 4.30: *Desempenho dos elementos de reamostragem* `GstAudioResample`*; a) (esquerda) tempo de processamento do downsampling; b) (centro) tempo de processamento do upsampling; c) (direita) soma dos tempos de processamento a) e b).*

Na Figura 4.30 pode-se ver o resultado do experimento onde cada valor mensurado corresponde a uma taxa de amostragem interna da cadeia. Confirma-se a observação de que há custos muito diferentes para taxas de amostragem próximas.

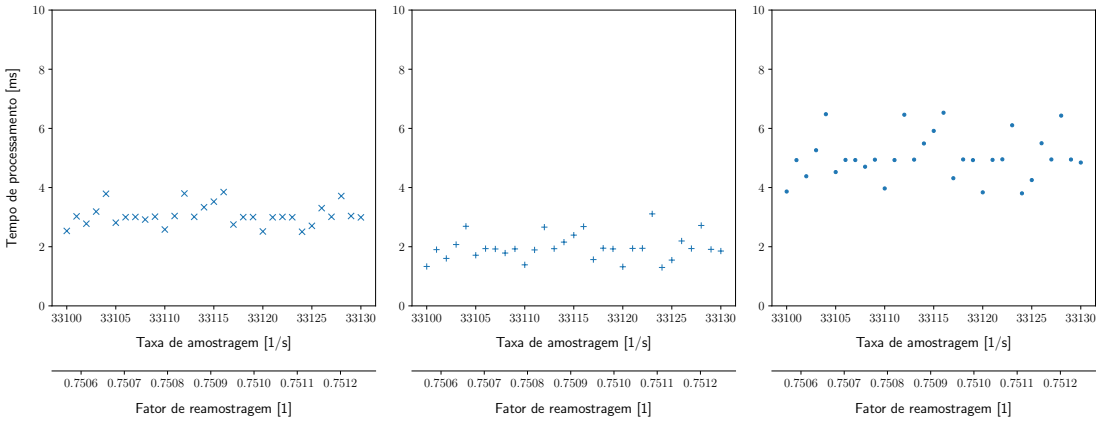

Figura 4.31: *Desempenho dos elementos de reamostragem* `GstAudioResample`*; a) (esquerda) tempo de processamento do downsampling; b) (centro) tempo de processamento do upsampling; c) (direita) soma dos tempos de processamento a) e b).*

Na Figura 4.31, que mostra um recorte dos resultados apresentados na Figura 4.30, pode ser visto o desempenho dos elementos de reamostragem para uma faixa estreita de fatores de reamostragem, que permite observar os efeitos da decomposição de $f = L/M$.

No entanto, fica evidente que existe uma tendência global linear da soma dos custos computacionais dos dois elementos de reamostragem (Equação 4.25). Essa tendência se explica pelo fato de que esta função tem a forma principal $z = c_i + c_{ii}n$, na qual a expressão $c_{ii} = c_1(L_1 + f_1 L_2) + c_2(f_1 + 1)$ produz valores independentes do número de amostras e dentro de uma faixa com limites fixos (ou seja, é aproximadamente constante).

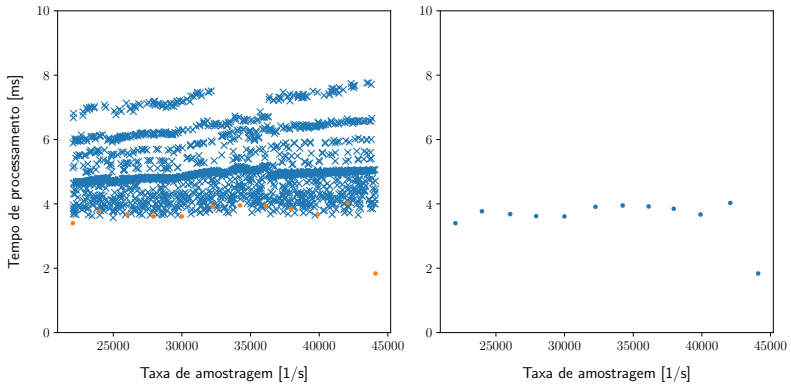

Figura 4.32: *Seleção de taxas de amostragem admissíveis para o elemento* `GstAudioResample`*.*

A Figura 4.32 ilustra o processo da seleção de taxas de amostragem admissíveis, que consiste nos seguintes passos: (a) divisão da faixa de valores úteis em segmentos; (b) escolha da taxa de amostragem com menores custos computacionais em cada segmento.

Surpreendentemente, os custos computacionais em função das taxas de amostragem selecionadas têm uma tendência constante, e não linear como sugere a fórmula. A principal exceção é o caso em que as taxas de amostragem dentro e fora da cadeia são iguais e os custos computacionais são mínimos, uma vez que não há reamostragem (na Figura 4.32, isso ocorre com a taxa de 44100 Hz). Nos experimentos subsequentes utiliza-se apenas o conjunto das taxas de amostragem selecionadas.

Nome do *plugin*	Intervalo da taxa de amostragem	Tipos de dados permitidos	Outros parâmetros
`GstAudioResample`	[1, 2147483647]	S8, S16LE, S32LE, F32LE, F64LE, ...	qualidade da reamostragem: 1 - 10

Tabela 4.13: *Parâmetros do elemento `GstAudioResample`.*

Mapeamento dos parâmetros do processamento aos custos computacionais Os parâmetros de configuração do elemento `GstAudioResample` que influenciam o desempenho podem ser vistos na Tabela 4.9. O parâmetro de qualidade da reamostragem, que pode ter valores na faixa $(1, 10)$, é mapeado em outros parâmetros do processo de reamostragem (como, por exemplo, a forma da janela usada para o filtro e o valor de *oversampling*). A relação entre esses parâmetros e o desempenho do elemento poderia ser obtida por análise dos respectivos códigos-fonte, porém a diferença de abordagens entre os diversos métodos tornaria essa análise demasiadamente complexa. Por isso, optou-se por uma abordagem experimental, estimando independentemente os parâmetros de funções de custos associadas a cada método (ou seja, a cada valor do parâmetro qualidade da reamostragem).

A execução do experimento para estimar este mapeamento segue o mesmo padrão dos experimentos com os outros elementos deste capítulo, varrendo o espaço dos seguintes parâmetros: duração do sinal, taxas de amostragem selecionadas e qualidade da reamostragem.

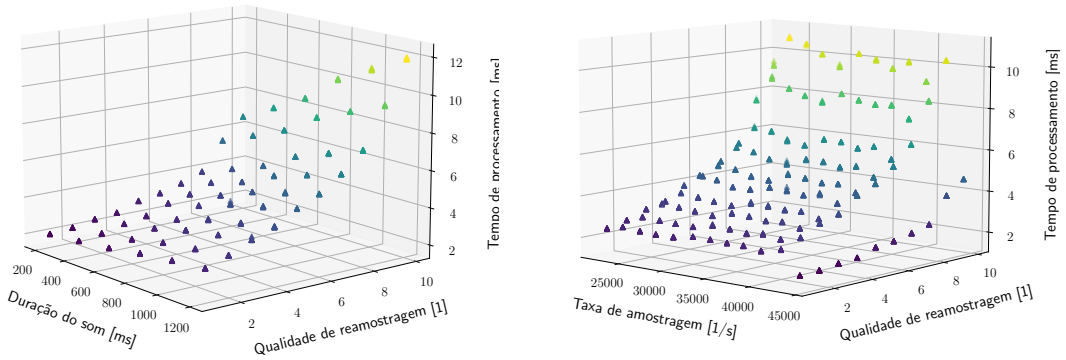

Figura 4.33: *Desempenho do elemento `GstAudioResample`: Tempo de processamento do elemento em função da taxa de amostragem e da qualidade de reamostragem.*

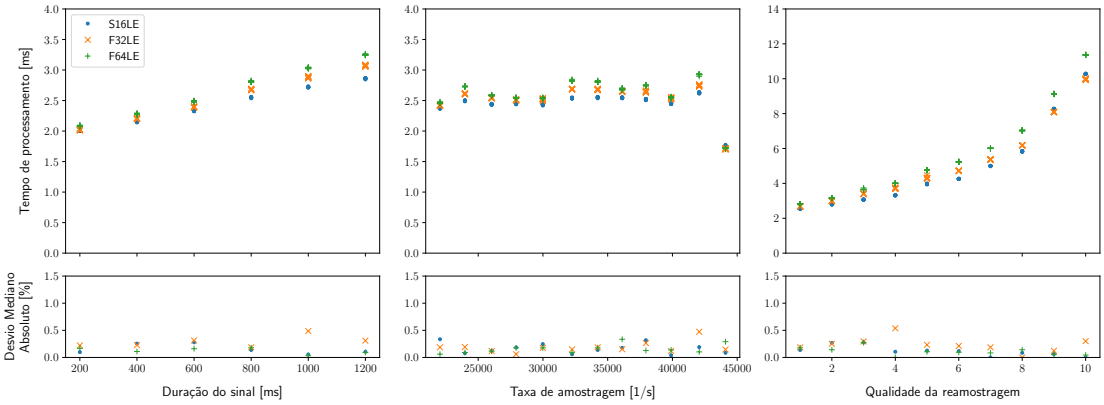

Figura 4.34: *Desempenho de pares de elementos de reamostragem* `GstAudioResample`*: a) (esquerda) tempo de processamento de um sinal com uma taxa de amostragem fixa em função da duração do sinal; b) (centro) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem; c) (direita) tempo de processamento de um sinal de duração fixa com uma taxa de amostragem fixa, em função da qualidade de reamostragem.*

Nas Figuras 4.33 e 4.34 podem ser vistos os resultados do experimento do mapeamento dos parâmetros nos custos computacionais. Pode ser observado que:

- a diferença entre os custos em função do tipo de dados é pequena;
- os custos computacionais em função da duração do sinal são lineares, supondo fixas a taxa de reamostragem e a qualidade de reamostragem;
- os custos computacionais em função da taxa de amostragem são aproximadamente constantes, supondo fixas a duração do sinal e a qualidade de reamostragem, com a exceção do caso em que não há mudança da taxa de amostragem, onde os custos são bem mais baixos;
- os custos computacionais crescem em função da qualidade de amostragem, supondo os demais parâmetros fixos.

Quando os custos computacionais desta combinação de dois elementos são pequenos em relação aos custos computacionais de uma aplicação completa, a imprecisão da função de custos computacionais destes dois elementos tem menos impacto na imprecisão do resultado final. Consequentemente, justifica-se neste caso a simplificação de supor que os custos computacionais em função da taxa de amostragem sejam aproximadamente constantes, supondo fixas a duração do sinal e a qualidade da reamostragem.

Mesmo que a função de custos tenha a forma $z = c_0 + c_1 n$, faz-se necessário estimar valores distintos das constantes c_0 e c_1 para cada valor do parâmetro de qualidade da reamostragem. Uma simplificação possível seria o uso de uma função polinomial para ajustar a curva dos custos computacionais em função do parâmetro de qualidade (veja a Figura 4.34c). Entretanto, nesta pesquisa, propõe-se como simplificação a utilização de apenas 4 níveis de qualidade, a fim de diminuir o número de coeficientes distintos necessários. A seleção destes níveis é realizada de modo a cobrir uma ampla faixa de custos computacionais (que serão úteis na flexibilização).

Obtém-se dessa maneira as seguintes funções de custos:

$$z_{2\text{res}}(t, r_{\text{out}}) = c_{0,q,r_{\text{in}},r_{\text{out}}} + c_{1,q,r_{\text{in}},r_{\text{out}}} t r_{\text{out}} \tag{4.26}$$

onde q é a qualidade da reamostragem, t é a duração do sinal, r_{in} é a taxa de amostragem entre os elementos de reamostragem ('dentro da cadeia'), r_{out} é a taxa de amostragem na entrada e na saída do pipeline, e $c_{0,q,r_{\text{in}},r_{\text{out}}}$ e $c_{1,q,r_{\text{in}},r_{\text{out}}}$ são os coeficientes em função do parâmetro q e das taxas de amostragem , r_{in} e r_{out}. Vale lembrar que os coeficientes são os mesmos em todos os casos em

que $r_{\text{in}} \neq r_{\text{out}}$, sendo diferentes apenas quando $r_{\text{in}} = r_{\text{out}}$, supondo fixo o parâmetro de qualidade q.

Os valores calculados dos coeficientes pelo método de ajuste de função podem ser vistos na Tabela 4.14.

Tipo de dados	Qualidade	Qualidade Taxa de amostragem [Hz]	c_0 [ms]	c_1 [ms/amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
S16LE	1	< 44100	1.811	3.247e-05	1.2	3.24
		$= 44100$	1.657	2.962e-06	0.1	1.79
	3	< 44100	2.208	4.55e-05	1.2	4.21
		$= 44100$	1.914	3.258e-06	0.2	2.06
	6	< 44100	3.078	6.649e-05	1.3	6.01
		$= 44100$	2.534	2.581e-06	0.1	2.65
	10	< 44100	9.513	12.22e-5	0.8	14.9
		$= 44100$	6.703	2.986e-06	0.1	6.83
F32LE	1	< 44100	1.749	3.8e-05	0.8	3.42
		$= 44100$	1.612	2.714e-06	0.2	1.73
	3	< 44100	1.98	5.768e-05	1.3	4.52
		$= 44100$	1.767	2.792e-06	0.0	1.89
	6	< 44100	2.661	8.385e-05	1.4	6.36
		$= 44100$	2.226	2.809e-06	0.1	2.35
	10	< 44100	6.028	18.32e-5	1.1	14.11
		$= 44100$	4.504	2.693e-06	0.0	4.62
F64LE	1	< 44100	1.736	4.123e-05	0.8	3.55
		$= 44100$	1.621	2.774e-06	0.4	1.74
	3	< 44100	1.994	6.706e-05	1.0	4.95
		$= 44100$	1.774	2.702e-06	0.1	1.89
	6	< 44100	2.671	10.59e-5	1.6	7.34
		$= 44100$	2.237	2.729e-06	0.2	2.36
	10	< 44100	6.067	23.67e-5	1.5	16.51
		$= 44100$	4.556	2.721e-06	0.1	4.68

Tabela 4.14: *Coeficientes da função de custos dos elementos de reamostragem.*

4.4.7.2 Flexibilização

A aplicação de um par de elementos de reamostragem no começo e no fim de uma cadeia de elementos de processamento de áudio tem como objetivo a flexibilização dos custos computacionais dentro dessas cadeias de elementos. Essa abordagem é sensata apenas para pipelines cujos custos são maiores que os custos dos elementos de reamostragem. Uma vez que os custos computacionais de todos elementos investigados dependem do número de amostras do sinal, pode-se controlar os custos da cadeia pela taxa de amostragem interna, de acordo com a equação $n_{\text{in}} = tr_{\text{in}}$, sendo t a duração (fixa) do sinal.

Observe que a flexibilização anterior não inclui os próprios elementos de reamostragem, mas apenas os elementos internos à cadeia. Adicionalmente, pode-se utilizar o parâmetro de qualidade da reamostragem para flexibilizar os custos computacionais dos elementos de reamostragem. Destaca-se que os custos computacionais com a qualidade máxima de reamostragem são aproximadamente 5x maiores do que os custos com a qualidade mínima permitida pelo algoritmo (Figura 4.34c).

4.4.8 Elemento de convolução: `GstFlexfir`

Nesta seção, as implementações das propostas de um filtro FIR flexibilizado da Seção 3.5.1 são analisadas e discutidas. As implementações das três abordagens apresentadas são investigadas quanto ao seu desempenho, como os outros elementos deste capítulo, mas também quanto à qualidade percebida. O elemento `GstFlexfir0` corresponde à implementação da abordagem de flexibilização dos últimos coeficientes do filtro, o elemento `GstFlexfir1` corresponde à abordagem da flexibilização dos coeficientes com menor valor, e o elemento `GstFlexfir2` corresponde à abordagem que usa versões do filtro obtidas por otimização linear. A consideração do desempenho e da qualidade percebida permite escolher a melhor implementação para ser usada na aplicação de teste `flexmix`.

O elemento `GstFlexfir0` utiliza o Algoritmo 3.2, apresentado na Seção 3.5.1. O Algoritmo 4.7 abaixo representa um resumo do processamento implementado nos elementos `GstFlexfir1` e `GstFlexfir2`, que é idêntico para esses dois elementos, sendo a única diferença a escolha dos coeficientes do filtro em tempo de configuração.

```
1  // x[]: bloco das amostras de entrada
2  // y[]: bloco das amostras de saída
3  // aind[]: índices dos coeficientes dos filtros
4  // aval[]: valores dos coeficientes
5
6  for each sample index i:
7    add_to_history (x[i])
8    for each coef index k:
9      y[i] += aval[k] * get_history_at (aind[k]) // aval[k]*x[i-aind[k]]
```

Algoritmo 4.7: *Resumo do algoritmo dos elementos `GstFlexfir1` e `GstFlexfir2`*

A representação do filtro acima é esparsa, ou seja, omite coeficientes que são nulos e, decorrentemente, depende não só dos valores dos coeficientes (aval[k]) mas também de seus respectivos índices (aind[k]). Por isso existe a indireção adicional (`get_history_at (aind[k])` ao invés de `get_history_at (k)`) na linha 9 do Algoritmo. Essa é a única diferença da implementação do elemento `GstFlexfir0` em relação ao Algoritmo 4.7: naquele elemento os índices são usados diretamente (k vai de 0 até o número de coeficientes menos 1), enquanto os elementos `GstFlexfir1` e `GstFlexfir2` usam um array (`aind[]`) de indireção, para armazenar os índices correspondentes aos coeficientes (esparsos) do filtro FIR.

A Função de custos 3.15, $z_{fir}(n) = c_0 + c_1 n + c_2 M n$, dada na Seção 3.5.1 é válida para todos os três elementos. Consequentemente, obtém-se a seguinte função de custos em relação à duração do sinal e à taxa de amostragem:

$$z_{fir}(t, r) = c_0 + c_1 tr + c_2 Mtr. \tag{4.27}$$

No experimento, são esperados valores maiores de c_2 para os elementos `GstFlexfir1` e `GstFlexfir2` em relação ao `GstFlexfir0`, em função da indireção na linha 9 do Algoritmo 4.7.

Ainda assim, é necessário considerar que o filtro (representado pelos coeficientes) deve ser adaptado a taxas de amostragens diferentes para preservar suas características no plano tempo-frequência. A reamostragem do vetor dos coeficientes tem como resultado um vetor de tamanho diferente e com coeficientes diferentes. O tamanho do vetor de coeficientes é proporcional à taxa

de amostragem associada; obtém-se assim a seguinte função:

$$M(r) = M_{r_0}\frac{r}{r_0}, \tag{4.28}$$

onde M_{r_0} é o número de coeficientes associado à taxa de amostragem r_0 (que pode ser flexibilizado), e r é a taxa de amostragem do filtro adaptado. Decorrentemente, obtém-se a seguinte função de custos:

$$z_{fir}(t, r) = c_0 + c_1 tr + c_2 \frac{M_{r_0}}{r_0} tr^2, \tag{4.29}$$

sendo c_0, c_1 e c_2 as constantes a se estimar experimentalmente. Observa-se que nessa equação o último termo depende linearmente do tempo t e quadraticamente da taxa de amostragem r, o que se deve à presença de r tanto na substituição $n \to tr$ quanto na reamostragem dos coeficientes do filtro (Equação 4.28).

4.4.8.1 Experimento

Nome do *plugin*	Intervalo da taxa de amostragem	Tipos de dados permitidos	Outros parâmetros
`GstFlexfir0`, `GstFlexfir1`, `GstFlexfir2`	[7350, 44100]	F32LE	número de coeficientes

Tabela 4.15: *Parâmetros dos elementos `GstFlexfir0`, `GstFlexfir1` e `GstFlexfir2`.*

Na Tabela 4.15 podem ser vistos os parâmetros de configuração dos elementos `GstFlexfir0`, `GstFlexfir1` e `GstFlexfir2`. Uma vez que os experimentos com os outros elementos indicam que o tipo de dados não é um parâmetro útil para a flexibilização, os elementos `GstFlexfirN` implementam apenas o tipo ponto flutuante de 32 bits (F32LE), que também funciona com todos os outros elementos investigados neste capítulo.

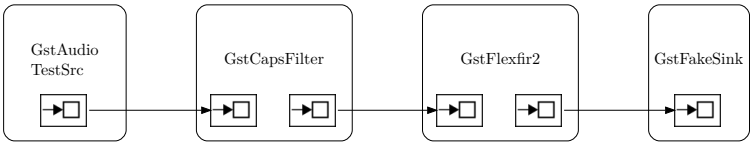

Figura 4.35: *Pipeline para a mensuração do desempenho dos elementos `GstFlexfir0`, `GstFlexfir1` e `GstFlexfir2`.*

Para a mensuração do desempenho dos elementos `GstFlexfir0`, `GstFlexfir1` e `GstFlexfir2`, constrói-se o pipeline como na Figura 4.35. No experimento, o pipeline é executado com diferentes configurações para os parâmetros, sendo registrado a cada vez o tempo de execução do elemento. Os parâmetros experimentais *duração de sinal* e *taxa de amostragem* varrem o espaço das configurações como nos outros experimentos deste capítulo; além destes, o parâmetro de flexibilização *número de coeficientes do filtro* também é variado experimentalmente no intervalo $(128, 1024)$.

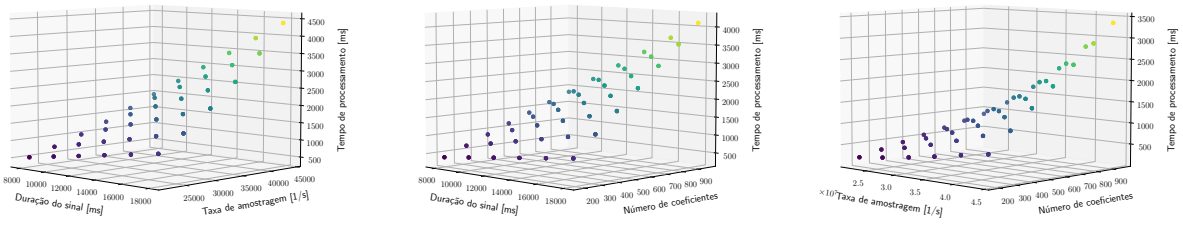

Figura 4.36: *Desempenho do elemento* `GstFlexfir1`*: a) (esquerda) tempo de processamento com o número de coeficientes fixo em função da duração do sinal e da taxa de amostragem; b) (centro) tempo de processamento de um sinal com a taxa de amostragem fixa em função da duração do sinal e do número de coeficientes; c) (direita) tempo de processamento de um sinal de duração fixa em função da taxa de amostragem e do número de coeficientes.*

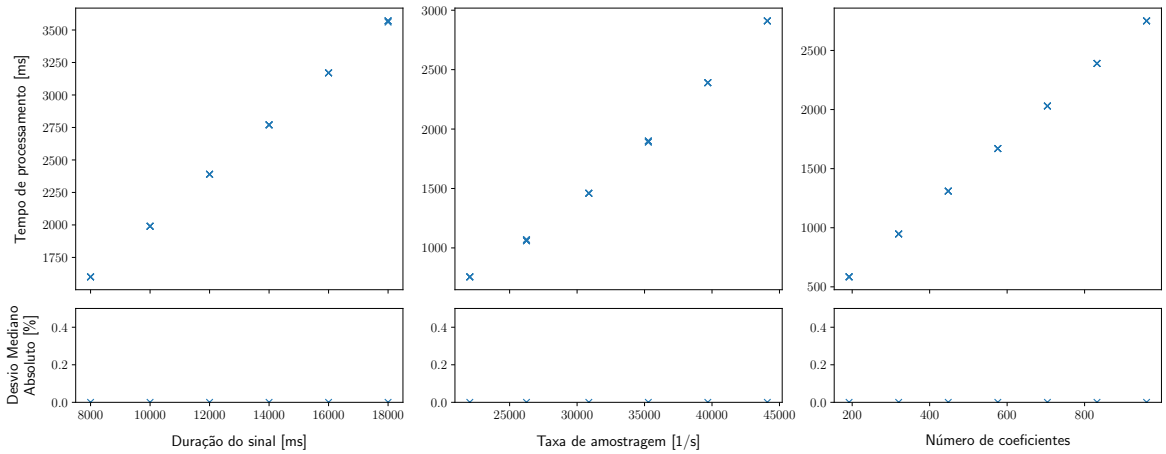

Figura 4.37: *Desempenho do elemento* `GstFlexfir1`*: a) (esquerda) tempo de processamento com a taxa de amostragem fixa em função da duração do sinal; b) (centro) tempo de processamento de um sinal de duração fixa com o número de coeficientes fixo, em função da taxa de amostragem; c) (direita) tempo de processamento de um sinal de duração fixa, em função da taxa de amostragem.*

Nas Figuras 4.36 e 4.37 podem ser vistos os resultados do experimento com o elemento `GstFlexfir1`. Vale lembrar que estes resultados são válidos também para o elemento `GstFlexfir2` visto que as diferenças entre os elementos correspondem à seleção dos coeficientes, o que não impacta nos custos computacionais, visto que os códigos-fonte desses elementos são idênticos, como visto anteriormente.

Observa-se que estes resultados são compatíveis com a função de custos computacionais teórica (Equação 4.29). Mesmo assim, a influência do termo quadrático desta função ($c_2 \frac{M_0}{r_0} tr^2$) parece pequena (veja a Figura 4.37b) no intervalo de valores testados, onde os custos parecem se comportar linearmente. Isso significa que poder-se-ia eliminar este termo da função dependendo dos intervalos de valores (para r_{in}) usados em uma aplicação real e dependendo dos erros de estimação de custos

computacionais admissíveis no contexto da aplicação real.

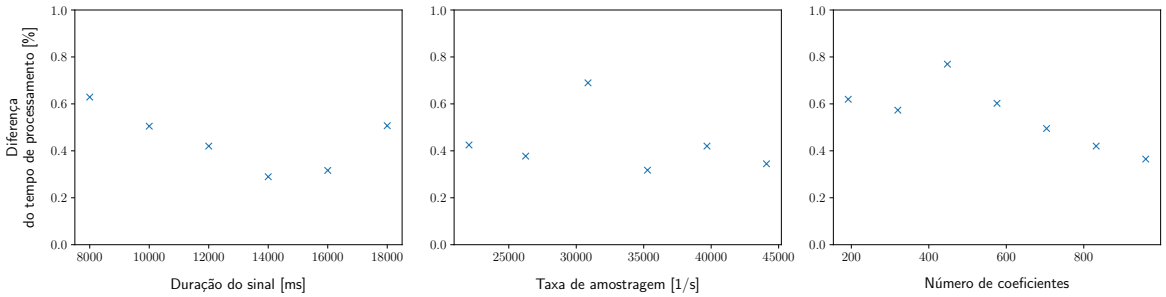

Figura 4.38: *Comparação do desempenho do elementos* `GstFlexfir0` *e* `GstFlexfir1`*: a) (esquerda) tempo de processamento com a taxa de amostragem fixa e número de coeficientes fixo em função da duração do sinal; b) (centro) tempo de processamento de um sinal de duração fixa e número de coeficientes fixo em função da taxa de amostragem; c) (direita) tempo de processamento de um sinal de duração fixa e taxa de amostragem fixa em função do número de coeficientes.*

Na Figura 4.38 pode ser vista a diferença de desempenho entre os elementos `GstFlexfir0` e `GstFlexfir1`. Observa-se que o desempenho do elemento `GstFlexfir0` é melhor do que aquele do elemento `GstFlexfir1`, o que se explica pela indireção no acesso aos índices esparsos dos coeficientes do filtro (linha 9 do Algoritmo 4.7 em comparação com a linha 8 do Algoritmo 3.2).

Adicionalmente, deve-se considerar que a função `get_history_at` acessa uma área de memória contígua quando chamada consecutivamente no laço com o Algoritmo 3.2, em contraste com o Algoritmo 4.7, no qual a área de memória acessada não é necessariamente contígua no caso geral. É fato que o acesso a uma área de memória contígua é acelerado pelos mecanismos de *caching* da CPU[46]. Apesar disso, a diferença de desempenho entre os elementos `GstFlexfir0` e `GstFlexfir1` (e, respectivamente, `GstFlexfir2`) é menor que 1% para todos os casos mensurados durante o experimento.

Os valores das constantes c_0, c_1 e c_2 obtidos pelo ajuste da função de custos (Equação 4.29) podem ser vistos na Tabela 4.16.

Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	c_2 [ms por amostra e coeficiente]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s}, M = 448)$ [ms]
F32LE	13.98	13.60e-5	6.471e-06	6.2	148

Tabela 4.16: *Coeficientes da função de custos do elemento* `GstFlexfir2`*.*

4.4.8.2 Qualidade

Nesta seção discute-se os elementos `GstFlexFir` quanto à qualidade percebida, com o objetivo de selecionar o elemento mais apropriado para seu uso em uma aplicação de processamento de áudio, especificamente para a aplicação de teste `flexmix` apresentada na Seção 4.5.

Para essa avaliação utiliza-se o método PEAQ, como foi discutido nas Seções 2.3.2 e 4.3.2, que permite a comparação entre os elementos quanto à qualidade percebida. O método PEAQ é indicado para aferir a qualidade de elementos isolados (por exemplo, alto-falantes), independentemente do contexto de inserção destes elementos. Neste trabalho não se pressupõe em geral que um elemento com a melhor avaliação (local) seja necessariamente a melhor alternativa dentro de uma aplicação flexibilizada (global), porém parece razoável supor, no contexto de uma aplicação flexibilizada com o `flexmix`, que a versão do elemento `GstFlexFir` que recebe a melhor avaliação pelo método PEAQ,

[46] Os mecanismos de *caching* da CPU são otimizados para um acesso a uma memória contígua, aproveitando a maior frequência de acesso a endereços consecutivos em muitas aplicações.

quando testada isoladamente, seja uma escolha apropriada para uma ampla gama de aplicações do `flexmix`.

No experimento, utiliza-se a versão avançada do método PEAQ (`GstPEAQ`), pois, por um lado, essa versão reproduz a norma ITU-R BS. 1387 mais fielmente do que a versão básica e, por outro lado, o experimento não possui restrições quanto ao tempo de execução da avaliação, visto que este não impacta no desempenho da aplicação final.

A coleção de sinais utilizada no experimento é uma coleção de sinais de teste publicada pela ITU-R para testar e avaliar implementações do método PEAQ[47] e contém exemplos de instrumentos musicais (violino, triângulo, instrumentos de percussão) e de voz (textos falados). Nestes sinais, várias características psicoacústicas (relevantes para a norma ITU-R BS. 1387) são codificadas, fato que indica a utilização destes exemplos para os testes dos elementos `GstFlexFir`, permitindo a avaliação da degradação destas características pelo método PEAQ.

No experimento, as três versões do elemento `GstFlexfir` são testadas da seguinte forma: o pipeline de teste é executado para cada taxa de amostragem, para cada número de coeficientes e para cada um dos sinais de teste. As taxas de amostragem foram selecionadas no experimento com os elementos de reamostragem (`GstAudioResample`) da Seção 4.4.7. Os valores do número de coeficientes do filtro FIR estão no intervalo $(128, 1024)$; esses extremos foram selecionados pois filtros FIR muito curtos são computacionalmente muito eficientes e não justificariam uma estratégia (custosa) de flexibilização, ao passo que 1024 é o parâmetro de tamanho mínimo de bloco no `Gstreamer` (e filtros maiores são geralmente implementados no domínio da frequência). Como saída de referência usa-se o sinal produzido com o maior número de coeficientes do filtro (1024).

Para a apresentação dos resultados da qualidade percebida (ODG, veja a Seção 2.3) obtidos, relativos aos distintos sinais de teste processados com a mesma taxa de amostragem e o mesmo número de coeficientes, utiliza-se: (a) a média dos valores e (b) o pior valor. O pior valor pode ser utilizado como critério adicional de desempate, quando a diferença entre os valores médios de dois algoritmos é pequena, visto que o pior valor corresponde ao cenário de pior caso perceptual, o qual deseja-se limitar.

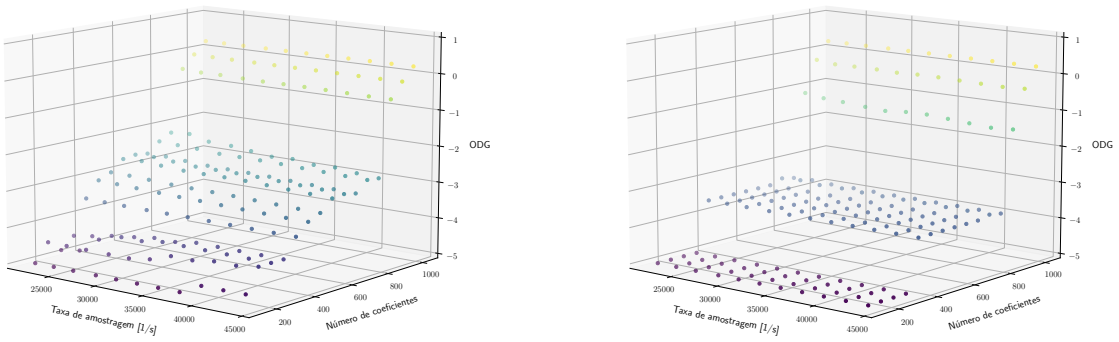

Figura 4.39: *Qualidade percebida do elemento `GstFlexfir0` em função do número de coeficientes e da taxa de amostragem: a) (esquerda) ODG médio; b) (direita) ODG no pior caso.*

[47] A ITU-R escolheu estes sons da coleção *EBU SQAM Disc*; www.itu.int/ITU-R.

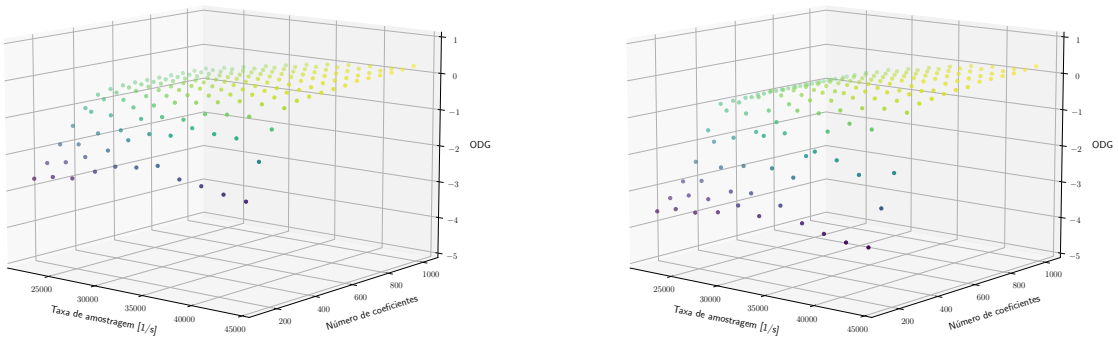

Figura 4.40: *Qualidade percebida do elemento* `GstFlexfir1` *em função do número de coeficientes e da taxa de amostragem: a) (esquerda) ODG médio; b) (direita) ODG no pior caso.*

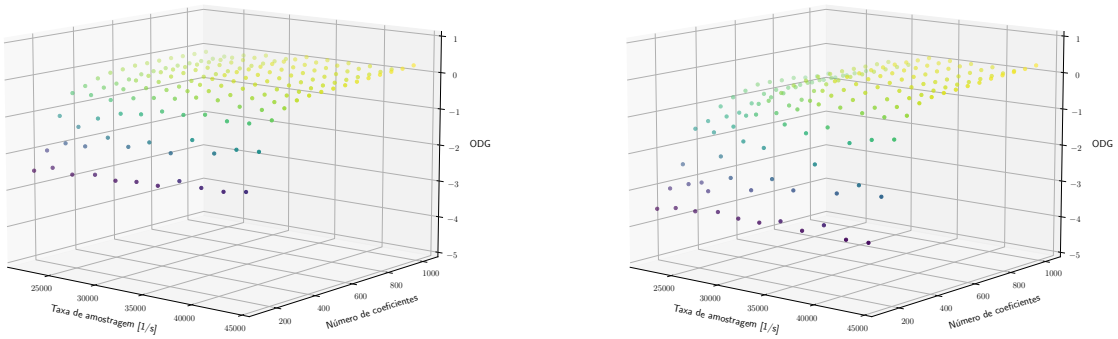

Figura 4.41: *Qualidade percebida do elemento* `GstFlexfir2` *em função do número de coeficientes e da taxa de amostragem: a) (esquerda) ODG médio; b) (direita) ODG no pior caso.*

Nas Figuras 4.39, 4.40 e 4.41 podem ser vistos os resultados do experimento quanto à média das qualidades percebidas em função do número de coeficientes e da taxa de amostragem para cada um dos elementos `GstFlexfir0`, `GstFlexfir1` e `GstFlexfir2`.

Observa-se que a avaliação do elemento `GstFlexfir0` (Figura 4.39) resulta em alta qualidade (valores de ODG médio entre 0 e -1) apenas com um alto número de coeficientes (maior que 900), e que o ODG médio cai rapidamente com menos coeficientes, indicando diferenças 'irritantes' (ODG igual a -3) ou 'muito irritantes' (ODG igual ou menor a -4). A Figura 4.39b (piores casos) indica que o elemento produz qualidades percebidas 'muito irritantes' para pelo menos um sinal de teste quanto o número de coeficientes é menor do que 900, o que sugere ser arriscado utilizar-se este elemento, na estratégia de flexibilização, quando se deseja obter melhorias computacionais

significativas.

Para o elemento `GstFlexfir1` (Figura 4.40), observa-se que o ODG médio estimado é melhor para uma faixa maior de valores do parâmetro *número de coeficientes* em comparação com o elemento `GstFlexfir0`, indicando que a seleção dos coeficientes do algoritmo `GstFlexfir1` (coeficientes com os maiores valores absolutos) é mais adequada do que aquela do algoritmo `GstFlexfir0` (primeiros coeficientes). Ainda assim, para filtros com muitos coeficientes (> 900), o `GstFlexfir1` possui um desempenho pior do que o `GstFlexfir0` para quase todas as taxas de amostragem (< 40 kHz). Vale lembrar que ambos os elementos adaptam os coeficientes a outras taxas de amostragem por reamostragem do filtro; adicionalmente, o elemento `GstFlexfir1` adapta também as amplitudes dos coeficientes (veja a Seção 3.5.1). Os resultados indicam que a abordagem do elemento `GstFlexfir1` não parece ser eficaz para taxas de amostragem menores.

Os resultados obtidos com o elemento `GstFlexfir2` (Figura 4.41) mostram que o ODG tem alta qualidade (entre 0 e -1) para uma ampla faixa de valores tanto do parâmetro *número de coeficientes* quanto do parâmetro *taxa de amostragem*, tendo uma queda acentuada quando o numero de coeficientes é menor que 400, de forma similar ao caso do elemento `GstFlexfir1`. Para taxas de amostragem mais baixas, as qualidades percebidas são melhores para o elemento `GstFlexfir2` do que para o elemento `GstFlexfir1`.

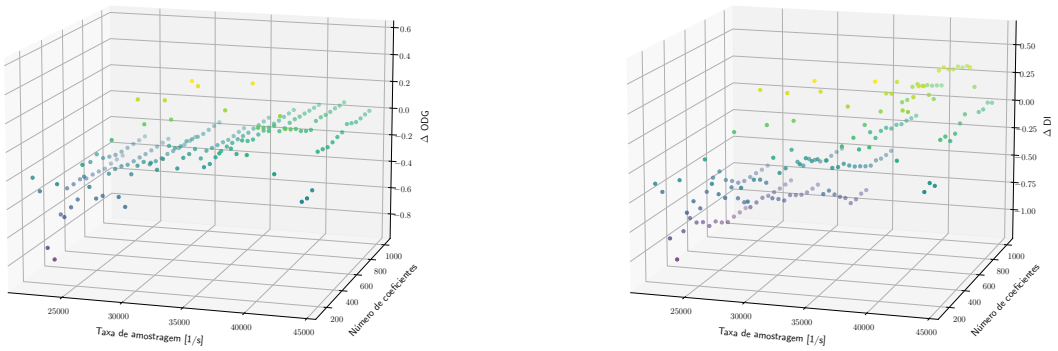

Figura 4.42: *Diferença de qualidade percebida entre os elementos `GstFlexfir1` e `GstFlexfir2`: a) (esquerda) Diferença das qualidades percebidas (ODG); b) (direita) Diferença dos índices de distorção (distortion index: DI).*

A diferença entre as qualidades percebidas médias dos elementos `GstFlexfir1` e `GstFlexfir2` pode ser vista na Figura 4.42. A diferença entre os índices de distorção (DI) permite uma visualização amplificada da diferença de qualidade percebida (ODG), uma vez que ODG é definido em função de DI através da equação $\text{ODG} = -3.98 + 4.2\ \text{sig}(\text{DI})$, onde $\text{sig}(x) = 1/(1 + e^{-x})$, sendo que a função sigmoide aproxima (nos valores de ODG) as diferenças entre valores de DI[48].

Nota-se que o elemento `GstFlexfir1` recebe avaliações piores do que o elemento `GstFlexfir2` para taxas de amostragem <39 kHz, porém recebe avaliações iguais ou melhores para taxas de amostragem ≥39 kHz. Além disso, pode ser observado que o elemento `GstFlexfir1` recebe avaliações melhores para algumas taxas de amostragem quando o número de coeficientes é pequeno, porém as vantagens desse elemento são pequenas ($\Delta\text{ODG} < 0.1$) à luz da escala do ODG (entre -4 e 0).

[48] Veja, por exemplo, (Kabal, 2003, pág. 54)

A partir dos resultados do experimento, o elemento `GstFlexfir2` é selecionado para ser utilizado na aplicação `flexmix`, visto que este recebe as melhores avaliações (ODG) para o maior número de configurações (taxa de amostragem e número de coeficientes). Deve-se notar ainda que, com este elemento, a qualidade percebida decai rapidamente no experimento quando o número de coeficientes é menor que aproximadamente 400, indicando uma faixa de valores a ser evitada no emprego deste elemento na aplicação `flexmix`.

Mesmo que o experimento da qualidade percebida indique degradações pequenas para grandes faixas dos parâmetros de processamento, deve-se ter em consideração que filtros com taxa de amostragem e número de coeficientes mais próximos do filtro original produzem em geral menor distorção da resposta em frequência, razão pela qual sugere-se que o elemento `GstFlexfir2` seja sempre executado com a maior taxa de amostragem e maior número de coeficientes possíveis, respeitando a condição temporal de execução em tempo real.

4.4.8.3 Flexibilização

O elemento `GstFlexfir2` é desenvolvido com o objetivo da flexibilização do parâmetro *número de coeficientes*. A Equação 4.29) mostra a influência deste parâmetro nos custos computacionais. Além disso, esse elemento pode ser empregado em uma cadeia maior contendo outros elementos, na qual o desempenho é controlado pela taxa de amostragem, como descrito na Seção 4.4.7 sobre os elementos de reamostragem. Ambas as opções de flexibilização são utilizadas na aplicação de teste `flexmix`.

4.5 Estudo de caso: a aplicação `flexmix`

Nesta seção discute-se a implementação de uma aplicação de processamento de áudio flexibilizada seguindo a metodologia proposta nesta pesquisa (veja o Capítulo 3). Considera-se ainda as estratégias do processamento em tempo real e da computação imprecisa discutidas na Seção 2.4 e as abordagens de flexibilização discutidas na Seção 2.5.

O objetivo desta seção é provar a viabilidade do conceito da flexibilização e da metodologia proposta. A respeito disso, deve-se:

- construir uma aplicação de processamento de áudio parametrizada e determinar sua função de custos;
- estabelecer um ambiente experimental para a avaliação da qualidade percebida para configurações específicas (parâmetros de flexibilização, sons de entrada e configurações *musicais*[49]);
- estabelecer um método de otimização que obtém parametrizações ótimas (com as quais a aplicação produz a melhor qualidade possível) considerando a viabilidade dos custos computacionais no processamento em tempo real;
- implementar um método de gerenciamento dinâmico que consegue controlar os custos computacionais da aplicação, reagindo à variação de carga computacional do sistema usando as parametrizações ótimas obtidas no passo anterior.

4.5.1 A aplicação `flexmix`

A aplicação `flexmix` implementa um mesa de mixagem simples, que combina um certo número de canais de entrada em um canal *master*. Esse é um tipo de aplicação de processamento de áudio fundamental, que aparece frequentemente ao lado de outras aplicações de estúdio digital, tais como efeitos de áudio e sintetizadores.

[49] Entende-se aqui como configurações do ponto de vista musical, ou seja, aquelas que um usuário da aplicação pode configurar e que são independentes dos parâmetros de flexibilização, como por exemplo os valores de amplificação e níveis de compressão ou o volume dos canais no elemento de mixagem.

Para a implementação da aplicação `flexmix` utiliza-se os elementos investigados anteriormente neste capítulo. Cada canal contém os seguintes elementos:

- entrada de som,
- pré-amplificação,
- equalizador com três faixas de frequência e
- filtragem (FIR).

Dentro desta configuração, o filtro FIR tem a função de imprimir efeitos de propriedades espaciais, como discutido nas Seções 2.5.5 e 3.5.1. Adicionalmente, são introduzidos dois elementos de reamostragem em cada canal, encapsulando os outros elementos com o objetivo de permitir a flexibilização da taxa de amostragem, como discutido nas Seções 3.1.2 e 4.4.7.

O canal *master* contém os seguintes elementos:

- equalizador com 10 faixas de frequência,
- compressão dinâmica,
- saída de som.

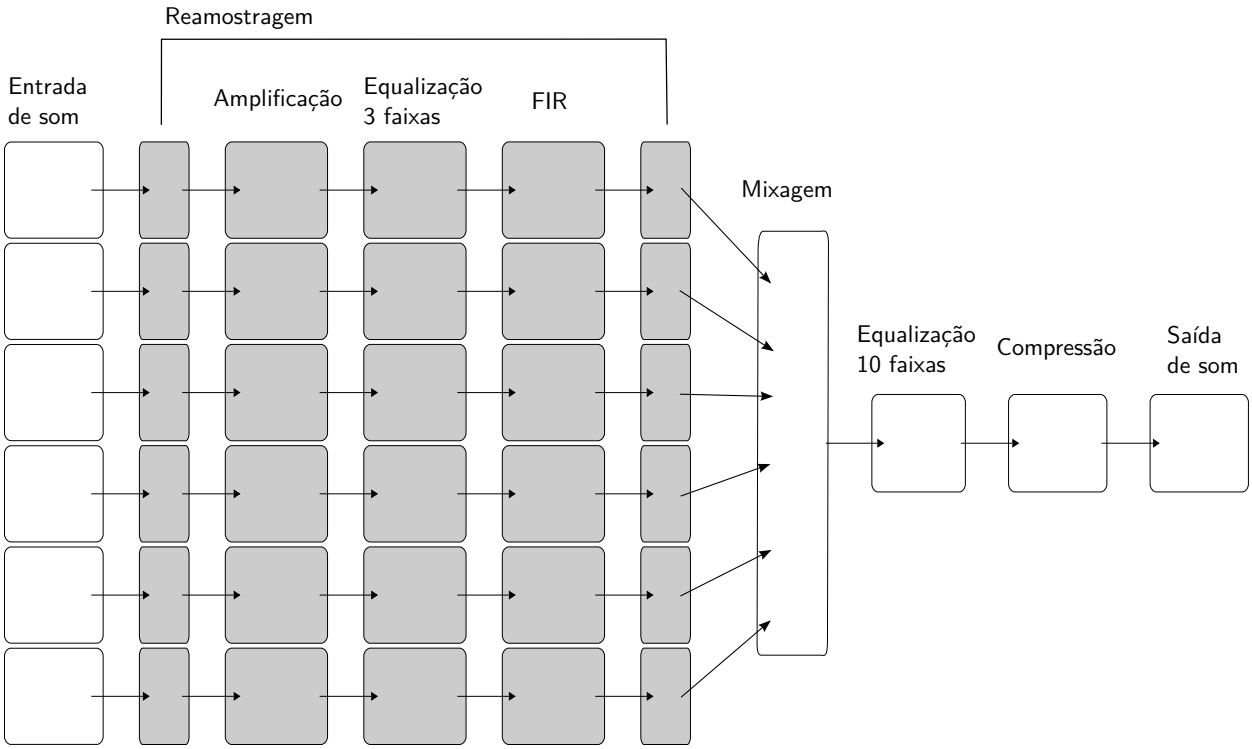

Figura 4.43: *Estrutura principal da aplicação* `flexmix`.

Na Figura 4.43 pode ser vista a estrutura principal da aplicação `flexmix`; os elementos de processamento em cinza são aqueles afetados pela flexibilização.

Além do pipeline ilustrado nesta figura, que é composto por todos os elementos de processamento, a aplicação possui um *gerenciador de flexibilização*, que realiza o *trade-off* entre desempenho e qualidade, considerando o estado do sistema computacional ou, mais especificamente, a carga do sistema. Como discutido na Seção 3.2.1, e diferentemente do que ocorre em outras aplicações em tempo real semelhantes, o `flexmix` não requer mudanças na camada do sistema operacional nem

nos componentes centrais do arcabouço `Gstreamer` (*core libraries*). O método de computação imprecisa utilizado equivale a uma adaptação do método de versões múltiplas, onde cada elemento é configurado (parametrizado) antes da execução, para viabilizar os custos computacionais em relação ao processamento em tempo real (veja a Seção 2.4).

Na aplicação `flexmix` são utilizados os seguintes parâmetros de flexibilização:

- taxa de amostragem em cada canal,
- qualidade da reamostragem (elementos de reamostragem) e
- número de coeficientes dos filtros FIR.

Os parâmetros *taxa de amostragem* e *número de coeficientes dos filtros FIR* são relacionados ao método de perfuração de código (veja a Seção 2.5.2). O grau de perfuração pode ser calculado como o quociente da taxa de amostragem utilizada pela taxa de amostragem máxima.

O conceito do nível de detalhe (veja a Seção 2.5.3) é implementado através da avaliação de qualidade percebida, a fim de determinar com qual precisão cada canal deve ser processado em função de sua relevância na mixagem final, como explicado abaixo na Seção 4.5.4.

Os experimentos com os elementos de processamento mostram que *tipo de dados* não é um parâmetro de flexibilização útil, uma vez que o tipo de dados não causa grande variação dos custos computacionais ou até mesmo resulta em custos maiores para tipos de dados menos precisos.

4.5.2 Os custos computacionais

A análise dos custos computacionais da aplicação `flexmix` é dividida em dois passos: primeiro investiga-se os custos de um canal da aplicação e os custos do canal *master*, e no segundo passo estima-se os custos da aplicação completa.

Para a identificação dos vários elementos e respectivos coeficientes nas funções de custo, utiliza-se caracteres gregos como pode ser visto na Tabela 4.17.

Elemento	Identificador	Função de custos
Leitura de arquivo (finput)	α	$z^{(\alpha)} = c_0^{(\alpha)} + c_1^{(\alpha)}\, tr$
Amplificação (amp)	β	$z^{(\beta)} = c_0^{(\beta)} + c_1^{(\beta)}\, tr$
Equalizador com 3 faixas (eq3)	γ	$z^{(\gamma)} = c_0^{(\gamma)} + c_1^{(\gamma)}\, tr$
Flexfir	δ	$z^{(\delta)} = c_0^{(\delta)} + c_1^{(\delta)}\, tr + c_2^{(\delta)} \frac{M_{r_0}}{r_0}\, tr^2$
Reamostragem (2res)	ϵ	$z^{(\epsilon)} = c^{(\epsilon)}_{0,q,r_{\text{in}},r_{\text{out}}} + c^{(\epsilon)}_{1,q,r_{\text{in}},r_{\text{out}}} tr$
Mixagem	ζ	$z^{(\zeta)} = c_0^{(\zeta)} + c_1^{(\zeta)} n_c + c_2^{(\zeta)} n_c\, tr$
Equalizador com 10 faixas (eq10)	η	$z^{(\eta)} = c_0^{(\eta)} + c_1^{(\eta)}\, tr$
Compressão Dinâmica (dyn)	θ	$z^{(\theta)} = c_0^{(\theta)} + c_1^{(\theta)}\, tr$
Saída de som (sndout)	ι	$z^{(\iota)} = c_0^{(\iota)} + c_1^{(\iota)}\, tr$

Tabela 4.17: *Identificadores das funções de custos dos elementos investigados.*

4.5.2.1 Desempenho de um canal da aplicação `flexmix`

A função de custos de um canal da aplicação `flexmix` pode ser calculada como a soma dos custos dos elementos contidos na cadeia que representa o canal:

$$\begin{aligned} z =& z^{(\alpha)} + z^{(\beta)} + z^{(\gamma)} + z^{(\delta)} + z^{(\epsilon)}_{q,r_{\text{in}},r_{\text{out}}} \\ =& \left(c_0^{(\alpha)} + c_0^{(\beta)} + c_0^{(\gamma)} + c_0^{(\delta)} + c^{(\epsilon)}_{0,q,r_{\text{in}},r_{\text{out}}}\right) \\ &+ \left(c_1^{(\alpha)} + c^{(\epsilon)}_{1,q,r_{\text{in}},r_{\text{out}}}\right) tr_{\text{out}} \\ &+ \left(c_1^{(\beta)} + c_1^{(\gamma)} + c_1^{(\delta)}\right) tr_{\text{in}} + c_2^{(\delta)} \frac{M_{r_0}}{r_0} tr_{\text{in}}^2. \end{aligned} \tag{4.30}$$

Substituindo

- $k = r_{\text{in}}/r_{\text{out}}$
- $c_{0,q,r_{\text{in}},r_{\text{out}}} = c_0^{(\alpha)} + c_0^{(\beta)} + c_0^{(\gamma)} + c_0^{(\delta)} + c^{(\epsilon)}_{0,q,r_{\text{in}},r_{\text{out}}}$
- $c_{1,q,r_{\text{in}},r_{\text{out}}} = c_1^{(\alpha)} + c^{(\epsilon)}_{1,q,r_{\text{in}},r_{\text{out}}}$
- $c_{1,\beta\gamma\delta} = c_1^{(\beta)} + c_1^{(\gamma)} + c_1^{(\delta)}$
- $c_2 = c_2^{(\delta)}$

pode-se expressar os custos como:

$$z(t, r_{\text{out}}, k) = c_{0,q,r_{\text{in}},r_{\text{out}}} + c_{1,q,r_{\text{in}},r_{\text{out}}} tr_{\text{out}} + c_{1,\beta\gamma\delta} k tr_{\text{out}} + c_2 k^2 \frac{M_{r_0}}{r_0} tr_{\text{out}}^2. \tag{4.31}$$

Para a aplicação `flexmix` define-se $r_0 = r_{\text{out}}$ e denota-se essa taxa de amostragem simplesmente por r, considerando-se que o número de coeficientes M do filtro FIR sempre está relacionado a essa taxa de amostragem ($M = M_{r_0}$):

$$z(t, r, k, q, M) = c_{0,q,r_{\text{in}},r_{\text{out}}} + \left(c_{1,q,r_{\text{in}},r_{\text{out}}} + c_{1,\beta\gamma\delta} k + c_2 k^2 M\right) tr. \tag{4.32}$$

Os parâmetros de flexibilização nesta nova formulação da função de custos são (a) o número de coeficientes do filtro FIR (M), (b) a razão entre as taxas de amostragem interna e externa ($k = r_{\text{in}}/r_{\text{out}}$) e (c) a qualidade de amostragem (que corresponde a distintos valores de $c_{0,q,r_{\text{in}},r_{\text{out}}}$ e $c_{1,q,r_{\text{in}},r_{\text{out}}}$ para cada qualidade q). a razão entre as taxas de amostragem interna e externa ($k = r_{\text{in}}/r_{\text{out}}$) e a qualidade de amostragem (que corresponde a distintos valores de $c_{0,q,r_{\text{in}},r_{\text{out}}}$ e $c_{1,q,r_{\text{in}},r_{\text{out}}}$ para cada qualidade q).

4.5.2.2 Experimento

Mesmo que a função de custos computacionais possa ser calculada com os coeficientes obtidos nos experimentos anteriores, executa-se um novo experimento para reconfirmar esse resultado e também para estimar o erro relativo da estimação dos custos.

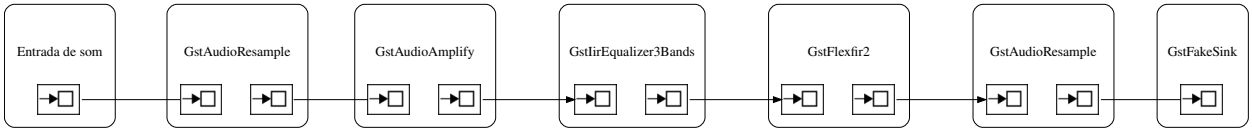

Figura 4.44: *Pipeline para a mensuração do desempenho de um canal da aplicação* `flexmix`.

O pipeline investigado pode ser visto na Figura 4.44; o experimento segue o procedimento dos experimentos anteriores com os elementos isolados. O espaço das configurações a ser percorrido

é estabelecido pelas faixas úteis dos parâmetros de flexibilização e da duração do som. A taxa de amostragem fora dos elementos de reamostragem (r_{out}) deve ser fixa, considerando-se que os elementos do canal *master* não são incluídos na flexibilização. No experimento escolhe-se uma taxa padrão ($r_{\text{out}} = 44100\text{Hz}$) que é compatível com o hardware, evitando-se assim uma reamostragem (implícita) no elemento de saída de som.

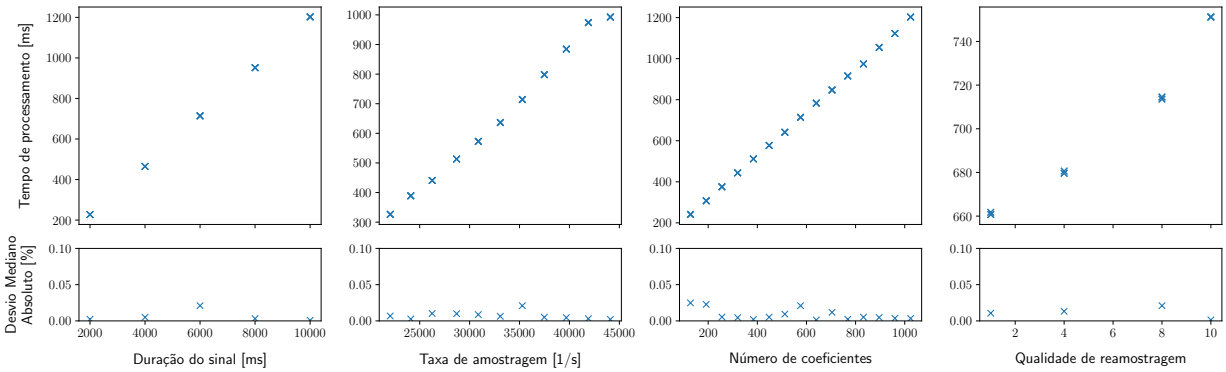

Figura 4.45: *Desempenho de um canal da aplicação* `flexmix`*: a) (esquerda) tempo de processamento em função da duração do sinal com os outros parâmetros (r_{in}, M_{r_0} e a qualidade de reamostragem) fixos; b) (centro esquerda) tempo de processamento em função da taxa de amostragem entre os elementos de reamostragem com os outros parâmetros fixos; c) (centro direita) tempo de processamento em função do número de coeficientes do filtro FIR com os outros parâmetros fixos; d) (direita) tempo de processamento em função da qualidade de reamostragem com os outros parâmetros fixos.*

Os resultados do experimento (Figura 4.45) reconfirmam a forma da função de custos estabelecida (Equação 4.32).

Qualidade de reamostragem	$c_{0,q,r_{\text{in}},r_{\text{out}}}$ [ms]	$c_{1,q,r_{\text{in}},r_{\text{out}}}$ [ms por amostra]	$c_{1,\beta\gamma\delta}$ [ms por amostra]	c_2 [ms por amostra e por coeficiente]	Erro mediano [%]	$z(r_{\text{in}} = 33075\text{Hz}, t = 1\text{s}, M = 448)$ [ms]
1.0	0.7537	3.474e-4	0.0	5.104e-06	8.2	72.8
3.0	0.4716	3.858e-4	0.0	5.116e-06	7.3	74.3
6.0	0.6619	4.317e-4	0.0	5.137e-06	6.8	76.8
10.0	2.573	6.571e-4	0.0	5.092e-06	7.6	88.1

Tabela 4.18: *Coeficientes da função de custos de um canal da aplicação* `flexmix`*.*

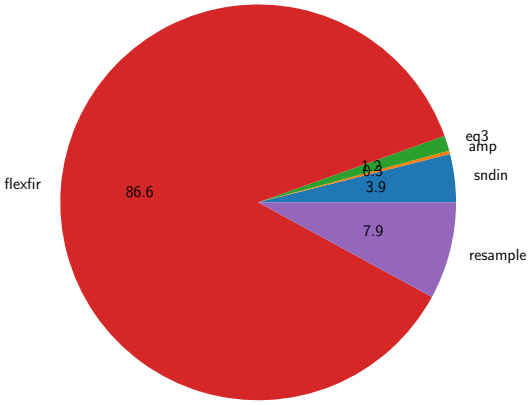

Figura 4.46: *Custos relativos dos elementos de um canal da aplicação* `flexmix` *para $t = 1s$, $r_{in} = 33075Hz$, $q = 4$ e $M_{r_0} = 512$.*

Na Tabela 4.18 podem ser vistos os coeficientes da função de custos computacionais estima-

dos experimentalmente (para três valores ilustrativos do parâmetro de qualidade). A Figura 4.46 apresenta os custos dos diversos elementos relativamente aos custos totais do pipeline do canal.

Observa-se que o coeficiente $c_{1,\beta\gamma\delta}$ é nulo em todos os casos, o que provavelmente decorre de dois fatores. O primeiro fator é o baixo custo dos elementos associados a $c_1^{(\beta)}$ e $c_1^{(\gamma)}$ (amplificação, equalização e filtro, veja a Figura 4.46), e ao fato de que a maior parte dos custos computacionais do elemento `flexfir` (veja a Equação 4.29) estão concentrados no termo $c_2^{(\delta)} \frac{M r_0}{r_0} tr^2$, sempre que o filtro tiver um número alto de coeficientes (por exemplo, no contexto de espacialização é usual $M > 200$), o que diminui a importância relativa do termo $c_1^{(\delta)}$. O segundo fator relaciona-se a características do modelo de ajuste por programação linear (veja a Seção 4.4.1), onde as soluções ótimas obtidas são sempre vértices, que são caracterizados por uma coleção de valores nulos (associados às variáveis não-básicas).

A Figura 4.46 revela também que os custos computacionais dos elementos de reamostragem são pequenos em comparação com os custos computacionais dos demais elementos, especialmente do elemento `flexfir`. Este fato mostra a viabilidade da abordagem de flexibilização da taxa de amostragem (r_{in}) dentro da cadeia de elementos.

4.5.2.3 Desempenho do canal *master* da aplicação `flexmix`

A função de custos do canal *master* da aplicação `flexmix` pode ser calculada como a soma das funções de custos dos seguintes elementos: mixagem, equalizador, compressão dinâmica e saída de som; tais funções de custos podem ser vistos na Tabela 4.17. Decorrentemente, obtém-se para a função de custos do canal *master*:

$$\begin{aligned} z(t, r, n_c) =& z^{(\zeta)} + z^{(\eta)} + z^{(\theta)} + z^{(\iota)} \\ =& z_0^{(\zeta)} + z_0^{(\eta)} + z_0^{(\theta)} + z_0^{(\iota)} + z_1^{(\zeta)} n_c + \left(z_2^{(\zeta)} n_c + z_1^{(\eta)} + z_1^{(\theta)} + z_1^{(\iota)} \right) tr \qquad (4.33) \\ & \qquad (4.34) \end{aligned}$$

Embora o número de canais n_c pudesse ser tratado como um parâmetro de flexibilização em outras aplicações, tais como simulação de salas (Seção 3.5.2) e espacialização (Seção 3.5.3), na aplicação `flexmix` este número é um parâmetro fixo, a exemplo do que ocorre em mesas de mixagens em hardware.

Substituindo as expressões abaixo

- $c_{0,n_c}^{(\lambda)} = z_0^{(\zeta)} + z_0^{(\eta)} + z_0^{(\theta)} + z_0^{(\iota)} + z_1^{(\zeta)} n_c$
- $c_{1,n_c}^{(\lambda)} = z_2^{(\zeta)} n_c + z_1^{(\eta)} + z_1^{(\theta)} + z_1^{(\iota)}$

na última equação, pode-se expressar os custos do canal *master* como:

$$z_{n_c}^{(\lambda)}(t, r) = c_{0,n_c}^{(\lambda)} + c_{1,n_c}^{(\lambda)} tr. \qquad (4.35)$$

Destaca-se que o canal *master* não contém elementos flexibilizados e possui uma taxa de amostragem fixa, razão pela qual os custos acima não dependem de parâmetros de qualidade ou de taxas de amostragem variáveis como os demais canais.

4.5.2.4 Experimento

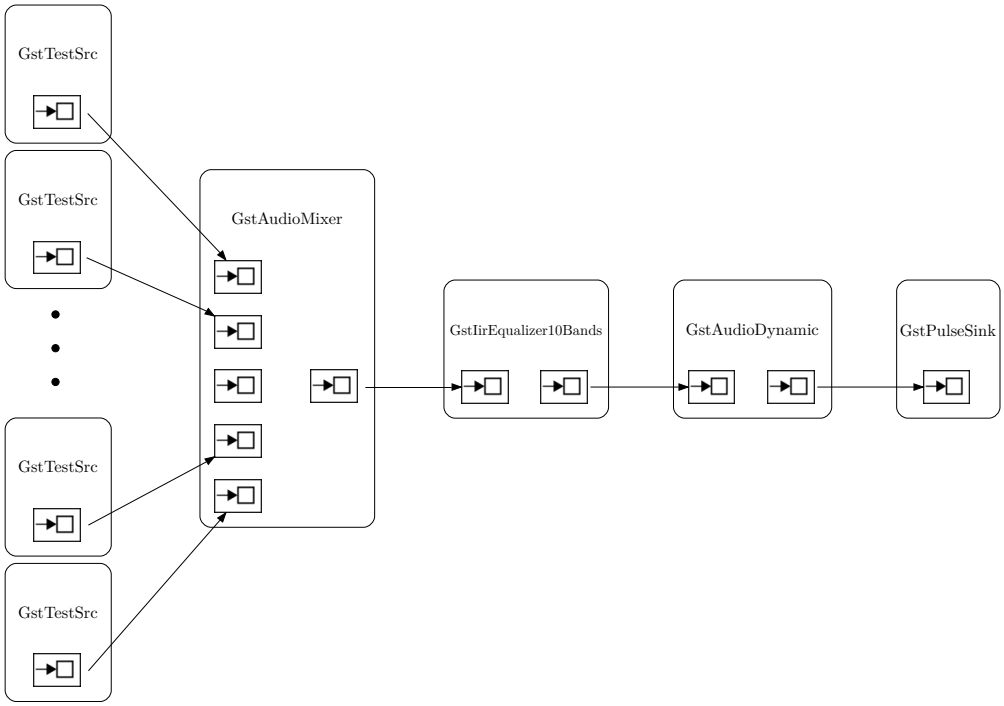

Figura 4.47: *Pipeline para a mensuração do desempenho do canal* master *da aplicação* `flexmix`.

O pipeline usado para a mensuração do desempenho do canal *master* pode ser visto na Figura 4.47. Neste experimento simples utiliza-se os elementos `GstTestSrc` para produzir os sinais de entrada de duração t com uma taxa de amostragem fixa ($r = 44100$ Hz) e um tipo de dados fixo ('S32LE').

Tipo de dados	c_0 [ms]	c_1 [ms por amostra]	Erro mediano [%]	$z(r = 44100\text{Hz}, t = 1\text{s})$ [ms]
F32LE	3.177	2.440e-4	0.28	13.94

Tabela 4.19: *Coeficientes da função de custos do canal* master *da aplicação* `flexmix`.

4.5.3 Desempenho da aplicação `flexmix`

A função dos custos computacionais da aplicação `flexmix` é composto pela soma dos custos de cada canal (veja a Equação 4.32) e dos custos dos elementos que compõem o canal *master*; esses elementos são o elemento de mixagem (veja a Equação 4.14), o elemento de equalização (veja a Equação 4.6), o elemento de compressão dinâmica (veja a Equação 4.12) e o elemento de saída de som (veja a Equação 4.18). Consequentemente obtém-se a função de custos computacionais

$$z_{n_c}(t, r) = \sum_{i=1}^{n_c} z_i^{(\epsilon)}(t, r, k_i, q_i, M_i) + z_{n_c}^{(\lambda)}(t, r), \tag{4.36}$$

onde $z_i^{(\epsilon)}$ corresponde à função dos custos de um canal. Os índices r_{in} e r_{out} são substituídos nessa versão pela razão k_i, que pode ser diferente para cada canal i, e é escrito como argumento da função de custos, assim como o parâmetro q_i de qualidade do canal i.

Os parâmetros de flexibilização considerados são os parâmetros de cada canal (k_i, q_i e M_i), sendo que os elementos do canal *master* não são flexibilizados.

4.5.3.1 Experimento

Nome	Intervalo da taxa de amostragem	Tipos de dados permitidos	Outros parâmetros
`flexmix`	[7350, 44100]	F32LE	número de coeficientes (`flexfir2`), taxa de amostragem e qualidade da reamostragem (`GstAudioResample`) de cada canal

Tabela 4.20: *Parâmetros da aplicação* `flexmix`.

O experimento para a estimação do desempenho da aplicação `flexmix` deve mensurar não somente os custos computacionais (tempo de *processamento*), mas também o tempo de *execução* (tempo real, *wall-clock time*) para estimar o fator de aceleração (veja a Seção 3.1.2).

Visto que as cadeias de elementos que formam os canais são estruturalmente iguais, não há necessidade de varrer o espaço de parâmetros de cada canal independentemente, como nos experimentos anteriores; ao invés disso, cada configuração na varredura do espaço de parâmetros é aplicada homogeneamente em todos os canais.

Em função disso, pode-se simplificar a função de custos anterior, obtendo-se:

$$z_{n_c}(t, r) = n_c z(t, r, k, q, M) + z_{n_c}^{(\lambda)}(t, r). \tag{4.37}$$

Na aplicação `flexmix` a ferramenta `gst-top` mostra-se pouco confiável devido ao grande número de plugins, levando a distorções nos valores mensurados. Por outro lado, a ferramenta `perf` permite apenas a mensuração de tempos de processamento da aplicação inteira. Para contornar estas dificuldades, neste experimento os tempos de processamento são mensurados diretamente na aplicação, o que permite medir o custo do processamento de sinais nos elementos da aplicação.

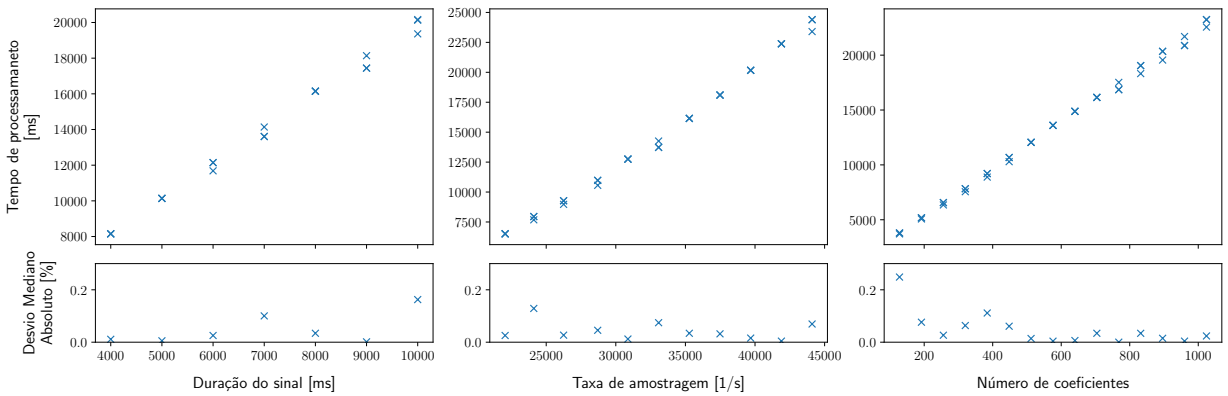

Figura 4.48: *Desempenho da aplicação* `flexmix`

Qualidade de reamostragem	$c_{0,q,r_{\text{in}},r_{\text{out}}}$ [ms]	$c_{1,q,r_{\text{in}},r_{\text{out}}}$ [ms por amostra]	$c_{1,\beta\gamma\delta}$ [ms por amostra]	c_2 [ms por amostra e por coeficiente]	Erro mediano [%]	$z(r_{\text{in}} = 33075\text{Hz}, t = 1\text{s}, M = 448)$ [ms]
1	11.71	1.084e-4	5.061e-05	6.912e-06	2.1	1344
4	11.55	1.761e-4	5.196e-05	6.932e-06	2.6	1387
8	10.98	3.715e-4	0.0	6.893e-06	4.0	1470
10	14.40	5.274e-4	0.0	6.920e-06	4.3	1618

Tabela 4.21: *Coeficientes da função de custos da aplicação* `flexmix` *simplificada.*

Na figura 4.48 pode ser visto o resultado do experimento com a aplicação `flexmix`. Os coeficientes da função de custos (Equação 4.37) obtidos experimentalmente através do ajuste do modelo teórico aos tempos de execução podem ser vistos na Tabela 4.21.

A fim de verificar a validade dos coeficientes da Tabela 4.21, foi realizado um experimento usando 100 configurações aleatórias dos parâmetros de cada canal. Considerando os tempos teóricos previstos com aqueles coeficientes, os erros de estimação obtidos foram de 6.6% (no máximo) e 4% (mediana dos erros), o que evidencia a generalidade dos resultados obtidos com a aplicação `flexmix` simplificada em relação a configurações mais gerais.

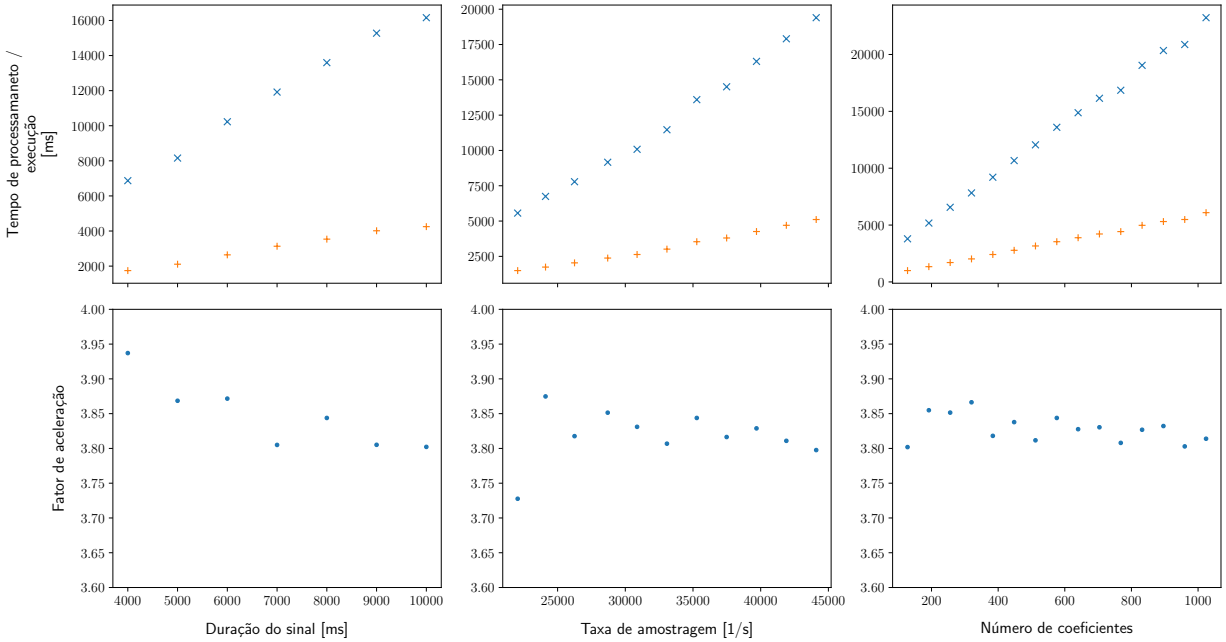

Figura 4.49: *Desempenho da aplicação `flexmix`: fator de aceleração.*

A figura 4.49 apresenta em azul os tempos de *processamento* (*CPU time*, tempo total da aplicação em uma única CPU) e em laranja os tempos de *execução* (*wall-clock time*, considerando múltiplos núcleos), bem como o fator de aceleração. Considerando-se que se trata de uma *workstation* com 4 núcleos, o fator de aceleração é bem alto, indicando que a aplicação é fortemente paralelizável, uma vez que os canais são processados independentemente. Observa-se que o fator de aceleração tem uma variação pequena (inferior a 4%) em relação aos parâmetros de configuração.

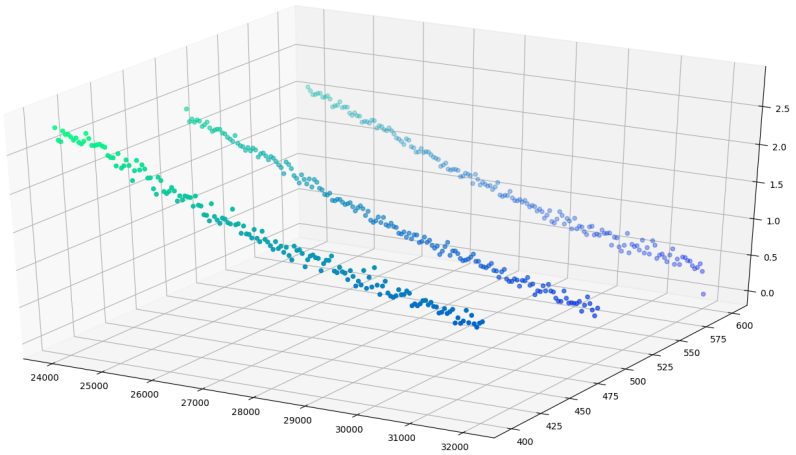

Figura 4.50: *WSD em função da taxa de amostragem e do número de coeficientes da aplicação* `flexmix`.

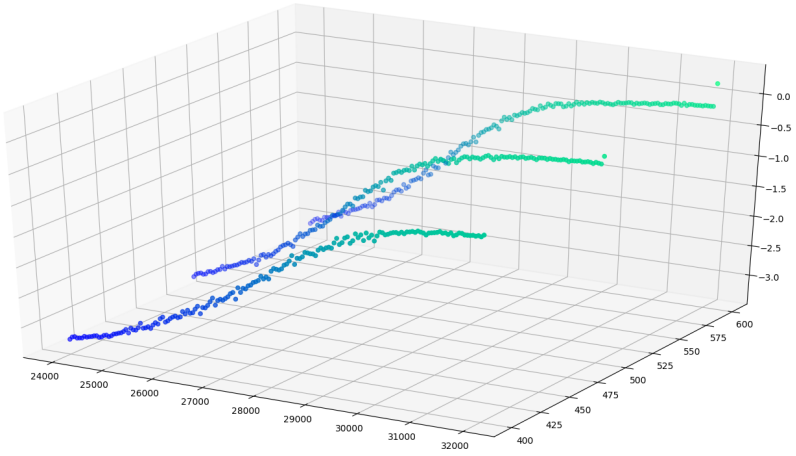

Figura 4.51: *ODG em função da taxa de amostragem e do número de coeficientes da aplicação* `flexmix`.

As Figuras 4.50 e 4.51 mostram a *Weighted Spectral Distortion* (Distorção espectral ponderada) (WSD) e o ODG da aplicação `flexmix` em função do número de coeficientes e da taxa de amostragem para três sinais de teste. Nesse experimento, todos canais são configurados com os mesmos valores do número de coeficientes e da taxa de amostragem. O experimento é concebido para produzir valores de referência das métricas de distorção e qualidade para os resultados da afinação automática (veja a próxima seção).

4.5.4 Afinação automática da aplicação `flexmix` - *autotuning*

O objetivo da afinação automática da aplicação `flexmix` é produzir um mapeamento da qualidade percebida e dos custos computacionais em função dos parâmetros de flexibilização (veja a

Seção 3.3.1). Para cada carga computacional considerada, procura-se pela afinação automática a configuração de parâmetros de flexibilização que resulta na melhor qualidade percebida possível. A carga computacional expressa-se como o quociente dos custos computacionais em relação aos custos máximos admissíveis. A duração do sinal produzido define o tempo máximo de execução da aplicação em tempo real (*wall-clock*); esse tempo máximo de execução traduz-se em tempo de processamento (*cpu time*) através do produto do tempo real t pelo fator de aceleração f_{accel} através da expressão $z_{\max} = f_{\text{accel}} t$.

No experimento realizado, considera-se um número fixo de cargas computacionais (20%, 30%, ... 100%) para as quais se obtém configurações de parâmetros ótimos (aqueles que resultam na melhor qualidade percebida) através da aplicação `OpenTuner`. O método de busca utilizado é a meta-técnica `AUCBanditMetaTechniqueA`[50], que combina quatro técnicas de otimização[51] em uma estratégia de *one-armed bandit*, sendo a mesma utilizada por Ansel e co-autores (Ansel *et al.*, 2013).

A função objetivo que a aplicação `OpenTuner` tenta minimizar deve combinar a qualidade percebida e o desempenho da aplicação `flexmix` em um único valor. A Função 4.38 abaixo é uma heurística resultante de testes preliminares, e combina o ODG (Q_{ODG}) e a diferença entre os custos máximos admissíveis e os custos associados a cada configuração dos parâmetros de flexibilização (Δt_{CPU} em milissegundos) em um único valor de penalização. Essa penalização cresce quando a qualidade piora e/ou quando os custos computacionais extrapolam os custos admissíveis. Para evitar que os custos computacionais excedam os custos máximos, emprega-se um fator de ponderação alto (1000.0) no termo de penalização dos custos.

$$p(Q_{\text{ODG}}, \Delta t_{\text{CPU}}) = -Q_{\text{ODG}} + \begin{cases} \Delta t_{\text{CPU}}, & \text{se } \Delta t_{\text{CPU}} \geq 0, \\ -1000\ \Delta t_{\text{CPU}}, & \text{c.c.} \end{cases} \tag{4.38}$$

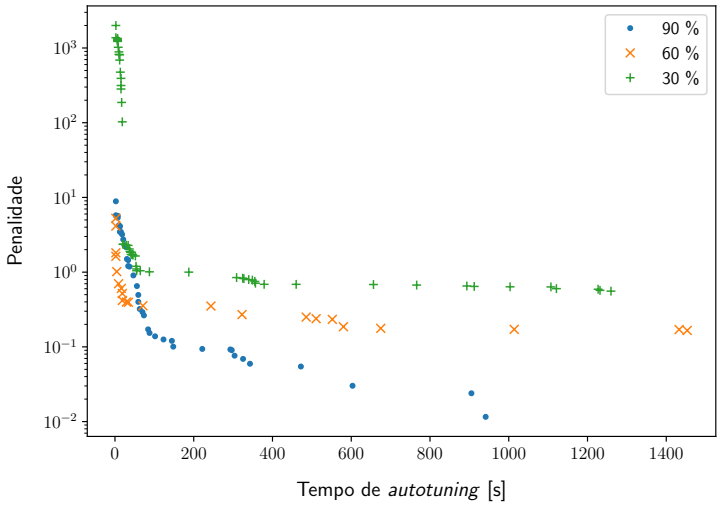

Figura 4.52: *`OpenTuner`: tempos gastos na otimização dos parâmetros de flexibilização.*

Na Figura 4.52 podem ser vistos os valores da função objetivo em função do tempo de execução da aplicação `OpenTuner`. Mostra-se que a penalidade decresce rapidamente nos primeiros 200 segundos, e a partir desse ponto o esforço para obter configurações melhores aumenta consideravelmente.

[50] Veja o arquivo `bandittechniques.py` no código fonte do OpenTuner.

[51] Essas técnicas são: `DifferentialEvolutionAlt`, `UniformGreedyMutation`, `NormalGreedyMutation` e `RandomNelderMead`; veja os respectivos arquivos: `differentialevolution.py`, `evolutionarytechniques.py` e `simplextechniques.py` no código fonte do OpenTuner.

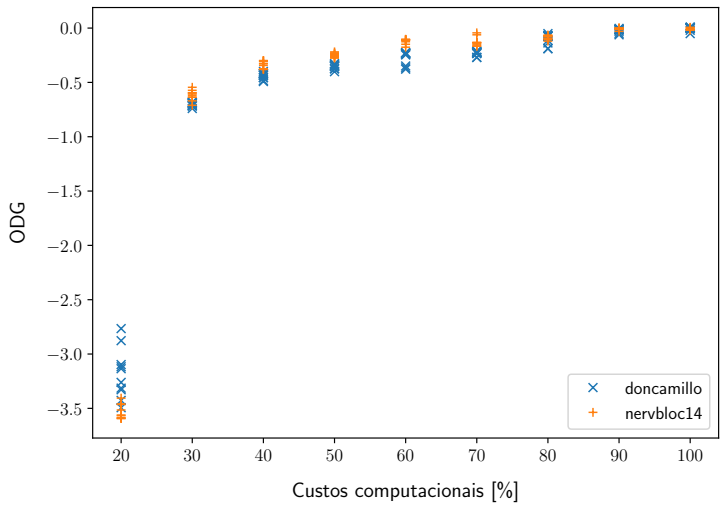

Figura 4.53: *Qualidade percebida em função dos custos computacionais da aplicação* `flexmix` *para dois sinais diferentes.*

A Figura 4.53 mostra a qualidade percebida resultante das configurações ótimas obtidas pela aplicação `OpenTuner` em função da carga computacional considerada para a aplicação `flexmix`. Vale lembrar que esses valores refletem as diferenças perceptuais associadas a entradas sonoras específicas utilizadas no experimento[52], e que estes valores podem sofrer alterações com outras entradas.

Observa-se que o ODG atinge valores considerados adequados[53] para a maioria das situações de carga computacional, sendo a diferença praticamente imperceptível para cargas acima de 50%. Estas observações evidenciam o fato de que a utilização da aplicação `OpenTuner` viabiliza a abordagem da flexibilização, porque permite produzir configurações de parâmetros que, por um lado, levam a custos computacionais que viabilizam a execução em tempo real e, por outro lado, são otimizados quanto à qualidade percebida.

4.5.5 `flexmix` em tempo real

O objetivo do experimento com a aplicação `flexmix` em tempo real é demonstrar a viabilidade da flexibilização de uma aplicação de processamento de áudio neste contexto de execução. O controle dos custos da aplicação é realizado por um gerenciador de flexibilização, que é implementado como parte da aplicação `flexmix`, como discutido na Seção 3.3.2.

O mapeamento das configurações (parâmetros de flexibilização) tem que ser estabelecido antes da execução da aplicação, através do mecanismo de afinação automática, utilizando-se entradas de sinais que sejam similares àquelas esperadas durante a execução em tempo real (mesmos instrumentos, mesmo repertório, etc.). Considerando sinais de entrada que possuem uma variação significante no tempo, os sinais são segmentados e um mapeamento é estabelecido para cada um desses segmentos. Os parâmetros de flexibilização relevantes para a aplicação `flexmix` encontram-se na Tabela 4.20.

Adicionalmente, deve-se considerar que cada mudança de um parâmetro da aplicação `flexmix` exige custos computacionais para a reconfiguração do pipeline (custos de transição), que podem ser diferentes para cada parâmetro de flexibilização. Por exemplo, a mudança do número de coeficientes ou da qualidade de reamostragem exige apenas reconfigurações de alguns elementos específicos, enquanto uma mudança da taxa de amostragem tem que ser sinalizada para todos os elementos do pipeline afetado. A fim de minimizar estes custos computacionais, o gerenciador de flexibilização

[52] As entradas sonoras usadas no experimento são cambridge-mt.com/ms/mtk/#DonCamilloChoir e cambridge-mt.com/ms/mtk/#Nervbloc do site *Cambridge Music Technology* (veja a Seção 4.1)

[53] lembrando que o valor -1.0 é interpretado como uma diferença "perceptível mas não irritante", veja a Figura 2.3 na Seção 2.3.2.

da aplicação `flexmix` considera não apenas a melhor configuração produzida pelo passo da afinação automática, mas um conjunto das melhores configurações, o que permite a escolha de uma configuração que leve a menores custos de reconfiguração.

No experimento, o gerenciador mostra-se capaz de reagir a mudanças da carga computacional, no entanto, o tempo de reação está relacionado a seu intervalo de execução.

Capítulo 5

Conclusões

Esta pesquisa se iniciou a partir da observação que aplicações de processamento de áudio em tempo real operam considerando uma qualidade fixa da saída, independentemente dos contextos de escuta ou outros fatore, e, em muitos casos, gastam recursos computacionais excessivos desnecessariamente. Por outro lado, não raros são os casos de sobrecarga nesse tipo de aplicação, o que produz interrupção do fluxo sonoro na saída.

Decorrentemente, se colocaram as questões de pesquisa, principalmente, como realizar um *trade-off* flexível entre custos computacionais e a qualidade percebida do resultado do processamento. Com o levantamento e desenvolvimento dos conceitos se esclareceu a necessidade de uma abordagem genérica, uma vez que as áreas de pesquisa envolvidas são diversas e precisavam que ser integradas. Esse assunto é tratado nos primeiros capítulos.

No capítulo 3 foi mostrado como a flexibilização de uma aplicação de processamento de áudio pode ser abordada, começando pela análise de elementos e cadeias de processamento, e detalhando o gerenciamento de recursos, sobretudo, do *trade-off* entre os custos computacionais e a qualidade percebida. Além disso, investigou-se de forma teórica elementos de processamento sonoro específicos, incluindo a proposta de um método de parametrização e otimização de um filtro FIR.

O Capítulo 3 apresenta esta abordagem como uma metodologia genérica para a abordagem e resolução das questões de pesquisa, independentemente de técnicas e aplicações concretas. A metodologia é exemplificada através de sua aplicação em algoritmos de áreas diferentes de sonorização, como por exemplo a convolução no domínio do tempo.

A metodologia discutida no capítulo 3 é submetida a um estudo de caso, sendo utilizada em uma aplicação de teste no capítulo 4. Neste capítulo são investigados os custos computacionais de elementos e cadeias de processamento concretos e suas possíveis parametrização. Adicionalmente, apresenta-se um método para avaliar a qualidade percebida, de forma objetiva, em função das parametrizações dos elementos da aplicação.

Neste capítulo as questões de pesquisa são revisitadas à luz dos experimentos e discussões dos capítulos 3 e 4, a fim de evidenciar os objetivos atingidos. São destacadas também as contribuições alcançadas e oferecidas algumas sugestões de pesquisas futuras.

5.1 Questões de pesquisa

A primeira questão de pesquisa colocada neste trabalho trata da possibilidade de proposição de um procedimento generalizado de flexibilização para algoritmos de processamento de áudio em tempo real. Os procedimentos propostos no Capítulo 3 constituem a abordagem teórica à flexibilização do *trade-off* entre qualidade e custo computacional, ao passo que no Capítulo 4 estes procedimentos são empregados em uma aplicação específica de processamento de áudio em tempo real. A abordagem tem limitações de aplicabilidade e de exequibilidade, como discutido na Seção 3.4, sendo que essas limitações ficam evidentes na análise e adaptação da aplicação de teste no Capítulo 4. Ainda que o experimento não demonstre a universalidade das observações e conclusões, estima-se que muitas outras aplicações relevantes em processamento de áudio admitem

o mesmo tipo de flexibilização.

A segunda questão de pesquisa aborda as formas com as quais um sistema de processamento de áudio pode controlar o *trade-off* entre custos computacionais e qualidade percebida de forma flexível. O controle do *trade-off* proposto neste trabalho apoia-se, por um lado, no controle dos custos computacionais dos elementos e cadeias flexibilizados e, por outro lado, no mapeamento das configurações dos parâmetros em relação à qualidade percebida. Um gerenciador de flexibilização, que detecta o estado da plataforma computacional através de sensores do sistema operacional, pode iniciar as reconfigurações necessárias dos elementos e cadeias da aplicação, considerando o mapeamento entre as configurações e a qualidade percebida.

A questão de pesquisa seguinte envolve o mapeamento dos custos computacionais em função dos parâmetros de flexibilização. Para responder a esta questão, esta pesquisa propõe a execução de uma análise dos custos computacionais dos elementos e cadeias de elementos da aplicação, a fim de estabelecer uma função teórica dos custos computacionais para a aplicação como um todo, cujos coeficientes são determinados por mensurações de desempenho dos elementos e cadeias de processamento. Esta abordagem *bottom-up*, ou seja, da análise dos elementos à análise da aplicação, mostra-se útil para confirmar a validade das funções teóricas em cada nível de detalhe e contribui para a consistência dos resultados obtidos.

Na sequência de questões de pesquisa, o mapeamento da qualidade percebida em função dos parâmetros de flexibilização é abordado. O espaço das configurações possíveis (parâmetros de flexibilização) pode ser grande demais para ser explorado de forma exaustiva, mesmo para aplicações simples como a aplicação de teste apresentada. Este fato exige duas coisas: uma métrica de avaliação de qualidade percebida que seja objetiva e automatizável, e um método de exploração capaz de encontrar configurações que respeitem os limites dos custos computacionais dados e que levem às melhores qualidades percebidas possíveis. Este trabalho propõe a utilização de um método de auto-afinação, que combina vários métodos de busca através de algoritmos de inteligência artificial e que se mostra eficiente para encontrar configurações adequadas.

A próxima questão de pesquisa trata da escolha dentre diferentes métodos de gerenciamento do trade-off em função dos objetivos da aplicação. O gerenciador de flexibilização utiliza o mapeamento produzido pelo processo de auto-afinação para aplicar as configurações necessárias (que evitem as situações de sobrecarga) de maneira global, dispensando o conhecimento das flexibilizações individuais de cada elemento ou de cada cadeia de elementos. Essa estratégia torna desnecessária a definição de estratégias locais aos elementos e cadeias, reduzindo o tempo de execução do gerenciador de flexibilização.

A última questão de pesquisa traz várias perguntas interrelacionadas, indagando em quais aplicações a abordagem proposta seria mais eficaz, em que situações o problema de sobrecarga computacional poderia ser aliviado ou resolvido, e até que ponto seria possível minimizar a perda de qualidade percebida adaptando-se dinamicamente os custos computacionais aos recursos disponíveis. A eficácia da abordagem proposta neste trabalho depende principalmente da amplitude de variação dos custos computacionais proporcionada pelos parâmetros de flexibilização, bem como da variação da qualidade percebida decorrente, sendo maior quando uma grande economia nos custos computacionais resulta em pequenas variações na qualidade percebida. Isso fica particularmente evidente na análise dos elementos e da aplicação de teste do Capítulo 4, onde se encontram elementos com custos computacionais relativamente pequenos e cuja flexibilização tem pouco impacto na variação dos custos computacionais, bem como nos elementos cuja flexibilização leva a mudanças drásticas de qualidade percebida sem uma correspondente economia nos custos computacionais, como por exemplo a flexibilização do número de canais no elemento de mixagem, que exemplifica os dois casos mencionados. Com a abordagem proposta neste trabalho, a variação de qualidade é inerente à abordagem de flexibilização, e a perda de qualidade inevitável em situações de sobrecarga; entretanto, a aplicação de teste mostra que a perda de qualidade pode ser desprezível em determinadas situações onde é possível obter uma grande economia computacional.

5.2 Contribuições alcançadas

Quanto às contribuições alcançadas, destaca-se a metodologia apresentada junto com as ferramentas teóricas desenvolvidas, que formam um arcabouço para pesquisas futuras que permite estender e aprofundar a abordagem da flexibilização, fornecendo não apenas uma perspectiva teórica, mas também métodos práticos relacionados. Tal extensão e aprofundamento torna-se possível porque a pesquisa que fundamenta a abordagem proposta conecta as áreas de processamento de sinais sonoros, a análise dos custos custos computacionais, métodos de otimização e a área de programação de aplicações de processamento de áudio.

A discussão das análises teóricas e dos experimentos, incluindo muitos exemplos concretos de flexibilização (especialmente o filtro FIR), permite uma compreensão do arcabouço que pode ser transformado e transferido para ser aplicado a outros elementos e também a outros arcabouços de áudio.

Outro ponto destacado é a adaptação do método de auto-afinação ao processamento sonoro flexível, que se mostra como uma abordagem eficaz para otimizar configurações sem a necessidade de uma formulação analítica conectando configurações, custos computacionais e qualidade percebida.

5.3 Sugestões para Pesquisas Futuras

Uma vez que a proposta desenvolvida neste trabalho é principalmente metodológica, esta pesquisa pode ser entendida como um convite para a sua aplicação, bem como para sua modificação e refinamento, em contextos análogos ou diversos aos apresentados. Destacam-se, como exemplos, as seguintes sugestões para pesquisa futura:

- utilizando as ferramentas apresentadas e desenvolvidas, muitos elementos e cadeias de processamento sonoro podem ser explorados, permitindo a flexibilização de outras aplicações de processamento de áudio;
- para a avaliação de qualidade, podem ser considerados métricas e algoritmos de avaliação adicionais, como por exemplo métodos de avaliação da fidelidade da espacialização (localização) de forma automatizada;
- considera-se a exploração do método de auto-afinação quanto à definição de funções objetivo alternativas, à alimentação do algoritmo com configurações específicas (*seed*) e à implementação de algoritmos de busca adicionais;
- pode-se integrar abordagens de gerenciamento de recursos de outros autores e, especialmente, aqueles que consideram análises espectrais e/ou análises baseadas na percepção (codificação perceptual);
- é possível melhorar a otimização de filtros FIR específicos à espacialização considerando-se características temporais importantes para a localização de fontes sonoras;
- considera-se o emprego do arcabouço experimental para investigar heurísticas de flexibilização e de gerenciamento, como, por exemplo, fatores de amplificação de erro de elementos e cadeias (*error-scaling factor*), ou o apoio pelo usuário que poderia definir prioridades para certos processamentos ou mesmo prioridades deduzidas automaticamente a partir de parâmetros psicoacústicos mensuráveis.

Apêndice A

Otimização do filtro FIR por otimização linear

O objetivo desta seção é apresentar um método para construir alternativas a um dado filtro FIR que usem um menor número de coeficientes. Esse método permite gerar um conjunto de filtros similares (ou seja, com espectros similares) para viabilizar a estratégia de flexibilização desenvolvida neste trabalho.

O método baseia-se no algoritmo proposto por Baran et al. (Baran *et al.*, 2010) que usa programação linear para a construção de filtros similares. Os autores seguem uma estratégia de diminuir progressivamente o número de coeficientes, repetindo os seguintes passos:

- construir um filtro novo que tem 1 coeficiente a menos do que o filtro do passo anterior (zerando o coeficiente com o menor valor absoluto);
- adaptar os coeficientes desse filtro por otimização linear para aproximá-lo do espectro do filtro original.

Neste trabalho usa-se a mesma abordagem porém com um critério de similaridade diferente, pois Baran et al. comparam espectros de magnitude e o método a seguir constrói um problema de otimização linear comparando espectros complexos.

A.1 Construção do problema de otimização linear

Dado um filtro FIR com M coeficientes $a_n,\ n = 0, \ldots, M-1$, como descrito na Seção 3.5.1, calcula-se sua transformada de Fourier como

$$A_k = \sum_{n=0}^{M-1} a_n e^{-i2\pi kn/M} \tag{A.1}$$

para cada componente espectral $k = 0, \ldots, M-1$.

Seja D o espectro do filtro original (espectro desejado) com componentes espectrais D_k, e A o espectro do filtro com menos coeficientes (e componentes espectrais A_k). Em uma primeira abordagem, define-se uma variável de folga $\hat{\delta} \in \mathbb{R}^+$ e a condição do problema linear:

$$W_k \left| A_k - D_k \right| \leq \hat{\delta}, \tag{A.2}$$

sendo W_k um fator de atenuação real e não negativo associado à frequência de Fourier $\omega_k = 2\pi k/M$. Estes fatores de atenuação podem ser usados para ponderar a folga associada a cada componente espectral de acordo com critérios perceptuais. Procura-se então o filtro A que tenha o menor $\hat{\delta}$, ou seja, o problema minimiza a função $\|A - D\|_\infty = \max_k\{|A_k - D_k|\}$.

Uma vez que não se pode construir um problema de programação linear a partir desta abordagem[1], tem-se que relaxar a condição[2]; propõe-se o uso de duas desigualdades, uma para as partes reais e outra para as partes imaginárias:

$$\begin{aligned} W_k \left| A_k^r - D_k^r \right| &\leq \delta \text{ e} \\ W_k \left| A_k^i - D_k^i \right| &\leq \delta. \end{aligned} \tag{A.3}$$

sendo A_k^r e A_k^i as componentes reais imaginárias de A_k, e $\delta \in \mathbb{R}^+$.
A seguir demonstra-se a construção do problema linear a partir das condições A.3 acima.
Substituindo as componentes A_k considerando a Equação A.1, obtém-se:

$$\begin{aligned} W_k \left| \sum_{n=0}^{M-1} a_n \cos(n\omega_k) - D_k^r \right| &\leq \delta, \\ W_k \left| \sum_{n=0}^{M-1} a_n \sin(-n\omega_k) - D_k^i \right| &\leq \delta. \end{aligned} \tag{A.4}$$

Para simplificar a notação, define-se:

$$f_r(k,n) = W_k \cos(n\omega_k) \tag{A.5}$$
$$f_i(k,n) = W_k \sin(-n\omega_k) \tag{A.6}$$
$$g_r(k) = W_k D_k^r \tag{A.7}$$
$$g_i(k) = W_k D_k^i. \tag{A.8}$$

Sendo W_k sempre positivo, obtém-se:

$$\left| \sum_{n=0}^{M-1} a_n f_r(k,n) - g_r(k) \right| \leq \delta, \tag{A.9}$$
$$\left| \sum_{n=0}^{M-1} a_n f_i(k,n) - g_i(k) \right| \leq \delta. \tag{A.10}$$

Eliminando-se o módulo, obtém-se:

$$\sum_{n=0}^{M-1} a_n f_r(k,n) - g_r(k) \leq \delta, \tag{A.11}$$
$$\sum_{n=0}^{M-1} a_n f_r(k,n) - g_r(k) \geq -\delta, \tag{A.12}$$
$$\sum_{n=0}^{M-1} a_n f_i(k,n) - g_i(k) \leq \delta \tag{A.13}$$
$$\sum_{n=0}^{M-1} a_n f_i(k,n) - g_i(k) \geq -\delta. \tag{A.14}$$

Sendo que $g_r(k)$ e $g_i(k)$ são valores constantes e conhecidos, pode-se formular o problema de otimização linear como encontrar $\delta \geq 0$ e $a_0, a_1, \ldots, a_{N-1} \in \mathbb{R}$ satisfazendo as condições abaixo:

[1] O cálculo da magnitude envolve termos quadráticos.
[2] Baran et. al consideram apenas uma certa classe de filtros (filtros causais tipo I com fase linear) para simplificar o problema.

$$
\begin{aligned}
&\min \delta \\
&\text{sujeito a} \\
&\qquad \text{para cada } k = 0, \ldots, M-1 \\
&\qquad \sum_{n=0}^{M-1} a_n f_r(k,n) - \delta && \leq g_r(k) \\
&\qquad \sum_{n=0}^{M-1} a_n f_r(k,n) + \delta && \geq g_r(k) \\
&\qquad \sum_{n=0}^{M-1} a_n f_i(k,n) - \delta && \leq g_i(k) \\
&\qquad \sum_{n=0}^{M-1} a_n f_i(k,n) + \delta && \geq g_i(k). \qquad \text{(A.15)}
\end{aligned}
$$

Tal formulação pode ser resolvida eficientemente pelo método Simplex, implementado em diversas bibliotecas.

Apêndice B

Ferramentas utilizadas

Plataforma / sistema operacional Linux

- `cpupower`: mostra e configura valores relacionados à potência da CPU
- `chrt`: manipula atributos de tempo real de um processo
- /proc/sys: interface do kernel para manipular a vários aspectos de um processo (por exemplo: `/usr/bin/echo 3 > /proc/sys/vm/drop_caches` para apagar a memória cache de um processo)
- `perf`: ferramentas de análise de desempenho para Linux
- `stress-ng`: ferramenta para estressar uma plataforma computacional
  github.com/ColinIanKing/stress-ng

`Gstreamer` gstreamer.freedesktop.org

- `gst-launch`: construir e executar pipelines (contida no `Gstreamer`)
- `gst-instruments`: ferramentas para inspecionar e perfilamento de pipelines
  github.com/kirushyk/gst-instruments
- `pipeviz`: editor visual de grafos para a construir e testar pipelines
  github.com/virinext/pipeviz
- `GstShark`: ferramentas para inspecionar e perfilamento de pipelines usando *tracers*
  github.com/RidgeRun/gst-shark
- `GstPEAQ`: *plugin* e aplicação para avaliar PEAQ
  github.com/HSU-ANT/gstpeaq

Qualidade

- `EAQUAL`: implementação ITU-R recommendation BS.1387 (Windows)
  github.com/godock/eaqual
- `peaqb`: evaluação PEAQ (Gottardi); desenvolvimento interrompido
  sourceforge.net/projects/peaqb
- `peaq-fast`: evaluação PEAQ (Akinori Ito) baseado no algoritmo de Gottardi
  github.com/akinori-ito/peaqb-fast
- `AFsp`: biblioteca utilizada com `peaqb` e `peaq-fast`
  www-mmsp.ece.mcgill.ca/MMSP/Documents/Software/Packages/AFsp/AFsp/AFsp.html

- `PQevalAudio`: implementação PEAQ básico (Kabal, 2003)
  www-mmsp.ece.mcgill.ca/MMSP/Documents/Downloads/PQevalAudio/PQevalAudio-v1r0.tar.gz
- `libtsp`: biblioteca utilizada com `PQevalAudio`
  www-mmsp.ece.mcgill.ca/MMSP/Documents/Software/Packages/libtsp/libtsp.html

Autotuning

- `OpenTuner`: ferramenta para a afinação automática
  github.com/jansel/opentuner

Outras Ferramentas e aplicações

- `loudness-scanner`: ferramenta estimar `loudness` seguindo a norma EBU R128
  github.com/jiixyj/loudness-scanner
- `pyroomacoustics`: ferramentas para a simulação de salas
  github.com/LCAV/pyroomacoustics

Referências Bibliográficas

Allen e Berkley(1979) J. B. Allen e D. A. Berkley. Image method for efficiently simulating small-room acoustics. *J. Acoust. Soc. Am*, 65(4):943–950. Citado na pág. 23

Ansel *et al.***(2013)** J. Ansel, S. Kamil, K. Veeramachaneni, U-M. OReilly e S. Amarasinghe. Opentuner: An extensible framework for program autotuning. Em *Computer Science and Artificial Intelligence Laboratory Technical Report*. MIT-CSAIL-TR-2013-026. Citado na pág. 52, 127

Baalman *et al.***(2007)** M. A. J. Baalman, T. Hohn, Schampijer S. e T. Koch. Renewed architecture of the swonder software for wave field synthesis on large scale systems. Em *Proceedings of the 5th International Linux Audio Conference*, páginas 76–83, Berlin, Germany. Citado na pág. 61

Bagnoli(2010) G. Bagnoli. Design and development of a mechanism for low-latency real time audio processing on linux. Dissertação de Mestrado, Facoltá di Ingegneria, Universitá di Pisa, Pisa, Italia. Citado na pág. 43

Baran *et al.***(2010)** T. Baran, D. Wei e A.V. Oppenheim. Linear programming algorithms for sparse filter design. *Signal Processing, IEEE Transactions on*, 58(3):1605–1617. ISSN 1053-587X. doi: 10.1109/TSP.2009.2036471. Citado na pág. 57, 58, 135

Bech e Zacharov(2006) S. Bech e N. Zacharov. *Perceptual Audio Evaluation - Theory, Method and Application*. John Wiley and Sons, Ltd, Chichester, UK, 1ª edição. ISBN 978-0470869239. Citado na pág. 11, 13

Berkhout *et al.***(1993)** A. Berkhout, Diemer Vries e P VOGEL. Acoustic control by wave field synthesis. *J.Acoust.Soc.Am.*, 93:2764–2778. doi: 10.1121/1.405852. Citado na pág. 61

Berkhout(1988) A.J. Berkhout. A holographic approach to acoustic control. *Journal of the Audio Engineering Society*, 36:977–995. Citado na pág. 24

Bianchi(2013) A. J. Bianchi. Processamento de áudio em tempo real em dispositivos computacionais de baixo custo e alta disponibilidade. Dissertação. orientador: Marcelo gomes de queiroz, Instituto de Matemática e Estatística, IME-USP, São Paulo. URL http://www.teses.usp.br/teses/disponiveis/45/45134/tde-23012014-190028/pt-br.php. Citado na pág. 19, 40, 56, 82

Bitto *et al.***(1999)** C. Bitto, R.; Schmidmer, T. Sporer, K. Brandenburg, T. Thiede, W.C. Treurniet, J.G. Beerends, C. Colomes, M. Keyhl, G. Stoll e B. Feiten. PEAQ: Der künftige ITU-Standard zur objektiven Messung der wahrgenommenen Audioqualität. Em *Verband Deutscher Tonmeister, Bildungswerk: 20. Tonmeistertagung 1998. Bericht*, München, Germany. Saur. Citado na pág. 14, 15

Blauert e Jekosch(2010) J. Blauert e U Jekosch. A layer model of sound quality. Em *Proceedings of 3rd International Workshop on Perceptual Quality of Systems (PQS 2010)*, páginas 18–23, Bautzen, Germany. doi: 10.21437/PQS.2010-4. Citado na pág. 50

Bonneel *et al.***(2010)** N. Bonneel, C. Suied, I. Viaud-Delmon e G. Drettakis. Bimodal perception of audio-visual material properties for virtual environments. *ACM Transactions on Applied Perception*, 7. doi: 10.1145/1658349.1658350. Citado na pág. 13

Boudreaux e Parks(1983) G. Boudreaux e T.W. Parks. Thinning digital filters: A piecewise-exponential approximation approach. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 31(1):105–113. ISSN 0096-3518. doi: 10.1109/TASSP.1983.1164058. Citado na pág. 57

Clark(1976) J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554. ISSN 0001-0782. doi: 10.1145/360349.360354. URL http://doi.acm.org/10.1145/360349.360354. Citado na pág. 21

Collins(2010) N. Collins. *Introduction to Computer Music*. John Wiley and Sons, Ltd, Chichester, UK, 1^a edição. Citado na pág. 37, 44

Cormen *et al.***(2000)** T.H. Cormen, C.E. Leiserson e R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 5^a edição. Citado na pág. 38

Cremer(1948) L. Cremer. *Die wissenschaflichen Grundlagen der Raumakustik*, volume 1. Hirzel-Verlag, Stuttgart, 1^a edição. Citado na pág. 23

Crochiere e Rabiner(1983a) R. E. Crochiere e L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1^a edição. Citado na pág. 44

Crochiere e Rabiner(1983b) R. E. Crochiere e L. R. Rabiner. *Multirate digital signal processing*. Prentice-Hall Signal Processing Series. Prentice-Hall Inc., Upper Saddle River, NJ, 1^a edição. Citado na pág. 103

De Moor *et al.***(2003)** B. De Moor, A. Nackaerts e B. Schiettecatte. Real-Time acoustics simulation using mesh-tracing. Em *Proceedings of the International Computer Music Conference 2003*. Citado na pág. 60

Dolson(1986) M. Dolson. The phase vocoder: A tutorial. *Computer Music Journal*, 10(4):14–27. Citado na pág. 19

Egger(2013) K. Egger. Implementation and evaluation of auditory models for sound localization. Relatório técnico, Institut für Elektronische Musik und Akustik, Kunst Uni Graz. URL amtoolbox.sourceforge.net/notes/amtnote005.pdf. Supervisor: Dr. Piotr Majdak. Citado na pág. 13, 75

Feng e Liu(1997) Wu-Chun Feng e J.W.S. Liu. Algorithms for scheduling real-time tasks with input error and end-to-end deadlines. *Software Engineering, IEEE Transactions on*, 23(2):93–106. Citado na pág. 2, 18, 51

Fletcher(1929) H. Fletcher. *Speech and Hearing*. D. Van Nostrand Company,Inc, New York. Citado na pág. 11

Fletcher(1940) H. Fletcher. Auditory patterns. *Reviews of Modern Physics*, 12:47–65. doi: 10.1103/RevModPhys.12.47. URL http://link.aps.org/doi/10.1103/RevModPhys.12.47. Citado na pág. 10

Fouad *et al.***(1997)** H. Fouad, J. K. Hahn e J. A. Ballas. Perceptually based scheduling algorithms for real time synthesis of complex sonic environments. Em *In Proc. of the 1997 International Conference on Auditory Display, Xerox Palo Alto Research*. Citado na pág. 28

Fouad *et al.***(2001)** H. Fouad, B. Narahari e J. K. Hahn. *A Real-Time Parallel Scheduler for the Imprecise Computation Model*, página 25–36. Nova Science Publishers, Inc., USA. Citado na pág. 17

Foucault(2012) M. Foucault. *A Arqueologia do Saber*. Forense Universitária, Rio de Janeiro, 8^a edição. Citado na pág. 1

Gardner(1995) W. G. Gardner. Efficient convolution without input-output delay. *Journal of the Audio Engineering Society*, 43(3):127–136. Citado na pág. 56

Gebali(2011) F Gebali. *Algorithms and Parallel Computing*. John Wiley and Sons, Ltd, 1ª edição. Citado na pág. 46

Hammershøi e Møller(2005) D. Hammershøi e H. Møller. Binaural technique — basic methods for recording, synthesis, and reproduction. Em Jens Blauert, editor, *Communication Acoustics*, páginas 223–254. Springer-Verlag, Berlin Heidelberg. ISBN 978-3-540-22162-3. Citado na pág. 62

Herder(1998) J. Herder. Sound spatialization framework: An audio toolkit for virtual environments. *Journal of the 3D-Forum Society*, 12(3):17 – 22. Citado na pág. 25

Herder(1999a) J. Herder. Optimization of sound spatialization resource management through clustering. *The Journal of Three Dimensional Images, 3D-Forum Society*, páginas 59–65. Citado na pág. 2, 25, 62

Herder(1999b) Jens Herder. *A Sound Spatialization Resource Managment Framework*. Tese de Doutorado, Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Japão. Citado na pág. 25

Hoffmann *et al.***(2009)** H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal e M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Relatório técnico, Massachusetts Institute Cambridge, Cambridge, ME, USA. Citado na pág. 20

Holters e Zölzer(2015) M. Holters e U. Zölzer. Gstpeaq – an open source implementation of the peaq algorithm. Em *18th International Conference on Digital Audio Effects (DAFx'15)*, Trondheim, Norway. Citado na pág. 67, 75

Kabal(2003) P. Kabal. An examination and interpretation of ITU-R BS. 1387: Perceptual evaluation of audio quality. Relatório técnico, Dept. Electrical & Computer Engineering, McGill University. Citado na pág. 75, 116, 140

Kenny e Lin(1991) K. B. Kenny e K. J. Lin. Building flexible real-time systems using the flex language. Em *ICCL'88-Part I Computer languages: A perspective*, volume 24, páginas 70 – 78, Los Alamitos, CA, USA. IEEE Computer Society. doi: 10.1109/2.76288. Citado na pág. 29

Kerrisk(2010) M. Kerrisk. *The Linux Programming Interface*. No Starch Press, Inc., San Francisco, CA, USA, 1ª edição. Citado na pág. 16, 47, 72, 75, 98

Kitchin *et al.***(2009)** D. Kitchin, A. Quark, W. Cook e J. Misra. The orc programming language. páginas 1–25. ISBN 978-3-642-02137-4. doi: 10.1007/978-3-642-02138-1_1. Citado na pág. 94

Kleczkowski e Pluta(2014) P. Kleczkowski e M. Pluta. Perceptual evaluation of the effect of threshold in selective mixing of sounds. *Acta Physica Polonica A*, 125(Issue 4A):117–121. doi: dx.doi.org/10.12693/APhysPolA.125.A-117. URL http://psjd.icm.edu.pl/psjd/element/bwmeta1.element.bwnjournal-article-appv125n4a23kz. Citado na pág. 28, 29

Knuth(1998) D.E. Knuth. *The Art of Computer Programming*, volume 1-3. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. Citado na pág. 38, 68

Koch(2008) T. Koch. Aufbau, betrieb und optimierung eines clusters von gnu/linux workstations für die wellenfeldsynthese. Dissertação de Mestrado, Institut für Sprache und Kommunikation, Technische Universität Berlin, Berlin, Deutschland. Citado na pág. 16

Krokstad(1968) A. Krokstad. Calculating the acoustical room response by the use of a ray tracing technique. *Journal of Sound and Vibration*, 8:118–125. ISSN 0022460X. doi: 10.1016/0022-460x(68)90198-3. Citado na pág. 23

Letz *et al.***(2005)** S. Letz, Y. Orlarey e D. Fober. Jack audio server for multi-processor machines. Em ICMA, editor, *Proceedings of the International Computer Music Conference (ICMA)*, páginas 1–4. Citado na pág. 31, 43

Lin e Natarajan(1991) K. J. Lin e S. Natarajan. Flex: Towards flexible real-time programs. Em *ICCL'88-Part I Computer languages: A perspective*, volume 16, páginas 65 – 79, Los Alamitos, CA, USA. doi: 10.1016/0096-0551(91)90017-4. Citado na pág. 29

Lin *et al.***(1987)** K.-J. Lin, S. Natarajan e J. W. S. Liu. Imprecise results: Utilizing partial computations in real-time systems. Em *Proceedings IEEE Eighth Real-Time Systems Symposium*, páginas 210–217. Madrid, Spain. Citado na pág. 2, 16

Liu *et al.***(1987)** J. Liu, K.J. Lin e S. Natarajan. Scheduling real-time, periodic jobs using imprecise results. *Proceedings of the IEEE 8th Real-Time Systems Symposium*. Citado na pág. 17, 28

Liu(2000) J. W. S. Liu. *Real-Time Systems*. Modern Acoustics and Signal Processing. Prentice Hall PTR, Upper Saddle River, NJ, 1^a edição. ISBN 0130996513. Citado na pág. 15, 16, 17

Luebke *et al.***(2003)** D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson e R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1^a edição. Citado na pág. 21

Marr *et al.***(2002)** D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller e M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6. Citado na pág. 71, 72

Moeck *et al.***(2007)** T. Moeck, N. Bonneel, N. Tsingos, G. Drettakis, I. Viaud-Delmon e D. Alloza. Progressive perceptual audio rendering of complex scenes. Em *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, páginas 189–196, New York, NY, USA. Association for Computing Machinery. ISBN 978-1-59593-628-8. doi: 10.1145/1230100.1230133. Citado na pág. 27

Moldrzyk *et al.***(2007)** C. Moldrzyk, A. Goertz, M. Makarski, S. Feistel, W. Ahnert e S. Weinzierl. Wellenfeldsynthese für einen großen Hörsaal. Em *Fortschritte der Akustik - DAGA 2007*, páginas 683–684. Deutsche Gesellschaft für Akustik. URL http://www2.ak.tu-berlin.de/~akgroup/ak_pub/2007/Moldrzyk_2007_Wellenfeldsynthese_fuer_einen_grossen_Hoersaal.pdf. Citado na pág. 22

Montesião de Sousa(2010) G. H. Montesião de Sousa. *Auralização de fontes sonoras móveis usando HRTFs*. Tese de Doutorado, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil. Citado na pág. 62

Moore(1990) F R. Moore. *Elements of Computer Music*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1^a edição. ISBN 0-13-252552-6. Citado na pág. 8, 9, 37, 44, 56, 83

Natarajan e Broman(2018) S. Natarajan e D. Broman. Timed c: An extension to the c programming language for real-time systems. Em *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, páginas 227–239, Porto, Portugal. Citado na pág. 29

Nunnally e Bernstein(1994) J. Nunnally e I. H. Bernstein. *Psychometric Theory*. The McGraw Hill Companies, New York, 3^a edição. Citado na pág. 12

Oppenheim *et al.***(1999)** A. V. Oppenheim, R. W. Schafer e J. R. Buck. *Discrete-time Signal Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2^a edição. ISBN 0-13-754920-2. Citado na pág. 7, 8, 56

Painter e Spanias(2000) T. Painter e A. Spanias. Perceptual coding of digital audio. *Proceedings of the IEEE*, 88(4):451–515. Citado na pág. 2, 9, 10

Pelzer *et al.***(2014)** S. Pelzer, L. Aspöck, D. Schröder e M. Vorländer. Integrating real-time room acoustics simulation into a cad modeling software to enhance the architectural design process. *Buildings*, 4(2):113. ISSN 2075-5309. doi: 10.3390/buildings4020113. URL http://www.mdpi.com/2075-5309/4/2/113. Citado na pág. 23

Pinto(2006) E. C. Pinto. Repensando os commons na comunicação científica. Dissertação de Mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, Brasil. Citado na pág. 66

Piotr *et al.***(2014)** M. Piotr, R. Baumgartner, M. Takanen, S. Olli, V. Pulkki e P. Søndergaard. Modeling human sound-source localization with the auditory modeling toolbox. Em *ICSV21 The 21st International Congress on Sound and Vibration*, Pequim, China. 13-17 July. Citado na pág. 13, 75

Pohlmann(2000) K. C. Pohlmann. *Principles of Digital Audio*. McGraw-Hill Professional, 4th edição. ISBN 0071348190. Citado na pág. 10

Proakis e Manolakis(1996) J. G. Proakis e D. G. Manolakis. *Digital Signal Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 3ª edição. ISBN 0-13-394289-9. Citado na pág. 55

Rayleigh(1907) L. Rayleigh. Xii. on our perception of sound direction. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 13(74):214–232. doi: 10.1080/14786440709463595. Citado na pág. 11

Rodgers(1985) D. P. Rodgers. Improvements in multiprocessor system design. Em *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, página 225–231, Washington, DC, USA. IEEE Computer Society Press. ISBN 0818606347. Citado na pág. 73

Sabine(1923) W. C. Sabine. *Collected Papers on Acoustics*. Harvard University Press, Cambridge, MA, 1ª edição. Citado na pág. 22

Schafer e Rabiner(1973) R.W. Schafer e L. Rabiner. A digital signal processing approach to interpolation. *Proceedings of the IEEE*, 61(6):692–702. ISSN 0018-9219. doi: 10.1109/PROC.1973.9150. Citado na pág. 57

Schröder(2011) D. Schröder. *Physically based real-time auralization of interactive virtual environments*. Tese de Doutorado, Technische Hochschule Aachen, Berlin. URL http://publications.rwth-aachen.de/record/50580. Citado na pág. 60

Schroeder *et al.***(1962)** M. R. Schroeder, B. S. Atal e C. Bird. Digital computers in room acoustics. Em *Proceedings Fourth International Congress on Acoustics*, página M21, Copenhagen. Citado na pág. 22

Sedgewick e Wayne(2011) R. Sedgewick e K. Wayne. *Algorithms*. Addison-Wesley, Boston, USA, 4ª edição. ISBN 978-0321573513. Citado na pág. 38

Sidiroglou *et al.***(2011)** S. Sidiroglou, S. Misailovic, H. Hoffmann e M. Rinard. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. Em *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)* , Szeged, Hungary. Citado na pág. 20

Steinmetz e Nahrstedt(1995) R. Steinmetz e K. Nahrstedt. *Multimedia: Computing, Communications, and Applications*. Innovative technology series. Prentice Hall, Upper Saddle River, NJ. Citado na pág. 2

Sterne(2012) J. Sterne. *MP3: The Meaning of a Format*. Sign, Storage, Transmission. Duke University Press, Durham and London, UK. Citado na pág. 11, 51

Tanenbaum(2009) A. S. Tanenbaum. *Modern Operating Systems*. Pearson Education International, Upper Saddle River, NJ, USA, 3^a edição. Citado na pág. 16, 71

Taymans *et al.***(2015)** W. Taymans, S. Baker, Wingo A., R. S. Bultje e S. Kost. *GStreamer Application Development Manual*, 2015. [Online; acessado 24-outobro-2015]. Citado na pág. 32, 43

Thakur *et al.***(2009)** A. Thakur, A. G. Banerjee e S. K. Gupta. A survey of cad model simplification techniques for physics-based simulation applications. *Computer Aided Design*, 41(2):65–80. Citado na pág. 60

Tsingos(2005) N. Tsingos. Scalable perceptual mixing and filtering of audio signals using an augmented spectral representation. Em *Proceedings of the International Conference on Digital Audio Effects*, Madrid, Spain. Citado na pág. 2, 26, 27, 62

Tsingos *et al.***(2003)** N. Tsingos, E. Gallo e G. Drettakis. Perceptual audio rendering of complex virtual environments. Relatório Técnico RR-4734, Institut national de recherche en sciences et technologies du numérique (INRIA), REVES/INRIA Sophia-Antipolis. Citado na pág. 27, 62

Vermeulen *et al.***(1995)** A. Vermeulen, G. Beged-dov e P. Thompson. The pipeline design pattern. Em *Pro eedings of OOPSLA'95 Workshop on Design Patterns for Conurrent, Parallel, and Distributed Object-Oriented Systems*. Citado na pág. 36

Vorländer(2008) M. Vorländer. *Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality*. RWTHedition. Springer-Verlag, Berlin Heidelberg, 1^a edição. ISBN 9783540488309. Citado na pág. 22, 24, 25, 54, 60, 61

Weinzierl(2008) S. Weinzierl. *Handbuch der Audiotechnik*. VDI-Buch. Springer-Verlag, Berlin Heidelberg. ISBN 9783540343004. URL https://books.google.de/books?id=OLgY0QpXD0YC. Citado na pág. 25

Zaunschirm *et al.***(2014)** M. Zaunschirm, M. Frank e A. Sontacchi. Audio quality: Comparison of peaq and formal listening test results. Em *Proceedings of 6th Congress of the Alps Adria Acoustics Association*, Graz, Austria. Alps Adria Acoustics Association. Citado na pág. 15

Zwicker e Fastl(2008) E. Zwicker e H. H. Fastl. *Psychoacoustics: facts and models*. Springer Series in Information Sciences. Springer Verlag, Berlin Heidelberg. Citado na pág. 10, 11

Zölzer(2011) U. Zölzer, editor. *DAFX: Digital Audio Effects*. John Wiley and Sons, Ltd, Hamburg, Germany, 2^a edição. Citado na pág. 37