AudioLazy:
Processamento digital de sinais expressivo e em tempo real

Danilo de Jesus da Silva Bellini

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

Programa: Ciência da Computação

Orientador: Prof. Dr. Marcelo Gomes de Queiroz

Durante parte do desenvolvimento deste trabalho o autor recebeu auxílio financeiro do CNPq

São Paulo, fevereiro de 2013

AudioLazy:
Processamento digital de sinais expressivo e em tempo real

Esta é a versão original da dissertação elaborada pelo candidato Danilo de Jesus da Silva Bellini, tal como submetida à Comissão Julgadora.

Comissão Julgadora:

- Prof. Dr. Marcelo Gomes de Queiroz (orientador) - IME-USP
- Prof. Dr. Miguel Arjona Ramírez - Poli-USP
- Prof. Dr. Guido Stolfi - Poli-USP

Resumo

Há muitas ferramentas e pacotes de cálculo numérico que permitem o desenvolvimento de *software* de maneira expressiva. Entretanto, a avaliação imediata realizada pela maioria dessas ferramentas torna difícil, senão impossível, usá-las para processamento digital de sinais em tempo real. Priorizando a expressividade, clareza e simplicidade de código por parte do programador, e objetivando o uso junto a ferramentas disponíveis na linguagem em que foi escrita, AudioLazy é um pacote escrito em puro Python propondo uma maneira alternativa de desenvolvimento em processamento digital de sinais que permite o processamento em tempo real.

Dos aspectos analisados nesse texto, fazem parte a avaliação da expressividade de linguagens e da plausibilidade de realização de processamento em tempo real, a busca por possibilidades de automação fornecidas pela linguagem, a influência prática de diferentes formas de implementação de filtros digitais, um resumo de parte da bibliografia sobre correlatos psicoacústicos e maneiras de obtenção de informação musical a partir do áudio, e de processamentos que modelam a parte periférica da audição humana (filtros *gammatone*), além de diversos outros aspectos associados direta ou indiretamente com o desenvolvimento do pacote de processamento de áudio e seu uso, incluindo modelos de síntese, interatividade, realização de testes automatizados e avaliação da cobertura de código, documentação técnica, entre outros.

A escolha de implementações de sinais ou fluxos de informação através de iteráveis com avaliação tardia, de filtros lineares através das transformadas Z de sua resposta ao impulso generalizados a fim de permitir coeficientes variantes no tempo, além de polinômios como casos particulares de somas de potências representadas por dicionários, envolvendo sempre o uso de sobrecarga de operadores e metaprogramação, faz parte daquilo que serviu de base para a elaboração do referido pacote de análise, processamento e síntese de áudio.
Palavras-chave: áudio, filtros, tempo real, transcrição, síntese musical, modelo de audição, análise de áudio musical, expressividade, avaliação tardia.

Abstract

There are many numerical tools and packages that allows expressive *software* development. However, the eager evaluation model adopted by most of these tools makes it difficult, perhaps impossible, to use them for real time digital signal processing. Prioritizing code expressiveness, clarity and simplicity for the programmer, and aiming to be used together with available tools in the language in which it was written, AudioLazy is a package written in pure Python proposing an alternative development in digital signal processing that allows real time processing.

Among the analyzed aspects in this text, are incuded the appraisal of language expressivity and actual real time processing plausibility, the pursuit for possible automation provided by language features, the practical influence of different digital filter implementation forms, a summary of some literature on psychoacoustic correlates and ways of performing music information retrieval from audio, and processing models for the peripheral part of human hearing (*gammatone* filters), besides several other aspects directly or indirectly linked with the audio processing package development and application, including synthesis models, interactivity, automated testing and code coverage evaluation, technical documentation, among others.

The choice of implementations of signals or information streams via iterables with lazy evaluation, of linear filters through the Z transform of its impulse response generalized to allow for time-varying coefficients, as well as polynomial as special cases of sums of powers represented by dictionaries, always involving the use of operator overloading and metaprogramming, is part of what was the basis for the design of this audio analysis, processing and synthesis package.

Keywords: audio, filters, real time, transcription, music synthesis, auditory model, music information retrieval, expressiveness, lazy evaluation.

Sumário

Lista de Abreviaturas

ADSR	Envoltória dinâmica: ataque, decaimento, sustentação, liberação (*Attack, Decay, Sustain, Release*).
AIM	Implementação do modelo auditivo de Roy Patterson (*Auditory Image Model*).
AMDF	Função média da diferença de magnitude (*Average Magnitude Difference Function*).
ASA	*Auditory Scene Analysis*.
ASCII	*American Standard Code for Information Interchange*.
BIBO	Entrada limitada, saída limitada (*Bounded Input, Bounded Output*).
CAS	Sistema computacional de manipulação algébrica (*Computer Algebra System*).
CASA	*Computational Auditory Scene Analysis*.
c.c.	Caso contrário (utilizado em equações).
CNPq	Conselho Nacional de Desenvolvimento Científico e Tecnológico.
CQT	Transformada com Q constante *Constant Q Transform*.
DAFx	Efeitos de áudio digital, conferência (*Digital Audio Effects*).
DFT	Transformada discreta (circular) de Fourier (*Discrete Fourier Transform*).
DRY	Princípio que consiste em evitar a repetição de tarefas automatizáveis (*Don't Repeat Yourself*).
DSL	Linguagem de domínio específico (*Domain Specific Language*).
DSP	Processamento digital de sinais. (*Digital Signal Processing*).
DTFT	Transformada de Fourier em tempo discreto (*Discrete Time Fourier Transform*).
DVI	Formato padrão de saída do LaTeX. (*Device independent file format*).
e.g.	Por exemplo (do latim *exempli gratia*).
EPD	Distribuição do Python da Enthought, voltada para cientistas (*Enthought Python Distribution*).
ePUB	Formato de publicação eletrônica em *e-books* baseado no HTML (*Electronic Publication*).
eq.	Equação.
ERB	Largura de banda equivalente retangular (*Equivalent Rectangular Bandwidth*).
FAUST	DSL funcional voltada para áudio (*Functional AUdio STream*).
FFT	Transformada discreta rápida de Fourier (*Fast Fourier Transform*).
FIR	Resposta ao impulso finita, em duração (*Finite Impulse Response*).
FM	Modulação em frequência (*Frequency Modulation*).
GPL	Licença pública geral do GNU (*General Public License*).
GM	Padrão geral de valores MIDI (*General MIDI*).
GUI	Interface gráfica com o usuário (*Graphical User Interface*).
HTML	Linguagem de marcação de hipertexto (*HyperText Markup Language*).
IDE	Ambiente de desenvolvimento integrado (*Integrated Development Environment*).
i.e.	Isto é (do latim *id est*).
IIR	Resposta ao impulso infinita, em duração (*Infinite Impulse Response*).

IME	Instituto de Matemática e Estatística da USP.
IRCAM	*Institut de Recherche et Coordination Acoustique/Musique*.
ISMIR	Sociedade internacional para MIR (*International Society for Music Information Retrieval*).
JIT	Sigla referente à compilação realizada no instante da execução (*Just In Time*).
LPC	Codificação preditiva linear (*Linear Predictive Coding*).
LSF	Frequência da raia espectral (*Line Spectral Frequency*).
LTI	Linear e invariante no tempo, tipo de filtros (*Linear Time Invariant*.
MARL	Laboratório de pesquisa em música e áudio na NYU (*Music and Audio Research Lab*).
MFCC	Coeficientes cepstrais em escala mel (*Mel-frequency cepstral coefficients*).
MIDI	*Musical Instrument Digital Interface*.
MIR	Coleta de informações musicais *Musical Information Retrieval*.
NES	Video-game de 8 bits da Nintendo (*Nintendo Entertainment System*).
NYU	Universidade de Nova York (*New York University*).
OSI	*Open Source Initiative*.
PARCOR	Coeficientes de reflexão ou de correlação parcial.
PDF	Formato portável de documento (*Portable Document Format*).
PG	(Progressão Geométrica).
Poli	Escola Politécnica da USP.
PyPI	Índice central e *host* de pacotes Python (*Python Package Index*).
RAM	Memória de acesso aleatório (*Random Access Memory*).
RGB	Vermelho, Verde e Azul (*Red, Green, Blue*).
RMS	Raiz da média quadrática (*Root Mean Square*).
STL	Biblioteca padrão do C++ (*Standard Template Library*).
SVN	Subversion, sistema de controle de versão.
SysEx	*System Exclusive*.
TDD	Desenvolvimento orientado a testes (*Test-Driven Development*).
URL	Endereço padronizado para localização de recursos (*Uniform Resource Locator*).
USP	Universidade de São Paulo.
UTF-8	Unicode de comprimento variável, mantendo compatibilidade com o padrão ASCII de 7 bits.
UTF-n	Um dos padrões de codificação unicode para caracteres (*Unicode Transformation Format*).
YAAFE	*Yet Another Audio Feature Extractor*.

Lista de Símbolos

$\delta[n]$	Função delta de Kronecker, impulso.
Δ	Tamanho de um intervalo, de tempo (ver T_T) ou frequência ($\sqrt[N]{2}$ na escala temperada de N sons).
η	Ordem do filtro gammatone.
τ	Período.
ω	Frequência angular ($2\pi/\tau$).
$\mathcal{F}(\cdot)$	Operador transformada de Fourier.
f	Frequência.
f_{Ny}	Taxa de Nyquist (equivalente a $\pi/2$ radianos por amostra).
f_s	Frequência de amostragem.
$h[n]$	Resposta ao impulso (filtro digital).
$H(z)$	Equação do sistema (transformada Z da resposta ao impulso).
$\mathbb{N}$	Conjunto dos números naturais.
$O(\cdot)$	Complexidade computacional relativa a um limite superior para a demanda por recursos, a menos de constantes multiplicativas.
$\mathbb{Q}$	Conjunto dos números racionais.
R	Raio (magnitude) dos pólos de um filtro ressonador ou gammatone.
$\mathbb{R}$	Conjunto dos números reais.
s	Variável da transformada de Laplace.
t	Tempo.
T_s	Período de amostragem.
T_T	Duração mínima (tatum) no MIDI, que pode ser visto como unidade de tempo do Δ.
$u[n]$	Função de Heaviside, degrau.
$x(t)$	Função no domínio $\mathbb{R}$ dos números reais.
$x[n]$	Sequência, em geral representando um sinal digital ou amostrado no domínio do tempo.
$X(\cdot)$	Transformada Z de $x[n]$ (argumento z) ou transformada de Fourier de $x(t)$ (argumento ω).
$\mathcal{Z}(\cdot)$	Operador transformada Z.
$\mathbb{Z}$	Conjunto dos números inteiros.
z	Frequência complexa, em geral descrita por coordenadas polares $Ae^{j\omega}$.

Lista de Figuras

Lista de Tabelas

Lista de códigos-fonte

Capítulo 1

Introdução

Áudio é uma palavra utilizada para representar um som ou uma gravação de um som. Seres humanos utilizam os sons para a comunicação, para o entretenimento e até mesmo para a sobrevivência, dependendo da situação. Dentre os diferentes conteúdos que o áudio pode conter, a voz falada e a música possuem notável aplicabilidade e, consequentemente, são de grande interesse para a pesquisa acadêmica.

Entre as linguagens e pacotes de software disponíveis para processamento de áudio, há aquelas em que a representação do áudio é dada através de amostras, algo que pode ser visto como equivalente do áudio ao *pixel* na computação gráfica. Há também aquelas que representam o áudio como um conjunto de símbolos que denotam compactamente um procedimento para a construção do áudio, tal como instantes de ocorrência, durações e valores quantizados de frequências sobre modelos tibrísticos enumerados, algo que pode ser visto como análogo à representação vetorial de uma imagem na computação gráfica, a qual utiliza primitivas geométricas (circunferências, polígonos, retas, etc.), interpolações (e.g. curvas de Beziér), entre outros. Porém, poucos são os softwares que possibilitam ambos os tipos de representação ou uma conversão entre elas para a obtenção do segundo formato a partir do primeiro.

Símbolos musicais em uma partitura ou em outro sistema de notação musical possuem algum significado, e de certa forma o significado de cada símbolo ou conjunto de símbolos está associado a algum som factual ou percebido. Talvez esse som ou percepto não seja totalmente determinado pela instrução, mas certamente existe algum vínculo entre esse símbolo e o som correspondente, caso contrário a própria existência de uma notação musical não faria sentido. É factível criar um sistema, em software ou até robôs mecânicos, para a síntese de áudio a partir de notação simbólica[1].

O interesse do presente trabalho está centrado no processamento de sinais de áudio, desde o entendimento do que representa uma amostra de áudio até a avaliação de premissas que diferentes algoritmos utilizam para obter informações a partir do áudio que sejam correlatas a aspectos perceptuais humanos. Uma das motivações incluem a busca por aspectos fundamentais à criação de uma audição sintética, no sentido de possibilitar uma representação objetiva e compacta do som munida de significado para o ouvinte humano. Este trabalho se limita a avaliar as possibilidades de representação de áudio, e a criar um pacote de *sofware* público que possibilite e facilite não apenas a pesquisa como também a aplicação de processamento de áudio em tempo real, incluindo e permitindo a elaboração de uma variedade de algoritmos que descrevam o processamento diretamente por amostras, por blocos regulares de amostras, além de hibridismos que incluem a possibilidade de uso de blocos de tamanhos diferentes para partes diferentes do algoritmo. A motivação para a criação de tal pacote foi o de coletar informações do áudio de maneira clara e expressiva para o desenvolvedor, evitando a necessidade de se aprofundar em detalhes da organização interna dos dados tal como ocorre em linguagens de tipagem fraca, mas também sem remover do desenvolvedor essa possibilidade. Visando o possível uso do pacote para análise de música, tornou-se necessário ilustrar a possibilidade de coletar informações do áudio que trouxessem significado para músicos e engenheiros de áudio, o que fez com que o pacote incluísse alguns componentes de MIR (*Music Information Retrieval*, ou coleta de informações musicais).

No presente capítulo, encontram-se algumas informações gerais sobre o conteúdo deste trabalho. Na seção 1.1 está um resumo de informações básicas das diversas áreas do conhecimento envolvidas. Na seção

[1] A banda CompressorHead é um exemplo, fazendo jus a seu slogan de banda de metal mais pesada do mundo. http://compressorheadband.com, acesso em 2013-01-08.

1.2, os objetivos do presente trabalho são detalhados, seguido por suas justificativas na seção 1.3, que inclui potenciais aplicações diretas e indiretas dos resultados obtidos com o trabalho. A seção 1.4 descreve o ambiente de trabalho utilizado no desenvolvimento, organização e publicação do pacote AudioLazy e de sua documentação. As contribuições que este trabalho trouxe encontram-se descritas na seção 1.5. Finalmente, a estrutura do texto é explicada em 1.6, sumarizando os capítulos que seguem.

1.1 Considerações preliminares

As considerações presentes nesta seção podem ser encontradas em Roederer [Roe98], Moore [Moo90], Sedra e Smith [SS00, cap. 1] Steiglitz [Ste96], e/ou Oppenheim [OSB99].

Fisicamente, o som é uma onda mecânica longitudinal caracterizada pela variação de pressão. Perceptualmente o som é visto como uma representação mental subjetiva das informações recebidas pelos órgãos sensoriais presentes em nosso ouvido. A fim de evitar essa dicotomia, sempre que for falado de "som", o texto se refere à representação física.

O som cuja pressão oscila mais rápido do que podemos ouvir, isto é, com frequência além de um certo limiar, é chamado de ultrassom. O som que oscila mais lentamente do que podemos ouvir é chamado de infrassom. É comum a utilização dos valores 20Hz e 20kHz como limiares da audição, embora esses valores não sejam iguais para todas as pessoas. Adicionalmente, uma única pessoa pode ter diferentes limiares para diferentes intensidades de um mesmo som.

1.1.1 DSP ou Processamento de sinais digitais

DSP (*Digital Signal Processing*, processamento digital de sinais ou processamento de sinais digitais) é uma área do conhecimento humano que envolve a manipulação da representação codificada da informação para uso prático em problemas de diversificadas áreas de aplicação, cada uma com naturezas ou realizações físicas específicas, incluindo áudio (música, codificação de voz, ultrassom, etc.), informação sísmica e climática (geologia, meteorologia), vídeo (cinema, televisão), entre muitas outras possibilidades. O nome também pode ser utilizado para fazer referência a dispositivos processadores de sinais digitais.

Um sinal é uma quantidade que varia em alguma dimensão tal como o tempo ou o espaço. Sinais analógicos são sinais que representam, por analogia, uma grandeza física através de outra grandeza física. Um exemplo típico é o som representado através da variação de tensão em um fio, de forma análoga à variação de pressão do ar. Os componentes que realizam a conversão de uma grandeza física em outra, como no exemplo do som em eletricidade, são chamados de transdutores, e esse processo de conversão é chamado de transdução. Sinais analógicos, embora conhecidos apenas através de uma quantidade finita de medidas discretas empiricamente coletadas, costumam ser matematicamente representados através de funções cujo domínio e contradomínio é o conjunto $\mathbb{R}$ dos números reais sob premissas envolvendo a continuidade do sinal e a interpolação dos valores coletados, processo cujo significado físico ou filosófico não faz parte deste trabalho se aprofundar.

Sinais podem ser descritos por sequências de valores, e cada elemento de uma sequência que representa um sinal é chamado de amostra. Sinais discretizados no tempo são sequências de valores (amostras) que representam uma variação no tempo de alguma grandeza física, e normalmente são obtidos a partir de um processo de amostragem de outro sinal, processo este caracterizado pela coleta de valores de um sinal em instantes de tempo específicos, como exemplificado na figura 1.1. A amostragem não é restrita a funções em $\mathbb{R}$ como ocorre no exemplo da figura 1.1, podendo ser realizada a partir de um sinal discretizado no tempo. É dado o nome de subamostragem ao processo de amostragem aplicado a sequências conhecidas ou a sinais previamente amostrados no caso em que o processo se refere a um corte ou diminuição da quantidade de amostras disponíveis.

Sinais digitais são sequências codificadas de valores discretos. Em contraste com a necessidade de uma forma física e unidade de grandeza definida dos sinais analógicos, sinais digitais referem-se à abstração realizada através da codificação, sendo representados por sequências de símbolos ou amostras quantizadas. As amostras são ditas quantizadas quando elas são representadas por números ou símbolos representando números dentro de um conjunto finito de valores. Considerando a sequência como representação de um sinal no tempo, a duração em segundos de cada amostra é uma metainformação associada à amostra ou à sequência na

qual a amostra está incluída, sendo essa duração normalmente um valor constante T_s chamado período de amostragem. Um mesmo sinal digital pode ser utilizado para a realização ou representação de diferentes sinais analógicos, variando-se apenas as metainformações associadas à codificação tais como o período de amostragem ou a unidade de grandeza de cada amostra. Em termos práticos, os símbolos usados para representar as amostras são números armazenados na memória de um computador, inteiros ou em ponto flutuante.

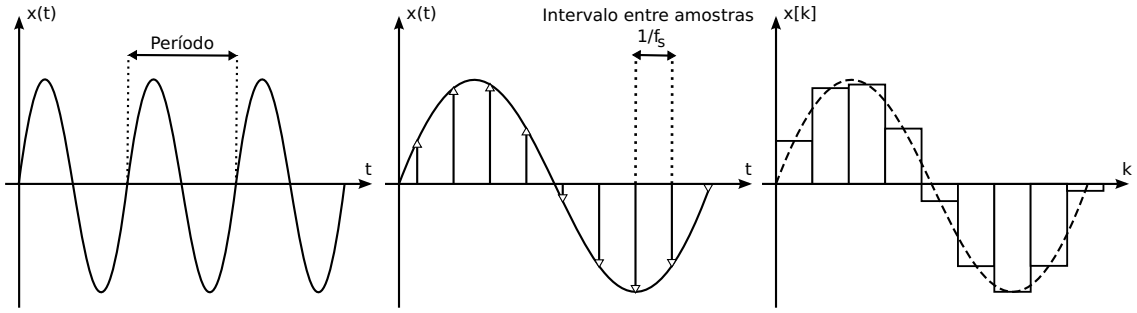


Figura 1.1: *Amostragem de um período de uma função contínua. Da esquerda para a direita, temos:* (1) *Uma função contínua periódica com seu período identificado;* (2) *Ilustração da amostragem de um período da função, com taxa de amostragem igual a f_s;* (3) *Um sinal digital obtido através de uma amostragem.*

Em todo o trabalho, as gravações digitais de áudio podem ser vistas como segmentos de um sinal digital armazenados ou persistentes em algum tipo de mídia ou memória. As gravações digitais podem ser convertidas em sons quantas vezes forem desejadas e não se desgastam com o tempo. Em termos de notação, $x(t)$ representa o valor de uma função x para um dado valor de $t \in \mathbb{R}$ considerado um instante de tempo, e $x[n]$ representa o valor da n-ésima amostra de uma sequência ou sinal digital x. Uma função $x(t)$ é dita periódica de período $\tau > 0$ caso satisfaça $x(t) = x(t + \tau)$, $\forall x \in \mathbb{R}$, como ilustrado na figura 1.1. Analogamente, sequências são ditas periódicas de período $N > 0$ quando mantém a relação $x[n] = x[n + N]$, $\forall n \in \mathbb{Z}$. Uma função real periódica de período τ, ao ser amostrada com taxa de amostragem $f_s = 1/T_s$, não necessariamente preservará o mesmo período τ, pois τ pode não ser um múltiplo inteiro de T_s.

Série de Fourier é o nome dado à decomposição de uma função periódica de período τ em uma soma de senoides cujas frequências são múltiplas de $1/\tau$. Cada uma dessas senoides é chamada de harmônico do sinal. Uma ampliação da ideia consiste em dizer que toda função contínua pode ser representada no domínio da frequência por uma transformada de Fourier[2]. Trata-se de um operador linear $\mathcal{F}(\cdot)$ que satisfaz a equação:

$$\mathcal{F}[x(t)] \;=\; X(\omega) \;=\; \int_{-\infty}^{\infty} x(t)\, e^{-j\omega t}\, dt \tag{1.1}$$

em que $j \in \mathbb{C}$ é a unidade imaginária e $\omega = 2\pi f$. Caso essa função seja interpretada como modelo ou aproximação de um sinal analógico, a unidade da frequência f será hertz quando t estiver em segundos.

O teorema da amostragem de Nyquist-Shannon diz que uma função em $\mathbb{R}$ que seja completamente definida por uma soma de componentes senoidais (integral da equação 1.1) na faixa de frequências $[0; f_{max}]$ é totalmente determinada por uma sequência de amostras de tal função se a frequência de amostragem f_s satisfizer:

$$f_{Ny} = \frac{f_s}{2} > f_{max} \tag{1.2}$$

em que f_{Ny} é conhecida como frequência de Nyquist, e indica o limite de frequência representável pelo sinal discretizado no tempo[3].

[2] Descontinuidades também são representáveis, embora o valor no ponto de descontinuidade seja considerado como o valor médio entre os extremos da descontinuidade, não o valor no ponto.

[3] A senoide com a frequência de Nyquist é amostrada duas vezes por período, e as duas amostras resultantes em cada período podem coincidir precisamente nos valores de fase 0 e π, tornando-as nulas, e portanto indiscerníveis com relação à ausência de tal componente frequencial. Caso esse alinhamento não ocorra, é possível identificar a presença da senoide e um valor mínimo para sua amplitude, mas não é possível assegurar sua fase e amplitude exatas.

1.1.2 Música

Nós, humanos, recebemos continuamente informações de áudio. John Cage aproveitou artisticamente e filosoficamente a ideia de que não podemos interromper esse fluxo, alegando a ausência do que seria o silêncio absoluto. Essa ausência pode ser justificada pelo fato de que a partir de um certo nível sonoro, quanto mais silencioso for um ambiente, melhor escutamos o metabolismo de nosso próprio corpo. Apesar disso, quando lidamos com áudio, em geral existe a preocupação com um início e um término, e com a existência de algum conteúdo que possa ser distinguido de um ruído ambiente ininteligível.

Música é uma forma de manifestação artística que se utiliza do som como meio material para sua expressão. Desde a antiguidade, muitas culturas demonstraram manifestações musicais. As notações musicais da Grécia antiga e os instrumentos com idade estimada de dezenas de milhares de anos antes da civilização grega antiga sugerem a importância que essa arte possui para a humanidade. Além disso, há evidências de que pássaros, baleias e outros animais também possuem, na música, uma forma de expressão [Rod09].

Há diferentes classificações para a textura musical; entre elas está a música monódica, que envolve apenas uma linha sequencial de eventos musicais, como ocorre em uma melodia sendo cantada *a capella*[4]. Outra classificação dada à textura pode ser vista como um complemento a essa ideia, na situação em que há pelo menos dois eventos musicais sendo realizados em simultâneo. A essa textura damos o nome de polifônica. Tais classificações de alguma forma se relacionam com a percepção dos eventos musicais, visto que elas não necessariamente correspondem ao número de fontes sonoras factuais. A reprodução de uma obra monódica por dois alto-falantes não faz dessa obra uma polifonia, assim como a reprodução de uma peça polifônica por um único alto-falante não faz da peça uma monodia. Entretanto as fontes sonoras podem ser as que geraram uma gravação que foi reproduzida por um alto-falante, e nesse caso é possível falar de simultaneidade mesmo na existência de, aparentemente, uma única fonte sonora, sem precisar recorrer a uma segmentação perceptual. É comum a palavra "polifônica" estar associada à música contrapontística do período renascentista, enfatizando não apenas a simultaneidade como também a independência perceptual entre os sons, nesse caso chamados de "vozes". Mantenho o nome "contrapontística" para representar a simultaneidade de vozes, e utilizo o nome "polifônica" para representar a simultaneidade de sons. É possível notar nessa descrição e definição de nomes que a organização do som é realizada como um conjunto de sequências de eventos no tempo, com nomes específicos para os diferentes casos de simultaneidade. O elemento de destaque é a capacidade de decomposição ou análise de áudio por seres humanos, embora recebam fisicamente como informação apenas a mistura. Existem outros nomes de texturas (homofônica, heterofônica, etc.), mas um maior detalhamento é desnecessário.

Altura (*pitch*) é um atributo de sensação, de certa forma associado à percepção de que um som está agudo ou grave. Plack e Oxenham [PO05] analisam essa definição e dizem que ela se confunde com a definição de brilho. Brilho pode ser visto como o quanto um som é claro ou escuro, em um sentido timbrístico. É realmente difícil separar uma palavra da outra em uma definição clara, e talvez deixar junto um exemplo de cada um seja suficientemente claro para ser utilizado como definição. Um modelo comumente utilizado para caracterizar a altura de sons monofônicos é a frequência fundamental do som. Esse modelo se justifica pelo fato de que uma soma de sons senoidais audíveis de frequências linearmente espaçadas como uma série harmônica são percebidos com uma mesma altura, entretanto mudando seu brilho e outras características acerca de seu timbre ou qualidade do som. Tessitura é o nome dado a uma faixa de alturas, conceito normalmente associado à capacidade de um dado instrumento ou voz.

Volume sonoro percebido (*loudness*) é o atributo de sensação que nos permite ordenar os sons em uma escala do silêncio ao estrondo. Em música, dinâmica (*dynamics*) também costuma se referir à intensidade perceptual de um som. Em alguns casos (e.g. MIDI), a palavra *velocity* é associada à intensidade por conta de instrumentos que, como ocorre com as teclas de um piano, produzem um som mais intenso quando a utilização de seus mecanismos é realizada com maior velocidade. Há situações em que o logaritmo da amplitude de variação de pressão em um som pode ser usado como um correlato físico do volume sonoro percebido, embora esse seja um modelo bastante simplificado.

Escala temperada é um conjunto de sons cujas frequências fundamentais estão uniformemente espaçadas

[4] Nome dado à música vocal sem acompanhamento instrumental. A tradução literal seria "como na capela", por associação à forma como a igreja católica realizava peças sacras nesse ambiente.

em uma escala logarítmica. Normalmente as teclas de um piano referem-se a uma escala temperada de 12 sons por oitava, fazendo com que o intervalo entre duas notas consecutivas seja de $\Delta = \sqrt[12]{2}$, isto é, uma nota associada à frequência f tem como vizinhos as notas associadas às frequências f/Δ e $f\Delta$.

As alturas em notas musicais foram descritas durante o trabalho utilizando o padrão americano de nomes, fixando A4 como 440 Hz. No sistema americano, as notas seguem as letras

```
[C, D, E, F, G, A, B]
```

ciclicamente, correspondentes ao que também é chamado de

```
[dó, ré, mi, fá, sol, lá, si].
```

A oitava do dó central é a de número 4, mas a referência adotada é sempre o A4. Uma outra montagem da escala de alturas é

```
[C # D # E F # G # A # B],
```

dessa vez incluindo 12 notas, em que o símbolo "#" representa uma altura intermediária, e essa escala é dada em intervalos de semitom. Ao passar o ciclo de nomes, a numeração de oitava é incrementada/decrementada de acordo, de maneira que o maior valor da oitava representa o som mais agudo. As alturas podem conter alterações, em que o sustenido ("#") representa meio tom acima e o bemol representa meio tom abaixo.

Durante os exemplos envolvendo altura (e.g., resultado da função *freq2str* da AudioLazy), o desvio em porcentagem de semitons é justaposto ao nome da altura, quando necessário.

Existem outros padrões para as alturas, como o padrão alemão que distingue entre o si natural e o bemol através do uso de H e G, ou o padrão europeu de numeração de oitavas que define A3 como 440 Hz e não inclui a oitava "zero", saltando direto de A1 para G-1.

1.2 Objetivos

Os objetivos do projeto evoluíram durante o decorrer da pesquisa, iniciando com a motivação em se chegar a um transcritor musical ideal. Objetivar tal transcritor talvez não seja utópico, mas certamente supera as expectativas de uma pesquisa de mestrado, não apenas com relação à quantidade de tarefas que precisariam ser realizadas como também com relação à dificuldade e distância entre a atualidade e o resultado de tal transcritor, a qual pode ser estimada através da quantidade de pesquisadores que estão envolvidos com MIR e uma comparação com os resultados presentes de seus sistemas. Dessa forma, entre os objetivos do trabalho está a avaliação de algoritmos de MIR que possam ser qualificados com relação à sua capacidade de transcrição, assim como a proposta de alternativas aos algoritmos que mantenham sua viabilidade como um próximo passo curto (*baby step*) ao desenvolvimento da pesquisa nessa área.

Para esse fim, um dos elementos que se tornou evidente rapidamente é o fato de que a maioria dos recursos disponíveis para o desenvolvimento de software para MIR ou se restringe à simulação, ou está preparado para o processamento de áudio em lotes, ou possui limitações práticas que dificultam o desenvolvimento de algoritmos que não sejam baseados no processamento em bloco, ou então mantém grande liberdade para o desenvolvimento às custas de uma sintaxe e verborragia que dificultam sobremaneira o desenvolvimento voltado ao processamento de áudio. Sob essas circunstâncias, esta pesquisa objetivou a elaboração de uma nova DSL (*Domain Specific Language*), voltada ao processamento de áudio, mas com base em recursos prontos de uma linguagem de uso geral, de forma a evitar que a especificidade da área impedisse seu uso com aplicações em geral.

Resumidamente, os objetivos do presente trabalho incluem:

- Desenvolver um pacote de software que auxilie o processamento de áudio em geral, visando sua possível utilização em análise, síntese, na elaboração de novos algoritmos de MIR, e na aplicação prática em tempo real;

- Avaliar e propor alternativas a algoritmos de MIR atuais a fim de direcioná-los à resolução do problema da transcrição.

O primeiro item entre os objetivos se refere ao pacote AudioLazy, que traz o nome deste trabalho. AudioLazy é um pacote de *software* desenvolvido em puro Python que seguiu seu desenvolvimento com os seguintes objetivos, os quais podem ser vistos como requisitos do projeto:

- Capacidade de funcionamento em tempo real, algo detalhado na seção 2.2 como um sistema que funcione com latência máxima de 50 milissegundos;
- Comunicação com o *hardware* para gravação e reprodução de áudio;
- Modelagem de parte do sistema auditivo humano, aproximando o sistema desenvolvido a um sistema CASA (Computational Auditory Scene Analysis);
- Modularidade, no sentido de trazer ao desenvolvedor uma gama de componentes com diferentes possibilidades de conexão e interfaces compatíveis que não impeçam o acesso a informação;
- Expressividade, no sentido de evitar a distância entre a linguagem de programação e a linguagem da engenharia que lida com áudio;
- Evitar que o programador tenha de se preocupar com a necessidade de estruturas intermediárias tais como índices de blocos, tamanhos, conversões entre unidades, mas sem impedir o acesso às mesmas quando requisitadas;
- Multi-plataforma: minimamente, o sistema deve funcionar tanto no Windows como no Linux;
- Realização de testes, com um mínimo de 50% de cobertura de código;
- Biblioteca de recursos básicos utilizados em DSP, tais como filtros LTI (*Linear Time Invariant*, lineares e invariantes no tempo), derivadas, processos não-lineares, etc.;
- Representação MIDI;
- Algoritmos básicos de síntese de áudio, tais como consulta à tabela (*table lookup*) e por modulação em frequência (FM, *Frequency Modulation*);
- Integração com outras estruturas de dados e outros pacotes, sobretudo com relação à álgebra linear, obtenção de gráficos (*plots*), e estruturas padrões da linguagem utilizada;
- Documentação completa do código, separada por módulos, incluindo a descrição dos componentes utilizáveis do pacote, um tutorial de introdução, exemplos e motivação;

Este trabalho incluiu a formulação de um enunciado claro para o problema da transcrição, explicitando a influência do sujeito ouvinte no processo, como o resultado do processo poderia ser representado e o que essa representação significa, pragmaticamente. Esse resultado encontra-se no primeiro apêndice do trabalho e se refere a apenas um aspecto inicial de motivação ao trabalho, que demonstrou a necessidade de um sistema de processamento de áudio com as características apresentadas.

1.3 Justificativas e motivações

As subseções que seguem enunciam algumas das possíveis justificativas ao projeto, à pesquisa e à implementação realizados. Na seção 1.3.1 consta uma justificativa à criação de um novo pacote de *software*, mas vale destacar que o capítulo 2 traz um maior detalhamento quanto ao desenvolvimento da AudioLazy, incluindo critérios que levaram à sua concepção e uma comparação com sistemas existentes.

Há também, na seção 1.3.2, uma lista de aplicações possíveis para sistemas transcritores ou, mais genericamente, sistemas que coletam informação musical a partir do áudio, dependente ou independentemente da necessidade de uma representação completa do áudio de entrada ao sistema. No primeiro apêndice do trabalho encontra-se uma ilustração do significado do processo de transcrição.

1.3.1 Demanda por pacotes de sofware

A principal justificativa que pode ser dada à realização de uma tarefa de longo prazo é a demanda, seja esta a representação de uma vontade ou necessidade individual, ou então a emergência coletiva, formada através da vontade e necessidade de diversos indivíduos. No dia 2013-02-04, um e-mail foi enviado à lista do IRCAM (*Institut de Recherche et Coordination Acoustique/Musique*) por Eric J. Humphrey[5]:

> "[...], eu gostaria de propor uma leve modificação na *ISMIR Conference* que pode beneficiar significativamente a nós e à nossa pesquisa: simplesmente fornecer aos autores a opção de publicar código fonte como parte da publicação.
>
> Por que vale a pena considerar isso? Adicional à observação de que outras comunidades de pesquisa têm feito por algum tempo (para publicar na *Science*, por exemplo, deve-se tornar o código fonte publicamente disponível[6]), as vantagens do *software* aberto para fins científicos são enormes[7]. Para ser breve, há três que valem a pena mencionar aqui:
>
> - **Código fonte serve de ponto de partida para trabalhos futuros**. Há muitos impactos positivos [...], desde possibilitar resultados reprodutíveis até adequadas comparações com novos algoritmos. Adicionalmente, novos métodos e ideias são bem mais sustentáveis se for fácil para outros construí-las e continuá-las.
> - **Simplesmente escrever código com a ideia de que outro humano pode na verdade o ler inerentemente o torna o código, assim como você, melhor**. [...] Seu código irá invariavelmente ser mais estável, legível e passível de manutenção. [...] código de pesquisa não é código de produção, [...], mas isso não é uma desculpa para escrever *spaghetti*[8]. Isso torna mais fácil continuar o trabalho que você já fez, [continuado] por você ou por outros.
> - **Não é sempre possível reproduzir com exatidão a pesquisa ou os resultados de uma publicação**. Por vezes, o único texto que pode fazer isso é o próprio código. Limitações [do número] de páginas restringem o nível de detalhe usado para descrever um sistema ou algoritmo, [e] pequenas diferenças de implementação pode mudar drasticamente o desempenho, e equações, assim como suas explicações, certamente não são imunes a erros de digitação. Acople isso com o desafio de expressar sistemas complexos em uma linguagem clara e concisa, e não é difícil perceber por que código fonte pode ser bastante útil.
>
> Dito isso, encorajar e dar suporte ao compartilhamento de código fonte no próximo ISMIR é uma mudança relativamente fácil de adotar, e agora, meses à frente da data limite de submissão, é o perfeito instante para considerar isso. O que poderia acontecer para tornar isso realidade?
>
> 1. **Planejamento com antecedência**. Compartilhamento de código é substancialmente mais prazeiroso para todos envolvidos se você pensar que pode desde o início.
> 2. **Fornecer aos autores a opção de enviar código fonte com o artigo**. [...].
> 3. **Reconhecer código fonte publicado**. Por exemplo, artigos com o código correspondente pode ser marcado na publicação com uma "estrela de código fonte" próxima ao título. A questão é o bom karma[9], pessoal.

[5] Original em inglês. A formatação foi mantida, a menos dos itálicos. Passagens entre colchetes são inserções feitas ao texto para tornar mais clara a tradução. As notas de rodapé estão no original mas foram modificadas para incluir explicitamente os endereços e manter a padronização de formatação deste trabalho. Símbolos "*[...]*" denotam cortes realizados à mensagem original.

[6] Brown. Journal pubs. *Data and software release policies*. Datado de 2012-07-14. http://ivory.idyll.org/blog/journal-data-policies.html

[7] Prlić A, Procter JB. *Ten Simple Rules for the Open Development of Scientific Software*. PLoS Computational Biology 2012, 8(12). http://www.ploscompbiol.org/article/info:doi/10.1371/journal.pcbi.1002802

[8] Nota sobre a tradução: esta expressão é utilizada sobretudo por evangelistas da programação estruturada contrários ao uso de saltos incondicionais como maneira pejorativa de expressar desorganização.

[9] Nota sobre a tradução: o "karma" é comumente o nome dado em fóruns à pontuação de uma conta, relativa à quantidade e importância de suas mensagens, com base em votos da comunidade.

4. **Tornar o código fonte disponível como parte da publicação**. Pense nisso como um foto de uma pesquisa, para ser mantida como um todo em um lugar, *online* e na publicação eletrônica. Nós não fazemos controle de versão nos artigos (por enquanto), então não precisamos realizar controle de versão no código (por enquanto). Uma coisa de cada vez[10].

Como comentário final, é importante lembrar que resultados e observações publicados no final das contas depende do código que os produziu; portanto, se um artigo é suficientemente bom para ser impresso, então também é o código em que ele é baseado. [...].

Desejando o melhor, Eric J. Humphrey, doutorando na MARL (*Music and Audio Research Lab*) da NYU (*New York University*) "

Pode-se observar que o julgamento de valor de Humphrey contém excessos, mas há informações e referências importantes presentes em sua mensagem e na discussão por ele iniciada. É certo que a reprodutibilidade é necessária em qualquer trabalho científico, e que esta é facilitada quando existe o acesso aos mecanismos que geraram os dados utilizados na pesquisa, mas isso não se aplica somente ao código-fonte utilizado como também ao *corpus* de dados. Brown[11] coleta a informação de requisitos à publicação em 6 revistas, incluindo Science, Nature e PNAS. Todas as revistas obviamente requisitam a reprodutibilidade do trabalho para a publicação, mas com diferentes indicações quanto à necessidade de acesso aos dados e materiais, por vezes com diferenças quanto à disponibilidade para revisores e para leitores em geral. A Science de fato exige que o trabalho inclua a disponibilidade do código-fonte a qualquer leitor. PNAS exige que dados não públicos não podem ser utilizados como justificativas nos trabalhos, e inclui menção explícita a códigos-fonte. Apesar de nem todas as revistas exigirem o código-fonte, tal publicação é um mínimo necessário para que outros autores possam dar continuidade ao trabalho sem a necessidade de tempo gasto com a reimplementação de algo que já foi feito por outra pessoa. Tal reimplementação talvez seja desejada pelo leitor de uma dada publicação, mas existe uma diferença significativa entre a opção de reimplementação e a ausência de uma implementação pública disponível.

Não necessariamente há uma equivalência entre qualidade de código e qualidade de publicação que utiliza o código, embora idealmente uma publicação não devesse ser dependente de um código de baixa qualidade, no sentido de que minimamente o autor e os revisores possuem responsabilidades quanto à validade dos resultados obtidos. Mas há critérios diferentes para a qualidade do código, tais como requisitos quanto à modularidade e à expressividade, que podem ter sido completamente deixados de lado em uma pesquisa direcionada a uma avaliação pontual de um modelo específico, de forma que o significado de "qualidade de *software*" não é suficientemente claro, ou objetivo, na ausência de critérios explícitos.

Brown também faz parte dos 12 autores que sintetizaram algumas práticas para o desenvolvimento de *software* para computação aplicada à ciência[12], as quais enumero a seguir:

1. **Escreva programas para pessoas, não computadores**;
2. **Automatize tarefas repetitivas**;
3. **Use computadores para armazenar históricos**;
4. **Faça mudanças incrementais**;
5. **Use um sistema de controle de versão**;

[10] Nota sobre a tradução: originalmente, a expressão usada foi "*Baby steps*", a qual pressupõe uma intenção do autor em levar à comunidade científica o controle de versão não apenas no software, mas também com relação aos artigos. Essa expressão é utilizada no desenvolvimento orientado a testes, conforme melhor descrito na seção 2.3.7.

[11] URL fornecida por Humphrey, último acesso dia 2012-02-14.

[12] Artigo arXiv:1210.0530v3 [cs.MS]:
Wilson, Greg; Aruliah, D. A.; Brown, C. Titus et al. *Best Practices for Scientific Computing*. arXiv.org, Cornell University Library, 2012, v3.
Disponível em:
http://arxiv.org/abs/1210.0530

6. **Não repita a si próprio (ou outros)**: Conhecido como princípio DRY (*Don't Repeat Yourself*), válido tanto para o código como para valores fornecidos em meio a um código (e.g. constantes). O desenvolvimento da AudioLazy visou aplicar o princípio ate mesmo em tarefas externas ao código central, tais como a parametrização de testes a geração automática da documentação nos diversos formatos a partir de uma única versão criada manualmente;

7. **Planeje incluindo a possibilidade de erros**:

   - Programação defensiva (uso de assertivas);
   - Escreva e execute testes;
   - Use uma variedade de oráculos;
   - Transforme *bugs* em casos de teste;
   - Utilize um depurador (*debugger*) simbólico;

   Alguns desses itens contém exemplos em um pseudo-código similar ao Python, e incluem a descrição de um processo de desenvolvimento orientado a testes, no qual o último item deixa de ser uma necessidade. O artigo não coloca o TDD (*Test-Driven Development*, ou desenvolvimento orientado a testes) como uma necessidade, mas expõe um interesse em tal prática;

8. **Otimize software apenas após este funcionar corretamente**;

9. **Documente o projeto e a intenção, não a mecânica**: Refere-se a evitar colocar como comentários algo que está explícito no código. Uma recomendação feita é a de inclusão da documentação de um código dentro do próprio código, além de citar o uso de pacotes de geração de documentação como o Sphinx;

10. **Colabore**: Uma defesa à prática do compartilhamento social de *software*;

O desenvolvimento da AudioLazy seguiu praticamente todos os itens rigorosamente, incluindo os critérios acerca da escolha de nomes e outros aspectos que visam a expressividade e a manutenibilidade do código, embora a leitura do artigo tenha sido realizada posteriormente à criação e desenvolvimento do pacote.

Nitidamente o julgamento de valor de Humphrey contém excessos, mas há informações e referências importantes. A maior importância da mensagem de Humphrey não é a própria mensagem, apesar de pública, mas as justificativas e referências fornecidas pelo próprio, além das mensagens que deram continuidade à discussão, que possibilitaram, por exemplo, encontrar as dez diretrizes de desenvolvimento supracitadas. Dadas as iniciais de Humphrey, e o explícito envolvimento com a MARL, foi simples identificar que o autor é aquele identificado como "*ejh333*" no BitBucket[13], e responsável pelo repositório público da MARLib[14]. Dessa forma, o próprio Humphrey está envolvido como desenvolvedor de um pacote em Python para processamento de áudio utilizando o NumPy, o que expõe um viés do autor no sentido de valorizar o fruto de seu esforço. A mensagem do Eric teve respostas no grupo, entre as quais pode-se destacar duas propostas sociais de auxílio ao desenvolvimento para cientistas:

- **Sustainable Software for Audio and Music Research**[15]: O *site* possui artigos, vídeos e slides online, incluindo uma diversidade de conteúdos. Há um artigo que enfatiza a diferença entre o Mercurial e o git como sistemas de controle de versão. Sobre material recente, estão incluídos os tutoriais realizados

[13] Sistema de hospedagem de repositórios, URL: https://bitbucket.org

[14] Pacote voltado ao processamento de áudio em Python, sob licença LGPL, utilizando NumPy e SoX e com um total de 6 pessoas envolvidas com o desenvolvimento. Disponível em:
https://bitbucket.org/ejh333/marlib, último acesso dia 2013-02-18.
Relativo ao último acesso realizado, a última atualização publicada ocorreu dia 2012-11-15. O repositório foi criado dia 2010-10-19. O pacote está disponível no PyPI, onde o nome de Eric Humphrey está explícito como autor da MARLib. Endereço:
https://pypi.python.org/pypi/marlib, último acesso dia 2013-02-18.
Vale salientar que a última atualização no PyPI data de 2013-01-08, o que sugere que as últimas atualizações da MARLib não estão sendo colocadas no BitBucket, possivelmente pelo fato de que a quantidade de *downloads* não é informado no BitBucket.

[15] http://soundsoftware.ac.uk/, último acesso dia 2013-02-15.

no DAFx 2012 e no ISMIR 2012 quanto a testes de unidade, controle de versão, gerência de dados e código fonte aberto. São quatro os tópicos principais listados através do *link* "*Topics*", todos aplicados ao desenvolvimento de *softwares*:

  - Controle de versão;
  - Hospedagem do projeto;
  - Licenças;
  - Testes.

  Vale salientar que há uma indicação do ano de 2011 como início do projeto.

- **Software Carpentry**[16]: Projeto com a missão de ajudar pesquisadores a serem mais produtivos através do ensino de habilidades básicas de computação, incluindo a construção de programas, controle de versão, testes, uso da linha de comando e gerência de dados.

Independentemente da fragilidade do significado de expressões como "melhor" ou "pior" nas situações em que estão ausentes os critérios que dão significado aos julgamentos, o fato de existirem tais referências indicam a existência uma insatisfação por parte da comunidade acadêmica que faz pesquisa em processamento de áudio para uso em música, insatisfação esta com relação aos *softwares* existentes para uso científico, aparentemente não por conta da quantidade, mas por questões de qualidade.

1.3.2 Aplicações da transcrição e MIR

Muitas são as possíveis aplicações que podem ser feitas a partir de um sistema de transcrição musical ou de áudio em geral. Nesta seção, há uma lista de aplicações que são viabilizadas ou fortemente melhoradas na presença de bons algoritmos de MIR, isto é, de obtenção ou coleta de informações musicais, ou então de um bom sistema de transcrição. A maioria das aplicações aqui presentes refletem possíveis trabalhos futuros, por envolverem tarefas por vezes além do MIR ou da transcrição, embora utilizando-os como base.

Em geral os sistemas abaixo são factíveis com um sistema transcritor ideal, mas com requisitos que vão além do presente trabalho. Um requisito comum em alguns dos sistemas é o de resposta em tempo real, ou pelo menos uma resposta suficientemente rápida para trabalhar diretamente com os sinais digitais e não com suas gravações[17], o que justifica o desenvolvimento da AudioLazy como um pacote que não se limite à simulação aplicada em processamento de áudio.

Bancos de dados de áudio

Bancos de dados de arquivos MIDI já existem, assim como existem sistemas de busca voltados para tais arquivos. Algumas das vantagens que um bom sistema transcritor fornece aos sistemas de busca em bancos de dados musicais são:

- Permite que o banco de dados seja populado de forma ágil se comparado com a criação manual de arquivos MIDI;
- Aumentaria drasticamente o número de arquivos MIDI de música popular disponíveis;
- Permite a busca através de segmentos de áudio, por exemplo a busca de uma música através de uma melodia cantada pelo usuário[18];

[16] http://software-carpentry.org/, último acesso dia 2013-02-15.

[17] A rigor, mesmo que poucas amostras de processamento sejam necessárias para a devolução de cada amostra, essas amostras de entrada precisam ser gravadas em algum ponto na memória acessível pelo processador digital. A distinção se refere à utilização de um fluxo de amostras de entrada, em contraste com o uma entrada restrita a amostras armazenadas em memória desde o início do processamento do áudio, isto é, uma requisição de amostras de áudio de acordo com a necessidade em contraste com uma gravação existente mesmo antes da devolução do primeiro símbolo de saída, seja este outra amostra ou o resultado de uma agregação.

[18] Um exemplo existente desse tipo de recurso pode ser visto na MusiPedia: http://www.musipedia.org/, último acesso dia 2013-02-15.

- Classificação automática da música por estilo, por exemplo baseadas em timbres e padrões rítmicos.

  Indexação em bancos de dados contendo áudio não precisam se restringir à música. Isso ocorre na transcrição de voz em arquivos contendo áudio, que pode ser aplicada por exemplo em *corpora* de vídeo-aulas, a fim de facilitar o aluno ouvinte a encontrar o vídeo ou o segmento de um vídeo que contém uma dada informação.

Registro

Muitas músicas são o resultado de uma única performance, e não de uma intenção composicional expressa graficamente. Existem músicas que não utilizam registros gráficos, mas possuem espaços reservados à improvisação, em que nem sempre há algum registro daquilo que foi realizado. Atualmente essas performances podem ser registradas como gravações, mas alguém interessado em analisar o que foi feito teria de transcrever de forma artesanal, ou buscar uma transcrição feita por outra pessoa.

Uma utilidade diretamente associada à questão do registro que não é necessariamente uma vantagem do sistema resultante, mas ao menos uma possibilidade, é a defesa dos direitos autorais sobre o uso comercial, visto que esses registros facilitam a comparação de diferentes músicas, auxiliando na detecção de plágios, e possivelmente em outras questões jurídicas.

Auxílio ao aprendizado

Na música popular, normalmente é função de cada instrumentista de uma banda obter as informações sobre o que é a sua parte em uma música. Quando há mais de uma pessoa com o mesmo instrumento, é comum haver divisões combinadas por meio de expressões simples como a posição no tempo em que uma passagem ocorre ou a tessitura utilizada. Entretanto, há passagens musicais em que essas expressões não são tão simples de serem realizadas.

É muito comum que existam transcrições prontas na Internet, e que elas sejam utilizadas e procuradas pelos músicos. O motivo disso normalmente não está associado à incapacidade, e sim à falta de tempo. Felizmente muitas pessoas disponibilizam seu trabalho na Internet, mas por vários motivos como o tempo e a ausência de remuneração, é comum a transcrição não possuir muita precisão em algum aspecto, de forma que transcritores apenas façam algo aproximado, por vezes com erros na escrita que outro músico, ao tentar tocar, pode ter dificuldades até mesmo em reconhecer a música. Para músicos que desejam tocar mas que não possuem tempo para realizar uma transcrição, ou músicos que desejam tocar coisas extremamente difíceis de serem transcritas, a existência de um sistema transcritor é de grande ajuda.

É pouco comum que uma banda amadora leve ao palco um coral e/ou uma orquestra, no entanto há alguns recursos que podem buscar a maior proximidade com o som, e não com a instrumentação originalmente utilizada na gravação. Em certos estilos musicais, essa simulação é normalmente tarefa dos tecladistas, que se encarregam de representar em um sintetizador todos os instrumentos que não pertençam à banda. Em geral, a qualidade do timbre resultante faz parte do que caracteriza a qualidade dos tecladistas. Devido à multiplicidade de tarefas, o tecladista portador de uma boa transcrição pode enfatizar sua dedicação aos timbres de seu teclado, ao invés de gastar a maior parte do tempo identificando o que uma orquestra fez em uma gravação. Obviamente essas colocações são típicas da música sem um registro oficial em partitura ou com um registro incompleto.

Algo muito importante que este item deixa claro com o exemplo da orquestra tocada em um sintetizador é que uma transcrição não necessariamente precisa identificar exatamente o que gerou a música para ser considerada útil ou correta. A mesma ideia é vista no item sobre composição musical.

Composição musical

A tecnologia pode abrir novas oportunidades, e é função do artista saber aproveitá-las. Um sistema transcritor, se bem utilizado, pode ajudar o compositor a organizar suas ideias, auxiliando-o na identificação de algo que ele considere, por alguma razão estética pessoal, "especial" em um som.

Um sistema transcritor com a especificação dada no presente trabalho pode ser intencionalmente utilizado de forma a auxiliar na composição de música espectral. Um compositor pode fornecer um resultado desejado

para que o sistema transcritor se preocupe em realizá-lo com o instrumental fornecido, intencionalmente, diferente do utilizado como áudio de entrada. Como o sistema transcritor não realiza uma mudança de estilo, se a solução for viável, espera-se que o sistema realize uma adaptação entre os diferentes instrumentais.

Edição de áudio

Edição de áudio pode se tornar uma tarefa com muito mais recursos e simplicidade se em posse de uma transcrição detalhada do que está sendo feito, particularmente quando for possível realizar uma ressíntese ou regravação do áudio de uma ou mais fontes sonoras.

Um sistema CASA pode utilizar uma transcrição para auxiliar a separação do áudio nas suas diferentes fontes sonoras. Um editor de áudio associado a um sistema desses é o sonho de muitos dos que trabalham nessa área. Para esse caso, a remixagem e a modificação de obras anteriormente gravadas poderiam ser bem mais simples, aproveitando ao máximo o conteúdo já gravado, mesmo que as faixas originais da mixagem sejam desconhecidas.

Análise musical

O registro de músicas improvisadas e étnicas, se feitas com qualidade, podem facilitar a pesquisa na área de musicologia, que poderá acessar as informações de forma ágil. Além disso, sistemas podem ser automatizados a fim de extrair informações sobre a estrutura da música, verificando se uma obra segue uma estrutura sem a necessidade de verificar manualmente todas as possibilidades. Um exemplo disso é a verificação de segmentos de 12 notas diferentes em série, algo que pode auxiliar o analista experiente a acelerar o seu trabalho.

Sistemas musicais iterativos

Um sistema de transcrição musical automática em tempo real permitiria que um instrumento musical tivesse seu sinal analógico convertido em um protocolo de comunicação entre equipamentos musicais, assim como o MIDI. Sistemas de acompanhamento automático e algoritmos de improvisação poderiam ser estendidos para utilizar esse resultado.

Sistemas de avaliação

Uma possível utilização de um sistema transcritor é o auxílio aos avaliadores, fornecendo-lhes informações sobre estatísticas como o número de notas que não foram tocadas ou valores de desvios rítmicos.

Para concursos musicais e vestibulares, um sistema desse tipo poderia, além de auxiliar os avaliadores, fazer parte do trabalho de avaliação. Por exemplo, em concursos nos quais são muitos os candidatos e se torna inviável manter uma banca de especialistas o tempo todo, seria de grande auxílio que uma filtragem inicial fosse realizada sem a necessidade da intervenção de especialistas. Adicionalmente, tais sistemas poderiam ser úteis, por exemplo, para os estudos de músicos autodidatas.

Pensando em uma aplicabilidade mais direta, atualmente os aparelhos de karaokê/videokê fornecem uma avaliação para o cantor. Com sistemas transcritores melhorados, essa avaliação poderia ser refinada, fornecendo informações precisas sobre a avaliação a aqueles que desejarem.

Compressão de arquivos de áudio

Arquivos de áudio são representados através de um sistema de codificação. Durante a conversão para muitas dessas codificações, podem haver perdas na qualidade áudio, e é comum essas perdas possuírem alguma justificativa que venha da psicoacústica. Em formatos de vídeo, algo similar ocorre, fazendo com que dois arquivos de mesmo tamanho com codificações diferentes possam ter qualidades diferentes, em função desse conhecimento sobre o que é mais relevante para a percepção. A remoção dos dados mais difíceis de perceber permite que essa codificação, mesmo com perdas, tenha menos perdas efetivamente percebidas do que a outra codificação. O conhecimento da psicoacústica fornece informações sobre como fazer isso, mas é possível que uma transcrição musical, por fornecer uma informação objetiva sobre a reconstrução do som, possibilite a

diminuição do tamanho do arquivo resultante. Um exemplo disso é a detecção de repetições, em que pequenos blocos podem ser responsáveis por uma música inteira através de suas combinações, possivelmente havendo simultaneidades nessas combinações. Para uma compressão sem perdas, talvez a quantidade de informações que a notação simbólica necessite torne o resultado muito maior do que o próprio áudio original. Por essas razões, a busca de sistemas de codificação baseado em transcrições é uma área de pesquisa potencialmente relevante.

1.4 Ambiente de trabalho

O pacote AudioLazy, software resultante deste trabalho, foi desenvolvido, testado e, para assegurar o funcionamento em processamento em tempo real, teve seu desempenho e latência avaliados. Mais de uma máquina foi utilizada para a realização do trabalho, mas há uma com particular destaque, por ter sido utilizada durante os em testes de desempenho e latência. Trata-se de um laptop Sager (a rigor, um Clevo P170EM) com processador Intel Core i7-3610QM (2.3GHz, 6MB de cache L3, suporte a 64 bits), 16GB de RAM (1600 MHz) em dois canais, com dispositivo de áudio *onboard* Intel "Panther Point High Definition Audio Controller", utilizando o Linux MINT (baseado em Ubuntu), kernel 3.2.0-37-generic (x86_64).

Testes gerais também foram realizados em outras máquinas, mantendo sua funcionalidade no Windows XP e Windows 7, sistemas operacionais da Microsoft, além de mantida sua funcionalidade em outras distribuições de Linux (testado com o Debian). Dado o desenvolvimento em Python puro, é plausível que o software funciona em todas as diversas plataformas as quais possuem um interpretador Python na sintaxe 2.x, embora seja necessário uma versão otimizada do Python para tal.

A versão do Python utilizada foi sempre a 2.7.3. Diferentes versões do NumPy (1.5.0, 1.6.1 e 1.6.2) e do MatPlotLib (1.0.1 e 1.2.0) foram utilizadas. O *software* precisou sofrer alterações para assegurar a compatibilidade com versões antigas do MatPlotLib (2 anos, como a 1.0.1), conforme documentado no histórico de atualizações da AudioLazy[19]. O Python 3.x teve uma quebra de compatibilidade com a sintaxe do Python 2.x, e devido à incompatibilidade do MatPlotLib com o Python 3.x ao início do desenvolvimento, foi utilizada a última versão do Python 2.x.

Como IDE (*Integrated Development Environment*), foi utilizado o *Spyder* 2.1.9[20], que no Windows é parte da distribuição *Python(x,y)* 2.7.3.0[21], ambos voltados para o desenvolvimento de software científico e de engenharia. O Spyder integra convenientemente diversos recursos, dentre os quais vale destacar seu *object inspector*, o qual, durante a escrita do código, automaticamente exibe a ajuda contida na *docstring* de um objeto em Python referenciado convertida em HTML (*HyperText Markup Language*).

O console *IPython* 0.12.1[22] foi utilizado durante o desenvolvimento, e os exemplos deste trabalho que necessitam de visualização de entrada e saída foram realizados utilizando tal console, o qual inicia suas linhas com "*In*" ou "*Out*", para entrada e saída, respectivamente, incluindo uma contagem do número de comandos (*statements*) fornecidos. Para a documentação geral no apêndice final do trabalho, há exemplos na forma de *doctests*, os quais são melhor descritos na seção 2.3.7, mas vale salientar que estes iniciam com o símbolo "*»>*" em linhas de entrada, e sem texto inicial em linhas de saída, relativos ao console padrão do Python.

A partir da versão 0.01 da AudioLazy, o histórico do desenvolvimento passou a ser armazenado em um repositório *git*, que garante a organização do projeto, o armazenamento completo do histórico de atualizações, comparação entre versões, recuperação de versões antigas, entre outros aspectos existentes no citado sistema de controle de versão. O *Mercurial* foi utilizado apenas para obtenção da versão de desenvolvimento da

[19] Disponível no repositório no GitHub, datada de 2013-02-06. A diferença incluiu a mudança de uma cor nominada (cinza claro) para a mesma utilizando componentes RGB (Vermelho, Verde e Azul, ou *Red, Green, Blue*) e outra mudança para evitar o uso de um método de alinhamento que ainda não existia em tal versão do MatPlotLib (método *tight_layout* da classe *matplotlib.figure.Figure*). Isso foi feito através da capacidade de programação reflexiva do Python, usando uma instalação do próprio MatPlotLib 1.0.1 para indicar quais eram as cores nominadas disponíveis, assim como os métodos da classe citada. Provavelmente o histórico de atualizações do MatPlotLib possui a inclusão de tais itens bem documentada, mas uma tal consulta não se fez necessária.

[20] Disponível em https://code.google.com/p/spyderlib/

[21] Disponível em http://code.google.com/p/pythonxy/

[22] Disponível em http://ipython.org/

MARLib e indiretamente para consulta *online* do código fonte do Python. O SVN (Subversion) foi utilizado indiretamente através da interface do SourceForce para consulta ao código do OctaveForge[23].

A versão mais recente da AudioLazy está sempre disponível em seu repositório público no GitHub:
http://github.com/danilobellini/audiolazy

Versões consideradas suficientemente estáveis foram colocadas junto ao PyPI (*Python Package Index*), local que armazena pacotes Python que estejam sob uma licença de *software* livre[24], caso o desenvolvedor ou os desenvolvedores se proponham a disponibilizá-lo dessa forma. A vantagem da presença do pacote no índice geral é a facilidade de instalação do mesmo por parte dos usuários em geral, bastando para isso utilizarem o software *pip*, recomendado pelo próprio PyPI, ou o *script* de instalação *setup.py*, maneiras que, em geral, facilitam a instalação de pacotes em Python[25]. A URL específica do projeto no PyPI é:
http://pypi.python.org/pypi/audiolazy/

O endereço contém o número de vezes em que o pacote foi obtido, seja por uma instalação via *pip* ou através do *link* de *download*. Mesmo na ausência de grandes divulgações, particularmente quanto à versão 0.03 que não teve nenhuma divulgação realizada além de comentários entre colegas, o número de downloads das versões foi de[26]:

Tabela 1.1: *Estatística de downloads da AudioLazy no PyPI*

Versão	Número de downloads
0.01	432
0.02	432
0.03	209

Em geral, não há controle sobre todos os locais onde o pacote de software se encontra atualmente, isto é, não se sabe quem são as pessoas que baixaram as centenas de cópias da AudioLazy. Adicionalmente, o número obtido com o PyPI serve apenas como limiar mínimo e estimador do número total de *downloads*, dado que não é possível obter a informação de quantas foram as vezes que o pacote foi obtido diretamente do repositório no GitHub.

A documentação em HTML estático, gerada utilizando o *Sphinx* 1.1.3 a partir do código[27] também foi disponibilizada para acesso, através do endereço:
http://pythonhosted.org/audiolazy/
que é o sistema de hospedagem oficial do Python para a documentação de seus pacotes. Infelizmente, o sistema não possui um contador de acessos, o que torna impossível a avaliação de frequência de uso no atual instante.

Para a avaliação utilizando versões específicas do MatPlotLib e do NumPy, foi utilizada o seguinte procedimento, utilizando para isso o gerenciador de pacotes Python *pip* e o sistema de criação de ambientes virtuais Python *virtualenv*, em uma máquina com o compilador C GCC 4.6.3 instalado:

```
$ [sudo] pip install virtualenv
$ mkdir ~/env_test
$ cd ~/env_test
$ virtualenv --distribute .
$ bin/pip install git+git://github.com/danilobellini/audiolazy
$ bin/pip install git+git://github.com/numpy/numpy@v1.5.0
$ bin/pip install git+git://github.com/matplotlib/matplotlib@v1.0.1
$ bin/pip install pyaudio
```

[23] http://sourceforge.net/p/octave/code, último acesso dia

[24] Alguns detalhes sobre tipos de licença e classificações de tipos de *software* encontram-se na seção 2.2. O endereço do PyPI é
http://pypi.python.org/

[25] Para a instalação via *pip* no Linux, basta digitar em um terminal "pip install audiolazy". No Windows, utilizando

[26] Valores avaliados em 2013-02-15. Vale salientar que a data de publicação da versão 0.03 foi 2013-01-23.

[27] A geração utiliza fundamentalmente a documentação inserida no próprio código, reformatando-a e organizando-a de maneira a ter o formato de um manual. Maiores detalhes na seção 2.3.8. O Sphinx está disponível em:
http://sphinx.pocoo.org/

O primeiro comando instala o *virtualenv*, e é necessário realizá-lo apenas na primeira vez, sendo que a versão instalada na realização desse procedimento foi a 1.8.4. Essa sequência de comandos obtém do repositório principal a última versão da AudioLazy e versões específicas do NumPy e do MatPlotLib (denotadas pelo número após o símbolo "@"), e já se encarregam de iniciar o processo de compilação do NumPy, do MatPlotLib e do PyAudio, visto que estes não estão escritos em puro Python. Para possibilitar o acesso ao wxPython para a exibição do gráfico em uma janela, foram necessários dois *links* simbólicos (o wxPython 2.8.12.1 foi originalmente instalado através do sistema de pacotes da distribuição Linux):

```
$ ln -s /usr/lib/python2.7/dist-packages/wxversion.py \
> lib/python2.7/site-packages/wxversion.py
$ ln -s /usr/lib/wx/python/wx.pth lib/python2.7/site-packages/wx.pth
```

Para acessar o Python do ambiente virtual, basta:

```
$ bin/python
```

E, no console Python, o MatPlotLib precisa ser associado ao wxPython com um par de comandos anteriores à importação do *matplotlib.pyplot* (ou do *matplotlib.pylab*), interna aos métodos e exemplos da AudioLazy que necessitam do MatPlotLib:

```
1 import matplotlib as mpl
2 mpl.use("wxagg")
```

Esse procedimento foi desnecessário ao usar o Spyder, o IPython ou o Python fora do ambiente virtual, e é válido apenas para a versão 1.2.0 do MatPlotLib. Para a versão 1.0.1, os gráficos foram exportados diretamente para PDF para testes de funcionalidade através do método *matplotlib.figure.Figure.savefig()*.

1.5 Contribuições do trabalho

A principal contribuição do trabalho é, sem dúvidas, a facilidade fornecida à continuidade da pesquisa em processamento de áudio através do desenvolvimento da AudioLazy, pacote de *software* que visou, desde o início, a expressividade, no sentido de ter seu código escrito na expectativa de que seres humanos tivessem facilidade ao utilizá-lo. Mas algumas consequências desse desenvolvimento incluem o que é melhor detalhado em cada uma das seções que seguem.

1.5.1 Educação

Nem todos os estudantes compreendem com facilidade o significado do processamento digital de áudio em diversas situações, por exemplo quando, apesar de serem utilizadas apenas poucas somas e multiplicações entre amostras sucessivas, se obtém eficientemente curvas não-lineares de resposta ao impulso, apesar do sistema ser um típico filtro linear invariante no tempo. Adicionalmente, representar o filtro através de sua formulação como equação do sistema no domínio da frequência (transformada Z) é algo bastante comum em engenharia elétrica, e utilizar essa representação diretamente no código evita que o aluno necessite de confusos passos intermediários à experimentação, encorajando-o à compreensão através da possibilidade de utilização prática imediata. Apesar desse exemplo dado se referir exclusivamente à utilização do objeto *audiolazy.lazy_filters.z*, o pacote AudioLazy possui diversos outros componentes que podem facilitar a compreensão do que são sinais de áudio, filtros não-lineares, o que representa a DFT, o que são bancos de filtros, de maneira integrada com um sintetizador que permite em tempo real o uso lúdico ou artístico dos conceitos. Em outras palavras, um simples filtro pode ser aplicado à entrada de um microfone e jogado à saída facilmente, mostrando sonoramente ao aluno o que acontece com o processamento realizado por tal filtro, algo que pode permitir ao aluno atribuir maior significado às equações e blocos utilizados na teoria por trás do DSP.

1.5.2 Arte

Embora o sintetizador presente na AudioLazy seja extremamente simples quando comparado com dispositivos de processamento de áudio digital comerciais da atualidade, os recursos fornecidos pelo algoritmo de consulta à tabela, o algoritmo de Karplus-Strong generalizado, a envoltória ADSR (*Attack, Decay, Sustain, Release*) linear, as primitivas de síntese (senoide, dente de serra e ruído branco uniforme), a possibilidade de criação e manipulação de tabelas para o algoritmo de consulta à tabela, mesmo enquanto esta está em uso, filtros lineares variantes e invariantes no tempo, filtros não-lineares, processamento matemático não-linear amostra por amostra, processamento em blocos, filtros *comb*, filtros ressonadores, passa-baixas, passa-altas entre diversos outros recursos presentes na AudioLazy e publicamente disponíveis permitem que um músico, um sonoplasta, um artista audiovisual ou algum outro artista envolvido com síntese e processamento de áudio possa criar uma obra de arte, integralmente ou parcialmente, a partir dos recursos da AudioLazy.

1.5.3 Ciência da computação

Modelos como o dicionário de estratégias (*audiolazy.lazy_core.StrategyDict*) implementam uma alternativa a possíveis *design patterns* a partir de uma variação do padrão estratégia (*strategy*), além do uso do padronizado *container* associativo "dicionário", possibilitando a resolução de problemas facilitada. O desenvolvedor pode encontrar a informação desejada consultando o dicionário de estratégias como um dicionário convencional que permite múltiplas chaves associadas a um único valor, o que permite a simultaneidade de nomes às estratégias ao mesmo tempo em que evita a repetição das mesmas e possibilita a varredura por todas as estratégias, caso necessário. É possível atribuir uma estratégia como padrão, e evitar que o usuário tenha de se preocupar com a quantidade ou com os nomes das estratégias, agrupando-as em um único lugar comum. Como em Python a função é um valor, não há nada que proíba o usuário de extrair uma estratégia desejada e armazená-la localmente com o nome desejado para um uso específico, o que significa que a existência dessa forma de organização não remove nenhuma capacidade da linguagem Python quanto ao uso das estratégias.

Outro elemento fundamental é a massiva sobrecarga de operadores *operator overload* realizada através da metaclasse abstrata *audiolazy.lazy_core.AbstractOperatorOverloaderMeta*). A título de exemplo, a classe *audiolazy.lazy_stream.Stream* possui 35 operadores sobrecarregados, algo que dificultaria a manutenibilidade do código caso estivesse escrito como 35 métodos individualizados, visto que pequenas alterações na classe necessitariam de grandes mudanças no código. Para evitar grandes alterações, a metaclasse abstrata foi criada, abstrata pois ela não possui os operadores, e uma concretização da metaclasse é a que contém os *templates* dos três tipos de operadores, e a instanciação da classe é feita de forma a garantir tanto a existência dos operadores na classe como permitir a generalização dos operadores quando estes possuem um padrão comum. Essa contribuição se refere a uma necessidade que torna justificável a utilização prática de uma metaclasse abstrata, e portanto de uma hierarquia de metaclasses.

A criação de uma DSL para processamento de sinais sobre a linguagem Python não foi, a rigor, uma DSL, mas sim um uso específico, seletivo e eficiente dos recursos já existentes na linguagem, servindo de demonstração das capacidades da linguagem e da importância de existência de recursos tais como avaliação tardia, sobrecarga de operadores e funções de ordem superior para a elaboração de *software* modular, reutilizável, expressivo e de simples manutenção.

1.5.4 Análise e síntese musical

A AudioLazy não é voltada à análise de partituras ou da análise musicológica. Entretanto, a integração entre a AudioLazy e o Music21, pacote escrito em Python e destinado ao uso em musicologia e análise de partituras e *corpora* de partituras[28], é possível e tal integração faz parte dos exemplos de uso da AudioLazy publicamente disponibilizados. Um dos itens exemplificados pelo Music21[29] é a intenção de possibilitar a reprodução de quartos de tom, com uma sintaxe e estrutura interna prontas mas na ausência de um sintetizador. Essa síntese, com a AudioLazy, é bastante simples de ser realizada, e auxilia o analista a ouvir aquilo que

[28] Maiores detalhes na seção 2.4.2.

[29] O exemplo consta na página inicial da Music21:
http://web.mit.edu/music21/, último acesso dia 2012-02-18.

está escrito em sua partitura. O exemplo de integração entre os pacotes fornecido junto com a AudioLazy toca um coral de J. S. Bach do *corpus* disponibilizado em conjunto com o pacote Music21, mas poderia ser utilizado para tocar qualquer outra partitura em MusicXML que o Music21 seja capaz de abrir, permitindo valores quaisquer de frequência, isto é, não se limitando a frequências da escala cromática temperada.

1.5.5 Possibilidades de pesquisas futuras

Diversas são as possibilidades de pesquisa futura que foram, no mínimo, facilitadas com a existência e disponibilidade pública do pacote de *software* AudioLazy. O fato deste pacote estar sob a GPLv3 garante que, em situações nas quais a lei é respeitada, todo aquele que utilizá-la no sentido de geração de novo código de utilização por outrem deve permitir a esse potencial utilizador a disponibilidade desse novo código, estratégia "viral" que pretende fomentar a disponibilidade de código em processamento de áudio.

Tais pesquisas futuras podem ocorrer em diversas áreas, incluindo MIR, codificação, processamento de sinais de voz, transcrição musical, síntese de áudio, etc., não se restringindo ao processamento de áudio.

1.6 Organização do texto

Há muitas passagens que incluem o uso de código-fonte para exemplos. Tais códigos, a menos quando explícito em contrário, estão em Python, e preferiu-se utilizar a própria AudioLazy para fazer cada explicação de conceito por possibilitar uma inclusão natural do vínculo entre o conceito e o pacote implementado. Diversas referências utilizadas são endereços da Internet indicados em notas de rodapé, junto com sua respectiva data de acesso. Nos casos em que o endereço foi indicado apenas por ser o local em que se encontra um *software* em uma versão específica explícita no texto, foi desnecessário indicar a data de acesso.

Este capítulo informa as características gerais sobre o trabalho, tanto na questão do desenvolvimento do pacote *AudioLazy* quanto às motivações que levaram a tal desenvolvimento, de forma a introduzir e justificar o título do trabalho. Além disso, traz consigo uma breve seção de considerações preliminares sobre as diferentes áreas do conhecimento envolvidas no problema, incluindo desde os fundamentos das áreas de processamentos de sinais até conceitos musicais e cognitivos relevantes à obtenção de informações musicais a partir do áudio.

O capítulo 2 introduz os conceitos relativos ao desenvolvimento de *softwares* na área de processamento de sinais digitais, incluindo uma análise sobre parte das linguagens que possibilitam esse tipo de desenvolvimento. Em continuidade, o capítulo introduz aspectos fundamentais do pacote AudioLazy, uma sucinta descrição sobre seus módulos e organização geral, assim como um maior detalhamento sobre a classe *Stream*, que representa um fluxo de amostras como um objeto que se comporta como uma sequência. Finalmente, são introduzidos tipos convencionais de sequências musicais.

Em seguida, o capítulo 3 apresenta filtros digitais, sua representação matemática e sua implementação junto à AudioLazy, incluindo um detalhamento sobre os desafios enfrentados, integração com outros pacotes de *software* e aspectos de base para a implementação de tais filtros, tais como diferentes possibilidades de representação de polinômios.

As conclusões sobre os experimentos e o trabalho como um todo são finalmente apresentadas no capítulo 4, enumerando as aplicações do trabalho realizado e trazendo ideias para a continuidade dessa pesquisa em trabalhos futuros, assim como propostas de continuidade ao pacote AudioLazy.

Há três apêndices no trabalho. No primeiro, há uma descrição do problema da transcrição, tópico que serviu de motivação inicial ao desenvolvimento. No segundo, são fornecidos códigos de exemplo utilizando a AudioLazy. Finalmente, o último apêndice contém a versão da documentação pública oficial da AudioLazy.

Capítulo 2

Softwares e estruturas de dados

Este capítulo introduz a AudioLazy e o componente que fundamentou o desenvolvimento, em torno do qual parte massiva do pacote está baseado: a classe *Stream*. Entretanto, para uma adequada compreensão dos objetos que representam as instâncias dessa classe, e também de sua necessidade, não é suficiente alegar que se tratam de fluxos de informação. Para esse fim, a explicação dos conceitos, motivações e justificativas por trás do desenvolvimento da citada classe e do pacote AudioLazy fazem parte do atual capítulo.

Na seção 2.1, são introduzidos conceitos básicos de representações de dados em *containers*, com base na relação entre seus elementos, no tipo de acesso e no que representam esses elementos, incluindo estratégias de avaliação dos mesmos. No que segue, a seção 2.2 avalia as características de linguagens que podem ser utilizadas para a implementação de *software* voltado ao processamento de áudio, incluindo aspectos sobre a representação utilizada para o áudio e um conjunto de critérios, sumarizados na tabela 2.1, que permitem a comparação destas linguagens. Em seguida, a seção 2.3 enuncia o pacote AudioLazy como solução aos problemas identificados, com uma explicação acerca de seus aspectos básicos e fundamentais, de sua organização e de características adicionais tais como documentação e testes. Ao final, a seção 2.4 aplica o conceito de sequências à representação simbólica do som, interpretável como estando em contraste com a representação sobre amostras utilizada nas demais seções, ao mesmo tempo que vincula os tipos de representação.

2.1 Containers

Um *container* (também conhecido como contentor ou coleção[1]) é o nome dado a qualquer estrutura de dados que sirva como agrupamento de elementos, ou como pode ser dito em orientação a objetos, um objeto que contém outros objetos. Os tipos de containers são classificados de acordo com suas potencialidades e limitações, tais como restrições e tipos de acesso, relações internas entre os objetos agrupados, possibilidades de alteração e efeitos colaterais à alteração.

No que segue, a palavra "elemento" sempre se refere a um objeto contido em um container. A ênfase nesta seção é dada aos aspectos das estruturas que são relevantes para o presente trabalho visando possibilitar a compreensão do container *Stream* que implementa um fluxo de dados na AudioLazy, suas semelhanças e diferenças com relação aos demais containers, e sua necessidade para a representação de áudio. Na seção 2.1.1, são fornecidas explicações do que são sequências e conjuntos, com base na necessidade de relações internas entre os elementos armazenados. Na seção 2.1.2 encontram-se explicações sobre os tipos de acesso aos elementos armazenados. Na seção 2.1.3 há uma explicação sobre os tipos de avaliação do conteúdo de

[1] Coleção é a sugestão feita aos tradutores do tutorial oficial da linguagem Python. *Contentor* é outra sugestão de alternativa à palavra *container*, segundo o dicionário Priberam. É também utilizada a escrita *contêiner* sob o Acordo Ortográfico da Língua Portuguesa de 1990, porém esta aparentemente é de uso restrito a um recipiente para transporte de mercadorias, segundo o mesmo dicionário. O uso da palavra *container* no presente texto restringe-se ao contexto da ciência da computação como uma classe ou como estrutura de dados abstrata.
www.priberam.pt/dlpo, último acesso dia 2013-02-13.
http://turing.com.br/pydoc/2.7/tutorial/NOTAS.html#traducoes-adotadas-para-termos-especificos, último acesso dia 2013-02-13.

uma variável, conceito que justifica o fato do pacote AudioLazy conter a expressão "Lazy" em seu nome. Em 2.1.4 são introduzidos conceitos relativos aos tipos de elementos que podem pertencer a um dado container.

2.1.1 Relações entre elementos

Para tornar possível o processamento de um sinal de áudio em um computador digital, não é nem necessária nem suficiente a possibilidade de se realizar a amostragem e a quantização de um som físico. Não é necessária pois é possível sintetizar áudio a partir de padrões e equacionamentos matemáticos *a priori*, a partir de geradores de números aleatórios, de sinais de outra natureza previamente amostrados e quantizados ou de uma combinação disso tudo. Não é suficiente pois precisa existir alguma forma de organização e armazenamento conforme a necessidade de um uso específico. Um pressuposto básico ao processamento de áudio em um DSP é o de acesso à informação, que normalmente é vista como um fluxo sequencial de amostras no tempo, isto é, como uma estrutura de dados que impõe uma relação de ordem na qual cada amostra está associada a um instante no tempo. Chamamos essa estrutura de dados, ou família de estruturas de dados, de sequência, a qual é caracterizada através da relação de ordem entre seus elementos, independente da natureza destes[2].

Um conjunto é uma estrutura ou classe que apenas garante o acesso à informação, mas não possui nenhuma meta-informação associada a seus elementos além do mínimo necessário para possibilitar seu acesso e armazenamento, nem mesmo a relação de ordem ou a quantidade de ocorrências de cada um de seus elementos. O que caracteriza diferentes elementos em um conjunto é apenas sua discernibilidade com relação a um procedimento de identificação ou de equivalência.

Na prática, em linguagens de programação orientadas a objeto, essa discernibilidade costuma ocorrer pela localização em memória, elemento associado à identidade ou referência ao objeto, ou por uma rotina associada a um operador de igualdade, um procedimento que verifica a equivalência entre objetos. Espera-se que esse procedimento seja uma relação de equivalência, isto é, que satisfaça as seguintes propriedades:

- Reflexiva: Todo objeto é equivalente a si próprio
- Simétrica (ou comutativa): Se A é equivalente a B, então B é equivalente a A.
- Transitiva: Se A é equivalente a B, e B é equivalente a C, então A é equivalente a C.

Como amostras de áudio em geral são números em ponto flutuante, há uma dificuldade associada à comparação de igualdade, ou equivalência. Com amostras inteiras de um mesmo tamanho (e.g. 32 bits) sob um padrão bem determinado de sinalização (e.g. números em complemento de 2), é possível comparar o conteúdo diretamente pela igualdade bit a bit. Com números representados por ponto flutuante de base binária com 32 ou 64 bits sob o padrão IEEE 754, há números com mais de uma representação, e talvez o exemplo de maior clareza seja o número zero, o qual possui sinal tanto positivo como negativo em tal representação. Maiores detalhes sobre isto são vistos na seção 2.3.7.

Deve-se tomar cuidado para não misturar a relação de ordem imposta aos objetos de uma sequência com uma possível relação de ordem associada a aspectos comuns à classe desses objetos. Números inteiros são objetos que permitem relações de ordem triviais, tais como a ordem ascendente ou descendente, entretanto o mesmo conjunto de números pode estar disposto de diferentes formas em sequência, sendo essas duas relações de ordem apenas duas possibilidades. A rigor, para a ordenação em sequência, não há necessidade de que os objetos pertençam a uma mesma classe e nem que sejam comparáveis, ou seja, a relação de ordem em uma sequência é imposta aos elementos, e não extraída destes.

Resumidamente, a diferença aqui exposta entre a sequência e o conjunto está na organização de seus elementos. Em uma sequência os elementos são ordenados, única relação necessária para caracterizar a estrutura, a qual é imposta e independe da natureza dos elementos, enquanto que em um conjunto os elementos são identificados por comparações e relações de equivalência entre os próprios elementos entre si.

Dos quatro containers padrões do Python, são exemplos de estruturas que se comportam como o conjunto descrito o próprio conjunto (*set*) e o dicionário (*dict*), embora este último seja um container associativo de

[2] Maiores detalhes sobre o tipo de elementos na seção 2.1.4

pares *{chave: valor}*. São exemplos de sequências a *n*-upla, também conhecida como tupla (*tuple*) e a lista (*list*). Há um resumo das características dessas estruturas de dados no Python em 2.2.4. O dicionário é melhor detalhado na seção que segue.

2.1.2 Tipos de acesso

Embora sequências e conjuntos sejam objetos que contém elementos, nada foi dito sobre como esse acesso aos elementos é realizado. Os containers em geral necessitam, para toda aplicação prática, de alguma interface que possibilite o acesso aos seus elementos. Essa interface existe, no mínimo tacitamente, ao referenciarmos os elementos para realizar até mesmo a mais simples descrição verbal informal, mas, na prática, caso não exista a possibilidade de obtenção de uma informação, pode-se argumentar que esta não existe, ou minimamente que sua existência não faz diferença, pois caso fizesse, tal diferença nos levaria precisamente à informação, direta ou indiretamente. O acesso a essa informação pode ser feito de duas maneiras:

- Aleatório: Através do uso de índices ou chaves, independente das amostras anteriores acessadas;
- Sequencial: Através de iteradores ou cursores, por meio de uma relação com a disposição do elemento anterior consultado.

O acesso aleatório à informação pode ser exemplificado como aquele realizado pela RAM (*Random Access Memory*), e se refere à possibilidade de acesso por endereçamento, a princípio sem restrições quanto à ordenação das requisições, organização interna da memória, variação de atraso de resposta ou tipo de acesso (leitura/escrita). O elemento enfatizado como acesso aleatório é a possibilidade de acessar diferentes endereços como se o resultado fosse independente da ordem das requisições, isto é, como se as requisições pudessem ser tratadas como uma variável aleatória.

Em sequências, o acesso aleatório é possível através de índices com base em um valor inicial convencionado para o índice, tipicamente 0 (zero) ou 1 (um), o que nos possibilita referenciar o *n*-ésimo elemento da sequência diretamente, sem a necessidade de conhecermos os $n-1$ elementos anteriores ou os elementos seguintes, independentemente da quantidade. A quantidade de elementos de uma sequência com acesso aleatório a seus elementos, na prática, deve ser finita para possibilitar o armazenamento dos elementos ou, minimamente, o armazenamento do número utilizado como índice para o acesso. Índices negativos também podem ser utilizados, possivelmente como referência a valores anteriores ao convencionado como inicial ou como uma forma de percorrer a sequência do maior para o menor índice, isto é, invertendo a relação de ordem imposta.

O acesso aleatório em uma sequência também pode ser feito de maneira sequencial através de índices. Para isso, cria-se uma uma sequência de índices $[0, 1, 2, ..., n-1]$ e se acessa, para cada índice, o respectivo valor da sequência. Isso é feito, por exemplo, ao utilizarmos um laço para percorrer índices, utilizando cada um desses índices sobre a sequência para obter seus valores, e incrementando o valor do índice atual para obter o índice seguinte. Podemos interpretar o vetor da linguagem C, embora esta não seja uma linguagem orientada a objetos[3], como uma sequência na qual o "elemento inicial" é indexado pelo número zero:

```
1 for(i = 0; i < n; i++) /* Índice i percorre valores de 0 a n-1 */
2   printf("%d", seq[i]) /* Exibe o i-ésimo elemento de seq */
```

O laço *for* da linguagem C possui como característica o fato de que ele nada mais é do que uma sintaxe compacta de um laço condicional com condição de término antes de cada iteração[4]. Percebe-se que não há realmente um objeto iterador, há apenas um número, especificamente um índice, sendo incrementado a cada iteração. Em Python[5], que é uma linguagem orientada a objetos, o mesmo exemplo utilizaria implicitamente um iterador para obter os índices:

[3] Em geral, índices em C realizam apenas desvios no acesso à memória. Em outras palavras, *seq[n]* obtém o elemento que reside no endereço de seq somado a *n* vezes o tamanho de cada elemento de seq, informação conhecida pelo compilador da linguagem através da definição da sequência. Isto tem a ver com o que é chamado na seção 2.1.4 de tipagem fraca.

[4] Laço este que também é conhecido como *while*.

[5] Maiores detalhes sobre o Python estão em 2.2.4.

```
for idx in range(n): # Em que range(n) = [0, 1, 2, ..., n-1]
  print seq[idx]
```

Infelizmente a linguagem C pode dificultar a compreensão e se tornar críptica ou inadequada como pseudo-código para o que segue, e o restante dos exemplos encontram-se apenas em Python.

Apesar de ser realizado "em sequência" e "sobre uma sequência", esse tipo de situação, a menos da obtenção do valor de *idx* no exemplo em Python, não é um "acesso sequencial", pois cada elemento de *seq* é acessado por meio de índices, não por meio de uma estrutura de iterações em sequência. Em outras palavras, nada impede alguém de romper a ordenação do acesso sobre a sequência *seq*.

```
for idx in range(n):
  if n == 3:
    print seq[0] # Primeiro elemento
  else:
    print seq[idx]
```

Estes são exemplos triviais de exibição de elementos de sequências, com o único objetivo de distinguir o acesso através de índices do acesso através de iteradores. Um iterador, estrutura necessária para o acesso sequencial, não é um container, mas um objeto que possibilita o acesso à informação de um container que, se este permite o acesso sequencial por meio de iteradores, é dito um iterável. Segue um exemplo em Python com uma lista de 3 elementos, que é uma sequência, sobre a qual é obtido um iterador:

```
In [1]: seq = [0, 1, 3] # Colchetes denotam uma lista em Python

In [2]: meu_iterador = iter(seq) # Cria um iterador sobre seq

In [3]: meu_iterador.next() # Requisita o elemento seguinte
Out[3]: 0

In [4]: meu_iterador.next()
Out[4]: 1

In [5]: meu_iterador.next()
Out[5]: 3

In [6]: meu_iterador.next()
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
...
```

Como exemplificado, o objeto iterador percorre os elementos do container sobre o qual foi construído, devolvendo sequencialmente um novo elemento a cada chamada de seu método *next()*. Na ausência de um elemento seguinte para ser devolvido, uma exceção específica é emitida, embora sejam possíveis implementações alternativas nas quais um valor padrão é devolvido. Os nomes *iter*, *next* e *StopIteration* são específicos e padronizados na linguagem Python[6], entretanto o padrão, ou protocolo, é o mesmo, independente da linguagem utilizada. É também possível outros tipos de iteradores, por exemplo:

[6] A rigor, a iteração no Python também é possível a partir de índices caso isso seja possível. A especificação completa pode ser encontrada no
PEP 234 http://www.python.org/dev/peps/pep-0234/, versão de 2007-06-28.
Vale salientar, entretanto, que o método *next()* foi renomeado para *__next__()* no Python 3.x, conforme consta no
PEP 3114 http://www.python.org/dev/peps/pep-3114/, versão de 2007-06-19.

- Iteradores reversos[7]: Percorrem a sequência de seu último até seu primeiro elemento, na ordem contrária à sequência.
- Iteradores com saltos de tamanho k: Devolvem um elemento a cada k elementos, descartando os demais $k-1$ elementos.

Em conjuntos, a situação é diferenciada. Na ausência de uma relação de ordem imposta aos elementos, o acesso aleatório com índices não possui um sentido claro. Qual é o primeiro elemento do conjunto *{quadrado, triângulo, hexágono}*? Se adotarmos como critério de ordenação a ordem de apresentação, o primeiro elemento seria o quadrado, mas se adotarmos como critério a ordem ascendente do número de lados do polígono representado, iniciaríamos com o triângulo, ao passo que a ordem lexicográfica iniciaria com o hexágono. Embora nem todas as ordenações tenham algum significado como o exemplificado com os polígonos, para um mesmo conjunto de N elementos, todas as $N!$ permutações de seus N elementos são possíveis maneiras de sequenciar o conjunto, e um iterador sobre qualquer uma delas é válido para realizar um acesso sequencial a seus elementos.

Há outras maneiras de acessar aleatoriamente a informação de conjuntos sem o uso do protocolo de iteração. Uma necessidade básica do conjunto é a possibilidade de realizar consultas de pertinência de um dado elemento ao conjunto, o que é uma consulta a um bit de informação[8]. Entretanto, somente com tais consultas, é possível conhecer toda informação de um conjunto apenas quando este seja um subconjunto de um conjunto conhecido. A iteração sobre um conjunto, além de mais eficiente do que uma consulta a cada possível elemento do conjunto, garante que todos os seus elementos sejam acessados.

O acesso aleatório não é possível apenas através de índices, sendo possível também através de nomes utilizados como chaves. Um dicionário[9] pode ser visto como um conjunto de pares *{chave: valor}* que permite o acesso a cada valor diretamente através de cada chave, de forma similar ao acesso em sequências por meio de índices. As chaves podem ser vistas como um conjunto, e a consulta não mais se limita à pertinência, mas a um objeto *valor* associado à chave.

Importante salientar que essas modalidades de acesso se referem à interface de uma linguagem com o programador, e que, internamente à linguagem, as representações podem possuir aspectos distintos aos daqueles fornecidos pela interface. Tipicamente as representações internas incluem ponteiros, blocos contiguos de memória, listas ligadas e árvores, mas transcende o objetivo do trabalho maior detalhamento sobre o assunto. Resumidamente, tanto conjuntos/dicionários como sequências permitem o acesso sequencial por meio de iteradores, e de acesso aleatório por meio de ou chaves de consulta, ou índices, respectivamente.

2.1.3 Estratégias para a avaliação de expressões

Apesar da facilidade e do baixo custo em se armazenar áudio na tecnologia atual, para possibilitar uma implementação física, é necessário que a memória utilizada seja finita, o que significa, entre outras coisas, que a duração total de áudio que pode ser armazenada em um estado de memória é finita. Em outras palavras, embora seja possível definir sequência como entidade potencialmente infinita através de uma regra de construção de suas ilimitadas amostras, somente é possível construir um conjunto finito de elementos de tal sequência. Nessa curta exposição, temos duas estratégias de avaliação expostas: em uma delas, todas as amostras de uma sequência são armazenadas de uma só vez em alguma das memórias de um computador digital, enquanto na outra o que importa é o caminho para a obtenção das amostras, as quais são criadas

[7] No Python, o *reversed()* realiza essa tarefa.
PEP 322 http://www.python.org/dev/peps/pep-0322/, versão de 2004-05-19.

[8] Um bit é a quantidade de informação necessária à resolução de uma dicotomia na qual os dois eventos possíveis são considerados equiprováveis. No caso, a dicotomia se refere à pertinência ou não-pertinência do elemento consultado ao conjunto, que podem ser representadas através dos dígitos 1 e 0. Entretanto, a representação da dicotomia "verdadeiro/falso", conhecida como variável booleana, pode ocupar um espaço de memória maior, tipicamente de 32 bits, valor relativo ao tamanho dos registradores do processador.

[9] O nome dicionário é utilizado neste trabalho em decorrência de seu uso na linguagem Python. A título de exemplo, o Ruby chama a mesma estrutura de *Hash*, por conta do uso de tabelas *hash*, ou tabelas de dispersão, em sua implementação. Outros nomes possíveis são o mapeamento (*mapping*) e o vetor associativo (*associative array*), embora o último nome seja desencorajado pelo fato de que vetores (*arrays*) são tipicamente sequências.

somente quando requisitadas, viabilizando o armazenamento de estruturas com quantidade de elementos potencialmente infinita. Isto nos leva à seguinte definição;

- Avaliação antecipada, imediata, gulosa ou ansiosa (*eager evaluation*): Ocorre quando a expressão é avaliada assim que associada a uma variável. A posterior obtenção do resultado a partir do conteúdo armazenado na variável é conhecido como *chamada por valor*.
- Avaliação tardia, preguiçosa (*lazy evaluation*): Ocorre quando a expressão é avaliada quando o valor da variável que a armazena é requisitado.

A avaliação tardia se refere a um aspecto importante de linguagens de programação funcionais [Hug89], contexto em que há dois tipos possíveis de chamada ou processamento das expressões associadas:

- Chamada por nome (*call by name*) ou computação suspensa: Ocorre quando a expressão é reavaliada a cada requisição.
- Chamada por necessidade (*call by need*): Ocorre quando o resultado da expressão é armazenado em seu lugar após sua primeira requisição, uma forma de memorização (*memoize*). Há quem utilize o nome de avaliação preguiçosa (*lazy evaluation*) exclusivamente para esse tipo de chamada [Hud89].

É notável a generalidade das estratégias de avaliação, as quais se referem a expressões quaisquer, não se restringindo ao acesso de elementos em containers. Ambas as estratégias de avaliações, quando aplicadas a containers, podem ser utilizadas tanto para o acesso aleatório como para o acesso sequencial. Na avaliação imediata, tradicional em linguagens imperativas, os valores já se encontram na memória no instante em que são requisitados, o que possibilita tanto o acesso aleatório como o acesso sequencial sobre os mesmos da forma como tais conceitos foram enunciados na seção 2.1.2. Na avaliação tardia, o resultado de uma expressão que depende do resultado de outra pode se tornar ineficiente, sobretudo no caso de chamadas por nome. Por exemplo, ao se tentar calcular o fatorial de um número recursivamente através da propriedade $n! = n.(n-1)!, n > 1, n \in \mathbb{Z}$, pode-se utilizar a estratégia imediata para obter todos os valores de entrada de 0 (zero) até um N suficientemente grande para não se esperar uma requisição maior. Em uma estratégia tardia, caso a requisição sobre o valor N ocorra e não seja a única, é necessário que exista uma memorização para que o desempenho não seja inferior ao da estratégia imediata. Mais precisamente, a estratégia imediata calcula $\Theta(N)$ multiplicações para as k requisições, enquanto a estratégia tardia sem uma memorização calcula $O(N.k)$ multiplicações, no pior caso (todas as k requisições iguais a N). Por outro lado, a estratégia tardia com memorização calcula $\Theta[max_{i=0,1,\ldots,k}(k_i)]$ multiplicações, ou $O(N)$ no pior caso. Há pelo menos duas vantagens no uso da estratégia tardia:

- O cálculo é realizado sob demanda, o que significa que não há um atraso inicial por aguardar a construção da tabela de valores;
- Não é necessário estipular o valor máximo N, este será dinamicamente ajustado conforme a necessidade;

Em geral, algumas das vantagens associadas ao processamento tardio incluem o aumento de desempenho por evitar cálculos desnecessários e a habilidade de criação de estruturas de dados potencialmente infinitas. Uma terceira vantagem pode ser vista como consequência das duas anteriores, e pode ser exemplificada com o iterador. Um iterador nada mais é que um objeto que percorre um container, cujas informações estão armazenadas em algum lugar. Caso o container não armazene essas informações e apenas se limite a fornecer a informação requisitada através do iterador, não existe a necessidade de que todos os valores estejam armazenados (primeira vantagem), Um container alternativo pode possuir apenas alguns números (e.g. $[1, 2, 3]$), e ter um iterador que o percorre circularmente (i.e., $[1, 2, 3, 1, 2, 3, 1, ...]$). Isto já faz do próprio iterador uma sequência com avaliação tardia, além de ilimitada (no caso, periódica, segunda vantagem listada). A terceira vantagem é a possibilidade de criação de um fluxo de controle como uma abstração armazenada em um objeto. No exemplo do iterador, isso ocorre no sentido em que o iterador evita uma construção do tipo:

```
val = 1
while True:
  faz_alguma_coisa(val)
  val = val + 1 if val < 3 else 1
```

No exemplo, a função *faz_alguma_coisa*(·) é chamada a cada instante com um valor da sequência [1, 2, 3, 1, 2, 3, 1, ...], entretanto esta é feita através das primitivas da linguagem e não como uma abstração referenciável na linguagem. É a esse tipo de referencialidade ou abstração sobre o fluxo de controle que se refere a terceira vantagem.

Funções geradoras[10] são funções que retornam geradores, um tipo especial de iteradores disponível no Python, que permite exatamente converter o tipo de estrutura exposto acima em uma abstração:

Código-fonte 2.1: *Função geradora em Python*

```
In [1]: def gerador_123():
   ...:   val = 1
   ...:   while True:
   ...:     yield val
   ...:     val = val + 1 if val < 3 else 1
   ...:

In [2]: gerador_123
Out[2]: <function __main__.gerador_123>

In [3]: gerador_123()
Out[3]: <generator object gerador_123 at 0x19e0050>

In [4]: sinal = gerador_123()

In [5]: sinal.next()
Out[5]: 1

In [6]: sinal.next()
Out[6]: 2

In [7]: sinal.next()
Out[7]: 3

In [8]: sinal.next()
Out[8]: 1
```

2.1.4 Elementos

Basicamente, há dois tipos de containers com relação ao tipo de conteúdo que lhe é permitido conter, por questões de implementação. Esse conteúdo, os elementos, podem ser todos homogeneamente de um mesmo tipo ou classe, ou heterogeneamente permitir uma mistura de tipos diversos sem necessariamente um padrão além de sua referencialidade.

Em geral, estruturas homogêneas de elementos de tamanho fixo são mais eficientes no sentido de evitarem o acesso indireto através de ponteiros, através do armazenamento em blocos contíguos de memória, entretanto esse tipo de preocupação costuma apresentar grandes limitações quando levado à programação em níveis que visam abstrair a maneira como o hardware implementa uma dada tarefa. Em geral, é dado o nome de tipagem fraca ao elemento cuja estrutura interna na memória é diretamente acessada, e de tipagem forte quando os

[10] Especificação e motivação para o desenvolvimento desse recurso podem ser encontradas em: PEP 255 http://www.python.org/dev/peps/pep-0255/, versão de 2009-01-18.

tipos, ou classes, realizam a organização da estrutura de memória permitindo ao usuário abstrair um bloco de memória como um tipo, uma estrutura ou um objeto, ao invés de se preocupar com uma sequência de números e ponteiros (referências). Um exemplo clássico de distinção entre tipagem forte e fraca é dado com a manipulação de cadeias de caracteres (i.e., *strings*), as quais em codificações como ASCII (*American Standard Code for Information Interchange*) permitem o mapeamento direto do número de caracteres e o número de *bytes* utilizado em sua representação. Porém, no caso de caracteres UTF-8, UTF-16 ou alguma outra codificação que utiliza quantidades variáveis do número de *bytes* por caractere representado, o mapeamento passa a ser não-trivial. Quando esse mapeamento precisa ser realizado manualmente através de tabelas, é um sinal de que a tipagem é fraca, e quando a linguagem se preocupa em realizar o mapeamento para o desenvolvedor, a tipagem é forte.

Há mais um ponto a ser enfatizado com relação à tipagem, já associado ao contexto de orientação a objetos. Uma tipagem é dita estática quando a validação de argumentos em uma chamada é realizada através de uma relação de pertinência, utilizando-se para isso a estrutura hierárquica da herança entre classes. Em outras palavras, qualquer entidade chamável (e.g. uma função, um método) que tenha parâmetros receberá argumentos os quais, sob uma linguagem de tipagem estática somente aceitará o parâmetro caso este seja do tipo declarado como esperado para o parâmetro. Por exemplo, um método que espera um objeto *Filtro* e recebe um objeto *FiltroLinear* somente aceitará o argumento como válido se FiltroLinear for uma classe derivada de Filtro, do contrário acusará um erro de tipo, seja na forma de uma exceção, seja na forma de um impedimento à compilação.

Uma alternativa à tipagem estática é a tipagem dinâmica (ou *duck typing*), na qual os parâmetros não são avaliados pela estrutura hierárquica, mas sim através de sua interface (métodos, atributos) em tempo de execução, interface esta que precisa somente ser capaz de responder às necessidades da específica tarefa requisitada. O nome em inglês referenciando o "pato" surge da máxima epistemológica e pragmática de James Whitcomb Riley:

> "*Quando vejo um pássaro que anda como um pato e nada como um pato e grasna como um pato, eu o chamo de pato.*"

Um exemplo prático desse conceito já na implementação da AudioLazy pode ser exemplificado com o caso dos filtros. Na AudioLazy, um filtro é qualquer objeto chamável que recebe um sinal de entrada e resulta em um sinal de saída. Isso independe se o objeto filtro é derivado ou não das classes que definem os filtros que são fornecidos junto ao pacote. Isso é, de fato o que ocorre na AudioLazy, conforme o exemplo realizado em IPython[11]:

Código-fonte 2.2: *Exemplos de tipagem dinâmica na AudioLazy*

```
In [1]: from audiolazy import *

In [2]: filt1 = lambda x: x + Stream(1, 2, 3)

In [3]: filt2 = lowpass(pi/2)

In [4]: cfilt = CascadeFilter(filt1, filt2)

In [5]: cfilt([1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2])
Out[5]: <audiolazy.lazy_stream.Stream at 0x206e350>

In [6]: list(_)
Out[6]:
[1.464101615138,
 2.5884572681201496,
 3.6217782649109465,
 2.4345541763850367,
```

[11] Esse exemplo utiliza conceitos vistos no capítulo 3, de forma que serve apenas para ilustrar a funcionalidade e a integração da AudioLazy com estruturas convencionais do Python através da tipagem dinâmica.

```
2.848489248198889,
3.6914536239792786,
3.185274440148736,
3.7816949441849594,
4.673556144159756,
3.4484280173155444,
3.8522067326721343,
4.6924497209417595]
```

Uma breve explicação de cada uma das linhas do código (numeração entre colchetes):

1. Importa todo o pacote audiolazy para o *namespace* atual. Essa é uma prática comum em consoles, embora não seja recomendada em arquivos, pois pode haver uma perda de controle com relação aos nomes disponíveis e à sobreposição de nomes.

2. Define uma função com uma entrada x e uma saída, que é a soma desse x com uma sequência periódica $[1, 2, 3, 1, 2, 3, 1, 2, 3, ...]$, denotada pelo construtor da classe Stream. Podemos chamar isso de "função que soma uma sequência pré-determinada à entrada"

3. Define um filtro passa-baixas com frequência de corte (isto é, frequência cujo ganho em potência é metade do máximo, o que equivale a aproximadamente -3dB) igual à taxa de Nyquist ($\pi/2$ radianos por amostra).

4. Cria um filtro como a cascata (aplicação em série) dos dois filtros definidos anteriormente.

5. Aplica ao filtro uma lista (sequência finita) de entrada, contendo 12 elementos. O resultado devolvido é um objeto Stream, um iterável sem tamanho definido.

6. Nos consoles Python, o "_" (caractere *underscore*) representa o último resultado, equivalente ao "ans" do console do MatLab e do Octave. O $list(\cdot)$ é o construtor do tipo lista, que aceita qualquer iterável. É um leve perigo realizar essa conversão sobre iteráveis sem finalização definida, onde o impasse (*deadlock*) seria evidente.

2.2 Linguagens

Há muitas linguagens de programação existentes para auxiliar no desenvolvimento de *softwares* em geral, alguns dos quais são brevemente comentados ou analisados nesta seção. Aparenta ser pouco produtivo realizar a reconstrução de algo que já está pronto, entretanto essa reconstrução pode ser até mesmo mais eficiente para se atingir um determinado resultado do que a tentativa de agrupar elementos prontos cujas interfaces não são compatíveis, além de possibilitar o reuso para aplicações futuras. Muitos podem ser os motivos que levam uma pessoa a reconstruir um pacote de software voltado ao desenvolvimento, tais como

- Hardware e softwares necessários como requisitos;
- A disponibilidade do código fonte;
- A licença do software;
- Diferentes necessidades com relação à interface, "*Não adianta ter uma roda de trator quando se quer trocar a de uma bicicleta*";
- Generalidade, parametrização ou possibilidade de realização de tarefas específicas, em geral uma consequência da modularidade do software;
- Desempenho.

Tabela 2.1: *Análise das linguagens antes da criação da AudioLazy*

Critério	MatLab	Octave	PureData	Python	NumPy
Amostras	Sim	Sim	Depende	Sim	Sim
Blocos	Possível	Possível	Sim	Possível	Possível
Álgebra linear	Sim	Sim	Depende	Possível	Sim
DSP	Sim	Sim	Sim	Possível	Sim
Tempo real			Sim	Possível	
Heterogeneidade				Sim	Possível
Avaliação tardia			Depende	Sim	
Funções de ordem superior				Sim	
Orientação a objetos	Possível	Possível		Sim	Sim
Linguagem de uso geral				Sim	Sim
Documentação	Sim	Sim	Sim	Sim	Sim
Licença	Proprietária	GNU GPL	BSD Modificada	PSFL	BSD
Preço	US$99.00	Gratuito	Gratuito	Gratuito	Gratuito
Código fonte	(Fechado)	C++	C	C, Python	C, Fortran, Python

Entre linguagens voltadas ao processamento de áudio, simulação e pesquisa, cinco deles foram destacados para uma breve comparação sobre suas capacidades para a realização de um trabalho que pudesse auxiliar na pesquisa em processamento de áudio.

Os *softwares* da tabela 2.1 estão melhor detalhados nas subseções que seguem, mas vale salientar que a coluna referente ao NumPy possui como requisito o próprio Python, e que os aspectos presentes na coluna do Python que não constam na coluna do NumPy se referem a possibilidades existentes no Python mas que não são utilizáveis quando a programação se restringe ao uso do NumPy. O NumPy é base para o uso do SciPy e do MatPlotLib, motivo pelo qual um maior detalhamento destes na tabela não foi necessário. Todas as linguagens da tabela foram avaliadas e utilizadas para processamento de áudio antes de se iniciar o desenvolvimento da AudioLazy.

Segue uma breve explicação dos critérios utilizados:

- Amostras: Refere-se à possibilidade de acesso direto às amostras do áudio. Algumas tarefas tais como o processamento não-linear de cada amostra independentemente de suas amostras adjacentes pode ser feita indiretamente através do processamento em bloco, mas há tarefas que se utilizam diretamente das amostras para o processamento, e pode-se exemplificar isso com a implementação de filtros IIR. Tais filtros são detalhados na seção 3.1.

- Blocos: O processamento em blocos refere-se à capacidade do sistema em abstrair sequências finitas de amostras de maneira a utilizá-las como estruturas de dados básicas do som sendo processado. Entretanto, isto pode ser visto como uma restrição quando torna impossível o acesso direto às amostras ou ao instante relativo entre blocos.

- Álgebra linear: Refere-se à possibilidade de realização de tarefas típicas de álgebra linear, tais como a decomposição de matrizes e a resolução de sistemas lineares.

- DSP: Informa sobre a existência de recursos voltados ao processamento digital de sinais, tais como filtros lineares invariantes no tempo, bancos de filtros, processamento não-linear sobre sequências temporais e representação de blocos no domínio da frequência.

- Tempo real: Para a realização de processamento de áudio em tempo real, é necessário se preocupar com alguns itens:

  - Latência (*lag*): Atraso entre os instantes inicial e final de um sinal em um processamento.
  - Variação de atraso (*jitter*): Amplitude do desvio com relação à latência.

  - Interface: Deve haver uma interface com o dispositivo de áudio, tanto para produção como para gravação de som.

  Em um filtro linear, a latência refere-se ao atraso máximo inserido a uma amostra de entrada. No processamento em bloco, a latência refere-se ao tamanho do bloco. No sistema de processamento de sinais digitais como um todo, a latência e a variação de atraso incluem aspectos relativos ao *hardware* e ao *software* utilizados como base, tais como atrasos relativos ao gerenciamento de memória pelo sistema operacional ou atrasos relativos ao processamento realizado pelo *driver* do dispositivo.

  São mantidos o hardware e o sistema operacional na avaliação de todos os itens da tabela. São considerados aspectos necessários para o processamento em tempo real a possibilidade de receber novos blocos de áudio do dispositivo enquanto blocos anteriores são reproduzidos, sem perdas de blocos (*overrun*) ou finalização da reprodução de um bloco antes de um novo bloco ser preparado para a reprodução (*underrun*).

  Chamo de tempo real, no contexto deste trabalho, o sistema que funciona com latência total máxima de 50 milisegundos, isto é, a latência do sistema como um todo, incluindo o tempo de processamento, deve ficar contida nesse intervalo.

- Heterogeneidade: Refere-se ao tipo de dado armazenado nas estruturas, ou ao significado do que é uma amostra. A rigor, não é necessário nem eficiente que cada amostra de áudio seja de um tipo diferente, mas a existência dessa possibilidade permite, por exemplo, o uso de números inteiros para amostras convencionais, e representações alternativas para situações que excedem o limite do número inteiro, a fim de evitar que o valor seja saturado (*clipping*).

- Avaliação tardia: Característica de linguagens de programação funcionais, conforme melhor detalhado na subseção 2.1.3. Segundo Hughes, é uma das caraterísticas da programação funcional que facilita a modularização de sistemas [Hug89].

- Funções de ordem superior: Característica importante da programação funcional [Hud89], refere-se ao uso de funções como valores. Em outras palavras, indica que funções podem devolver funções como resultado, assim como receber como parâmetros outras funções. Segundo Hughes, é uma das caraterísticas da programação funcional que facilita a modularização de sistemas [Hug89].

- Orientação a objetos: Refere-se à possibilidade de se desenvolver em um paradigma orientado a objetos em meio à linguagem, isto é, de forma a interpretar os valores manipulados como objetos munidos de atributos e métodos, generalizados por classes. Outros aspectos incluem a herança, o polimorfismo e o encapsulamento.

  Para o encapsulamento, é suficiente o reconhecimento de um componente de um objeto como privado, sendo desnecessária (e, dependendo da situação, indesejável) a impossibilidade de acessá-lo.

- Linguagem de uso geral: Uma linguagem de uso geral é aquela que não está restrita a uma aplicação específica ou destinada a um único *hardware*. Contrasta com uma DSL (*Domain Specific Language*), tal como SQL que é destinada a bancos de dados relacionais, ou HTML que é destinada especificamente para a escrita de hipertexto.

- Documentação: Existência de tutoriais, glossários, lista de componentes e descrição da sintaxe da linguagem.

- Licença: Embora a linguagem seja uma estrutura sintática abstrata, o *software* compilador ou interpretador que possibilita o uso da linguagem para o desenvolvimento em geral está protegido sob leis de *copyright* ou direitos autorais. Há licenças padrões utilizadas, tais como a GNU GPL e a BSD[12], que referem-se ao software livre no sentido de ter um código fonte aberto e com a possibilidade de utilização e alteração desse código.

[12] Essas e outras licenças de *software* livre podem ser encontradas por completo junto à OSI (*Open Source Initiative*): http://opensource.org/licenses.

Um conceito importante nos *softwares* com código fonte aberto é o de *copyleft*, que se refere à necessidade de abertura de código a todos que dele derivem um novo código. Das licenças listadas na tabela 2.1, essa necessidade é parte da GPL, apenas.

É chamado de *software* livre aquele que possui o código fonte aberto, isto é, disponível para acesso público. Sua antítese é o *software* proprietário, o qual possui o código fechado ou restrito a um grupo de pessoas.

- Preço: Aplicável apenas a *softwares* comerciais.

  Nem todo *software* livre é gratuito, e um exemplo importante é o EPD (*Enthought Python Distribution*)[13], distribuição Python voltada à comunidade científica contendo fundamentalmente pacotes de software livre.

- Código fonte: No caso de código fonte aberto, a linguagem em que o código fonte está disponível. Embora as contribuições deste trabalho não incluam alterações no código fonte de nenhum dos grandes projetos listados na tabela 2.1, estes foram visualizados e parcialmente estudados durante o desenvolvimento da AudioLazy.

2.2.1 MathWorks MatLab

O MatLab é um *software* da Mathworks robusto e bastante completo para simulações em diversas áreas diferentes do conhecimento que envolvem processamento de dados em matrizes, origem de seu nome. Fundamentalmente, trata-se de uma linguagem de alto nível interpretada e voltada ao cálculo numérico. Sua licença é proprietária e seu código fonte é fechado. Seu preço [14] varia conforme as necessidades de um projeto, mas atualmente esse valor não seria inacessível para um estudante bolsista. Embora o código seja fechado, há uma grande quantidade de documentação oficial disponível publicamente, além de pacotes disponibilizados por usuários e fórum de perguntas e respostas[15]. Existem muitos pacotes de *software*, chamados de *toolboxes* no contexto do MatLab quando organizados em pacotes. Entre os disponíveis gratuitamente para uso e de interesse na pesquisa em processamento de sinais, pode-se destacar dois:

- **Auditory toolbox**, de Malcolm Slaney [Sla98][16]. Contém:
  - O modelo auditivo de Richard F. Lyon;
  - O modelo psicoacústico de Roy Patterson através de filtros gammatone, baseado em filtragens em bandas críticas;
  - O modelo em dois estágios de Stephanie Seneff;

[13] Disponível em http://www.enthought.com/, último acesso em 2013-02-14.

[14] O preço do pacote para estudante, versão 2012a é de US$99.00, para um dos sistemas operacionais disponíveis: Windows, Linux e Mac (OS X Lion ou Snow Leopard). O pacote inclui: MATLAB, Simulink, *Control System Toolbox*, *Simulink Control Design*, *Image Processing Toolbox*, *Optimization Toolbox*, *Signal Processing Toolbox*, *DSP System Toolbox*, *Statistics Toolbox* e *Symbolic Math Toolbox*. Este era também o preço para a versão 2010a para estudante, sem *toolkits*, entretanto a MathWorks não inclui em sua lista de produtos a possibilidade de compra de versões antigas para a estudante, exceto *toolkits* para a versão anterior, que no instante de escrita deste texto no caso é a 2011a. Na versão 2012a, o preço de cada *toolbox* extra é de US$29.00. Os valores foram obtidos diretamente do site da MathWorks, empresa criadora do produto, entretanto atualmente é necessário um cadastro para se obter essa informação. Segundo Sharpe, entre as características das versões para estudante do MatLab está a limitação no tamanho das matrizes a 8192 elementos, as quais devem ter não mais que 32 colunas ou linhas. Armazenar um segundo de áudio amostrado a 44100 amostras/segundo nessas condições seria, no mínimo, tortuoso, entretanto a própria MathWorks afirma que não há esse tipo de limitação em seu software. É possível que versões antigas tenham realmente sido realizadas com a citada limitação, porém esta é irrelevante para uma análise atual sobre o software.

MathWorks http://www.mathworks.com/, acesso em novembro de 2010 e em 2013-02-14. Página sobre diferenças entre a versão de estudante e a versão profissional: http://www.mathworks.com/academia/student_version/details.html

Sharpe, William F. "*Macro-Investment Analysis*". Livro texto em HTML, inacabado e sem planos de finalização. URL: http://www.stanford.edu/~wfsharpe/mia/mia.htm. último acesso em 2013-02-14. A sessão que discute o MatLab encontra-se em: http://www.stanford.edu/~wfsharpe/mia/mat/mia_mat3.htm.

[15] Documentação: http://www.mathworks.com/help/

MatLab Central: http://www.mathworks.com/matlabcentral/, último acesso dia 2013-02-16.

[16] Disponível em https://engineering.purdue.edu/~malcolm/interval/1998-010/

  - Análise FFT para representação em espectrogramas;
  - MFCC (*Mel-frequency cepstral coefficients*), coeficientes cepstrais em escala mel (com *rasta*);
  - Analisador LPC (*Linear Predictive Coding*).

- **MIR Toolbox**[17]. Segundo a documentação, contém uma coletânea bastante completa de recursos, incluindo, por exemplo, funções para obtenção de rugosidade, altura e harmonicidade. Está desenvolvido em uma estrutura orientada a objetos, o que faz o pacote de certa forma manter uma relação indireta com as matrizes (estrutura onipresente) no MatLab, entretanto o sistema continua baseado em matrizes e no sistema de filtros do MatLab, utilizando por base o AuditoryToolbox de Slaney.

O próprio Slaney indica em seu relatório técnico [Sla98] a existência de diversos outros pacotes de modelos auditivos, com diferentes filosofias e estilos de código, destacando os modelos AIM (disponível em C, C++ e MatLab, ver seção 2.2.5) e LUTEAR.

Algo em comum no MatLab é a organização em matrizes, na qual filtros lineares denotados por suas equações de sistema[18] são dois vetores (matrizes unidimensionais), um representando o polinômio do numerador e outro representando o polinômio do denominador, e o áudio também é representado por matrizes (ou vetor, caso exista apenas um canal de áudio). Em termos organizacionais, a compreensão é simplificada, e o uso de operações tais como multiplicação de matrizes, obtenção de determinantes, etc. é ideal na linguagem, a qual é voltada para esse fim. Matrizes são homogêneas, e combinação entre matrizes criam novas matrizes bidimensionais, entretanto há uma sintaxe alternativa para matrizes de mais dimensões. Um elemento muito importante do MatLab é o uso de gráficos (*plots*), que permitem parametrizações ao mesmo tempo em que são bastante simples de serem realizados através de vetores ou matrizes contendo seus valores.

Entretanto para o uso junto ao processamento de áudio:

- O filtro não é um par de vetores, e tal representação dificulta a expressividade quando tarefas simples são realizadas, tais como uma multiplicação ou uma exponenciação entre filtros;
- Preocupação com o índice final ao realizar a segmentação do áudio em blocos. Essa é exatamente uma das tarefas realizadas pela MIR Toolbox, embora sob uma estrutura encapsulada e paramétrica (i.e., os objetos não são matrizes);
- O MatLab é feito para processamento matemático, e é extremamente útil em simulações, mas não é voltado ao processamento de áudio em tempo real;
- É uma DSL, isto é, não se trata de uma linguagem de uso geral. Alguém que não tenha o MatLab não tem como executar seus *scripts*, e além do MatLab não ser gratuito, suas exigências quanto ao hardware dificultam a utilização para além da pesquisa científica, exigindo muitas vezes uma reimplementação voltada para a produção do que foi anteriormente desenvolvido como protótipo ou simulação. Seria interessante poder desenvolver uma única vez e permitir que o *software* desenvolvido não se restrinja à comunidade acadêmica.

2.2.2 GNU Octave

Além de *software* livre sob licença GNU GPL, é uma linguagem de alto-nível interpretada e voltada ao cálculo numérico[19] suficientemente similar ao MatLab para permitir até mesmo grande compatibilidade de código. Nem todos os pacotes do MatLab funcionam sobre o GNU Octave, por este possuir algumas diferenças de sintaxe, por exemplo na presença de matrizes multidimensionais.

Em termos gerais, suas vantagens e desvantagens são comparáveis às do MatLab, e idênticas na maioria dos casos, mas algumas das *toolboxes* do MatLab não funcionam no Octave, exatamente por utilizar recursos do MatLab ausentes ou diferenciados no Octave. Para gráficos, o Octave usa internamente o *GNUPlot*, que não se comporta de maneira idêntica ao MatLab mas é satisfatório para a maioria das finalidades envolvendo

[17] Disponível em https://www.jyu.fi/hum/laitokset/musiikki/en/research/coe/materials/mirtoolbox
[18] Há maiores detalhes sobre filtros e seus equacionamentos no capítulo 3.
[19] Disponível em *http://www.gnu.org/software/octave/*

gráficos, incluindo gráficos em 3D. O modelo de decomposição cromática proposto no primeiro apêndice foi escrito em Octave, assim como experimentos diversos quanto à viabilidade de processamento de áudio em bloco. Um recurso importante é o de reprodução sonora através do *Sox*, um aplicativo desacoplado do Octave que é chamado para reproduzir seu som. Entretanto, essa chamada ao *Sox*, além de possuir uma latência relativa à chamada de outro aplicativo, envia vetores com o áudio por completo para serem reproduzidos, o que inviabiliza o processamento em tempo real.

Vale dizer que, analogamente às *toolboxes* que vêm junto ao MatLab, o Octave tem um sub-projeto chamado *Octaveforge*[20] formado por pacotes que em muitos casos replicam as funcionalidades daqueles, porém implementados sob uma licença de *software* livre.

2.2.3 PureData

Também conhecido como Pd, é um ambiente de programação gráfico voltado ao processamento multimídia em tempo real[21]. É um *software* livre desenvolvido em C com versões pré-compiladas disponíveis para Windows, IRIX, GNU/Linux, BSD e Mac OS X.

Seu processamento é realizado através de objetos conectados de maneira gráfica, com uma distinção explícita entre sinais de controle e sinais de dados, inclusive no nome dos comandos, os quais terminam com um caractere de til "~" quando lidam com áudio. Embora existam componentes que realizam processamento por amostra, seu processamento fundamentalmente se baseia em blocos, permitindo manipulações imediatas como multiplicações envolvendo sinais convertidos para o domínio da frequência com o comando "*rfft~*". Em geral, para inserir eficientemente uma nova funcionalidade do Pd, são feitos *externals*, os quais são escritos em outra linguagem. Apesar da ênfase no tempo real, seu processamento é voltado para um uso multimídia imediato, encapsulando o acesso direto aos dados processados.

2.2.4 Python, NumPy, SciPy e MatPlotLib

Python é uma linguagem de alto nível e de uso geral com as seguintes características:

- Compilada e interpretada (existe um *bytecode* Python);
- Tipagem forte e dinâmica;
- Gerenciamento automático de memória;
- Suporte a múltiplos paradigmas, incluindo um estilo estrutural imperativo (paradigma imperativo);
- Orientada a objetos, com sobrecarga de operadores. O encapsulamento é realizado por convenção de nomes, em que o "privado" inicia por um caractere "_" (*underscore*). Vale salientar que tudo, inclusive funções, classes e módulos/pacotes, são objetos em Python (paradigma orientado a objetos);
- Com recursos de, ou inspirados em, programação funcional, incluindo funções de ordem superior, a possibilidade de avaliação tardia, *map-reduce*, *list comprehension* e funções como expressões *lambda* (paradigma funcional);
- Grande biblioteca padrão de pacotes e componentes, além de grande facilidade de obtenção de outros pacotes através do PyPI;
- Multi-plataforma. Códigos escrios para Windows podem ser compilados em executáveis;
- Metaprogramação e reflexão dinâmica (paradigma reflexivo);
- Utiliza a indentação para delimitar os blocos, ao invés de símbolos delimitadores.

[20] Disponível em http://octave.sourceforge.net/
[21] Disponível em http://puredata.info/

Código-fonte 2.3: *PEP-1 O Zen do Python*

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!
```

As informações desta seção foram retiradas diretamente dos endereços oficiais do Python[22] Segundo o site oficial, o Python é uma linguagem de programação que permite ao programador trabalhar mais rapidamente e integrar sistemas mais efetivamente. Você pode aprender a usar o Python e ver quase de imediato os ganhos de produtividade e menores custos de manutenção. Embora essa possa parecer uma descrição de difícil confiabilidade por conta de que dificilmente um site dedicado a uma linguagem denegriria a própria linguagem, o importante é notar que esse é um dos objetivos do desenvolvimento do Python. A disponibilidade do Python para Windows, Linux/Unix e Mac OS X, além da possibilidade de uso sobre as máquinas virtuais Java (Jython) e .NET (IronPython), justificam dizer que o Python é uma linguagem multi-plataforma[23].

O Python é livre para uso, inclusive para produtos comerciais. Está disponível gratuitamente, assim como seu código fonte, sua documentação e todo seu histórico de desenvolvimento, inclusive discussões acerca de propostas de melhoramento, realizadas no grupo de e-mail de desenvolvedores e organizadas através de PEPs (*Python Enhancement Proposals*), indexados através do PEP-0, que contém o *link* para todos os demais PEPs. Mesmo a documentação e os PEPs estão organizadas sob um sistema de controle de versão.

O primeiro PEP, numerado como PEP-1 e conhecido como "Zen do Python", pode ser observado no console da linguagem através do comando "*import this*" (sem as aspas), e expõe diretrizes vistas como objetivos da linguagem, de uma maneira bastante vaga e possivelmente controversa, demostrando certa rigidez ao mesmo tempo em que há uma intenção artística:

Lembrando que cada linha do Zen refere-se ao código fonte, uma tradução possível é:

[22] Último acesso dia 2013-02-16:
Official Website: http://www.python.org/
PEP-0: http://www.python.org/dev/peps/
PyPI: http://pypi.python.org/
Documentação (Python 2.7.3): http://docs.python.org/2.7/
Código fonte (Repositório Mercurial): http://hg.python.org/

[23] Embora não seja um objetivo deste trabalho, é possível também utilizar o Python para programar jogos para o NES (*Nintendo Entertainment System*), embora existam limitações:
Código fonte do PyNES:
https://github.com/gutomaia/pyNES, último acesso dia 2013-02-16
URL da apresentação que será realizada em março no PyCon 2013:
https://us.pycon.org/2013/schedule/presentation/77/, último acesso dia 2013-02-16

"*O Zen do Python, por Tim Peters*

Bonito é melhor que feio.
Explícito é melhor que implícito.
Simples é melhor que complexo.
Complexo é melhor que complicado.
Plano é melhor que aninhado.
Esparso é melhor que denso.
Facilidade de leitura tem valor.
Casos especiais não são especiais o suficiente para quebrarem as regras.
Embora a prática vença a pureza.
Erros jamais devem passar silenciosamente.
A menos que explicitamente silenciados.
Em face à ambiguidade, recuse a tentação de palpitar.
Deve haver uma - e preferencialmente apenas uma - maneira óbvia de o fazer.
Embora tal maneira possa não ser óbvia desde o início a menos que você seja Holandês.
Agora é melhor do que nunca.
*Embora nunca seja frequentemente melhor do que *exatamente* agora.*
Se a implementação é difícil de explicar, é uma má ideia.
Se a implementação é fácil de explicar, ela pode ser uma boa ideia.
Namespaces são uma ideia bombástica - façamos mais desses! "

O interesse na expressividade do código é explícito através de diversas linhas desse curto poema. Os demais PEPs, mais técnicos, contêm informações sobre os recursos da linguagem, alguns dos quais referenciados em notas de rodapé na seção 2.1. Um item que vale mencionar é a última linha, em que "*Namespaces*" são, a rigor, pacotes, e a mensagem serve de motivação ao desenvolvimento de novos pacotes para o Python.

O Python possui diversas implementações, sendo o CPython a implementação de referência escrita em C. Como alternativa que possui destaque, existe o PyPy, implementação do Python escrita em uma versão reduzida do Python com compilador JIT (*Just In Time*), permitindo alto desempenho ao código escrito em Python[24].

Os seguintes containers padrão do Python são hetereogêneos, dinâmicos (tamanho variável) e possuem avaliação imediata:

Tabela 2.2: *Containers do Python heterogêneos com avaliação imediata*

Nome	Construtor	Exemplo por construção direta
Lista	*list*(·)	[1, 2, 3, 1]
Tupla	*tuple*(·)	(1, 2, 3, 1)
Conjunto	*set*(·)	{1, 2, 3}
Dicionário	*dict*(·)	{1: 0, 2: 0, 3: 0}
Fila para ambos os lados	*collections.deque*(·)	(Não há)

Em contraste aos tipos da tabela 2.2, faz parte da biblioteca padrão do Python o pacote *itertools*, um conjunto de ferramentas que lidam com iteráveis ou iteradores, de maneira a permitir o desenvolvimento sob avaliação tardia. Entretanto, não existem operadores definidos em nenhum desses containers além de concatenação em listas e em tuplas:

```
In [1]: [3, 2, 1] * 2
Out[1]: [3, 2, 1, 3, 2, 1]
```

[24] Maiores informações e o pacote disponível para *download* em: http://pypy.org/, último acesso em 2013-02-16.

```
In [2]: [3, 2, 1] * 3
Out[2]: [3, 2, 1, 3, 2, 1, 3, 2, 1]

In [3]: [3, 2, 1] + [1, 7]
Out[3]: [3, 2, 1, 1, 7]
```

NumPy, SciPy e MatPlotLib são três pacotes que, em conjunto, fazem do Python um ambiente voltado ao cálculo numérico[25]. Diferentemente do MatLab, há estruturas separadas para vetores e matrizes bidimensionais, e os vetores podem naturalmente ser N-dimensionais com base na sintaxe de listas de listas. As estruturas *numpy.ndarray* (vetor N-dimensional) e *numpy.matrix* (matriz) possuem comportamentos diferentes com relação às multiplicações, mas é significativamente simples converter uma representação em outra, incluindo conversões de/para listas aninhadas, além de facilitarem a compreensão do significado dos operadores: todo operador sobre vetores é aplicado elemento-por-elemento (*elementwise*) até a última dimensão possível, a partir da qual os valores passam a ser distribuídos (*broadcast*). No MatLab, operações elemento-a-elemento são feitas através de operadores precedidos por um ponto, e essa possibilidade inexiste no Python[26]. Um exemplo dessa característica dos vetores do NumPy pode ser exemplificada com uma sintaxe similar à do MatLab:

```
In [1]: from numpy import array

In [2]: array([1, 5, 2]) * 4 # Distribuir (broadcast)
Out[2]: array([ 4, 20,  8])

In [3]: array([1, 5, 2]) + array([2, 4, 3]) # Elemento a elemento
Out[3]: array([3, 9, 5])
```

O NumPy é um pacote voltado à computação numérica, escrito em C, Fortran e Python, baseado no LAPACK escrito para Fortran 90[27], e inclui, segundo sua *docstring*, fundamentalmente três tipos de recursos:

- Um objeto vetor para itens homogêneos;
- Rápidas operações matemáticas sobre vetores;
- Álgebra linear, transformadas de Fourier, geração de números aleatórios.

Embora seja otimizado para itens/elementos homogêneos, é possível utilizar os vetores do NumPy como heterogêneos, embora isso não seja eficiente. Como alternativa ao NumPy, existem os módulos padrões de matemática os quais trazem ao Python os recursos básicos existentes no C, mas são aplicáveis a apenas um número por vez, em ponto flutuante (*math*) ou com números complexos (*cmath*)[28].

O SciPy possui diversos subpacotes para auxílio ao desenvolvimento na área científica, tal como o subpacote de otimização e de processamento de sinais, que inclui um sistema de aplicação de filtros utilizando

[25] Disponíveis em:
http://www.numpy.org/
http://www.scipy.org/
http://matplotlib.org/

[26] Provavelmente por conta da ambiguidade em expressões do tipo "*1.+b*", em que *b* é um objeto. Seria essa expressão a aplicação do operador binário ".+" ao inteiro 1 e *b*, ou seria o operador "+" aplicado ao número em ponto flutuante 1. (i.e., 1.0) e *b*? Essa ambiguidade é irrelevante no caso do MatLab porque o operador é usado sempre entre matrizes, mas em uma linguagem orientada a objetos que permite sobrecarga de operadores, essa ambiguidade faz diferença.

Vale salientar que o operador de potenciação do Python segue o "**", símbolo léxico também utilizado pelo Fortran, mas que pode confundir usuários do MatLab e do Octave acostumados com o "^", em um primeiro instante.

[27] Antigo porém eficiente pacote contendo implementações de funções de álgebra linear. Disponível em:
http://www.netlib.org/lapack/

[28] Entre diferentes representações de números possíveis, o Python possui nativamente a representação de inteiros, inteiros longos (potencialmente ilimitados), números em ponto flutuante e números complexos. Possui também outras representações em sua biblioteca padrão, tal como números decimais de precisão fixa e frações.

vetores do NumPy análogo ao sistema utilizado no MatLab. O MatPlotLib é responsável pela realização de gráficos (*plots*) a partir de sequências iteráveis de tamanho finito (tais como vetores do NumPy), de maneira personalizável.

2.2.5 Outras linguagens e possibilidades

Há muitas outras linguagens e pacotes de softwares que podem ser utilizadas para processamento de áudio além das supralistadas. Além das linguagens, existem os *plug-ins* Vamp de análise que, embora não sejam uma linguagem nem um pacote de uma linguagem (e muito menos uma DSL), são destinados à obtenção de dados analíticos de música (i.e., MIR).

Outro item que merece destaque é o YAAFE (*Yet Another Audio Feature Extractor*)[29], pacote de áudio voltado à obtenção de diversos tipos de informação a partir do áudio e por processamento em bloco, dado através de uma gravação, de forma a ser incompatível com processamento em tempo real. Está escrito em C++, mas possui adaptações para sua utilização junto com o Python e o MatLab. Não se encontra disponível no PyPI e nem nos repositórios de pacotes das distribuições Debian e Ubuntu do Linux, sendo necessária sua quase manual compilação em C para a utilização junto ao Python. Um aspecto interessante do YAAFE é a documentação disponibilizada em seu *site* feito com o Sphinx, a qual inclui os equacionamentos utilizados e referências bibliográficas.

Retomando o que foi dito na seção 1.3.1 sobre o pacote MARLib de Humphrey, este encontra-se escrito em Python puro e disponível no PyPI, descrito como "uma biblioteca para pesquisa em música e áudio"[30]. O pacote inclui a leitura de arquivos de áudio[31] com armazenamento em vetores do NumPy, cálculo de transformadas como a CQT (*Constant Q Transform*, ou transformada com Q constante), algumas aplicações de transformadas tais como a obtenção de MFCC (*Mel Frequency Cepstral Coefficients*, ou coeficientes cepstrais em escala mel), etc., além de incluir como funções algumas tarefas que foram requisitos às funcionalidades descritas, por exemplo funções de conversão entre as escalas de frequências Hz e mel ($f_{mel} = 2595 \cdot \log_{10}(1 + f_{Hz}/700)$). Apesar de não ser feito para o trabalho em tempo real, esse pacote é um *software* livre (licença LGPL, sem *copyleft*), tem o código-fonte em Python e inclui certas funcionalidades desejáveis em pacotes de processamento de áudio, o que motivou a análise do pacote visando sua atualização para incluir processamento em tempo real, e sua integração com a AudioLazy. Tanto a versão 0.3.4, a versão de desenvolvimento disponível no BitBucket e a versão 0.3.9[32] foram instaladas. O pacote MARLib contém apenas um teste manual, disponível apenas na versão de desenvolvimento presente no repositório do BitBucket. Possui uma versão em Python dos comandos *wavread* e *wavwrite* do MatLab[33]. Existe o aparente intuito de implementar o algoritmo de transcrição polifônica de Klapuri[34], visto que há uma classe *MultiPitchTrack* em uma parte do código de desenvolvimento inexistente na versão 0.3.9 cuja *docstring* referenciava o artigo de Klapuri. A implementação dos filtros *gammatone* inclui apenas o modelo de Klapuri, e sugere ter sido feito de maneira apressada ou sem uma reflexão sobre o significado dos cálculos, visto que seu código-fonte possui passagens trivialmente desnecessárias, tais como o cálculo de um arco-cosseno para imediato cálculo do cosseno posteriormente (função *marlib.timefreq.reson_params*). Ainda sobre resultados obtidos investigando diretamente e

[29] Disponível em http://yaafe.sourceforge.net/

[30] "*A Music and Audio Research Library*", conforme visto no PyPI em 2012-10-17 e em todas as versões obtidas posteriormente.

[31] Processo realizado através de chamadas de sistema ao SoX, aplicativo externo ao Python para reprodução, gravação e conversão entre taxas de amostragem e tipos de arquivos, incluindo algumas rotinas prontas de processamento de áudio tais como mudanças na altura e duração (*pitch shift* e *time stretch*), funcionando como um editor de áudio com interface em texto. Pode ser obtido em:

http://sox.sourceforge.net/, último acesso dia 2012-10-17.

A MARLib utiliza o SoX apenas para a leitura de arquivos de áudio, e, embora o SoX esteja disponível para Windows, a MARLib não acessa o SoX nesse sistema operacional, sendo essa funcionalidade restrita ao Linux e sistemas operacionais similares.

[32] A versão de desenvolvimento é anterior à versão 0.3.9 disponível no PyPI, o que sugere um abandono do repositório público presente no BitBucket. Há maiores informações sobre as datas na nota de rodapé 14, localizada na seção 1.3.1.

[33] Há pelo menos um porém: o NumPy não trabalha com inteiros de 24 bits, formato que pode ser utilizado em arquivos WAV/PCM contendo áudio para fins artísticos. A AudioLazy, por não depender do NumPy e por não impor o uso de amostras homogêneas, não possui essa restrição, mas necessita um pacote que possa ler tais arquivos, tal como o pacote *wave* da biblioteca padrão do Python.

[34] Filtro *gammatone* e algoritmo definidos em [Kla08]. O filtro *gammatone* foi implementado na AudioLazy de maneira completamente independente da MARLib, e há maiores detalhes sobre o mesmo na seção 3.2.8).

exclusivamente o código-fonte do pacote, foi possível inferir que os filtros foram organizados como um par de matrizes em que cada linha representa os coeficientes de um ressonador, sendo uma matriz de numeradores e outra de denominadores. Embora marcada com o status *beta*[35], as duas primeiras versões não eram sequer importáveis, precisando de pequenas alterações no código para possibilitar tal tarefa. Dificuldades extras nessa tentativa de utilizar a MARLib, incluindo correções para possibilitar o uso da funcionalidade que implementa os filtros utilizando a função *scipy.signal.lfilter* internamente, demonstraram que há erros no código em partes fundamentais deste, o que tornou inviável tanto seu uso como a avaliação da implementação do modelo de transcrição de Klapuri. Apesar de ser anti-intuitivo ou inexpressivo representar 4 objetos (filtro ressonadores) em 2 estruturas em que cada uma contém apenas uma parte desses objetos, essa representação indica que a organização do filtro *gammatone* foi realizada seguindo a descrição de Klapuri como uma cascata de filtros, um fato importante para a estabilidade numérica do resultado[36]. Mesmo o *site* da MARL[37] não contém nenhuma informação sobre o pacote MARLib, o que sugere o reconhecimento de que o desenvolvimento ainda não é considerado qualificado para publicação. Dada a incompatibilidade de objetivos e a qualidade do código atual da MARLib, foi abandonada a intenção de se integrar os pacotes.

Wolfram Mathematica

Esse é um *software* proprietário e uma linguagem que não foi utilizada diretamente no desenvolvimento do trabalho, entretanto o Mathematica foi utilizado por Slaney [Sla93] e possui recursos para processamento de áudio[38], o que torna sua citação meritória no presente trabalho. Diferente do enfoque numérico do MatLab e do Octave, o enfoque do Mathematica é principalmente voltado ao processamento matemático simbólico, algo conhecido como CAS (*Computer Algebra System*). Há uma interface *web* que possibilita a realização de tarefas simples com o Mathematica[39]. Realizando uma verificação com o site apenas para ilustração, foi usado como entrada a expressão "`a * b ^ 2 + b * c ^ 2 = 0`", a qual resultou em uma representação alternativa, uma lista de raízes (simbólica, em cada uma das variáveis, incluindo $c = \pm\sqrt{a * b}$), e derivadas parciais.

Exatamente esse tipo de recurso está disponível no Python através do pacote *Sympy*[40], o qual foi utilizado durante a implementação dos filtros da AudioLazy, para a obtenção de equações analíticas envolvendo o ganho e a largura de banda de filtros ressonadores (seção 3.2.8). Há também um exemplo de integração da AudioLazy com o SymPy na seção 3.2.4.

C, C++ e Java

Como indicado por Slaney [Sla98], o pacote com o modelo de Patterson em C, e atualmente em C++ incluindo uma GUI (*Graphical User Interface*, interface gráfica com o usuário) está disponível no pacote AIM (*Auditory Image Model*), disponível em:

http://www.pdn.cam.ac.uk/groups/cnbh/research/aim.php

Embora inegavelmente existam muitos pacotes de processamento de áudio desenvolvidos para essas linguagens, estas não foram consideradas para o desenvolvimento do pacote AudioLazy, por conta da expressividade, da tipagem estática e da distância com relação ao paradigma funcional de programação.

DSL para processamento de áudio

Assim como o PureData, há outras linguagens voltadas ao processamento multimídia, ou especificamente ao processamento de áudio.

[35] Refere-se às etapas de desenvolvimento do *software*. Versões *alpha* são as disponibilizadas para testes por usuários finais em ambientes controlados, versões *beta* são disponibilizadas para o próprio ambiente do usuário, ainda em estágio de avaliação.

[36] Assunto melhor tratado na seção 3.2.5.

[37] http://marl.smusic.nyu.edu, último acesso em 2013-02-18.
Foi utilizado o sistema de busca do próprio *site*, indicando zero ocorrências.

[38] http://reference.wolfram.com/mathematica/tutorial/AudioProcessing.html, último acesso em 2013-02-07.

[39] http://www.wolframalpha.com, último acesso em 2013-02-14.

[40] Disponível em http://sympy.org/en/index.html
A versão utilizada foi a 0.7.1.rc1.

FAUST (*Functional AUdio STream*)[41] é uma linguagem que certamente merece destaque. A descoberta tardia da linguagem impediu que uma análise apropriada da mesma pudesse ter sido feita antes do desenvolvimento do que foi o início do pacote AudioLazy. As capacidades do Faust não são negligenciáveis, no sentido de este ser uma linguagem que permite programação funcional de áudio e conversão para posterior utilização em diversos ambientes diferentes, incluindo o PureData, além de possuir um sistema de geração automática de documentação. Um exemplo de sintaxe amigável sobre a implementação de filtros nessa linguagem pode ser encontrado em[42]:

```
process = firpart : + ~ feedback
with {
  bw = 100; fr = 1000; g = 1; // parameters - see caption
  SR = fconstant(int fSamplingFreq, <math.h>); // Faust fn
  pi = 4*atan(1.0);    // circumference over diameter
  R = exp(0-pi*bw/SR); // pole radius [0 required]
  A = 2*pi*fr/SR;      // pole angle (radians)
  RR = R*R;
  firpart(x) = (x - x'') * g * ((1-RR)/2);
  // time-domain coefficients ASSUMING ONE PIPELINE DELAY:
  feedback(v) = 0 + 2*R*cos(A)*v - RR*v';
};
```

Embora pareça críptica, a realidade é que há elementos da notação que são bastante claros, tais como o uso do apóstrofo para representar o atraso de uma amostra sobre o sinal x, e a parte FIR do filtro (*firpart* no código) indica que este é um dos filtros presentes na AudioLazy, o filtro ressonador com um par de zeros em -1 e 1, implementado na AudioLazy como:

Código-fonte 2.4: *Código-fonte do audiolazy.lazy_filters.resonator.z_exp*

```
1 def resonator(freq, bandwidth): # Valores em rad/amostra
2   bandwidth = thub(bandwidth, 1) # Largura de banda
3   R = exp(-bandwidth * .5) # Raio (magnitude do polo)
4   R = thub(R, 5) # Número de cópias de R usadas
5   cost = cos(freq) * (1 + R ** 2) / (2 * R) # Cosseno de A (ou theta)
6   gain = (1 - R ** 2) * .5 # Ganho
7   numerator = 1 - z ** -2 # Numerador
8   denominator = 1 - 2 * R * cost * z ** -1 + R ** 2 * z ** -2 # Denominador
9   return gain * numerator / denominator
```

Esse exemplo cria uma comparação entre a expressividade do Faust e da AudioLazy no Python. Talvez seja confuso pensar no atraso da parte de retroalimentação (denominador do filtro) olhando para o código do Faust com o zero somado no lugar da unidade, valores negativos e um atraso implícito (última linha do bloco *with*) quando comparado com o denominador $1 - 2Rcos(A)z^{-1} + R^2z{-}2$ do ressonador representado (equivalente à linha 8 do código da AudioLazy fornecido), mas a sintaxe da linguagem Faust não apresenta uma grande dificuldade à compreensão, como pode-se observar no restante do código, que contém basicamente equações e definições em notação infixa. Nesse aspecto, pode-se observar similaridade entre a escrita dos códigos em Python utilizando a AudioLazy e em Faust fornecidos para a comparação, a menos da parte de retroalimentação, na qual o Faust se baseia no equacionamento do filtro no domínio do tempo (o que justifica os valores negativos e a ausência da unidade na linha destacada) enquanto a AudioLazy se baseia na representação do filtro através de transformadas Z.

A AudioLazy exige reconhecimento do número de vezes que um dado sinal é utilizado apenas para permitir que o filtro seja variante no tempo, isto é, tal linha seria desnecessária caso a largura de banda (*bandwidth*) fosse um número fixo (o que é o caso no particular exemplo em Faust, o valor foi fixado em 100Hz, com

[41] Disponível em http://faust.grame.fr/

[42] https://ccrma.stanford.edu/realsimple/faust/Simple_Faust_Program.html, último acesso em 2013-02-16.

frequência central em 1kHz). Outra diferença é que, por padrão, todos os filtros da AudioLazy são implementados usando unidades independentes da taxa de amostragem, tais como amostras (tempo) e radianos por amostra (frequência), ao invés de valores em hertz como o exemplo fornecido. Maiores detalhes sobre filtros no capítulo3.

Outras linguagens que merecem ser citadas aqui, embora não havendo necessidade de um maior aprofundamento sobre elas, são o CSound, o SuperCollider e o Chunk.

2.3 Fundamentação da AudioLazy

AudioLazy é um pacote expressivo para processamento digital de sinais (DSP) em Python. As duas preocupações centrais ao desenvolvimento são a expressividade (como consta no *slogan*) e a avaliação tardia (como consta no nome do pacote). Um resumo da AudioLazy é dado a seguir, com um detalhamento de aspectos fundamentais e de origem da AudioLazy.

2.3.1 Pacote DSP expressivo para o Python

Dos softwares analisados e utilizados para processamento de áudio conforme consta na tabela 2.1, o mais satisfatório foi o Python, que conta com recursos similares aos do MatLab quando junto aos pacotes NumPy, SciPy e MatPlotLib. Mas esses pacotes possuem uma expressividade voltada para manipulação de vetores (*arrays*) e matrizes em simulação numérica, não para processamento de áudio em tempo real.

Visando solucionar isso, almeja-se realizar os cálculos através de avaliação tardia, cuja ideia central é a de possibilitar:

- Aplicação em tempo-real (não é necessário aguardar todos os dados serem processados para ter um resultado);
- Representação de sequências sem final definido;
- Representação de fluxos de dados;
- Eliminação de tarefas quando uma tarefa inversa é realizada.

É difícil criar código expressivo para processar áudio em blocos através de índices e vetores com a necessidade de sempre se preocupar com a indexação. Por vezes, essa preocupação é inevitável, e remover essa possibilidade limitaria o potencial do sistema. Sequências de blocos podem ser obtidas a partir de sequências de amostras quando ambos são objetos, em que o segundo é o resultado de um método do primeiro, ou o equivalente com uma função ao invés de um método, que requer como informação apenas o tamanho do bloco e o salto em amostras entre blocos adjacentes (*hop*). Embora se possa imaginar a quantidade exata de blocos de uma sequência finita ou o índice do último bloco, a maior parte do tempo gasto nessas tarefas é um desperdício com questões de implementação que mantém a atenção distante do problema sendo trabalhado. Para possibilitar uma sequência sem final definido, não deveria ser necessário conhecer a finalização de sequências.

Um engenheiro provavelmente tem maior familiaridade com equações e estruturas das teorias utilizadas em engenharia elétrica do que com o armazenamento generalizado em vetores, particularmente quando operações sobre essas representações são necessárias. Qual seria o produto de um filtro cujo numerador é representado por $[1, 7, 2]$ e denominador é representado por $[1, 0.5, 0.2]$ como sua equação do sistema, com outra que possui os vetores retrogradados como $[2, 7, 1]$? Pode ser simples, e a retrogradação evitaria questionamentos do tipo "quem vem antes, o zero ou o expoente -2? Mas talvez seja mais eficiente fazer algo mais simples: a multiplicação poderia ser escrita uma vez por definitivo, e com uma representação usual. Seria ainda mais expressivo se a multiplicação se livrasse da assimetria de uma chamada de um método como *filtro1.multiplicar_por(filtro2)*, dado que a multiplicação nesse caso deveria ser comutativa. A sobrecarga de operadores é possível em Python, mas para isso é necessário antes descrever essas equações e estruturas como objetos e relações entre objetos.

Uma das coisas mais simples que se pode imaginar é a utilização de constantes como intermediárias para auxiliar nos cálculos, mas estas também podem ser utilizadas para aumentar a expressividade do código. O nome "Hz" pode ser utilizado como um número que converte a frequência em hertz para radianos por amostra, uma representação interna do DSP que permite tornar suas funções independentes da taxa de amostragem ao mesmo tempo que faz o código incluir expressões do tipo "freq = 220 * Hz". Provavelmente não há dificuldade alguma em se criar tais constantes em provavelmente qualquer linguagem. Podemos definir um filtro "comb.tau(delay=30*s, tau=40*s)" que representa um filtro *comb* com o dado atraso (expoente do denominador, a menos do sinal) e constante de tempo (intervalo entre cada amostra do sinal e o cruzamento com o zero de sua derivada no mesmo ponto), de maneira que ambos os argumentos são passados como valores em amostras, mas convertidos por uma constante de conversão que visa tornar mais claro o código criado, além de ter mais significado para potenciais leitores do código do que quando comparados com uma expressão como "[1] + [0] * 239999 + [alpha]". Por que calcular todos esses zeros se o objetivo do filtro for apenas obter um gráfico de resposta em frequência? A representação do filtro como um par de vetores, um para o numerador e um para o denominador, utilizará desnecessariamente espaço em casos esparsos como o exemplificado, sendo que para o cálculo de resposta em frequência é suficiente atribuir $z = e^{j.\omega}$ na formulação da transformada Z, que possui apenas dois coeficientes no denominador.

É possível evitar alguns desses problemas com tais constantes, tipagem dinâmica, sobrecarga de operadores, funções de ordem superior, orientação a objetos em simultâneo com programação funcional, entre outros. Esses recursos o Python nos fornece livremente.

Nativamente ao Python, há um pacote chamado *wave*, o qual contém uma interface com arquivos de áudio, que junto com o módulo *struct* possibilita o acesso ao conteúdo de arquivos de áudio. Dessa forma, não há necessidade de grandes preocupações com relação ao processamento de arquivos de áudio, bastando a possibilidade de conversão entre os formatos do Python. Para o processamento de cada bloco, o NumPy possui as ferramentas necessárias (FFT, álgebra linear), sendo necessário principalmente recursos para processamento por amostra e para conversão do sinal em blocos.

2.3.2 Resumo dos recursos disponíveis

AudioLazy é um pacote escrito em puro Python que, de maneira não exaustiva, inclui:

- Uma classe *Stream* (fluxo de informação) para representação de sinais de duração limitada ou ilimitada, heterogêneos, iteráveis, com operadores aplicados elemento-a-elemento, mas aplicado a todos os elementos quando um dos operandos não é um iterável (mesmo comportamento dos vetores do NumPy);
- Representação orientada a amostras (classe Stream) com fácil conversão em blocos utilizando o método Stream.blocks(size, hop);
- Processamento interativo (por amostra) através da classe *ControlStream*;
- Mixagem de iteráveis dados os intervalos de tempo entre o início de cada iterável (classe *Streamix*);
- Entrada e saída de áudio multi-thread e integrada com o PyAudio;
- Filtragem linear com filtros utilizando a equação de sistema dos filtros (transformada Z) diretamente (e.g. filt = 1 / (1 - .3 * z ** -1));
- Filtros lineares variantes no tempo através do uso de objetos Stream como coeficientes;
- Filtros em paralelo e em cascata, com comportamento equivalente ao de uma lista de filtros;
- Filtros ressonadores, *comb*, passa-baixas, passa-altas e três diferentes implementações de filtros gammatone lineares;
- Integração com o MatPlotLib para a realização de gráficos de resposta em frequência de filtros e diagramas de zeros e polos;

- Codificação preditiva linear (LPC), diretamente como objetos ZFilter representando filtros de análise, a partir dos quais é possível obter coeficientes PARCOR e LSFs (*Line Spectral Frequency*), incluindo testes de estabilidade do filtro por meio desses valores;
- Ferramentas de análise e processamento tanto voltado para a representação em sequência de amostras (e.g. taxa de cruzamentos por zero, envoltória, média móvel, *clip*, *unwrap*) como para cada bloco (e.g. funções de apodização ou janelamento, DFT para frequências específicas, autocorrelação e matriz de atrasos);
- Um simples sintetizador (consulta a tabela e modelo de Karplus-Strong generalizado) com ferramentas de processamento (envoltória ADSR linear, *fade in/out*, sequência finita por interpolação linear entre um par de valores e uma duração), e gerador de sinais básicos (senoide, ruído branco, impulso);
- Modelo auditivo de parte do aparato auditivo periférico (ERB e filtros gammatone);
- Organização da capacidade de implementação de múltiplas estratégias através de dicionários de estratégias, instâncias de *StrategyDict*. Isto é utilizado em parte significativa do pacote, incluindo os dicionários *erb*, *gammatone*, *comb*, *lpc*, *resonator*, etc.;
- Conversores entre representações de alturas em números MIDI, texto como "F#4" e valores de frequência em hertz;
- Polinômios, fundamentais à existência de filtros;
- Polimorfismo aplicado aos componentes dos módulos *itertools*, *math* e *cmath* de forma a tornar suas funcionalidades baseadas em objetos Stream;
- Comparação por aproximação entre números e iteráveis contendo números.

Para uso parcial do pacote AudioLazy, não há necessidade de nenhum pacote além dos que já estão inclusos na biblioteca padrão do Python. O MatPlotLib é necessário apenas nos métodos de plotagem de filtros. O NumPy é necessário para a obtenção de raízes de polinômios, o que inclui indiretamente o cálculo de zeros, polos e LSFs de filtros. Por esse motivo, a verificação de estabilidade de filtros pela alternância das LSFs e a capacidade de geração de diagramas de zeros e polos também necessitam do NumPy. Há também dependência com relação ao NumPy em resolução de sistemas lineares para obtenção dos coeficientes LPC nas estratégias "*nautocor*" (autocorrelação com o NumPy) e "*covar*" (covariância com o NumPy), além de possivelmente na estratégia "*autocor*", que privilegia a utilização do algoritmo de Levinson-Durbin para obtenção dos coeficientes devido à complexidade computacional reduzida na resolução de sistemas lineares baseados em matrizes Toeplitz simétricas (quadrática ao invés de complexidade cúbica da eliminação de Gauss-Jordan). Finalmente, é necessário o PyAudio para a reprodução de arquivos de áudio. Outras dependências, fornecidas oportunamente no texto, referem-se aos testes (seção 2.3.7), à compilação da documentação oficial (seção 2.3.8) e aos exemplos (seção 2.3.9). O importante é salientar que todo o núcleo da AudioLazy é independente do NumPy, e disponível para uso. Graças à dinamicidade do Python, o NumPy e o MatPlotLib são importados apenas quando requisitada uma funcionalidade da AudioLazy que deles necessita, fazendo até mesmo a importação ser tardia, e possibilitando o uso em ambientes sem tais pacotes (e.g. não há versão do MatPlotLib disponível para o PyPy).

O pacote AudioLazy funciona tanto em Linux como em Windows, e é esperado que funcione em outros sistemas. Com o objetivo de facilitar a instalação por parte do usuário, a última versão estável encontra-se no PyPI, em um arquivo no formato *audiolazy-versao.tar.gz* que pode ser instalado com um dos dois seguintes comandos, após descompactado (com "*tar -xvzf*"); não se deve utilizar os dois comandos abaixo, basta um:

```
$ [sudo] python setup.py install
$ [sudo] pip install .
```

Com o *pip*, a instalação pode ser feita diretamente, incluindo o *download* do pacote no PyPI (utilizando "-U", para atualizar a versão existente ou reinstalar)

```
$ [sudo] pip install audiolazy
```

Versões específicas podem ser instaladas diretamente do repositório. Na ausência de indicação de versão (i.e., sem o conteúdo a partir do "@", inclusive), a última versão (na ramificação principal) será instalada:

```
$ [sudo] pip install -U git+git://github.com/danilobellini/audiolazy.git@v0.03
```

2.3.3 Classe Stream

Sobrecarga de operadores (*operator overload*) é a possibilidade de atribuir diferentes significados aos operadores, através da implementação de suas funcionalidades com tipos de parâmetros específicos. Apesar do uso ingênuo permitir códigos extremamente crípticos por se afastar do leitor do código quanto ao significado intuitivo dos operadores, a possibilidade de sobrecarga deles é uma ferramenta poderosa em prol da expressividade. Isso pode ser notado, por exemplo, ao comparar pares de números. Sejam $X = (x_0, x_1)$ e $Y = (y_0, y_1)$ dois pares. Podemos compará-los através de uma expressão grande como "x_0.igual_a(y_0) e x_1.igual_a(y_1)", mas se pudermos dizer "$x_0 == y_0$ e $x_1 == y_1$", seria mais claro, dado que se pressupõe que a igualdade seja simétrica e a chamada de um método mantendo um elemento dentro e outro fora do parêntesis cria uma assimetria visual de densa compreensão, dificilmente imediata. Caso o operador de igualdade possa ser sobrecarregado, até mesmo a curta expressão $X == Y$ poderia ser usada, expressando adequada e diretamente a ideia de que o valor obtido é verdadeiro apenas quando os pares são iguais. O operador de igualdade, sua negação e os comparadores (maior, menor ou igual, etc.) são todos sobrecarregados na classe Stream, e possuem o mesmo comportamento elemento-a-elemento de vetores do NumPy. Isso significa que, sendo sig um sinal, a expressão $sig > 3$ devolve um sinal de booleanos verdadeiro ou falso, os quais podem ser vistos como variável indicadora junto a números, como se esses valores fossem 1 e 0. Essa facilidade permite que expressões muito curtas e expressivas sejam utilizadas para descrever um dado processamento desejado.

Um dos itens que motivou o início do desenvolvimento foi o da obtenção de uma representação do áudio como um fluxo de informação que permitisse seu processamento independentemente de sua duração ou preocupação com índices sequenciais absolutos, os quais necessariamente limitariam a duração total representável (indexando cada amostra a uma taxa de amostragem de 44100 amostras/s com inteiros de 32 bits sem sinal, após 27 horas, 3 minutos e 12 segundos já teríamos estourado a representação e requisitado mais um bit).

Tal restrição não foi a única: para representar áudio como um objeto autônomo (i.e., uma abstração) correspondente a um fluxo de dados, e para permitir a existência de uma estrutura que esteja preparada para acessar dados que inexistem no presente (entrada de áudio) e assim permitir o processamento em tempo real, é necessária a avaliação tardia, elemento de fundamentação da classe Stream.

Entretanto, já existiam no Python os geradores e as funções do módulo *itertools* que realizavam essa tarefa. Porém, sua sintaxe dificultava a aplicação de operadores diretamente aos geradores, e havia a intenção de manter a interface de operadores análogos aos dos vetores do NumPy: funcionando elemento por elemento em pares, ou aplicado a todos os elementos quando não se trata de dois iteráveis. De outra forma, um dos objetivos era o de permitir a soma, assim como outras operações, diretamente entre os iteráves:

Código-fonte 2.5: *Soma de dois sinais periódicos*

```
In [1]: from audiolazy import Stream

In [2]: dados = Stream(5, 7, 1, 2, 5, 3, 2) # Periódico

In [3]: dados2 = Stream(0, 1) # Idem

In [4]: (dados + dados2).take(15)
Out[4]: [5, 8, 1, 3, 5, 4, 2, 6, 7, 2, 2, 6, 3, 3, 5]
```

O método "*take*(n)" foi originalmente criado para auxiliar nos testes e permitir uma compreensão rápida do que está se passando, devolvendo os primeiros n valores do objeto. Para o processamento em bloco recomenda-se o uso do método "*blocks*($size$, hop)", cujos parâmetros são precisamente o tamanho do bloco e o tamanho do hop, devolvendo um Stream de blocos como filas para ambos os lados (i.e., objetos *collections.deque*; a

rigor, sempre o mesmo objeto deque, a fim de evitar excessivo gasto de tempo com alocação de memória, principalmente no caso em que o passo "*hop*" é menor que o tamanho "*size*" do bloco). Pode-se perceber nesse uso da mesma classe Stream para transmitir amostras e blocos de amostras que a classe Stream consiste em um container heterogêneo.

Vetores unidimensionais do NumPy também podem ser utilizados diretamente junto à classe Stream e somados. O construtor da classe Stream quando recebe iteráveis devolve um objeto Stream que apenas percorre a concatenação dos iteráveis, e quando recebe não-iteráveis, devolve um objeto periódico que varre circularmente os valores fornecidos. Em Python, existe uma sintaxe que permite "abrir" um iterável finito através do uso do operador unário "*", como se este contivesse todos os argumentos posicionais que seguem à chamada:

Código-fonte 2.6: *Construtor para sinais periódicos e sinais finitos*

```
In [5]: Stream([2, 3, 4]).take(5) # Lista de entrada
Out[5]: [2, 3, 4]

In [6]: Stream(2, 3, 4).take(5) # Números de entrada
Out[6]: [2, 3, 4, 2, 3]

In [7]: Stream(*[2, 3, 4]).take(5) # Lista com "*"
Out[7]: [2, 3, 4, 2, 3]
```

Um decorador em Python em seu caso mais simples refere-se a uma função que recebe como parâmetro uma função, devolvendo outra função, normalmente adicionando um novo recurso à função original. Segue abaixo um exemplo de um decorador que ordena os parâmetros posicionais de uma função antes de chamá-la. A linha comentada indicando a sintaxe de decorador do Python (início com um caractere "@") é equivalente a colocar "`subtrai = ordena_entradas(subtrai)`" após a definição da função.

Código-fonte 2.7: *Exemplo de decorador em Python*

```
In [1]: def ordena_entradas(func):
   ...:   def func_nova(*args): # args (sem *) é uma tupla
   ...:     return func(*sorted(args))
   ...:   return func_nova
   ...:

In [2]: @ordena_entradas # Decorador
   ...: def subtrai(a, b):
   ...:   return a - b
   ...:

In [3]: subtrai(5, 7)
Out[3]: -2

In [4]:

In [4]: subtrai(7, 5)
Out[4]: -2

In [5]:

In [5]: subtrai(-2, -22)
Out[5]: -20
```

O exemplo dado mostra uma função (*ordena_entradas*) que define outra (*func_nova*) e a devolve. A função *sorted* do Python recebe um iterável finito e devolve uma lista com seus elementos. Utilizando o recurso de decorador, as funções geradoras do Python podem ser facilmente convertidas em funções que devolvem objetos Stream, através do decorador *tostream*:

Código-fonte 2.8: *Decorador tostream*

```
In [1]: from audiolazy import tostream

In [2]: @tostream
   ...: def impulse():
   ...:   yield 1
   ...:   while True:
   ...:     yield 0
   ...:

In [3]: impulse # De fato, uma função
Out[3]: <function __main__.impulse>

In [4]: impulse() # Devolve um objeto Stream
Out[4]: <audiolazy.lazy_stream.Stream at 0x30824d0>

In [5]: impulse().take(5)
Out[5]: [1, 0, 0, 0, 0]

In [6]: (impulse() + 1).take(5) # Outro objeto instanciado
Out[6]: [2, 1, 1, 1, 1]
```

Dessa forma, a mesma sintaxe habitual dos que programam em Python pode ser utilizada para a criação de objetos Stream quaisquer, os quais possuem seus operadores sobrecarregados, em contraste com os geradores.

A classe Stream nada mais é que uma aplicação de um polimorfismo sobre qualquer iterável em Python, incluindo a tupla de argumentos posicionais, quando este for o caso (i.e., não se trata de um polimorfismo de subtipo, típico da orientação a objetos). Entretanto, para permitir o funcionamento de todos os 35 métodos relativos aos operadores de maneira a facilitar a manutenibilidade do código, tornou-se necessário uma forma de agrupar todos esses operadores em um lugar único e confiável. A rigor, são três lugares, dado que existem operadores unários e binários, e os binários podem ser aplicados reversamente, o que tem uma diferença para o Python na resolução das chamadas e verificação dinâmica de tipos. Primeiramente o operador binário do objeto à esquerda do operador é chamado, e se este devolver o objeto "*NotImplemented*" ou não existir, o operador reverso do objeto à direita é chamado. Essa criação massiva dos 35 métodos exigiu a criação de um componente que permitisse a criação eficiente dos mesmos, evitando a repetição de código, o que culminou no que é hoje a metaclasse abstrata AbstractOperatorOverloaderMeta.

2.3.4 Sobrecarga massiva de operadores com AbstractOperatorOverloaderMeta

Metaclasse é uma classe cujas instâncias são classes. O uso de metaclasses não é muito comum, e recomenda-se que iniciantes evitem utilizar[43]. A tarefa que caracteriza uma metaclasse pode ser vista como a mesma de uma classe, exceto pelo fato de que suas instâncias são classes. Assim como o construtor de uma classe pode inicializar o objeto com um certo conjunto de valores iniciais, a metaclasse pode fazer o mesmo inicializando a classe com um conjunto de atributos ou métodos. É o que acontece com a classe Stream: o construtor em sua metaclasse *StreamMeta* instancia seus 35 operadores como funções, e "inicializa" a classe Stream com esses operadores.

Existem alternativas para isso. Um simples decorador pode instanciar os operadores na classe, ou uma rotina executada durante a importação do pacote, porém após a criação da classe Stream. De fato, essa última possibilidade chegou a ser a maneira como a classe Stream tinha seus operadores criados. Porém,

[43] Presente no capítulo 2 do livro online:
Chaturvedi, Shalabh. *Python Types and Objects*. Cafepy.com
http://www.cafepy.com/article/python_types_and_objects/index.html, último acesso dia 2012-10-16.
O livro, embora curto, contém informações detalhadas e completas sobre a estrutura de tipos, classes e objetos em Python. A citação realizada foi:
"*Pergunta: Quando se deve utilizar metaclasses?*
Resposta: Nunca (enquanto você estiver se perguntando isso)"

outras classes também poderiam se utilizar da possibilidade de uma sobrecarga massiva de operadores o que justifica a necessidade de modularização da capacidade de instanciação massiva de operadores em classes. No diagrama de classes da AudioLazy exposto na figura 2.1, é possível observar que existem 5 classes diferentes que terminam com o nome "Meta", além da que leva o título desta seção. São todas metaclasses, e formam uma hierarquia de metaclasses com a AbstractOperatorOverloaderMeta.

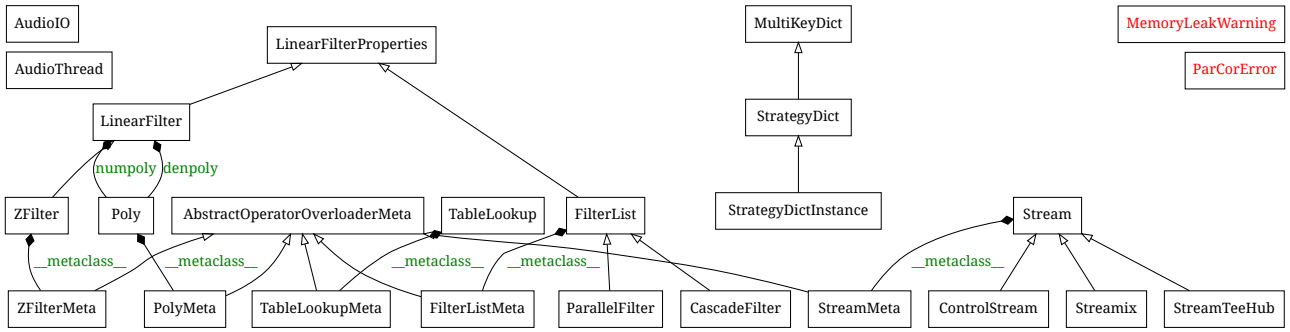


Figura 2.1: *Diagrama de classes da AudioLazy. Gerado a partir do código com o pyreverse, parte do pylint. As associações marcadas com __metaclass__ são relações entre metaclasse e classe. São associações devido à maneira como o vínculo é realizado no Python 2.x.*

Uma classe é dita abstrata quando possui recursos especificados, mas sem uma implementação, e não permitem instanciação na ausência de tais implementações, realizadas por seus descendentes. Isso ocorre na AbstractOperatorOverloaderMeta no sentido em que ela apenas fornece os recursos para a massiva sobrecarga de operadores, mas espera que seu descendente forneça os três *templates* de métodos de operadores (binário direto, binário reverso e unário). Cada uma das classes da figura 2.1 que herdam de AbstractOperatorOverloaderMeta têm o papel de criar as versões de cada um desses *templates*, além de indicar quais são os operadores que devem ser sobrecarregados/inseridos na classe. Apesar do nome grande e talvez complicado, este não passa de uma descrição sucinta de sua funcionalidade e classificação.

2.3.5 O dicionário de estratégias StrategyDict

Existem apenas três nomes no módulo que representa o núcleo da AudioLazy. Um deles é a recém-discutida classe *AbstractOperatorOverloaderMeta*. Outra é o dicionário de estratégias, a classe *StrategyDict*, assunto desta seção, mas existe outra classe, o dicionário multi-chaves, representado através da classe *MultiKeyDict*, que é necessário ilustrar antes, visto que é dessa classe que a classe StrategyDict deriva partes importantes de sua funcionalidade, por herança.

Um dicionário nada mais é que um container associativo, ou seja, um mapeamento entre chaves e valores. Uma generalização à ideia do dicionário é a possibilidade de utilizar não apenas uma única chave identificadora a um valor, mas um conjunto de chaves. Fundamentalmente, nada muda, visto que não há nada que proíba o dicionário de ter mais de uma ocorrência de chave associada a um mesmo valor. O inverso exigiria uma forma de representação do valor como conjunto, mas não é o caso. Em Python, o dicionário é um iterável com relação às suas chaves, isto é, este se comporta como o conjunto de suas chaves quando utilizado em uma função que apenas espera um iterável. Usando a *sorted* do Python para obter uma lista ordenada a partir de um iterável, podemos observar esse comportamento:

Código-fonte 2.9: *Comportamento do dicionário em Python*

```
In [1]: dicionario = {1:4, 2:5, -7:4}

In [2]: sorted(dicionario)
Out[2]: [-7, 1, 2]

In [3]: sorted(dicionario.itervalues())
Out[3]: [4, 4, 5]
```

```
In [4]: len(dicionario) # Verifica o tamanho
Out[4]: 3
```

É importante notar que, embora seu comportamento seja o de um conjunto com relação às chaves, os valores podem aparecer mais de uma vez: o que determina o tamanho do dicionário é o número de chaves, não o de valores distintos. Como uma alternativa a essa representação, a classe *MultiKeyDict* da AudioLazy implementa um dicionário alternativo, iterável por seus valores mas consultável normalmente através de suas chaves:

Código-fonte 2.10: *O dicionário multi-chave MultiKeyDict*

```
In [1]: from audiolazy import MultiKeyDict

In [2]: dicionario = MultiKeyDict({1:4, 2:5, -7:4})

In [3]: sorted(dicionario) # Iterando pelos valores
Out[3]: [4, 5]

In [4]: sorted(dicionario.iterkeys()) # Iterando pelas chaves
Out[4]: [(1, -7), (2,)]

In [5]: len(dicionario) # Verifica o tamanho
Out[5]: 2

In [6]: dicionario[-7] # Consultas
Out[6]: 4

In [7]: dicionario[2]
Out[7]: 5
```

O dicionário de estratégias é um dicionário multi-chave que possui estratégias como valores. Uma estratégia é qualquer objeto que pode ser chamado como uma função, com ou sem restrição ou existência de argumentos (i.e., *callables*). Esse dicionário permite organizar em um lugar e com apenas um nome um conjunto de implementações de uma mesma tarefa. Por exemplo, há seis funções de janelamento ou apodização inseridas na AudioLazy, todas sob o nome de *window*:

Código-fonte 2.11: *Objeto window, um dicionário de estratégias*

```
In [1]: from audiolazy import window

In [2]: window # Vejamos as estratégias disponíveis
Out[2]:
{('bartlett',): <function audiolazy.lazy_analysis.bartlett>,
 ('blackman',): <function audiolazy.lazy_analysis.blackman>,
 ('hamming',): <function audiolazy.lazy_analysis.hamming>,
 ('hann', 'hanning'): <function audiolazy.lazy_analysis.hann>,
 ('rectangular', 'rect'): <function audiolazy.lazy_analysis.rectangular>,
 ('triangular', 'triangle'): <function audiolazy.lazy_analysis.triangular>}

In [3]: window["rect"](3) # Obtém a estratégia, chamando com 1 argumento
Out[3]: [1.0, 1.0, 1.0]

In [4]: window.triangle(3) # Idem, mas feito com outra sintaxe (dicionário)
Out[4]: [0.5, 1.0, 0.5]

In [5]: hm_wnd = window.hamming # Referenciando fora do dicionário

In [6]: hm_wnd # Esta estratégia é uma função comum
Out[6]: <function audiolazy.lazy_analysis.hamming>
```

```
In [7]: for wi in window: # Permitindo o processamento em lote
   ...:     name = wi.__name__
   ...:     data = wi(3)
   ...:     print "{0:<15}{1}".format(name, data) # Alinhamento
   ...:
hamming        [0.08000000000000002, 1.0, 0.08000000000000002]
hann           [0.0, 1.0, 0.0]
blackman       [0.0, 1.0, 0.0]
triangular     [0.5, 1.0, 0.5]
bartlett       [0.0, 1.0, 0.0]
rectangular    [1.0, 1.0, 1.0]
```

Essa classe não obriga a interface das estratégias a serem iguais, o que permite que algoritmos mais generalistas implementados em alguma das estratégias tenham, por exemplo, um maior número de parâmetros do que as demais estratégias. Devido à existência de valores padrão para os parâmetros, e também à possibilidade de criação de dicionários de parâmetros, essa liberdade é compatível com o uso iterativo das estratégias. A varredura pelas estratégias conforme visto no último código-fonte fornecido é um recurso que possibilita um ganho de desempenho com relação ao tempo gasto com o desenvolvimento por parte do programador, através de uma interface com a qual tal usuário sequer precisa conhecer os nomes ou a quantidade de estratégias disponíveis, bastando para isso que as estratégias mantenham alguma padronização quanto aos parâmetros de entrada. A vantagem da organização em objetos StrategyDict é permitir que um iterável organize estratégias ao mesmo tempo em que permite o acesso por nome, por vezes com mais de uma opção de nome, sem prejuízo quanto à generalidade de acesso e organização. Há dez objetos StrategyDict na AudioLazy:

Tabela 2.3: *Instâncias de dicionários de estratégias presentes na AudioLazy*

Nome	Estratégias	Módulo	Descrição
window	6	lazy_analysis	Funções de janelamento ou apodização
envelope	3	lazy_analysis	Filtros de obtenção de envoltória dinâmica
maverage	3	lazy_analysis	Criador de filtros de média móvel
erb	2	lazy_auditory	Largura de banda equivalente retangular (ERB)
gammatone	3	lazy_auditory	Filtros gammatone
comb	3	lazy_filters	Filtros comb
resonator	4	lazy_filters	Ressonadores
lowpass	2	lazy_filters	Passa-baixas
highpass	1	lazy_filters	Passa-altas
lpc	5	lazy_lpc	Codificação linear preditiva (LPC)

A rigor, o comportamento dinâmico do Python permite que praticamente tudo tenha seu comportamento selecionado em tempo de execução (característica fundamental do padrão estratégia), e esse modelo de dicionário de estratégias permite não apenas uma organização de todas as estratégias em tempo de execução como também a consulta, a inserção, a remoção e a alteração de estratégias.

2.3.6 Organização geral do código

A maior parte do código se encontra na forma de pequenos componentes modularizados interligados, com poucas classes e muitas funções, parte dessas organizadas através de dicionários de estratégias.

Um módulo em Python, embora possa ser instanciado em tempo de execução devido à dinamicidade da linguagem, costuma ser um arquivo que permita sua importação. Na AudioLazy, os módulos são arquivos da forma "*lazy_nome.py*" que agrupam os componentes com base no que eles representam. Embora seja possível separar um arquivo por classe, função ou dicionário de estratégias, tal prática exigiria o uso de subpacotes para permitir a organização dos componentes, o que dificultaria o trabalho do usuário, além de ser desnecessário dado que muitos dos componentes são demasiado pequenos em termos de quantidade de linhas de código.

Em Python, dividir o código em um arquivo por classe não é uma prática recomendada. Resumidamente, módulos são agrupamentos de componentes, e o uso da palavra modularização no presente texto refere-se à componentização do pacote, isto é, à divisão em componentes e possibilidade de conexão entre componentes, não à divisão em módulos de acordo com a padronização do Python.

Um pacote Python é um diretório com módulos que contém um arquivo "*__init__.py*". Os pacotes são importáveis como módulos, e podem ser vistos como uma forma de organização de conjuntos de módulos. o pacote audiolazy contém os seguintes módulos:

Tabela 2.4: *Módulos da AudioLazy*

Nome	Descrição
lazy_analysis	Análise de áudio
lazy_auditory	Modelagem do aparato auditivo humano periférico
lazy_core	Núcleo com as três classes de fundamentação do pacote
lazy_filters	Filtros
lazy_io	Gravação e reprodução de áudio (via PyAudio), multi-thread
lazy_itertools	Conteúdo da *itertools* "decorado" ou adaptado para objetos Stream
lazy_lpc	Codificação linear preditiva (LPC)
lazy_math	Funções matemáticas para uso em iteráveis com manutenção de tipo
lazy_midi	Representação MIDI e relações entre nota e frequência (altura)
lazy_misc	Diversas ferramentas de uso geral e constantes
lazy_poly	Polinômios
lazy_stream	Definição da classe Stream e derivadas
lazy_synth	Pequeno sintetizador

Os módulos são internamente conectados conforme descrito pela figura 2.2. Tais conexões refletem a importância do núcleo e da classe Stream. Existem casos de importação circular entre módulos, entretanto não se tratam de um problema devido às seguintes razões:

- Os módulos são importados pelo Python apenas uma vez. Novas importações apenas fazem uma referência ao que já consta no dicionário *sys.modules*.
- Há importações que são dinâmicas e internas à tarefa específica. Isso ocorre, por exemplo, no uso do MatPlotLib dentro do método *audiolazy.lazy_filters.LinearFilter.zplot*(·), relativo à criação de diagramas de zeros e polos de filtros lineares.
- As ocorrências de dependência circular são localizadas, e não se referem a dependências de fato. As duas que envolvem o módulo *lazy_filters* estão contidas nos dois métodos de criação de gráficos com o MatPlotLib, de forma que, em geral, pode-se dizer que o módulo não depende nem do *lazy_synth* e nem do *lazy_analysis*, embora os utilize. O mesmo ocorre com a *lazy_misc*, a qual não depende de nenhum módulo, mas em determinadas circunstâncias precisa avaliar se um dado objeto é um Stream ou um StrategyDict, situações nas quais ela precisa dos outros módulos. Dessa forma, não há dependências circulares, mas há referências que criam a circularidade exposta. Todos os demais casos expostos na figura 2.2 são de dependências.

Visando complementar o diagrama de classes da figura 2.1, segue na figura 2.3 um diagrama de classes incluindo o primeiro nível de ancestrais. Um fato notável é que a quantidade de classes da AudioLazy não é grande. Há 6 metaclasses, que já foram discutidas na seção 2.3.4, e 19 classes, as quais são brevemente descritas na tabela 2.5.

Além das classes, módulos e dicionários de estratégias, há muitas funções definidas no pacote para a realização de pequenas tarefas. Uma dessas funções é o decorador *tostream*, visto na seção 2.3.3. Uma lista completa, com explicações individualizadas, de todas as funções, estratégias, dicionários de estratégias, módulos e classes encontra-se no manual oficial desenvolvido para o pacote, incluído no apêndice ao final deste trabalho.

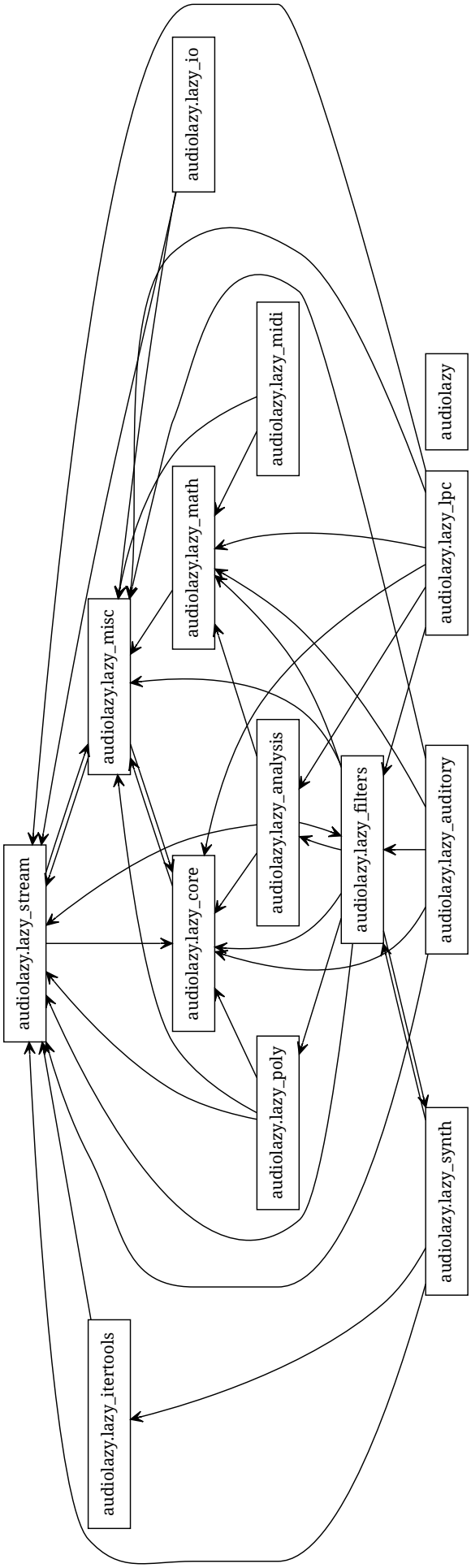


Figura 2.2: *Relações de dependência e referência entre pacotes da AudioLazy. Gerado a partir do código com o pyreverse, parte do pylint.*

Tabela 2.5: *Classes da AudioLazy, exceto metaclasses*

Nome	Bases (herança)	Módulo	Descrição
AudioIO	object	lazy_io	Reprodutor/gravador de áudio
AudioThread	threading.Thread	lazy_io	Thread representando objetos sendo reproduzidos
LinearFilterProperties	object	lazy_filters	*Mixin* com conversores de propriedades de filtros lineares
LinearFilter	LinearFilterProperties	lazy_filters	Filtro linear
ZFilter	LinearFilter	lazy_filters	Filtro linear representado por equações em Z
FilterList	list, LinearFilterProperties	lazy_filters	Lista de filtros
CascadeFilter	FilterList	lazy_filters	Filtros em cascata
ParallelFilter	FilterList	lazy_filters	Filtros em paralelo
Poly	object	lazy_poly	Polinômios, polinômios de Laurent, soma de potências
TableLookup	object	lazy_synth	Sintetizador por consulta à tabela
MultiKeyDict	dict	lazy_core	Dicionário multi-chave
StrategyDict	MultiKeyDict	lazy_core	Dicionário de estratégias
StrategyDictInstance	StrategyDict	lazy_core	Uma classe para cada dicionário de estratégias, ver seção 2.3.8
Stream	collections.Iterable	lazy_stream	Iterável com operadores elemento a elemento e avaliação tardia
ControlStream	Stream	lazy_stream	Stream que devolve um valor controlável, permitindo interatividade
Streamix	Stream	lazy_stream	Misturador (*mixer*) de objetos Stream baseado na temporização do MIDI
StreamTeeHub	Stream	lazy_stream	Gerenciador de cópias de Stream
MemoryLeakWarning	Warning	lazy_stream	Número de usos de um StreamTeeHub menor que o especificado
ParCorError	ZeroDivisionError	lazy_lpc	Erro ao tentar obter coeficientes PARCOR

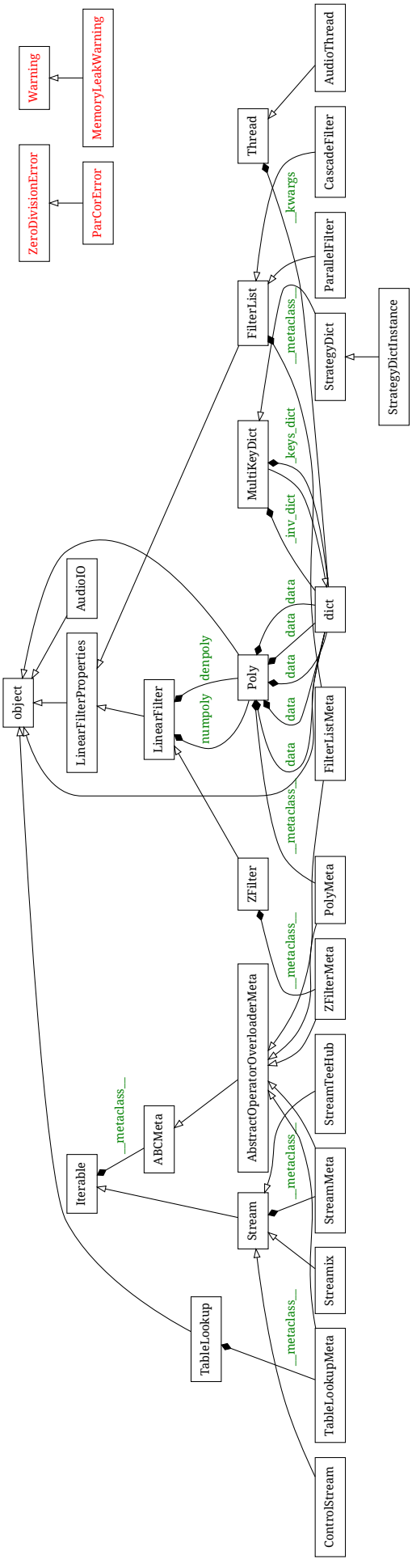


Figura 2.3: *Diagrama de classes da AudioLazy com um nível de ancestrais. Gerado a partir do código com o pyreverse, parte do pylint. As associações marcadas com __metaclass__ são relações entre metaclasse e classe. São associações devido à maneira como o vínculo é realizado no Python 2.x. As múltiplas associações do polinômio ao dicionário referem-se às múltiplas possibilidades de entrada para o construtor.*

Através dos recursos do paradigma reflexivo do Python, o pacote automaticamente detecta todos os arquivos iniciados por "lazy_" que fazem parte da AudioLazy e os importa, fazendo com que a importação da AudioLazy seja completa ao início (isto é, imediata, em contraste com uma importação tardia), e possibilitando a importação de qualquer um de seus componentes diretamente de "audiolazy", sem requerer do desenvolvedor usuário do pacote a preocupação com a organização interna em módulos. Em termos de encapsulamento, é definido o valor de "__all__" em todos os módulos e no pacote, de forma a indicar ao usuário quais são os componentes utilizáveis e permitir que a sintaxe "**from** audiolazy **import** $*$" seja utilizável sem depender de externalidades. Há também uma limpeza de nomes (i.e., referências) usadas temporariamente no pacote para esse fim e para a geração automática de sumários (ver seção 2.3.8), de forma a deixar apenas referências relevantes disponíveis no pacote.

2.3.7 Testes

A versão atual da AudioLazy conta com mais de 5400 testes, compreendendo 75% de cobertura de código, sendo a quantidade exata de testes difícil de mensurar. Testes de *software* são rotinas que realizam alguma tarefa utilizando o *software*, comparando os resultados, os erros, os avisos e os efeitos colaterais (e.g. criação de um arquivo) obtidos com os esperados, desfazendo os efeitos colaterais após sua finalização.

Em Python, há muitas maneiras de se realizar testes. Uma delas é o módulo *unittest* da biblioteca padrão, que implementa a XUnit no Python. Originalmente, a AudioLazy tinha seus testes escritos dessa forma, passando posteriormente a utilizar o *nose* para facilitar a chamada de todos os testes, e finalmente todo o código dos testes foi reescrito para utilização do *pytest*[44], que possibilitou o uso de testes parametrizados.

Testes parametrizados são testes que incluem parâmetros para sua realização. Uma rotina de teste parametrizado incorpora diversos testes, um para parametrização que realiza uma chamada à rotina de teste. Um exemplo ocorre com os testes dos filtros lineares usando os filtros lineares invariantes no tempo do SciPy:

Código-fonte 2.12: *Classes de testes utilizando o SciPy como oráculo*

```
import pytest
p = pytest.mark.parametrize

from scipy.signal import lfilter
from audiolazy import ZFilter, almost_eq

class TestZFilterScipy(object):

  @p("a", [[1.], [3.], [1., 3.], [15., -17.2], [-18., 9.8, 0., 14.3]])
  @p("b", [[1.], [-1.], [1., 0., -1.], [1., 3.]])
  @p("data", [range(5), range(5, 0, -1), [7, 22, -5], [8., 3., 15.]])
  def test_lfilter(self, a, b, data):
    filt = ZFilter(b, a) # Cria um filtro com a AudioLazy
    expected = lfilter(b, a, data).tolist() # Aplica o filtro com o SciPy
    assert almost_eq(filt(data), expected) # Compara os resultados
```

Esse pequeno código utiliza a função *lfilter* do SciPy como oráculo para a validação dos resultados com a classe ZFilter da AudioLazy. Um oráculo é um componente de *software* cujos resultados são considerados verdadeiros, utilizado em testes para a validação de implementações alternativas. Embora o exemplo dado tenha poucas linhas de código, ela compreende 80 testes: há 5 possibilidades de denominador (variável *a*), 4 possibilidades de numerador (variável *b*) e 4 possibilidades de dados de entrada para testar o filtro (variável *data*). O produto cartesiano desses parâmetros é utilizado para testar todos os arranjos de entradas possíveis com esses dados, verificando se a funcionalidade do filtro é equivalente à do SciPy. Embora os filtros da AudioLazy possam usar números inteiros ou outros objetos como coeficientes, o teste se limitou ao uso de pontos flutuantes devido a restrições da função *scipy.signal.lfilter*.

[44] A versão em uso do pacote é a 2.3.4.1.
Disponível em: http://pytest.org/latest/

É possível observar no teste realizado o uso da função *almost_eq*, um dos dois sistemas de comparação de pontos flutuantes da AudioLazy, sendo *almost_eq_diff* o segundo. Ambos são alternativas à comparação com o operador de igualdade "==", a qual, quando realizada sobre pontos flutuantes, detectaria números com pequenos desvios numéricos como diferentes. As duas alternativas se distinguem quanto à forma como é tratado o desvio numérico.

A comparação por valor absoluto da diferença, implementada na *almost_eq_diff* comporta-se de maneira similar ao *unittest.TestCase.assertAlmostEquals*, comparando se a diferença em valor absoluto de um par a e b de entrada é de, no máximo, um certo limiar l, isto é, os valores do par são considerados quase iguais quando satisfazem:

$$|a - b| <= l \tag{2.1}$$

Um valor típico para o limiar é o de $l = 10^(- 7)$, que pode ser grafado $1e - 7$ em uma notação mais apropriada para o que segue. Esse valor é o utilizado por padrão na *unittest*, possivelmente devido ao número de ordens de grandeza de precisão numérica de pontos flutuantes de 32 bits. Porém, diferentemente do método de comparação por aproximação da *unittest*, a versão disponibilizada na AudioLazy e utilizada em seus testes permite a avaliação elemento-a-elemento sobre iteráveis quaisquer, recursivamente, possivelmente comparando também os tipos/classes dos containers, a depender de parâmetros. Por padrão, essas diferenças de tipos são ignoradas (i.e., os elementos são avaliados, não os tipos de containers), e quando o número de elementos é diferente, há uma verificação sobre os elementos adicionais do container maior na dada dimensão, para assegurar que estes contenham apenas zeros. Isso permite avaliar objetos Stream e outros iteráveis com certa flexibilidade, entretanto, assim como ocorre com o uso da implementação na *unittest*, esta pode ser uma comparação perigosa quando realizada com números significativamente maiores que o limiar, o que pode ser facilmente ilustrado com o uso do número $1e30$ como um dos valores da comparação. Sua notação em ponto flutuante de 32 bits (padrão IEEE 754, com base 2) o força a ter a mesma representação que o número $1.000000015047e30$. Importante notar que, devido à restrição do número de bits de mantissa no ponto flutuante, os dígitos 15047 ao final referem-se a uma limitação quanto às possibilidades de codificação do número (quantificação), o que torna outros valores, como por exemplo o $1.000000015046e30$, indiscerníveis com relação ao anterior, isto é, outro número com a mesma representação em ponto flutuante. Entretanto, mesmo sendo a diferença entre os números fornecidos de apenas $1e18$, estes são considerados significativamente diferentes quando comparados com o limiar de $1e - 7$. Dessa forma, números grandes precisam ser exatamente iguais para garantir a "quase igualdade" desejada, visto que um bit de mantissa modificado já é suficiente para ultrapassar o limiar.

Na comparação por bits de mantissa, implementada na AudioLazy na função *almost_eq* com as mesmas características da *almost_eq_diff* relativas ao funcionamento sobre iteráveis, o desvio é calculado a partir de um número que estima quantos bits de mantissa precisariam ser alterados para converter um dos números no outro, isto é, comparando números a e b, ambos em ponto flutuante de base 2 e com s bits de mantissa em suas representações, estes são considerados quase iguais quando satisfazem:

$$|a - b| <= 2^{(t-s-1)|a+b|} \tag{2.2}$$

Em que t é o número de bits de tolerância, ou seja, até quantos bits de mantissa podem ser alterados para que os valores sejam considerados iguais.

Como indicado através do exemplo fornecido, os testes da AudioLazy dependem do NumPy e do SciPy, utilizados como oráculos para a avaliação de procedimentos que poderiam ser feitos em tais pacotes, mas que foram recriados de uma maneira que aceitasse outras possibilidades de tipos de dados na AudioLazy, além da já comentada ênfase na avaliação tardia. Os módulos de teste que utilizam tais pacotes encontram-se em separado, contendo essa informação no próprio nome do arquivo. São testes com oráculos:

- 80 testes utilizando como oráculo a implementação de filtro linear implementado no subpacote de processamento de sinais do SciPy (análogo à *toolbox* de processamento de sinais do MatLab), como descrito no último código fonte fornecido;
- 64 testes utilizando o subpacote de otimização do SciPy para avaliar numericamente se o ganho máximo

coincide com o projetado em todas as estratégias de filtros ressonadores;

- 2 testes com o NumPy, visando assegurar que a diferença entre os valores resultantes pela AudioLazy e pelo NumPy no exemplo que realiza a comparação de desempenho seja negligenciável.

Dessa forma, para a realização de toda a suíte de testes são necessários o NumPy e o SciPy, além de, obviamente, o *pytest*. Inclui-se nessa lista o *pytest-cov*, um *plug-in* do *pytest* feito para realizar um relatório de cobertura dos testes, explicitando o percentual de comandos (*statements*) do código que foram utilizados por algum teste, assim como as linhas de código que ainda não foram testadas. No caso de blocos condicionais (utilizações do "*(*if)"), é verificado se há pelo menos um teste para o verdadeiro e pelo menos um para o falso (*branch*). O relatório do último teste realizado dia 2013-02-17 sobre a AudioLazy foi[45]:

```
--------------- coverage: platform linux2, python 2.7.3-final-0 ---------------
Name             Stmts   Miss Branch BrPart  Cover   Missing
------------------------------------------------------------
__init__            44      1     18      4    92%   72
lazy_analysis      124      9     62      7    91%   210, 255-260, 339, 367, 394
lazy_auditory       60      0     14      0   100%
lazy_core          130      7     56      5    94%   93-94, 108, 151, 211, 328, 335
lazy_filters       508    180    249    118    61%   53, 60, 81, 91, 102-105, 150, [...]
lazy_io            100     73     36     36    20%   35-37, 63-70, 77, 83, 89, [...]
lazy_itertools      14      3     10      8    54%   61-64
lazy_lpc           127     15     42      7    87%   120, 134-135, 191-194, 392, [...]
lazy_math           36      1     12      0    98%   98
lazy_midi           49      8     22      4    83%   68, 97-99, 106, 145, 151, 153
lazy_misc          208     28    138     30    83%   136, 201-202, 239, 252-253, [...]
lazy_poly          126     30     89     34    70%   90, 146-149, 160, 173, 180, [...]
lazy_stream        139      2     52      4    97%   61, 590
lazy_synth         241     50    118     48    73%   277-299, 319-323, 344-345, [...]
------------------------------------------------------------
TOTAL             1906    407    918    305    75%

======================== 5434 passed in 21.66 seconds ========================
```

Pode-se observar que, enquanto há um módulo com 100% de cobertura nos testes, há também um com 20%, um valor significativamente baixo. O principal motivo pelo qual há menos testes automatizados no módulo *lazy_io* é o fato de que tudo o que é feito na AudioLazy utilizando reprodução e gravação de áudio necessita desse módulo, de forma que os testes sobre seu funcionamento existem, minimamente através dos exemplos que utilizam a entrada e a saída de áudio, mas como testes manuais. Trata-se de uma parte importante do pacote, porém de difícil realização de testes automatizados: é necessário criar um substituto para os componentes do PyAudio relativos à reprodução/gravação de áudio para simular, internamente, a

[45] É completa a informação dada pelo *pytest-cov* acerca da numeração das linhas de código que não são utilizadas por teste algum, ou são utilizadas em apenas uma das condições, entretanto esta foi parcialmente removida para facilitar a visualização dos resultados. Segue uma breve descrição das colunas fornecidas:

- *Name*: Nome do módulo. O nome *__init__* refere-se ao módulo de inicialização do pacote;
- *Stmts*: Número de linhas de código que iniciam comandos;
- *Miss*: Número de comandos que não foram utilizados em testes;
- *Branch*: Número de comandos que permitem a continuação em dois caminhos possíveis;
- *BrPart*: Número de comandos que passou se continuou em apenas um dos caminhos possíveis;
- *Cover*: Porcentagem de comandos que passaram por testes, contando duas vezes aqueles que podem ter duas continuidades possíveis, isto é, $1 - (\text{Miss} + \text{BrPart})/(\text{Stmts} + \text{Branch})$;
- *Missing*: Enumeração detalhada de cada linha ou faixa de linhas que não foram utilizadas em testes.

especificação do PyAudio e possibilitar os testes. A esse substituto utilizado para o isolamento em testes é dado o nome de mock, algo que também seria necessário para simular os métodos de criação de gráficos com o MatPlotLib, o que justifica o valor de 61% no maior módulo da AudioLazy (mais de 170 linhas de código da *lazy_filters* são destinadas aos gráficos). Outra parte não testada são as rotinas que, direta ou indiretamente, obtêm os valores das raízes de polinômios, tarefa realizada pelo NumPy e que exigiria ou uma alternativa à obtenção das raízes, ou um *mock* para testes exclusivos à interface com o NumPy.

Não é necessário, tampouco comum, obter uma cobertura de 100% nos testes, mas vale salientar que essas não são as únicas maneiras de se medir a cobertura por testes. A título de exemplo, é possível realizar testes verificando caminhos possíveis dentro de funções, em que blocos condicionais sequenciais teriam um número exponencial de ramificações de caminho possíveis a serem testados. Entretanto, esse tipo de abordagem possivelmente criaria caminhos impossíveis de serem testados, assim como a linha 72 do *__init__*, a qual apenas verifica se o interpretador atual mantém ou não o nome do índice de um laço *for* após a conclusão do laço, para então eliminá-lo, como parte da "limpeza de nomes" para o usuário do pacote. A verificação nesse caso é necessária, pois a tentativa de eliminar uma referência que não existe resultaria em erro, e uma avaliação foi feita mostrando que o CPython manteve a referência do índice do laço, enquanto o PyPy (versão 1.9.1-dev0, sob a especificação do Python versão 2.7.2) elimina a referência após o término do laço. Em outras palavras, caminhos diferentes são utilizados para a linha destacada em diferentes interpretadores, e o caminho alternativo não tem como ocorrer em um único interpretador. Dessa forma, a cobertura de teste que inclua a mais simples ramificação não tem como, ao usar um único interpretador, chegar a 100% de cobertura de código.

A quantidade de testes não é um valor simples de ser obtido. O número 5434 refere-se à quantidade de chamadas a funções e métodos de testes somada à quantidade de módulos. O motivo dessa soma é o fato de que há testes além das funções e classes de testes do pacote: existem testes escritos na forma de *doctestsdoctests*, ou testes de documentação, que são testes escritos em meio ao texto de documentação de funções, classes e métodos. Tais textos de documentação são chamados de *docstrings* e são melhor detalhados na seção seguinte. O número de testes de documentação é superior ao número de módulos, entretanto o valor exato depende do que é considerado um teste, dado que é possível separar ao meio um único teste com comentários em texto, ou unir vários testes seguidos sem separação na forma de uma história para o leitor. Em uma contagem realizada de maneira manual, há 38 blocos contíguos de testes em meio ao texto de documentação, número maior que o total de módulos envolvidos nesses testes. As *doctests* são voltadas ao usuário e servem como exemplos de uso típico da função, do método e da classe, e são exatamente os exemplos existentes no manual da AudioLazy, presente como apêndice final deste trabalho.

Há muitas outras formas de se classificar os testes, tais como:

- **Testes de unidade**: Voltados a testar uma funcionalidade em isolado, ou utilizando recursos considerados confiáveis;
- **Testes de integração**: Testa a funcionalidade de uma dada integração entre diferentes unidades;
- **Testes de sistema**: Testa o sistema como um todo, através de sua interface externa ao invés de possíveis interfaces de componentes internos do sistema;
- **Testes aleatórios**: Realizam chamadas com valores quaisquer de entrada, ou sistematicamente corrompidos por valores quaisquer. Costumam verificar a funcionalidade de um sistema mesmo na inserção de ruído;

Cada módulo da AudioLazy está associado a pelo menos um módulo de teste, que contém seu nome mas inicia com o prefixo "test_" ao invés de "lazy_". Há testes de unidade, de integração e de sistema, mas é desnecessário individualizar e classificar cada teste. Existem 2 testes no módulo *test_analysis* que envolvem aleatoriedade, mas se trata de algo muito pontual para permitir dizer que a suíte de testes inclui testes aleatórios: tais testes apenas avaliam que a saída sob dados parâmetros independe do valor da entrada, a qual é considerada aleatória.

Assim como grande quantidade de testes não garantem grande cobertura de código, uma grande cobertura de código não garante a qualidade do *software*. Entretanto, os testes servem de indicativo quanto à qualidade,

e é difícil, senão impossível, atribuir um valor objetivo quanto à qualidade do *software* que não tenha como base testes e observações empíricas do *software* em funcionamento. Além disso, o fato de existirem testes permite que alterações no código sejam avaliadas não apenas localmente, possibilitando que refatorações e otimizações sejam feitas com segurança, enquanto o código estiver passando nos testes. Embora o desenvolvimento não tenha seguido sistematicamente uma metodologia ágil orientada a testes (ou comportamento) com *sprints* regulares, pode-se dizer que existiu, desde o início, uma ênfase na tentativa de manter parte significativa do código coberta por testes, algo que pode ser objetivamente avaliado através do histórico de 142 atualizações (*commits*) no decorrer do último semestre, as quais trouxeram principalmente refatorações, documentação, testes ou novos recursos já incluindo seus testes.

2.3.8 Documentação

Um dos aspectos centrais que guiou o desenvolvimento da AudioLazy como um todo é o objetivo de que o pacote fosse suficientemente simples de se usar, preferencialmente intuitivo. Entretanto, mesmo a mais intuitiva das interfaces com o usuário não dispensa uma documentação sobre seu uso, que formaliza suas capacidades e potencialidades, evidencia suas limitações, premissas, detalhamentos, etc., além de ser um aspecto de grande importância para a manutenibilidade do código. Em termos computacionais, e em se tratando de um pacote de processamento de áudio escrito em uma linguagem de alto nível, esse detalhamento refere-se a uma breve explicação da funcionalidade realizada, bem como às entradas e saídas esperadas por cada componente, incluindo cada estrutura utilizada, para que as possibilidades de conexões entre componentes possam ser reconhecidas pelo usuário ou leitor.

Docstrings, ou textos de documentação, são explicações escritas junto ao código de um componente e que podem ser consultadas pelo desenvolvedor usuário deste componente através do comando *help* do Python, do "?" do IPython ou do *object inspector* no Spyder. As documentações foram feitas de forma a possibilitar que fossem exibidas de maneira formatada (cores, tamanhos, estilos, etc.) no Spyder, usando para isso uma linguagem chamada reStructuredText.

Como dito na seção anterior, existem também testes realizados em meio à documentação (*doctests*), os quais são importantes no sentido de reuso de código: o mesmo código escrito para servir de exemplo na documentação é avaliado pelo *pytest*, verificando se as entradas e saídas correspondem à realidade do pacote. Essa comparação é feita por meio do texto de entrada e de saída, e não por meio de estruturas de dados internas, o que impõe, por exemplo, que estruturas sem ordenação sejam ordenadas em exemplos.

O Sphinx é uma ferramenta criada no Python que permite gerar a documentação de um projeto a partir do código, utilizando para isso as *docstrings* escritas em reStructuredText, entretanto em um formato diferente daquele utilizado pelo Spyder[46], além do fato de que o Sphinx compreendeu os dicionários de estratégias como módulos, embora estes sejam simples instâncias. Uma configuração foi feita para automatizar o processo de conversão das *docstrings* do formato do Spyder para o formato do Sphinx, e uma adaptação para utilizar o *automodule* do Sphinx em dicionários de estratégia. Isso permitiu a criação da documentação em diversos formatos diferentes, como registrado no último apêndice deste trabalho.

Para contribuir com a documentação, os recursos de programação reflexiva do Python foram utilizados. Na importação do pacote, após a identificação de todos os módulos, é feito um sumário desses pacotes e inserido à *docstring* do pacote. Posteriormente, mas ainda antes de finalizar a importação, um sumário do conteúdo de cada pacote é feito e inserido na *docstring* de seu respectivo pacote. Esses sumários automatizados aparecem como tabelas na documentação do Sphinx, e como um texto comum devidamente alinhado na *docstring* caso esta seja consultada, o que auxilia a manutenibilidade da documentação por evitar a repetição. A ideia é similar à de sobrecarga massiva de operadores (seção 2.3.4), porém aplicado à documentação.

Os dicionários de estratégias são objetos, os quais possuem como documentação as *docstrings* de suas classes. Adicionalmente, classes possuem uma restrição adicional em Python que os módulos não possuem: suas *docstrings* são imutáveis, qualquer tentativa de alteração em seu valor resulta em um erro. Uma forma

[46] Vale salientar que o formato do Spyder é o mesmo utilizado pelo NumPy, dividindo explicações sobre parâmetros, valor devolvido, exemplos, sugestões de outras funcionalidades que podem ser vistas, etc. como diferentes seções, ao invés de comandos reStructuredText dentro de uma única seção (formato esperado pelo Sphinx). A MARLib, embora não tenha sido utilizada, também utiliza em seu código a padronização do Spyder-NumPy de formatação de *docstrings*.

encontrada de contornar isso foi através do uso de propriedades. Em Python, uma propriedade, ou mais genericamente um descritor, pode ser visto como um atributo convencional de um objeto, entretanto seu acesso, modificação e remoção é realizado através de métodos específicos, os quais possibilitam o comportamento de um atributo. Resumidamente, uma propriedade se comporta como um atributo, mas chama métodos a cada acesso para obter o valor a ser devolvido. Isso possibilita à *docstring*, em geral armazenada em uma classe no atributo com nome *__doc__*, se comportar de maneira dinâmica como sendo uma propriedade, porém nesse caso a própria classe deixaria de ter uma *docstring*, pois a tentativa de acesso a esta devolveria a própria propriedade, ao invés do resultado de uma chamada a um método, como ocorreria com o objeto. A solução para o problema foi a criação de mais um nível hierárquico entre a classe StrategyDict e suas instâncias. Uma nova classe StrategyDictInstance, derivada de StrategyDict, é instanciada internamente ao construtor da StrategyDict a cada chamada, e saltando o que poderia ser encarado como um impasse devido ao aspecto cíclico dos construtores, é criada uma instância da classe StrategyDictInstance. Esse objeto dessa nova classe é posteriormente devolvido pelo construtor da StrategyDict. Em suma, a criação de um objeto StrategyDict cria uma nova classe dinamicamente e devolve a instância dessa nova classe. Isso foi necessário para que, ao mesmo tempo, a classe StrategyDict e as instâncias devolvidas por seu construtor tivessem *docstrings*, garantindo a documentação tanto para o uso dos comandos do console como para a documentação gerada pelo Sphinx. Dessa forma, a classe StrategyDictInstance nas figuras 2.1 e 2.3 não é única, existe uma dessa para cada objeto StrategyDict, e a justificativa de sua existência é a busca de uma maneira de facilitar o acesso à documentação pelo desenvolvedor.

No repositório, há outros arquivos relativos à documentação em reStructuredText, também escritos de forma a possibilitar seu reuso: o mesmo texto escrito no simples *README.rst* é compilado como página inicial do repositório, processado e adaptado pelo *script* de instalação *setup.py* para a criação da descrição no PyPI e processado pelos scripts de criação da documentação através do Sphinx, conteúdo este relativo ao primeiro capítulo do manual presente no apêndice ao final deste trabalho.

2.3.9 Exemplos e experimentos

Alguns exemplos de funcionalidade foram inseridos no repositório da AudioLazy, e estão disponíveis no segundo apêndice do presente trabalho. Algumas dependem de pacotes como o Music21, o wxPython e o MatPlotLib, os quais demonstram, além de exemplificar, a integração da AudioLazy com diferentes pacotes do Python. Duas foram as motivações para a criação dos exemplos: a didática, em termos de permitir que desenvolvedores usuários da biblioteca tenham tarefas resolvidas como material para consulta, e a elaboração de experimentos que avaliassem a funcionalidade do pacote, como um complemento aos testes, sobretudo no que tange a entrada e saída de áudio e a criação de gráficos através do MatPlotLib.

Dentre os exemplos presentes no apêndice, há um cuja criação teve como intenção avaliar a possibilidade de processamento em tempo real. Para isso, tornou-se necessário criar uma classe que permitisse a variação do valor devolvido, em tempo de execução, isto é, além das durações envolvidas, era necessário algo que permitisse a interação de um usuário. A solução envolve a criação de uma nova classe, derivada da classe Stream mas que permite receber valores do usuário que de alguma forma controla o sinal. Instâncias da classe *ControlStream* são objetos Stream que devolvem o valor de seu atributo *value* a cada requisição. Em uma tal instância, o valor pode ser mudado a qualquer instante, e na requisição seguinte o objeto *ControlStream* devolverá o novo valor. O exemplo foi criado como uma interface, a qual está melhor detalhada na seção 3.2.10, que trata dos modelos de síntese.

Ao se falar em tempo real, a primeira questão que vem em mente é a da latência envolvida no processamento do áudio, e esse é o assunto da seção que segue.

2.3.10 Latência

Um item importante para a pesquisa realizada é a avaliação de latência do pacote. Esse item é essencial para avaliar objetivamente a possibilidade de processamento em tempo real pelo pacote AudioLazy. Para realizar a avaliação da latência, um experimento precisou ser feito. O ambiente, em termos de dispositivos e suas especificações, está descrito na seção 1.4. A descrição foi necessária pois a medição da latência incluiu

aspectos do hardware utilizado. O objetivo é a medição da duração entre a síntese pela AudioLazy e a análise, também pela AudioLazy, do mesmo sinal, passando por todo o percurso de reprodução e gravação de áudio.

Para reprodução com o PyAudio, este foi conectado ao JACK (via PortAudio), que foi previamente configurado para trabalhar com 2 buffers de 256 amostras cada a 44100 amostras por segundo, o que já impõe uma latência mínima de 512 amostras ao sistema, o que corresponde a aproximadamente $11,6$ milissegundos. A saída de áudio do computador foi conectada através de um cabo P2-P2 à entrada (*loopback*). Foram necessários dois scripts, ambos escritos utilizando a AudioLazy, chamados em processos separados. Em um, um sinal senoidal de 440 Hz após silêncio foi sintetizado, indicando o instante de tempo do relógio do computador imediatamente antes do envio da primeira amostra. No outro *script*, um detector de amostras que passassem de 50% do valor de pico representável, indicando o instante de tempo e interrompendo essa análise após sua primeira deteção. Primeiramente, o hardware foi preparado, e o *script* de análise foi executado. Em seguida, o *script* de síntese foi acionado, e a diferença entre os instantes de tempo fornecidos por cada *script* foi utilizada como valor de latência total do sistema. Os valores brutos das 16 medições encontram-se na tabela 2.6, assim como o valor médio e o desvio padrão.

Segue abaixo o *script* de síntese, desenvolvido para manter o áudio em silêncio e depois de um segundo de sua chamada disparar uma senoide de 440 Hz. O tempo de um segundo é importante para evitar a contaminação pelo transitório inicial de chamada do *script*, no qual tarefas como a alocação de memória e construção dos objetos são realizadas. São enviados ao PyAudio pequenos blocos de 16 amostras, o que também exerce influência sobre a latência mínima.

Código-fonte 2.13: *Disparo informando o instante de tempo*

```
from audiolazy import sHz, tostream, izip, count, sinusoid, AudioIO
import time

rate = 44100
s, Hz = sHz(rate)

@tostream
def alt_impulse(time_to_send):
  time_to_send = int(abs(round(time_to_send)))
  for x, s in izip(count(), sinusoid(440*Hz)):
    if x == time_to_send:
      start = time.time()
      print repr(start) # Ponto flutuante com todos os dígitos
    yield s if x >= time_to_send else 0.

with AudioIO(True) as player:
  player.play(alt_impulse(1 * s),
              chunk_size=16,
              rate=rate)
```

Um dos motivos para a síntese ocorrer com uma senoide de 440 Hz ao invés de um impulso ou um degrau foi o fato de que o experimento se deu envolvendo o hardware. Isso significa que o tempo resultante inclui o atraso de conversão do sinal digital em analógico, e também o tempo de conversão do sinal analógico em digital, mas não se deve esquecer que tais conversões também realizam uma filtragem ao sinal, a qual torna incompatível o uso de sinalizações pontuais tais como o impulso e o degrau. Finalizando a implementação, segue o *script* de análise, que aguarda pela primeira amostra que ultrapasse o valor do limiar, registra o instante de tempo e finaliza.

Código-fonte 2.14: *Identificação de sinal por limiar, informando o instante de tempo*

```
from audiolazy import AudioIO
import time

rate = 44100

```

```
with AudioIO() as recorder:
  input_data = recorder.record(chunk_size=16, rate=rate)
  for el in input_data:
    if el > .5:
      stop = time.time()
      print repr(stop) # Ponto flutuante com todos os dígitos
      break
```

Finalmente, os dados coletados estão na tabela 2.6, indicando que a AudioLazy na dada configuração tem latência de aproximadamente 35 milissegundos como atraso do sistema como um todo. A validade da comparação entre os instantes de tempo devolvidos em diferentes processos existe pelo fato de que é um mesmo relógio registrando os instantes de tempo.

Tabela 2.6: *Coleta de valores de latência da AudioLazy*

Valores			(milissegundos)
35.39609909	35.39180756	35.09497643	35.33506393
34.53612328	34.41786766	35.33387184	34.40213203
35.44521332	35.38012505	34.42001343	34.54184532
34.37614441	34.34801102	34.49606895	34.47413445
Média	34.83684361		
Desvio padrão	0.45156109		

2.4 Sequências musicais

Há diversos formatos de representação digital de partituras. Pode-se enumerar alguns deles:

- MusicXML;
- Kern (Humdrum)[47];
- Lilypond[48];

Além desses formatos, existem formatos proprietários usados pelos diversos editores de partituras (Encore, Finale, Sibelius, Guitar Pro, etc.). Uma capacidade que todos esses formatos listados possuem é a de conversão da partitura em arquivos MIDI. Nessa seção é discutido o que é o MIDI e um pacote em Python que facilita o acesso às partituras, além de consultas sobre seus conteúdos.

2.4.1 MIDI

O protocolo MIDI[49] (*Musical Instrument Digital Interface*) foi criado para estabelecer uma comunicação entre diferentes sistemas musicais digitais. Criado a partir da padronização realizada para esse protocolo, existe um formato de arquivo MIDI, dividido em partes (*chunks*), sendo um cabeçalho (*header chunk*) e um conjunto de trilhas (*track chunks*), a qual pode ser chamada de sequência MIDI, dado que as trilhas contêm uma sequência de eventos, além das informações sobre o instante de ocorrência dos mesmos (uma sequência de instantes relativos).

Apesar da divisão em trilhas, estas são úteis apenas como organização do conteúdo musical descrito. Para o som, o MIDI é dividido em canais, cada um deles associado a um programa MIDI, o qual é responsável

[47] Mais de 100 mil partituras disponíveis em:
http://kern.humdrum.org/, último acesso dia 2012-02-18.

[48] *Software* (Parte do projeto GNU), especificação e documentação em:
http://lilypond.org/, último acesso dia 2012-02-18.

[49] Uma referência didática para a compreensão do formato de arquivo MIDI é o endereço:
http://www.sonicspot.com/guide/midifiles.html, último acesso dia 2010-07-18.

pelos sons das notas enviadas através de eventos a seus canais. Cada trilha possui diferentes eventos ou metaeventos MIDI, e são os eventos que são associados aos diferentes programas, não as trilhas.

Metaeventos MIDI contém informações gerais, tais como o nome da música ou o valor da fórmula de compasso, que em sua maioria não são utilizadas para a reprodução sonora[50] mas que podem ser úteis para um editor de partitura ou para análise musicológica. Os metaeventos costumam ter tamanho variável.

Um evento MIDI comum está em uma trilha, possui 3 *bytes* na mensagem, sendo 2 *bytes* de parâmetros e 1 *nibble*[51] para identificar o tipo de evento e 1 *nibble* para identificar a qual dos 16 canais o evento está associado. Os diferentes tipos de eventos disponíveis são:

- Início de nota (*Note On*)
  A mensagem contém dois valores, um refere-se à intensidade e outro à altura. Em sons percussivos, o parâmetro que teria a altura é usado para selecionar um dentre vários timbres do programa em questão.

- Fim de nota (*Note Off*)
  A nota é finalizada, efeito similar ocorre quando é realizado um evento de início de nota com o parâmetro de intensidade igual a zero.

- Bend (*Pitch Bend*)
  Trata de uma variação na afinação do canal como um todo.

- Controle ou CC (*Control Change*)
  Mudança em algum parâmetro de controle, como o volume, *pan* ou *reverb*. O GM (*General MIDI*) possui uma tabela de valores padrões para esses eventos, usando um *byte* da mensagem para a escolha do parâmetro sendo controlado, e outro *byte* para o valor.

- Programa (*Program Change*)
  Mudança do número do programa associado ao som das notas. Normalmente refere-se a uma mudança no timbre. Por exemplo, o primeiro programa no GM refere-se a um piano acústico, e o programa número 69 está associado ao timbre de oboé.

- *Aftertouch* da nota e do canal (*Note Aftertouch* e *Channel Aftertouch*)
  Controle realizado após o início das notas através de um aumento de pressão em uma tecla já pressionada, útil para controlar vibratos individuais em teclados com esse recurso.

- SysEx (*System Exclusive*)
  Único dos eventos cuja mensagem não tem tamanho igual a três *bytes* e não está associado a um canal. É dependente do fabricante do sistema. Há apenas duas padronizações no GM, que são a mudança do volume geral e o habilitar/desabilitar o modo de compatibilidade GM.

As durações dos eventos são obtidas a partir de tempos relativos ao evento anterior, como múltiplos de uma duração básica comum T_T. Cada trilha contém uma sequência de eventos justapostos como pares $(\Delta, evento)$, em que o inteiro Δ denota a duração relativa ao evento anterior como múltiplo inteiro de T_T, isto é, a duração entre a ocorrência do evento anterior e a ocorrência do evento atual é de $T_T\Delta$. Características mais específicas como a maneira como o tamanho variável (de 1 a 4 *bytes*) é especificado para representar os valores dos Δ e as diferentes possibilidades acerca dos eventos *SysEx* e metaeventos são necessárias apenas para a implementação de um sistema que lida com arquivos MIDI, sendo desnecessário aqui entrar em maiores detalhes.

Existem duas maneiras de indicar a duração positiva mínima T_T em arquivos MIDI. A maneira mais comum é separada em duas partes, em que o cabeçalho MIDI indica o número de divisões por pulso e um metaevento indica o andamento através da duração do pulso em microssegundos. Como alternativa, pode-se indicar as durações em quadros e número de divisões por quadro, diretamente no cabeçalho (SMPTE), porém esse tipo de temporização dificulta a importação de arquivos MIDI em editores de partitura, os quais passam a

[50] Há exceções importantes como, por exemplo, o metaevento de fim de trilha, de indicação de atraso inicial da trilha (*offset*) e de indicação de andamento.

[51] Um *nibble* são 4 bits, ou meio *byte*. Normalmente é representado por um dígito hexadecimal.

ser responsáveis pela estrutura rítmica da sequência (fórmula de compasso, andamento, etc.), a menos da existência dessas informações na forma de metaeventos. De uma forma ou de outra, não é difícil configurar T_T para que seja 1 milissegundo, 1/3 de milissegundo ou mesmo um valor menor, o que permite que as durações envolvidas em uma sequência sejam especificadas com grande nível de detalhe. Entretanto a precisão da realização e outras características que envolvem a capacidade de atendimento às requisições registradas em arquivos MIDI depende do sequenciador que reproduz o arquivo, tanto com relação ao *software* como com relação ao *hardware* do sequenciador.

Há duas características importantes no MIDI que influenciaram a AudioLazy. Uma delas é a organização de uma sequência como trilhas segregadas por deltas, e a segunda é a numeração MIDI para as alturas.

Streamix é uma classe da AudioLazy cuja funcionalidade foi inspirada no descrito sistema de temporização de trilhas MIDI. Essa classe permite que sejam adicionados novos eventos na forma de pares *(Δ, iterável)* com o método *add*, unificando em um único objeto Stream a soma de todos os objetos adicionados, em que os valores de Δ são intervalos de tempo, em amostras, entre eventos adjacentes.

O módulo *lazy_midi* realiza conversões entre números MIDI, strings contendo nomes do tipo "C4" para denotar a nota dó central[52] e valores de frequências em hertz (Hz). Isto permite expressividade ao código no sentido em que notas e acordes podem ser diretamente representados através de suas alturas, como no exemplo dado na seção 3.2.10, em que um acorde de lá menor foi escrito como "C3 A3 E4 A4". A vantagem da numeração MIDI é que os intervalos podem ser medidos em semitom, e um curto exemplo disso é a expressão `midi2freq(str2midi("C3") + 2)`, a qual devolve a frequência de um tom acima da nota dó grave do registro de um tenor (i.e., a frequência da nota "D3"). Uma segunda vantagem é poder encontrar valores de frequência para intervalos fracionários tais como quartos de tom.

2.4.2 Music21

O Music21[53] é um pacote Python voltado ao processamento de partituras nos formatos Kern e MusicXML, incluindo limitado suporte ao formato Lilypond e à criação de arquivos MIDI.

Entre os itens existentes no pacote, encontram-se *corpora* de partituras, presentes para possibilitar a análise musicológica com questionamentos do tipo "*Quantos intervalos harmônicos de sétima maior há nos corais de Bach?*".

Um exemplo demonstrando a capacidade de integração da AudioLazy com o Music21 foi realizado e encontra-se entre os exemplos no segundo apêndice deste trabalho. Nele, o *corpus* com 400 corais de Bach presente no pacote Music21 é acessado, um dos corais é selecionado ao acaso e então reproduzido, utilizando o algoritmo de Karplus-Strong com memória inicial alterada (ver 3.2.10).

[52] Tanto a numeração como as letras utilizadas seguem o padrão americano de oitavas e de escrita. Há maiores detalhes sobre alturas na seção 1.1.2.

[53] Disponível em http://web.mit.edu/music21/

Capítulo 3

Filtros Digitais, Síntese e Análise na AudioLazy

Há muitos possíveis significados para o que pode vir a ser um filtro nos mais diferentes contextos onde o termo é utilizado mas, em geral, trata-se de um sistema de seleção de elementos com propriedades desejadas entre os elementos de um conjunto potencialmente maior. Intuitivamente, uma filtragem é um processo em que sua saída não contém informação que já não estivesse de alguma maneira presente na entrada. Filtros são componentes fundamentais à área de processamento digital de sinais, e em muitos casos podem ser implementados de maneira extremamente eficiente, possibilitando seu uso em tempo real.

Em um exemplo intuitivo, sinais definidos a partir de seus componentes parciais (i.e., senoides) com suas frequências e amplitudes podem ser filtrados, e a atuação de um filtro pode se referir a uma seleção sobre esses parciais, alterando a relação de amplitudes de acordo com suas frequências, atenuando algumas componentes em detrimento de outras. A implementação de um filtro como tal é realizada através de uma transformação linear do sinal, e, por conseguinte, transformações lineares mais gerais envolvendo ganho de amplitude maior do que a unidade (amplificação) também são chamadas de filtros. Uma continuação aos aspectos gerais sobre filtros está na seção 3.1.

Do sistema implementado na AudioLazy, a maior parte do código e de seus testes refere-se à parte de filtragem linear, e a seção 3.2 discute os desafios associados à implementação dos recursos fornecidos junto ao pacote. Complementando o conteúdo, há nessa mesma seção algumas informações sobre os algoritmos de síntese implementados na AudioLazy.

Finalmente, na seção 3.3, após uma introdução que visa a distinção entre a altura e o croma, encontram-se modelos de representação e análise de áudio e MIR com ênfase em aspectos frequenciais, incluindo exemplos implementados com a AudioLazy, recursos de análise do pacote, e breves comentários sobre algoritmos com base em suas capacidades e pressupostos.

3.1 Introdução ao processo de filtragem

Duas sequências que são necessárias para a compreensão do que segue são o impulso, ou delta de Kronecker, definido como:

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & c.c. \end{cases} \tag{3.1}$$

E o degrau, ou função discreta de Heaviside, definida como:

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & c.c. \end{cases} \tag{3.2}$$

Foram obtidas de DAFx [Zöl11, capítulo 2], Moore [Moo90, capítulo 2] e Oppenheim [OSB99, capítulos

1 e 2] as nomenclaturas e definições presentes nas partes que trazem a fundamentação teórica deste capítulo, sobretudo no discutido nas seções 3.1.1 e 3.1.2, que tratam de filtragem linear e transformadas Z, respectivamente.

3.1.1 Sistemas lineares e invariantes no tempo (LTI)

Sistemas lineares são sistemas que satisfazem as propriedades de superposição e de homogeneidade. Superposição refere-se à capacidade de um sistema S em garantir que a soma dos resultados de diferentes entradas seja igual ao resultado da aplicação direta da soma das entradas, isto é:

$$S(x_1 + x_2) = S(x_1) + S(x_2) \tag{3.3}$$

Homogeneidade refere-se à multiplicação por um escalar, que não deve modificar o resultado se aplicado anteriormente ou posteriormente à aplicação do sistema, isto é:

$$S(\lambda x) = \lambda S(x) \tag{3.4}$$

Não se deve confundir esta noção de homogeneidade, vista como propriedade de sistemas lineares em manter seu resultado independente da ordem entre a aplicação do sistema e a multiplicação por escalar, com a homogeneidade vista como propriedade de *containers* limitados a elementos de mesmo tipo ou classe, como visto no capítulo 2.

Um processo linear e digital de filtragem sobre sinais digitais pode ser definido a partir de sua formulação no domínio do tempo. Utilizando $x[n]$ como sinal ou sequência de entrada, e $y[n]$ como sinal ou sequência resultante do processo, podemos definir uma importante família de filtros lineares através da equação:

$$y[n] = \sum_{i=0}^{M-1} b_i x[n-i] + \sum_{i=1}^{N} a_i y[n-i] \tag{3.5}$$

isto é, filtros cujas saídas são resultantes da combinação linear de um histórico de amostras de entrada e de um histórico de amostras de saída com base em atrasos i relativos à amostra atual n. No caso em que os coeficientes a_i e b_i são constantes, tais sistemas são conhecidos como filtros causais e LTI (*Linear Time Invariant*, ou lineares e invariantes no tempo). São invariantes no tempo porque os atrasos i são mantidos independentes do índice n para toda amostra. Pode-se facilmente observar sua linearidade através da propriedade de superposição:

$$y_1[n] = \sum_{i=0}^{M-1} b_i x_1[n-i] + \sum_{i=1}^{N} a_i y_1[n-i] \tag{3.6}$$

$$y_2[n] = \sum_{i=0}^{M-1} b_i x_2[n-i] + \sum_{i=1}^{N} a_i y_2[n-i] \tag{3.7}$$

$$y_3[n] = y_1[n] + y_2[n] \tag{3.8}$$

$$= \sum_{i=0}^{M-1} b_i x_1[n-i] + \sum_{i=0}^{M-1} b_i x_2[n-i] + \sum_{i=1}^{N} a_i y_1[n-i] + \sum_{i=1}^{N} a_i y_2[n-i] \tag{3.9}$$

$$= \sum_{i=0}^{M-1} (b_i x_1[n-i] + b_i x_2[n-i]) + \sum_{i=1}^{N} (a_i y_1[n-i] + a_i y_2[n-i]) \tag{3.10}$$

$$= \sum_{i=0}^{M-1} b_i (x_1[n-i] + x_2[n-i]) + \sum_{i=1}^{N} a_i (y_1[n-i] + y_2[n-i]) \tag{3.11}$$

$$= \sum_{i=0}^{M-1} b_i (x_1[n-i] + x_2[n-i]) + \sum_{i=1}^{N} a_i y_3[n-i] \tag{3.12}$$

que é exatamente a aplicação do filtro definido em 3.5 à soma das entradas usadas em 3.6 e 3.7. A homogeneidade também pode ser observada:

$$y_4[n] = \sum_{i=0}^{M-1} b_i x[n-i] + \sum_{i=1}^{N} a_i y_4[n-i] \tag{3.13}$$

$$y_5[n] = \lambda y_4[n] \tag{3.14}$$

$$= \lambda \sum_{i=0}^{M-1} b_i x[n-i] + \lambda \sum_{i=1}^{N} a_i y_4[n-i] \tag{3.15}$$

$$= \sum_{i=0}^{M-1} b_i \lambda x[n-i] + \sum_{i=1}^{N} a_i \lambda y_4[n-i] \tag{3.16}$$

$$= \sum_{i=0}^{M-1} b_i \lambda x[n-i] + \sum_{i=1}^{N} a_i y_5[n-i] \tag{3.17}$$

que é a aplicação do filtro à entrada $\lambda x[n]$.

Outra propriedade existente nos filtros definidos através da equação 3.5 é a causalidade, já que as somatórias da equação 3.5 não possuem índices i negativos. Filtros causais são filtros que dependem apenas do histórico passado de amostras para a computação de sua próxima amostra. Outra forma de definir a causalidade do filtro é através de sua resposta $h[n]$ ao impulso $\delta[n]$, que deve ser nula para índices negativos. Chama-se resposta ao impulso o sinal devolvido pelo filtro como saída quando a este é aplicada a função impulso $\delta[n]$ como entrada. Todo filtro LTI é completamente caracterizado através de sua resposta ao impulso.

Os filtros LTI não são restritos aos definidos segundo a equação 3.5. Caso as somatórias incluíssem índices i negativos com algum coeficiente a_i ou b_i não nulo para tais índices, teríamos filtros não-causais e, portanto, não computáveis durante algum tempo, por dependerem de uma informação que somente estará disponível no futuro. Nesses casos, podemos adaptar os valores de forma a isolar o valor $y[n_{max}]$ na equação, eliminando o problema com relação aos coeficientes a_i sem, de fato, modificar o equacionamento[1]. Se após essa adaptação os novos coeficientes b_i continuarem a requisitar amostras futuras (dessa vez, somente da entrada), é possível computar tardiamente o resultado da saída, desde que as pelo menos $|i_{min}|$ amostras já tenham sido recebidas, em que i_{min} é o mais negativo valor de i associado a um coeficiente b_i não nulo. Em outras palavras, a causalidade, embora não seja um requisito a todas as formulações de filtros LTI, é uma necessidade prática, e pode-se impor a uma formulação não-causal com i_{min} finito a inserção de uma latência bem determinada, definindo um $y_c[n]$ causal a partir de um $y[n]$ não-causal (lembrando que i_{min} é negativo):

$$y_c[n] = y[n + i_{min}] \tag{3.18}$$

A convolução entre dois sinais $h[k]$ e $g[k]$ é a operação definida através da equação:

$$(h * g)[n] = \sum_{i=-\infty}^{\infty} h[i]\, g[n-i] \tag{3.19}$$

que, quando h é zero para valores de n fora do intervalo $[0; \tau - 1]$, fornece:

$$(h * g)[n] = \sum_{i=0}^{\tau-1} h[i]\, g[\tau - i] \tag{3.20}$$

No caso em que $h[n]$ e $g[n]$ são sinais periódicos de período τ, a equação 3.20 também pode ser aplicada, e nesse caso ela é chamada de convolução periódica entre os sinais. A aplicação de um filtro também pode ser vista como a convolução de sua resposta ao impulso com o sinal a ser filtrado.

[1] É possível fazer o mesmo com $y[n_{min}]$ de forma a criar um filtro para aplicação retrógrada no tempo.

Um filtro com resposta ao impulso $h[n]$ é chamado de FIR (*Finite Impulse Response*) quando o número de amostras $h[n] \neq 0$ é finito. A equação 3.20 é um exemplo de filtro FIR com τ amostras não nulas de resposta em frequência, caso consideremos $b_i = h[i]$. Um filtro é estável no sentido BIBO (*Bounded Input, Bounded Output*) quando, dada uma entrada limitada em amplitude, sua saída também é limitada. Por utilizar apenas uma soma ponderada de número limitado de amostras de entrada para obter uma saída, a saída dos filtros FIR é limitada quando a entrada é limitada. Em outras palavras, todo filtro FIR é estável.

A equação 3.5 nos apresenta duas somatórias, uma envolvendo apenas amostras do histórico de entrada do sinal, e outra envolvendo apenas amostras do histórico de saída. O primeiro somatório realiza um mapeamento direto das saídas a partir das entradas. Uma das características dos filtros FIR é que estes são representáveis apenas com o primeiro somatório, embora esta não seja a única possibilidade de representação. O segundo somatório da equação 3.5 lida com uma retroalimentação do sinal, permitindo que o filtro tenha uma resposta ao impulso de duração indeterminada, potencialmente infinita[2], ou mesmo explosiva. Filtros IIR (*Infinite Impulse Response*) são filtros cuja resposta ao impulso não possui um instante determinado a partir do qual toda amostra resultante é nula. Para tais filtros, é importante avaliar a estabilidade. Uma condição necessária para a estabilidade é a convergência da resposta ao impulso ao zero. Uma condição suficiente é que o sinal tenha energia finita, isto é, que $h[n]$ satisfaça:

$$\sum_{i=-\infty}^{\infty} |h[i]|^2 < \infty \tag{3.21}$$

O processo de filtragem descrito através da equação 3.5 não impõe a estabilidade, embora garanta a linearidade, a invariância com relação ao tempo e a causalidade. Também não restringe os coeficientes ou a entrada a amostras reais, inteiras, discretizadas na amplitude, etc., permitindo seu uso mesmo com números complexos ou vetores.

Como exemplo simples de filtro FIR, pode-se criar uma média das últimas M amostras consecutivas do sinal, de forma a utilizar o filtro da equação 3.5 com $a_i = 0$ para todo i, $b_i = 1/M$ para $i \in \{0, 1, ..., M-1\}$ e $b_i = 0$ caso contrário. A esse filtro é dado o nome média móvel (*moving average*):

$$y[n] = \frac{1}{M} \sum_{i=0}^{M-1} x[n-i] \tag{3.22}$$

Segue abaixo uma implementação de um tal filtro com $M = 3$ feita utilizando a AudioLazy, porém ainda sem utilizar recursos importantes do pacote tais como a representação por transformadas Z. É de certa forma intuitivo esperar que o filtro "suavize" o sinal, atenuando mudanças abruptas, e essa implementação visa apenas ilustrar essa influência do filtro sobre sinais de diferentes frequências. Vale salientar que a AudioLazy possui o filtro média móvel implementado sob o nome de *maverage*[3], com três estratégias diferentes para o cálculo, e que o método *Stream.blocks* não se restringe ao processamento linear.

Código-fonte 3.1: *Média móvel (FIR) usando Stream.blocks*

```
In [1]: from audiolazy import (tostream, Stream, sinusoid, lag_to_freq,
   ...:                        pi, impulse) # Poderia ser "*"

In [2]: @tostream
   ...: def exemplo_filtro_fir(sig):
   ...:   sig = Stream([0., 0.], sig) # Concatena zeros antes
   ...:   for x0, x1, x2 in sig.blocks(size=3, hop=1): # De 1 em 1
   ...:     yield (x0 + x1 + x2) / 3
   ...:

In [3]: # Resposta ao impulso
```

[2] Em sinais digitais, a necessidade de que o filtro seja estável impossibilita que tais filtros tenham uma resposta infinita, pois a necessidade de convergência ao zero para assegurar a estabilidade faz com que, a partir de certo ponto, o sinal quantizado seja indiscernível do zero.

[3] Este nome tem origem no objeto existente em PureData.

```
In [4]: exemplo_filtro_fir(impulse()).take(10)
Out[4]:
[0.3333333333333333,
 0.3333333333333333,
 0.3333333333333333,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0]

In [5]: # Senoides

In [6]: sig_nyquist = sinusoid(freq=lag_to_freq(2), phase=pi/2) # Nyquist

In [7]: sig_nyquist.take(2) # Visualizar um período
Out[7]: [1.0, -1.0]

In [8]: sig_other = sinusoid(freq=lag_to_freq(8), phase=pi/2)

In [9]: sig_other.take(8) # Visualizar um período
Out[9]:
[1.0,
 0.7071067811865476,
 1.2246467991473532e-16,
 -0.7071067811865475,
 -1.0,
 -0.7071067811865477,
 0.0,
 0.7071067811865475]

In [10]: # Aplica o filtro

In [11]: res_nyquist = exemplo_filtro_fir(sig_nyquist)

In [12]: res_nyquist.take(8)
Out[12]:
[0.3333333333333333,
 0.0,
 0.3333333333333333,
 -0.3333333333333333,
 0.3333333333333333,
 -0.3333333333333333,
 0.3333333333333333,
 -0.3333333333333333]

In [13]: res_other = exemplo_filtro_fir(sig_other)

In [14]: res_other.take(8)
Out[14]:
[0.3333333333333333,
 0.5690355937288492,
 0.5690355937288493,
 7.401486830834377e-17,
```

```
 -0.5690355937288492,
 -0.8047378541243649,
 -0.5690355937288493,
 -7.401486830834377e-17]

In [15]: # Ganho do filtro

In [16]: max(res_nyquist.take(10))
Out[16]: 0.3333333333333333

In [17]: max(res_other.take(10))
Out[17]: 0.8047378541243649
```

O exemplo criou um filtro de média móvel sobre $M = 3$ amostras, imediatamente mostrando sua resposta ao impulso para $n = 0, 1, ..., 9$, a qual, como era de se esperar, possui apenas 3 amostras diferentes de zero. Importante dizer que a função *impulse*() apenas cria um objeto Stream que devolve o valor 1.0 uma única vez e, posteriormente, devolve valores 0.0, sem finalização definida.

Posteriormente à avaliação da resposta ao impulso, o filtro foi aplicado a duas senoides, uma na taxa de Nyquist (período de 2 amostras) e outra com período de 8 amostras. As senoides foram sistematicamente colocadas com fase inicial igual a $\pi/2$, para garantir que o valor de pico da senoide se mantivesse constante e igual a 1, não apenas para facilitar a comparação com o resultado mas também para garantir que a amostragem da senoide na frequência de Nyquist não ocorresse precisamente nos valores em que a senoide é nula. O objetivo do exemplo foi o de mostrar que existe uma atenuação diferente para cada senoide fornecida, de forma que a amplitude do sinal na taxa de Nyquist foi atenuada para 1/3 da original, enquanto a amplitude da senoide com período de 8 amostras manteve mais de 80% de seu valor original.

Segue abaixo um segundo exemplo, visando ilustrar a possibilidade de realização de filtros potencialmente explosivos, e a resposta ao impulso potencialmente infinita. O exemplo é o de um acumulador ou integrador, que a cada amostra de entrada, devolve a soma desta ($x[n]$) com a última saída ($y[n-1]$), isto é, na equação 3.5, $b_0 = a_1 = 1$, e todos os demais a_i e b_i são zeros:

$$y[n] = x[n] + y[n-1] \tag{3.23}$$

A implementação que segue como exemplo utiliza a AudioLazy.

Código-fonte 3.2: *Acumulador (IIR) a partir de uma função geradora*

```
In [1]: from audiolazy import tostream, Stream, impulse

In [2]: @tostream
   ...: def exemplo_filtro_iir(sig):
   ...:   y_1 = 0.
   ...:   for x0 in sig:
   ...:     y_1 += x0 # Acumula
   ...:     yield y_1
   ...:

In [3]: # Resposta ao impulso
In [4]: exemplo_filtro_iir(impulse()).take(10)
Out[4]: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

In [5]: # Aplicado ao degrau de Heaviside
In [6]: # (sinal limitado estritamente positivo)
In [7]: exemplo_filtro_iir(Stream(1)).take(10)
Out[7]: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

Pode-se observar que a resposta ao impulso do acumulador é exatamente o degrau de Heaviside, exposta no exemplo em uma representação restrita a índices. Caso tivesse sido utilizada uma entrada como, por

exemplo[4], $[0, 1, 0, 0, 0, 0]$, que se refere a um segmento deslocado do impulso, teríamos precisamente uma resposta deslocada $[0, 1, 1, 1, 1, 1]$, indicando que se trata de uma função degrau de Heaviside e não um sinal igual a 1. De uma forma ou de outra, trata-se de uma resposta sem finalização definida no sentido de um suporte limitado, isto é, não existe um valor n_max a partir do qual sua resposta ao impulso $h[n]$ possua apenas zeros para $n > n_max$, visto que para qualquer $n \geq 0$, $h[n] = 1$. Tomando um exemplo com o filtro aplicado à própria função degrau $u[n]$, vemos que o resultado é uma "rampa" na qual $y[n] = n$. Apesar da entrada ser limitada em amplitude por 1, o resultado da aplicação do filtro não é limitado em amplitude, possuindo um valor igual ao deslocamento no tempo com relação ao início do degrau, algo que, no limite para $n \to \infty$ claramente demonstra ser ilimitado e, portanto, instável com relação ao critério BIBO.

Não foi necessário falar em taxa de amostragem, mas isso seria necessário para indicar os períodos em segundos e as frequências em hertz, ao invés de períodos em amostras e frequências em radiano por amostra como utilizados nos exemplos. A resposta em frequência de filtros IIR como o do exemplo fornecido não é tão simples de ser obtida como no caso de filtros FIR pelo fato de que é necessário saltar um período transitório para coletar o valor de ganho ou atenuação para uma dada senoide de entrada. Além disso, é necessário considerar a possibilidade de que tal valor não seja definido devido à uma possível instabilidade do filtro. A seção que segue entra em maiores detalhes com relação à resposta em frequência dos filtros lineares, e traz outra forma de representação aos filtros.

3.1.2 Transformada Z

A transformada Z representa um sinal no domínio da frequência, de maneira vinculada a atrasos temporais, e pode ser vista como uma generalização da transformada de Fourier. A definição da transformada Z de um sinal $x[n]$ discretizado no tempo é[5]:

$$\mathcal{Z}(x[n]) = X(z) = \sum_{n=-\infty}^{\infty} x[n] z^{-n} \tag{3.24}$$

em que z é uma variável complexa, geralmente expressa em coordenadas polares $z = Ae^{j\Phi}$. Para $n = 0$, convenciona-se que o elemento da somatória é $x[0]$ e este independe de z. Um exemplo imediato de aplicação do equacionamento indica que,

$$\mathcal{Z}(\delta[n]) = 1 \tag{3.25}$$

Nos casos em que a somatória converge, devido à linearidade da somatória, é imediato demonstrar que a transformada Z é um operador linear, isto é, que o operador $\mathcal{Z}(\cdot)$ satisfaz as propriedades da superposição:

$$\mathcal{Z}(x_1[n] + x_2[n]) = \mathcal{Z}(x_1[n]) + \mathcal{Z}(x_2[n]) \tag{3.26}$$

e homogeneidade:

$$\mathcal{Z}(\lambda x[n]) = \lambda \mathcal{Z}(x[n]) \tag{3.27}$$

Um aspecto importante da transformada Z é a sua relação com atrasos no tempo. Seja $x[n]$ um sinal no tempo e $X(z)$ sua transformada Z, a transformada desse sinal atrasado por k amostras é:

[4] Considerando a representação do número como ponto flutuante e inteiro como equivalentes.

[5] Essa é, a rigor, a transformada Z bilateral. É possível definir a transformada Z da mesma forma a menos da faixa de valores de n, iniciando a somatória a partir do zero. Entretanto, é simples restringir o domínio de $x[n]$ através do produto com a função de Heaviside $u[n]$, e essa é a maneira como Oppenheim [OSB99] lida com transformadas Z.

$$\mathcal{Z}(x[n-k]) = \sum_{n=-\infty}^{\infty} x[n-k]z^{-n} \tag{3.28}$$

$$= \sum_{m=-\infty}^{\infty} x[m]z^{-(m+k)} \tag{3.29}$$

$$= \sum_{m=-\infty}^{\infty} x[m]z^{-m}z^{-k} \tag{3.30}$$

$$= z^{-k} \sum_{m=-\infty}^{\infty} x[m]z^{-m} \tag{3.31}$$

$$= z^{-k}X(z) \tag{3.32}$$

Dessa forma, é possível representar atrasos de k amostras no domínio do tempo através da multiplicação por elementos z^{-k} no domínio da frequência. Pode-se utilizar tal resultado para representar sistemas LTI causais (eq. 3.5) de uma maneira alternativa, aplicando a transformada Z em ambas as partes da igualdade:

$$Y(z) = \sum_{i=0}^{M-1} b_i X(z)z^{-i} + \sum_{i=1}^{N} a_i Y(z)z^{-i} \tag{3.33}$$

$$Y(z) = X(z) \sum_{i=0}^{M-1} b_i z^{-i} + Y(z) \sum_{i=1}^{N} a_i z^{-i} \tag{3.34}$$

$$(1 - \sum_{i=1}^{N} a_i z^{-i})Y(z) = X(z) \sum_{i=0}^{M-1} b_i z^{-i} \tag{3.35}$$

em que $Y(z) = \mathcal{Z}(y[n])$ e $X(z) = \mathcal{Z}(x[n])$. Admitindo $H(z) = Y(z)/X(z)$, temos:

$$H(z) = \frac{\sum_{i=0}^{M-1} b_i z^{-i}}{(1 - \sum_{i=1}^{N} a_i z^{-i})} \tag{3.36}$$

À equação 3.36 é dada o nome de fórmula do sistema ou equação do sistema. Quando a entrada $x[n]$ aplicada ao sistema é o impulso $\delta[n]$, temos que $X(z) = \mathcal{Z}(\delta[n]) = 1$, e como $Y(z) = H(z)X(z)$, temos que $H(z)$ representa a saída do sistema definido pela equação 3.36, ou equivalentemente pela equação 3.5 quando a este é aplicado o impulso como entrada. Em outras palavras, $H(z)$ é a transformada Z da resposta ao impulso do sistema linear. Isso pode ser exemplificado através dos filtros definidos na seção anterior. O filtro de média móvel pode ser reescrito como aquele que possui como equação do sistema:

$$H(z) = \frac{1}{M} \sum_{i=0}^{M-1} z^{-i} \tag{3.37}$$

A presença apenas do numerador (ou de apenas uma constante como denominador, que no caso é 1) denota que o filtro é FIR, isto é, ele depende apenas das entradas para obter sua saída, não havendo retroalimentação. Entretanto, o inverso não é válido, isto é, um filtro com denominador em sua equação do sistema não necessariamente é um filtro FIR. Multiplicando a equação 3.37 por z^{-1}

$$z^{-1}H(z) = \frac{1}{M}\sum_{i=0}^{M-1} z^{-(i+1)} \tag{3.38}$$

$$z^{-1}H(z) = \frac{1}{M}\sum_{k=1}^{M} z^{-k} \tag{3.39}$$

subtraindo o resultado da própria equação 3.37, temos:

$$H(z) - z^{-1}H(z) = \frac{1}{M}\sum_{i=0}^{M-1} z^{-i} - \frac{1}{M}\sum_{k=1}^{M} z^{-k} \tag{3.40}$$

$$(1 - z^{-1})H(z) = \frac{1}{M}\left(\sum_{i=0}^{M-1} z^{-i} - \sum_{k=1}^{M} z^{-k}\right) \tag{3.41}$$

$$(1 - z^{-1})H(z) = \frac{1}{M}\left(1 + \cancel{\sum_{i=1}^{M-1} z^{-i}} - \cancel{\sum_{k=1}^{M-1} z^{-k}} - z^{-M}\right) \tag{3.42}$$

$$H(z) = \frac{1 - z^{-M}}{M(1 - z^{-1})} \tag{3.43}$$

De forma que o filtro FIR representado pela média móvel pode ser implementado facilmente através de um sistema recursivo.

Da mesma forma, o sistema ou filtro acumulador possui uma equação do sistema que pode ser obtida através da aplicação da transformada Z em ambos os termos da igualdade de seu equacionamento descrito na equação 3.23, resultando em:

$$Y(z) = X(z) + Y(z)z^{-1} \tag{3.44}$$

$$(1 - z^{-1})Y(z) = X(z) \tag{3.45}$$

$$H(z) = \frac{1}{(1 - z^{-1})} \tag{3.46}$$

Esse é um aspecto importante e de fundamentação da AudioLazy, a qual utiliza como base a equação de sistema dos filtros para representá-los em código. Os dois filtros apresentados são implementados abaixo:

Código-fonte 3.3: *Média movel e acumulador utilizando o objeto z*

```
In [1]: from audiolazy import z, Stream, maverage

In [2]: M = 5

In [3]: media_movel_5 = (1 - z ** -M) / (M * (1 - z ** -1))

In [4]: acumulador = 1 / (1 - z ** -1)

In [5]: media_movel_5(Stream(5)).take(10)
Out[5]: [1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0]

In [6]: acumulador(Stream(5)).take(10)
Out[6]: [5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0]

In [7]: maverage.recursive(4)
Out[7]:
```

```
0.25 - 0.25 * z^-4
-------------------
      1 - z^-1
```

Há três estratégias do filtro média móvel implementadas e disponíveis com o AudioLazy sob o dicionário de estratégias *maverage*, uma das quais exemplificada no último código fonte fornecido, indicando que ela resulta exatamente no filtro formulado com a retroalimentação, o que torna desnecessário para o usuário, no caso desse filtro, o conhecimento prévio de tal formulação. Pode-se perceber a partir do exemplo que o filtro é admitido com condições iniciais nulas, e como a entrada e a definição dos filtros do último exemplo não inclui nenhum número em ponto flutuante, pode-se notar que o valor zero utilizado está em ponto flutuante. Há maiores informações sobre esse tipo de consideração na seção 3.2.2.

Retomando o equacionamento da equação do sistema conforme descrito na equação 3.36, pode-se observar que tanto o numerador como o denominador podem ser vistos como polinômios em z^{-1}, bastando multiplicar ambos o numerador e o denominador por z^N para expressar o mesmo equacionamento como polinômios em z. A cada valor de z que torna nulo o valor do numerador é dado o nome de *zero*, e a cada valor de z que torna nulo o valor do denominador é dado o nome de *polo*. Seja o seguinte filtro de exemplo:

$$H(z) = \frac{1}{9} \cdot \frac{4 - z^{-2}}{1 + (4/5)z^{-1} + (2/5)z^{-2}} \tag{3.47}$$

Os polos e zeros podem ser representados em um diagrama, chamado *plano z*, que é precisamente o plano complexo de Argand-Gauss para a representação de números complexos. No caso do filtro acima, temos dois zeros iguais a 2 e -2, e dois polos complexos conjugados internos ao círculo unitário.

A AudioLazy permite facilmente a criação de tais diagramas, através do método *zplot*() presente nas classes que representam ou lidam com filtros lineares[6]:

Código-fonte 3.4: *Exemplo de diagrama de zeros e polos com a AudioLazy*

```
from audiolazy import z
filt = 1./9. * (4 - z ** -2) / (1 + .8 * z ** -1 + .4 * z ** -2)
fig = filt.zplot()
# Escolher uma das linhas abaixo
fig.show() # Para exibir na tela
fig.savefig("zero_polo.pdf") # Para salvar o diagrama em PDF
```

Maiores detalhes sobre a integração do pacote AudioLazy com o MatPlotLib, necessária para a realização do diagrama de zeros e polos conforme descrito no último código-fonte, encontram-se na seção 3.2.4. O diagrama gerado pode ser visto na figura 3.1.

A relação do equacionamento da transformada Z com componentes frequenciais do sinal de entrada pode ser obtida utilizando-se $z = e^{j\omega}$, isto é, restringindo o domínio da transformada Z ao círculo unitário, domínio este indicado em tracejado na figura 3.1. O equacionamento resultante a essa restrição é conhecido como DTFT (*Discrete Time Fourier Transform*, ou transformada de Fourier em tempo discreto), e se refere à decomposição do sinal (que, no caso da figura, é a resposta ao impulso do filtro) em senoides, analogamente à transformada de Fourier descrita na seção 1.1.1, que tinha como domínio de aplicação o conjunto $\mathbb{R}$ dos números reais. Explicitamente, a DTFT de um sinal $x[n]$ discretizado no tempo e conhecido em todo o domínio $\mathbb{Z}$ dos números inteiros é:

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{3.48}$$

O equacionamento mantém o valor da exponencial complexa para indicar que se trata de um caso particular

[6] Necessário dizer, as figuras possuem a legenda originalmente dada em inglês. Um *monkey patch* foi feito para possibilitar que as legendas e título pudessem ser traduzidas para o português sem a necessidade de preocupação por parte do desenvolvedor. *Monkey patch* é a possibilidade de alteração do código em tempo de execução. O código-fonte 9, referente a esse procedimento, encontra-se no final do segundo apêndice.

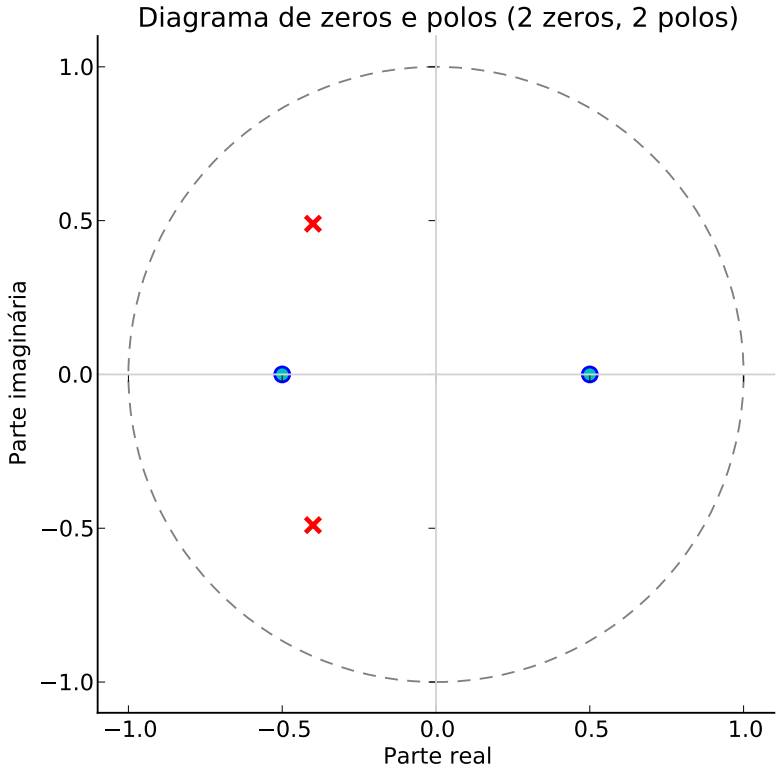


Figura 3.1: *Exemplo de diagrama de zeros e polos com a AudioLazy*

da transformada Z. A inversa da DTFT é dada através da equação:

$$x[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(e^{j\omega}) e^{j\omega n} d\omega \tag{3.49}$$

que é, precisamente, uma soma (integral) de componentes senoidais (complexas, dado que $e^{j\Phi} = \cos(\Phi) + j\sin(\Phi)$, equação esta conhecida como fórmula de Euler). Pode-se provar que os equacionamentos da DTFT e de sua inversa são processos que se revertem aplicando uma equação na outra, e recomenda-se a leitura de Oppenheim [OSB99, p. 48-51] para maiores detalhes sobre esse procedimento.

A representação da DTFT permite a obtenção de uma informação importante dos filtros definidos por suas equações de sistema: a resposta em frequência. Para esse fim, basta utilizar a restrição $z = e^{j\omega}$ sobre a equação de sistema do filtro para cada valor de ω dado representando uma componente frequencial (em radianos por amostra). A rigor, a resposta $H(e^{j\omega})$ para um dado valor ω de frequência é um número complexo que pode ser representado como um ganho em amplitude $|H(e^{j\omega})|$ (ou atenuação, caso seja menor do que 1) e uma defasagem $\angle H(e^{j\omega})$ em radianos, representando o deslocamento realizado à fase de uma componente senoidal de entrada.

Para exemplificar a importância da DTFT, retomemos o filtro definido através da equação 3.47. Modificando uma única letra no código-fonte 3.4, substituímos o método *zplot*() pelo método *plot*(), a fim de obter os gráficos de resposta em frequência do filtro fornecido, cujo resultado encontra-se exposto na figura 3.2.

3.2 Filtros digitais, síntese e análise na AudioLazy

É difícil criar uma interface que seja capaz de indicar para todo e qualquer objeto se este é um filtro digital ou não. A tarefa básica de um filtro é a presença de uma sequência de entrada e a devolução de uma sequência como saída. Dessa forma, filtros devem ser "chamáveis" como funções, a exemplo do que ocorre nas listagens da seção 3.1.1, e seu primeiro parâmetro deve ser um objeto iterável, representando o sinal digital. A saída de filtros é um objeto Stream, iterável de amostras de saída do filtro, o qual avalia a entrada

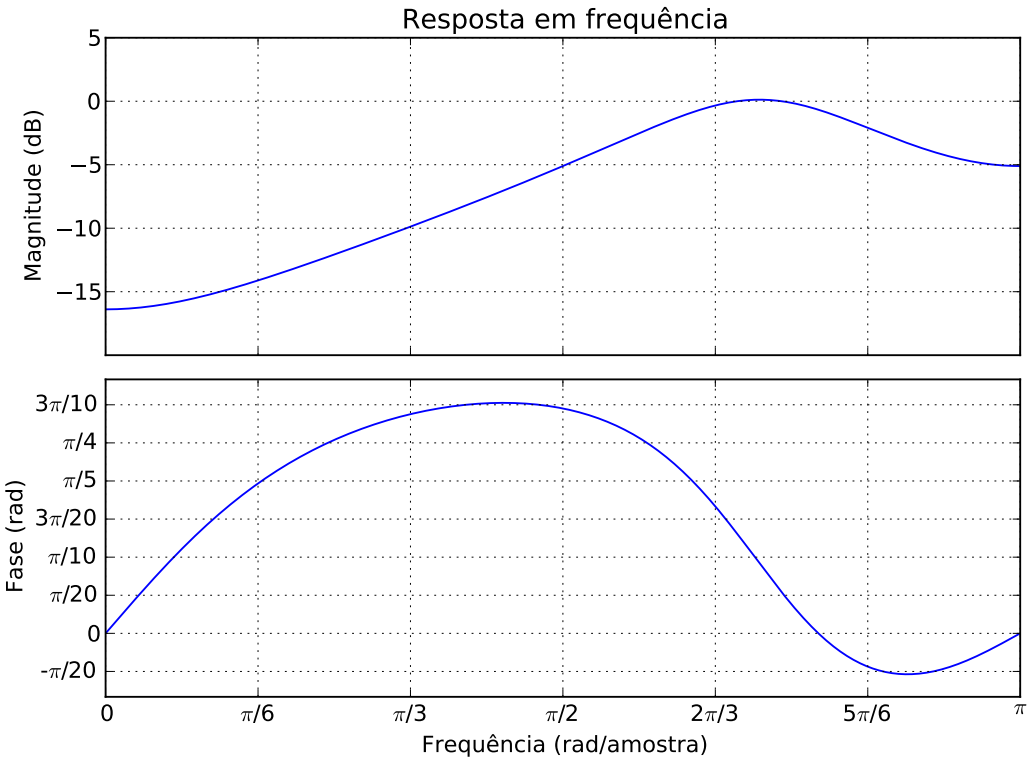


Figura 3.2: *Exemplo de gráfico resposta em frequência com a AudioLazy*

sob demanda, isto é, a avaliação dos iteráveis dos filtros é tardia. Entretanto, visando facilitar a conexão com componentes externos à AudioLazy, não existe uma classe de filtros na AudioLazy que generalize filtros lineares e não lineares, de forma que quaisquer funções ou objetos chamáveis podem ser utilizados como filtros, embora a utilização de uma entrada ou saída que não seja correspondente à descrição dada possa resultar em erro, dependendo das conexões.

Na seção anterior já foram vistos alguns casos de uso de filtros e a sintaxe básica para a utilização de filtros na AudioLazy. Esta seção visa especificar em maior detalhe as características gerais dos filtros implementados através das classes *LinearFilter*, *ZFilter*, *CascadeFilter* e *ParallelFilter*, além do objeto z que permite a criação dos filtros já vistos. Uma característica geral da maioria das classes envolvidas nessa seção é o uso do *AbstractOperatorOverloaderMeta* para sobrecarga massiva de operadores, algo que está detalhado na seção 2.3.4 e não precisou de maior aprofundamento nesta seção. Outra classe importante para os filtros é a classe *Poly*, a qual surgiu da refatoração dos primeiros filtros criados para a AudioLazy, trazendo uma forma natural de organização de filtros através de suas equações de sistema.

3.2.1 Poly: Polinômios, polinômios de Laurent, soma de potências

A classe Poly é fundamental para possibilitar a representação dos filtros como equações escritas diretamente através de sua transformada Z. Em linguagens como o MatLab, o Octave e até mesmo no NumPy e SciPy (subpacote de processamento de sinais), o polinômio é implementado como um vetor de coeficientes tal como $[1, 2, 1]$, representando, a menos do SciPy, o polinômio $x^2 + 2x + 1$. Nas linguagens citadas, os polinômios são sempre representados como vetores de coeficientes em ordem decrescente de expoente. Isso faz com que o índice no vetor não corresponda ao expoente que possui aquele coeficiente, isto é, o vetor v de coeficientes de tamanho 3 representa o polinômio $v[0]x^2 + v[1]x + v[2]$ no Python (índices começam em 0), e $v[1]x^2 + v[2]x + v[3]$ no MatLab e Octave (índices começam em 1). Mesmo que seja simples manter a regra de que os coeficientes estão em ordem decrescente de expoente, o caso fica um pouco mais confuso quando se trata de vetores contendo o par de polinômios que definem filtros. Um filtro FIR definido como $1 - z^{-1} + 2z^{-2}$ é representado como $[1, -1, 2]$, e dessa vez mantendo uma relação interessante com os

índices do vetor, particularmente no Python, em que o i-ésimo coeficiente é o que multiplica o componente z^{-i}. A soma e subtração de polinômios neste caso é um simples procedimento elemento-a-elemento, bastando garantir que os polinômios tenham o mesmo tamanho, algo que pode ser feito através da inserção de zeros ao vetor. A multiplicação de polinômios, embora possível, já se torna uma tarefa menos direta.

O NumPy facilita essas operações sobre polinômios através de suas funções *polyadd*, *polysub* e *polymul*, além da avaliação de valores aplicados ao polinômio através do *polyval*, este último também presente no Octave. Talvez por conta de uma insatisfação com a expressividade ou com a fragmentação excessiva de funções que lidam com polinômios, o Numpy inclui a classe *poly1d*, a qual permite o uso de operadores sobre os polinômios, além de métodos para derivação, integração e obtenção de raízes. O construtor recebe um vetor como parâmetro, e o acesso ao vetor de coeficientes pode ser realizado através da propriedade *poly1d.coeffs*.

Com base no objetivo de possibilitar e enfatizar a expressividade do código, a AudioLazy não foi criada para ser um pacote que impusesse ao desenvolvedor a necessidade de organização dos dados através de vetores e matrizes, embora tenha sido criada para aceitar essa possibilidade. Mesmo que um polinômio de ordem N possa ser representado como um vetor de tamanho $N + 1$, essa representação impõe que o polinômio ocupe um espaço de memória $O(N)$, e que algoritmos que o utilizem tenham complexidade no mínimo linear (ainda que apenas para acessar todos os seus elementos). No caso de um polinômio como $1 + x^{500}$, teríamos 499 posições de memória com zeros armazenados, desnecessariamente. Visando evitar esse tipo de situação, a organização interna dos polinômios na AudioLazy é feita através de um dicionário, isto é, o polinômio é implementado como um conjunto de pares {*expoente*: emphcoeficiente}, na classe Poly. Dessa forma, a complexidade resultante se baseia no número de coeficientes não nulos do polinômio. Algumas consequências que ocorreram a partir dessa decisão, quando somadas à tipagem dinâmica do Python, foram:

1. O valor máximo $N \in \mathbb{N}$ para um expoente ser implementado é superior a qualquer necessidade prática, seja por conta da facilidade de representação de polinômios esparsos, seja por conta da representação de inteiros do Python não ser limitada pela linguagem;

2. A avaliação do polinômio não é feita através do algoritmo de Horner, como ocorre com o NumPy, mas com uma adaptação desse algoritmo voltada para o uso na organização dos coeficientes em um dicionário. A adaptação refere-se a utilizar a diferença entre expoentes adjacentes, permitindo que a complexidade computacional seja linear com relação ao número de coeficientes não nulos, ao invés de linear com relação ao valor do maior coeficiente;

3. Valores negativos de expoentes são possíveis, possibilitando que os objetos da classe sejam polinômios de Laurent. O argumento sobre o valor máximo do expoente no primeiro item apresentado também vale para o valor mínimo (i.e., mais negativo);

4. Valores fracionários de expoentes são possíveis, possibilitando que os objetos da classe sejam quaisquer somas de potências;

5. Coeficientes não precisam ser números, possibilitando que polinômios sejam variáveis no tempo através do uso de objetos Stream como coeficientes;

Na AudioLazy, tais polinômios podem ser criados a partir de uma lista de coeficientes (como ocorre no NumPy), a partir de um dicionário de pares conforme o especificado, ou como o resultado de operações envolvendo outros polinômios. O único recurso que necessita do NumPy é a obtenção de raízes do polinômio, recurso restrito a polinômios de expoentes em $\mathbb{N}$. Um exemplo dos recursos envolvendo polinômios na AudioLazy encontra-se no código-fonte que segue.

Código-fonte 3.5: *Exemplo com objetos Poly*

```
In [1]: from audiolazy import Poly, Stream

In [2]: #Exemplo 1

In [3]: p = Poly([1, 2, -1])
```

```
In [4]: p
Out[4]: 1 + 2 * x - x^2

In [5]: p.roots # Utilizando o NumPy, internamente
Out[5]: [2.414213562373095, -0.4142135623730951]

In [6]: p(10) # Aplicando x = 10
Out[6]: -79.0

In [7]: p[2] # Consulta ao coeficiente do dado expoente
Out[7]: -1

In [8]: # Exemplo 2

In [9]: coef_variavel = Stream(0, 1) # [0, 1, 0, 1, 0, 1, 0, 1, ...]

In [10]: p2 = Poly({-1: 20, 2: coef_variavel})

In [11]: p2 # Um Poly variável no tempo
Out[11]: 20 * x^-1 + a2 * x^2

In [12]: d = p2(10)

In [13]: d # Sim, um Stream!
Out[13]: <audiolazy.lazy_stream.Stream at 0x3c14ad0>

In [14]: d.take(10)
Out[14]: [2.0, 102.0, 2.0, 102.0, 2.0, 102.0, 2.0, 102.0, 2.0, 102.0]
```

O texto a_i é utilizado automaticamente para caracterizar o i-ésimo coeficiente do polinômio, caso este seja variável (i.e., um objeto Stream). O construtor a partir de listas utiliza a ordem inversa à do NumPy, visando porém garantir que o i-ésimo coeficiente, iniciando do zero, é o coeficiente com expoente igual a i.

3.2.2 Filtros lineares e listas de filtros

A parte de filtros lineares foi uma das primeiras a serem inseridas na AudioLazy, e refere-se a um dos pontos centrais do desenvolvimento. Esses filtros foram originalmente baseados nos filtros do SciPy, que requisita como parâmetros de entrada 2 vetores correspondentes ao polinômio do numerador e ao polinômio do denominador do filtro linear, pensado como sua equação de sistema (eq. 3.36). Como representado no código-fonte 2.12, o construtor dos filtros da AudioLazy através da classe *ZFilter* e a função de aplicação de filtros *scipy.signal.lfilter* são compatíveis, no sentido de que ambas mantém o mesmo comportamento para um dado par de listas representando os polinômios do numerador e do denominador do filtro. Em outras palavras, embora o sistema de polinômios seja explicitamente diferente, o fato da classe *ZFilter* lidar com polinômios tais que $x = z^{-1}$ faz com que a ordem dos elementos na lista que representa o polinômio seja a mesma utilizada pelo SciPy.

Filtros lineares, na AudioLazy, são pares de polinômios, e encontram-se definidos na classe *LinearFilter*. Estes podem ser acessados a partir das propriedades *numpoly* e *denpoly*. A classe *LinearFilterProperties* pode ser vista como um *mixin*[7] que insere outras propriedades, dependentes da existência dos dois polinômios. Essas propriedades inseridas se referem a conversões entre estruturas de dados. Por exemplo, o par de propriedades *numlist* e *denlist* cria uma lista de coeficientes representando o polinômio do numerador ou do denominador, respectivamente. Por razões históricas do desenvolvimento da AudioLazy, essas últimas duas propriedades citadas também possuem os nomes de *numerator* e *denominator*, respectivamente. Outras propriedades incluem *numdict* e *dendict* (conversão em dicionários ordenados por expoente) e *numpolyz* e *denpolyz*, que

[7] Literalmente "misturar em". Padrão de desenvolvimento de software similar ao *plug-in*, porém aplicado a classes (por herança), ao invés de instâncias.

representam os polinômios admitindo $x = z$, ao invés de $x = z^{-1}$. Todas as 8 propriedades inseridas pela classe *LinearFilterProperties* criam dinamicamente o objeto requisitado, e não aceitam atribuição.

Objetos *LinearFilter* são chamáveis, isto é, comportam-se como funções, aceitando uma sequência de entrada. Por padrão, a aplicação do filtro é feita com condições iniciais nulas, mas é possível contornar isso através do uso do argumento nominado "*zero*", de forma a, por exemplo, manter todos os valores inteiros, ao invés de ponto flutuante (o zero padrão está em ponto flutuante). A memória das últimas saídas fornecidas pelo filtro também pode ser especificada, através do argumento nominado "*memory*". Essa memória pode ser qualquer iterável contendo os valores a serem utilizados, ou uma função a ser chamada com um único parâmetro igual ao número de amostras de saída que precisa ser armazenado (tamanho da memória). Um exemplo de uso da memória ocorre no algoritmo de Karplus-Strong (seção 3.2.10).

Originalmente (versões 0.01 e 0.02), filtros lineares correspondiam a uma classe de nome LTI, a qual foi renomeada para LinearFilter, e os filtros com representação no domínio da frequência correspondiam à classe LTIFreq, renomeada para ZFilter. A mudança dos nomes não reflete apenas uma escolha cosmética, mas trata de um aspecto importante que mudou no decorrer do desenvolvimento: os filtros passaram a aceitar coeficientes variantes no tempo, através de objetos Stream, de maneira similar à vista com polinômios na seção 3.2.1.

Fundamentalmente, os *ZFilter* são apenas filtros lineares como os *LinearFilter*, mas com os operadores sobrecarregados para permitir o uso como equações de sistema. O objeto z é apenas uma instância comum `z = ZFilter({-1: 1})`, isto é, um atraso negativo de uma unidade associada a um ganho unitário. Além dos operadores, os objetos *ZFilter* permitem a derivação e a substituição (e.g., seja $H(z) = z + 1$, logo $H(1/z)$ é igual a $1 + z^{-1}$, i.e., chamadas do objeto *ZFilter* com um *ZFilter* como parâmetro efetuam a composição de funções, devolvendo outro objeto *ZFilter*).

Resumidamente, alguns aspectos importantes com relação aos filtros lineares incluem:

- São chamáveis, de maneira que o primeiro argumento posicional deve ser uma sequência (e.g. um objeto *Stream* ou uma lista);
- Devolvem um objeto *Stream*;
- Permitem consulta a suas informações:
  1. Método *freq_response*: Devolve a resposta em frequência para uma dada frequência ou um iterável de frequências;
  2. Método *is_causal*: Devolve um booleano indicando se o filtro é ou não causal. Isso significa que filtros não-causais (atrasos negativos) são representáveis, visto que podem ser úteis como representações intermediárias para se chegar a um dado filtro como objetivo;
  3. Método *is_lti*: Verifica se nenhum coeficiente é iterável, i.e., se o filtro não é variante no tempo;
  4. Propriedades *zeros* e *poles*: Devolve a lista de zeros ou polos do filtro, respectivamente.
- Suportam a comparação por proximidade numérica com *almost_eq* e *almost_eq_diff* (seção 2.3.7);
- Podem ser variantes no tempo;
- São organizados como pares de polinômio;
- Permitem a criação de gráficos através dos métodos *plot*() (resposta em frequência) e *zplot*() (diagrama de zeros e polos).

Apesar da itemização acima se referir às instâncias da classe *LinearFilter*, o mesmo se aplica aos filtros em cascata (classe *CascadeFilter*) e aos filtros em paralelo (classe *ParallelFilter*). Esses últimos são listas de filtros (classe *FilterList*) que se comportam como filtros, e há maiores detalhes sobre eles na seção que segue. Possuem também o método *is_linear* de consulta aos filtros da lista, a fim de assegurar que o filtro resultante é linear (único caso em que a maioria dos recursos disponibilizados às listas de filtros fazem sentido). Um recurso importante das listas de filtros é a aceitação de valores constantes de ganho como filtros, convertendo-os em chamáveis quando necessário (e.g., ao chamar um objeto *CascadeFilter*).

3.2.3 JIT em Python e a necessidade de diferentes configurações

Assim como ocorreu no caso da média móvel, os mesmos filtros digitais podem ser implementados de diversas maneiras ou configurações. Há uma lista abrangente de possibilidades descritas por Oppenheim [OSB99, capítulo 6], e a nomenclatura que segue é utilizada por essa referência. A maneira mais imediata é o uso de laços como realizado no código-fonte 3.2, em que os valores passados são guardados em variáveis. Na presença de entradas, essas também seriam guardadas. Os objetos *LinearFilter* implementados na AudioLazy realizam o que é chamado de forma direta I, isto é, armazenam as entradas e saídas para realizar o cálculo conforme a equação 3.5 para cada amostra de entrada. A forma direta II refere-se a uma maneira de diminuir o número de blocos de atraso necessários, criando um sinal intermediário que se utiliza de parte da entrada e parte do histórico de saída do filtro, mas esta não foi inserida na AudioLazy.

Filtros em cascata são filtros que realizam a composição serial de outros filtros, isto é, a aplicação de um filtro ocorre sobre a saída de um filtro anterior. O efeito é similar ao da multiplicação das equações de sistemas, mas uma importante diferença é que filtros em cascata não possuem as mesmas dificuldades relativas à precisão numérica. Filtros em paralelo são outra alternativa, que consiste em aplicar todos os seus diversos filtros componentes diretamente ao sinal de entrada, e então devolver a soma dos resultados. Vale salientar que não existe paralelismo no sentido de execução simultânea nessa classe, embora isso também seja uma possibilidade de implementação.

Mais importante é dizer, entretanto, que os filtros são compilados sob demanda e de maneira específica para um dado filtro. Existe um compilador JIT (*Just In Time*) que, dada a requisição de uso do filtro (i.e., o filtro é chamado para aplicação sobre uma entrada), cria o objeto que realiza o processamento. Em outras palavras, um código-fonte em Python análogo ao do filtro IIR no código-fonte 3.2 é criado em tempo de execução representando o filtro que recebeu uma requisição de uso, sendo qeu esse código é compilado com o próprio Python e executado com o comando *exec*, em tempo de execução. O resultado devolvido é o resultado desse novo objeto criado. Isso possibilita que filtros mais complicados sejam feitos sem comprometer o desempenho. A inserção dessa capacidade na AudioLazy não teve, originalmente, a otimização como única meta: um dos principais objetivos foi o de possibilitar a implementação de filtros variantes no tempo, algo necessário para a implementação de, por exemplo, o modelo de síntese de trompete de Horner-Beauchamp [HB95], algo que foi utilizado para avaliar o algoritmo de Klapuri [Kla08]. Alguma das características desse código auto-gerado para compilação JIT é

- Coeficientes constantes iguais a 1 não precisam ser multiplicados, bastando somar o elemento correspondente ao resultado em cada iteração;
- Coeficientes constantes iguais a -1 são denotados através de um operador unário ao invés de uma multiplicação;
- Coeficientes constantes iguais a 0 significam que o componente não precisa ser somado ao resultado;
- Coeficientes iteráveis devem requisitar seu próximo valor uma única vez antes de cada uso como coeficiente.

Por conta do último item listado, o JIT se fez necessário como uma maneira simples e eficiente de implementação. Segue abaixo um simples exemplo de filtro variante no tempo, em que o denominador do filtro é periódico assim como a entrada, embora não seja necessário em geral nem que o período seja igual, nem que qualquer um dos coeficientes variantes no tempo seja periódico. Variações muito rápidas nos coeficientes do filtro trazem consigo uma dificuldade de interpretação, dado que eles não irão mais se referir à resposta em frequência, aos polos ou aos zeros esperados, mas fazem sentido quando o filtro é visto como uma combinação linear de atrasos no tempo, como ocorre na equação 3.5.

Código-fonte 3.6: *Exemplo de filtro variante no tempo*

```
In [1]: from audiolazy import z, Stream

In [2]: filt = (1 - z ** -3) / (1 - Stream(.3, .2, .1) * z ** -1)
```

```
In [3]: filt(Stream(5, 10, 0)).take(5)
Out[3]: [5.0, 11.0, 1.1, 0.33, 0.066]

In [4]: filt
Out[4]:
  1 - z^-3
-------------
1 + a1 * z^-1
```

Para o filtro emphfilt criado no código-fonte 3.6, o código-fonte 3.7 escrito em Python foi gerado automaticamente pelo filtro, e compilado para uso imediato (JIT) como função geradora para a realização do processo de filtragem:

Código-fonte 3.7: *Exemplo de código de filtro gerado pelo JIT*

```
1  def gen(seq, a1):
2    m1 = 0.0
3    d1 = 0.0
4    d2 = 0.0
5    d3 = 0.0
6    for d0 in seq:
7      m0 = d0 + -d3 + -m1 * a1.next()
8      yield m0
9      m1 = m0
10     d3 = d2
11     d2 = d1
12     d1 = d0
```

3.2.4 Integração com outras estruturas de dados e outros pacotes

Como já foi visto com os polinômios na seção 3.2.1, existe uma dependência com o NumPy para a obtenção dos valores das raízes de um polinômio. Observando o código-fonte do NumPy[8], pode-se observar que este cria uma matriz específica com os coeficientes do polinômio de forma a possuir os autovalores idênticos às raízes do polinômio. A partir disso, os autovalores são obtidos por meio de rotinas internas ao NumPy que utilizam o LAPACK[9]. É possível otimizar este procedimento, no sentido de que a matriz gerada é esparsa e que o cálculo dos autovalores pode ser feito levando em consideração tal característica a fim de evitar o armazenamento e o acesso desnecessário de recursos. Entretanto, para os fins práticos de pontuais análises de equações de filtros, tal tarefa não necessita de uma tamanha otimização, e esse empreendimento se fez desnecessário. Uma vantagem prática relevante de tal implementação seria a possibilidade de diminuir a dependência entre o NumPy e a AudioLazy, mas não se trata de algo fundamental.

As dependências são minimizadas no sentido de que estas são acionadas apenas quando necessárias. Apesar da dependência poder ser interpretada como algo indesejável, é extremamente importante que a AudioLazy mantenha a integração, isto é, a possibilidade de conexão entre elementos de pacotes distintos. Essa possibilidade de conexão existe com relação a diversos pacotes, dentre os quais se pode destacar o NumPy, o SciPy, o MatPlotLib e o Music21, que possuem exemplos (vide segundo apêndice) ou testes (e.g., o código-fonte 2.12) envolvendo a integração com a AudioLazy.

Para a obtenção de polos e zeros de filtros, é utilizada a propriedade *roots* da classe *Poly*, de forma que tais tarefas também dependem do NumPy. Para a elaboração do diagrama de polos e zeros do filtro (método

[8] Disponível em https://github.com/numpy/numpy/.
O arquivo acessado refere-se ao:
https://github.com/numpy/numpy/blob/master/numpy/lib/polynomial.py, último acesso dia 2013-02-18.
O processo é melhor descrito em:
R. A. Horn, C. R. Johnson. *Matrix Analysis*. Cambridge, Reino Unido: Cambridge University Press, 1999, p. 146-147.

[9] Pacote de álgebra linear do Fortran 90.
Disponível em http://www.netlib.org/lapack/

zplot() de filtros lineares), há uma integração clara da AudioLazy com o MatPlotLib, situação em que a dependência com relação ao NumPy passa a ser não apenas justificável, mas irrelevante, dado que o NumPy é uma dependência do MatPlotLib. Para a elaboração de gráficos de resposta em frequência (método *plot*() de filtros lineares), há também uma integração com o MatPlotLib. O valor devolvido pelo método *zplot*() é um objeto de figura do MatPlotLib que, como mostrado no código-fonte 3.4, permite que o diagrama seja exibido na tela ou salvo diretamente em PDF[10]. A menos da importação do pacote AudioLazy, nada proíbe que um filtro seja criado e exibido na tela (ou salvo em PDF) em uma única linha de código, como, por exemplo, *(1 - z ** -1).plot().show()*. Este é um dos resultados da integração da AudioLazy com o MatPlotLib, e talvez o mais significativo em termos de expressividade e concisão de código.

Facilitando a integração com outros pacotes e com as estruturas padrões do Python, foi privilegiado o uso do protocolo de iteração e de iteráveis em geral, tanto para coeficientes de filtros variáveis no tempo como para entradas de filtros, além de conversões envolvendo objetos *Stream*. A heterogeneidade dos valores em objetos *Stream* permite que estes incluam objetos quaisquer, tal como ocorre com os blocos (objetos *collections.deque*), mas também garante a funcionalidade com matrizes e vetores do NumPy como elementos, o que pode vir a ser útil quando em conjunto com o fato de que tais matrizes e vetores já possuem os operadores sobrecarregados, permitindo que os objetos *Stream* e filtros lidem com fluxos de dados *N*-dimensionais.

Segue um exemplo para ilustrar tal possibilidade, criando um *Stream* de vetores do NumPy, e aplicando-o a um acumulador (filtro linear), para posteriormente multiplicar o *Stream* do resultado por uma matriz de mudança de base e converter as matrizes resultantes em listas de listas através da chamada dos métodos *dot* (multiplicação de matrizes usando objetos *numpy.ndarray*) e *tolist* de cada elemento. Os métodos, propriedades ou atributos consultados de um objeto *Stream* são aplicados a cada elemento, a menos que seja um nome existente no próprio objeto ou classe *Stream*, e sua avaliação é tardia (i.e., os resultados intermediários são objetos *Stream*).

Código-fonte 3.8: *Exemplo de Stream bidimensional com filtros lineares*

```
In [1]: from audiolazy import z, repeat

In [2]: import numpy as np

In [3]: data = repeat(np.array([1, 3])) # Stream repetindo um iterável

In [4]: data
Out[4]: <audiolazy.lazy_stream.Stream at 0x3427950>

In [5]: data.take(3) # Vetores (ou matrizes linha)
Out[5]: [array([1, 3]), array([1, 3]), array([1, 3])]

In [6]: acc = 1 / (1 - z ** -1)

In [7]: acc(data).take(4) # Aplica o filtro à sequência de vetores
Out[7]: [array([ 1.,  3.]), array([ 2.,  6.]), array([ 3.,  9.]),
         array([  4.,  12.])]

In [8]: uma_matriz = np.matrix([[8, 1], [-7, -1]]) # P/ produto entre matrizes

In [9]: resultado = acc(data).dot(uma_matriz).tolist() # Métodos dos elementos

In [10]: resultado # Um stream de listas
Out[10]: <audiolazy.lazy_stream.Stream at 0x3427e10>

In [11]: resultado.take(4) # Mudança de base em um Stream de vetores
Out[11]: [[[-13.0, -2.0]], [[-26.0, -4.0]], [[-39.0, -6.0]], [[-52.0, -8.0]]]
```

[10] Nem todas as versões do MatPlotLib incluem o método *show* na classe *matplotlib.figure.Figure*. No MatPlotLib 1.2.0, esse recurso está presente. De qualquer forma, existem outras maneiras de se exibir na tela uma figura, mas não faz parte do presente trabalho entrar em maiores detalhes sobre a interface do MatPlotLib.

Além disso, a AudioLazy permite, até certo ponto, a integração com o SymPy, um pacote voltado à manipulação algébrica simbólica. Em certos aspectos, o objetivo do SymPy é significativamente diferente daquele que fundamentou a AudioLazy, no sentido de que o SymPy não espera manipular dados mas sim fórmulas, e o processamento de fórmulas matemáticas dificilmente necessita de ser realizado em tempo real. Porém, a integração citada faz com que os sinais da AudioLazy representados como objetos Stream possam ser parametrizados com base em símbolos. Seja um sinal discreto no tempo linear por partes cujos segmentos possuem 10 e 8 amostras, possuem amplitudes iguais a x e y e são ascendente e descendente, respectivamente, como na figura 3.3.

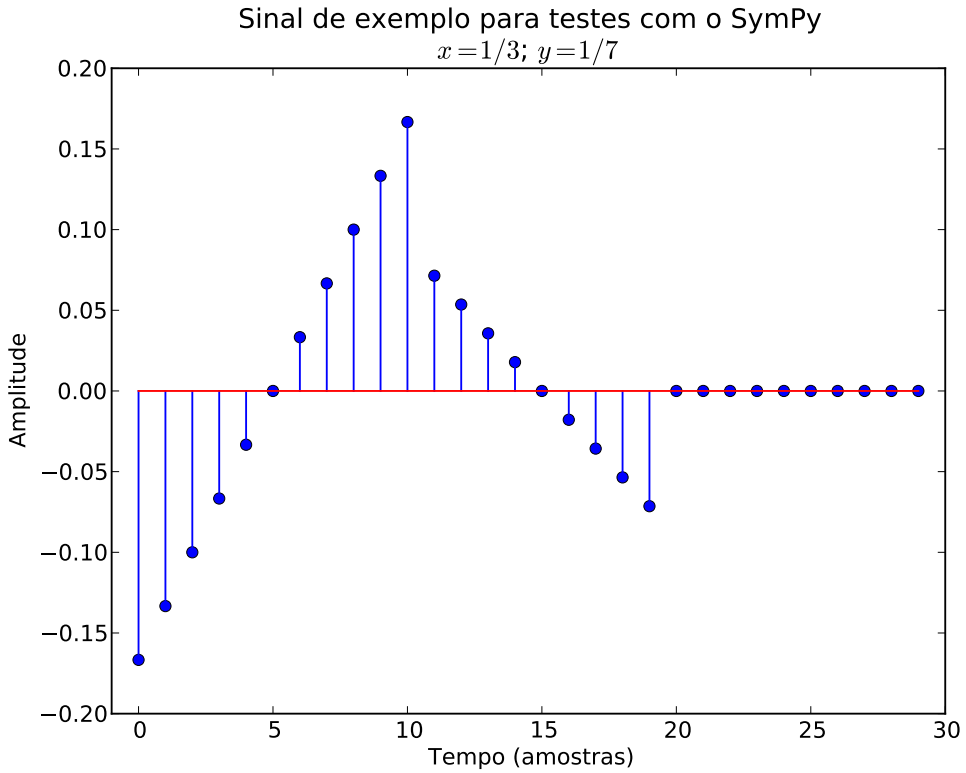


Figura 3.3: *Exemplo de sinal paramétrico usando o SymPy*

No código-fonte 3.9, temos a implementação do sinal descrito como dependente dos valores de amplitude x e y, o qual foi aplicado ao filtro digital:

$$H(z) = \frac{2}{2 + z^{-2} - z^{-7}} \tag{3.50}$$

O valor da 256-ésima amostra devolvida é uma função de x e y:

$$-\frac{4059613816817038761354557945525658 89}{4253529586511730793292182592897102643 20}x + \frac{7776633737493100619779614785983743}{5316911983139663491615228241121378304}y$$

Ou, arredondando e utilizando notação em ponto flutuante:

$$-0.000954410621637707 \cdot x + 0.00146262224429394 \cdot y$$

Que para $x = 1/3$ e $y = 1/7$ vale aproximadamente -0.000109190838980101, o mesmo resultado obtido na avaliação numérica usando ponto flutuante de 64 bits. Esse é um exemplo de aplicação da generalidade presente na classe *Stream*, a qual permite que os valores percorridos sejam de qualquer classe, inclusive símbolos matemáticos e equações matemáticas simbólicas do SymPy. Para a obtenção de valores de resposta

ao impulso muito distantes do inicial em filtros IIR, a utilização dessa abordagem pode possibilitar a descrição de seu valor com precisão arbitrária.

Código-fonte 3.9: *Exemplo de integração entre AudioLazy e SymPy*

```
import sympy
from audiolazy import z, Stream, chain

x, y = sympy.symbols("x y", complex=True) # Assim temos símbolos do SymPy
zero = x - x # Uma forma de obter o zero simbólico

sizex = 10
sizey = 8

# Cria a entrada parametrizada em x e y (Stream de símbolos)
data = chain([el * x / sizex for el in range(- sizex // 2, sizex // 2 + 1)],
             [el * y / sizey for el in range(- sizey // 2, sizey // 2 + 1)],
             Stream(zero))

# Valores para substituir posteriormente
xv = sympy.Rational(1, 3) # São números racionais de precisão arbitrária
yv = sympy.Rational(1, 7)

# Aplica o filtro duas vezes
filt = 2 / (2 + z ** -2 - z ** -7)
dur = 256
filtered1 = filt(data.copy(), zero=0).take(dur) # Primeiro em uma cópia
filtered2 = filt(data.subs(x, xv).subs(y, yv).evalf(), zero=0).take(dur)

# Exibe os dados
print filtered1[-1].simplify() # Resultado 100% simbólico
print repr(filtered1[-1].evalf())
print repr(filtered1[-1].subs(x, xv).subs(y, yv).evalf())
print repr(filtered2[-1]) # Resultado 100% numérico
```

3.2.5 Problemas acerca da representação em ponto flutuante em filtros

Durante a implementação do filtro gammatone, observou-se um problema presente no gráfico de sua resposta em frequência. Oppenheim [OSB99, p. 370-418] descreve casos de erros de arredondamento devido à quantização, com particular ênfase na influência desses erros nas diferentes configurações de implementação de filtros digitais. Nas figuras 3.4 e 3.5, encontra-se um filtro digital gammatone, de 8 polos e conforme a descrição de Slaney [Sla93]. O código-fonte 3.10 indica o código utilizado na AudioLazy para a elaboração dos gráficos. Cada uma das duas implementações difere apenas na maneira como os filtros são organizados: em um caso, é utilizado o filtro diretamente com sua equação que inclui 4 pares conjugados idênticos de polos, porém multiplicados, formando o denominador; no segundo caso, é utilizado um filtro em cascata. A taxa de amostragem utilizada é de 44100, e a frequência central de 100 Hz.

A diferença dos resultados é notável, e refere-se a apenas uma mudança na implementação. O ruído em torno de 100 Hz não tem como ser um resultado de um filtro com apenas 8 polos, sendo necessariamente um ruído que tem origem na proximidade com limites da representação numérica, e refere-se às operações de subtração presentes na aplicação do filtro como ocorre na equação 3.5, sobretudo em um filtro de ordem $N = 8$. Subtrações de números em ponto flutuante podem gerar grande ruído numérico, como discutido em 2.3.7, e é precisamente o que a figura 3.4 ilustra.

Para valores menores de taxa de amostragem e maiores de frequência central, tal ruído passa a ser menos significativo ou até mesmo desaparece, mas é possível notar, através desse exemplo, que não é confiável utilizar a forma direta I de implementação para filtros que envolvem pares conjugados de polos próximos entre

si. A solução para o problema foi a utilização de um filtro em cascata, que desacopla cada par de polos conjugados em uma aplicação separada sobre o sinal, isto é, aplicações de somas e subtrações conforme ocorre na equação 3.5 se restringem a filtros de ordem $N = 2$, embora ocorram 4 vezes.

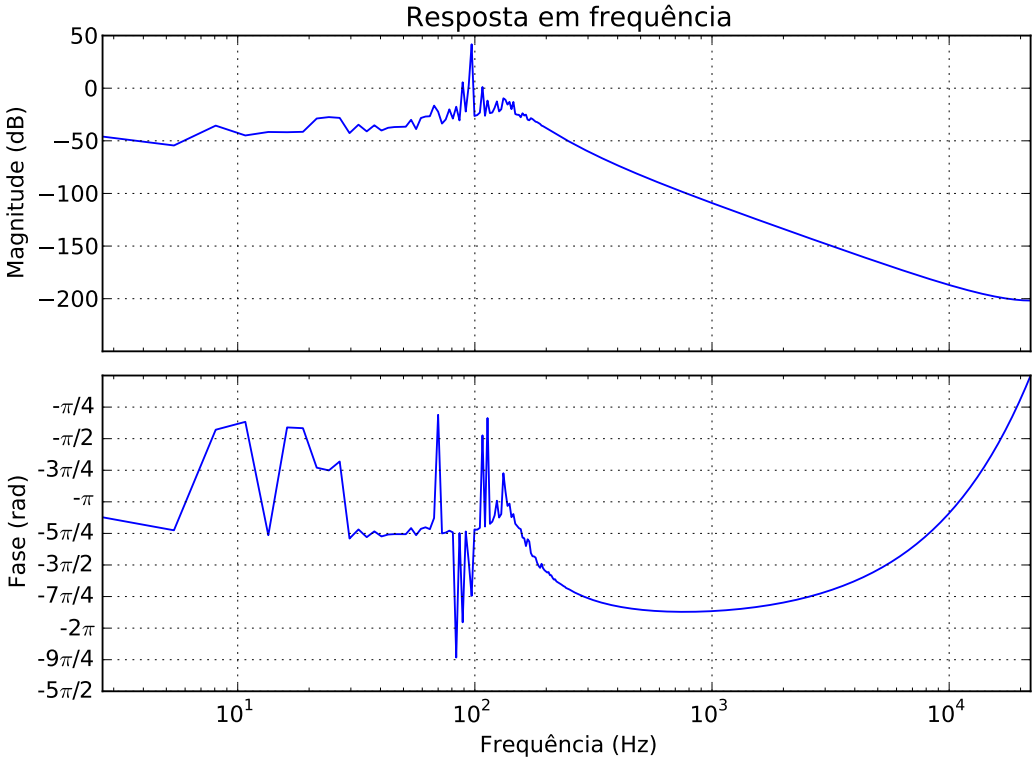


Figura 3.4: *Resposta em frequência de um filtro gammatone (forma direta I)*

Código-fonte 3.10: *Figuras para avaliação da influência da quantização*

```
rate = 44100
s, Hz = sHz(rate)
cg = gammatone_erb_constants(4)[0]
freq = 100 * Hz
bw = erb(freq, Hz)
fgamma = gammatone(freq, bw) # Devolve um CascadeFilter com 4 ressonadores
fgamma_multiplicado = fgamma[0] * fgamma[1] * fgamma[2] * fgamma[3]

# Figura da resposta em frequência do filtro de 8 polos
fgamma_multiplicado.plot(freq_scale="log", rate=rate, samples=8192).show()

# Figura da resposta em frequência do filtro em cascata
fgamma.plot(freq_scale="log", rate=rate, samples=8192).show()
```

3.2.6 Filtro gammatone paramétrico

A definição do filtro gammatone dada por Slaney [Sla93] é realizada através de sua resposta ao impulso (delta de Dirac):

$$g(t) = at^{\eta-1}e^{-2pib_w t}\cos(2\pi f_c t + \phi) \tag{3.51}$$

em que b_w é uma largura de banda em hertz, f_c é a frequência central do filtro em hertz, ϕ representa uma

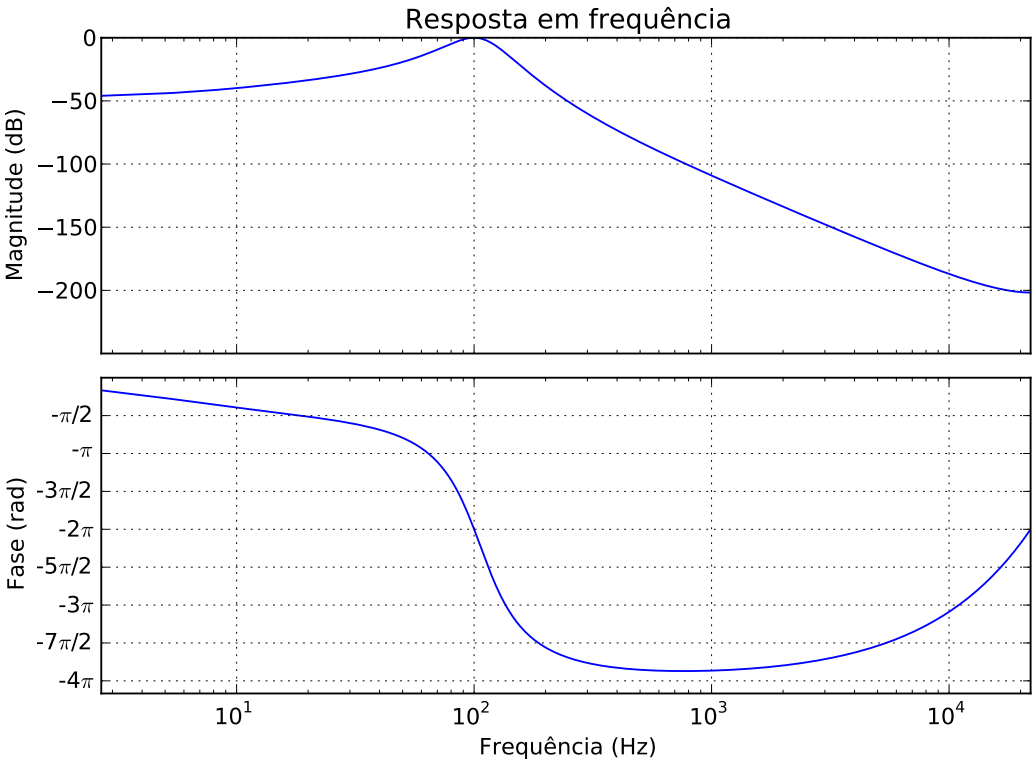


Figura 3.5: *Resposta em frequência de um filtro gammatone (cascata)*

defasagem em radianos e $t \geq 0$ é um instante de tempo em segundos. O valor de η representa a ordem do gammatone, que é fixo em 4 na implementação feita por Slaney. O valor do ganho a refere-se a um fator de escala para garantir que o ganho seja de 0 dB (i.e., unitário) na frequência central f_c. Como é descrito no texto que segue, a ordem do gammatone é metade da ordem do filtro resultante. A técnica utilizada por Slaney para chegar ao equacionamento do filtro digital é o casamento de polos e zeros, isto é, Slaney encontra a transformada de Laplace da equação 3.51, e cada raiz s do numerador ou do numerador é vinculada a uma raiz z equivalente no filtro projetado através da equação:

$$z = e^{sT_s} \tag{3.52}$$

A documentação de Slaney é bastante completa, incluindo a proposta de implementação, avaliações utilizando o *Mathematica* e códigos para a implementação do filtro nesse mesmo *software* e no MatLab. Contudo, desejava-se implementar filtros que pudessem ter o valor de ϕ variável, e a implementação de Slaney utilizou $\phi = 0$, incluindo apenas um exemplo alternativo com a função seno ao invés de cosseno, avaliando a posição de polos e zeros na transformada de Laplace mas sem utilizar essa informação para a implementação de filtros digitais. Esse filtro está implementado na AudioLazy, como uma das estratégias do dicionário de estratégias *gammatone*.

Em alternativa à proposta de Slaney, o filtro gammatone foi implementado na AudioLazy utilizando a invariância ao impulso, isto é, a equação 3.51 foi amostrada com $t = n\ T\_s$, de forma a obter a sequência de resposta ao impulso:

$$g_\eta[n] = aT_s{}^{\eta-1} n^{\eta-1} e^{-2pib_w T_s n} \cos(2\pi f_c T_s n + \phi) u[n] \tag{3.53}$$

$$g_\eta[n] = C n^{\eta-1} e^{Bn} \cos(\omega n + \phi) u[n] \tag{3.54}$$

Em que as substituições fazem de ω a frequência central em radianos por amostra, B a largura de banda em radianos por amostra e n o índice da amostra na sequência; C é apenas uma constante, e o produto por $u[n]$ visa apenas explicitar que o filtro é causal. Dessa forma, o equacionamento da resposta ao impulso no

tempo discreto através da amostragem da função contínua mantém o formato geral de sua equação original como um produto de uma potência da variável de tempo, uma exponencial e uma senoide. Primeiramente, utilizando a fórmula de Euler, sabe-se que:

$$cos(\Phi) = \frac{e^{j\Phi} + e^{-j\Phi}}{2} \tag{3.55}$$

De forma que o filtro gammatone discretizado no tempo definido na equação 3.54 pode ser descrito como:

$$g_\eta[n] = \frac{C}{2} n^{\eta-1} e^{Bn} \left(e^{j(\omega n+\phi)} + e^{-j(\omega n+\phi)}\right) u[n] \tag{3.56}$$

Sabe-se, através da definição da transformada Z (eq. 3.24), que

$$\mathcal{Z}(e^{\alpha n}u[n]) = \sum_{n=-\infty}^{\infty} e^{\alpha n}u[n]z^{-n} \tag{3.57}$$

$$= \sum_{n=0}^{\infty} \left(e^{\alpha} z^{-1}\right)^n \tag{3.58}$$

$$= \frac{1}{1 - e^{\alpha} z^{-1}} \tag{3.59}$$

$$\tag{3.60}$$

pois a equação 3.58 é a soma de uma PG (Progressão Geométrica), que somente converge para o resultado fornecido quando $e^{\alpha} z^{-1} < 1$. Esse resultado permite a obtenção da transformada Z do filtro gammatone para $\eta = 1$. Utilizando apenas a linearidade do operador transformada Z, obtemos:

$$G_1(z) = \mathcal{Z}(g_1[n]) \tag{3.61}$$

$$= \frac{C}{2} \mathcal{Z}\left[e^{Bn}\left(e^{j(\omega n+\phi)} + e^{-j(\omega n+\phi)}\right) u[n]\right] \tag{3.62}$$

$$= \frac{C}{2} \left[\mathcal{Z}\left(e^{Bn+j(\omega n+\phi)} u[n]\right) + \mathcal{Z}\left(e^{Bn-j(\omega n+\phi)} u[n]\right)\right] \tag{3.63}$$

$$= \frac{C}{2} \left[e^{j\phi}\mathcal{Z}\left(e^{(B+j\omega)n} u[n]\right) + e^{-j\phi}\mathcal{Z}\left(e^{(B-j\omega)n} u[n]\right)\right] \tag{3.64}$$

Podemos agora aplicar a equação 3.60 e a fórmula de Euler de forma a obter o filtro correspondente:

$$G_1(z) = \frac{C}{2} \left[\frac{e^{j\phi}}{1 - e^{B+j\omega} z^{-1}} + \frac{e^{-j\phi}}{1 - e^{B-j\omega} z^{-1}}\right] \tag{3.65}$$

$$= \frac{C}{2} \left[\frac{e^{j\phi}\left(1 - e^{B-j\omega} z^{-1}\right) + e^{-j\phi}\left(1 - e^{B+j\omega} z^{-1}\right)}{\left(1 - e^{B+j\omega} z^{-1}\right)\left(1 - e^{B-j\omega} z^{-1}\right)}\right] \tag{3.66}$$

$$= \frac{C}{2} \left[\frac{e^{j\phi} + e^{-j\phi} - \left(e^{j\phi} e^{B-j\omega} + e^{-j\phi} e^{B+j\omega}\right) z^{-1}}{1 - e^{B}\left(e^{j\omega} + e^{-j\omega}\right) z^{-1} + e^{B+\cancel{j\omega}} e^{B-\cancel{j\omega}} z^{-2}}\right] \tag{3.67}$$

$$= \frac{C}{2} \left[\frac{2\cos(\phi) - \left(e^{-j(\omega-\phi)} + e^{j(\omega-\phi)}\right) e^{B} z^{-1}}{1 - 2e^{B}\cos(\omega) z^{-1} + e^{2B} z^{-2}}\right] \tag{3.68}$$

$$= \frac{C}{2} \left[\frac{2\cos(\phi) - 2\cos(\omega - \phi) e^{B} z^{-1}}{1 - 2e^{B}\cos(\omega) z^{-1} + e^{2B} z^{-2}}\right] \tag{3.69}$$

Adotando $R = e^B$ como raio (magnitude) dos polos do filtro, chegamos ao resultado:

$$G_1(z) = C\frac{\cos(\phi) - \cos(\omega - \phi)Rz^{-1}}{1 - 2R\cos(\omega)z^{-1} + R^2z^{-2}} \tag{3.70}$$

que é um filtro ressonador.

Para a implementação dos filtros gammatone seguindo a definição dada, é suficiente conhecer o equacionamento dos filtros, sendo que maiores detalhes sobre ressonadores encontram-se na seção 3.2.8. Deseja-se obter filtros para outros valores de η, e para uma implementação geral desse filtro, faz-se necessária a utilização de uma propriedade importante da transformada Z. Pode-se derivar com relação a z ambos os lados da igualdade que define a transformada (eq. 3.24), resultando em:

$$\frac{d}{dz}X(z) = \frac{d}{dz}\sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{3.71}$$

$$= \sum_{n=-\infty}^{\infty} x[n]\frac{d}{dz}z^{-n} \tag{3.72}$$

$$= \sum_{n=-\infty}^{\infty} -nx[n]z^{-n-1} \tag{3.73}$$

$$= \left[\sum_{n=-\infty}^{\infty} -nx[n]z^{-n}\right]z^{-1} \tag{3.74}$$

$$= \mathcal{Z}(-nx[n])z^{-1} \tag{3.75}$$

$$\tag{3.76}$$

caso as somatórias de definição da transformada Z convirjam. Escrevendo de outra forma a mesma equação, tem-se a propriedade:

$$\mathcal{Z}(nx[n]) = -z\frac{d}{dz}X(z) \tag{3.77}$$

A partir da definição do filtro gammatone discreto obtida através da amostragem da resposta ao impulso do filtro gammatone contínuo (eq. 3.54), pode-se notar uma relação entre filtros com diferentes valores de η:

$$g_{\eta+1}[n] = Cn^{\eta}e^{Bn}\cos(\omega n + \phi)u[n] \tag{3.78}$$

$$= nCn^{\eta-1}e^{Bn}\cos(\omega n + \phi)u[n] \tag{3.79}$$

$$= ng_{\eta}[n] \tag{3.80}$$

que, aplicando a transformada Z em ambos os lados da igualdade e em conjunto com a propriedade definida na equação 3.77, nos fornece:

$$G_{\eta+1}(z) = -z\frac{d}{dz}G_{\eta}(z) \tag{3.81}$$

Em conjunto com o caso base com $\eta = 1$ definido na equação 3.70, a equação 3.81 fornece um algoritmo recursivo para a obtenção de filtros gammatone discretos para quaisquer valores de $\eta \in \mathbb{N}^*$, além de permitir um controle sobre a fase do filtro projetado. O valor da constante C pode ser admitido inicialmente igual à unidade, e posteriormente corrigido para que a resposta em frequência na frequência central seja igual à unidade.

Essa é, de fato, a implementação da estratégia *gammatone.sampled* existente no pacote AudioLazy, a menos do fato de que o resultado pode estar contaminado por erros numéricos, como visto na seção 3.2.5. Para contornar esse problema, o cálculo com a derivação é realizado a fim de obter o numerador do filtro

gammatone, e o denominador é descartado pelo fato deste ser conhecido, isto é, o denominador de $G_\eta(z)$ é igual ao de $G_1(z)$ elevado à η-ésima potência[11]. O código-fonte da AudioLazy relativo à implementação desse filtro encontra-se abaixo:

Código-fonte 3.11: *Estratégia gammatone.sampled: filtro gammatone amostrado paramétrico*

```
def gammatone(freq, bandwidth, phase=0, eta=4):
  assert eta >= 1

  A = exp(-bandwidth) # Raio A = R
  numerator = cos(phase) - A * cos(freq - phase) * z ** -1
  denominator = 1 - 2 * A * cos(freq) * z ** -1 + A ** 2 * z ** -2

  # A derivada multiplica por mul_after depois de cada passo de derivação
  filt = (numerator / denominator).diff(n=eta-1, mul_after=-z)

  # Tudo pronto, exceto o ganho. O denominador pode ter erros numéricos!
  f0 = ZFilter(filt.numpoly) / denominator # Descarta o denominador de filt
  f0 /= abs(f0.freq_response(freq)) # Ganho máximo == 1.0 (0 dB)
  fn = 1 / denominator
  fn /= abs(fn.freq_response(freq)) # Ganho máximo == 1.0 (0 dB)
  return CascadeFilter([f0] + [fn] * (eta - 1)) # Implementa em cascata
```

Porém, para esse fim, foi necessário um procedimento que realizasse a derivação sem replicar desnecessariamente o denominador. A continuação traz maiores detalhes sobre esse procedimento. O filtro gammatone pode ser utilizado conforme expresso no código fonte 3.10, bastando utilizar a estratégia *gammatone.sampled* ao invés da estratégia padrão. A vantagem em se usar essa estratégia consiste no fato de que tanto a fase como a ordem do filtro são parametrizados, diferentemente do que ocorre nas demais estratégias.

3.2.7 A derivada em Z

Seja $H(z) = X(z)/Y(z)$ a equação de sistema de um filtro digital linear e $D_M[H(z)] = M(z)\frac{d}{dz}[H(z)]$ a derivada do filtro multiplicada por um filtro FIR $M(z)$ sem retroalimentação (i.e., o denominador de $M(z)$ é igual a 1). A escrita $D_M^{(\beta)}[H(z)]$ denota a recursão realizada β vezes, isto é,

$$D_M^{(\beta)}[H(z)] = \begin{cases} D_M\left[D_M^{(\beta-1)}[H(z)]\right], & \beta \in \mathbb{N} \setminus \{0,1\} \\ M(z)\dfrac{d}{dz}[H(z)], & \beta = 1 \\ H(z), & \beta = 0 \end{cases} \tag{3.82}$$

Deseja-se obter o valor de $D_M^{(\beta)}[H(z)]$ para um valor arbitrário $\beta \in \mathbb{N}$, sem criar polos e zeros que obviamente se cancelem. Sabe-se que, aplicando a derivação com multiplicação para $\beta = 1$:

$$D_M^{(1)}[H(z)] = M(z)\frac{d}{dz}H(z) \tag{3.83}$$

$$= M(z)\frac{\left[Y(z)\dfrac{d}{dz}[X(z)]\right] - \left[X(z)\dfrac{d}{dz}[Y(z)]\right]}{Y^2(z)} \tag{3.84}$$

$$= \frac{N_1(z)}{Y^2(z)} \tag{3.85}$$

Em que $N_\beta(z)$ representa o numerador de $D_M^{(\beta)}[H(z)]$. Continuando, quando $\beta = 2$:

[11] Essa é uma consequência da derivação sem cancelamento de polos e zeros, tópico da seção que segue.

$$D_M^{(2)}[H(z)] = D_M\left[D_M^{(1)}[H(z)]\right] \tag{3.86}$$
$$= D_M\left[\frac{N_1(z)}{Y^2(z)}\right] \tag{3.87}$$
$$= M(z)\frac{\left[Y^2(z)\frac{d}{dz}[N_1(z)]\right] - \left[N_1(z)\frac{d}{dz}[Y^2(z)]\right]}{[Y^2(z)]^2} \tag{3.88}$$
$$= M(z)\frac{\left[Y(z)\cancel{Y(z)}\frac{d}{dz}[N_1(z)]\right] - \left[2N_1(z)\cancel{Y(z)}\frac{d}{dz}[Y(z)]\right]}{Y^3(z)\cancel{Y(z)}} \tag{3.89}$$
$$= M(z)\frac{\left[Y(z)\frac{d}{dz}[N_1(z)]\right] - \left[2N_1(z)\frac{d}{dz}[Y(z)]\right]}{Y^3(z)} \tag{3.90}$$
$$= \frac{N_2(z)}{Y^3(z)} \tag{3.91}$$

Pode-se observar que uma implementação recursiva ingênua da derivação com multiplicação (e também da derivação, caso em que $M(z) = 1$) em que os polos e zeros não são avaliados, propagaríamos um crescimento exponencial da ordem do denominador (e consequentemente do numerador) desnecessariamente. Em outras palavras, o cancelamento de polos e zeros que ocorre na equação 3.89 é importante para conter o crescimento do denominador.

Como hipótese de indução, suponha que seja possível representar o denominador com um crescimento linear do expoente, isto é,

$$D_M^{(\beta)}[H(z)] = \frac{N_\beta(z)}{Y^{\beta+1}(z)} \tag{3.92}$$

em que o numerador $N_\beta(z)$ é dado por:

$$N_\beta(z) = M(z)\left[Y(z)\frac{d}{dz}[N_{\beta-1}(z)]\right] - \left[\beta N_{\beta-1}(z)\frac{d}{dz}[Y(z)]\right] \tag{3.93}$$

Para a aplicação $\beta + 1$, temos:

$$D_M^{(\beta+1)}[H(z)] = D_M\left[D_M^{(\beta)}[H(z)]\right] \tag{3.94}$$
$$= D_M\left[\frac{N_\beta(z)}{Y^{\beta+1}(z)}\right] \tag{3.95}$$
$$= M(z)\frac{\left[Y^{\beta+1}(z)\frac{d}{dz}[N_\beta(z)]\right] - \left[N_\beta(z)\frac{d}{dz}[Y^{\beta+1}(z)]\right]}{[Y^{\beta+1}(z)]^2} \tag{3.96}$$
$$= M(z)\frac{\left[Y^{\cancel{\beta}+1}(z)\frac{d}{dz}[N_\beta(z)]\right] - \left[(\beta+1)N_\beta(z)\cancel{Y^{\beta}(z)}\frac{d}{dz}[Y(z)]\right]}{Y^{\cancel{2}\beta+2}(z)} \tag{3.97}$$
$$= M(z)\frac{\left[Y(z)\frac{d}{dz}[N_\beta(z)]\right] - \left[(\beta+1)N_\beta(z)\frac{d}{dz}[Y(z)]\right]}{Y^{\beta+2}(z)} \tag{3.98}$$
$$= \frac{N_{\beta+1}(z)}{Y^{\beta+2}(z)} \tag{3.99}$$

Em conjunto com o caso base (eq. 3.85 ou 3.91), a equação 3.99 demonstra por indução que a aplicação da derivada em z nos filtros digitais mantém, no pior caso, crescimento linear da ordem de seus denominadores, mesmo havendo uma multiplicação envolvendo o numerador entre cada passo de derivação. Adicionalmente, sabe-se a partir do mesmo procedimento de indução (mas com caso base dado pela eq. 3.84 ou 3.90) que a cada passo o numerador do filtro resultante é:

$$N_{\beta+1}(z) = M(z)\left[Y(z)\frac{d}{dz}[N_\beta(z)]\right] - \left[(\beta+1)N_\beta(z)\frac{d}{dz}[Y(z)]\right] \tag{3.100}$$

em que todos os filtros envolvidos nesse último equacionamento possuem denominador constante igual a 1, de forma que a derivação que persiste se restringe à derivação em somas de potências (normalmente polinômios de Laurent), casos de implementação trivial.

O método *diff* da classe *ZFilter* na AudioLazy implementa a recursão dada pela equação 3.100, e devolve o filtro $D_M^{(\beta)}[H(z)]$, desde que fornecidos $H(z)$, β (padrão igual a 1) e $M(z)$ (padrão igual a 1). Esse procedimento permite assegurar que filtros gammatone vistos na seção 3.2.6 sejam implementados diretamente através dos parâmetros sem a necessidade de preocupação de avaliação de polos e zeros que se cancelam. Por conta da propriedade da transformada Z dada pela equação 3.77, é de se esperar que o uso de $M(z) = -z$ seja comum.

3.2.8 Filtros digitais implementados

Esta seção lista alguns filtros comuns, ou de alguma forma padronizados que fazem parte da AudioLazy. Os gráficos foram gerados utilizando os já discutidos métodos *plot* e *zplot*. A maioria dos filtros aqui definidos funcionam como variantes no tempo, Os casos particulares que não são variantes no tempo são explicados em separado, indicando o motivo da exceção. Steiglitz fornece maiores detalhes sobre os filtros *comb* e ressonadores [Ste96, capítulos 4, 5 e 6] discutidos nesta seção.

Comb

Filtros *comb* possuem esse nome por conta do formato de sua resposta em frequência, cujo gráfico é similar a um pente (*comb*). Há três versões do filtro implementadas, organizadas como estratégias no dicionário de estratégias homônimo com o filtro. Quatro gráficos foram criados para ilustrar o comportamento do filtro, e encontram-se nas figuras 3.6, 3.7, 3.8 e 3.9, com a especificação dos filtros no código-fonte 3.12.

- Estratégia *comb.fb*: é a estratégia padrão, com um filtro IIR $1/(1 - \alpha z^{-D})$. O nome "fb" vem de *feedback*;
- Estratégia *comb.tau*: idêntica à anterior, a menos do parâmetro de tempo τ dado em amostras que é utilizado como entrada em detrimento do ganho α, o qual é calculado como $\alpha = e^{-D/\tau}$. O valor de τ é definido como a duração para que o decaimento chegue a e^{-1}.
- Estratégia *comb.ff*: um filtro FIR $1 + \alpha z^{-D}$. O nome "ff" vem de *feedforward*.

Esse filtro foi inserido na AudioLazy como base necessária para possibilitar a implementação de um simples porém importante algoritmo de síntese, o algoritmo de Karplus-Strong.

Código-fonte 3.12: *Filtros comb e algoritmo de Karplus-Strong*

```
from audiolazy import comb, white_noise, zeros, pi

# Filtros dos gráficos:
fcomb_iir = comb.fb(25, alpha=.9) # Atraso de 25 amostras
fcomb_fir = comb.ff(25, alpha=.9)

# Algoritmo de Karplus-Strong:
def karplus_strong(freq, tau=2e4, memory=white_noise):
  return comb.tau(2 * pi / freq, tau).linearize()(zeros(), memory=memory)
```

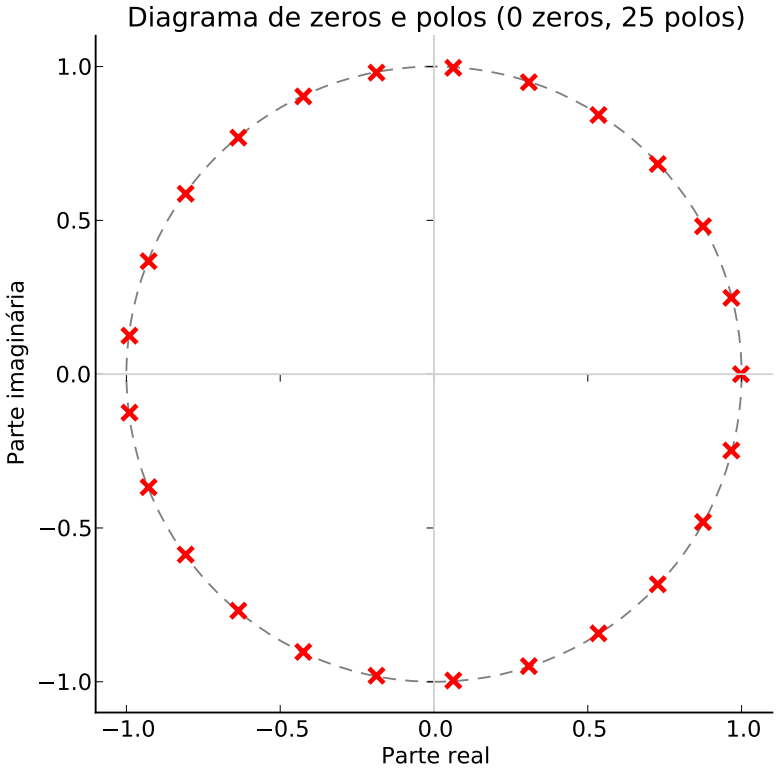


Figura 3.6: *Diagrama de zeros e polos de um filtro comb IIR*

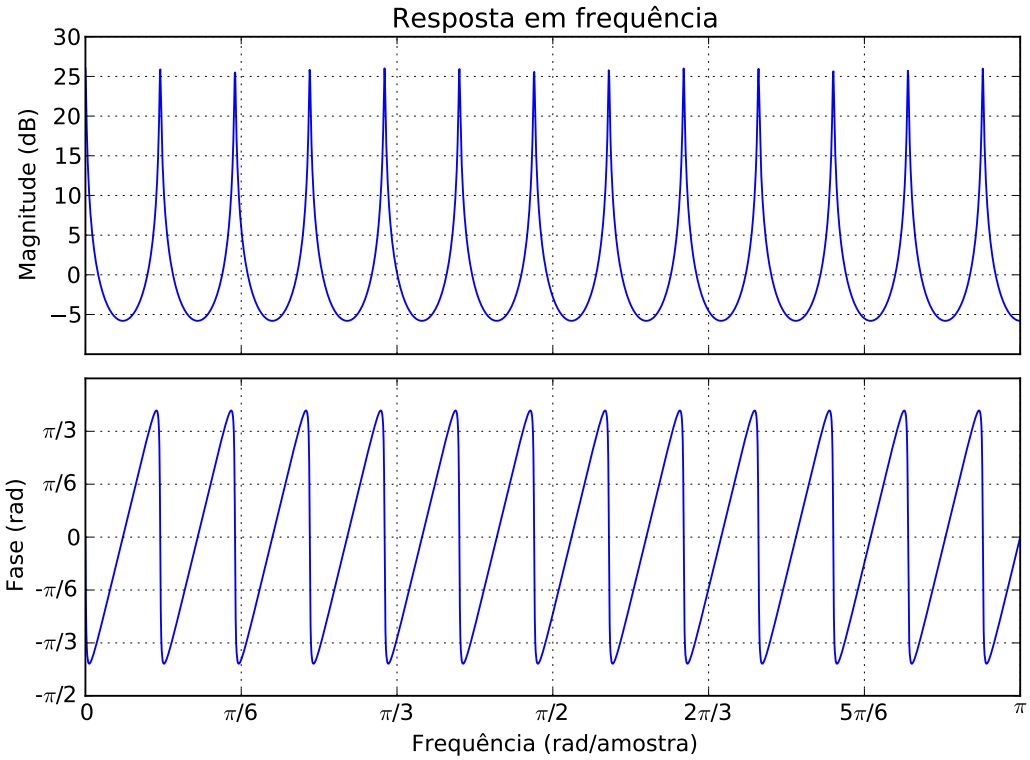


Figura 3.7: *Resposta em frequência de filtro comb IIR*

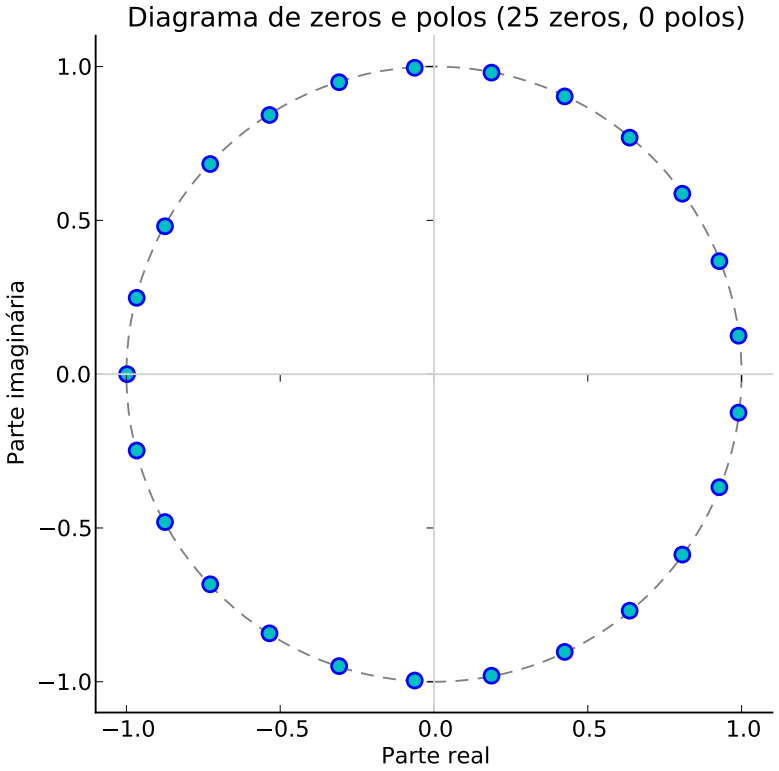


Figura 3.8: *Diagrama de zeros e polos de um filtro comb FIR*

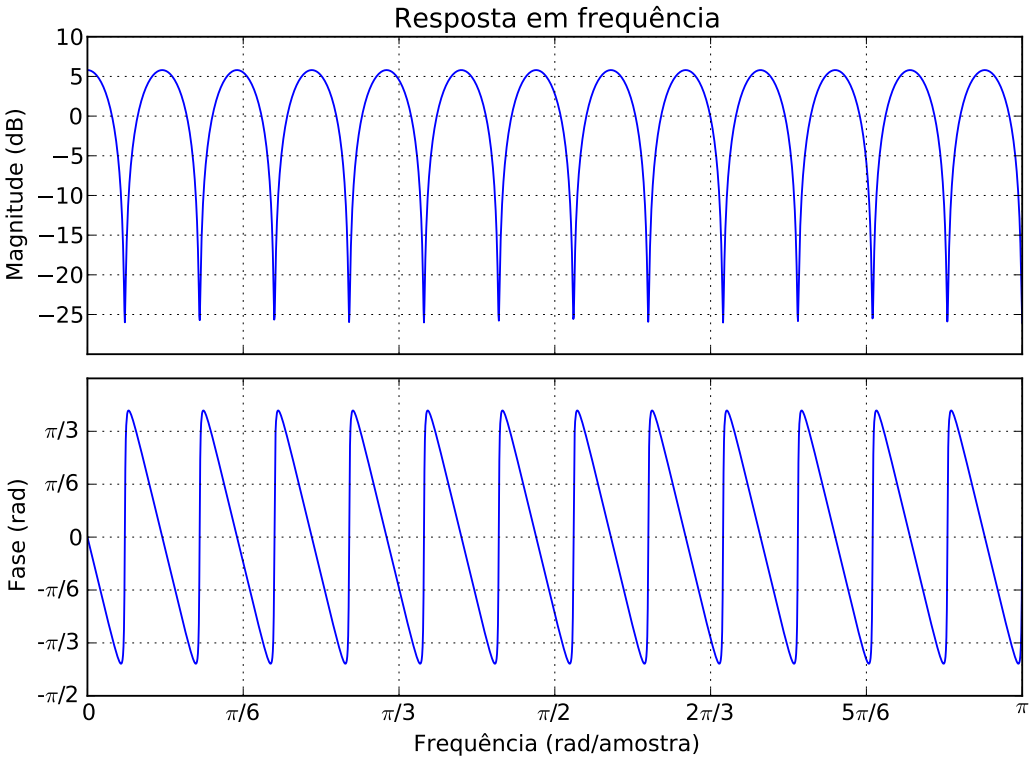


Figura 3.9: *Resposta em frequência de filtro comb FIR*

Passa-baixas e passa-altas de um polo

Frequência de corte é uma frequência a partir da qual a resposta em frequência de um filtro passa a atenuar pelo menos metade da potência do sinal. Como a potência é quadrática, tem-se que esse valor se refere-se a uma amplitude de $\sqrt{2}/2$, ou um ganho de aproximadamente $-3,0103$ dB. Embora sejam filtros simples, a implementação de tais filtros podem enganar alguém que acredita fielmente no casamento de polos e zeros com a transformada de Laplace, sobretudo com relação ao valor exato da frequência de corte.

Criando um filtro passa-baixas a partir do modelo de um capacitor e um resistor ideais em série (impedâncias $(sC)^{-1}$ e R, respectivamente), pode-se utilizar os componentes como um divisor de tensão e obter, de imediato, o filtro $1/(1+sRC)$, cujo único polo é claramente $s = -(RC)^{-1}$, e cuja frequência de corte é de $(RC)^{-1}$, em radianos por segundo. Utilizando esse resultado na equação 3.52, obtém-se o filtro $1/(1 - e^{-T_s/(RC)})z^{-1}$ por casamento de polos e zeros, que, com um ajuste de ganho para tornar seu ganho máximo unitário, pode ser representado como:

$$\frac{1-R}{1-Rz^{-1}} \tag{3.101}$$

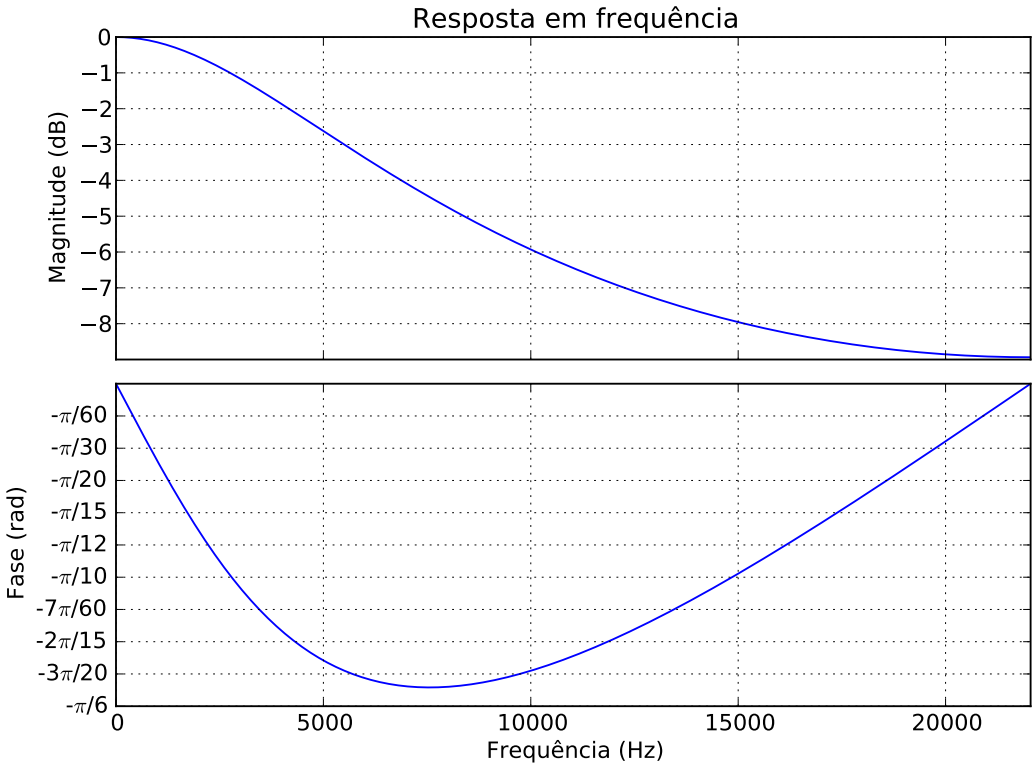


Figura 3.10: *Resposta em frequência do filtro passa-baixas exato a* 44100 *amostras por segundo com frequência de corte de* 11025 *Hz*

em que o raio R, seguindo o equacionamento fornecido, seria de $e^{-T_s/(RC)} = e^{-c}$, e a frequência de corte c é dada em radianos por amostra. Esse valor, entretanto, não corresponde à frequência de corte real do filtro projetado, sendo o valor exato igual a $v - \sqrt{v^2-1}$ com $v = 1 - \cos(\pi - c)$, valor este que pode ser encontrado igualando o módulo da equação de sistema do filtro ao valor $\sqrt{2}/2$. As duas estratégias foram inseridas na AudioLazy, em que a estratégia padrão utiliza o valor exato.

Avaliando numericamente com o SciPy[12] as duas estratégias de filtros passa-baixas implementadas, pode-se notar que o desvio pode ser significativo. Nesse experimento, os filtros foram projetados com o valor de

[12] Usando a função *scipy.optimize.brentq* para encontrar numericamente as raízes da função definida através da resposta em frequência do filtro (método *LinearFilter.freq_response*) adaptada para a raiz coincidir com a frequência de corte, isto é, a resposta em frequência é convertida para uma representação logarítmica (função *dB20* da AudioLazy, que obtém o valor $20log_10(x)$ para seu único argumento x) e transladada através da subtração de $10log_10\frac{1}{2}$, consequência da definição de frequência de corte. Essa

metade da frequência de Nyquist (i.e., $\pi/4$ radianos por amostra, ou 11025 Hz quando a taxa de amostragem é de 44100 amostras por segundo). A estratégia exata, implementada em *lowpass.pole* quando avaliada numericamente quanto ao valor da frequência de corte teve o valor de 1.5707963267945773 radianos por amostra, valor preciso para representar $\pi/4$ (ou 11025 Hz) em ponto flutuante. A estratégia *lowpass.pole_exp* implementa o filtro passa-baixas com o raio (polo) fornecido através do casamento de polos e zeros, e o valor de sua frequência de corte obtido numericamente é de 2.1050263234844335 radianos por amostra (14774.6177022 Hz), cerca de 34% maior que o esperado.

Também foi implementado o filtro passa-altas (o *highpass* da AudioLazy) definido através da equação:

$$\frac{1 - R}{1 + Rz^{-1}} \tag{3.102}$$

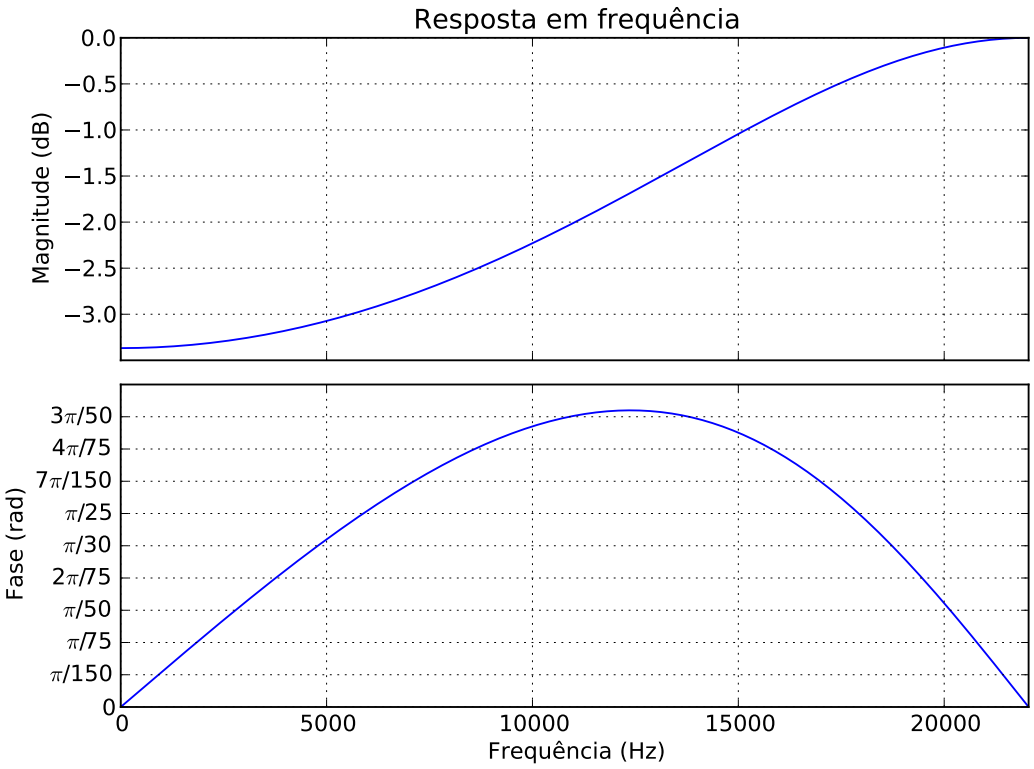


Figura 3.11: *Resposta em frequência do filtro passa-altas exato a* 44100 *amostras por segundo com frequência de corte de* 11025 *Hz*

Com o valor de R que permite a conversão para o valor exato de frequência de corte, que é precisamente o mesmo utilizado no filtro passa-baixas *lowpass.pole*. Caso seja desejado o passa-altas complementar a um passa-baixas (ou vice-versa), pode-se fazer um equacionamento do tipo $1 - \text{filtro}$ para obter o novo filtro, porém não se deve esquecer de que essa abordagem insere no filtro um zero, pertinente ao círculo unitário.

Ressonadores

Um filtro ressonador é um passa-faixas com um par de polos conjugados, que possui uma frequência central de "ressonância". A característica de filtros ressonadores está na presença de denominadores contendo um par

função do SciPy utiliza o algoritmo de R. P. Brent, e segundo a *docstring* da função, maiores informações podem ser encontradas em:
Brent, R. P. *Algorithms for Minimization Without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall, 1973. Ch. 3-4.

de complexos conjugados.

$$\frac{1}{1 - 2R\cos(\theta)z^{-1} + R^2 z^{-2}} \tag{3.103}$$

Há quatro estratégias implementadas, todas elas terminando o nome com "exp", como referência ao modo de uso do valor da largura de banda para obtenção do raio (magnitude) de cada um dos dois polos conjugados. Esse valor é a exponencial $e^{-B/2}$), com B medido em radianos por amostra, e refere-se a um procedimento de conversão do filtro descrito como uma função contínua em um filtro digital, analogamente ao realizado para a obtenção do filtro gammatone $G_1(z)$ (eq. 3.70). Um desses ressonadores pode ser visto no código fonte 2.4.

Nas 4 estratégias implementadas há dois tipos de filtros. Os filtros com "poles" no nome referem-se a filtros só-polos, isto é, filtros idênticos à equação 3.103, a menos do ganho $(1 - R^2)\sin(\theta)$ para permitir resposta em frequência máxima igual a 1, em módulo. Os outros dois filtros possuem "z" no nome, indicando que se referem a filtros com um par de zeros nos limites DC e frequência de Nyquist, ou seja, um numerador igual a $1 - z^{-2}$ (raízes 1 e -1), o qual possui ganho igual a $(1 - R^2)/2$.

Um dos aspectos que envolveu a escolha dos nomes foi a busca por uma maneira de descrever os diferentes tipos de valores de entrada possível pelo usuário desenvolvedor. Dessa forma, estratégias iniciadas com "*freq*" indicam que a frequência fornecida é a frequência θ utilizada no denominador do filtro, enquanto que as outras estratégias realizam uma correção para que a frequência fornecida seja a frequência de ganho máximo.

O cálculo da largura de banda é aproximado, assim como o que ocorre com a frequência de corte com os filtros passa-altas e passa-baixas, entretanto para valores pequenos de largura de banda (i.e., valores de $|R|$ próximos de 1) esse erro não se apresenta significativo. Um exemplo de caso no qual a influência é significativa consta na figura 3.12, referindo-se a um valor superior a 50% da frequência de Nyquist. Maiores detalhes sobre o cálculo da resposta em frequência em ressonadores podem ser encontrados em [OSB99, p. 258-268].

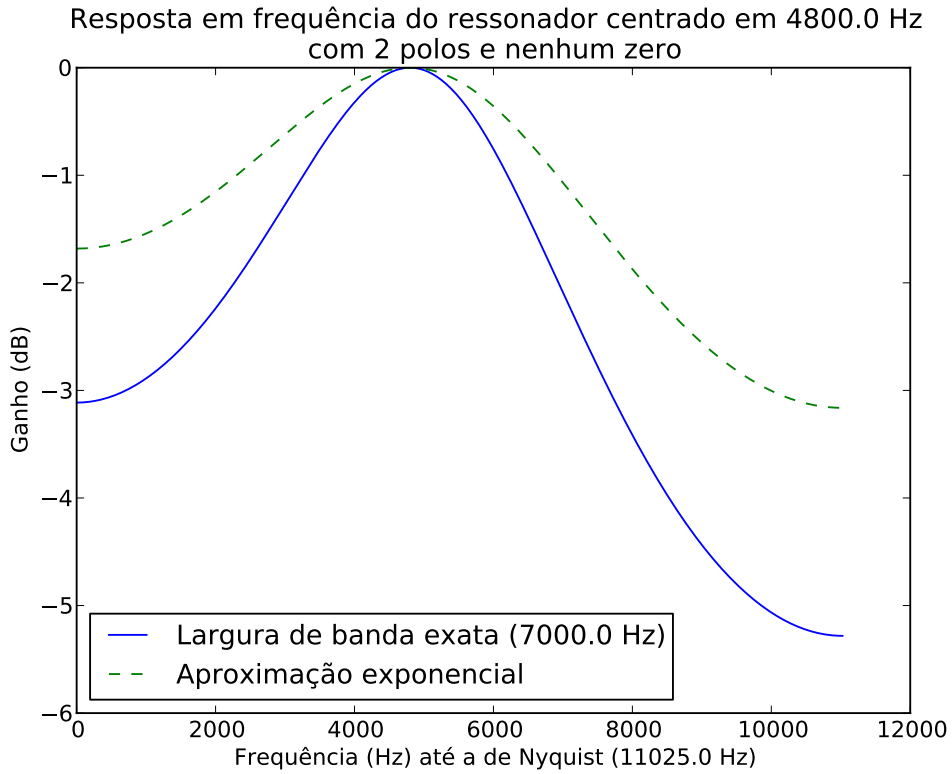


Figura 3.12: *Ressonadores com largura de banda exata e obtida através da exponencial*

A implementação do ressonador que possibilitou a figura 3.12 pode ser vista no código-fonte 3.13. O equacionamento foi encontrado utilizando o SymPy. Devido às restrições no domínio de aplicação dessa implementação, o ressonador de largura de banda exata não foi incluído no pacote AudioLazy.

Código-fonte 3.13: *Ressonador com valor exato de largura de banda*

```
def resonator(freq, bandwidth):
  if (bandwidth <= 0) or (bandwidth >= pi): # Faixa representável de valores
    raise ValueError("Invalid bandwidth")
  cosf = cos(freq)
  k = cos(bandwidth)
  if 2 * cosf ** 2 > 1 + k: # Assegura que o R será um número real
    raise ValueError("Invalid bandwidth")
  asq = cosf ** 2
  bsq = (1 - k) * (1 + k - 2 * asq) / (2 * (1 + k))
  p = bsq + asq
  q = bsq - asq
  sqdata = sqrt(p ** 2 + 2 * q + 1)
  R = sqrt(p + sqdata - sqrt(2) * sqrt(p**2 + p * sqdata + q))
  cost = cosf * (2 * R) / (1 + R ** 2)
  gain = (1 - R ** 2) * sqrt(1 - cost ** 2)
  denominator = 1 - 2 * R * cost * z ** -1 + R ** 2 * z ** -2
  filt = gain / denominator
  gain3db = dB20(.5) / 2
  if (dB20(filt.freq_response(0.)) > gain3db or
      dB20(filt.freq_response(pi)) > gain3db):
    raise ValueError("Invalid bandwidth")
  return filt
```

Gammatone

Muito já foi dito sobre os filtros gammatone nas seções 3.2.5 e 3.2.6, mas sua importância está associada à capacidade de oferecer um modelo simplificado da parte periférica do sistema auditivo humano. Slaney [Sla93] sintetizou o modelo de Patterson-Holdsworth de células ciliares da cóclea, modeladas através de filtros gammatone, indicando que um dos passos iniciais trata da obtenção do ERB (*Equivalent Rectangular Bandwidth*, ou largura de banda equivalente retangular) de um dado filtro gammatone, propondo modelos diferentes de ERB. Esses modelos apenas visam obter um valor de largura de banda para um filtro gammatone a partir de sua frequência central. Os modelos que foram implementados na AudioLazy foram os dois propostos por Glasberg e Moore [GM90, MG83], e encontram-se no dicionário de estratégias *erb*.

Para utilizar os valores de ERB junto ao gammatone, há uma constante para cada dada ordem η do gammatone, visando seu uso na equação 3.51. Uma tabela de tais constantes com o equacionamento que a gera foi fornecida por Holdsworth, Patterson e outros [HPNSR88]. Para uso prático, trata-se de uma constante multiplicativa com valor de $1,0185916357881302$ quanto $\eta = 4$. Assim como Slaney, Klapuri [Kla08] utiliza o valor arredondado $1,019$ na definição de seu filtro gammatone, seguindo a sugestão de Holdsworth. Há falhas na tabela de constantes, claramente justificáveis como erros de digitação e de arredondamento, mas isso não impediu a íntegra dessa tabela de ser utilizada como testes na AudioLazy para a função *gammatone_erb_constants*, responsável por construir a tabela para valores arbitrários de $\eta \in \mathbb{N}^*$, maneira como a constante recém citada foi obtida.

Três modelos de filtros gammatone foram implementados no dicionário de estratégias homônimo. O código-fonte 6 no apêndice de exemplos foi utilizado para a criação das figuras 3.13 e 3.14, trazendo a resposta em frequência e a resposta ao impulso dos filtros gammatone para diferentes valores de frequência central utilizando o modelo de ERB de Glasberg e Moore de 1990. Todos os filtros dos gráficos possuem $\eta = 4$.

O modelo de Klapuri consiste em definir quatro ressonadores em cascata, dois modelos com um par de zeros (*resonator.z_exp* na AudioLazy), e dois só-polos (*resonator.poles_exp*). A ideia de Klapuri foi aproveitar o fato de que os filtros gammatone aplicados como modelo auditivo são filtros com $\eta = 4$ pares de polos conjugados em torno de uma frequência central, e utilizou os ressonadores para criar a mesma situação, porém definindo o valor de θ dos denominadores de forma a fazer com que a frequência de máximo ganho de cada ressonador seja a frequência fornecida como central. Graças a essa simplicidade, foi possível fazer

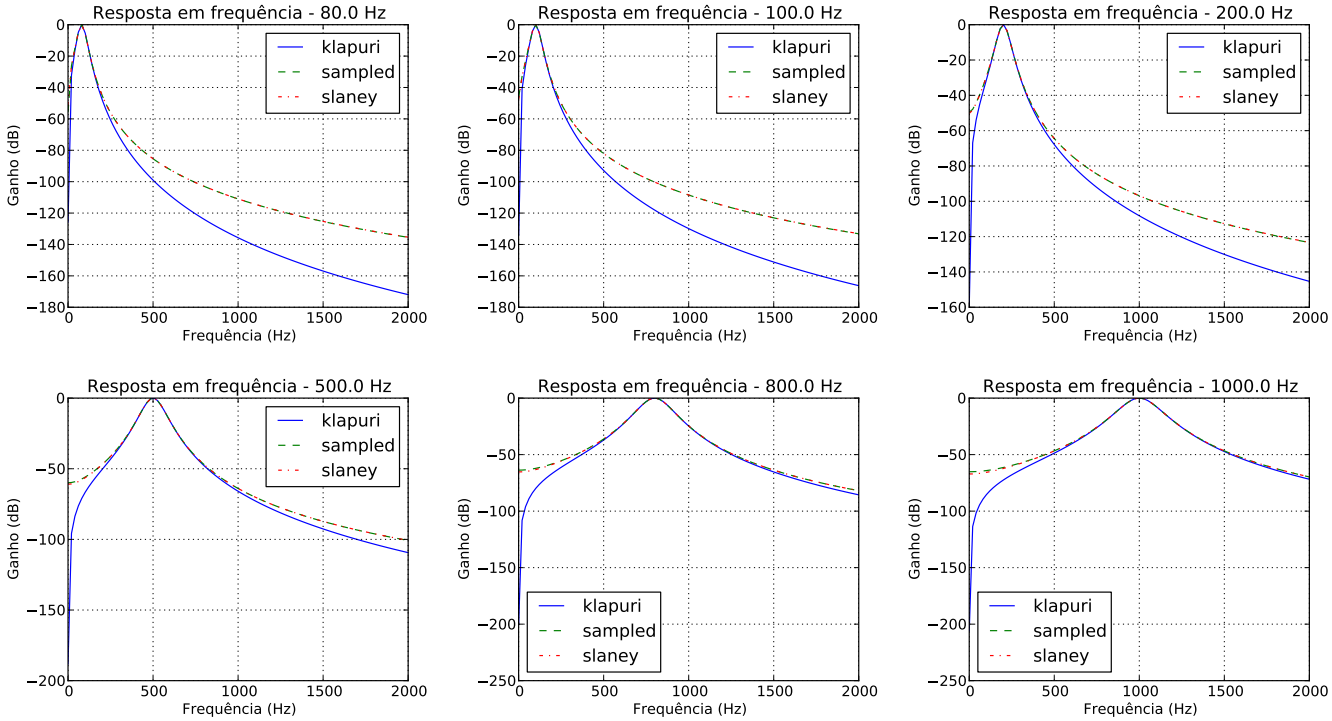


Figura 3.13: *Resposta em frequência dos filtros gammatone implementados*

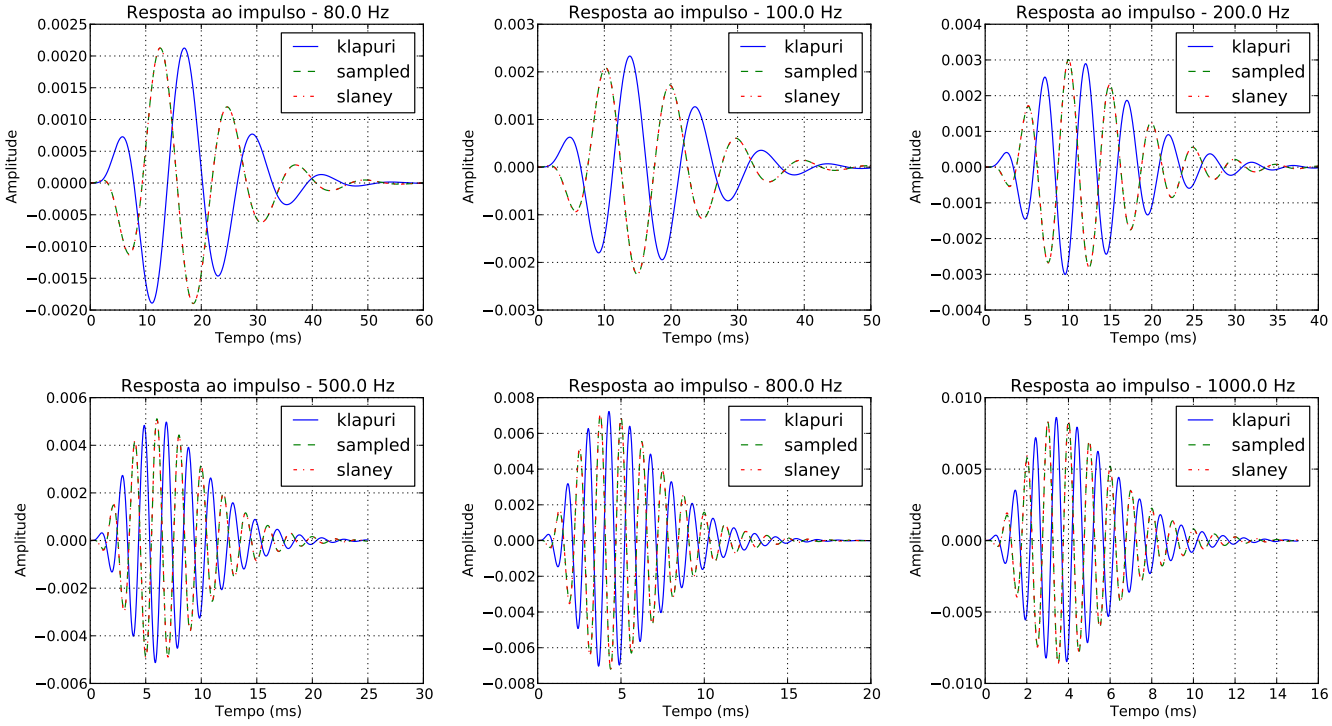


Figura 3.14: *Resposta ao impulso dos filtros gammatone implementados*

a implementação do filtro gammatone de Klapuri na AudioLazy ser variável no tempo, i.e., a frequência e a largura de banda desse filtro não precisam ser constantes. As duas demais estratégias são fixas com relação aos valores de frequência central e largura de banda.

Os zeros 1 e -1 que aparecem duas vezes no filtro de Klapuri criam uma diferença notável na resposta

em frequência dos filtros com relação a frequências muito baixas e muito altas. Como o objetivo é o de modelar o aparato auditivo humano, não há grandes problemas em considerar essa diferença no ultrassom e no infrassom, por não serem audíveis. Um aspecto notável é a defasagem no filtro do Klapuri, que difere completamente dos demais.

Em uma comparação entre o filtro de Slaney e o filtro proposto na seção 3.2.6 e implementado como *gammatone.sampled*, as diferenças encontradas são pequenas, visualizáveis no gráfico em valores extremos de frequências. Muito provavelmente são diferenças devido ao arredondamento, pois há diferenças na ordem das operações realizadas, e os filtros propostos mantêm um único componente ressonador com todos os zeros do filtro, algo que não ocorre com os filtros de Slaney da forma como foram implementados[13] de forma que pode-se dizer que o modelo proposto para o filtro gammatone generaliza o modelo de Slaney, parametrizando-o.

3.2.9 Envelope ou envoltória temporal

Uma alternativa ao uso de um modelo da parte periférica do sistema auditivo vista na seção 3.2.8 através de filtros gammatone (como, por exemplo, Klapuri [Kla04] sugere) é a utilização de um modelo de volume sonoro percebido como algo dependente da frequência do som que está sendo ouvido. Cada curva de Fletcher-Munson [Moo90, p. 21] refere-se ao volume sonoro constante para uma senoide com frequência variável no tempo, passando pelos diferentes valores na curva. Tal modelo não foi implementado na AudioLazy, na qual preferiu-se privilegiar o modelo de filtros gammatone. Entretanto, pode-se observar que a intensidade dinâmica é um modelo importante para MIR, e as curvas citadas, se implementadas, seriam utilizadas em procedimentos de normalização de dados de áudio.

Klapuri [Kla97] indica ainda outra alternativa para a representação da intensidade dinâmica, dessa vez voltada à aplicação prática de obtenção de instantes iniciais de eventos no áudio, na qual o som é decomposto em poucas faixas de frequências por filtros passa-faixas, e a envoltória dinâmica de cada uma dessas faixas é considerada em separado. Variações abruptas em cada um desses sinais são considerados instantes iniciais de eventos.

A envoltória dinâmica ou temporal é a representação no tempo da intensidade de um dado sinal. Há três modelos de envoltória dinâmica implementados na AudioLazy no dicionário de estratégias *lazy_analysis.envelope*. Todas tratam basicamente de tornar o sinal estritamente positivo para então aplicar um filtro passa-baixas, variando apenas a maneira como isso é feito. A estratégia padrão *envelope.rms*, por exemplo, obtém o quadrado da sinal, aplica o filtro passa-baixas e extrai a raiz quadrada de cada amostra. Esse é um modelo muito simples de intensidade dinâmica. A média móvel, como foi vista nas seções 3.1.1 e 3.1.2, também poderia ser utilizada como filtro passa-baixas, e outras alternativas incluem a criação de um filtro mediana móvel. Sobre filtros estatísticos de normalização, vale dizer que Klapuri [Kla08] utiliza um filtro baseado em uma potência do desvio padrão de um bloco a fim de normalizar seu sinal de entrada, e o efeito do filtro consiste justamente em alterar a intensidade dinâmica do bloco como um todo. Envoltórias dinâmicas também são utilizadas em síntese de áudio, a exemplo do que ocorre com a ADSR (*Attack, Decay, Sustain, Release*), utilizada em síntese para moldar a intensidade de um dado sinal no tempo.

Transcrição por envoltória reversa

Dada a simplicidade e potencialidade do uso de envoltórias, foi criado um exemplo utilizando a estratégia RMS (*Root Mean Square*, ou raiz da média quadrática) do dicionário de estratégias *envelope*. Visando complementar o que Klapuri objetivou, o exemplo aqui realizado visou encontrar os instantes de início de eventos caracterizados reversamente no tempo, isto é, um som que inicia com um aumento de intensidade

[13] Vale dizer que Slaney propôs a implementação em cascata, porém seus exemplos em [Sla93] utilizam um único numerador e denominador. Todas as estratégias de filtros gammatones implementados na AudioLazy devolvem filtros em cascata. Durante a pesquisa, descobriu-se uma implementação publicamente disponível do modelo do gammatone de Slaney escrita em Python, com o nome "ThH" como autor:
http://work.thaslwanter.at/CSS/Code/GammaTones.py, último acesso 2012-08-29.
O código é uma tradução do modelo de Slaney utilizando a função *lfilter* do SciPy, como um único numerador e denominador (i.e., sem implementar o filtro em cascata).

lento e gradual, finalizando abruptamente. Esse exemplo encontra-se na figura 3.15 como possibilidade de uso da AudioLazy junto ao NumPy para a transcrição de instantes de início de eventos conhecidos. Foi sintetizado utilizando o algoritmo de Karplus-Strong a nota *A4* repetidas vezes. Espera-se que as notas tenham sido diferentes no sentido de que o algoritmo de síntese utilizado é inicializado com valores aleatórios. Esse não foi o sinal utilizado para o processo mas sim a versão retrogradada no tempo desse mesmo sinal. Sobre esse som, foi obtida a envoltória RMS, subamostrada de maneira a conter apenas a média de 1024 amostras vizinhas, blocos esses espaçados de 50 amostras, cujo resultado é apresentado no primeiro gráfico da figura 3.15. Inspirado no procedimento presente no primeiro apêndice deste trabalho, uma síntese envolvendo uma única nota retrogradada foi realizada, e usada como modelo empírico da ferramenta de síntese. Apenas com essa informação, um simples cálculo da correlação cruzada entre os sinais fornece um terceiro sinal que, se adequadamente alinhado, possui como valores de pico os instantes de início dos eventos, mesmo sendo estes completamente sem um ataque definido.

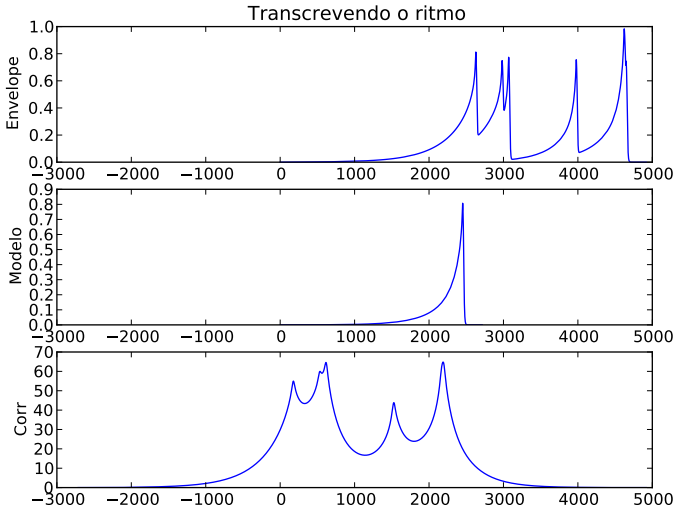


Figura 3.15: *Transcrição de ritmo por envoltória (reversa). Um sinal foi criado à taxa de* 44100 *amostras por segundo utilizando o modelo de síntese de Karplus-Strong com uma nota de* 440 *Hz recriada nos instantes* $[0.01, 0.04, 0.77, 1.8, 1.9, 2.3]$ *em segundos, com intensidades* $[0.3, 0.5, 0.8, 0.9, 0.6, 0.7]$. *Seu sinal foi retrogradado no tempo, e sua envoltória RMS amplificada em* 2.5*, encontra-se como primeiro dos gráficos. O segundo gráfico contém apenas um ataque com o modelo de Karplus-Strong retrogradado. Os valores desses dois gráficos são a média de blocos de* 1024 *amostras, espaçados de* 50 *amostras entre si. O último gráfico contém a convolução dos sinais, expondo em seus valores de pico os instantes de início dos eventos de envoltória reversa. O único ataque efetivamente perdido foi o último, devido à proximidade no tempo (*30 *milissegundos).*

Sinal analítico e a transformada discreta de Hilbert

A seção 3.1 indicou que os filtros LTI que são aplicáveis, a menos de adaptações e inserções de atrasos, se restringem aos filtros causais, restrição na qual nenhum filtro deve conter atrasos negativos. Nesta seção, há uma tentativa de se obter algo parecido, mas dessa vez no domínio da frequência, e referente a valores nulos em frequências negativas.

Quando os sinais utilizados são reais (i.e., não há amostras complexas), essa informação torna parte da resposta da DTFT (eq. 3.48) redundante, por esta possuir a propriedade [OSB99, p.56]:

$$\mathcal{F}[x[n]] = X(e^{j\omega}) = X^*(e^{-j\omega}) \tag{3.104}$$

quando x[n] é real, em que o "*" denota a operação de conjugação do número complexo resultante. Isso significa que, no domínio da frequência, existe uma redundância, e que a informação no domínio $\omega \in [0, pi]$ é suficiente para reconstruir a informação para todo $\omega \in \mathbb{R}$, quando incluída a propriedade de periodicidade de período 2π radianos por amostras existente na DTFT.

Ao sinal que possui resposta em frequência nula para valores negativos de frequência, considerando o domínio $\omega \in (-\pi; \pi]$, é dado o nome de sinal analítico[14]. Um sinal analítico $x_a[n]$ discretizado no tempo obtido a partir de um sinal real $x_r[n]$ discretizado no tempo em um suporte compacto possui duas propriedades [Mar99]:

- A parte real do sinal analítico coincide com o sinal real $x_r[n]$;
- A parte real e a parte imaginária do sinal analítico são ortogonais[15].

Dessa forma, a intenção é obter a parte imaginária do sinal analítico, à qual é dado o nome de transformada discreta de Hilbert. Há diversas formas de se obter o valor ou uma aproximação numérica do sinal analítico $x_a[n]$ a partir de um sinal real $x_r[n]$ discretizado no tempo [OSB99, capítulo 11], mas uma que merece destaque é a utilização de coeficientes DFT (vide seção 3.3.3) conforme proposto por Marple [Mar99]:

$$x_a[n] = \text{iDFT}\left[h[k] \cdot \text{DFT}(x_r[n])\right] \tag{3.105}$$

em que os coeficientes $h[k]$ visam apenas tornar nulos os componentes frequenciais negativos da DFT, de forma a obter:

$$h[k] = \begin{cases} 2, & 0 < k < \frac{N}{2} \\ 0, & k > \frac{N}{2} \\ 1, & c.c. \end{cases} \tag{3.106}$$

em que N é o número de amostras usadas na DFT, de forma que $N/2$ é a frequência de Nyquist dada em número de ciclos. Os valores são dobrados para manter a energia original do sinal. A remoção dos valores acima da taxa de Nyquist equivale à remoção dos valores negativos de frequência, devido a periodicidade. Os valores iguais à 1 na fronteira são escolhidos de forma a garantir as duas propriedades dos sinais analíticos[16].

Essa formulação existe pronta no SciPy através da função *scipy.signal.hilbert* que, apesar do nome, devolve um bloco de sinal analítico dado um bloco de sinal real. O nome se justifica a partir do fato de que a transformada de Hilbert de fato faz parte do valor retornado. Essa funcionalidade também pode ser facilmente implementada na AudioLazy:

Código-fonte 3.14: *Obtenção de um bloco analítico a partir de um bloco real*

```
def hilbert(blk, size=None):
  if size == None:
    size = len(blk)
  if size <= 2:
    raise ValueError("Too small")

  hker = Stream((idx <= size*.5) + (idx > 0) * (idx < size * .5)
                for idx in range(size))
  return ifft(fft(blk, size) * hker, size)
```

em que as funções que calculam a FFT e iFFT são melhor detalhadas na seção 3.3.3.

Envoltória a partir do sinal analítico

Como é dito por Marple [Mar99], um dos usos práticos do sinal analítico inclui a obtenção da envoltória de sinais. Segundo Oppenheim [OSB99, p. 796-799], o sinal analitico pode ser descrito em uma representação polar:

$$x_a[n] = A[n]e^{j\phi[n]} \tag{3.107}$$

[14] Definido por Marple [Mar99], com a ressalva de que há valores negativos de frequência com resposta não nula nesse caso, devido à periodicidade da DTFT, o que faz Marple utilizar aspas ao utilizar o nome "analítico" em tais sinais.

[15] Utilizando a soma do produto elemento a elemento dos sinais.

[16] Marple enfatiza a necessidade de se utilizar o valor 1 para a frequência de Nyquist. Entretanto, vale salientar que na situação em que o valor de N é ímpar, não existe coeficiente da DFT representando a frequência de Nyquist.

em que $A[n]$ é a envoltória dinâmica (*envelope*), e $\phi[n]$ é a fase. Em outras palavras, o valor absoluto do sinal analítico é a própria envoltória dinâmica do sinal.

Essa estratégia não está implementada na versão atual da AudioLazy devido a uma inconsistência que ela gera com relação à padronização do passo na divisão em blocos. O código fonte em 3.15 explicita como uma tal implementação é possível, dependendo de um único recurso não disponível com o pacote, que é o *overlap-add* ou "sobrepor e somar". Há maiores detalhes sobre esse assunto no segundo apêndice, na seção em que trata de uma maneira de inserir dinamicamente à AudioLazy o recurso utilizado por esta estratégia.

Código-fonte 3.15: *Envoltória através da obtenção do sinal analítico (transformada de Hilbert)*

```
from audiolazy import *

@envelope.strategy("hilbert")
@overlap_add()
def envelope(sig, size=256, hop=None):
  for blk in chain(sig, Stream(0)).blocks(size=size, hop=hop):
    yield abs(hilbert(blk))
```

A figura 3.16 permite a avaliação das diferentes formas de se obter a envoltória dinâmica do sinal, porém ainda sem esgotar as implementações disponíveis com a AudioLazy[17], e o código-fonte 3.16 é responsável por criar os sinais representados na figura. Pode-se observar, de imediato, que para um sinal de 3 segundos de duração as senoides envolvidas são bastante lentas. A taxa de amostragem é de 8000 amostras por segundo. O filtro passa-baixas (de um polo) teve frequência de corte de 1 Hz para evitar que este apenas acompanhasse o módulo do sinal.

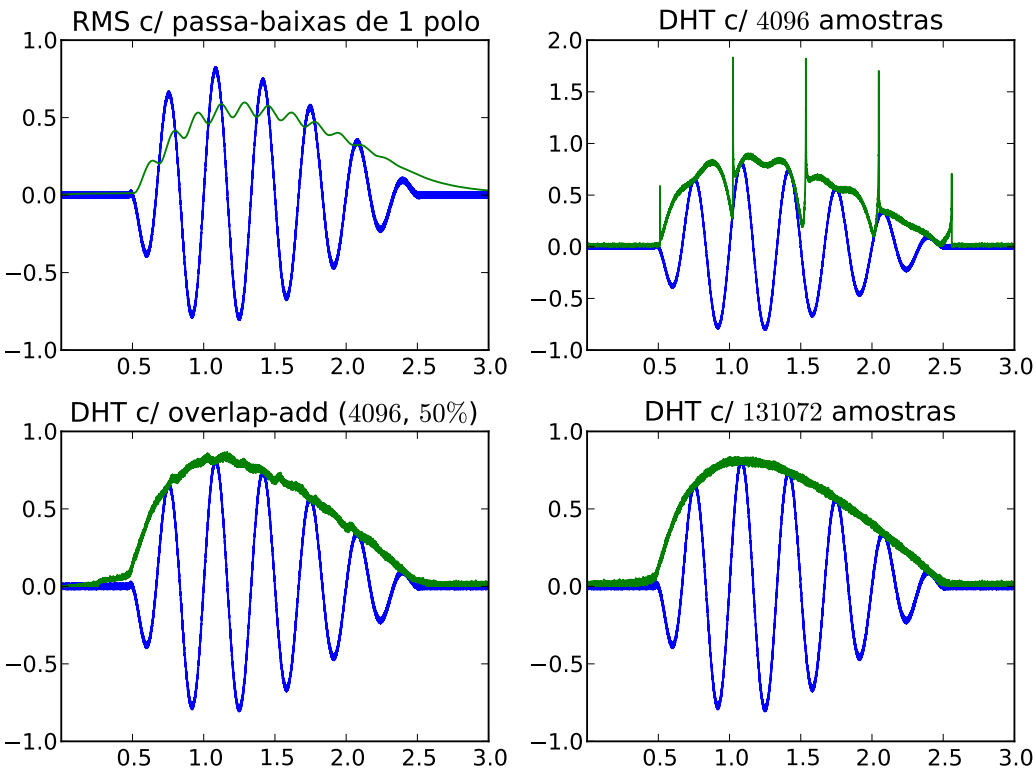


Figura 3.16: *Diferentes estratégias para a obtenção da envoltória temporal*

Código-fonte 3.16: *Diferentes possibilidades de obtenção da envoltória dinâmica. As listas em e1, e2, e3 e e4 são as envoltórias utilizadas junto com sig para a obtenção dos gráficos.*

```
from audiolazy import *
```

[17] Especificamente, as estratégias *abs* e *squared* não foram utilizadas.

```
2
3 rate = 8000
4 s, Hz = sHz(rate)
5 dur = int(round(3 * s))
6 time_data = list(line(dur, 3., 0.))[::-1]
7 sig = -(1.2 / Stream(time_data)) - Stream(time_data) + 3
8 sig = clip(sig, low=0., high=inf)
9 sig *= sinusoid(3 * Hz)
10 sig += white_noise() * .02
11 sig = sig.take(dur)
12
13 e1 = envelope(sig, cutoff=1 * Hz).take(dur))
14 e2 = envelope.hilbert(sig, size=4096).take(dur))
15 e3 = envelope.hilbert(sig, size=4096, hop=2048).take(dur))
16 e4 = envelope.hilbert(sig, size=2**17).take(dur))
```

No gráfico do canto superior esquerdo, pode-se observar que o a implementação com a estratégia *rms* teve dificuldade em acompanhar a senoide tão lenta, de forma que esta perdeu o valor de pico original, não se desfez da oscilação, e chegou ao zero com um pouco de atraso ao final. Não atingir o valor de pico é um resultado esperado para a envoltória RMS. O ruído de alta frequência do sinal foi suficientemente atenuado para ser desprezível no resultado.

No gráficos dos cantos superior direito, tentou-se utilizar a estratégia com o sinal analítico (e a transformada de Hilbert) diretamente em blocos de 4096 amostras. Pode-se visualizar com clareza que o resultado gerou saltos abruptos com *spikes* nos pontos de transição entre os blocos. Visando resolver isso, os blocos passaram a ser tomados a cada 2048 amostras, multiplicados por uma função triangular de pesos (*window.triangle* na AudioLazy) e somados com o devido deslocamento no tempo. O resultado, conhecido como *overlap-add*, possibilitou o uso prático do recurso sem incluir degenerações e saltos indesejáveis.

O último gráfico foi realizado com um número significativamente maior de amostras, de forma a cobrir a figura inteira em uma única aplicação da equação 3.105. O resultado confirma que a amplitude do sinal analítico como um todo, nesse caso, é a própria envoltória dinâmica. Pode-se notar que em todos os casos envolvendo o sinal analítico, o ruído de alta frequência persistiu.

Como vantagem do uso do passa-baixas na estratégia envolvendo valores em RMS, esta não necessita de processamento em bloco, e é mais eficiente computacionalmente (mas não assintoticamente, caso o tamanho do bloco seja constante) que o algoritmo implementado com o sinal analítico, dado que o passa-baixas apenas utiliza uma amostra armazenada em memória e é factível com duas multiplicações e uma subtração por amostra, enquanto a obtenção do sinal analítico necessita de uma organização em blocos para uma aplicação e utiliza da DFT, que necessita de tempo da ordem $O(N \log N)$ sobre o tamanho N do bloco, mas que se torna $O(n \log N) = O(n)$ quando aplicado diversas vezes consecutivas em blocos de tamanho fixo, isto é, a complexidade vista com relação ao tamanho de bloco indica que o filtro passa-baixas é mais eficiente, mas vista com relação ao tempo, são ambas lineares.

3.2.10 Síntese

Segue nesta seção uma breve introdução à síntese sonora, com base nas capacidades que foram inseridas na AudioLazy ou que são potencialmente realizáveis, de maneira imediata, utilizando o pacote. Esta lista certamente não visa esgotar todas as possibilidades de algoritmos de síntese em computação musical. Sobre esses e outros algoritmos de síntese, podem ser encontradas maiores informações no livro de Moore [Moo90] e no DAFx [Zöl11].

Síntese Aditiva

A síntese aditiva é aquela realizada através da soma de sinais primitivos, tais como senoides ou formas de onda consideradas básicas, determinísticas como a forma de onda dente-de-serra (*sawtooth*), ou estocásticas como o ruído branco.

Na AudioLazy, essas primitivas foram inseridas, além do modelo por consulta a tabela, o qual permite o uso de formas de onda variáveis no tempo assim como frequência variável no tempo. Trata-se de um modelo simples de síntese por consulta a uma tabela de valores, armazenados em uma lista considerada como período de uma sequência cíclica, e acessados em posições indexadas, possivelmente fracionárias. Nos casos de valores em posições fracionárias, o valor é obtido por interpolação. Na classe *TableLookup*, que implementa tal recurso na AudioLazy, é utilizada a interpolação linear entre amostras, mas essa não é a única possibilidade de interpolação existente.

Síntese Subtrativa

O que caracteriza a síntese subtrativa é a presença de uma fonte de áudio da qual parte da informação é filtrada, subtraída. Isto é, a síntese subtrativa é caracterizada quando um algoritmo tem como componente fundamental a filtragem. Um modelo simples de síntese é o modelo de Karplus-Strong, que se restringe a filtrar com um filtro *comb* sua memória inicial que, por padrão, é um ruído branco.

Como o atraso desse particular filtro pode ser fracionário, o algoritmo da forma como foi implementado na AudioLazy utiliza uma interpolação linear dos atrasos, uma das cinco propostas de implementação de atrasos fracionários listadas em [Zöl11, p. 73-75].

Código-fonte 3.17: *Síntese de um acorde com o algoritmo de Karplus-Strong*

```
from audiolazy import *
from time import sleep

rate = 44100
s, Hz = sHz(rate)

freqs = [str2freq(note) * Hz for note in "C3 A3 E4 A4".split()]
chord = sum(karplus_strong(f) for f in freqs)

with AudioIO() as player:
  player.play(chord, rate=rate)
  sleep(3) # Seconds
```

Outro possível modelo de síntese é o trompete de Horner e Beauchamp [HB95], algoritmo implementado apenas para a avaliação do algoritmo de Klapuri [Kla08], mas que não faz parte do pacote AudioLazy. A importância desse modelo de síntese de trompete é a de que ele necessita de um ressonador variante no tempo, e serviu de motivação para a inserção do recurso de filtros variantes no tempo e JIT na AudioLazy.

Síntese AM (Modulação em amplitude)

A modulação em anel e a modulação em amplitude são diferentes formas de modulação de um sinal a partir de outro. Na AudioLazy isso pode ser feito facilmente multiplicando dois objetos Stream (modulação em anel), ou somando 1 a algum deles antes da multiplicação (modulação em amplitude), caso em que esse 1 representa a soma da portadora ao sinal resultante. Klapuri [Kla08] exemplificou que seu algoritmo baseado em filtros gammatone recupera a altura de ruído branco modulado em amplitude para possuir uma frequência fundamental perceptualmente definida. Há um curto exemplo de sinal de sintese por modulação em amplitude na seção 3.3.6

Síntese FM (Modulação em frequência ou fase)

A síntese FM realiza a aplicação de uma senoide como frequência de outra senoide, ou uma senoide como fase de outra. O código-fonte 4 no segundo apêndice inclui um exemplo de síntese de áudio em tempo real utilizando a senoide da AudioLazy. O exemplo realiza uma avaliação prática, porém incompleta (pois não há gravação de áudio e valores objetivamente mensurados), sobre a capacidade de processamento em tempo real da AudioLazy. Isso foi feito como uma GUI em wxPython, na qual a AudioLazy foi utilizada tanto para a

realização de uma síntese FM (Frequency Modulation) como da parte gráfica na qual há uma janela retangular preta contendo um círculo amarelo girando em uma órbita no formato de uma elipse branca que se adapta ao tamanho da janela. A animação do círculo girando é feita utilizando a função *modulo_counter* da AudioLazy, um contador que devolve um valor sempre na faixa de $[0, M)$, resultado do resto da divisão por M após cada incremento, função esta utilizada também como parte fundamental do algoritmo de consulta à tabela da AudioLazy. A velocidade de rotação é controlada pela posição na tela, a intensidade dinâmica é controlada pelo tamanho da janela, e a movimentação da janela na tela modifica as relações frequenciais de forma a tornar o exemplo de funcionalidade e aplicação um *software* de uso lúdico.

3.3 Altura, croma e timbre

Esta seção visa lidar com MIR, sobretudo com a obtenção da frequência fundamental de um segmento de um som, que tem como correlato psicoacústico a percepção de altura, e a obtenção da informação do croma, que possui de certa forma uma correlação com a harmonia em música. Há também informações sobre formantes e envoltória espectral, aspectos relacionados ao timbre.

Bregman [Bre94, p. 92] alega que a palavra timbre passou a ser uma "lata de lixo", pois tudo aquilo que não tinha nome em um som passou a fazer parte do timbre, sugerindo que seu uso dessa forma seja evitado. Duas definições comuns para o timbre são:

1. qualidade do som que permite identificar sua fonte;
2. conjunto de qualidades do som que estão além da altura e da intensidade média (visto neste trabalho como envoltória dinâmica ou temporal);

O brilho costuma ser visto como correlato perceptual do centróide espectral [Pee04], possibilitando sua simples modelagem em sistemas computacionais. Este modelo não foi implementado na AudioLazy até o atual momento, mas sua implementação poderia ser realizada facilmente com

$$\frac{\sum_k k|f[k]|}{\sum_k |f[k]|} \tag{3.108}$$

em que $f[k]$ é o k-ésimo coeficiente da DFT de um bloco de áudio. Há maiores detalhes sobre a DFT na seção 3.3.3.

Apesar de, no caso da altura, a informação desejada ser correlata a uma frequência, tal informação precisa estar presente de alguma forma no domínio do tempo, visto que a única informação existente sobre o áudio são suas amostras no tempo, e a reprodução deste áudio é suficiente para um ouvinte humano percebê-la, quando esta existe. Entretanto, humanos percebem polifonia, e a maioria dos algoritmos de MIR que são apresentados nas seções que seguem não estão preparados para a polifonia.

3.3.1 O modelo de Shepard

A palavra *pitch* do inglês está associada a dois elementos perceptuais distintos utilizados em português: a altura (*pitch height*) e o croma (*pitch chroma* ou *tonality*) [She64]. Esses elementos sugerem que a modelagem conjunta da altura e do croma se dê em um mapeamento geométrico helicoidal, pois há um ciclo a cada oitava na dimensão do croma, enquanto a altura pode se tornar mais aguda ou mais grave monotonicamente. Shepard comentou que a sensação associada a intervalos de sétima ascendentes tinha algum componente que caminhava na direção oposta[18]. Parncutt [Par89] utiliza explicações similares para efetuar uma distinção entre a distância com relação à altura (*pitch distance*) e a distância com relação ao croma (*pitch commonality*).

[18] Tais intervalos são formados quando as frequências dos componentes parciais de um primeiro som são multiplicadas por um número próximo de $2^{10/12}$ (sétima menor) ou $2^{11/12}$ (sétima maior), valores próximos ao número 2, que representa o intervalo de oitava. Esses números se referem à escala temperada de 12 sons. É possível notar que para um dado f, $2f/2^{1/12} = f2^{11/12}$, ou seja, intervalos ascendentes de sétima são intervalos descendentes de segunda a partir da oitava, justificativa da sensação relatada por Shepard.

Shepard propôs um experimento com áudio sintético que cria a ilusão de que o som está sempre "subindo" ou sempre "descendo" em sua altura, explicitando a circularidade do croma [She64]. Para isso, componentes senoidais variáveis no tempo espaçadas logaritmicamente têm sua frequência modificada de forma a manter uma envoltória espectral fixa, isto é, a amplitude de cada senoide é definida a partir da frequência. Com uma envoltória espectral fixa que permite a introdução de novos componentes senoidais e a finalização de outros de maneira suave com variações de intensidades, tenta-se esconder que a rigor, trata-se de um som cíclico, mas que traz o efeito de que tudo está permanentemente subindo ou descendo[19]. O código-fonte 3.18 traz uma possível implementação do som de Shepard realizado apenas com oitavas, embora com algumas adaptações como, por exemplo, o uso de uma função de apodização de Blackman no lugar de um cosseno deslocado (como Hamming e Hann), e a aplicação de uma taxa de amostragem significativamente superior a 10 mil amostras por segundo.

Código-fonte 3.18: *Implementação do som de Shepard na AudioLazy*

```
from audiolazy import *
rate = 44100
s, Hz = sHz(rate)
kHz = 1e3 * Hz

# Parâmetros
table_len = 8192
min_freq = 20 * Hz
max_freq = 20 * kHz
duration = 60 * s
normalization_factor = 50.8 # Apenas para controle de ganho

# Parciais
noctaves = abs(log2(max_freq/min_freq))
octave_duration = duration / noctaves
smix = Streamix()
data = [] # Global: história de um parcial para uso futuro
env_table_data = window.blackman(table_len)
env_table = TableLookup(env_table_data)
env_table /= normalization_factor # Normaliza considerando as senoides somadas

# Inicializa a lista "data"
def partial():
  smix.add(octave_duration, partial_cached()) # Evento do parcial seguinte
  # Sequência de valores de frequência em escala logarítmica:
  partial_freq = (2 ** line(duration) - 1) * (max_freq - min_freq) + min_freq
  env = Stream(env_table[f] for f in line(duration, 0, table_len)) # Envelope
  for el in env * sinusoid(partial_freq):
    yield el
    data.append(el)

# Replicador sem fim
def partial_cached():
  smix.add(octave_duration, partial_cached()) # Evento do parcial seguinte
  for el in data:
    yield el

# Caso inicial
smix.add(0, partial())

with AudioIO(True) as player:
```

[19] Isso pode ser visto como um equivalente sonoro do efeito cíclico que o artista plástico M. C. Escher trouxe em sua obra, sobretudo na década de 1960.

```
42 player.play(smix)
```

3.3.2 Série harmônica

Série harmônica é o nome dado ao conjunto de harmônicos de um som periódico. À frequência que gera a série harmônica é dado o nome de frequência fundamental , e os harmônicos são senoides indexadas através de sua relação de frequência com a fundamental. Séries harmônicas são muito comuns em sinais de áudio, devido às condições físicas de diversos instrumentos musicais, que privilegiam a vibração nas frequências aproximadamente múltiplas de uma frequência fundamental.

Na prática, nenhum som é infinito, de forma que não faz sentido falar em som puramente periódico, mas faz sentido falar em periodicidade ou quase periodicidade em um segmento localizado de um sinal. Em um caso mais livre, no qual a periodicidade é apenas aproximada, cada um desses componentes frequenciais senoidais é chamado de parcial, e o nome série harmônica é mantido no caso em que um conjunto de parciais possui uma mesma frequência fundamental.

Um som caracterizado por uma série harmônica que possui alguns harmônicos mas que não inclui a frequência fundamental pode ser percebido com altura correlata à sua frequência fundamental omissa. Há pelo menos dois elementos nesse som que podem servir de auxílio à percepção do ouvinte:

1. o som descrito é periódico no tempo, com período τ; e a frequência fundamental omissa tem valor $1/\tau$;

2. a diferença de frequência entre dois parciais consecutivos é igual à frequência fundamental omissa.

Com a intenção de possibilitar a transcrição polifônica, Klapuri [Kla97] obteve um resultado generalista sobre a interação de diferentes séries harmônicas exatas, concluindo que:

1. duas séries harmônicas somente podem possuir algum harmônico em comum quando suas frequências fundamentais f_1 e f_2 satisfazem $f_1/f_2 \in \mathbb{Q}$;

2. se duas séries harmônicas possuem dois (ou mais) harmônicos primos em comum, as frequências fundamentais f_1 e f_2 satisfazem $f_1 = 2^n f_2$ para algum $n \in \mathbb{Z}$, ou seja, pertencem à mesma classe de oitavas.

Porém esse tipo de formulação feita por Klapuri se restringe a harmônicos exatos. Qualquer desvio entre duas frequências que seja classificado como não pertencente ao conjunto $\mathbb{Q}$ dos números racionais torna os harmônicos abruptamente independentes, e pelo fato de $\Delta = \sqrt[12]{2}$ ser o intervalo de semitom na escala cromática, segue que, em teoria, apenas os intervalos de oitava possuem harmônicos em comum na escala cromática, com base na identificação realizada por Klapuri. Uma outra forma de se chegar a um tipo de conclusão similar quanto à intersecção entre séries harmônicas é através da soma da resposta em frequência de filtros *comb*, com a diferença de que uma tal soma inclui valores de intensidade, e permite que valores próximos tenham suas influências avaliadas. Quanto maior a intensidade das intersecções das respostas dos filtros *comb* em uma dada frequência, menor é a informação fornecida por um componente frequencial que possua essa frequência, caso a intenção seja classificar esse componente como pertencente a uma das séries harmônicas plausíveis. Para obter o resultado visual da soma de respostas em frequência com a AudioLazy basta utilizar a configuração em paralelo dos filtros *comb*, como feito no código-fonte 3.19, colocado na figura 3.17. Há, contudo, um fato que deve ser levado em consideração: a interferência entre senoides de mesma frequência. Sendo filtros lineares, a resposta em frequência da soma dos filtros é a soma das respostas em frequência de cada filtro, entretanto a soma desses números complexos de resposta em frequência não implica na soma de suas amplitudes. Dependendo da diferença de fase de cada filtro somado em cada frequência, essa interferência pode ser construtiva ou destrutiva, de forma que não se pode afirmar como as componentes de filtros somados irão se comportar sem a informação da fase. A figura 3.18 contém a soma do módulo das respostas em frequência dos três filtros, também em escala logarítmica de amplitude[20], para

[20] Apesar da escala logarítmica da figura, a soma é realizada com valores absolutos, i.e., trata-se de uma simulação do que ocorreria na situação de máxima interferência construtiva.

fins de comparação. Pode-se observar que em torno de 730 Hz há um vale de interferência destrutiva na figura 3.17 trazendo um ganho pouco menor que 10 dB, enquanto que a situação de máxima interferência construtiva descrita na figura 3.18 indica um ganho em torno de 17 dB para a mesma frequência. Tudo depende do uso para o qual o filtro será aplicado: o uso em paralelo de três filtros *comb* pode ser visto como um análogo do modelo de síntese de Karplus-Strong, mas aplicado em um acorde de mi maior ao invés de uma altura única.

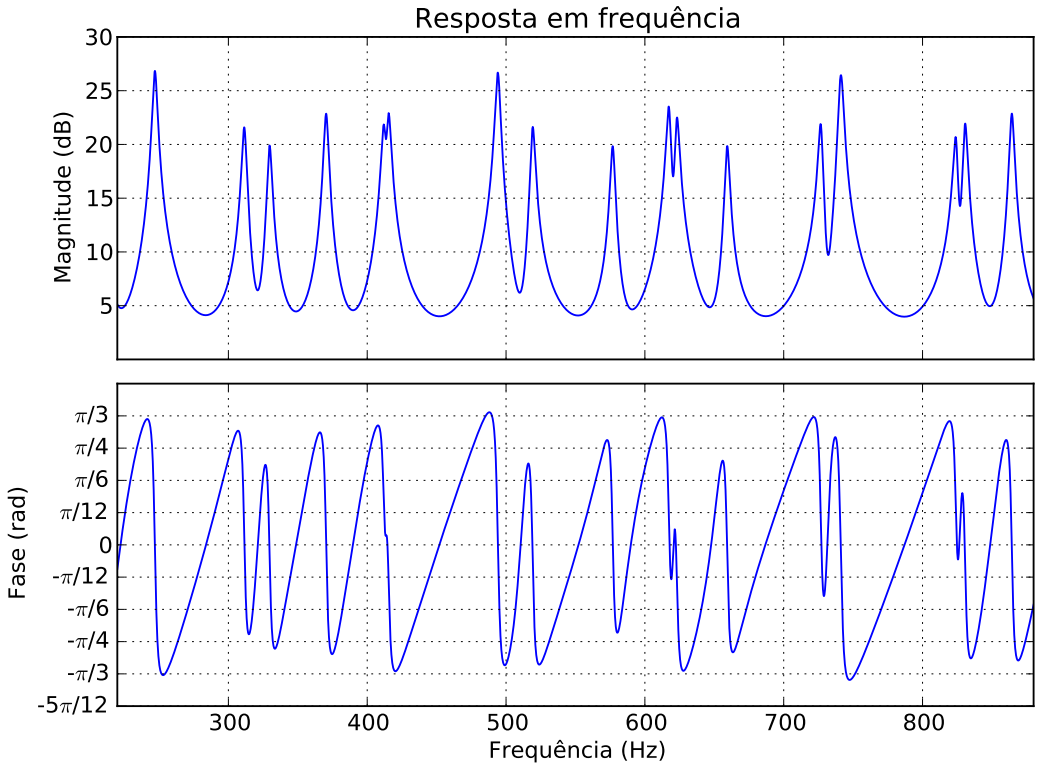


Figura 3.17: *Sobreposição de filtros comb relativos a um acorde de mi maior*

Código-fonte 3.19: *Sobreposição de filtros comb relativos a um acorde de mi maior*

```
from audiolazy import *

rate = 44100
s, Hz = sHz(rate)
freqs = [str2freq(note) for note in "E2 G#2 B2".split()] # Mi maior
filt = ParallelFilter(comb.tau(freq_to_lag(freq * Hz), .1 * s)
                      for freq in freqs)

filt.plot(samples=8192, rate=rate, min_freq=220*Hz, max_freq=880*Hz).show()
```

3.3.3 Transformada de Fourier em tempo discreto (DFT)

A transformada de Fourier vista na seção 1.1.1 utiliza uma função em $\mathbb{R}$ para representá-la no domínio da frequência, enquanto que a transformada em tempo discreto (DTFT) vista na seção 3.1.2 realiza um processo similar de decomposição de uma entrada em componentes frequenciais, com a diferença de que a entrada passou a ser uma sequência de valores reais, e a saída uma função real periódica de período 2π. Uma extensão à ideia é a transformada discreta (circular) de Fourier, ou DFT (*Discrete Fourier Transform*)[21], que é realizada

[21] Esses e maiores detalhes em [OSB99, capítulos 8 e 9]

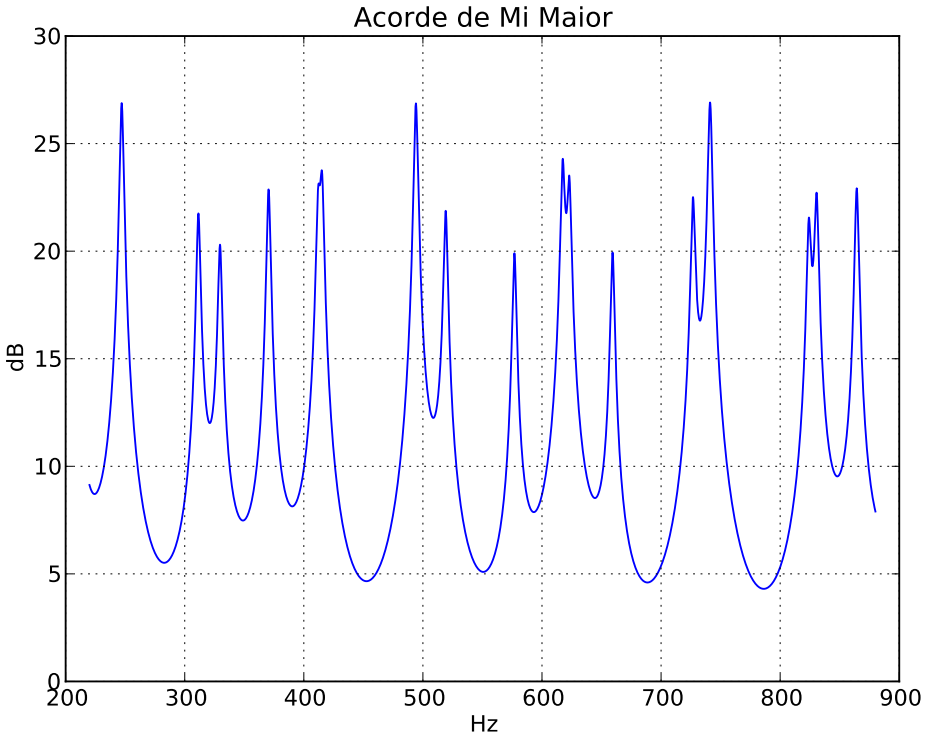


Figura 3.18: *Soma do módulo da resposta em frequência de filtros comb relativos a um acorde de mi maior*

sobre apenas um segmento de N amostras consecutivas do sinal digital admitido como um "período":

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi nk}{N}}, \quad k = 0, 1, \cdots, N-1 \tag{3.109}$$

em que j é a unidade imaginária e $\omega = 2\pi k/N$ é a frequência em radianos por amostra do k-ésimo componente senoidal da transformada, representado pelo coeficiente $X[k]$. A transformada inversa da DFT é dada por:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi nk}{N}}, \quad n = 0, 1, \cdots, N-1 \tag{3.110}$$

Uma compreensão da DFT pode ser obtida através da concatenação do segmento de sinal utilizado com cópias dele próprio, formando um sinal discretizado no tempo periódico e de período igual a N amostras. Tal sinal periódico também poderia ser representado por uma série de Fourier: cada um dos N coeficientes dessa transformada está associado a um número inteiro de ciclos dentro do período, equivalentemente ao harmônico da série de Fourier.

Existem algoritmos conhecidos como FFT (*Fast Fourier Transform*) que realizam a transformada em tempo $\Theta(N \log N)$ para calcular o valor dos N coeficientes quando N é uma potência inteira de 2.

Uma característica notável na DFT é que quanto maior for o valor de N, maior é o número de coeficientes e de amostras necessárias para o cálculo, o que significa que a resolução no domínio da frequência aumenta enquanto a resolução no domínio do tempo fica cada vez mais suavizada, além do fato de que esse procedimento passa a criar uma latência mínima inevitável ao sinal: para processar 512 amostras de uma vez, é necessário possuir as 512 amostras, o que significa que depois do recebimento de uma primeira amostra ainda é necessário aguardar outras 511 amostras antes que o resultado seja obtido a partir do bloco[22].

Uma maneira de implementar a DFT para valores específicos de frequência é o algoritmo de Goertzel, o qual representa o equacionamento na forma de um filtro digital [OSB99, p. 633-635]. Trata-se de um

[22] É possível, entretanto, utilizar a amostra junto a outro bloco quando há um passo menor que o tamanho do bloco.

ressonador em torno da frequência central ω (radianos por amostra) com polos no círculo unitário (i.e., $R = 1$, situação na qual a frequência do cosseno do denominador coincide com a frequência de ganho máximo), e com um único zero igual a $e^{-j\omega}$. O algoritmo de Goertzel é computacionalmente mais eficiente que a FFT apenas quando a quantidade de coeficientes DFT avaliados é pequena, especificamente quando é utilizado para avaliar menos de $log_2(N)$ coeficientes. Adicionalmente, é um filtro que acumula a influência de todo o histórico, não satisfazendo o critério BIBO de estabilidade. Espera-se que esse algoritmo seja associado a algum processamento não linear de reinício ou esvaziamento de memória de tempos em tempos, ou que uma alternativa seja utilizada para que seus polos estejam dentro do círculo unitário.

Existem ainda outras alternativas para o cálculo da FFT. Por exemplo, Desainte-Catherine e Marchand [DCM00] propuseram uma maneira de, através da utilização da derivada do sinal, encontrar valores refinados para os parciais de um som, admitindo que estes variam lentamente.

A AudioLazy não inclui a FFT em seu pacote, mas isso pode ser feito, por exemplo, através do NumPy. A AudioLazy contém a DFT permitindo o cálculo direto com a definição da transformada sobre um conjunto de frequências, mas esse procedimento existe apenas para facilitar a conversão entre unidades (parâmetros em radianos por amostra) e a realização de gráficos de resposta em frequência envolvendo LPC, sendo indesejável seu uso para aplicações diretas em todos os valores inteiros de ciclos da DFT de 0 a $N - 1$ em multiplos blocos, visto que o algoritmo terá complexidade computacional $O(N^2)$ nessas situações.

Dentre as diferenças notáveis entre a DFT da AudioLazy e a FFT do NumPy inclui-se o fato de que a FFT do NumPy, quando aplicada com um valor de N mas um vetor de tamanho maior que N, descarta os valores adicionais, enquanto que a DFT utiliza todos os valores, utilizando o equacionamento original para frequências ω quaisquer. É possível implementar o algoritmo FFT usando a AudioLazy como se vê no código-fonte 3.20, mas sugere-se que o desenvolvedor utilize a FFT do NumPy, tanto por questões de otimização como por conta da interface, visto que o NumPy já está preparado para realizar o cálculo com blocos de entrada com tamanhos que não sejam potências de 2, enquanto o código-fonte 3.20 possui essa restrição quanto ao tamanho de seu primeiro argumento posicional.

Código-fonte 3.20: *Algoritmo FFT de Cooley-Tukey implementado sobre a AudioLazy*

```
1  def fft(blk, size):
2    n = len(blk) if size is None else size
3    result = Stream(blk, zeros()).take(n)
4    if n != 1:
5      ndiv2 = n >> 1
6      yev = fft(result[0::2], ndiv2)
7      yod = fft(result[1::2], ndiv2)
8      omega_n = cexp(-2j * pi / n)
9      omega = 1
10     for idx, (yevi, yodi) in enumerate(zip(yev, yod)):
11       omega_yodi = omega * yodi
12       result[idx] = yevi + omega_yodi
13       result[idx + ndiv2] = yevi - omega_yodi
14       omega *= omega_n
15   return result
```

Uma implementação da transformada inversa da FFT pode ser facilmente obtida adaptando-se o código-fonte 3.20 de forma a possuir expoente positivo de `omega_n`, chamar a própria inversa da FFT ao invés da FFT, e multiplicar o resultado por $1/n$, algo que pode ser feito dividindo-se por 2 a atribuição das linhas 13 e 14.

3.3.4 Taxa de cruzamentos no zero

Sinais de áudio, quando possuem suas amostras armazenadas em ponto flutuante, costumam tê-las como amplitudes na faixa de $[-1; 1]$, normalmente com componente constante igual a zero, de maneira a maximizar o aproveitamento da faixa de valores disponível. Na ausência de um tal componente constante (frequência nula), o sinal, caso tenha alguma componente, cruzará o zero a partir de alguma amostra. A detecção de

cruzamento por zeros é um simples algoritmo que devolve 1 quando o sinal de duas amostras consecutivas é diferente, e 0 caso contrário. Outra forma de visualizar o mesmo algoritmo é observar o zero como um limiar, e um sinal 1 é emitido quando esse limiar é ultrapassado.

Uma variante desse processo inclui o que é chamado de *histerese*, na qual há dois limiares disponíveis e o limiar válido/engatilhado em um dado instante é variável: no instante em que um limiar é ultrapassado, este é modificado, a fim de evitar que pequenas oscilações em torno do limiar criem um ruído indesejável (*bounce*).

Sinais periódicos mantêm uma taxa de cruzamentos por zero constante, de forma que o algoritmo pode ser utilizado para identificar a frequência fundamental de um som. Como é de se esperar, podem haver erros, por exemplo quando em um perído o sinal passa por zero mais vezes do que o esperado, fazendo com que o valor medido tenha uma relação de oitava com o valor real, ou corresponda a um harmônico de ordem superior à oitava.

Esse algoritmo foi inserido na AudioLazy sob o nome de *zcross*, no módulo de análise, e inclui a possibilidade de histerese. Segue abaixo um exemplo de obtenção de alturas através do zcross, incluindo um caso de acerto e um caso de erro com relação à oitava.

Código-fonte 3.21: *Obtenção das alturas a partir da taxa de cruzamentos no zero*

```
In [1]: from audiolazy import *
In [2]: rate = 44100
In [3]: s, Hz = sHz(rate)

In [4]: # Primeiro sinal: senoide, dente de serra e ruído multiplicativo
In [5]: data1 = .5 * sinusoid(880 * Hz)
In [6]: data1 += .5 * saw_table(880*3 * Hz)
In [7]: data1 *= .9 + .1 * white_noise()

In [8]: # Segundo sinal: harmonicos ímpares e ruído aditivo
In [9]: table = sin_table.harmonize({1: 1, 3: 1, 5: 1, 7: .5, 9: .2})
In [10]: data2 = .9 * table(220 * Hz)
In [11]: data2 += .1 * white_noise()

In [12]: @tostream
    ....: def zcross_pitch(sig, size=2048):
    ....:   "Devolve a altura em cada bloco com o dado tamanho"
    ....:   for blk in zcross(sig, hysteresis=.2).blocks(size):
    ....:     yield lag_to_freq(2. * size / sum(blk))
    ....:

In [13]: # Avalia o primeiro sinal
In [14]: pitch1 = zcross_pitch(data1) / Hz

In [15]: pitch1.take(10) # Resultado em Hz
Out[15]:
[872.0947265625001,
 882.861328125,
 872.0947265625001,
 882.861328125,
 882.861328125,
 882.861328125,
 882.861328125,
 872.0947265625001,
 882.861328125,
 872.0947265625001]

In [16]: freq2str(pitch1).take(10) # Resultado em nomes de notas
Out[16]:
['A5+5.62%',
```

```
 'A5+5.62%',
 'A5+5.62%',
 'A5+5.62%',
 'A5+5.62%',
 'A5-15.62%',
 'A5+5.62%',
 'A5-15.62%',
 'A5+5.62%',
 'A5+5.62%']

In [17]: # Avalia o segundo sinal (errando a oitava)
In [18]: pitch2 = zcross_pitch(data2) / Hz

In [19]: pitch2.take(10) # Resultado em Hz
Out[19]:
[430.6640625,
 441.4306640625,
 441.4306640625,
 441.4306640625,
 441.4306640625,
 441.4306640625,
 441.4306640625,
 430.6640625,
 441.4306640625,
 441.4306640625]

In [20]: freq2str(pitch2).take(10) # Resultado em nomes de notas
Out[20]:
['A4+5.62%',
 'A4+5.62%',
 'A4+5.62%',
 'A4+5.62%',
 'A4-37.13%',
 'A4+5.62%',
 'A4+5.62%',
 'A4+5.62%',
 'A4+5.62%',
 'A4+5.62%']
```

3.3.5 Outros algoritmos para a obtenção da frequência fundamental

Há diversos outros algoritmos para a obtenção da frequência fundamental de um som, e dez destes foram listados por McLeod e Wyvill [MW05], sem incluir o método proposto por esses dois autores. Barbosa também destacou alguns dos mesmos algoritmos [Bar02, p.44-46].

Um desses métodos que já está implementado na AudioLazy é o AMDF (*Average Magnitude Difference Function*, ou função média da diferença de magnitude), a qual está implementada na AudioLazy na forma de um filtro digital eficiente para um dado valor de atraso. O equacionamento da AMDF refere-se à soma de diferenças com relação a seus atrasos:

$$y_\tau[n] = \frac{1}{N} \sum_{i=0}^{N-1} |x[n-i] - x[n-i-\tau]| \tag{3.111}$$

que pode ser visto como três blocos: um filtro linear FIR cuja saída é aplicada a um bloco não-linear de obtenção do valor absoluto, para este resultado ser utilizado em um filtro média móvel. Tal equacionamento é utilizado como complemento em filtros de predição linear (seção 3.3.7) para modelar a altura [Bar02, Mak75].

Esse método foi utilizado para a criação da figura 3.19, na qual um sinal periódico de 11 amostras iguais a $[1, 2, 4, 3, 2, 3, 3, 3, 1, 1, 2]$ foi somado a um ruído branco uniforme de -1 a 1.

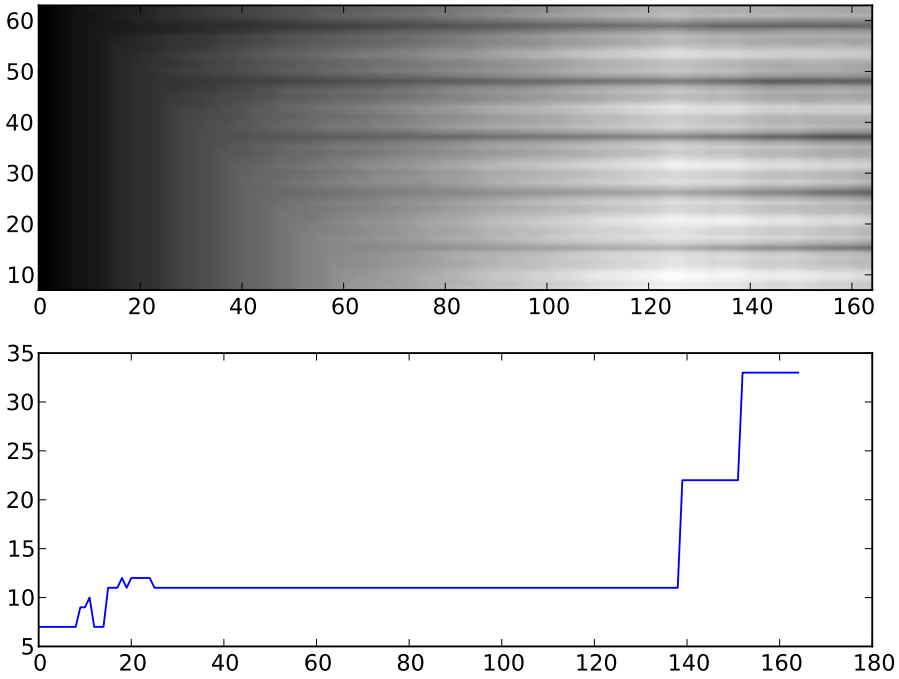


Figura 3.19: *Uso da AMDF para a obtenção da altura. O primeiro gráfico contém a AMDf para diversos valores de atraso, em que o eixo horizontal indica o tempo e o eixo vertical indica a magnitude. Na escala de tons de cinza da figura, o preto representa o zero. O segundo gráfico contém a frequência fundamental detectada em função do tempo através do valor de atraso que minimiza a integral do sinal da AMDF. Pode-se notar um erro quando os intervalos de tempo utilizados são muito grandes ou muito pequenos. O tamanho da janela de análise (N da média móvel) utilizado é igual ao dobro do maior atraso representável.*

Outros modelos incluem o quadrado da diferença que pode ser implementada analogamente à AMDF:

$$y_\tau[n] = \frac{1}{N}\sum_{i=0}^{N-1}(x[n-i] - x[n-i-\tau])^2 \tag{3.112}$$

e a função de autocorrelação, a qual pode ser vista como complementar à AMDF e ao quadrado da diferença no sentido de que o período é caracterizado pelo valor máximo do sinal resultante, ao invés do valor mínimo:

$$y_\tau[n] = \frac{1}{N}\sum_{i=0}^{N-1} x[n-i]x[n-i-\tau] \tag{3.113}$$

em que McLeod e Wyvill destacam duas possibilidades quanto à somatória envolvida no cálculo da ACF em blocos, quanto à inclusão ou não não de valores de índice fora de uma região delimitada.

O modelo proposto por McLeod e Wyvill trata de unificar essas duas últimas abordagens (eq. 3.112 e eq. 3.113), adaptadas para processamento em bloco, através de um processo de normalização da autocorrelação utilizando o quadrado da diferença[23]. Uma vantagem desse método proposto é que, além de fornecer o valor da frequência fundamental, a função de autocorrelação é utilizada como medida de qualidade da resposta, de forma a permitir avaliar em um segmento analizado quais regiões pode-se ou não confiar no resultado do

[23] A diferença entre a função quadrado da diferença e a variância móvel restringe-se a uma constante de normalização, de maneira que é bastante natural o procedimento proposto, de um ponto de vista estatístico.

modelo. Entretanto, nenhum desses modelos está preparado para a transcrição de polifonias, ou de múltiplas frequências fundamentais.

3.3.6 Múltiplas frequências fundamentais

Em 2008, Anssi Klapuri apresenta um modelo de transcrição polifônica de alturas baseado em filtros gammatone como modelo auditivo [Kla08]. O algoritmo foi implementado utilizando a AudioLazy e o NumPy, entretanto não foi disponibilizado junto ao pacote AudioLazy. Vale aqui realizar uma análise do que são pressupostos do algoritmo, e justificar o motivo do algoritmo não ter sido incluído no pacote AudioLazy. Infelizmente, não há uma implementação publicamente disponível do algoritmo para assegurar possíveis detalhes que Klapuri originalmente tinha em mente durante o projeto. Algumas dessas informações que não estão explícitas no artigo incluem o tipo de filtros passa-baixas utilizado e o valor da constante ζ utilizada em seu equacionamento, uma constante de normalização necessária para a implementação da otimização proposta. Outros valores foram interpretados com base no contexto. O filtro passa-baixas foi admitido ser um filtro de um pólo (ver seção 3.2.8), com uma exponencial como resposta ao impulso, embora não seja possível assegurar, com base apenas no texto do artigo, que esse tenha sido o filtro utilizado pelo autor. A ausência de implementações de referência e de certos detalhamentos na especificação dificulta, ou até mesmo impossibilita, a avaliação objetiva de certos aspectos de um algoritmo. Porém, nesse caso, acredita-se que há informação suficiente no artigo para permitir uma avaliação objetiva da proposta como um todo.

Originalmente, Klapuri explicitou sua intenção em criar um algoritmo suficientemente simplificado para funcionar em tempo real, entretanto a implementação realizada para esta seção teve como intenção apenas avaliar a funcionalidade do algoritmo e identificar as premissas utilizadas pelo autor, embora essas nem sempre estejam explícitas no artigo. Dessa forma, a descrição que fora dada por Klapuri foi mantida na implementação, embora sem incluir as simplificações que visavam sua otimização. Como é de se esperar em uma implementação não otimizada para a tarefa da obtenção de múltipla frequência fundamental, o algoritmo resultante é suficientemente lento para inviabilizar seu uso em tempo real, um dos motivos pelos quais o algoritmo não foi incluído no pacote AudioLazy. Por outro lado, o simplificado modelo de filtro gammatone proposto por Klapuri como uma cascata de 4 ressonadores faz parte do pacote AudioLazy (vide seção 3.2.8), o que significa que existe uma parte do algoritmo implementada na AudioLazy.

As figuras 3.20, 3.21, 3.22 e 3.23 presentes nesta seção visaram representar as figuras 3, 4, 5 e 6 do artigo de Klapuri, mas utilizando o pacote AudioLazy na implementação. Através da terceira figura do artigo foi possível inferir a taxa de amostragem de 22050 amostras por segundo. O modelo de implementação adotado por Klapuri segue quatro etapas em sequência, e são referentes a cada uma das divisões desta seção.

O banco de filtros gammatone de Klapuri

Os filtros gammatone de Klapuri definidos na seção seção 3.2.8 possuem uma fase não parametrizada e de identificação dificultada, dado que a única informação sobre a fase fornecida por Klapuri é a de que o sistea não foi testado para valores diferentes de fase. Pode-se alinhar as fases utilizando o filtro paramétrico com fase igual à fase da resposta em frequência na frequência central do filtro de Klapuri, conforme representado na figura 3.20.

Klapuri utiliza o ERB de Glasberg e Moore [GM90], um dos utilizados por Slaney [Sla93], que vale precisamente:

$$\mathrm{ERB}_{GM90}(f_c) = 24.7(4.37 \cdot 10^{-3} f_c + 1) \tag{3.114}$$

com frequência central f_c dada em hertz, de forma que a largura de banda em hertz para um um filtro gammatone centrado em f_c vale [HPNSR88]:

$$b_w(f_c) = \mathrm{ERB}(f_c) \frac{[(n-1)!]^2}{\pi[2(n-2)]2^{-2(n-2)}} \tag{3.115}$$

São utilizados 70 filtros gammatone com as dadas especificações, distribuídos em uma escala logarítmica referente a bandas críticas. O primeiro passo do algoritmo trata de aplicar o sinal em cada filtro, resultando

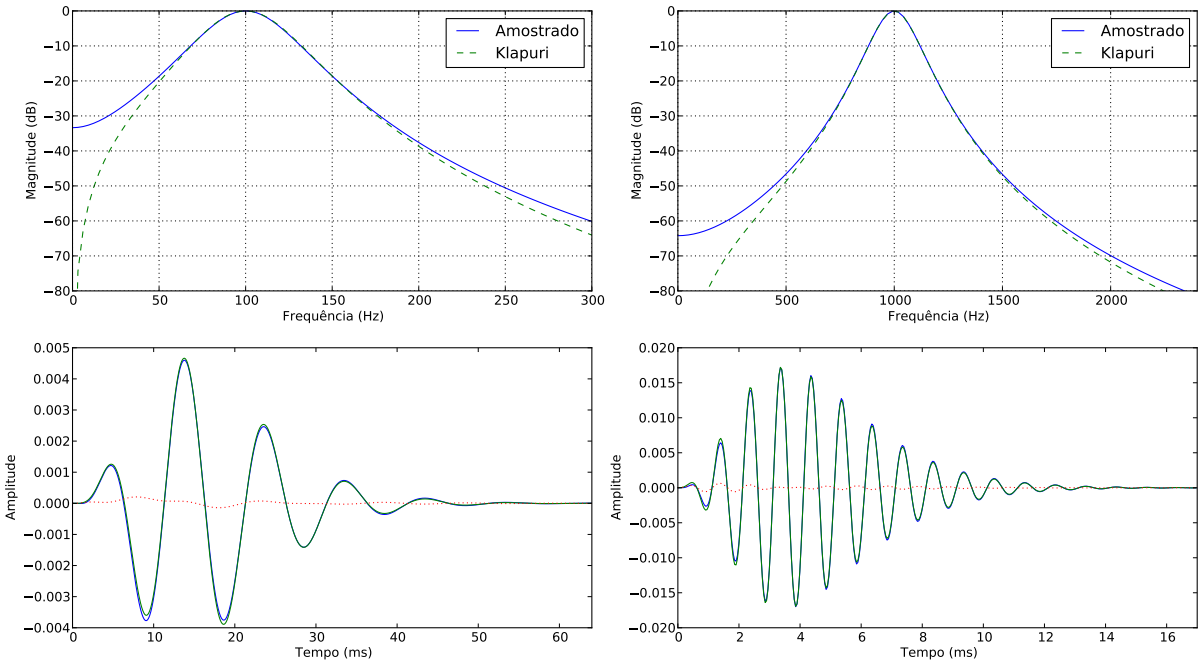


Figura 3.20: *Alinhamento de fase entre os filtros gammatone*

em 70 sinais.

Os equacionamentos supracitados relativos ao ERB e à obtenção da largura de banda estão implementadas no módulo *lazy_auditory* da AudioLazy. O caso de uso descrito no código-fonte 6 (apêndice de exemplos) e que gerou as figuras 3.13 e 3.14 serve de exemplo de como criar filtros gammatone a partir do dicionário de estratégias *erb*.

Processamento não-linear

O primeiro processamento não-linear realizado é o de compressão por uma potência do desvio padrão do bloco. Cada elemento de um bloco do sinal é dividido por $\sigma^{\nu-1}$, em que *sigma* é o desvio padrão do bloco e $\nu = 0.33$ é uma constante adotada por Klapuri para o procedimento. Segue abaixo uma implementação dessa rotina na AudioLazy que desfaz o processamento em bloco após a aplicação. Klapuri não é explícito quanto ao número de graus de liberdade a serem utilizados para o cálculo do desvio padrão, sendo considerado o valor 1 devido ao fato de que se trata de uma amostra e não uma população.

Código-fonte 3.22: *Compressão dinâmica (normalização) utilizada por Klapuri*

```
from audiolazy import tostream
import numpy as np
@tostream
def compression_klapuri(sig, size=1024, nu=.33):
  for block in sig.blocks(size):
    gain = np.std(block, ddof=1) ** (nu - 1)
    for el in block:
      yield gain * el
```

Klapuri é enfático ao atribuir toda a importância do modelo do aparato auditivo periférico ao processamento não-linear realizado por um corte abrupto (*clip*) sobre os valores negativos do áudio, e é dada pouca importância aos filtros gammatone. Seguindo a explicação dada pelo autor, o corte abrupto poderia ser realizado através do processamento não-linear:

$$\mathrm{HWR}(x) = \max(0, x) \tag{3.116}$$

que também poderia ser descrito como

$$\mathrm{HWR}(x) = \frac{x + |x|}{2} \tag{3.117}$$

mas que em nenhum dos casos corresponde à intenção de Klapuri. A descrição segue de maneira que o corte abrupto é posteriormente modificado para:

$$\mathrm{HWR}_{\mathrm{Klapuri}}(x) = \frac{x + L(|x|)}{2} \tag{3.118}$$

Em que $L(\cdot)$ refere-se à aplicação de um filtro passa-baixas na frequência de corte f_c. O objetivo desse procedimento, segundo Klapuri, é atenuar os componentes que surgem após o processamento não-linear em torno de $2f_c$. Esta etapa é exemplificada por Klapuri através de três figuras, as quais foram reconstruídas utilizando a AudioLazy. Da maneira como ficou descrito no artigo, não ficou claro a partir de que figura Klapuri estava utilizando o filtro passa-baixas, visto que tal detalhamento apenas foi descrito após a explicação de todas as figuras associadas à presente etapa do algoritmo, além de não ter explicitado como o processo de filtragem deve ser realizado. Visando manter a clareza e a coerência com as etapas seguintes do processo, todos os exemplos aqui presentes utilizam o filtro-passa baixas de um polo, com frequência de corte exata e igual à frequência central do gammatone.

Todo bloco, antes de ter sua DFT calculada, é multiplicado elemento-a-elemento por uma função de Hamming. Na figura 3.21, foi utilizado um sinal de trompete a 185 Hz no filtro gammatone com frequência central de 2700 Hz. Ausente o dado original utilizado por Klapuri nesse exemplo, adotou-se um modelo trompete para que fosse implementado na AudioLazy. O modelo de síntese de Horner e Beauchamp [HB95] foi implementado a partir de uma adaptação de seu código fonte em CSound, presente como anexo à sua especificação e análise[24]. A informação importante na figura é o surgimento de componentes de baixa frequência inexistentes anteriormente e, em particular, o surgimento da frequência fundamental. Não foi possível reconhecer os fatores de escala e tamanhos de blocos de análise originalmente adotados por Klapuri nessa figura, assim como não foi possível fazer a frequência fundamental ser mais proeminente que as demais frequências, como ocorre na figura do artigo, embora isto possa ser resultado da diferença entre o trompete utilizado por Klapuri e o modelo utilizado para a elaboração do gráfico.

Na figura 3.22, o mesmo som de trompete foi utilizado, dessa vez sobre diversos filtros gammatone logaritmicamente espaçados. A magnitude do resultado final é a soma das magnitudes encontradas, isto é, qualquer possibilidade de interferência destrutiva foi descartada[25]. Neste caso, pode-se observar claramente a maior proeminência do componente frequencial relativo à frequência fundamental.

Na figura 3.23, foi utilizado o mesmo procedimento que gerou a figura 3.22, porém utilizando um sinal modulado em amplitude, especificamente

```
from audiolazy import sHz, white_noise, sinusoid
s, Hz = sHz(22050)
am_sound = white_noise() * (1 + sinusoid(185 * Hz)) # Um objeto Stream
```

que se trata de uma senoide de 185 Hz modulada em amplitude por um ruído branco. Pode-se notar que a frequência da senoide, que é facilmente percebida por humano, também é adequadamente identificada pelo algoritmo.

Devido à trivialidade em se implementar o bloco não-linear da equação 3.118 na AudioLazy, não se fez necessária sua inclusão junto ao pacote.

Este já é um ponto bastante importante do algoritmo de Klapuri: o procedimento realizado já pode ser utilizado para fornecer um estimador da frequência fundamental de um som, inspirado em modelos do aparato auditivo humano, bastando para isso um um sistema de deteção de picos que obtenha o primeiro valor de pico

[24] Esse modelo de trompete tornou necessária a existência de ressonadores variantes no tempo, o que motivou a elaboração do JIT na AudioLazy.

[25] Trata-se de um procedimento análogo ao utilizado para a obtenção da figura 3.18. Klapuri generaliza a ideia para a soma dos módulos elevados a p, embora utilize apenas $p = 1$ (este caso) e $p = 2$.

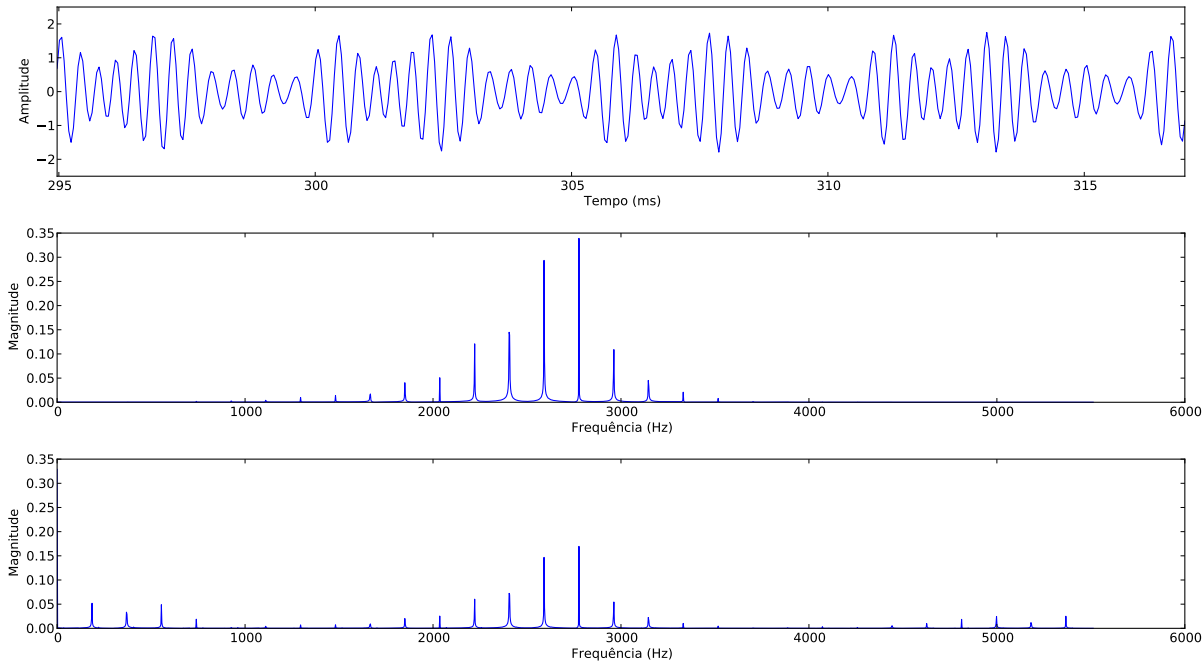


Figura 3.21: *Sinal e espectro de um trompete sintetizado e processado. No primeiro gráfico, encontra-se um segmento de sustentação de um sinal que contém um som similar a uma nota de trompete, sintetizado com frequência fundamental de* 185 *Hz e após a aplicação de um filtro gammatone centrado em* 2700 *Hz. O segundo contém o espectrode um fragmento desse sinal (magnitude da DFT). O terceiro equivale ao segundo, porém aplica o procedimento não-linear descrito antes da avaliação do espectro. Foram usados blocos de* 8192 *amostras, mantendo a taxa de amostragem em* 22050 *amostras por segundo.*

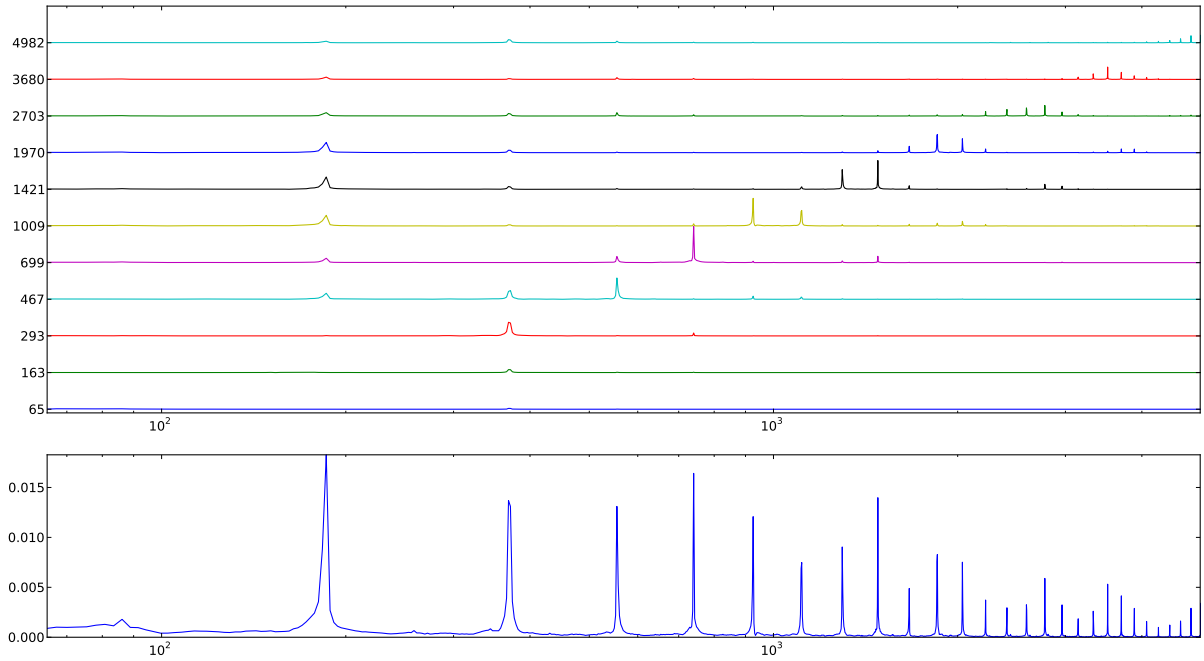


Figura 3.22: *Aplicação de filtros gammatone logaritmicamente espaçados sobre o trompete sintetizado. Som de trompete a* 185 *Hz avaliado em sua região de sustentação, blocos de análise de* 8192 *amostras, taxa de amostragem de* 22050 *amostras por segundo.*

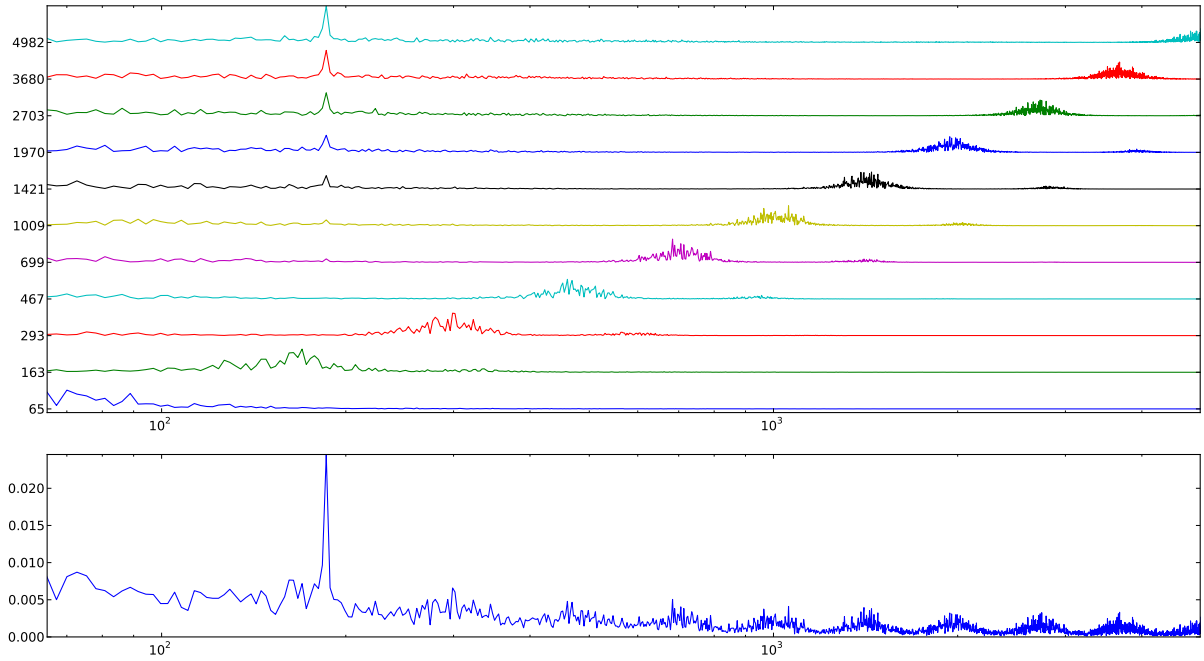


Figura 3.23: *Aplicação de filtros gammatone logaritmicamente espaçados sobre um exemplo de síntese AM. Senoide de* 185 *Hz modulada em amplitude por um ruído branco, blocos de análise de* 8192 *amostras, taxa de amostragem de* 22050 *amostras por segundo.*

proeminente do espectro somado, algo que pode ser facilmente obtido através de um limiar fixo e um sistema de detecção de máximo local.

Análise da periodicidade

Klapuri propõe um modelo de análise de periodicidade baseado em uma medida chamada saliência (*pitch salience*), que trata da correspondência de harmônicos de uma série harmônica montada sobre regiões do espectro discreto da DFT. Isso é mensurado através dos valores máximos de faixas de coeficientes DFT representando frequências as quais incluem esses harmônicos. O algoritmo então diminui tais faixas iterativamente. Klapuri indica que existe uma analogia entre o procedimento realizado por seu algoritmo e aquele envolvendo filtros *comb*, embora seu método seja fortemente baseado na DFT e em processamento em blocos. Há, de fato, uma analogia, no sentido de que a saliência mensurada se comporta como uma avaliação de filtros *comb* com seus passa-faixas largos, em que o algoritmo avalia cada vez filtros com passa-faixas mais estreitos até estes "se encaixarem" ao sinal precisamente o suficiente para que a frequência fundamental da iteração seja considerada presente no sinal.

O algoritmo utiliza um procedimento binário de investigação sobre os potenciais casos de frequência fundamental. A cada passo, há uma divisão das faixas pela metade. As faixas são definidas através do perído máximo e mínimo em número de amostras por ciclo.

Segue abaixo uma implementação direta desta etapa em Python mantendo a estrutura e os comentários do pseudocódigo de Klapuri incluído em seu artigo. Os blocos de entrada já são blocos de coeficientes DFT. Os valores da constante `E1` são atribuídos dependendo do tamanho do bloco (1024 ou 2048), como sugestões de Klapuri. A implementação resultante não depende nem do NumPy e nem da AudioLazy, e o valor resultante é uma tupla (*período em número de ciclos*, *saliência*).

Código-fonte 3.23: *Algoritmo de Klapuri para obtenção da mais proeminente frequência fundamental*

```
1 def find_one_harmonic_series(block, tmin=5., tmax=1000.,
2                               tprec=.0000001, rate=22050.,
3                               E1=20., # Hz, ou 5 Hz
4                               E2=320. # Hz
```

```
                                ):
    K = 2 * (len(block) - 1)
    Q = 0
    tau_low = [tmin]
    tau_up = [tmax]
    qbest = 0
    smax = [0.]
    while tau_up[qbest] - tau_low[qbest] > tprec:
      # Divide o melhor bloco e calcula os novos limites
      Q = Q + 1
      tau_low.append((tau_low[qbest] + tau_up[qbest]) * .5)
      tau_up.append(tau_up[qbest])
      tau_up[qbest] = tau_low[Q]
      smax.append(0)

      # Calcula as novas saliências para as duas metades
      for q in [qbest, Q]:
        w = lambda m: (rate / tau_low[q] + E1) / (m * rate / tau_up[q] + E2)
        tau = (tau_low[q] + tau_up[q]) * .5
        delta_tau = tau_up[q] - tau_low[q]
        Ktaum = lambda m: range(int(round(m * K / (tau + .5 * delta_tau))),
                                int(round(m * K / (tau - .5 * delta_tau)) + 1))
        Uk = lambda m: [block[k-1] for k in Ktaum(m) if 0 <= k < len(block)]
        smax[q] = sum(w(m) * max(Uk(m)) for m in range(1, 21) if len(Uk(m)) > 0)

      # Procura pelo melhor bloco
      qbest = max(xrange(len(smax)), key=lambda idx: smax[idx])

    return (tau_low[qbest] + tau_up[qbest]) * .5, smax[qbest]
```

Esse código não ficou expressivo, e visou apenas ser similar ao pseudocódigo fornecido por Klapuri. Há diversas dificuldades que mereceriam melhor tratamento, por exemplo uma separação entre a parte que trata das frequências em hertz daquela que utiliza coeficientes da DFT, ou algo que evite a necessidade de tantas variáveis. Essa falta de expressividade, junto com a ausência de rotinas de testes automatizados, justifica a ausência da rotina na AudioLazy.

Das diversas premissas envolvidas nesse algoritmo, há algumas que merecem consideração. O m-ésimo harmônico da série harmônica tem peso

$$w(\tau, m) = \frac{f_s/\tau + E1}{m f_s/\tau + E2} \tag{3.119}$$

que modela a envoltória espectral do sinal harmônico como uma função similar à hipérbole $1/m$. A justificativa para o uso dessa equação é o desempenho, um equacionamento simples e eficiente para calcular os pesos diversas vezes por bloco, entretanto nem o bloco nem o sinal tiveram um pré-processamento de normalização da envoltória espectral. A rigor, o peso neste passo pode ser justificado pelo fato de que quanto mais distante uma frequência f está de sua fundamental, maior o número de outras potenciais e próximas frequências fundamentais f/m, $m \in \mathbb{N}^*$.

Existem ainda considerações sobre o uso de coeficientes DFT, sobretudo para a interpolação por truncamento realizado principalmente ao final do processo quando os valores de τ passam a ser fracionários, sobre a limitação da precisão do resultado representável por N coeficientes com a inevitável latência associada, e a ausência de um treinamento ou modelo coletado empiricamente de um sinal para a avaliação de outros sinais, de forma que o algoritmo é fortemente direcionado a atribuições ou definições analíticas realizadas *a priori*.

O algoritmo não continha testes no artigo que permitissem sua fácil avaliação de funcionamento, de forma que ele foi avaliado em conjunto com a etapa seguinte. Embora não seja possível dizer que Klapuri abandonou o modelo de séries harmônicas perfeitas (seção 3.3.2, [Kla97]), é notável a diferença entre as propostas realizadas nas diferentes épocas pelo autor.

Estimação iterativa e cancelamentos

Com a etapa anterior, temos uma maneira de encontrar a mais proeminente frequência fundamental de um bloco. Para a obtenção de outros valores, de alguma maneira a entrada ao algoritmo dado no código-fonte 3.23 precisa ser diferente, dado que este é determinístico. A diferença é realizada através da subtração da frequência fundamental encontrada. Uma versão do espectro original é armazenada, e a cada frequência fundamental encontrada, um dos valores é removido do espectro resultante. Isso é feito diretamente no espectro de magnitude, e o valor removido é proporcional ao valor de seu peso na saliência.

Código-fonte 3.24: *Algoritmo de Klapuri para obtenção multipitch*

```
import numpy as np
from numpy.fft import rfft
from audiolazy import *

def find_all_harmonic_series(block, rate=22050., d=1., gamma=.66,
                             E1=20., E2=320., maxpol=5):
  # Parte 1
  residual = block
  detected = np.zeros(len(residual))

  # Parte 2 (inicialização)
  freqs = []

  # Parte 3 (inicialização)
  K = 2 * (len(block) - 1)
  wnd = window.hamming(K)
  wdft = np.abs(rfft(wnd) / len(wnd))

  # Parte 5 (inicialização)
  all_sum = []
  last_s = 0.

  for unused in xrange(maxpol): # Originalmente seria um laço incondicional

    # Parte 2
    tau, smax = find_one_harmonic_series(residual, rate=rate, E1=E1, E2=E2)
    freq = rate / tau

    # Parte 3
    wsum = np.zeros(len(wdft))
    for m in range(1, 21):
      dft_bin = int(round(m * K / tau))
      if dft_bin > len(block):
        break
      weight = (freq + E1) / (m * freq + E2)
      weight *= residual[dft_bin]
      wdftm = wdft * weight
      cwdftm = np.hstack((wdftm[:dft_bin][::-1], wdftm[:len(wdftm) - dft_bin]))
      wsum += cwdftm
    detected += wsum

    # Parte 4
    residual = block - d * detected
    residual = np.clip(residual, 0, inf)

    # Parte 5 (critério de parada)
    all_sum.append(smax)
```

```
48      new_s = sum(all_sum) / len(all_sum) ** gamma
49      if not (new_s > last_s):
50        break
51      last_s = new_s
52
53      # Parte 2 "Sim, esta frequência também está por lá", se fizer sentido.
54      if freq not in freqs: # Originalmente essa questão não precisava ser feita
55        freqs.append(freq) # Originalmente isto seria feito antes da condição
56                           # de quebra de laço
57
58    return freqs
59
60 # Exemplo de uso utilizando síntese por consulta a tabela
61 rate = 44100
62 s, Hz = sHz(rate)
63 size = 2048
64 table = sin_table.harmonize({1: 1, 2: 5, 3: 3, 4: 2, 5: 1, 6: 1}).normalize()
65
66 data = sum(table(str2freq(note) * Hz) for note in "C3 D3 G3 A4".split()) / 4
67 U = klapuri_blockenizer(data, size=size, rate=rate)
68
69 freqs = find_all_harmonic_series(U.take(), rate=rate)
70 print freqs # Hz
71 print freq2str(freqs)
```

A rotina *klapuri_blockenizer* refere-se aos primeiros dois passos, e devolve um Stream de blocos de coeficientes DFT. Havia dois problemas no algoritmo original do Klapuri que foram corrigidos no código acima. O primeiro deles tem a ver com a finalização forçada de iteração, passando um parâmetro de polifonia máxima, visto que haviam casos nos quais o algoritmo continuava a buscar minuciosamente os valores, e o segundo trata de negligenciar a repetição de valores: muitas vezes a mesma nota era detectada diversas vezes por conta da eliminação daquilo que já havia sido identificado subtrair apenas uma parte proporcional ao peso do sinal, de forma que se tornava possível a mesma frequência aparecer diversas vezes como saída.

Ao final, há um exemplo utilizando a AudioLazy para avaliar o resultado do algoritmo para o acorde "C3 D3 G3 A4"de entrada. O resultado encontra-se abaixo e refere-se às fequências em Hz e em nomes de notas.

```
[440.9820557343213, 66.39404296781065, 44.4122314453594, 197.83630370785707]
['A4+3.86%', 'C2+25.95%', 'F1+29.83%', 'G3+16.16%']
```

Pode-se observar que duas das notas foram obtidas corretamente (A4 e G3), mas uma está uma oitava abaixo do sintetizado (C2), o que de certa forma é esperado dado que ao terceiro harmônico[26] de C2 é precisamente a altura G3, que está incluída no áudio. A nota F1 já é mais difícil de justificar nesse sentido. Alguns outros testes foram feitos mudando apenas o acorde utilizado:

Pode-se observar que o sistema não é muito estável, embora ele raramente erre todas as notas de uma só vez. Um fato é que a subtração considerando a envoltória espectral fixa como decrescente e hiperbólica não se aplica a uma que inicia com o timbre gerado nos exemplos com uma fundamental atenuada[27].

Outros aspectos

Ao otimizar o algoritmo visando viabilizar seu funcionamento em tempo real, Klapuri realizou diversas aproximações de forma a descrever todo o algoritmo como realizado por processamento em bloco utilizando a DFT. Alguns deles sugerem que Klapuri deixou de lado a importância dos filtros gammatone, centralizando sua atenção no processamento não-linear e na saliência no decorrer do artigo.

[26] Contando o fundamental como primeiro.

[27] Referente às amplitudas das componentes da tabela utilizada: $[1, 5, 3, 2, 1, 1]$.

Tabela 3.1: *Avaliação da transcrição polifônica do algoritmo de Klapuri*

Entrada	"G2 D4 D5"
Saída	[296.0815430138562, 589.4714357652784, 294.73571781699707]
	$['D4 + 14.19\%', 'D5 + 6.3\%', 'D4 + 6.3\%']$
Entrada	"C1 C2 C3 C4"
Saída	[263.78173831313245, 131.89086914342167, 66.39404296781065, 44.143066406630936]
	$['C4 + 14.21\%', 'C3 + 14.21\%', 'C2 + 25.95\%', 'F1 + 19.31\%']$
Entrada	"F#4 G4 G#4"
Saída	[44.143066406630936, 48.44970703196278, 371.4477538448085]
	$['F1 + 19.31\%', 'G1 - 19.53\%', 'F\#4 + 6.79\%']$

Alguns valores fornecidos por Klapuri não deixam uma indicação clara sobre o procedimento realizado para a obtenção dos mesmos. Por exemplo, o valor de $\gamma = 0.66$ fornecido é apenas dito como "encontrado empiricamente", embora este seja um parâmetro do algoritmo.

Além disso, o modelo possui algumas premissas fortes. Ao utilizar os pesos para a subtração de uma série harmônica da mistura de coeficientes DFT somados, passa a depender de uma envoltória espectral similar a uma hipérbole para não criar novas séries harmônicas a partir de harmônicos do som. A intensidade dinâmica nas notas do acorde sendo transcrito precisa ser idêntica ou similar para garantir a funcionalidade do algoritmo, e um ajuste dos parâmetros para tratar casos de intensidade desigual mostra que as alturas correspondentes a harmônicos de um som proeminente são descobertas antes de sons menos proeminentes. O artigo inclui a informação de os testes mantivera essa restrição de igualdade de intensidades, mas não que a funcionalidade seria comprometida caso contrário.

3.3.7 Predição linear

A envoltória espectral de um sinal é um contorno suave de seus componentes frequenciais. Formantes são valores de pico da envoltória espectral, e podem ser utilizadas como modelo para o som de vogais [Bar02]. Genericamente, a envoltória espectral contém informações sobre o timbre de um som, e se refere a um aspecto que poderia ser melhorado no algoritmo de Klapuri analisado na seção 3.3.6.

A predição linear ou LPC refere-se a um modelo autoregressivo de um sinal que visa representá-lo por meio de um filtro digital aplicado a um sinal residual, utilizando como critério a minimização desse sinal residual [Bar02, Mak75]. Pode ser visto como um modelo que separa uma fonte de informação (resíduo) de um filtro que processa essa informação, modelo este conhecido como fonte-filtro, o qual pode ser utilizado para representar o aparato fonatório humano.

Existem na AudioLazy estratégias de obtenção dos coeficientes de predição linear de um dado segmento de um sinal, incluindo os métodos da autocorrelação e da covariância. A figura 3.24 contém o resultado do exemplo presente no código-fonte 3.25, no qual a análise é realizada através do modelo da autocorrelação (estratégia padrão) aplicado a um segmento de 512 amostras de um sinal criado através do algoritmo de consulta à tabela (síntese aditiva).

Código-fonte 3.25: *Exemplo de modelo autoregressivo de predição linear*

```
from audiolazy import *

rate = 22050
s, Hz = sHz(rate)
size = 512
table = sin_table.harmonize({1: 1, 2: 5, 3: 3, 4: 2, 6: 9, 8: 1}).normalize()

data = table(str2freq("Bb3") * Hz).take(size) # Nota si bemol da 3a oitava
filt = lpc(data, order=14) # Filtro de análise
G = 1e-2 # Ganho apenas para alinhamento na visualização com a DFT

```

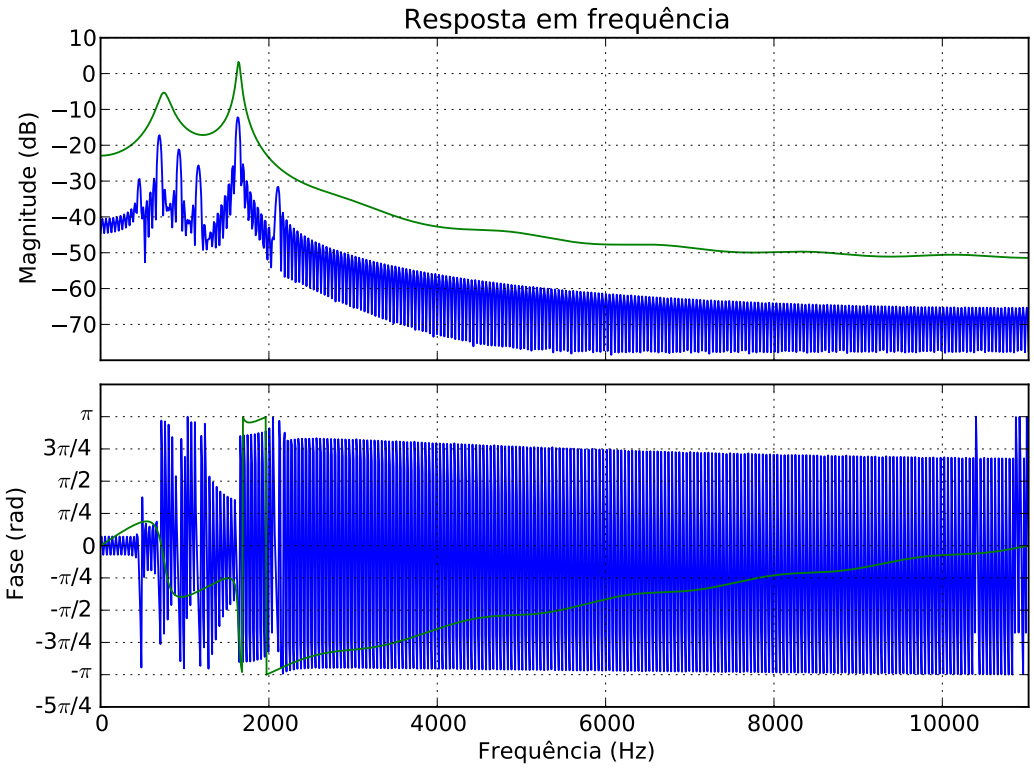


Figura 3.24: *Exemplo de modelo autoregressivo de predição linear. A curva mais suave refere-se ao filtro de síntese de décima quarta ordem do sinal. A outra curva informa a DFT desse mesmo sinal. É possível observar que o modelo autoregressivo está intimamente relacionado com a envoltória espectral do sinal. O exemplo possui dois formantes detectados, caracterizados como valores de máximo local do gráfico de magnitude.*

```
12 # Filtro de síntese
13 (G / filt).plot(blk=data, rate=rate, samples=1024, unwrap=False).show()
```

A predição linear teve importância o suficiente no projeto para possuir um módulo próprio, incluindo rotinas para a obtenção do filtro de análise (dicionário de estratégias *lpc*) e para a obtenção de coeficientes de reflexão ou correlação parcial (função *parcor*, dado que tais coeficientes são tabém nominados como PARCOR) relativos a uma possível implementação do filtro digital em treliça aplicando reversamente o algoritmo de Levinson-Durbin, além de rotinas de detecção de estabilidade de filtros usando os valores das LSFs[28] ou dos coeficientes PARCOR. O módulo é uma camada que utiliza a classe ZFilter de maneira a permitir que os coeficientes de predição sejam obtidos e manipulados diretamente como filtros de análise (ou de síntese), sem a necessidade de uma preocupação com índices ou padronizações de sinais positivo ou negativo aos coeficientes resultantes.

Há no pacote uma função que implementa o algoritmo de Levinson-Durbin utilizado para a resolução de sistemas lineares definidos por matrizes toeplitz[29] tais como os que surgem da utilização da estratégia ou método da autocorrelação para obtenção do filtro de análise. O código-fonte 3.26 traz uma adaptação do código da AudioLazy em que o algoritmo é aplicado, indicando um dos pontos altos da expressividade do pacote, de maneira que os cálculos são feitos através de iterações utilizando os filtros incluindo uma substituição de z por $1/z$.

Código-fonte 3.26: *Algoritmo de Levinson-Durbin*

```
1 def levinson_durbin(acdata, order):
2   # Produto interno entre filtros
3   def inner(a, b): # Cuidado, depende de acdata !!!
```

[28] Disponíveis para filtros LTI quaisquer, incluindo a obtenção do valor das LSFs.

[29] Matrizes em que $a_{(i,j)} = a_{(i+k,j+k)}$, $\forall k \in \mathbb{Z}$, desde que tais posições existam na matriz.

```
    return sum(acdata[abs(i-j)] * ai * bj
               for i, ai in enumerate(a.numlist)
               for j, bj in enumerate(b.numlist)
              )

  A = ZFilter(1)
  for m in range(1, order + 1):
    B = A(1 / z) * z ** -m
    A -= inner(A, z ** -m) / inner(B, B) * B
  return A
```

A vantagem do algoritmo de Levinson-Durbin sobre o uso de métodos gerais de resolução de sistemas lineares consiste no fato de que tal algoritmo se aproveita da estrutura toeplitz da matriz para resolver em tempo $O(N^2)$ o sistema, enquanto sistemas gerais como a decomposição LU[30] necessitam de tempo $O(N^3)$, em que N é o número de linhas e de colunas da matriz.

3.3.8 Cromagrama e decomposição cromática

A equivalência de oitava é um agrupamento de frequências que distam por um número inteiro de oitavas[31]. Nesse contexto, dada uma frequência f, as frequências equivalentes a ela são suas oitavas, isto é, as frequências $2^k f$, $\forall k \in \mathbb{Z}$. Em uma escala logarítmica, essas frequências ficam da forma $\log(2^k f) = \log f + k \log 2$, o que passa a ser linear em k. Classe de oitavas é o nome dado a uma faixa de frequências em conjunto a todas as suas frequências equivalentes. O nome dado ao som gerado por uma classe de oitavas é *Shepard tone* [Moo90, p. 221] [EP07], em homenagem ao trabalho do autor com esse tipo de som [She64], algo que pode ser exemplificado com o código-fonte 3.18.

Como visto na seção 3.3.1, o croma trata da percepção de uma entidade cíclica, a qual pode ser modelada como uma classe de oitavas. Um vetor croma é um vetor do $\mathbb{R}^C$ em que cada dimensão representa uma classe de oitavas. Uma forma de obter o cromagrama é através da utilização da envoltória dinâmica de cada um dos sinais resultantes da decomposição ou análise cromática apresentada nas subseções que seguem.

Diversos autores trabalham com um tipo de representação do sinal de áudio chamada cromagrama. O cromagrama é a representação no tempo de um modelo da percepção do croma, algo que tem como aplicação prática a representação de acordes e de tonalidade, identificadas por sistemas de reconhecimento de padrões [Fuj99, BW01, Pau04, JCEJ08, PMR06, Lee06]. A nomenclatura varia de autor para autor, porém todos possuem o objetivo em comum de, para um dado segmento de áudio, obter um vetor representando o croma, normalmente um vetor do $\mathbb{R}^{12}$ tal que cada dimensão representa uma "intensidade" perceptual de uma das 12 notas da escala cromática temperada utilizada por músicos[32]. Uma generalização imediata é a de utilizar um número C qualquer para o tamanho dos vetores, ao invés de 12, como fez Lee [Lee06].

Esta seção tem como objetivo obter, a partir de um sinal de entrada, C sinais de áudio, cada um relativo a uma diferente dimensão do croma, de forma a aplicar uma função de agregação no sinal para obter o valor do croma. O cromagrama resultante é então utilizado para identificar o ritmo em um exemplo de áudio contendo um coral de Bach sintetizado com o modelo de Karplus-Strong.

O banco de filtros gammatone da decomposição cromática

Deseja-se um banco de filtros para decompor ou analisar um sinal em diversos outros, cada um dos quais referentes a uma classe de equivalência de oitavas. A primeira coisa a ser decidida são as frequências centrais a

[30] Utilizada por *numpy.linalg.solve*, função que soluciona os sistemas lineares nas estratégias *lpc.ncovar* e *lpc.nautocor* da AudioLazy.

[31] A equivalência de oitava possui não apenas uma justificativa presente nas séries harmônicas e na concentração de energia nos primeiros parciais, como também uma justificativa cognitiva, visto que costumamos agrupar perceptualmente intervalos harmônicos de oitava em um único som.

[32] Em geral, os valores são representados digitalmente utilizando números em ponto flutuante. Não é impossível, entretanto, usar uma notação simbólica para representar o conjunto dos números reais, mas nenhum dos autores citados utilizou esse tipo de abordagem.

serem utilizadas, e há uma intercalação entre elementos de cada uma das classes de equivalência. Limitando-se a apenas uma dessas frequências de cada classe, nos preocupamos apenas com a quantidade C de classes, o que pode ser resolvido encontrando um representante de cada classe de oitavas dado por:

$$f_{base} \cdot 2^{(n/C)}, n = 0, 1, 2, ..., C \tag{3.120}$$

em que f_base é um valor de referência, por exemplo os 440 Hz relativos à nota A4. Para cada um desses valores de frequência, os demais pertencentes à sua classe de oitavas pode ser encontrados a partir de consecutivos passos de multiplicação ou divisão por 2, enquanto dentro de uma faixa de valores. Essa funcionalidade faz parte da AudioLazy e encontra-se na função *octaves*, a qual devolve os valores de frequência ordenadamente na faixa de 20 Hz e a 20 kHz, recebendo valores em hertz:

Código-fonte 3.27: *Equivalência de oitava com a função octaves*

```
In [1]: from audiolazy import octaves

In [2]: octaves(440)
Out[2]: [27.5, 55.0, 110.0, 220.0, 440, 880, 1760, 3520, 7040, 14080]
```

A rigor, não se quer que o banco de filtros resultante tenha um canal para cada uma dessas frequências, mas um único canal agrupando todas. Dessa forma, a criação de um único filtro para cada classe de oitava pode ser realizada iterando nessa lista fornecida pelo *octaves* para a criação de um único filtro em paralelo que unifica (soma) os filtros com cada uma dessas frequências centrais. Os filtros utilizados são os filtros gammatone parametrizáveis desenvolvidos neste trabalho. O banco de filtros resultante pode então ser criado conforme consta no código-fonte 3.28. Algo curioso quanto à implementação que esse código-fonte demonstra é o fato de que basta um comando, a menos de preparos quanto à entrada, para criar todo o banco de filtros. Ao final, as respostas em frequência do primeiro filtro do banco são criadas, e estas se encontram nas figuras 3.25, 3.26 e 3.27.

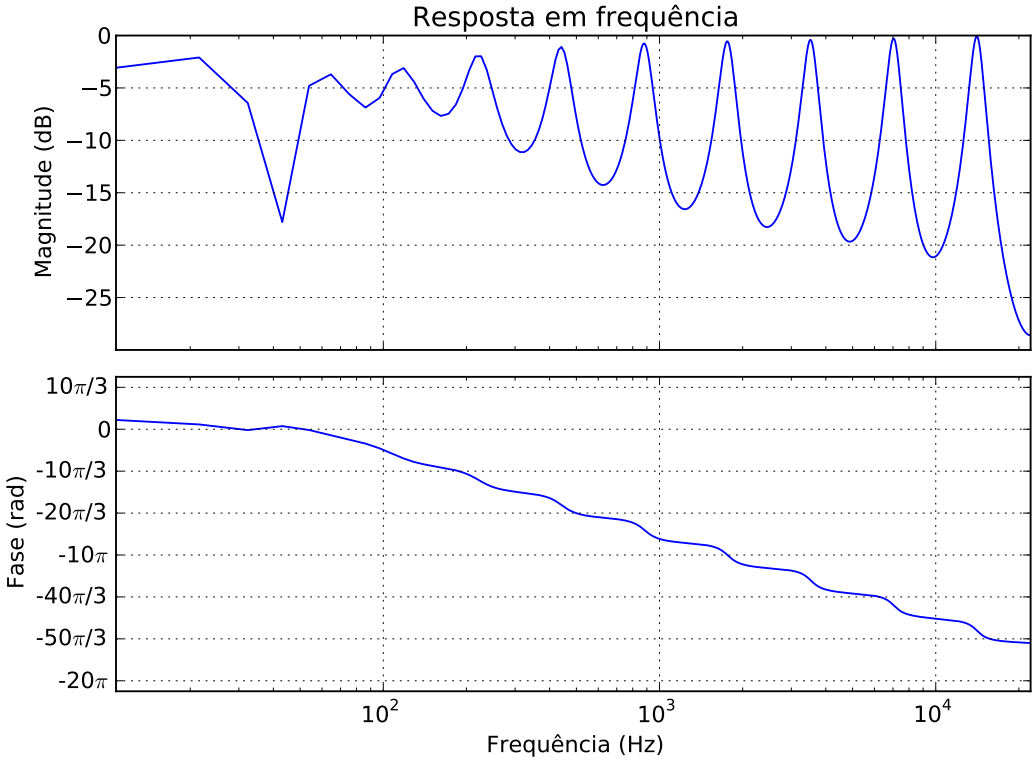


Figura 3.25: *Filtro paralelo composto por filtros gammatone com $\eta = 2$*

Código-fonte 3.28: *Banco de filtros da decomposição cromática*

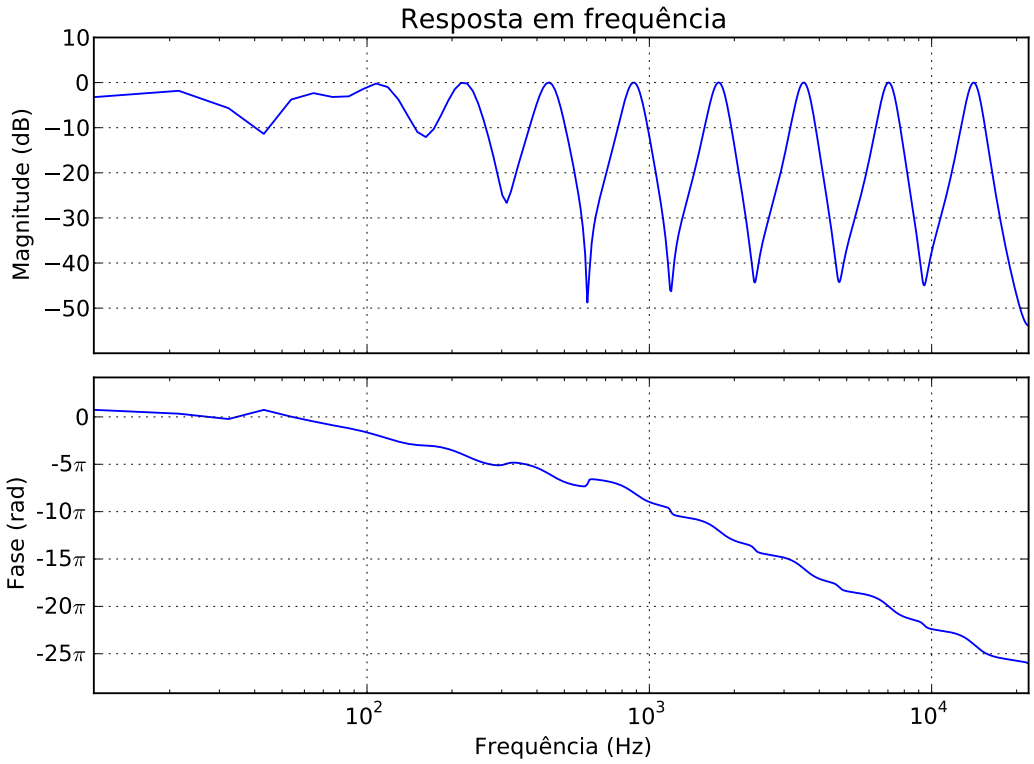


Figura 3.26: *Filtro paralelo composto por filtros gammatone com* $\eta = 4$

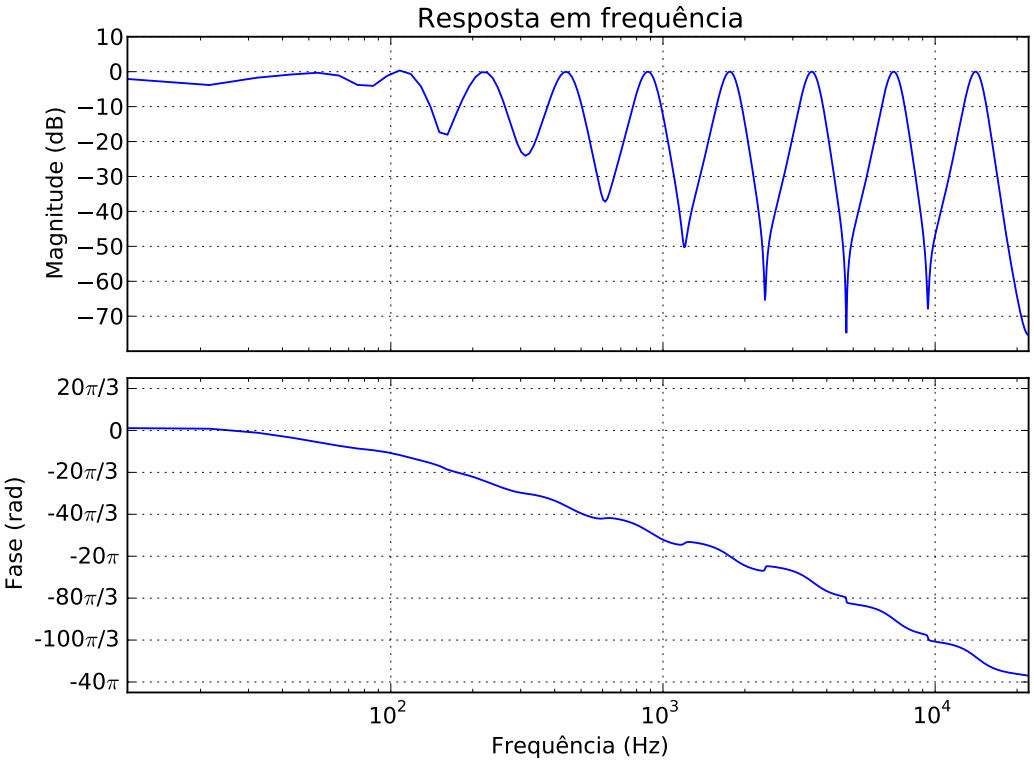


Figura 3.27: *Filtro paralelo composto por filtros gammatone com* $\eta = 7$

```
from audiolazy import *

def cromafb(classes=12, fmin=None, fmax=None, fbase=None, rate=None,
            **kwargs):

  # Validação da entrada
  if fmin is None:
    fmin = 0. if rate is None else 20.
  if fmax is None:
    fmax = pi / 2 if rate is None else min(2e4, rate * .5)
  if fbase is None:
    fbase = pi / 2 if rate is None else FREQ_A4
  if rate is not None:
    Hz = sHz(rate)[1]
    fmin, fmax, fbase = fmin * Hz, fmax * Hz, fbase * Hz
  eta = kwargs["eta"] if "eta" in kwargs else 4

  # Constante pra converter a largura de banda a partir do ERB
  cg = gammatone_erb_constants(eta)[0]

  # Cria o banco de filtros
  return [ParallelFilter(gammatone.sampled(f, cg * erb(f, Hz), **kwargs)
                         for f in octaves(fbase * 2 ** (n / float(classes)),
                                          fmin=fmin, fmax=fmax)
                        ) for n in xrange(classes)]

rate = 44100
cromafb(rate=rate)[0].plot(freq_scale="log", rate=rate)
cromafb(rate=rate, eta=7)[0].plot(freq_scale="log", rate=rate)
cromafb(rate=rate, eta=2)[0].plot(freq_scale="log", rate=rate)
```

Em todo o texto que segue, manteve-se os valores padrões $\eta = 4$, $C = 12$, faixa de frequências de 20 Hz a 20 kHz e frequência de referência igual à constante *FREQ_A4* da AudioLazy, que contém o número 440 em ponto flutuante. Isso fornece um total de 10 filtros gammatone por filtro em paralelo do banco de filtros, totalizando 120 filtros gammatone. Como todo filtro gammatone é implementado na AudioLazy como uma cascata de η ressonadores, essa configuração é realizada com 480 filtros ressonadores.

Da envoltória dinâmica relativa ao croma

Na existência de um sistema de decomposição ou análise cromática já preparado, a filtragem pode ser facilmente aplicada de forma a obtermos todos os canais. Como procedimento inicial de normalização, a envoltória dinâmica RMS do sinal de cada canal é dividida pela envoltória dinâmica RMS do sinal original, com um leve cuidado de saturar a mesma em um valor positivo para evitar divisão por zero, isto é:

```
envelope.rms(resultado_de_um_canal, 5 * Hz) / clip(envelope.rms(song, 5 * Hz),
                                                   low=1e-200, high=None)
```

cujo resultado já é um cromagrama que não exigiu processamento em bloco para sua obtenção.

Utilizando o exemplo de síntese detalhado no código-fonte 5, presente no segundo apêndice deste trabalho, criou-se um exemplo para a representação gráfica do cromagrama, o qual pode ser visto na figura 3.28. A obra de J. S. Bach sintetizada é identificada como BWV416 (choral_file = "bwv416.mxl"), utilizando uma variação do modelo de síntese de Karplus-Strong com uma memória inicial incluindo o primeiro, terceiro e nono harmônicos de cada nota, ruído branco e a frequência de Nyquist, todas essas 5 componentes com o mesmo peso. A taxa de amostragem é de 44100 amostras por segundo, e a partitura foi sintetizada com um

andamento de 90 bpm. Através de uma função afim aplicada em todas as amostras do resultado[33], a escala de amplitudes foi mudada para $[0; 1]$ visando a utilização da escala de tons de cinza na imagem. Apenas os primeiros 16 segundos da síntese foram utilizados.

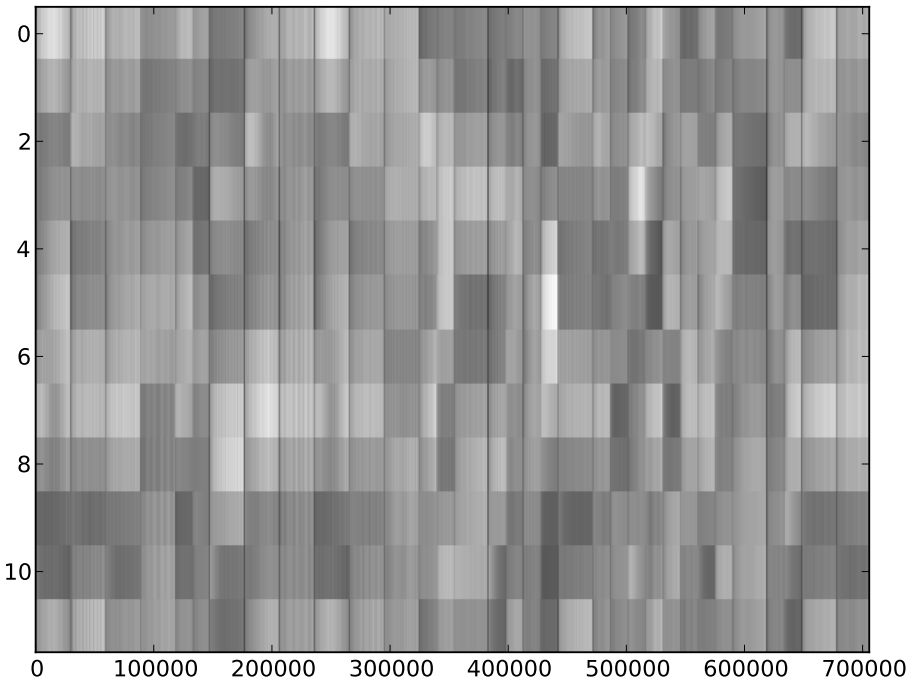


Figura 3.28: *Exemplo de cromagrama utilizando a AudioLazy*

Avaliando o ritmo com derivação discreta suavizada

Pode-se notar claramente por inspeção visual que o ritmo harmônico da obra está presente na figura 3.28. É possível obter os instantes de transição entre acordes, e de início tentou-se utilizar o filtro $1 - z^{-1}$ como modelo da derivada no tempo (diferença entre amostras adjacentes). A rigor, como o sinal utilizado é uma sequência de vetores do NumPy contendo valores do croma em cada instante, o procedimento obtém um vetor diferença ou gradiente. Pode-se observar o ritmo harmônico do segmento através dos pontos de pico na figura 3.29 contendo a norma do vetor gradiente. Em geral, os pontos de transição entre acordes podem ser encontrados como pontos de máximo local acima de um certo limiar. Porém, há pontos de transição que estão muito próximos de um nível de ruído, e deseja-se que a diferenciação encontre com a maior clareza possível quais são os pontos de transição.

Visando solucionar esse problema, foi implementada uma versão alternativa de obtenção do valor do gradiente. Jianwen Luo desenvolveu em MatLab um filtro de diferenciação que considera um maior número de amostras para a obtenção do resultado[34], e incluiu um exemplo, ambos com o código fonte disponível. Uma versão de ambos o algoritmo que cria filtros de diferenciação como o exemplo fornecido por Luo foi criada utilizando a AudioLazy, e encontra-se no código-fonte 3.29 e na figura 3.30.

[33] Subtração do valor mínimo e posterior divisão pelo novo valor máximo.

[34] http://www.mathworks.com/matlabcentral/fileexchange/6170, último acesso dia 2013-02-04.
Luo inclui duas referências bibliográficas pra o filtro em questão:
Usui, S.; Amidror, I., Digital Low-Pass Differentiation for Biological Signal-Processing. IEEE Transactions on Biomedical Engineering 1982, 29, (10), 686-693.
Luo, J. W.; Bai, J.; He, P.; Ying, K., Axial strain calculation using a low-pass digital differentiator in ultrasound elastography. IEEE Transactions on Ultrasonics Ferroelectrics and Frequency Control 2004, 51, (9), 1119-1127.

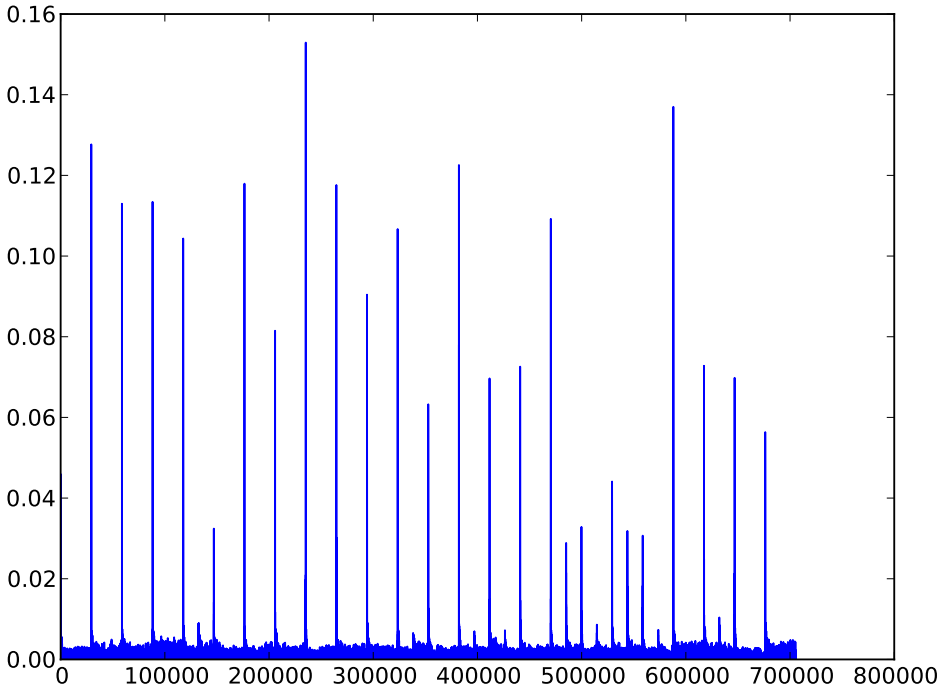


Figura 3.29: *Norma do gradiente do cromagrama a partir da diferença entre vetores adjacentes no tempo*

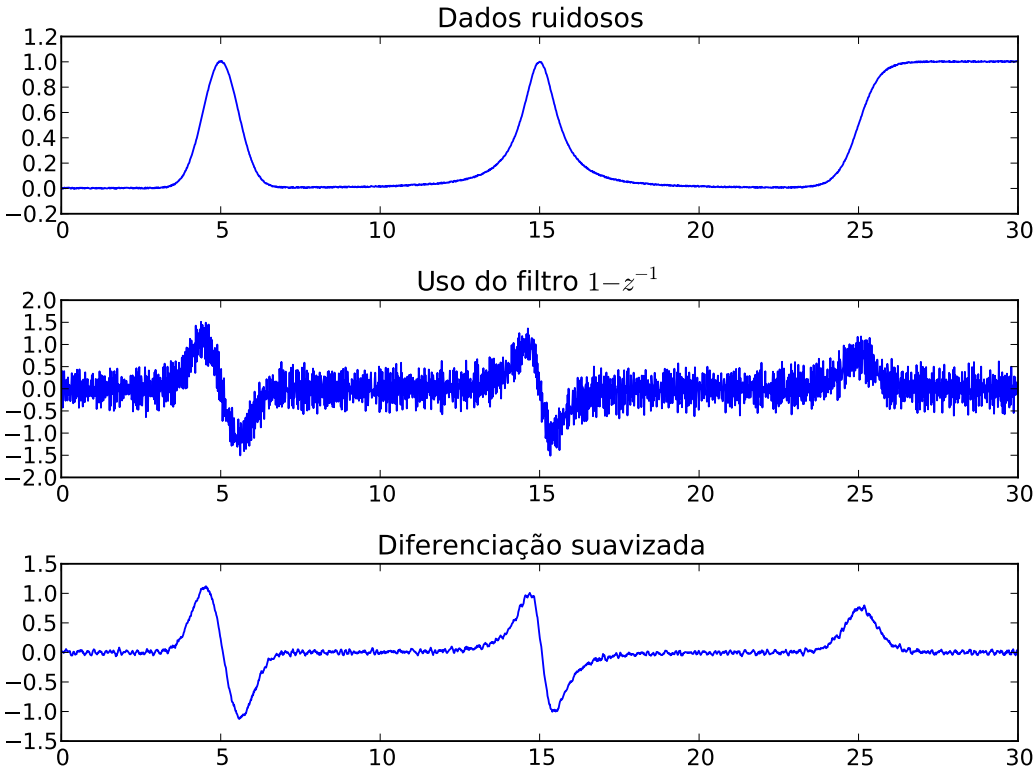


Figura 3.30: *Exemplo de funcionalidade da diferenciação suavizada*

Código-fonte 3.29: *Filtros de diferenciação suavizada incluindo um exemplo*

```
1 from audiolazy import *
```

```
2
3  def diff(size=2):
4    """
5    Função que devolve um filtro para obtenção da diferença suavizada
6    """
7    if size < 2:
8      raise ValueError("Impulse response too small")
9    if not isinstance(size, int):
10     raise ValueError("Impulse response with fractional size in samples")
11
12   return sum(-cmp(i, (size - 1) / 2.) * z ** -i for i in range(size)) \
13          / float((size // 2) * ((size + 1) // 2))
14
15 # Cria o exemplo
16 import numpy as np
17 k = 1. / (2 * ln(2) ** .5)
18 x = np.linspace(0, 30, 2000)
19 y = e ** (-((x - 5.)/(1.3 * k)) ** 2)
20 y += 1. / (1 + 4 * ((x - 15.) / 1.3) ** 2)
21 y += np.ones(len(x)) / (1 + e ** (-3.0 * (x - 25.)))
22 y += list(white_noise(len(x)) * 0.005)
23
24 import pylab
25 pylab.figure()
26 pylab.subplot(3, 1, 1)
27 pylab.plot(x, y)
28 pylab.title(u"Dados ruidosos")
29
30 pylab.subplot(3, 1, 2)
31 tunit = x[1] - x[0]
32 pylab.plot(x, list((1 - z ** -1)(y) / tunit))
33 pylab.title(u"Uso do filtro $1 - z^{-1}$")
34
35 pylab.subplot(3, 1, 3)
36 pylab.plot(x, list(diff(10)(y) / tunit))
37 pylab.title(u"Diferenciação suavizada")
38
39 pylab.tight_layout()
40 pylab.show()
```

Trata-se de um filtro FIR com uma resposta ao impulso de tamanho definido pelo parâmetro passado. De certa forma, o modelo de derivação suavizada remove os componentes mais agudos do sinal quando o tamanho da resposta ao impulo é grande, mas o filtro suavizado coincide com o $1 - z^{-1}$ proposto quando o tamanho é igual a 2.

Utilizando esse modelo de diferenciação para um tamanho em amostras equivalente a 10 milissegundos (por truncamento), o gradiente da figura 3.28 foi reavaliado, e seu novo resultado encontra-se na figura 3.31. Comparando com a figuras 3.29, esse gráfico contém alguns pontos de transição mais claros, sobretudo para amostras depois de 450000, mas também criou um ruído cujos picos dificultam uma avaliação (e.g., região em torno da ammostra de índice 200000).

Dessa forma, através do uso de componentes da AudioLazy foi possível criar um banco de filtros para a decomposição ou análise cromática, um cromagrama, além da detecção do ritmo harmônico através das derivadas, ou gradiente, do sinal. Para isso foi necessário usar um Stream de vetores do NumPy, aplicado a um filtro de diferenciação. É difícil afirmar se houve uma melhora ou degradação no resultado como um todo com a mudança do filtro de diferenciação, mas o notável é que, em ambos os casos, o filtro utilizado obtém os valores dos pontos de transição entre acordes, servindo como modelo para a transcrição de ritmo harmônico.

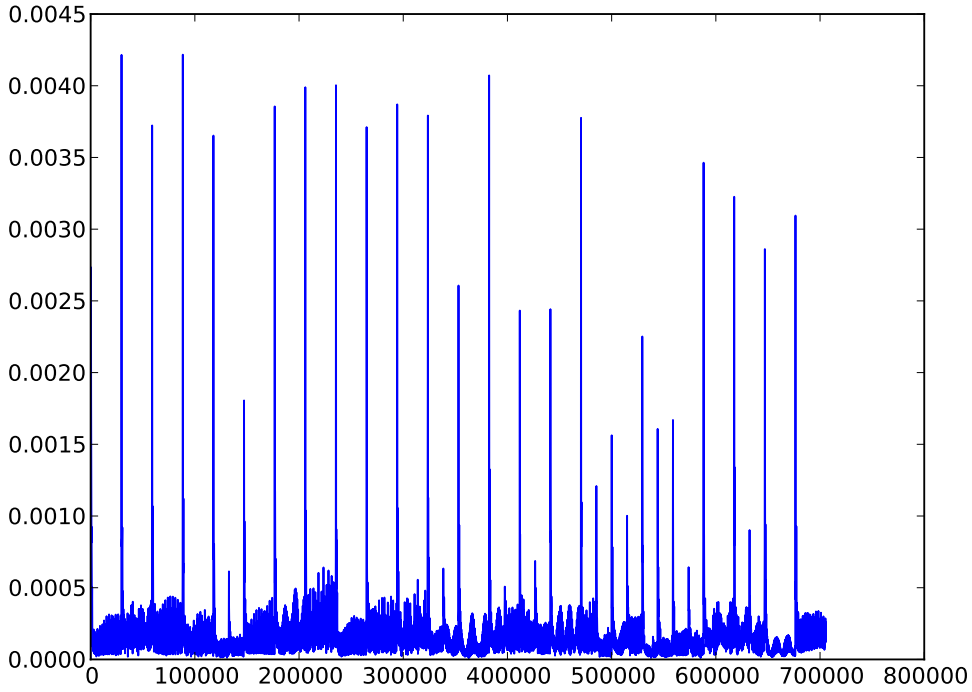


Figura 3.31: *Norma do gradiente do cromagrama suavizado em aproximadamente* 10 *milissegundos*

Uma alternativa ao procedimento realizado poderia ter sido o uso de outras métricas, ou equacionamentos para a norma do vetor gradiente. Embora não tenha sido avaliado por este trabalho, o YAAFE inclui em sua lista de funcionalidades a capacidade de obter a derivada de um sinal, através de um modelo alternativo de derivação do sinal no tempo.

Capítulo 4

Conclusões

Finalizando o trabalho, encontra-se aqui um sumário de tudo o que foi desenvolvido, tanto em termos de pesquisa realizada como em termos de desenvolvimento de *software* organizado e publicado junto à AudioLazy.

Na seção 4.1, são dadas considerações finais sobre os resultados obtidos.Muitas são as possibilidades de continuação do desenvolvimento da AudioLazy, e a seção 4.2 lista algumas que são viáveis e desejáveis para a utilização futura em processamento digital de sinais.

4.1 Considerações finais

Muitos paradigmas de desenvolvimento de *software* surgiram historicamente, cada um com seu ideal a respeito de algum aspecto que se considerava importante. Comandos, expressões, objetos, estruturas e funções: cada uma dessas palavras pode ser usada para fazer referência a um paradigma de desenvolvimento de *software*, mas não são essas palavras, nem os nomes dos paradigmas e nem uma possível intenção de maior "pureza" de algum paradigma em relação aos demais o que realmente importa. Há apenas duas coisas que importam: o resultado e o meio pelo qual se chega ao resultado. Sem algum meio ou caminho que possibilite um resultado, não há como este último existir. Uma linguagem que impõe um distanciamento entre a implementação em *software* de um algoritmo da maneira com a qual sua formalização teórica é realizada, a menos de quando tal formalização é incompleta ou dúbia, estará impondo ao desenvolvedor a responsabilidade sobre uma conversão do algoritmo na implementação, para que este seja possível utilizando somente os recursos fornecidos. Uma forma de minimizar esse distanciamento entre a descrição formal do algoritmo e sua implementação é através da existência de uma pluralidade de recursos, e a possibilidade de uma implementação multi-paradigma é um exemplo que caminha nessa direção.

Há muitas linguagens e muitos recursos disponíveis para desenvolvedores, embora nem sempre estas tenham sido criadas visando possibilitar a utilização em determinados casos de uso específicos. As linguagens de programação são um meio de expressão do desenvolvedor, indispensáveis para a realização de suas tarefas. Aquele que cria uma dessas linguagens tem em mente uma expectativa sobre o que provavelmente um desenvolvedor desejará ao desenvolver, isto é, a elaboração de uma linguagem inclui a hipótese sobre quais serão as intenções do desenvolvedor no futuro, e o que precisará ser expressado.

A expressividade de uma linguagem de programação refere-se ao quanto esta corresponde às expectativas e intenções do desenvolvedor, e o quanto esta facilita a comunicação entre diferentes desenvolvedores. Para uma DSL que visa possibilitar processamento de áudio de maneira expressiva, faz-se necessário reconhecer quais são os casos de uso envolvidos nas diversas possibilidades de aplicação, incluindo casos de síntese e de análise, além do processamento propriamente dito. Avaliando o que o Python e outras linguagens ofereciam nesse sentido, foi possível elaborar uma DSL restrita à sintaxe do Python que mantivesse a expressividade com relação a uma gama de estruturas básicas, tais como filtros expressos através de suas equações de sistema (transformada Z) mesmo quando variantes no tempo, polinômios e somas de potências representados com base em pares {*expoente*: *coeficiente*}, sinais representados como fluxos de informação, alturas de notas representadas como nomes em *strings*, operadores aplicáveis às estruturas de forma a replicar o uso dos mesmos em suas respectivas teorias, etc., sempre mantendo a possibilidade de conversão entre estruturas, de forma a possibilitar ao desenvolvedor a ênfase nos aspectos relacionados ao seu problema em específico, em

detrimento a uma possível ênfase em detalhes de implementação tais como a manipulação de índices para a segmentação do áudio ou a ordem dos argumentos em uma chamada de função.

Embora não exista um registro completo de acessos e buscas ao pacote desenvolvido, principalmente por conta da ausência de um registro de acessos diretos ao repositório de desenvolvimento do pacote, é seguro dizer que a demanda por esse tipo de *software* existe e, através do registro existente no PyPI, que nos informa sobre a realização de centenas de *downloads* do pacote em cada uma de suas versões (vide tabela 1.1), tem-se um indicador quanto a essa demanda[1].

Segundo Hudak [Hud89], dois elementos importantes da programação funcional levados para a programação em outros paradigmas são as funções de ordem superior e a avaliação tardia. Estes são vistos como elementos fundamentais à modularidade do *software*, dado que o importante para a existência de modularidade não é a pluralidade de módulos, mas sim a possibilidade de conexão entre os módulos. Com poucas estruturas (e.g., objeto *z*, classe *Stream*), torna-se possível criar muitas outras estruturas e conexões compatíveis entre si. Filtros são objetos chamáveis, que se assemelham fundamentalmente a funções, e seu tratamento como objetos que interagem com operadores bem definidos é essencial à expressividade possibilitada pelo pacote. Sem a metaprogramação, também seria inviável a realização de tantas tarefas de automação: os blocos conectados são bastante abstratos, incluindo criação massiva de operadores em classes, sumários automáticos, conversões de formatos de documentação, entre outros. Embora seja inegável a fundamental importância da avaliação tardia e das funções de ordem superior no resultado obtido com a AudioLazy, toda a generalidade envolvida no pacote não veio apenas dos dois elementos que Hudak destacou, mas também da programação reflexiva, metaclasses, orientação a objetos e outras características presentes no Python.

Nem todo *software* ou pacote associado a uma linguagem é criado visando sua funcionalidade em tempo real, e normalmente aquilo que é projetado sem essa restrição como requisito costuma se limitar a aplicações de prototipação e simulação. O inverso também pode ocorrer: ferramentas que encapsulam os dados intermediários do processamento de maneira a impossibilitar ou dificultar análises. Entretanto, mesmo pacotes com objetivos tão distintos como o SymPy e a AudioLazy puderam ser, até certo ponto, integrados. Isso possibilita o processamento de áudio em amostras com precisão arbitrária, sem perder a expressividade no sentido de continuar a utilizar o sinal e os símbolos como abstrações, diretamente. Na integração com o NumPy, o mesmo ocorre com sinais multidimensionais, utilizando vetores e matrizes. O elemento essencial da linguagem que possibilitou isso é uma filosofia baseada em interfaces ao invés de herança, e uma tipagem dinâmica em detrimento do reconhecimento estático e preemptivo de tipos e classes. Qualquer objeto pode fazer parte de um *Stream*, e sua interface, sobretudo seus operadores, passam a ser aqueles disponibilizados pela classe Stream, embora a disponibilidade dos recursos seja avaliada somente quando há uma requisição (avaliação tardia). A heterogeneidade é uma consequência gratuita dessa funcionalidade baseada em interfaces, fundamento da classe *Stream*.

Mas vale dizer que esta é "gratuita mas nem tanto". Pode-se notar através da avaliação feita com o código-fonte 7 presente no segundo apêndice que, comparado com uma implementação alternativa de uma mesma rotina de síntese usando o NumPy, a AudioLazy é lenta para operações em lote, tanto no CPython (implementação padrão do Python) como no PyPy (implementação do Python em RPython, com JIT), e que o PyPy é mais rápido que o CPython para essa aplicação:

Tabela 4.1: *Avaliação de desempenho para processamento em lote. Valores médios de 30 testes consecutivos, sem incluir o tempo inicial enquanto os pacotes são importados. O teste consiste em calcular numericamente uma integral de* 5 *segundos de uma síntese FM com uma envoltória ADSR com a taxa de amostragem de* 44100 *amostras por segundo. Os valores estão em milissegundos.*

	CPython	**PyPy**	**PyPy (segunda medição)**
AudioLazy	271.490502357	57.9847653707	53.2383282979
NumPy	19.446738561	22.7807998657	21.0646947225

Entretanto, ainda há muitas otimizações possíveis para a AudioLazy. Todas as rotinas estão escritas em Python, e o objetivo foi o de maximizar a expressividade, não o desempenho. O desempenho necessário é o de processamento em tempo real, e os componentes mais otimizados são os filtros por conta do JIT, os quais

[1] Os dados fornecidos pelo PyPI não são suficientes para mensurar a demanda, mas são úteis para avaliar sua existência.

não foram utilizados nesse teste. Seja como for, é importante notar que o PyPy conseguiu reduzir para cerca de 1/5 o tempo de processamento necessário com o CPython. Mesmo com essa desvantagem com relação às durações, a maneira como os dados são organizados no NumPy inviabiliza o seu uso para processamento em tempo real, diferentemente do que ocorre com a AudioLazy.

Um ponto de grande preocupação durante o desenvolvimento deste trabalho foi o da fragilidade da representação numérica e ponto flutuante e suas consequências, em outras palavras, aspectos que envolvem a aplicação prática. Números representados em ponto flutuante segundo o padrão IEEE 754, mesmo quando possuem 64 bits (e portanto 53 bits de mantissa), não são suficientes para lidar com casos de subtrações aparentemente simples, mas que eliminam ou mascaram parte importante da informação existente. Tal situação pode ocorrer em somas e subtrações internas à implementação de filtros, e traz uma preocupação especial quando se trata de valores utilizados para representar uma retroalimentação, normalmente descrita através dos polos dos filtros, situação na qual o ruído pode inutilizar uma implementação. Dessa forma, a pluralidade de estruturas para implementações de filtros se faz uma necessidade prática.

Ainda sobre o problema da representação numérica, existem atualmente alternativas para possibilitar a representação de números de precisão arbitrária, tal como ocorre com os números inteiros nativamente em Python, mas nem sempre essas alternativas são suficientes para cobrir todos os casos de uso, sobretudo quando existem requisitos quanto ao desempenho. Além disso, comparar por aproximação é uma necessidade, não apenas uma consequência da representação em ponto flutuante: números coletados por meio de sensores, tais como sequências obtidas a partir de transdutores seguidos por um conversor analógico-digital, estão sujeitas à presença de ruído em todo o caminho até a obtenção de sua representação digital. Mesmo que os valores sejam precisos, a tolerância ao ruído é necessária. Isso é o que faz os teoremas de Klapuri quanto às propriedades de frequências em séries harmônicas necessitarem de uma adaptação prática, por exemplo através de filtros *comb*.

Não é possível saber o que teria ocorrido com o desenvolvimento da AudioLazy caso este tivesse sido realizado sem a intenção de manter parte significativa do código coberto por testes. Muito de sua funcionalidade básica está coberto por diversos testes, incluindo testes utilizando o NumPy e o SciPy como oráculos. A influência dos testes no desenvolvimento não é algo simples de mensurar objetivamente, e dificilmente estará livre de um viés devido ao que representa tal procedimento para quem realiza a análise. É possível dizer que os testes ajudam a conferir se o código está correto, e permitem que refatorações sejam feitas sem a necessidade de grandes preocupações, dado que, enquanto todos os testes passarem com sucesso, não há falhas conhecidas registradas. Embora não seja impossível, é de se esperar que um pacote com mais de 5400 testes automatizados tenha uma confiabilidade superior a um pacote sem nenhum teste automatizado. Se confiabilidade, expressividade, modularidade, integração com diferentes pacotes, abstração em diferentes níveis e em simultâneo, documentação, automação e outros elementos presentes na AudioLazy implicam em qualidade ou não, trata-se de uma interpretação, ou de uma afirmação enfática de uma norma sobre qualidade de *software*, mas pelo menos é possível dizer que se trata de uma consideração verossímil.

Muitos foram os exemplos de MIR incluídos no presente trabalho, de maneira a demonstrar o potencial de uso do pacote para esse tipo de tarefa. É esperado que diversas dessas funcionalidades sejam inseridas no pacote, organizadas de acordo com dicionários de estratégias que permitam um agrupamento daquilo que possui um mesmo objetivo em comum, por exemplo rotinas para a obtenção da frequência fundamental de um som. Nem todos os modelos presentes no YAAFE e nem todos os modelos listados por Peeters[2] [Pee04] foram implementados com a AudioLazy até o atual momento, mas a maior parte ou faz parte do que está na AudioLazy (e.g. coeficientes LPC, LSFs, taxa de cruzamento por zeros) ou pode ser implementado com facilidade (e.g. centróide espectral).

Filtros gammatone foram generalizados para permitir seu uso com base em parâmetros. Essa generalização permitiu, por exemplo, a verificação das respostas em frequência de diferentes filtros durante o projeto de um cromagrama. Porém a implementação desse algoritmo para obtenção os filtros gammatone teve como requisito a obtenção de derivadas de filtros com relação a z. Este trabalho criou um modelo geral de como tais derivadas podem ser computadas, incluindo um passo de multiplicação por um filtro FIR a cada passo de derivação. O uso dessa rotina de derivação permite que os coeficientes do filtro resultante sejam obtidos

[2] Peeters centralizou uma grande lista de possíveis *features* de áudio em um único documento, contendo modelos utilizados nas áreas de psicoacústica, reconhecimento de voz e classificação automática de instrumentos musicais.

sem a necessidade de preocupação por parte do desenvolvedor, bastando este conhecer propriedades básicas da transformada Z para saber quando a derivação pode ou deve ser utilizada.

Deve-se tomar cuidado com a aproximação realizada ao utilizar em um filtro digital valores que surgiram de uma fórmula no domínio da transformada de Laplace, dado que tanto o casamento de zeros e polos como a amostragem da resposta ao impulso não garantem que os valores no filtro digital corresponderão ao esperado. Entretanto, a praticidade pode falar mais alto que a pureza nesses casos, visto que não são simples os cálculos envolvendo, por exemplo, a obtenção de valores exatos de largura de banda de ressonadores. Da mesma forma, deve-se tomar cuidado com a utilização da DFT, particularmente quando se trata de interpolar seus valores para obtenção de um número fracionário de ciclos.

Além da sintaxe de como o processamento éisso é feito, por vezes envolvendo um único comando para realizar tarefas complexas tais como a criação de um banco de filtros personalizado, há também um sistema que automatiza as legendas, títulos, localizadores e formatadores de eixos em gráficos. Utilizando o MatPlotLib, a AudioLazy cria gráficos que incluem múltiplos fracionários de π nos eixos, diagrama de polos e zeros incluindo a multiplicidade destes, entre outros recursos importantes que não puderam ser melhor detalhados no decorrer deste texto, mas que fazem parte daquilo que traz expressividade e facilidade para o desenvolvedor que utilize a AudioLazy em seu projeto.

Além das considerações técnicas, tem-se agora um pacote de processamento de áudio em tempo real livre (GPLv3), gratuito e publicamente disponível. Espera-se que o pacote de *software* que este trabalho trouxe como principal contribuição seja cada vez mais utilizado, de forma a possibilitar que pesquisas diferentes possam ser realizadas pela comunidade envolvida em processamento de sinais de áudio.

4.2 Continuação para a AudioLazy

Há muitos recursos que podem ser inseridos na versão atual da AudioLazy. O pacote certamente será continuado, dado o interesse que aparentemente a comunidade demonstrou, além da possibilidade de solucionar problemas difíceis em poucas e expressivas linhas de código. Muito do código atual da AudioLazy surgiu de experimentos realizados fora do pacote mas que, depois de devidamente organizados, passaram a incorporar o pacote. Talvez nem todos os itens abaixo sejam implementados, e podem existir itens que não sejam desejáveis por criar ambiguidades ou deteriorar as capacidades atuais do pacote, mas tal informação, atualmente, é desconhecida. Em termos de implementação de novos recursos, a AudioLazy pode ser ampliada para possuir no futuro:

1. **Análise**
   (a) Alternativas de sintaxe amigáveis para a DFT, mas otimizadas para $O(n \log n)$ com a FFT (algoritmo de Cooley-Tukey), possivelmente utilizando o NumPy internamente. Há possibilidades de implementação descritas em [OSB99, capítulo 9];
   (b) Heurísticas que auxiliem ou possibilitem a transcrição musical;
   (c) Possibilidades de realizar outras transformadas sobre blocos, incluindo wavelets, transformada discreta de Hilbert, etc..
2. **Modelagem da audição humana**
   (a) Outros modelos de ERBs dadas frequências centrais;
   (b) Conversão expressiva entre a largura de banda x-dB e ERB;
   (c) Ganho paramétrico do gammatone de Slaney e da implementação a partir da amostragem da resposta ao impulso (permite que seja variante no tempo, como a implementação na AudioLazy do modelo de Klapuri);
   (d) Modelo de Lyon das células ciliares da cóclea (órgãos de Corti), como presente na *toolbox* de Slaney [Sla98];
   (e) Modelo de Seneff, *ibidem*.

3. **Filtros**
   (a) Expoentes de Z variantes no tempo. Isto potencialmente inclui uma dificuldade com relação ao tamanho da memória das filas circulares que armazenam as saídas e as entradas na primeira forma direta, além de complicações com relação à resolução de coeficientes fracionários, entretanto é possível manter a sintaxe "*z ** a*", na qual o expoente "*a*" é um objeto Stream, mas a especificação do que seria o "esquecimento" precisa ficar clara;
   (b) Outras técnicas para lidar com expoentes fracionários em Z, tais como a interpolação por funções sinc ou splines [Zöl11, p.73-75];
   (c) Filtros com notação em código-fonte baseada na forma como suas equações são descritas no domínio do tempo (e.g. *y[n] = x[n] + 2 * x[n-3]*);
   (d) Filtros não-lineares baseados em equações no tempo, como consequência do item anterior;
   (e) Decomposição do filtro em um ParallelFilter, por frações parciais;
   (f) Decomposição do filtro em um CascadeFilter por ganho, zeros e polos. Isto pode ser muito útil para evitar constrangimentos com relação a possíveis problemas associados à precisão numérica na aplicação de filtros lineares (tópico discutido na seção 3.2.5);
   (g) Possibilidade de visualização do código Python gerado pelo JIT;
   (h) Otimização através de um JIT para agrupamentos de filtros tais como filtros em cascata e paralelo, a fim de evitar minimizar o número de chamadas de função;
   (i) Implementação do filtro na segunda forma direta, também com um JIT;
   (j) Implementação do filtro em treliça (*LatticeFilter*);
   (k) Cálculo e gráficos do atraso de grupo;
   (l) Integração com o SymPy para que os coeficientes dos filtros lineares também possam ser símbolos;
   (m) Ressonadores com valores exatos de largura de banda.

4. **Síntese e processamento para fins musicais imediatos**
   (a) Generalizar o modelo de síntese do trompete implementado, como visto na seção 3.3.6, e inseri-lo junto à AudioLazy, a fim de possibilitar a utilização de conjuntos de dados (ou *corpora*) de áudio fornecidos e modelando assim timbres a partir de exemplos;
   (b) *Wah*, aproveitando os filtros variantes no tempo [Zöl11, p.67-68];
   (c) *Phaser* [Zöl11, p.68-69], *flanger*, *chorus* e eco [Zöl11, p.76-78];
   (d) Otimização de blocos fundamentais para a síntese;
   (e) Uso de amplitudes complexas no algoritmo de síntese por consulta à tabela a fim de permitir o controle dos valores das fases de cada componente.

5. **Testes**
   (a) Chegar aos 100% de cobertura;
   (b) Usar *mock* para testar partes dependentes do MatPlotLib e do PyAudio;
   (c) Usar outras maneiras de se calcular a cobertura de código, por exemplo incluindo não apenas toda possibilidade de ramificação em cada linha de código, mas também todo caminho possível em ramificações adjacentes.

6. **Outros**
   (a) Compatibilidade com o Python 3.3;
   (b) Integração com o PureData;
   (c) Mais exemplos de casos de uso;

(d) Tutorial passo-a-passo como parte da documentação;

(e) Possibilidade de uso junto às padronizações LADSPA, DSSI e LV2;

(f) Funcionamento como *host* de *plugins* Vamp;

(g) Suporte à entrada e saída MIDI (protocolo);

(h) Maior suporte a formatos de arquivo de entrada e de saída;

(i) Melhoramentos e inserção de uma interface facilitadora para o *overlap-add* aplicado a objetos *Stream* contendo blocos;

(j) Suporte nativo à tradução de legendas e títulos em gráficos e personalização de saídas sem a necessidade de *monkey patch*, incluindo um banco de dados de traduções prontas.

Referências Bibliográficas

[Bar02] Lucas de Melo Jorge Barbosa. Algoritmos de busca em decodificadores ACELP. Dissertação de Mestrado, Unicamp, Novembro 2002.

[BDG+09] Janet M. Baker, Li Deng, James Glass, Sanjeev Khudanpur, Chin-Hui Lee, Nelson Morgan e Douglas O'Shaugnessy. Research developments and directions in speech recognition and understanding, part 1. *IEEE Signal Processing Magazine*, 26(3):75–80, 2009.

[Bre93] Albert S. Bregman. *Auditory scene analysis: Hearing in complex environments*, chapter 2, páginas 10–36. Thinking in Sound: The Cognitive Psychology of Human Audition. Oxford University Press, USA, 1ª edição, Maio 1993.

[Bre94] Albert S. Bregman. *Auditory scene analysis: The perceptual organization of sound*. The MIT Press, 1994.

[BW01] Mark A. Bartsch e Gregory H. Wakefield. To catch a chorus: Using chroma-based representations for audio thumbnailing. *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, páginas 15–19, 2001.

[DCM00] Myriam Desainte-Catherine e Sylvain Marchand. High Precision Fourier Analysis of Sounds using Signal Derivatives. *Journal of the Audio Engineering Society*, 48(7-8):654–667, 2000.

[EP07] Daniel P. W. Ellis e Graham E. Poliner. Identifying 'Cover Songs' with Chroma Features and Dynamic Programming Beat Tracking. Em *IEEE International Conference on Acoustics, Speech and Signal Processing, 2007. ICASSP 2007*, volume 4, 2007.

[Fuj99] Takuya Fujishima. Realtime chord recognition of musical sound: A system using common lisp music. Em *Proceedings of the International Computer Music Conference*, páginas 464–467, 1999.

[GM90] B. R. Glasberg e B. C. J. Moore. Derivation of auditory filter shapes from notched-noise data. *Hearing Research*, 47(1-2):103–138, 1990.

[HB95] Andrew Horner e James Beauchamp. Synthesis of Trumpet Tones Using a Wavetable and a Dynamic Filter. *J. Audio Eng. Soc.*, 43(10), Outubro 1995.

[HPNSR88] John Holdsworth, Roy Patterson, Ian Nimmo-Smith e Peter Rice. Implementing a GammaTone Filter Bank. Relatório técnico, Cambridge Electronic Design, MRC Applied Psychology Unit, Fevereiro 1988.

[Hud89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[Hug89] John Hughes. Why Functional Programming Matters. *Comput. J.*, 32(2):98–107, 1989.

[JCEJ08] Jesper H. Jensen, Mads G. Christensen, Daniel P. W. Ellis e Soren H. Jensen. A tempo-insensitive distance measure for cover song identification based on chroma features. Em *IEEE International Conference on Acoustics, Speech and Signal Processing, 2008. ICASSP 2008*, páginas 2209–2212, 2008.

[Kla97] Anssi Klapuri. Automatic Transcription of Music. Dissertação de Mestrado, Tampere University of Technology, Finland, 1997.

[Kla04] Anssi Klapuri. *Signal Processing Methods for the Automatic Transcription of Music*. Tese de Doutorado, Tampere University of Technology, Finland, 2004.

[Kla08] Anssi Klapuri. Multipitch Analysis of Polyphonic Music and Speech Signals Using an Auditory Model. *IEEE Transactions on Audio, Speech & Language Processing*, 16(2):255–266, 2008.

[KV08] Anssi Klapuri e Tuomas Virtanen. *Automatic Music Transcription*, chapter 20, páginas 277–303. Handbook of Signal Processing in Acoustics. Springer, 2008.

[Lee06] Kyogu Lee. Automatic chord recognition from audio using enhanced pitch class profile. Em *Proceedings of the International Computer Music Conference (ICMC), New Orleans, USA*, 2006.

[Mak75] J. Makhoul. Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63(4):561–580, Abril 1975.

[Mar99] Jr. L. Marple. Computing the discrete-time "analytic" signal via FFT. *Trans. Sig. Proc.*, 47(9):2600–2603, Setembro 1999.

[MG83] B. C. J. Moore e B. R. Glasberg. Suggested formulae for calculating auditory filter bandwidths and excitation patterns. *J. Acoust. Soc. Am.*, 74:750–753, 1983.

[Moo90] Richard F. Moore. *Elements of computer music*. Prentice-Hall PTR, 1990.

[MW05] Philip McLeod e Geoff Wyvill. A Smarter Way to Find Pitch. *Proc. International Computer Music Conference*, páginas 138–141, Setembro 2005.

[OSB99] Alan V. Oppenheim, Ronald W. Schafer e John R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, segunda edição edição, February 1999.

[Par89] Richard Parncutt. *Harmony: A Psychoacoustical Approach*. Springer Verlag, 1989.

[Pau04] Steffen Pauws. Musical key extraction from audio. Em *Proceedings of 5th International Conference on Music Information Retrieval*, 2004.

[Pee04] Geoffroy Peeters. A large set of audio features for sound description (similarity and classification) in the CUIDADO project. Technical report, Institut de Recherche et Coordination Acoustique/-Musique, 2004.

[PMR06] Steve Van De Par, Martin McKinney e André Redert. Musical key extraction from audio using profile training. Em *Proceedings of the 7th International Conference on Music Information Retrieval*, 2006.

[PO05] Christopher J. Plack e Andew J. Oxenham. *Overview: the present and future of pitch*, chapter 1, páginas 1–6. Pitch: Neural Coding and Perception. Springer, 2005.

[Rod09] Felipe Viegas Rodrigues. Fisiologia da música: uma abordagem comparativa. *Revista da Biologia*, 2, Março 2009. Laboratório de Neurociência e Comportamento IB-USP.

[Roe98] Juan G. Roederer. *Introdução à física e psicofísica da música*. Edusp, 1ª edição, 1998.

[She64] Roger N. Shepard. Circularity in judgements of relative pitch. *Acoustical Society of America*, 36(12):2346–2353, Julho 1964.

[Sla93] Malcolm Slaney. An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank. Relatório técnico, Apple Computer, 1993.

[Sla98] Malcolm Slaney. Auditory Toolbox Version 2. Relatório técnico, 1998.

[Sri06] Soudararajan Srinivasan. *Integrating computational auditory scene analysis and automatic speech recognition*. Tese de Doutorado, The Ohio State University, 2006.

[SS00] Adel S. Sedra e Kenneth C. Smith. *Microeletrônica*. Maktron Books, 4ª edição, 2000. (Tradução realizada em 2004 a partir da edição em inglês).

[Ste96] Ken Steiglitz. *A digital signal processing primer - with applications to digital audio and computer music.* Addison-Wesley, 1996.

[Wit12] Ludwig Joseff Johann Wittgenstein. *Investigações Filosóficas*. Petrópolis, RJ: Vozes; Bragança Paulista, SP: Editora Universitária São Francisco, sétima edição edição, 2012.

[Zöl11] Udo Zölzer, editor. *DAFX: Digital Audio Effects*. John Wiley & Sons, segunda edição edição, 2011.

Índice Remissivo

A transcrição realizada por seres humanos

Este apêndice visa apenas descrever o processo de transcrição realizado por seres humanos e o significado pragmático associado aos símbolos obtidos em tal processo. A transcrição foi uma das motivações para o desenvolvimento da AudioLazy, motivo pelo qual esta reflexão se fez necessária, entretanto o assunto tratado neste apêndice apenas ilustra uma fundamentação filosófica para possíveis trabalhos futuros.

A transcrição musical é uma tarefa realizada por músicos visando representar através de símbolos uma música que foi escutada, normalmente na forma de partituras, tablaturas ou cifras. Diferentes pessoas podem realizar diferentes transcrições de uma mesma gravação, por conta de diferentes ênfases, precisões ou objetivos, e chegar a diferentes resultados. Do ponto de vista computacional, o fato do áudio conter uma música é apenas uma contingência, que não necessariamente representam uma música para outro ouvinte, e a informação transcrita pode ser exposta de muitas formas diferentes. Assim como a cifragem da harmonia em geral deixa de lado a rítmica da música em sua notação, pode-se realizar o inverso registrando a rítmica mas não a harmonia. Se o objetivo é a obtenção de informações de voz, talvez a informação mais importante seja identificar se o que foi pronunciado em um dado segmento de áudio pode ou não ser uma vogal.

Um caso talvez inconsciente de transcrição ocorre quando uma pessoa tenta reproduzir com a própria voz ou corpo o que está escutando. Essa pessoa precisou identificar aquilo que ela está ouvindo, além de descobrir como produzir um efeito pelo menos similar ao escutado. Uma música pode ser feita com segmentos de sons de gritos, motores, colisões, etc., sem uma contextualização melódica óbvia, e a identificação do processo realizado sobre os segmentos originais para a obtenção do resultado também é uma transcrição. Um dos objetivos dessa ilustração é o de evitar julgamentos de valor estético sobre o áudio sendo transcrito e pressupostos acerca do que são informações "importantes" do áudio.

O problema da transcrição está intimamente relacionado com um outro problema chamado ASA (*Auditory Scene Analysis*), que trata da segregação perceptual do som [Bre94]. De alguma forma, somos capazes de separar diferentes informações do áudio que recebemos, como ocorre em uma conversa em um ambiente ruidoso quando conseguimos separar o que é a fala do restante. Bregman [Bre93] diz que a audição utiliza modelos (*schemas*) que existiam antes dessa segregação ser efetuada, e que um exercício útil para a compreensão da audição é imaginarmos que queremos passar a capacidade de nossa audição a um robô. A narrativa que segue, em texto e em quadrinhos, ilustra como seria realizado o processo da transcrição por seres munidos de um sentido de audição, enfatizando a influência do significado pragmaticamente associada à representação do resultado.

Uma narrativa de ficção

Imaginemos por um instante que um sujeito T, curioso por música, está assistindo a uma apresentação musical informal, realizada por P, seu amigo pianista. Após o término da apresentação, T manifestou interesse em tocar a peça apresentada, pedindo a P para que ele o ensinasse. Porém T completou dizendo que nunca tocou piano na vida. P, sabendo disso, propôs:

> *Ensino-lhe somente se você vencer um desafio que vou propor.*

T aceitou essa restrição. Depois disso, P colocou dois pianos de armário um de frente para o outro, e falou para que T se sentasse em frente a um deles, como um pianista. P explicou brevemente o procedimento do desafio:

Quero que você, sem olhar para minhas mãos ou pés, descubra o que eu vou tocar naquele piano. Você pode pedir para eu repetir quantas vezes quiser, e eu repetirei. Quando souber o que eu toquei, avise e mostre sua conclusão. Você terá uma única chance para apresentar o que eu toquei, e se conseguir descobrir, te ensinarei a tocar a música que você tanto deseja tocar.

T ficou assustado, com medo de não conseguir, mas já havia aceito tal restrição. P tocou um fragmento de uma música. T desesperou-se, não fazia ideia do que fazer. Parecia-lhe impossível realizar tal desafio. P, notando a dificuldade de T, forneceu-lhe uma dica:

Você está parado em frente a um piano, certo? Ele pode ser útil de alguma forma?

T notou que poderia tocar o que quisesse ao piano, mesmo sem técnica. Depois de pressionar algumas teclas, T repetiu uma delas, e pediu para que P tocasse novamente. T tocou novamente essa tecla e então disse:

Essa foi a última tecla que você pressionou.

P considerou a colocação ousada, mas não comentou. Após repetir algumas vezes, T começava a se empolgar e, inclusive, achar divertido o desafio antes amedrontador. T tocou diversas passagens, por vezes em simultaneidade com o que ouvia. Finalmente, após muitas repetições e tentativas, T disse:

Já descobri o que você fez! Tenho certeza de que você fez isto.

E seguiu tocando, notavelmente sem muita técnica, o fragmento. P viu o resultado, e disse:

Meus parabéns! Vou continuar a te ensinar como havia me pedido.

Enquanto T comemorava seu sucesso, P completa:

Porém não foi isso que você tocou que eu fiz.

T assustou-se grandiosamente, pensando em como ele poderia ter errado. P convidou-o para que olhasse o que foi realizado. P pressionou um botão de um dispositivo eletrônico associado ao piano, e o piano começou a tocar sozinho. P então disse:

Tudo o que fiz foi pressionar um botão. Esse piano é automático, mas diferente das pianolas antigas, é capaz de tocar um arquivo MIDI.

T expressa em sua face a indignação e comenta:

Eu nunca ia descobrir que a resposta era apenas o pressionar de um botão. Como eu poderia identificar isso sem saber que essa possibilidade existia? No entanto, eu continuo achando que acertei, visto que as teclas que o piano está tocando são as que eu mostrei.

P replica:

Eu realmente não poderia esperar que sua resposta fosse o pressionar de um botão, pois você não possuía nenhuma evidência dessa possibilidade. Mas você tem certeza de que as suas foram as mesmas teclas pressionadas pelo piano? Esse piano está com outra afinação, veja.

Nesse instante, P pressiona o botão do dispositivo eletrônico. T confere, e nota que realmente as teclas sendo pressionadas são outras. Em sua última tentativa, ele sugere uma transposição:

Esse é somente um outro mapeamento entre as teclas e seu som. Efetuando o mapeamento entre as diferentes afinações dos pianos, o que eu toquei passa a ser exatamente o que esse piano tocou.

P sorri, e completa mostrando que não era somente o mapeamento. T demorou para entender, visto que se trata de algo incomum. Algumas teclas estavam associadas a mais de um som, pois suas duas ou três cordas estavam afinadas diferentemente, e algumas passagens eram realizadas com oitavas paralelas, de uma forma estratégica para não permitir o efeito de *una corda* que a afinação diferenciada fornecia. Em outras palavras, há um mapeamento que ainda poderia ser feito entre as cordas, ao invés das teclas. Porém o timbre dessas cordas de diferentes posições e espessuras também era diferente, e isso também contribuiu para que a solução de T não fosse igual ao atípico mapeamento entre as teclas através das cordas. Indignado com isso tudo, T finaliza:

> *Mas foi isso que eu ouvi quando eu toquei no outro piano.*

P fica muito feliz, e concorda com T. P diz que não poderia dizer que a solução era errada a menos que pudesse fornecer a T outra solução nas condições em que T se encontrava, e T concordasse que essa nova solução era melhor que a anterior.

Apesar da aparência intrinsecamente subjetivista desta narrativa, o único ser humano presente nesta história é o narrador onipresente, em terceira pessoa. P e T, para todos os aspectos práticos, podem ser vistos como robôs programados para realizar a tarefa acima apresentada, em que suas emoções descritas são simulações comportamentais. O conhecimento de como implementar um *software* que realize o comportamento do robô T, transcritor, é a intenção desta dissertação, excetuando os aspectos puramente afetivos do personagem.

Reconstrução do áudio

Uma forma de interpretar a transcrição musical é observá-la como um processo de engenharia reversa realizada por músicos treinados, a fim de obter instruções que possam reproduzir o que eles estão escutando [Kla04, p. 1]. Porém o processo que gerou o áudio original pode ser irreversível, no sentido de que podem existir muitas combinações diferentes de instruções que gerariam a mesma gravação, ou em que houve interferência de fatores desconhecidos até mesmo para seu autor, conforme ilustra a narrativa recém-exposta. Em outras palavras, é esperado que o áudio não possua informações suficientes para fazer todo tipo de distinção, tais como o número de instrumentistas de uma orquestra ou as diferentes configurações físicas de cada instrumento. A ideia de engenharia reversa não deve ser levada como um processo que identifica, em exatidão, como o áudio foi criado, e sim como o áudio poderia ser reconstruído. Em outras palavras, podem haver diferenças entre as ferramentas utilizadas originalmente na gravação e as que o portador da transcrição hipoteticamente utilizará em uma eventual reconstrução.

O objetivo na transcrição de áudio é obter um conjunto de instruções que permitam a reconstrução de uma gravação, dado o conhecimento prévio das possibilidades de reconstrução. O uso de gravações de segmentos de áudio, fonemas, ou notas musicais fazem parte do que seriam possibilidades de reconstrução, desde que exista um processo de síntese para converter tais instruções em sons, independente de quem ou o que realiza a síntese.

Significado da transcrição

Muitos autores [Kla97] se baseiam na ideia de obter apenas algum parâmetro, efetuando apenas parcialmente uma transcrição na forma de análise musical. Dessa forma, a transcrição passa a ser fortemente dependente dos parâmetros que o autor considerava mais relevante, uma abordagem de MIR na qual existe a premissa de que o som não é representável através das informações obtidas. Essa dependência não é um erro, e também não há motivos para ser evitada, além de ter sido, em partes, a abordagem implementada sobre a AudioLazy no atual trabalho, porém a definição do que é o problema da transcrição nesses casos passa a ser relativa ao problema específico do autor da forma como este foi descrito.

Apesar da tarefa da transcrição musical aparentar ser imanentemente subjetiva, o que não é comunicável não tem como pertencer a um objeto observado[3]. Dessa forma, se existem representações alternativas de

[3] Argumentação de Wittgenstein acerca da plausibilidade de uma linguagem privada [Wit12, blocos 243 a 293 da primeira parte].

Figura 1: *Uma história de ficção em quadrinhos: Parte 1*

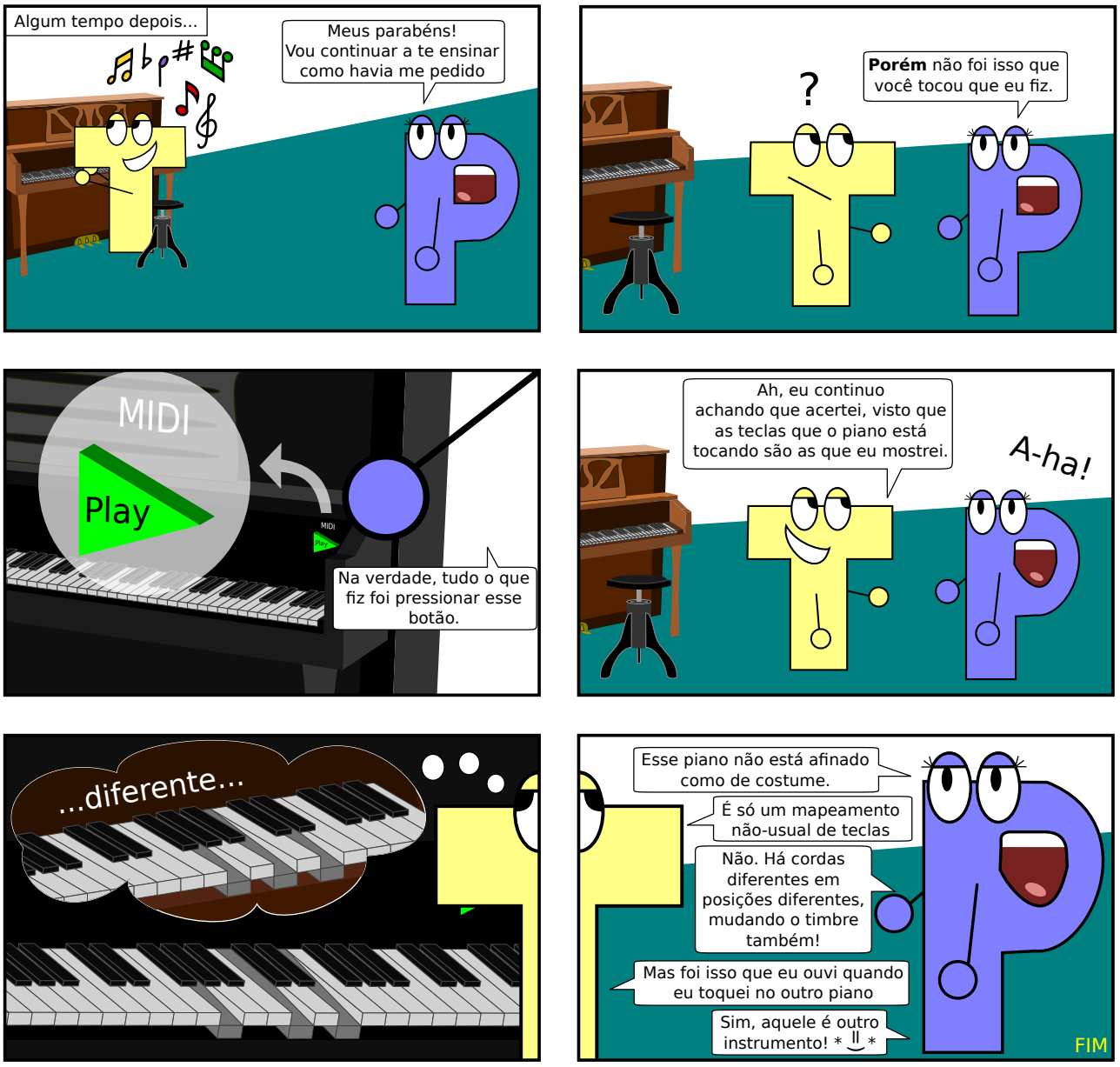


Figura 2: *Uma história de ficção em quadrinhos: Parte 2*

um mesmo áudio, essas representações se referem a diferentes procedimentos e critérios referentes à sua codificação, mas não a uma subjetividade que impõe a necessidade de uma linguagem privada inacessível. Em outras palavras, o problema da transcrição, embora esteja realmente intimamente ligado a aspectos subjetivos da percepção auditiva, tanto no sentido de limiares de frequência a partir dos quais a vibração não atinge informativamente órgãos sensitivos responsáveis pela audição como no sentido de uma possível associação mental entre o som e sua causa (e.g. o ligar de um motor) ou consequência (e.g. uma explosão pode significar problema), necessariamente deve ser comunicável para ser vista como um problema munido de significado para um coletivo.

O conceito de transcrição, embora tenha sido enfatizada neste trabalho no contexto musical por conta do uso de algoritmos como o de Klapuri (seção 3.3.6), também se aplica ao áudio em geral, incluindo, por exemplo, a transcrição fonética de sinais de voz. Há técnicas específicas utilizadas para a transcrição e reconhecimento de voz, como heurísticas baseadas em palavras e estruturas comuns em um idioma específico, por exemplo para para identificar uma palavra mesmo na falta de algum fonema [BDG+09, Sri06].

Em 2008, Klapuri [KV08] caracteriza o objetivo da transcrição através do conteúdo de uma partitura tradicional, explicitando que os parâmetros são a altura[4], o instrumento, o ritmo (*timing*) e durações, de forma bastante similar ao objetivo explicitado em seus trabalhos anteriores [Kla97, Kla04]. Em 1997, Klapuri [Kla97] explicita que busca uma representação intermediária como objetivo, referindo-se a algo que esteja entre a notação da partitura do compositor e o áudio da performance.

[4] Como *pitch*. Talvez o objetivo fosse incluir o croma, mas Klapuri não foi explícito.

Exemplos de código com a AudioLazy

Os exemplos listados na primeira parte deste apêndice fazem parte do projeto AudioLazy e podem ser encontrados diretamente no diretório "audiolazy/examples" de seu repositório público:

http://github.com/danilobellini/audiolazy

Além do *e-mail* do autor dos exemplos (redator do presente texto), data de criação e descrição do exemplo, todos os exemplos iniciam com o cabeçalho que evidencia a licença GPLv3 e a pertinência do arquivo junto ao pacote:

Código-fonte 1: *Cabeçalho presente em todos os arquivos*

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# This file is part of AudioLazy, the signal processing Python package.
# Copyright (C) 2012 Danilo de Jesus da Silva Bellini
#
# AudioLazy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 3 of the License.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Ao final deste apêndice, encontra-se detalhes sobre uma possibilidade de implementação do *overlap-add* como um decorador e um método da classe *Stream*, além de uma rotina feita para a tradução dos gráficos gerados pelo pacote. Apesar de ausente o cabeçalho GPLv3 para evitar a repetição, todos os códigos-fonte do presente trabalho, incluindo aqueles dos capítulos e o *monkey patch* de tradução, devem ser considerados sob a licença GPLv3. Essa indicação está presente em todos os arquivos de código-fonte publicados no repositório no GitHub e no PyPI, assim como os arquivos em reStructuredText da documentação, incluindo os arquivos auto-gerados.

Exemplos fornecidos com o pacote

Segue abaixo os exemplos, com essas informações iniciais relativas à identificação e licença removidas a fim de evitar redundância.

Código-fonte 2: *Ligação direta de entrada de áudio à saída de áudio*

```
from audiolazy import AudioIO

with AudioIO(True) as player_recorder:
  input_data = player_recorder.record(chunk_size=16)
```

```
5   player_recorder.play(input_data, chunk_size=16)
```

Código-fonte 3: *Gráfico com todas as estratégias de função de apodização ou janelamento da AudioLazy*

```
1 from matplotlib import pyplot as plt
2 from audiolazy import window
3
4 M = 256
5
6 for func in window:
7   plt.plot(func(M), label=func.__name__)
8 plt.legend(loc="best")
9 plt.axis(xmin=-5, xmax=M + 5 - 1, ymin=-.05, ymax=1.05)
10 plt.title("AudioLazy windows for size of {M} samples".format(M=M))
11 plt.show()
```

Código-fonte 4: *AudioLazy para uso interativo com uma GUI em wxPython: Animação utilizando o contador em módulo da AudioLazy com áudio gerado por síntese FM*

```
1 # The GUI in this example is based on the dose TDD semaphore source code
2 # https://github.com/danilobellini/dose
3
4 import wx
5 from math import pi
6 from audiolazy import (ControlStream, modulo_counter,
7                        AudioIO, sHz, sinusoid)
8
9 MIN_WIDTH = 15 # pixels
10 MIN_HEIGHT = 15
11 FIRST_WIDTH = 200
12 FIRST_HEIGHT = 200
13 MOUSE_TIMER_WATCH = 50 # ms
14 DRAW_TIMER = 50
15
16 s, Hz = sHz(44100)
17
18 class McFMFrame(wx.Frame):
19
20   def __init__(self, parent):
21     frame_style = (wx.FRAME_SHAPED |      # Allows wx.SetShape
22                    wx.FRAME_NO_TASKBAR |
23                    wx.STAY_ON_TOP |
24                    wx.NO_BORDER
25                   )
26     super(McFMFrame, self).__init__(parent, style=frame_style)
27     self.Bind(wx.EVT_ERASE_BACKGROUND, lambda evt: None)
28     self._paint_width, self._paint_height = 0, 0 # Ensure update_sizes at
29                                                  # first on_paint
30     self.ClientSize = (FIRST_WIDTH, FIRST_HEIGHT)
31     self.Bind(wx.EVT_PAINT, self.on_paint)
32     self._draw_timer = wx.Timer(self)
33     self.Bind(wx.EVT_TIMER, self.on_draw_timer, self._draw_timer)
34     self.on_draw_timer()
35     self.angstep = ControlStream(pi/90)
36     self.rotstream = modulo_counter(modulo=2*pi, step=self.angstep)
37     self.rotation_data = iter(self.rotstream)
38
```

```
39    def on_draw_timer(self, evt=None):
40      self.Refresh()
41      self._draw_timer.Start(DRAW_TIMER, True)
42
43    def on_paint(self, evt):
44      dc = wx.AutoBufferedPaintDCFactory(self)
45      gc = wx.GraphicsContext.Create(dc) # Anti-aliasing
46
47      gc.SetPen(wx.Pen("blue", width=4))
48      gc.SetBrush(wx.Brush("black"))
49      w, h = self.ClientSize
50      gc.DrawRectangle(0, 0, w, h)
51
52      gc.SetPen(wx.Pen("gray", width=2))
53      w, h = w - 10, h - 10
54      gc.Translate(5, 5)
55      gc.DrawEllipse(0, 0, w, h)
56      gc.SetPen(wx.Pen("red", width=1))
57      gc.SetBrush(wx.Brush("yellow"))
58      gc.Translate(w * .5, h * .5)
59      gc.Scale(w, h)
60      rot = self.rotation_data.next()
61      gc.Rotate(-rot)
62      gc.Translate(.5, 0)
63      gc.Rotate(rot)
64      gc.Scale(1./w, 1./h)
65      gc.DrawEllipse(-5, -5, 10, 10)
66
67
68  class InteractiveFrame(McFMFrame):
69    def __init__(self, parent):
70      super(InteractiveFrame, self).__init__(parent)
71      self._timer = wx.Timer(self)
72      self.Bind(wx.EVT_RIGHT_DOWN, self.on_right_down)
73      self.Bind(wx.EVT_LEFT_DOWN, self.on_left_down)
74      self.Bind(wx.EVT_TIMER, self.on_timer, self._timer)
75
76    @property
77    def player(self):
78      return self._player
79
80    @player.setter
81    def player(self, value):
82      # Also initialize playing thread
83      self._player = value
84      self.volume_ctrl = ControlStream(.2)
85      self.carrier_ctrl = ControlStream(220)
86      self.mod_ctrl = ControlStream(440)
87      sound = sinusoid(freq=self.carrier_ctrl * Hz,
88                       phase=sinusoid(self.mod_ctrl * Hz)
89                      ) * self.volume_ctrl
90      self.playing_thread = player.play(sound)
91
92    def on_right_down(self, evt):
93      self.Close()
94
95    def on_left_down(self, evt):
```

```
96      self._key_state = None # Ensures initialization
97      self.on_timer(evt)
98
99    def on_timer(self, evt):
100     """
101     Keep watching the mouse displacement via timer
102     Needed since EVT_MOVE doesn't happen once the mouse gets outside the
103     frame
104     """
105     ctrl_is_down = wx.GetKeyState(wx.WXK_CONTROL)
106     ms = wx.GetMouseState()
107
108     # New initialization when keys pressed change
109     if self._key_state != ctrl_is_down:
110       self._key_state = ctrl_is_down
111
112       # Keep state at click
113       self._click_ms_x, self._click_ms_y = ms.x, ms.y
114       self._click_frame_x, self._click_frame_y = self.Position
115       self._click_frame_width, self._click_frame_height = self.ClientSize
116
117       # Avoids refresh when there's no move (stores last mouse state)
118       self._last_ms = ms.x, ms.y
119
120       # Quadrant at click (need to know how to resize)
121       width, height = self.ClientSize
122       self._quad_signal_x = 1 if (self._click_ms_x -
123                                   self._click_frame_x) / width > .5 else -1
124       self._quad_signal_y = 1 if (self._click_ms_y -
125                                   self._click_frame_y) / height > .5 else -1
126
127     # "Polling watcher" for mouse left button while it's kept down
128     if ms.leftDown:
129       if self._last_ms != (ms.x, ms.y): # Moved?
130         self._last_ms = (ms.x, ms.y)
131         delta_x = ms.x - self._click_ms_x
132         delta_y = ms.y - self._click_ms_y
133
134         # Resize
135         if ctrl_is_down:
136           # New size
137           new_w = max(MIN_WIDTH, self._click_frame_width +
138                                  2 * delta_x * self._quad_signal_x
139                       )
140           new_h = max(MIN_HEIGHT, self._click_frame_height +
141                                   2 * delta_y * self._quad_signal_y
142                       )
143           self.ClientSize = new_w, new_h
144           self.SendSizeEvent() # Needed for wxGTK
145
146           # Center should be kept
147           center_x = self._click_frame_x + self._click_frame_width / 2
148           center_y = self._click_frame_y + self._click_frame_height / 2
149           self.Position = (center_x - new_w / 2,
150                            center_y - new_h / 2)
151
152           self.Refresh()
```

```
153             self.volume_ctrl.value = (new_h * new_w) / 3e5
154
155           # Move the window
156           else:
157             self.Position = (self._click_frame_x + delta_x,
158                              self._click_frame_y + delta_y)
159
160             # Find the new center position
161             x, y = self.Position
162             w, h = self.ClientSize
163             cx, cy = x + w/2, y + h/2
164             self.mod_ctrl.value = 2.5 * cx
165             self.carrier_ctrl.value = 2.5 * cy
166             self.angstep.value = (cx + cy) * pi * 2e-4
167
168         # Since left button is kept down, there should be another one shot
169         # timer event again, without creating many timers like wx.CallLater
170         self._timer.Start(MOUSE_TIMER_WATCH, True)
171
172
173 class McFMApp(wx.App):
174
175   def OnInit(self):
176     self.SetAppName("dose")
177     self.wnd = InteractiveFrame(None)
178     self.wnd.Show()
179     self.SetTopWindow(self.wnd)
180     return True # Needed by wxPython
181
182
183 if __name__ == "__main__":
184   with AudioIO() as player:
185     app = McFMApp(False, player)
186     app.wnd.player = player
187     app.MainLoop()
```

Código-fonte 5: *Reprodução de um coral de J. S. Bach: Uso do algoritmo de Karplus-Strong para sintetizar um coral selecionado aleatoriamente entre os disponíveis em um corpus do Music21*

```
1  from music21 import corpus
2  from music21.expressions import Fermata
3  import audiolazy as lz
4  import random
5  import operator
6
7  def ks_mem(freq):
8    """ Alternative memory for Karplus-Strong """
9    return (sum(lz.sinusoid(x * freq) for x in [1, 3, 9]) +
10           lz.white_noise() + lz.Stream(-1, 1)) / 5
11
12 # Configuration
13 rate = 44100
14 s, Hz = lz.sHz(rate)
15 ms = 1e-3 * s
16
17 beat = 90 # bpm
18 step = 60. / beat * s
```

```
19
20 # Open the choral file
21 choral_file = corpus.getBachChorales()[random.randint(0, 399)]
22 choral = corpus.parse(choral_file)
23 print u"Playing", choral.metadata.title
24
25 # Creates the score from the music21 data
26 score = reduce(operator.concat,
27                [[(pitch.frequency * Hz, # Note
28                   note.offset * step, # Starting time
29                   note.quarterLength * step, # Duration
30                   Fermata in note.expressions) for pitch in note.pitches]
31                                               for note in choral.flat.notes]
32               )
33
34 # Mix all notes into song
35 song = lz.Streamix()
36 last_start = 0
37 for freq, start, dur, has_fermata in score:
38   delta = start - last_start
39   if has_fermata:
40     delta *= 2
41   song.add(delta, lz.karplus_strong(freq, memory=ks_mem(freq)) * lz.ones(dur))
42   last_start = start
43
44 # Play the song!
45 with lz.AudioIO(True) as player:
46   song = song.append(lz.zeros(.5 * s)) # To avoid a click at the end
47   player.play(song, rate=rate)
```

Código-fonte 6: *Criação de gráfico com todas as estratégias de filtros gammatone*

```
1 from __future__ import division
2 from audiolazy import (erb, gammatone, gammatone_erb_constants, sHz, impulse,
3                         dB20)
4 from numpy import linspace, ceil
5 from matplotlib import pyplot as plt
6
7 # Initialization info
8 rate = 44100
9 s, Hz = sHz(rate)
10 ms = 1e-3 * s
11 plot_freq_time = {80.: 60 * ms,
12                   100.: 50 * ms,
13                   200.: 40 * ms,
14                   500.: 25 * ms,
15                   800.: 20 * ms,
16                   1000.: 15 * ms}
17 freq = linspace(0.1, 2 * max(freq for freq in plot_freq_time), 100)
18
19 fig1 = plt.figure("Frequency response", figsize=(16, 9), dpi=60)
20 fig2 = plt.figure("Impulse response", figsize=(16, 9), dpi=60)
21
22 # Plotting loop
23 for idx, (fc, endtime) in enumerate(sorted(plot_freq_time.iteritems()), 1):
24   # Configuration for the given frequency
25   num_samples = int(round(endtime))
```

```
  time_scale = linspace(0, num_samples / ms, num_samples)
  bw = gammatone_erb_constants(4)[0] * erb(fc * Hz, Hz)

  # Subplot configuration
  plt.figure(1)
  plt.subplot(2, ceil(len(plot_freq_time) / 2), idx)
  plt.title("Frequency response - {0} Hz".format(fc))
  plt.xlabel("Frequency (Hz)")
  plt.ylabel("Gain (dB)")

  plt.figure(2)
  plt.subplot(2, ceil(len(plot_freq_time) / 2), idx)
  plt.title("Impulse response - {0} Hz".format(fc))
  plt.xlabel("Time (ms)")
  plt.ylabel("Amplitude")

  # Plots each filter frequency and impulse response
  for gt, config in zip(gammatone, ["b-", "g--", "r-.", "k:"]):
    filt = gt(fc * Hz, bw)

    plt.figure(1)
    plt.plot(freq, dB20(filt.freq_response(freq * Hz)), config,
             label=gt.__name__)

    plt.figure(2)
    plt.plot(time_scale, filt(impulse()).take(num_samples), config,
             label=gt.__name__)

# Finish
for graph in fig1.axes + fig2.axes:
  graph.grid()
  graph.legend(loc="best")

fig1.tight_layout()
fig2.tight_layout()

plt.show()
```

Código-fonte 7: *Comparação entre o desempenho da AudioLazy e do NumPy para o CPython e o PyPy*

```
from __future__ import unicode_literals, print_function
from timeit import timeit
import sys


# ==================
# Some initialization
# ==================
num_tests = 30
is_pypy = sys.subversion[0] == "PyPy"
if is_pypy:
  print("PyPy detected!")
  print()
  numpy_name = "numpypy"
else:
  numpy_name = "numpy"

```

```
18
19 # ======================
20 # AudioLazy benchmarking
21 # ======================
22 kws = {}
23 kws["setup"] = """
24 from audiolazy import sHz, adsr, sinusoid
25 from math import pi
26 """
27 kws["number"] = num_tests
28 kws["stmt"] = """
29 s, Hz = sHz(44100)
30 ms = 1e-3 * s
31 env = adsr(dur=5*s, a=20*ms, d=30*ms, s=.8, r=50*ms)
32 sin_data = sinusoid(freq=440*Hz,
33                     phase=sinusoid(220*Hz) * pi)
34 result = sum(env * sin_data)
35 """
36
37 print("=== AudioLazy benchmarking ===")
38 print("Trials:", kws["number"])
39 print()
40 print("Setup code:")
41 print(kws["setup"])
42 print()
43 print("Benchmark code (also executed once as 'setup'/'training'):")
44 kws["setup"] += kws["stmt"] # Helpful for PyPy
45 print(kws["stmt"])
46 print()
47 print("Mean time (milliseconds):")
48 print(timeit(**kws) * 1e3 / num_tests)
49 print("==============================")
50 print()
51
52
53 # ==================
54 # Numpy benchmarking
55 # ==================
56 kws_np = {}
57 kws_np["setup"] = "import {0} as np".format(numpy_name)
58 kws_np["number"] = num_tests
59 kws_np["stmt"] = """
60 rate = 44100
61 dur = 5 * rate
62 sustain_level = .8
63 # The np.linspace isn't in numpypy yet; it uses float64
64 attack = np.linspace(0., 1., num=np.round(20e-3 * rate), endpoint=False)
65 decay = np.linspace(1., sustain_level, num=np.round(30e-3 * rate),
66                     endpoint=False)
67 release = np.linspace(sustain_level, 0., num=np.round(50e-3 * rate),
68                       endpoint=False)
69 sustain_dur = dur - len(attack) - len(decay) - len(release)
70 sustain = sustain_level * np.ones(sustain_dur)
71 env = np.hstack([attack, decay, sustain, release])
72 freq220 = 220 * 2 * np.pi / rate
73 freq440 = 440 * 2 * np.pi / rate
74 phase220 = np.arange(dur, dtype=np.float64) * freq220
```

```
phase440 = np.arange(dur, dtype=np.float64) * freq440
sin_data = np.sin(phase440 + np.sin(phase220) * np.pi)
result = np.sum(env * sin_data)
"""

# Alternative for numpypy (since it don't have "linspace" nor "hstack")
stmt_npp = """
rate = 44100
dur = 5 * rate
sustain_level = .8
len_attack = int(round(20e-3 * rate))
attack = np.arange(len_attack, dtype=np.float64) / len_attack
len_decay = int(round(30e-3 * rate))
decay = (np.arange(len_decay - 1, -1, -1, dtype=np.float64
                   ) / len_decay) * (1 - sustain_level) + sustain_level
len_release = int(round(50e-3 * rate))
release = (np.arange(len_release - 1, -1, -1, dtype=np.float64
                     ) / len_release) * sustain_level
env = np.ndarray(dur, dtype=np.float64)
env[:len_attack] = attack
env[len_attack:len_attack+len_decay] = decay
env[len_attack+len_decay:dur-len_release] = sustain_level
env[dur-len_release:dur] = release
freq220 = 220 * 2 * np.pi / rate
freq440 = 440 * 2 * np.pi / rate
phase220 = np.arange(dur, dtype=np.float64) * freq220
phase440 = np.arange(dur, dtype=np.float64) * freq440
sin_data = np.sin(phase440 + np.sin(phase220) * np.pi)
result = np.sum(env * sin_data)
"""

try:
  if is_pypy:
    import numpypy as np
  else:
    import numpy as np
except ImportError:
  print("Numpy not found. Finished benchmarking!")
else:
  if is_pypy:
    kws_np["stmt"] = stmt_npp

  print("Numpy found!")
  print()
  print("=== Numpy benchmarking ===")
  print("Trials:", kws_np["number"])
  print()
  print("Setup code:")
  print(kws_np["setup"])
  print()
  print("Benchmark code (also executed once as 'setup'/'training'):")
  kws_np["setup"] += kws_np["stmt"] # Helpful for PyPy
  print(kws_np["stmt"])
  print()
  print("Mean time (milliseconds):")
  print(timeit(**kws_np) * 1e3 / num_tests)
  print("==========================")
```

Overlap-add, ou sobrepor e somar

Este é um método utilizado para reunificar blocos em uma sequência quando existe um salto menor que o tamanho do bloco. Isso é realizado somando ambos os blocos com uma medida de ponderação. O código que segue expõe um *monkey patch* para a inserção do recurso à classe *Stream* e um decorador, conforme utilizado no código-fonte 3.15. Resumidamente, há dois casos de uso para os quais o procedimento foi feito:

- **Quando o tamanho do bloco e do salto são idênticos por padrão**: apenas o tamanho é passado como parâmetro e os pesos são sempre iguais a 1.
- **Quando o tamanho do bloco e do salto são fornecidos pelo usuário**: ambos o tamanho e o salto são passados como parâmetros, e os pesos são sempre uma aplicação de uma função de janelamento triangular sobre cada bloco.

Código-fonte 8: *Inserindo o recurso de overlap-add na AudioLazy 0.04*

```
1
2  from functools import wraps
3  from audiolazy import *
4
5  def stream_monkey_patch():
6    def overlap_add(self, hop=None, zero=0., wnd=None):
7      if hop is None:
8        hop = lambda x: x
9        wnd = window.rect
10     if wnd is None:
11       wnd = window.triangle
12
13     smix = Streamix(zero=zero)
14     ib = iter(self)
15     blk = ib.next()
16     smix.add(0, blk * Stream(wnd(len(blk))))
17
18     def add_after_event():
19       blk = ib.next()
20       lblk = len(blk)
21       smix.add(hop(lblk) if callable(hop) else hop,
22                blk * Stream(wnd(lblk)))
23       smix.add(0, add_after_event())
24       if False:
25         yield # Just to be a generator function
26
27     smix.add(0, add_after_event())
28     return smix
29
30   Stream.overlap_add = overlap_add
31
32 stream_monkey_patch()
33
34 def overlap_add(**kws):
35   def overlap_add_decorator(func):
36     @wraps(func)
37     def new_func(*args, **kwargs):
38       kwsup = kws.copy()
39       kwsup.update(kwargs)
40       kwoa = {k: kwsup[k] for k in "hop zero wnd".split() if k in kwsup}
41       return Stream(func(*args, **kwargs)).overlap_add(**kwoa)
```

```
42      return new_func
43    return overlap_add_decorator
```

Tradução dos gráficos feitos com o pacote AudioLazy

Esta seção contém a rotina que traduz as legendas e títulos dos gráficos, alterando a funcionalidade do pacote AudioLazy em tempo real através de um decorador aplicado como *monkey patch* aos métodos de criação de gráficos. Essa rotina foi necessária para possibilitar que os gráficos da parte escrita deste trabalho estivessem em português, mas os valores do dicionário *table* no código-fonte que segue podem ser alterados para seguir qualquer idioma. No caso de uma tradução para outro idioma, pode ser necessário inserir os valores que não foram alterados na tradução do inglês para o português tais como "Magnitude (dB)", mas estes foram deixados em comentários para facilitar a tarefa de tradução. No caso de uma nova tradução, recomenda-se que o tradutor utilize *unicode* a fim de garantir a portabilidade do código.

Recomenda-se que o código-fonte que segue seja salvo como um módulo a ser importado pelo Python, independente se antes ou depois de importação explícita junto à AudioLazy. Tal importação é suficiente para alterar a funcionalidade do sistema de geração de gráficos. Nas últimas 5 linhas deste código, são realizados dois exemplos para a visualização do resultado da tradução. Esses exemplos somente ocorrem ao se utilizar o módulo como um *script*, fazendo de um mesmo arquivo útil para dois casos de uso distintos, além de permitir a avaliação de seu conteúdo de maneira limpa com testes manuais presentes no próprio código.

Código-fonte 9: *Monkey patch para a tradução de gráficos*

```
#!/usr/bin/env python
from audiolazy import LinearFilter, FilterList, z
import pylab
import re

# Decorador para traduzir os gráficos
def plot_decorator(method):
  table = {u"Frequency response": u"Resposta em frequência",
           u"Frequency (rad/sample)": u"Frequência (rad/amostra)",
           u"Frequency (Hz)": u"Frequência (Hz)",
          #u"Magnitude (dB)": u"Magnitude (dB)",
          #u"Magnitude (linear)": u"Magnitude (linear)",
           u"Magnitude (squared)": u"Magnitude (quadrática)",
           u"Phase (rad)": u"Fase (rad)",
           u"Zero-Pole plot": u"Diagrama de zeros e polos",
           u"Real part": u"Parte real",
           u"Imaginary part": u"Parte imaginária",
           u"poles": u"polos",
          #u"zeros": u"zeros",
          }

  def mapper(key):
    if key in table:
      return table[key]
    if key.startswith("Zero-Pole plot ("):
      data = re.findall(r"(.*?) \((\d*) (.*?), (\d*) (.*)\)", key)[0]
      return "{0} ({1} {2}, {3} {4})".format(*[mapper(k) for k in data])
    return key

  def plot_em_portugues(self, *args, **kwargs):
    fig = method(self, *args, **kwargs)
    for ax in fig.axes:
      ax.set_xlabel(mapper(ax.get_xlabel()))
```

```
34        ax.set_ylabel(mapper(ax.get_ylabel()))
35        ax.set_title(mapper(ax.get_title()))
36      return fig
37
38    return plot_em_portugues
39
40 # Monkey patch
41 LinearFilter.plot = plot_decorator(LinearFilter.plot)
42 LinearFilter.zplot = plot_decorator(LinearFilter.zplot)
43 FilterList.plot = plot_decorator(FilterList.plot)
44 FilterList.zplot = plot_decorator(FilterList.zplot)
45
46 # Exemplos
47 if __name__ == "__main__":
48    (1 + z ** -2).plot(mag_scale="squared")
49    ((1 - z ** -7) / (1 + z ** -2)).zplot()
50    pylab.show()
```

Documentação do pacote AudioLazy via Sphinx

Boa parte do código escrito para o pacote *AudioLazy* foi feito com a ideia de possibilitar o desenvolvedor de ter acesso à sem a necessidade de nada além do próprio pacote. Isso é feito através de *docstrings*, incluindo *doctests*, conforme descrito nas seções 2.3.8 e 2.3.7.

No que segue, estão acopladas as páginas da versão da documentação da AudioLazy gerada pelo Sphinx a partir do código. O mesmo documento está nos formatos HTML, ePUB, Texinfo, páginas de manual (*man pages*, relativo ao uso do *man* no Linux), arquivos de texto comuns ("**.txt*" em UTF-8) e LaTeX. Este último garante que a documentação também seja facilmente convertida nos formatos DVI (*Device independent file format*), *PostScript* e PDF (*Portable Document Format*). As páginas que seguem referem-se ao formato PDF da documentação. Como indicado na seção 1.4, a versão em HTML atualizada encontra-se em:

http://pythonhosted.org/audiolazy/

O idioma das páginas que seguem, como é de se esperar em uma documentação oficial de um pacote de *software* para uso internacional, é o inglês.

AudioLazy documentation

Release 0.04

Danilo de Jesus da Silva Bellini

2013-02-18

CONTENTS

Expressive Digital Signal Processing (DSP) package for Python.

CHAPTER

ONE

INTRODUCTION

This is the main AudioLazy documentation, whose contents are mainly from the repository documentation and source code docstrings, tied together with Sphinx (http://sphinx.pocoo.org). The sections below can introduce you to the AudioLazy Python DSP package.

1.1 Lazyness and object representation

There are several tools and packages that let the Python use and expressiveness look like languages such as MatLab and Octave. However, the eager evaluation done by most of these tools make it difficult, perhaps impossible, to use them for real time audio processing. To avoid such eagerness, one can make the calculations only when data is requested, not when the path to the data is given. This is the core idea in lazyness that allows:

- Real-time application (you don't need to wait until all data is processed to have a result);
- Endless data sequence representation;
- Data-flow representation;
- Task elimination when a reverse task is done: instead of doing something to then undo, nothing needs to be done, and no conscious optimization need to be done for that.

Another difficulty concerns expressive code creation for audio processing in blocks through indexes and vectors. Sometimes, that's unavoidable, or at least such avoidance would limit the power of the system that works with sequence data.

Block sequences can be found from sample sequences being both objects, where the latter can be the result of a method or function over the former. The information needed for such is the block size and where would start the next block. Although one can think about the last block and the exact index where it would start, most of the time spent in steps like this one happens to be an implementation issue that just keep the focus away from the problem being worked on. To allow a thing like an endless data sequence, there should be no need to know when something stops.

Probably an engineer would find the use of equations and structures from electrical engineering theory much cleaner to understand than storing everything into data arrays, mainly when common operations are done to these representations. What is the product of the filter with numerator `[1, 7, 2]` and denominator `[1, 0.5, 0.2]` as its system equation with the one that has the arrays reversed like `[2, 7, 1]`? That might be simple, and the reversed would avoid questions like "what comes first, the zero or the [minus] two exponent?", but maybe we could get more efficient ourselves if we had something easier: multiplication could be written once and for all and with a representation programmers are used to see. This would be even more expressive if we could get rid from the asymmetry of a method call like `filt1.multiply_by(filt2)`, since multiplication in this case should be commutative. The use of natural operators is possible in a language that allows operator overloading, but for such we need to describe those equations and structures as objects and object relationships.

The name `Hz` can be a number that would allow conversion to a default DSP internal rad/samples unit, so one can write things like `freq = 440 * Hz`. This isn't difficult in probably any language, but can help in expressiveness, already. If (almost) everything would need data in "samples" or "rad/sample" units, constants for converting these from "second" and "hertz" would help with the code expressiveness. A comb filter

`comb.tau(delay=30*s, tau=40*s)` can represent a comb filter with the given delay and time constant, both in samples, but with a more clear meaning for the reader than it would have with an expression like `[1] + [0] * 239999 + [alpha]`. Would it be needed to store all those zeros while just using the filter to get a frequency response plot?

It's possible to avoid some of these problems with well-chosen constants, duck typing, overloaded operators, functions as first-class citizens, object oriented together with functional style programming, etc.., resources that the Python language gives us for free.

1.2 What does it do?

Prioritizing code expressiveness, clarity and simplicity, without precluding the lazy evaluation, and aiming to be used together with Numpy, Scipy and Matplotlib as well as default Python structures like lists and generators, AudioLazy is a package written in pure Python proposing digital audio signal processing (DSP), featuring:

- A `Stream` class for finite and endless signals representation with elementwise operators (auto-broadcast with non-iterables) in a common Python iterable container accepting heterogeneous data;
- Strongly sample-based representation (Stream class) with easy conversion to block representation using the `Stream.blocks(size, hop)` method;
- Sample-based interactive processing with `ControlStream`;
- `Streamix` mixer for iterables given their starting time deltas;
- Multi-thread audio I/O integration with PyAudio;
- Linear filtering with Z-transform filters directly as equations (e.g. `filt = 1 / (1 - .3 * z ** -1)`), including linear time variant filters (i.e., the `a` in `a * z ** k` can be a Stream instance), cascade filters (behaves as a list of filters), resonators, etc.. Each `LinearFilter` instance is compiled just in time when called;
- Zeros and poles plots and frequency response plotting integration with MatPlotLib;
- Linear Predictive Coding (LPC) directly to `ZFilter` instances, from which you can find PARCOR coeffs and LSFs;
- Both sample-based (e.g., zero-cross rate, envelope, moving average, clipping, unwrapping) and block-based (e.g., window functions, DFT, autocorrelation, lag matrix) analysis and processing tools;
- A simple synthesizer (Table lookup, Karplus-Strong) with processing tools (Linear ADSR envelope, fade in/out, fixed duration line stream) and basic wave data generation (sinusoid, white noise, impulse);
- Biological auditory periphery modeling (ERB and gammatone filter models);
- Multiple implementation organization as `StrategyDict` instances: callable dictionaries that allows the same name to have several different implementations (e.g. `erb`, `gammatone`, `lowpass`, `resonator`, `lpc`, `window`);
- Converters among MIDI pitch numbers, strings like "F#4" and frequencies;
- Polynomials, Stream-based functions from itertools, math, cmath, and more! Go try yourself! =)

1.3 Installing

The package works both on Linux and on Windows. You can find the last stable version at PyPI (http://pypi.python.org/pypi/audiolazy) and install it with the usual Python installing mechanism:

```
python setup.py install
```

If you have pip, you can go directly (use `-U` for update or reinstall):

```
pip install audiolazy
```

for downloading (from PyPI) and installing the package for you, or:

```
pip install -U .
```

To install from a path that has the `setup.py` file and the package data uncompressed previously.

For the *bleeding-edge* version, you can install directly from the github repository (requires `git` for cloning):

```
pip install -U git+git://github.com/danilobellini/audiolazy.git
```

For older versions, you can install from the PyPI link or directly from the github repository, based on the repository tags. For example, to install the version 0.03 (requires `git` for cloning):

```
pip install -U git+git://github.com/danilobellini/audiolazy.git@v0.03
```

The package doesn't have any strong dependency for its core besides the Python itself and its standard library, but you might need:

- PyAudio: needed for playing and recording audio (`AudioIO` class);
- NumPy: needed for doing some maths, such as finding the LSFs from a filter or roots from a polynomial;
- MatPlotLib: needed for all default plotting, like in `LinearFilter.plot` method and several examples;
- SciPy (testing only): used as an oracle for LTI filter testing;
- pytest and pytest-cov (testing only): runs test suite and shows code coverage status;
- wxPython (example only): used by one example with FM synthesis in an interactive GUI;
- Music21 (example only): there's one example that gets the Bach chorals from that package corpora for synthesizing and playing;
- Sphinx (documentation only): it can create the software documentation in several different file formats.

Beside examples and tests, only the filter plotting with `plot` and `zplot` methods needs MatPlotLib. Also, the routines that needs NumPy up to now are:

- Root finding with `zeros` and `poles` properties (filter classes) or with `roots` property (Poly class);
- Some Linear Predictive Coding (`lpc`) strategies: `nautocor`, `autocor` and `covar`;
- Line Spectral Frequencies `lsf` and `lsf_stable` functions.

1.4 Getting started

Before all examples below, it's easier to get everything from audiolazy namespace:

```
from audiolazy import *
```

All modules starts with "lazy_", but their data is already loaded in the main namespace. These two lines of code do the same thing:

```
from audiolazy.lazy_stream import Stream
from audiolazy import Stream
```

Endless iterables with operators (be careful with loops through an endless iterator!):

```
>>> a = Stream(2) # Periodic
>>> b = Stream(3, 7, 5, 4) # Periodic
>>> c = a + b # Elementwise sum, periodic
>>> c.take(15) # First 15 elements from the Stream object
[5, 9, 7, 6, 5, 9, 7, 6, 5, 9, 7, 6, 5, 9, 7]
```

And also finite iterators (you can think on any Stream as a generator with elementwise operators):

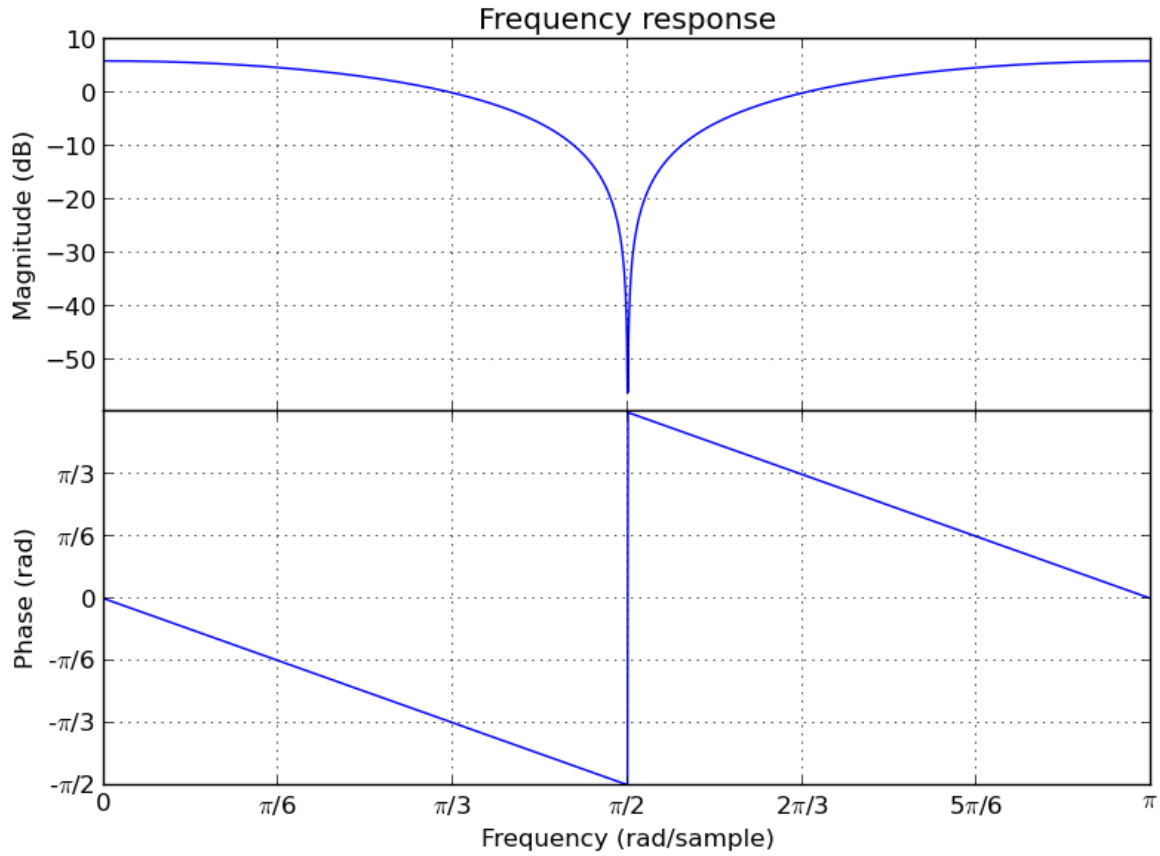
```
>>> a = Stream([1, 2, 3, 2, 1]) # Finite, since it's a cast from an iterable
>>> b = Stream(3, 7, 5, 4) # Periodic
>>> c = a + b # Elementwise sum, finite
>>> list(c)
[4, 9, 8, 6, 4]
```

LTI Filtering from system equations (Z-transform). After this, try summing, composing, multiplying ZFilter objects:

```
>>> filt = 1 - z ** -1 # Diff between a sample and the previous one
>>> filt
1 - z^-1
>>> data = filt([.1, .2, .4, .3, .2, -.1, -.3, -.2]) # Past memory has 0.0
>>> data # This should have internally [.1, .1, .2, -.1, -.1, -.3, -.2, .1]
<audiolazy.lazy_stream.Stream object at ...>
>>> data *= 10 # Elementwise gain
>>> [int(round(x)) for x in data] # Streams are iterables
[1, 1, 2, -1, -1, -3, -2, 1]
>>> data_int = filt([1, 2, 4, 3, 2, -1, -3, -2], zero=0) # Now zero is int
>>> list(data_int)
[1, 1, 2, -1, -1, -3, -2, 1]
```

LTI Filter frequency response plot (needs MatPlotLib):

```
(1 + z ** -2).plot().show()
```



But the `matplotlib.figure.Figure.show` method won't work unless you're using a newer version of MatPlotLib (works on MatPlotLib 1.2.0), but you still can save the above plot directly to a PDF, PNG, etc. with older versions (e.g. MatPlotLib 1.0.1):

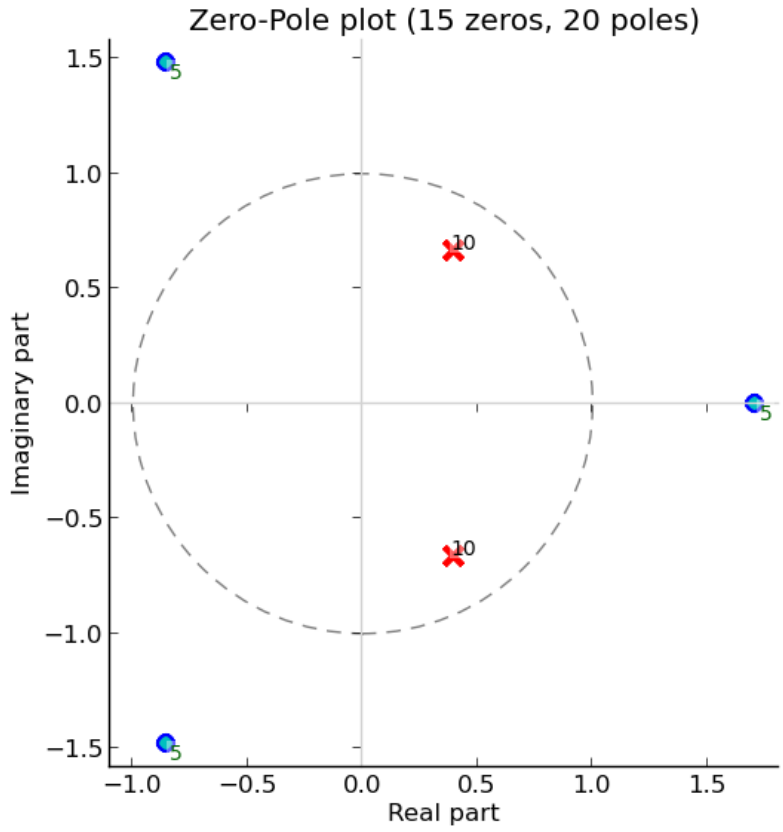
```
(1 + z ** -2).plot().savefig("my_plot.pdf")
```

On the other hand, you can always show the figure using MatPlotLib directly:

```
from matplotlib import pyplot as plt # Or "import pylab as plt"
filt = 1 + z ** -2
fig1 = filt.plot(plt.figure()) # Argument not needed on the first figure
fig2 = filt.zplot(plt.figure()) # The argument ensures a new figure
plt.show()
```

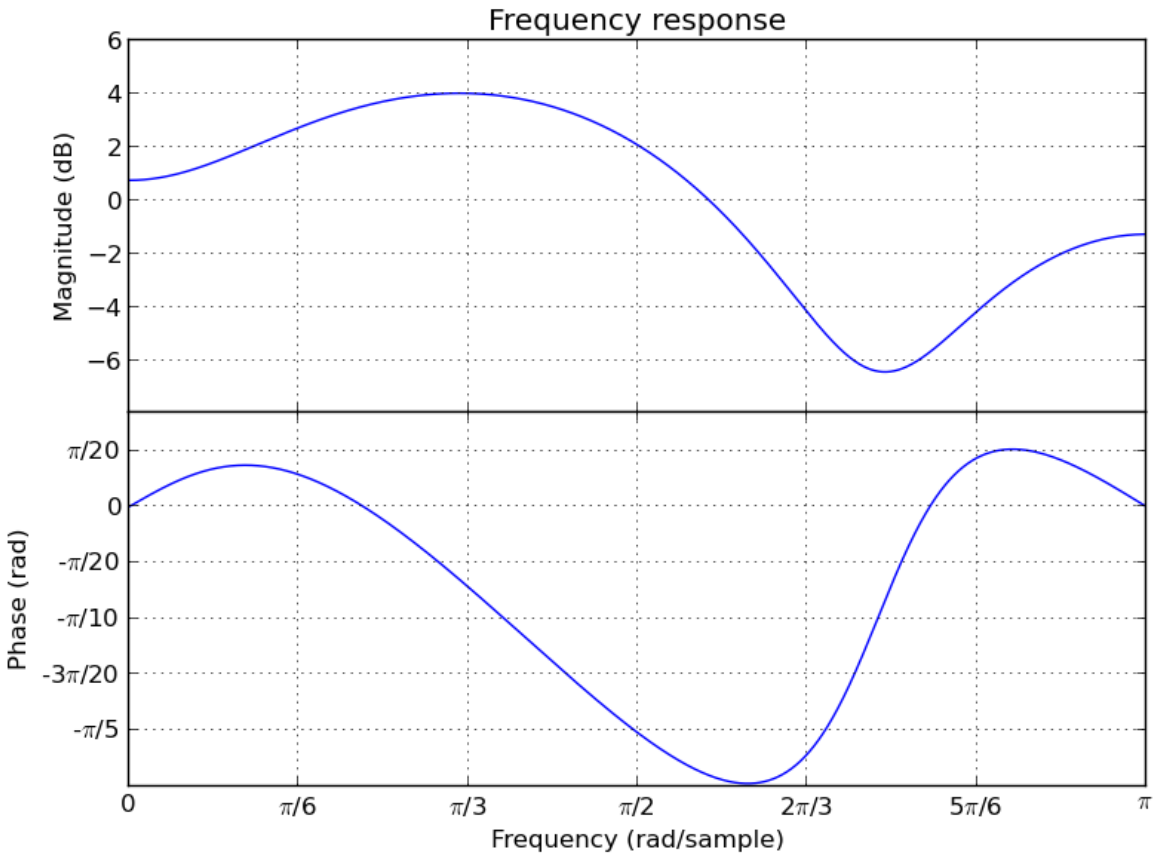
CascadeFilter instances and ParallelFilter instances are lists of filters with the same operator behavior as a list, and also works for plotting linear filters. Constructors accepts both a filter and an iterable with filters. For example, a zeros and poles plot (needs MatPlotLib):

```
filt1 = CascadeFilter(0.2 - z ** -3) # 3 zeros
filt2 = CascadeFilter(1 / (1 -.8 * z ** -1 + .6 * z ** -2)) # 2 poles
# Here __add__ concatenates and __mul__ by an integer make reference copies
filt = (filt1 * 5 + filt2 * 10) # 15 zeros and 20 poles
filt.zplot().show()
```



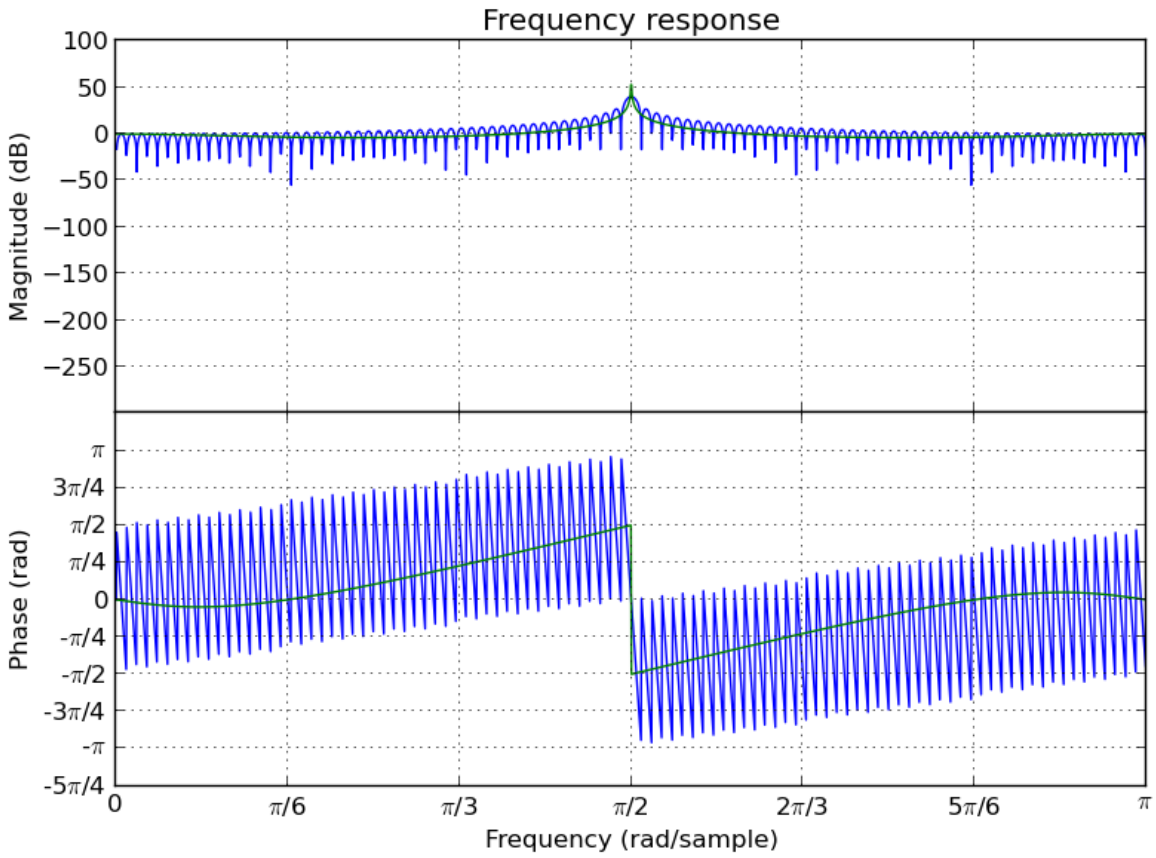
Linear Predictive Coding (LPC) autocorrelation method analysis filter frequency response plot (needs MatPlotLib):

```
lpc([1, -2, 3, -4, -3, 2, -3, 2, 1], order=3).plot().show()
```



Linear Predictive Coding covariance method analysis and synthesis filter, followed by the frequency response plot together with block data DFT (MatPlotLib):

```
>>> data = Stream(-1., 0., 1., 0.) # Periodic
>>> blk = data.take(200)
>>> analysis_filt = lpc.covar(blk, 4)
>>> analysis_filt
1 + 0.5 * z^-2 - 0.5 * z^-4
>>> residual = list(analysis_filt(blk))
>>> residual[:10]
[-1.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> synth_filt = 1 / analysis_filt
>>> synth_filt(residual).take(10)
[-1.0, 0.0, 1.0, 0.0, -1.0, 0.0, 1.0, 0.0, -1.0, 0.0]
>>> amplified_blk = list(Stream(blk) * -200) # For alignment w/ DFT
>>> synth_filt.plot(blk=amplified_blk).show()
```



AudioLazy doesn't need any audio card to process audio, but needs PyAudio to play some sound:

```
rate = 44100 # Sampling rate, in samples/second
s, Hz = sHz(rate) # Seconds and hertz
ms = 1e-3 * s
note1 = karplus_strong(440 * Hz) # Pluck "digitar" synth
note2 = zeros(300 * ms).append(karplus_strong(880 * Hz))
notes = (note1 + note2) * .5
sound = notes.take(int(2 * s)) # 2 seconds of a Karplus-Strong note
with AudioIO(True) as player: # True means "wait for all sounds to stop"
  player.play(sound, rate=rate)
```

See also the docstrings and the "examples" directory at the github repository for more help. Also, the huge test suite might help you understanding how the package works and how to use it.

1.5 License and auto-generated reST files

All project files, including source and documentation, are free software, under GPLv3. This is free in the sense that the source code have to be always available to you if you ask for it, as well as forks or otherwise derivative new source codes, however stated in a far more precise way by experts in that kind of law text. That's at the same time far from the technical language from engineering, maths and computer science, and more details would be beyond the needs of this document. You should find the following information in all Python (**.py*) source code files and also in all reStructuredText (**.rst*) files:

```
This file is part of AudioLazy, the signal processing Python package.
Copyright (C) 2012 Danilo de Jesus da Silva Bellini

AudioLazy is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
```

```
    the Free Software Foundation, version 3 of the License.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program. If not, see <http://www.gnu.org/licenses/>.
```

This is so also for auto-generated reStructuredText documentation files. However, besides most reStructuredText files being generated by a script, their contents aren't auto-generated. These are spread in the source code, both in reStructuredText and Python files, organized in a way that would make the same manually written documentation be used as:

- Spyder (https://code.google.com/p/spyderlib/) (Python IDE made for scientific purposes) *Rich Text* auto-documentation at its *Object inspector*. Docstrings were written in a reStructuredText syntax following its conventions for nice HTML rendering;
- Python docstrings (besides some docstring creation like what happens in StrategyDict instances, that's really the original written data in the source);
- Full documentation, thanks to Sphinx, that replaces the docstring conventions to other ones for creating in the that allows automatic conversion to:
  - HTML
  - PDF (LaTeX)
  - ePUB
  - Manual pages (man)
  - Texinfo
  - Pure text files

License is the same in all files that generates those documentations. Some reStructuredText files, like the README.rst that generated this whole chapter, were created manually. They're also free software as described in GPLv3. The main project repository includes a message:

```
Copyright (C) 2012 Danilo de Jesus da Silva Bellini
danilo [dot] bellini [at] gmail [dot] com
License is GPLv3. See COPYING.txt for more details.
```

This should be applied to all files that belongs to the AudioLazy project. Although all the project files were up to now created and modified by a sole person, this sole person had never wanted to keep such status for so long. If you found a bug or otherwise have an issue or a patch to send, show the issue or the pull request at the main AudioLazy repository (https://github.com/danilobellini/audiolazy), so that the bug would be fixed, or the new resource become available, not only for a few people but for everyone.

CHAPTER

TWO

MODULES DOCUMENTATION

Below is the table of contents, with processed data from docstrings. They were made and processed in a way that would be helpful as a stand-alone documentation, but if it's your first time with this package, you should see at least the *Getting started* (page 5) before these, since the full module documentation isn't written for beginners.

2.1 `audiolazy` Package

AudioLazy package

This is the main package file, that already imports all modules into the system. As the full name might not be small enough for typing it everywhere, you can import with a helpful alias:

```
>>> import audiolazy as lz
>>> lz.Stream(1, 3, 2).take(8)
[1, 3, 2, 1, 3, 2, 1, 3]
```

But there's some parts of the code you probably will find it cleaner to import directly, like the `z` object:

```
>>> from audiolazy import z, Stream
>>> filt = 1 / (1 - z ** -1) # Accumulator linear filter
>>> filt(Stream(1, 3, 2), zero=0).take(8)
[1, 4, 6, 7, 10, 12, 13, 16]
```

For a single use within a console or for trying some new experimental ideas (perhaps with IPython), you would perhaps find easier to import the full package contents:

```
>>> from audiolazy import *
>>> s, Hz = sHz(44100)
>>> delay_a4 = freq_to_lag(440 * Hz)
>>> filt = ParallelFilter(comb.tau(delay_a4, 20 * s),
...                       resonator(440 * Hz, bandwidth=100 * Hz)
...                      )
>>> len(filt)
2
```

There's documentation inside the package classes and functions docstrings. If you try `dir(audiolazy)` [or `dir(lz)`] after importing it [with the suggested alias], you'll see all the package contents, and the names starting with `lazy` followed by an underscore are modules. If you're starting now, try to see the docstring from the Stream and ZFilter classes with the `help(lz.Stream)` and `help(lz.ZFilter)` commands, and then the help from the other functionalities used above. If you didn't know the `dir` and `help` built-ins before reading this, it's strongly suggested you to read first a Python documentation or tutorial, at least enough for you to understand the basic behaviour and syntax of `for` loops, iterators, iterables, lists, generators, list comprehensions and decorators.

This package was created by Danilo J. S. Bellini and is a free software, under the terms of the GPLv3.

Summary of package modules:

Module	Description
lazy_analysis	Audio analysis and block processing module
lazy_auditory	Peripheral auditory modeling module
lazy_core	Core classes module
lazy_filters	Stream filtering module
lazy_io	Audio recording input and playing output module
lazy_itertools	Itertools module "decorated" replica, where all outputs are Stream instances
lazy_lpc	Linear Predictive Coding (LPC) module
lazy_math	Math modules "decorated" and complemented to work elementwise when needed
lazy_midi	MIDI representation data & note-frequency relationship
lazy_misc	Common miscellanous tools and constants for general use
lazy_poly	Polynomial model
lazy_stream	Stream class definition module
lazy_synth	Simple audio/stream synthesis module

2.2 `lazy_analysis` Module

Audio analysis and block processing module

Summary of module contents:

Name	Description
window	This is a StrategyDict instance object called `window`. Strategies stored: 6.
acorr	Calculate the autocorrelation of a given 1-D block sequence.
lag_matrix	Finds the lag matrix for a given 1-D block sequence.
dft	Complex non-optimized Discrete Fourier Transform
zcross	Zero-crossing stream.
envelope	This is a StrategyDict instance object called `envelope`. Strategies stored: 3.
maverage	This is a StrategyDict instance object called `maverage`. Strategies stored: 3.
clip	Clips the signal up to both a lower and a higher limit.
unwrap	Parametrized signal unwrapping.
freq_to_lag	Converts between frequency (rad/sample) and lag (number of samples).
lag_to_freq	Converts between frequency (rad/sample) and lag (number of samples).
amdf	Average Magnitude Difference Function non-linear filter for a given size and a fixed lag.

acorr(*blk*, *max_lag=None*)

Calculate the autocorrelation of a given 1-D block sequence.

Parameters

- **blk** – An iterable with well-defined length. Don't use this function with Stream objects!
- **max_lag** – The size of the result, the lags you'd need. Defaults to `len(blk) - 1`, since any lag beyond would result in zero.

Returns A list with lags from 0 up to max_lag, where its `i`-th element has the autocorrelation for a lag equals to `i`. Be careful with negative lags! You should use abs(lag) indexes when working with them.

Examples:

```
>>> seq = [1, 2, 3, 4, 3, 4, 2]
>>> acorr(seq) # Default max_lag is len(seq) - 1
[59, 52, 42, 30, 17, 8, 2]
>>> acorr(seq, 9) # Zeros at the end
[59, 52, 42, 30, 17, 8, 2, 0, 0, 0]
>>> len(acorr(seq, 3)) # Resulting length is max_lag + 1
4
```

```
>>> acorr(seq, 3)
[59, 52, 42, 30]
```

lag_matrix(*blk*, *max_lag=None*)
Finds the lag matrix for a given 1-D block sequence.

Parameters

- **blk** – An iterable with well-defined length. Don't use this function with Stream objects!
- **max_lag** – The size of the result, the lags you'd need. Defaults to `len(blk) - 1`, the maximum lag that doesn't create fully zeroed matrices.

Returns The covariance matrix as a list of lists. Each cell (i, j) contains the sum of `blk[n - i] * blk[n - j]` elements for all n that allows such without padding the given block.

dft(*blk*, *freqs*, *normalize=True*)
Complex non-optimized Discrete Fourier Transform

Finds the DFT for values in a given frequency list, in order, over the data block seen as periodic.

Parameters

- **blk** – An iterable with well-defined length. Don't use this function with Stream objects!
- **freqs** – List of frequencies to find the DFT, in rad/sample. FFT implementations like numpy.fft.ftt finds the coefficients for N frequencies equally spaced as `line(N, 0, 2 * pi, finish=False)` for N frequencies.
- **normalize** – If True (default), the coefficient sums are divided by `len(blk)`, and the coefficient for the DC level (frequency equals to zero) is the mean of the block. If False, that coefficient would be the sum of the data in the block.

Returns A list of DFT values for each frequency, in the same order that they appear in the freqs input.

Note: This isn't a FFT implementation, and performs $O(M.N)$ float pointing operations, with M and N equals to the length of the inputs. This function can find the DFT for any specific frequency, with no need for zero padding or finding all frequencies in a linearly spaced band grid with N frequency bins at once.

---

zcross(**args*, ***kwargs*)
Zero-crossing stream.

Parameters

- **seq** – Any iterable to be used as input for the zero crossing analysis
- **hysteresis** – Crossing exactly zero might happen many times too fast due to high frequency oscilations near zero. To avoid this, you can make two threshold limits for the zero crossing detection: `hysteresis` and `-hysteresis`. Defaults to zero (0), which means no hysteresis and only one threshold.
- **first_sign** – Optional argument with the sign memory from past. Gets the sig from any signed number. Defaults to zero (0), which means "any", and the first sign will be the first one found in data.

Returns A Stream instance that outputs 1 for each crossing detected, 0 otherwise.

clip(*sig*, *low=-1.0*, *high=1.0*)
Clips the signal up to both a lower and a higher limit.

Parameters

- **sig** – The signal to be clipped, be it a Stream instance, a list or any iterable.
- **[low** – ...

- **high]** – Lower and higher clipping limit, "saturating" the input to them. Defaults to -1.0 and 1.0, respectively. These can be None when needed one-sided clipping. When both limits are set to None, the output will be a Stream that yields exactly the `sig` input data.

Returns Clipped signal as a Stream instance.

unwrap (**args*, ***kwargs*)
Parametrized signal unwrapping.

Parameters

- **sig** – An iterable seen as an input signal.
- **max_delta** – Maximum value of $\Delta = sig_i - sig_{i-1}$ to keep output without another minimizing step change. Defaults to π.
- **step** – The change in order to minimize the delta is an integer multiple of this value. Defaults to $2.\pi$.

Returns The signal unwrapped as a Stream, minimizing the step difference when any adjacency step in the input signal is higher than `max_delta` by summing/subtracting `step`.

freq_to_lag (*x*)
Converts between frequency (rad/sample) and lag (number of samples).

lag_to_freq (*x*)
Converts between frequency (rad/sample) and lag (number of samples).

amdf (*lag*, *size*)
Average Magnitude Difference Function non-linear filter for a given size and a fixed lag.

Parameters

- **lag** – Time lag, in samples. See `freq_to_lag` if needs conversion from frequency values.
- **size** – Moving average size.

Returns A callable that accepts two parameters: a signal `sig` and the starting memory element `zero` that behaves like the `LinearFilter.__call__` arguments. The output from that callable is a Stream instance, and has no decimation applied.

See Also:

`freq_to_lag` **(page 14)** Frequency to lag and lag to frequency converter.

2.2.1 `lazy_analysis.window` StrategyDict

This is a StrategyDict instance object called `window`. Strategies stored: 6.

Strategy window.bartlett. Docstring starts with:

Bartlett (triangular with zero-valued endpoints) window function with the given size.

Strategy window.blackman. Docstring starts with:

Blackman window function with the given size.

Strategy window.hamming (Default). Docstring starts with:

Hamming window function with the given size.

Strategy window.hann. An alias for it is `window.hanning`. Docstring starts with:

Hann window function with the given size.

Strategy window.rectangular. An alias for it is `window.rect`. Docstring starts with:

Rectangular window function with the given size.

Strategy window.triangular. An alias for it is `window.triangle`. Docstring starts with:

Triangular (with no zero end-point) window function with the given size.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

hamming (*size*)
Hamming window function with the given size.

Returns List with the desired window samples. Max value is one (1.0).

hann (*size*)
Hann window function with the given size.

Returns List with the desired window samples. Max value is one (1.0).

blackman (*size*, *alpha=0.16*)
Blackman window function with the given size.

Parameters

- **size** – Window size in samples.
- **alpha** – Blackman window alpha value. Defaults to 0.16.

Returns List with the desired window samples. Max value is one (1.0).

triangular (*size*)
Triangular (with no zero end-point) window function with the given size.

Returns List with the desired window samples. Max value is one (1.0).

See Also:

window.bartlett (page 15) Bartlett window, triangular with zero-valued end-points.

bartlett (*size*)
Bartlett (triangular with zero-valued endpoints) window function with the given size.

Returns List with the desired window samples. Max value is one (1.0).

See Also:

window.triangular (page 15) Triangular with no zero end-point.

rectangular (*size*)
Rectangular window function with the given size.

Returns List with the desired window samples. All values are ones (1.0).

2.2.2 lazy_analysis.envelope StrategyDict

This is a StrategyDict instance object called `envelope`. Strategies stored: 3.

Strategy envelope.abs. Docstring starts with:

Envelope non-linear filter.

Strategy envelope.rms (Default). Docstring starts with:

Envelope non-linear filter.

Strategy envelope.squared. Docstring starts with:

Squared envelope non-linear filter.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

abs(*sig*, *cutoff=0.006135923151542565*)
Envelope non-linear filter.

This strategy make an ideal half wave rectification (get the absolute value of each signal) and pass the resulting data through a low pass filter.

Parameters

- **sig** – The signal to be filtered.
- **cutoff** – Lowpass filter cutoff frequency, in rad/sample. Defaults to `pi/512`.

Returns A Stream instance with the envelope, without any decimation.

See Also:

`maverage` **(page 16)** Moving average linear filter.

rms(*sig*, *cutoff=0.006135923151542565*)
Envelope non-linear filter.

This strategy finds a RMS by passing the squared data through a low pass filter and taking its square root afterwards.

Parameters

- **sig** – The signal to be filtered.
- **cutoff** – Lowpass filter cutoff frequency, in rad/sample. Defaults to `pi/512`.

Returns A Stream instance with the envelope, without any decimation.

See Also:

`maverage` **(page 16)** Moving average linear filter.

squared(*sig*, *cutoff=0.006135923151542565*)
Squared envelope non-linear filter.

This strategy squares the input, and apply a low pass filter afterwards.

Parameters

- **sig** – The signal to be filtered.
- **cutoff** – Lowpass filter cutoff frequency, in rad/sample. Defaults to `pi/512`.

Returns A Stream instance with the envelope, without any decimation.

See Also:

`maverage` **(page 16)** Moving average linear filter.

2.2.3 `lazy_analysis.maverage` StrategyDict

This is a StrategyDict instance object called `maverage`. Strategies stored: 3.

Strategy maverage.deque (Default). Docstring starts with:

Moving average

Strategy maverage.fir. Docstring starts with:

Moving average

Strategy maverage.recursive. An alias for it is `maverage.feedback`. Docstring starts with:

Moving average

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

fir(*size*)

Moving average

Linear filter implementation as a FIR ZFilter.

Parameters **size** – Data block window size. Should be an integer.

Returns A ZFilter instance with the FIR filter.

See Also:

envelope (page 15) Signal envelope (time domain) strategies.

recursive(*size*)

Moving average

Linear filter implementation as a recursive / feedback ZFilter.

Parameters **size** – Data block window size. Should be an integer.

Returns A ZFilter instance with the feedback filter.

See Also:

envelope (page 15) Signal envelope (time domain) strategies.

deque(*size*)

Moving average

This is the only strategy that uses a `collections.deque` object instead of a ZFilter instance. Fast, but without extra capabilites such as a frequency response plotting method.

Parameters **size** – Data block window size. Should be an integer.

Returns A callable that accepts two parameters: a signal `sig` and the starting memory element `zero` that behaves like the `LinearFilter.__call__` arguments. The output from that callable is a Stream instance, and has no decimation applied.

See Also:

envelope (page 15) Signal envelope (time domain) strategies.

2.3 `lazy_auditory` Module

Peripheral auditory modeling module

Summary of module contents:

Name	Description
erb	This is a StrategyDict instance object called `erb`. Strategies stored: 2.
gammatone	This is a StrategyDict instance object called `gammatone`. Strategies stored: 3.
gammatone_erb_constants	Constants for using the real bandwidth in the gammatone filter, given its order. Returns a pair $(x, y) = (1/a_n, c_n)$.

gammatone_erb_constants(*n*)

Constants for using the real bandwidth in the gammatone filter, given its order. Returns a pair $(x, y) = (1/a_n, c_n)$.

Based on equations from:

```
Holdsworth, J.; Patterson, R.; Nimmo-Smith, I.; Rice, P.
Implementing a GammaTone Filter Bank.  In:  SVOS Final Report,
Annex C, Part A: The Auditory Filter Bank.  1988.
```

First returned value is a bandwidth compensation for direct use in the gammatone formula:

```
>>> x, y = gammatone_erb_constants(4)
>>> central_frequency = 1000
>>> round(x, 3)
1.019
>>> bandwidth = x * erb["moore_glasberg_83"](central_frequency)
>>> round(bandwidth, 2)
130.52
```

Second returned value helps us find the `3 dB` bandwidth as:

```
>>> x, y = gammatone_erb_constants(4)
>>> central_frequency = 1000
>>> bandwidth3dB = x * y * erb["moore_glasberg_83"](central_frequency)
>>> round(bandwidth3dB, 2)
113.55
```

2.3.1 `lazy_auditory.erb` StrategyDict

This is a StrategyDict instance object called `erb`. Strategies stored: 2.

Strategy erb.gm90 (Default). Aliases available are `erb.glasberg_moore_90`, `erb.glasberg_moore`. Docstring starts with:

ERB model from Glasberg and Moore in 1990.

Strategy erb.mg83. An alias for it is `erb.moore_glasberg_83`. Docstring starts with:

ERB model from Moore and Glasberg in 1983.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

mg83(**args*, ***kwargs*)

ERB model from Moore and Glasberg in 1983.

```
B. C. J. Moore and B. R. Glasberg, "Suggested formulae
for calculating auditory filter bandwidths and excitation
patterns".  J. Acoust.  Soc.  Am., 74, 1983, pp.  750-753.
```

Parameters

- **freq** – Frequency, in rad/sample if second parameter is given, in Hz otherwise.

- **Hz** – Frequency conversion "Hz" from sHz function, i.e., `sHz(rate)[1]`. If this value is not given, both input and output will be in Hz.

Returns Frequency range size, in rad/sample if second parameter is given, in Hz otherwise.

gm90 (**args*, ***kwargs*)
ERB model from Glasberg and Moore in 1990.

```
B. R. Glasberg and B. C. J. Moore, "Derivation of auditory
filter shapes from notched-noise data".  Hearing Research, vol.
47, 1990, pp.  103-108.
```

Parameters

- **freq** – Frequency, in rad/sample if second parameter is given, in Hz otherwise.
- **Hz** – Frequency conversion "Hz" from sHz function, i.e., `sHz(rate)[1]`. If this value is not given, both input and output will be in Hz.

Returns Frequency range size, in rad/sample if second parameter is given, in Hz otherwise.

2.3.2 `lazy_auditory.gammatone` StrategyDict

This is a StrategyDict instance object called `gammatone`. Strategies stored: 3.

Strategy gammatone.klapuri. Docstring starts with:

Gammatone filter based on Anssi Klapuri's IIR cascading filter model.

Strategy gammatone.sampled (Default). Docstring starts with:

Gammatone filter based on a sampled impulse response.

Strategy gammatone.slaney. Docstring starts with:

Gammatone filter based on Malcolm Slaney's IIR cascading filter model.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

klapuri (*freq*, *bandwidth*)
Gammatone filter based on Anssi Klapuri's IIR cascading filter model.

Model is described in:

```
A. Klapuri, "Multipich Analysis of Polyphonic Music and Speech
Signals Using an Auditory Model".  IEEE Transactions on Audio,
Speech and Language Processing, vol.  16, no.  2, 2008, pp.
255-266.
```

Parameters

- **freq** – Frequency, in rad/sample.
- **bandwidth** – Frequency range size, in rad/sample. See gammatone_erb_constants for more information about how you can find this.

Returns A CascadeFilter object with ZFilter filters, each of them a pole-conjugated IIR filter model. Gain is normalized to have peak with 0 dB (1.0 amplitude). The total number of poles is twice the value of eta (conjugated pairs), one pair for each ZFilter.

sampled (*freq*, *bandwidth*, *phase=0*, *eta=4*)
Gammatone filter based on a sampled impulse response.

```
n ** (eta - 1) * exp(-bandwidth * n) * cos(freq * n + phase)
```

Parameters

- **freq** – Frequency, in rad/sample.
- **bandwidth** – Frequency range size, in rad/sample. See gammatone_erb_constants for more information about how you can find this.
- **phase** – Phase, in radians. Defaults to zero (cosine).
- **eta** – Gammatone filter order. Defaults to 4.

Returns A CascadeFilter object with ZFilter filters, each of them a pole-conjugated IIR filter model. Gain is normalized to have peak with 0 dB (1.0 amplitude). The total number of poles is twice the value of eta (conjugated pairs), one pair for each ZFilter.

slaney (*freq*, *bandwidth*)

Gammatone filter based on Malcolm Slaney's IIR cascading filter model.

Model is described in:

```
Slaney, M. "An Efficient Implementation of the
Patterson-Holdsworth Auditory Filter Bank", Apple Computer
Technical Report #35, 1993.
```

Parameters

- **freq** – Frequency, in rad/sample.
- **bandwidth** – Frequency range size, in rad/sample. See gammatone_erb_constants for more information about how you can find this.

Returns A CascadeFilter object with ZFilter filters, each of them a pole-conjugated IIR filter model. Gain is normalized to have peak with 0 dB (1.0 amplitude). The total number of poles is twice the value of eta (conjugated pairs), one pair for each ZFilter.

2.4 lazy_core Module

Core classes module

Summary of module contents:

Name	Description
AbstractOperatorOverloaderMeta	Abstract metaclass for classes with massively overloaded operators.
MultiKeyDict	Multiple keys dict.
StrategyDict	Strategy dictionary manager creator with default, mainly done for callables and multiple implementation algorithms / models.

class AbstractOperatorOverloaderMeta

Bases: abc.ABCMeta

Abstract metaclass for classes with massively overloaded operators.

Dunders dont't appear within "getattr" nor "getattribute", and they should be inside the class dictionary, not the class instance one, otherwise they won't be found by the usual mechanism. That's why we have to be eager here. You need a concrete class inherited from this one, and the "abstract" enforcement and specification is:

- You have to override __operators__ with a string, given the operators to be used by its dunder name "without the dunders" (i.e., "__add__" should be written as "add"), all in a single string separated by spaces and including reversed operators, like "add radd sub rsub eq". Its a good idea to tell all operators that will be used, since the metaclass will enforce their existance.

- All operators should be implemented by the metaclass hierarchy or by the class directly, and the class has priority when both exists, neglecting the template in this case.
- There are three templates: __binary__, __rbinary__, __unary__, all receives 2 parameters (the class being instantiated and the operator function) and should return a function for the specific dunder.

__binary__(*op_func*)
This method should be overridden to return the dunder for the given operator function.

static __new__(*mcls*, *name*, *bases*, *namespace*)

__operators__
Should be overridden by a string with all operator names to be overloaded. Reversed operators should be given explicitly.

__rbinary__(*op_func*)
This method should be overridden to return the dunder for the given operator function.

__unary__(*op_func*)
This method should be overridden to return the dunder for the given operator function.

new_dunder(*op_func*, *is_reversed*, *ninputs*)
Insert a new dunder (double underscore method) to the class.

Parameters

- **op_func** – A function from the operator module.
- **is_reversed** – If true, reverses the operands order. For example, it means that besides op_func being perhaps `operator.add`, what is being asked as return value is a `__radd__` method dunder, as a function.
- **ninputs** – An int (1 or 2) telling if is the dunder is `unary(self)` or `binary(self, other)`, respectively.

Returns A function to be used as a dunder for the given operator.

Note: Commonly you don't need to worry with this, since this is done by the `__new__` constructor when you implement the operator dunder templates as said in the class docstring.

---

unary = 'invert'

class MultiKeyDict(**args*, ***kwargs*)
Bases: `dict`

Multiple keys dict.

Can be thought as an "inversible" dict where you can ask for the one hashable value from one of the keys. By default it iterates through the values, if you need an iterator for all tuples of keys, use iterkeys method instead.

Examples: Assignments one by one:

```
>>> mk = MultiKeyDict()
>>> mk[1] = 3
>>> mk[2] = 3
>>> mk
{(1, 2): 3}
>>> mk[4] = 2
>>> mk[1] = 2
>>> len(mk)
2
>>> mk[1]
2
>>> mk[2]
3
>>> mk[4]
```

```
2
>>> sorted(mk)
[2, 3]
>>> sorted(mk.iterkeys())
[(2,), (4, 1)]
```

Casting from another dict:

```
>>> mkd = MultiKeyDict({1:4, 2:5, -7:4})
>>> len(mkd)
2
>>> sorted(mkd)
[4, 5]
>>> del mkd[2]
>>> len(mkd)
1
>>> sorted(mkd.keys()[0]) # Sorts the only key tuple
[-7, 1]
```

__delitem__(*key*)

__getitem__(*key*)

__init__(**args*, ***kwargs*)

__iter__()

__setitem__(*key*, *value*)

class StrategyDict(**args*, ***kwargs*)

Bases: audiolazy.lazy_core.MultiKeyDict (page 21)

Strategy dictionary manager creator with default, mainly done for callables and multiple implementation algorithms / models.

Each strategy might have multiple names. The names can be any hashable. The "strategy" method creates a decorator for the given strategy names. Default is the first strategy you insert, but can be changed afterwards. The default strategy is the attribute StrategyDict.default, and might be anything outside the dictionary (i.e., it won't be changed if you remove the strategy).

It iterates through the values (i.e., for each strategy, not its name)

Examples:

```
>>> sd = StrategyDict()
>>> @sd.strategy("sum") # First strategy is default
... def sd(a, b, c):
...     return a + b + c
>>> @sd.strategy("min", "m") # Multiple names
... def sd(a, b, c):
...     return min(a, b, c)
>>> sd(2, 5, 0)
7
>>> sd["min"](2, 5, 0)
0
>>> sd["m"](7, -5, -2)
-5
>>> sd.default = sd["min"]
>>> sd(-19, 1e18, 0)
-19
```

Note: The StrategyDict constructor creates a new class inheriting from StrategyDict, and then instantiates it before returning the requested instance. This singleton subclassing is needed for docstring personalization.

__call__(**args*, ***kwargs*)

__getattr__(*name*)

__iter__()

static __new__(*name='strategy_dict_unnamed_instance'*)
Creates a new StrategyDict class and returns an instance of it. The new class is needed to ensure it'll have a personalized docstring.

__setitem__(*key*, *value*)

default()

strategy(**names*)

2.5 `lazy_filters` Module

Stream filtering module

Summary of module contents:

Name	Description
LinearFilterProperties	Class with common properties in a linear filter that can be used as a mixin.
LinearFilter	Base class for Linear filters, time invariant or not.
ZFilterMeta	* * * * ...no docstring... * * * *
ZFilter	Linear filters based on Z-transform frequency domain equations.
z	z
FilterListMeta	* * * * ...no docstring... * * * *
FilterList	Class from which CascadeFilter and ParallelFilter inherits the common part of their contents. You probably won't need to use this directly.
CascadeFilter	Filter cascade as a list of filters.
ParallelFilter	Filters in parallel as a list of filters.
comb	This is a StrategyDict instance object called `comb`. Strategies stored: 3.
resonator	This is a StrategyDict instance object called `resonator`. Strategies stored: 4.
lowpass	This is a StrategyDict instance object called `lowpass`. Strategies stored: 2.
highpass	This is a StrategyDict instance object called `highpass`. Strategies stored: 1.

class LinearFilterProperties
Bases: `object`

Class with common properties in a linear filter that can be used as a mixin.

The classes that inherits this one should implement the `numpoly` and `denpoly` properties, and these should return a Poly instance.

dendict

denlist

denominator

denpolyz
Like denpoly, the linear filter denominator (or backward coefficients) as a Poly instance based on `x = z` instead of denpoly's `x = z ** -1`, useful for taking roots.

numdict

numerator

numlist

numpolyz
Like numpoly, the linear filter numerator (or forward coefficients) as a Poly instance based on `x = z` instead of numpoly's `x = z ** -1`, useful for taking roots.

class `LinearFilter`(*numerator=None*, *denominator=None*)

Bases: `audiolazy.lazy_filters.LinearFilterProperties` (page 23)

Base class for Linear filters, time invariant or not.

`__call__`(*seq*, *memory=None*, *zero=0.0*)

IIR, FIR and time variant linear filtering.

Parameters

- **seq** – Any iterable to be seem as the input stream for the filter.
- **memory** – Might be an iterable or a callable. Generally, as a iterable, the first needed elements from this input will be used directly as the memory (not the last ones!), and as a callable, it will be called with the size as the only positional argument, and should return an iterable. If `None` (default), memory is initialized with zeros.
- **zero** – Value to fill the memory, when needed, and to be seem as previous input when there's a delay. Defaults to `0.0`.

Returns A Stream that have the data from the input sequence filtered.

`__eq__`(*other*)

`__init__`(*numerator=None*, *denominator=None*)

`__iter__`()

`__ne__`(*other*)

`freq_response`(**args*, ***kwargs*)

Frequency response for this filter.

Parameters **freq** – Frequency, in rad/sample. Can be an iterable with frequencies.

Returns Complex number with the frequency response of the filter.

See Also:

`dB10` **(page 44)** Logarithmic power magnitude from data with squared magnitude.

`dB20` **(page 44)** Logarithmic power magnitude from raw complex data or data with linear amplitude.

`phase` **(page 44)** Phase from complex data.

`is_causal`()

Causality test for this filter.

Returns Boolean returning True if this filter is causal, False otherwise.

`is_lti`()

Test if this filter is LTI (Linear Time Invariant).

Returns Boolean returning True if this filter is LTI, False otherwise.

`linearize`()

Linear interpolation of fractional delay values.

Returns A new linear filter, with the linearized delay values.

Examples:

```
>>> filt = z ** -4.3
>>> filt.linearize()
0.7 * z^-4 + 0.3 * z^-5
```

plot (*fig=None*, *samples=2048*, *rate=None*, *min_freq=0.0*, *max_freq=3.141592653589793*, *blk=None*, *unwrap=True*, *freq_scale='linear'*, *mag_scale='dB'*)

Plots the filter frequency response into a formatted MatPlotLib figure with two subplots, labels and title, including the magnitude response and the phase response.

Parameters

- **fig** – A matplotlib.figure.Figure instance. Defaults to None, which means that it will create a new figure.
- **samples** – Number of samples (frequency values) to plot. Defaults to 2048.
- **rate** – Given rate (samples/second) or "s" object from `sHz`. Defaults to 300.
- **[min_freq** – ...
- **max_freq]** – Frequency range to be drawn, in rad/sample. Defaults to [0, pi].
- **blk** – Sequence block. Plots the block DFT together with the filter frequency. Defaults to None (no block).
- **unwrap** – Boolean that chooses whether should unwrap the data phase or keep it as is. Defaults to True.
- **freq_scale** – Chooses whether plot is "linear" or "log" with respect to the frequency axis. Defaults to "linear". Case insensitive.
- **mag_scale** – Chooses whether magnitude plot scale should be "linear", "squared" or "dB". Defaults do "dB". Case insensitive.

Returns The matplotlib.figure.Figure instance.

See Also:

sHz (page 51) Second and hertz constants from samples/second rate.

LinearFilter.zplot (page 25) Zeros-poles diagram plotting.

poles

Returns a list with all poles (denominator roots in `z`). Needs Numpy.

See Also:

LinearFilterProperties.numpoly Numerator polynomials where *x* is `z ** -1`.

LinearFilterProperties.denpoly Denominator polynomials where *x* is `z ** -1`.

LinearFilterProperties.numpolyz (page 23) Numerator polynomials where *x* is `z`.

LinearFilterProperties.denpolyz (page 23) Denominator polynomials where *x* is `z`.

zeros

Returns a list with all zeros (numerator roots in `z`), besides the zero-valued "zeros" that might arise from the difference between the numerator and denominator order (i.e., the roots returned are the inverse from the `numpoly.roots()` in `z ** -1`). Needs Numpy.

See Also:

LinearFilterProperties.numpoly Numerator polynomials where *x* is `z ** -1`.

LinearFilterProperties.denpoly Denominator polynomials where *x* is `z ** -1`.

LinearFilterProperties.numpolyz (page 23) Numerator polynomials where *x* is `z`.

LinearFilterProperties.denpolyz (page 23) Denominator polynomials where *x* is `z`.

zplot (*fig=None*, *circle=True*)

Plots the filter zero-pole plane into a formatted MatPlotLib figure with one subplot, labels and title.

Parameters

- **fig** – A matplotlib.figure.Figure instance. Defaults to None, which means that it will create a new figure.
- **circle** – Chooses whether to include the unit circle in the plot. Defaults to True.

Returns The matplotlib.figure.Figure instance.

Note: Multiple roots detection is slow, and roots may suffer from numerical errors (e.g., filter `f = 1 - 2 * z ** -1 + 1 * z ** -2` has twice the root `1`, but `f ** 3` suffer noise from the root finding algorithm). For the exact number of poles and zeros, see the result title, or the length of LinearFilter.poles() and LinearFilter.zeros().

See Also:

`LinearFilter.plot` (page 24) Frequency response plotting. Needs MatPlotLib.

`LinearFilter.zeros` (page 25), `LinearFilter.poles` (page 25) Filter zeros and poles, as a list. Needs NumPy.

class `ZFilterMeta`

Bases: `audiolazy.lazy_core.AbstractOperatorOverloaderMeta` (page 20)

`__operators__` = 'pos neg add radd sub rsub mul rmul div rdiv truediv rtruediv pow '

`__rbinary__`(*op_func*)

`__unary__`(*op_func*)

class `ZFilter`(*numerator=None*, *denominator=None*)

Bases: `audiolazy.lazy_filters.LinearFilter` (page 23)

Linear filters based on Z-transform frequency domain equations.

Examples: Using the `z` object (float output because default filter memory has float zeros, and the delay in the numerator creates another float zero as "pre-input"):

```
>>> filt = (1 + z ** -1) / (1 - z ** -1)
>>> data = [1, 5, -4, -7, 9]
>>> stream_result = filt(data) # Lazy iterable
>>> list(stream_result) # Freeze
[1.0, 7.0, 8.0, -3.0, -1.0]
```

Same example with the same filter, but with a memory input, and using lists for filter numerator and denominator instead of the `z` object:

```
>>> b = [1, 1]
>>> a = [1, -1] # Each index ''i'' has the coefficient for z ** -i
>>> filt = ZFilter(b, a)
>>> data = [1, 5, -4, -7, 9]
>>> stream_result = filt(data, memory=[3], zero=0) # Lazy iterable
>>> result = list(stream_result) # Freeze
>>> result
[4, 10, 11, 0, 2]
>>> filt2 = filt * z ** -1 # You can add a delay afterwards easily
>>> final_result = filt2(result, zero=0)
>>> list(final_result)
[0, 4, 18, 39, 50]
```

`__add__`(*other*)

`__call__`(*seq*, *memory=None*, *zero=0.0*)

IIR, FIR and time variant linear filtering.

Parameters

- **seq** – Any iterable to be seem as the input stream for the filter, or another ZFilter for substituition.
- **memory** – Might be an iterable or a callable. Generally, as a iterable, the first needed elements from this input will be used directly as the memory (not the last ones!), and as a callable, it will be called with the size as the only positional argument, and should return an iterable. If `None` (default), memory is initialized with zeros. Neglect when `seq` input is a ZFilter.
- **zero** – Value to fill the memory, when needed, and to be seem as previous input when there's a delay. Defaults to `0.0`. Neglect when `seq` input is a ZFilter.

Returns A Stream that have the data from the input sequence filtered.

Examples: With ZFilter instances:

```
>>> filt = 1 + z ** -1
>>> filt(z ** -1)
z + 1
>>> filt(- z ** 2)
1 - z^-2
```

With any iterable (but ZFilter instances):

```
>>> filt = 1 + z ** -1
>>> data = filt([1.0, 2.0, 3.0])
>>> data
<audiolazy.lazy_stream.Stream object at ...>
>>> list(data)
[1.0, 3.0, 5.0]
```

__div__(*other*)

__metaclass__
alias of `ZFilterMeta` (page 26)

__mul__(*other*)

__neg__()

__pos__()

__pow__(*other*)

__radd__(*other*)

__rdiv__(*other*)

__repr__()

__rmul__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__str__()

__sub__(*other*)

__truediv__(*other*)

diff(*n=1*, *mul_after=1*)
Takes n-th derivative, multiplying each m-th derivative filter by mul_after before taking next (m+1)-th derivative or returning.

class FilterListMeta
Bases: `audiolazy.lazy_core.AbstractOperatorOverloaderMeta` (page 20)

__binary__(*op_func*)
Binary dunder factory.

__operators__ = 'add mul rmul lt le gt ge'

__rbinary__(*op_func*)
Binary dunder factory.

class FilterList(**filters*)
Bases: `list`, `audiolazy.lazy_filters.LinearFilterProperties` (page 23)

Class from which CascadeFilter and ParallelFilter inherits the common part of their contents. You probably won't need to use this directly.

__add__(*other*)
This operator acts just like it would do with lists.

__eq__(*other*)

__ge__(*other*)
This operator acts just like it would do with lists.

__gt__(*other*)
This operator acts just like it would do with lists.

__init__(**filters*)

__le__(*other*)
This operator acts just like it would do with lists.

__lt__(*other*)
This operator acts just like it would do with lists.

__metaclass__
alias of `FilterListMeta` (page 27)

__mul__(*other*)
This operator acts just like it would do with lists.

__ne__(*other*)

__rmul__(*other*)
This operator acts just like it would do with lists.

callables
List of callables with all filters, casting to LinearFilter each one that isn't callable.

is_causal()
Tests whether all filters in the list are causal (i.e., no future-data delay in positive `z` exponents). Non-linear filters are seem as causal by default. CascadeFilter and ParallelFilter are causal if all the filters they group are causal.

is_linear()
Tests whether all filters in the list are linear. CascadeFilter and ParallelFilter instances are also linear if all filters they group are linear.

is_lti()
Tests whether all filters in the list are linear time invariant (LTI). CascadeFilter and ParallelFilter instances are also LTI if all filters they group are LTI.

plot(*fig=None*, *samples=2048*, *rate=None*, *min_freq=0.0*, *max_freq=3.141592653589793*, *blk=None*, *unwrap=True*, *freq_scale='linear'*, *mag_scale='dB'*)
Plots the filter frequency response into a formatted MatPlotLib figure with two subplots, labels and title, including the magnitude response and the phase response.

Parameters

- **fig** – A matplotlib.figure.Figure instance. Defaults to None, which means that it will create a new figure.

- **samples** – Number of samples (frequency values) to plot. Defaults to 2048.
- **rate** – Given rate (samples/second) or "s" object from `sHz`. Defaults to 300.
- **[min_freq** – ...
- **max_freq]** – Frequency range to be drawn, in rad/sample. Defaults to [0, pi].
- **blk** – Sequence block. Plots the block DFT together with the filter frequency. Defaults to None (no block).
- **unwrap** – Boolean that chooses whether should unwrap the data phase or keep it as is. Defaults to True.
- **freq_scale** – Chooses whether plot is "linear" or "log" with respect to the frequency axis. Defaults to "linear". Case insensitive.
- **mag_scale** – Chooses whether magnitude plot scale should be "linear", "squared" or "dB". Defaults do "dB". Case insensitive.

Returns The matplotlib.figure.Figure instance.

See Also:

sHz (page 51) Second and hertz constants from samples/second rate.

LinearFilter.zplot (page 25) Zeros-poles diagram plotting.

zplot (*fig=None*, *circle=True*)

Plots the filter zero-pole plane into a formatted MatPlotLib figure with one subplot, labels and title.

Parameters

- **fig** – A matplotlib.figure.Figure instance. Defaults to None, which means that it will create a new figure.
- **circle** – Chooses whether to include the unit circle in the plot. Defaults to True.

Returns The matplotlib.figure.Figure instance.

Note: Multiple roots detection is slow, and roots may suffer from numerical errors (e.g., filter `f = 1 - 2 * z ** -1 + 1 * z ** -2` has twice the root `1`, but `f ** 3` suffer noise from the root finding algorithm). For the exact number of poles and zeros, see the result title, or the length of LinearFilter.poles() and LinearFilter.zeros().

See Also:

LinearFilter.plot (page 24) Frequency response plotting. Needs MatPlotLib.

LinearFilter.zeros (page 25), LinearFilter.poles (page 25) Filter zeros and poles, as a list. Needs NumPy.

class CascadeFilter (**filters*)

Bases: `audiolazy.lazy_filters.FilterList` (page 28)

Filter cascade as a list of filters.

Note: A filter is any callable that receives an iterable as input and returns a Stream.

Examples:

```
>>> filt = CascadeFilter(z ** -1, 2 * (1 - z ** -3))
>>> data = Stream(1, 3, 5, 3, 1, -1, -3, -5, -3, -1) # Endless
>>> filt(data, zero=0).take(15)
[0, 2, 6, 10, 4, -4, -12, -12, -12, -4, 4, 12, 12, 12, 4]
```

__add__(*other*)
This operator acts just like it would do with lists.

__call__(**args*, ***kwargs*)

__ge__(*other*)
This operator acts just like it would do with lists.

__gt__(*other*)
This operator acts just like it would do with lists.

__le__(*other*)
This operator acts just like it would do with lists.

__lt__(*other*)
This operator acts just like it would do with lists.

__mul__(*other*)
This operator acts just like it would do with lists.

__rmul__(*other*)
This operator acts just like it would do with lists.

denpoly

freq_response(**args*, ***kwargs*)

numpoly

poles

zeros

class ParallelFilter(**filters*)
Bases: audiolazy.lazy_filters.FilterList (page 28)

Filters in parallel as a list of filters.

This list of filters that behaves as a filter, returning the sum of all signals that results from applying the the same given input into all filters. Besides the name, the data processing done isn't parallel.

Note: A filter is any callable that receives an iterable as input and returns a Stream.

Examples:

```
>>> filt = 1 + z ** -1 -  z ** -2
>>> pfilt = ParallelFilter(1 + z ** -1, - z ** -2)
>>> list(filt(range(100))) == list(pfilt(range(100)))
True
>>> list(filt(range(10), zero=0))
[0, 1, 3, 4, 5, 6, 7, 8, 9, 10]
```

__add__(*other*)
This operator acts just like it would do with lists.

__call__(**args*, ***kwargs*)

__ge__(*other*)
This operator acts just like it would do with lists.

__gt__(*other*)
This operator acts just like it would do with lists.

__le__(*other*)
This operator acts just like it would do with lists.

__lt__(*other*)
This operator acts just like it would do with lists.

__mul__(*other*)
This operator acts just like it would do with lists.

__rmul__(*other*)
This operator acts just like it would do with lists.

denpoly

freq_response(**args*, ***kwargs*)

numpoly

poles

zeros

2.5.1 lazy_filters.comb StrategyDict

This is a StrategyDict instance object called `comb`. Strategies stored: 3.

Strategy comb.fb (Default). Aliases available are `comb.alpha`, `comb.fb_alpha`, `comb.feedback_alpha`. Docstring starts with:

Feedback comb filter for a given alpha (and delay).

Strategy comb.ff. Aliases available are `comb.ff_alpha`, `comb.feedforward_alpha`. Docstring starts with:

Feedforward comb filter for a given alpha (and delay).

Strategy comb.tau. Aliases available are `comb.fb_tau`, `comb.feedback_tau`. Docstring starts with:

Feedback comb filter for a given time constant (and delay).

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

ff(*delay*, *alpha=1*)
Feedforward comb filter for a given alpha (and delay).

```
y[n] = x[n] + alpha * x[n - delay]
```

Parameters

- **delay** – Feedback delay, in number of samples.
- **alpha** – Memory value gain.

Returns A ZFilter instance with the comb filter.

tau(*delay*, *tau=inf*)
Feedback comb filter for a given time constant (and delay).

```
y[n] = x[n] + alpha * y[n - delay]
```

Parameters

- **delay** – Feedback delay, in number of samples.
- **tau** – Time decay (up to `1/e`, or -8.686 dB), in number of samples, which allows finding `alpha = e ** (-delay / tau)`. Defaults to `inf` (infinite), which means alpha = 1.

Returns A ZFilter instance with the comb filter.

fb(*delay*, *alpha=1*)
Feedback comb filter for a given alpha (and delay).

```
y[n] = x[n] + alpha * y[n - delay]
```

Parameters

- **delay** – Feedback delay, in number of samples.
- **alpha** – Exponential decay gain. You can find it from time decay `tau` in the impulse response, bringing us `alpha = e ** (-delay / tau)`. See `comb.tau` strategy if that's the case. Defaults to 1 (no decay).

Returns A ZFilter instance with the comb filter.

2.5.2 `lazy_filters.resonator` StrategyDict

This is a StrategyDict instance object called `resonator`. Strategies stored: 4.

Strategy resonator.freq_poles_exp. Docstring starts with:

> Resonator filter with 2-poles (conjugated pair) and no zeros (constant numerator), with exponential approximation for bandwidth calculation. Given frequency is the denominator frequency, not the resonant frequency.

Strategy resonator.freq_z_exp. Docstring starts with:

> Resonator filter with 2-zeros and 2-poles (conjugated pair). The zeros are at the *1* and *-1* (both at the real axis, i.e., at the DC and the Nyquist rate), with exponential approximation for bandwidth calculation. Given frequency is the denominator frequency, not the resonant frequency.

Strategy resonator.poles_exp (Default). Docstring starts with:

> Resonator filter with 2-poles (conjugated pair) and no zeros (constant numerator), with exponential approximation for bandwidth calculation.

Strategy resonator.z_exp. Docstring starts with:

> Resonator filter with 2-zeros and 2-poles (conjugated pair). The zeros are at the *1* and *-1* (both at the real axis, i.e., at the DC and the Nyquist rate), with exponential approximation for bandwidth calculation.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

freq_poles_exp(*freq*, *bandwidth*)
Resonator filter with 2-poles (conjugated pair) and no zeros (constant numerator), with exponential approximation for bandwidth calculation. Given frequency is the denominator frequency, not the resonant frequency.

Parameters

- **freq** – Denominator frequency in rad/sample (not the one with max gain).
- **bandwidth** – Bandwidth frequency range in rad/sample following the equation: `R = exp(-bandwidth / 2)` where R is the pole amplitude (radius).

Returns A ZFilter object. Gain is normalized to have peak with 0 dB (1.0 amplitude).

freq_z_exp(*freq*, *bandwidth*)
Resonator filter with 2-zeros and 2-poles (conjugated pair). The zeros are at the *1* and *-1* (both at the real axis, i.e., at the DC and the Nyquist rate), with exponential approximation for bandwidth calculation. Given frequency is the denominator frequency, not the resonant frequency.

Parameters

- **freq** – Denominator frequency in rad/sample (not the one with max gain).
- **bandwidth** – Bandwidth frequency range in rad/sample following the equation: `R = exp(-bandwidth / 2)` where R is the pole amplitude (radius).

Returns A ZFilter object. Gain is normalized to have peak with 0 dB (1.0 amplitude).

poles_exp (*freq*, *bandwidth*)
Resonator filter with 2-poles (conjugated pair) and no zeros (constant numerator), with exponential approximation for bandwidth calculation.

Parameters

- **freq** – Resonant frequency in rad/sample (max gain).
- **bandwidth** – Bandwidth frequency range in rad/sample following the equation: `R = exp(-bandwidth / 2)` where R is the pole amplitude (radius).

Returns A ZFilter object. Gain is normalized to have peak with 0 dB (1.0 amplitude).

z_exp (*freq*, *bandwidth*)
Resonator filter with 2-zeros and 2-poles (conjugated pair). The zeros are at the *1* and *-1* (both at the real axis, i.e., at the DC and the Nyquist rate), with exponential approximation for bandwidth calculation.

Parameters

- **freq** – Resonant frequency in rad/sample (max gain).
- **bandwidth** – Bandwidth frequency range in rad/sample following the equation: `R = exp(-bandwidth / 2)` where R is the pole amplitude (radius).

Returns A ZFilter object. Gain is normalized to have peak with 0 dB (1.0 amplitude).

2.5.3 `lazy_filters.lowpass` StrategyDict

This is a StrategyDict instance object called `lowpass`. Strategies stored: 2.

Strategy lowpass.pole (Default). Docstring starts with:

> Low-pass filter with one pole and no zeros (constant numerator), with high-precision cut-off frequency calculation.

Strategy lowpass.pole_exp. Docstring starts with:

> Low-pass filter with one pole and no zeros (constant numerator), with exponential approximation for cut-off frequency calculation, found by a matching the one-pole Laplace lowpass filter.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

pole (*cutoff*)
Low-pass filter with one pole and no zeros (constant numerator), with high-precision cut-off frequency calculation.

Parameters **cutoff** – Cut-off frequency in rad/sample. It defines the filter frequency in which the squared gain is *50%* (a.k.a. magnitude gain is *sqrt(2) / 2* and power gain is about *3.0103 dB*). Should be a value between 0 and pi.

Returns A ZFilter object. Gain is normalized to have peak with 0 dB (1.0 amplitude) at the DC frequency (zero rad/sample).

pole_exp(*cutoff*)
Low-pass filter with one pole and no zeros (constant numerator), with exponential approximation for cut-off frequency calculation, found by a matching the one-pole Laplace lowpass filter.

Parameters **cutoff** – Cut-off frequency in rad/sample following the equation: `R = exp(-cutoff)` where R is the pole amplitude (radius).

Returns A ZFilter object. Gain is normalized to have peak with 0 dB (1.0 amplitude) at the DC frequency (zero rad/sample).

2.5.4 lazy_filters.highpass StrategyDict

This is a StrategyDict instance object called `highpass`. Strategies stored: 1.

Strategy highpass.pole (Default). Docstring starts with:

High-pass filter with one pole and no zeros (constant numerator), with high-precision cut-off frequency calculation.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

---

pole(*cutoff*)
High-pass filter with one pole and no zeros (constant numerator), with high-precision cut-off frequency calculation.

Parameters **cutoff** – Cut-off frequency in rad/sample. It defines the filter frequency in which the squared gain is *50%* (a.k.a. magnitude gain is *sqrt(2) / 2* and power gain is about *3.0103 dB*). Should be a value between 0 and pi.

Returns A ZFilter object. Gain is normalized to have peak with 0 dB (1.0 amplitude) at the Nyquist frequency (pi rad/sample).

2.6 lazy_io Module

Audio recording input and playing output module

Summary of module contents:

Name	Description
AudioIO	Multi-thread stream manager wrapper for PyAudio.
AudioThread	Audio output thread.

class AudioIO(*wait=False*)
Bases: `object`

Multi-thread stream manager wrapper for PyAudio.

__del__()
Default destructor. Use close method instead, or use the class instance as the expression of a with block.

__enter__()
To be used only internally, in the with-expression protocol.

__exit__(*etype*, *evalue*, *etraceback*)
Closing destructor for use internally in a with-expression.

__init__(*wait=False*)
Constructor to PyAudio Multi-thread manager audio IO interface. The "wait" input is a boolean about the behaviour on closing the instance, if it should or not wait for the streaming audio to finish. Defaults to False. Only works if the close method is explicitly called.

close()
Destructor for this audio interface. Waits the threads to finish their streams, if desired.

play(*audio*, ***kwargs*)
Start another thread playing the given audio sample iterable (e.g. a list, a generator, a NumPy np.ndarray with samples), and play it. The arguments are used to customize behaviour of the new thread, as parameters directly sent to PyAudio's new stream opening method, see AudioThread.__init__ for more.

record(**args*, ***kwargs*)
Records audio from device into a Stream.

Parameters

- **chunk_size** – Number of samples per chunk (block sent to device).
- **dfmt** – Format, as in chunks(). Default is "f" (Float32).
- **num_channels** – Channels in audio stream (serialized).
- **rate** – Sample rate (same input used in sHz).

Returns Endless Stream instance that gather data from the audio input device.

terminate()
Same as "close".

thread_finished(*thread*)
Updates internal status about open threads. Should be called only by the internal closing mechanism of children threads.

class **AudioThread**(*device_manager*, *audio*, *chunk_size=2048*, *dfmt='f'*, *nchannels=1*, *rate=44100*, *daemon=True*)
Bases: `threading.Thread`

Audio output thread.

This class is a wrapper to ease the use of PyAudio using iterables of numbers (Stream instances, lists, tuples, NumPy 1D arrays, generators) as audio data streams.

__init__(*device_manager*, *audio*, *chunk_size=2048*, *dfmt='f'*, *nchannels=1*, *rate=44100*, *daemon=True*)
Sets a new thread to play the given audio.

Parameters

- **chunk_size** – Number of samples per chunk (block sent to device).
- **dfmt** – Format, as in chunks(). Default is "f" (Float32).
- **num_channels** – Channels in audio stream (serialized).
- **rate** – Sample rate (same input used in sHz).
- **daemon** – Boolean telling if thread should be daemon. Default is True.

pause()
Pauses the audio.

play()
Resume playing the audio.

run()
Plays the audio. This method plays the audio, and shouldn't be called explicitly, let the constructor do so.

stop()
Stops the playing thread and close

2.7 lazy_itertools Module

Itertools module "decorated" replica, where all outputs are Stream instances

Summary of module contents:

Name	Description
tee	Tee or "T" copy to help working with Stream instances as well as with numbers.
count	count(start=0, step=1) –> count object
product	product(*iterables) –> product object
repeat	repeat(object [,times]) -> create an iterator which returns the object for the specified number of times. If not specified, returns the object endlessly.
starmap	starmap(function, sequence) –> starmap object
chain	chain(*iterables) –> chain object
compress	compress(data, selectors) –> iterator over selected data
izip	izip(iter1 [,iter2 [...]]) –> izip object
permutations	permutations(iterable[, r]) –> permutations object
combinations_with_replacement	combinations_with_replacement(iterable, r) –> combinations_with_replacement object
groupby	groupby(iterable[, keyfunc]) -> create an iterator which returns (key, subiterator) grouped by each value of key(value).
imap	imap(func, *iterables) –> imap object
islice	islice(iterable, [start,] stop [, step]) –> islice object
combinations	combinations(iterable, r) –> combinations object
izip_longest	izip_longest(iter1 [,iter2 [...]], [fillvalue=None]) –> izip_longest object
cycle	cycle(iterable) –> cycle object
ifilter	ifilter(function or None, sequence) –> ifilter object
takewhile	takewhile(predicate, iterable) –> takewhile object
ifilterfalse	ifilterfalse(function or None, sequence) –> ifilterfalse object
dropwhile	dropwhile(predicate, iterable) –> dropwhile object

tee(*data*, *n=2*)
Tee or "T" copy to help working with Stream instances as well as with numbers.

Parameters

- **data** – Input to be copied. Can be anything.
- **n** – Size of returned tuple. Defaults to 2.

Returns Tuple of n independent Stream instances, if the input is a Stream or an iterator, otherwise a tuple with n times the same object.

See Also:

thub **(page 59)** use Stream instances *almost* like constants in your equations.

count(**args*, ***kwargs*)
count(start=0, step=1) –> count object

Return a count object whose .next() method returns consecutive values. Equivalent to:

def count(firstval=0, step=1): x = firstval while 1:

yield x x += step

product(**args*, ***kwargs*)
product(*iterables) –> product object

Cartesian product of input iterables. Equivalent to nested for-loops.

For example, product(A, B) returns the same as: ((x,y) for x in A for y in B). The leftmost iterators are in the outermost for-loop, so the output tuples cycle in a manner similar to an odometer (with the rightmost element changing on every iteration).

To compute the product of an iterable with itself, specify the number of repetitions with the optional repeat keyword argument. For example, product(A, repeat=4) means the same as product(A, A, A, A).

product('ab', range(3)) –> ('a',0) ('a',1) ('a',2) ('b',0) ('b',1) ('b',2) product((0,1), (0,1), (0,1)) –> (0,0,0) (0,0,1) (0,1,0) (0,1,1) (1,0,0) ...

repeat (*object*[, *times*]) → create an iterator which returns the object
for the specified number of times. If not specified, returns the object endlessly.

starmap (**args*, ***kwargs*)
starmap(function, sequence) –> starmap object

Return an iterator whose values are returned from the function evaluated with a argument tuple taken from the given sequence.

chain (**args*, ***kwargs*)
chain(*iterables) –> chain object

Return a chain object whose .next() method returns elements from the first iterable until it is exhausted, then elements from the next iterable, until all of the iterables are exhausted.

compress (**args*, ***kwargs*)
compress(data, selectors) –> iterator over selected data

Return data elements corresponding to true selector elements. Forms a shorter iterator from selected data elements using the selectors to choose the data elements.

izip (**args*, ***kwargs*)
izip(iter1 [,iter2 [...]]) –> izip object

Return a izip object whose .next() method returns a tuple where the i-th element comes from the i-th iterable argument. The .next() method continues until the shortest iterable in the argument sequence is exhausted and then it raises StopIteration. Works like the zip() function but consumes less memory by returning an iterator instead of a list.

permutations (**args*, ***kwargs*)
permutations(iterable[, r]) –> permutations object

Return successive r-length permutations of elements in the iterable.

permutations(range(3), 2) –> (0,1), (0,2), (1,0), (1,2), (2,0), (2,1)

combinations_with_replacement (**args*, ***kwargs*)
combinations_with_replacement(iterable, r) –> combinations_with_replacement object

Return successive r-length combinations of elements in the iterable allowing individual elements to have successive repeats. combinations_with_replacement('ABC', 2) –> AA AB AC BB BC CC

groupby (*iterable*[, *keyfunc*]) → create an iterator which returns
(key, sub-iterator) grouped by each value of key(value).

imap (**args*, ***kwargs*)
imap(func, *iterables) –> imap object

Make an iterator that computes the function using arguments from each of the iterables. Like map() except that it returns an iterator instead of a list and that it stops when the shortest iterable is exhausted instead of filling in None for shorter iterables.

islice (**args*, ***kwargs*)
islice(iterable, [start,] stop [, step]) –> islice object

Return an iterator whose next() method returns selected values from an iterable. If start is specified, will skip all preceding elements; otherwise, start defaults to zero. Step defaults to one. If specified as another

value, step determines how many values are skipped between successive calls. Works like a slice() on a list but returns an iterator.

combinations(**args*, ***kwargs*)
combinations(iterable, r) –> combinations object

Return successive r-length combinations of elements in the iterable.

combinations(range(4), 3) –> (0,1,2), (0,1,3), (0,2,3), (1,2,3)

izip_longest(**args*, ***kwargs*)
izip_longest(iter1 [,iter2 [...]], [fillvalue=None]) –> izip_longest object

Return an izip_longest object whose .next() method returns a tuple where the i-th element comes from the i-th iterable argument. The .next() method continues until the longest iterable in the argument sequence is exhausted and then it raises StopIteration. When the shorter iterables are exhausted, the fillvalue is substituted in their place. The fillvalue defaults to None or can be specified by a keyword argument.

cycle(**args*, ***kwargs*)
cycle(iterable) –> cycle object

Return elements from the iterable until it is exhausted. Then repeat the sequence indefinitely.

ifilter(**args*, ***kwargs*)
ifilter(function or None, sequence) –> ifilter object

Return those items of sequence for which function(item) is true. If function is None, return the items that are true.

takewhile(**args*, ***kwargs*)
takewhile(predicate, iterable) –> takewhile object

Return successive entries from an iterable as long as the predicate evaluates to true for each entry.

ifilterfalse(**args*, ***kwargs*)
ifilterfalse(function or None, sequence) –> ifilterfalse object

Return those items of sequence for which function(item) is false. If function is None, return the items that are false.

dropwhile(**args*, ***kwargs*)
dropwhile(predicate, iterable) –> dropwhile object

Drop items from the iterable while predicate(item) is true. Afterwards, return every element until the iterable is exhausted.

2.8 `lazy_lpc` Module

Linear Predictive Coding (LPC) module

Summary of module contents:

Name	Description
ParCorError	Error when trying to find the partial correlation coefficients (reflection coefficients) and there's no way to find them.
toeplitz	Find the toeplitz matrix as a list of lists given its first line/column.
levinson_durbin	Solve the Yule-Walker linear system of equations.
lpc	This is a StrategyDict instance object called `lpc`. Strategies stored: 5.
parcor	Find the partial correlation coefficients (PARCOR), or reflection coefficients, relative to the lattice implementation of a given LTI FIR LinearFilter with a constant denominator (i.e., LPC analysis filter, or any filter without feedback).
parcor_stable	Tests whether the given filter is stable or not by using the partial correlation coefficients (reflection coefficients) of the given filter.
lsf	Find the Line Spectral Frequencies (LSF) from a given FIR filter.
lsf_stable	Tests whether the given filter is stable or not by using the Line Spectral Frequencies (LSF) of the given filter. Needs NumPy.

exception `ParCorError`

Bases: `exceptions.ZeroDivisionError`

Error when trying to find the partial correlation coefficients (reflection coefficients) and there's no way to find them.

`toeplitz` (*vect*)

Find the toeplitz matrix as a list of lists given its first line/column.

`levinson_durbin` (*acdata*, *order=None*)

Solve the Yule-Walker linear system of equations.

They're given by:

$$R.a = r$$

where R is a simmetric Toeplitz matrix where each element are lags from the given autocorrelation list. R and r are defined (Python indexing starts with zero and slices don't include the last element):

$$R[i][j] = acdata[abs(j - i)]$$
$$r = acdata[1 : order + 1]$$

Parameters

- **acdata** – Autocorrelation lag list, commonly the `acorr` function output.
- **order** – The order of the resulting ZFilter object. Defaults to `len(acdata) - 1`.

Returns A FIR filter, as a ZFilter object. The mean squared error over the given data (variance of the white noise) is in its "error" attribute.

See Also:

`acorr` **(page 12)** Calculate the autocorrelation of a given block.

`lpc` **(page 41)** Calculate the Linear Predictive Coding (LPC) coefficients.

`parcor` **(page 40)** Partial correlation coefficients (PARCOR), or reflection coefficients, relative to the lattice implementation of a filter, obtained by reversing the Levinson-Durbin algorithm.

Examples:

```
>>> data = [2, 2, 0, 0, -1, -1, 0, 0, 1, 1]
>>> acdata = acorr(data)
>>> acdata
[12, 6, 0, -3, -6, -3, 0, 2, 4, 2]
>>> ldfilt = levinson_durbin(acorr(data), 3)
>>> ldfilt
1 - 0.625 * z^-1 + 0.25 * z^-2 + 0.125 * z^-3
>>> ldfilt.error # Squared! See lpc for more information about this
7.875
```

The Levinson-Durbin algorithm used to solve the equations needs $O(order^2)$ floating point operations.

parcor (*fir_filt*)
Find the partial correlation coefficients (PARCOR), or reflection coefficients, relative to the lattice implementation of a given LTI FIR LinearFilter with a constant denominator (i.e., LPC analysis filter, or any filter without feedback).

Parameters **fir_filt** – A ZFilter object, causal, LTI and with a constant denominator.

Returns A generator that results in each partial correlation coefficient from iterative decomposition, reversing the Levinson-Durbin algorithm.

Examples:

```
>>> filt = levinson_durbin([1, 2, 3, 4, 5, 3, 2, 1])
>>> filt
1 - 0.275 * z^-1 - 0.275 * z^-2 - 0.4125 * z^-3 + 1.5 * z^-4 - 0.9125 * z^-5 - 0.275 * z^
>>> round(filt.error, 4)
1.9125
>>> k_generator = parcor(filt)
>>> k_generator
<generator object parcor at ...>
>>> [round(k, 7) for k in k_generator]
[-0.275, -0.3793103, -1.4166667, -0.2, -0.25, -0.3333333, -2.0]
```

See Also:

levinson_durbin (page 39) Levinson-Durbin algorithm for solving Yule-Walker equations (Toeplitz matrix linear system).

parcor_stable (*filt*)
Tests whether the given filter is stable or not by using the partial correlation coefficients (reflection coefficients) of the given filter.

Parameters **filt** – A LTI filter as a LinearFilter object.

Returns A boolean that is true only when all correlation coefficients are inside the unit circle. Critical stability (i.e., when outer coefficient has magnitude equals to one) is seem as an instability, and returns False.

See Also:

parcor (page 40) Partial correlation coefficients generator.

lsf_stable (page 40) Tests filter stability with Line Spectral Frequencies (LSF) values.

lsf (*fir_filt*)
Find the Line Spectral Frequencies (LSF) from a given FIR filter.

Parameters **filt** – A LTI FIR filter as a LinearFilter object.

Returns A tuple with all LSFs in rad/sample, alternating from the forward prediction and backward prediction filters, starting with the lowest LSF value.

lsf_stable (*filt*)
Tests whether the given filter is stable or not by using the Line Spectral Frequencies (LSF) of the given filter. Needs NumPy.

Parameters **filt** – A LTI filter as a LinearFilter object.

Returns A boolean that is true only when the LSF values from forward and backward prediction filters alternates. Critical stability (both forward and backward filters has the same LSF value) is seem as an instability, and returns False.

See Also:

`lsf` **(page 40)** Gets the Line Spectral Frequencies from a filter. Needs NumPy.

`parcor_stable` **(page 40)** Tests filter stability with partial correlation coefficients (reflection coefficients).

2.8.1 `lazy_lpc.lpc` StrategyDict

This is a StrategyDict instance object called `lpc`. Strategies stored: 5.

Strategy lpc.autocor (Default). Aliases available are `lpc.acorr`, `lpc.autocorrelation`, `lpc.auto_correlation`. Docstring starts with:

> Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the autocorrelation method, using the Levinson-Durbin algorithm or Numpy pseudo-inverse for linear system solving, when needed.

Strategy lpc.covar. Aliases available are `lpc.cov`, `lpc.covariance`, `lpc.ncovar`, `lpc.ncov`, `lpc.ncovariance`. Docstring starts with:

> Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the covariance method, assuming a zero-mean stochastic process, using numpy.linalg.pinv as a linear system solver.

Strategy lpc.kautocor. Aliases available are `lpc.kacorr`, `lpc.kautocorrelation`, `lpc.kauto_correlation`. Docstring starts with:

> Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the autocorrelation method, using the Levinson-Durbin algorithm.

Strategy lpc.kcovar. Aliases available are `lpc.kcov`, `lpc.kcovariance`. Docstring starts with:

> Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation is based on the covariance method, assuming a zero-mean stochastic process, finding the coefficients iteratively and greedily like the lattice implementation in Levinson-Durbin algorithm, although the lag matrix found from the given block don't have to be toeplitz. Slow, but this strategy don't need NumPy.

Strategy lpc.nautocor. Aliases available are `lpc.nacorr`, `lpc.nautocorrelation`, `lpc.nauto_correlation`. Docstring starts with:

> Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the autocorrelation method, using numpy.linalg.pinv as a linear system solver.

Note: StrategyDict instances like this one have lazy self-generated docstrings. If you change something in the dict, the next docstrings will follow the change. Calling this instance directly will have the same effect as calling the default strategy. You can see the full strategies docstrings for more details, as well as the StrategyDict class documentation.

`kcovar`(*blk*, *order=None*)

Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation is based on the covariance method, assuming a zero-mean stochastic process, finding the coefficients iteratively and greedily like the lattice implementation in Levinson-Durbin algorithm, although the lag matrix found from the given block don't have to be toeplitz. Slow, but this strategy don't need NumPy.

`autocor`(*blk*, *order=None*)

Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the autocorrelation method, using the Levinson-Durbin algorithm or Numpy pseudo-inverse for linear system solving, when needed.

Parameters

- **blk** – An iterable with well-defined length. Don't use this function with Stream objects!

- **order** – The order of the resulting ZFilter object. Defaults to `len(blk) - 1`.

Returns A FIR filter, as a ZFilter object. The mean squared error over the given block is in its "error" attribute.

Examples:

```
>>> data = [-1, 0, 1, 0] * 4
>>> len(data) # Small data
16
>>> filt = lpc.autocor(data, 2)
>>> filt # The analysis filter
1 + 0.875 * z^-1
>>> filt.numerator # List of coefficients
[1, 0.875]
>>> filt.error # Prediction error (squared!)
14.125
```

See Also:

levinson_durbin (page 39) Levinson-Durbin algorithm for solving Yule-Walker equations (Toeplitz matrix linear system).

lpc.nautocor (page 42) LPC coefficients from linear system solved with Numpy pseudo-inverse.

lpc.kautocor (page 42) LPC coefficients obtained with Levinson-Durbin algorithm.

kautocor(*blk*, *order=None*)

Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the autocorrelation method, using the Levinson-Durbin algorithm.

Parameters

- **blk** – An iterable with well-defined length. Don't use this function with Stream objects!
- **order** – The order of the resulting ZFilter object. Defaults to `len(blk) - 1`.

Returns A FIR filter, as a ZFilter object. The mean squared error over the given block is in its "error" attribute.

Examples:

```
>>> data = [-1, 0, 1, 0] * 4
>>> len(data) # Small data
16
>>> filt = lpc.kautocor(data, 2)
>>> filt # The analysis filter
1 + 0.875 * z^-1
>>> filt.numerator # List of coefficients
[1, 0.875]
>>> filt.error # Prediction error (squared!)
14.125
```

See Also:

levinson_durbin (page 39) Levinson-Durbin algorithm for solving Yule-Walker equations (Toeplitz matrix linear system).

covar(*blk*, *order=None*)

Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the covariance method, assuming a zero-mean stochastic process, using numpy.linalg.pinv as a linear system solver.

nautocor(*blk*, *order=None*)

Find the Linear Predictive Coding (LPC) coefficients as a ZFilter object, the analysis whitening filter. This implementation uses the autocorrelation method, using numpy.linalg.pinv as a linear system solver.

Parameters

- **blk** – An iterable with well-defined length. Don't use this function with Stream objects!
- **order** – The order of the resulting ZFilter object. Defaults to `len(blk) - 1`.

Returns A FIR filter, as a ZFilter object. The mean squared error over the given block is in its "error" attribute.

Examples:

```
>>> data = [-1, 0, 1, 0] * 4
>>> len(data) # Small data
16
>>> filt = lpc["nautocor"](data, 2)
>>> filt # The analysis filter
1 + 0.875 * z^-1
>>> filt.numerator # List of coefficients3
[1, 0.875]
>>> filt.error # Prediction error (squared!)
14.125
```

2.9 `lazy_math` Module

Math modules "decorated" and complemented to work elementwise when needed

Summary of module contents:

Name	Description
abs	abs(number) -> number
pi	`3.14159265359`
e	`2.71828182846`
cexp	exp(x)
ln	log(x[, base])
log2	* * * * ...no docstring... * * * *
factorial	Factorial function that works with really big numbers.
dB10	Convert a gain value to dB, from a squared amplitude value to a power gain.
dB20	Convert a gain value to dB, from a amplitude value to a power gain.
inf	`inf`
nan	`nan`
phase	phase(z) -> float
sign	* * * * ...no docstring... * * * *
acos	acos(x)
acosh	acosh(x)
asin	asin(x)
asinh	asinh(x)
atan	atan(x)
atanh	atanh(x)
ceil	ceil(x)
cos	cos(x)
cosh	cosh(x)
degrees	degrees(x)
erf	erf(x)
erfc	erfc(x)

Continued on next page

Table 2.1 – continued from previous page

Name	Description
exp	exp(x)
expm1	expm1(x)
fabs	fabs(x)
floor	floor(x)
frexp	frexp(x)
gamma	gamma(x)
isinf	isinf(x) -> bool
isnan	isnan(x) -> bool
lgamma	lgamma(x)
log	log(x[, base])
log10	log10(x)
log1p	log1p(x)
modf	modf(x)
radians	radians(x)
sin	sin(x)
sinh	sinh(x)
sqrt	sqrt(x)
tan	tan(x)
tanh	tanh(x)
trunc	trunc(x:Real) -> Integral

abs(*number*) → number
Return the absolute value of the argument.

cexp(**args*, ***kwargs*)
exp(x)

Return the exponential value e**x.

ln(**args*, ***kwargs*)
log(x[, base])

Return the logarithm of x to the given base. If the base not specified, returns the natural logarithm (base e) of x.

log2(**args*, ***kwargs*)

factorial(**args*, ***kwargs*)
Factorial function that works with really big numbers.

dB10(**args*, ***kwargs*)
Convert a gain value to dB, from a squared amplitude value to a power gain.

dB20(**args*, ***kwargs*)
Convert a gain value to dB, from a amplitude value to a power gain.

phase(*z*) → float
Return argument, also known as the phase angle, of a complex.

sign(**args*, ***kwargs*)

acos(*x*)
Return the arc cosine (measured in radians) of x.

acosh(*x*)
Return the hyperbolic arc cosine (measured in radians) of x.

asin(*x*)
Return the arc sine (measured in radians) of x.

asinh(*x*)
Return the hyperbolic arc sine (measured in radians) of x.

atan (*x*)
Return the arc tangent (measured in radians) of x.

atanh (*x*)
Return the hyperbolic arc tangent (measured in radians) of x.

ceil (*x*)
Return the ceiling of x as a float. This is the smallest integral value >= x.

cos (*x*)
Return the cosine of x (measured in radians).

cosh (*x*)
Return the hyperbolic cosine of x.

degrees (*x*)
Convert angle x from radians to degrees.

erf (*x*)
Error function at x.

erfc (*x*)
Complementary error function at x.

exp (*x*)
Return e raised to the power of x.

expm1 (*x*)
Return exp(x)-1. This function avoids the loss of precision involved in the direct evaluation of exp(x)-1 for small x.

fabs (*x*)
Return the absolute value of the float x.

floor (*x*)
Return the floor of x as a float. This is the largest integral value <= x.

frexp (*x*)
Return the mantissa and exponent of x, as pair (m, e). m is a float and e is an int, such that x = m * 2.**e. If x is 0, m and e are both 0. Else 0.5 <= abs(m) < 1.0.

gamma (*x*)
Gamma function at x.

isinf (*x*) → bool
Check if float x is infinite (positive or negative).

isnan (*x*) → bool
Check if float x is not a number (NaN).

lgamma (*x*)
Natural logarithm of absolute value of Gamma function at x.

log (*x*[, *base*])
Return the logarithm of x to the given base. If the base not specified, returns the natural logarithm (base e) of x.

log10 (*x*)
Return the base 10 logarithm of x.

log1p (*x*)
Return the natural logarithm of 1+x (base e). The result is computed in a way which is accurate for x near zero.

modf (*x*)
Return the fractional and integer parts of x. Both results carry the sign of x and are floats.

radians (*x*)
Convert angle x from degrees to radians.

sin(*x*)
Return the sine of x (measured in radians).

sinh(*x*)
Return the hyperbolic sine of x.

sqrt(*x*)
Return the square root of x.

tan(*x*)
Return the tangent of x (measured in radians).

tanh(*x*)
Return the hyperbolic tangent of x.

trunc(*x:Real*) → Integral
Truncates x to the nearest Integral toward 0. Uses the __trunc__ magic method.

2.10 `lazy_midi` Module

MIDI representation data & note-frequency relationship

Summary of module contents:

Name	Description
MIDI_A4	`69`
FREQ_A4	`440.0`
SEMITONE_RATIO	`1.05946309436`
str2freq	Given a note string name (e.g. "F#2"), returns its frequency in Hz.
str2midi	Given a note string name (e.g. "Bb4"), returns its MIDI pitch number.
freq2str	Given a frequency in Hz, returns its note string name (e.g. "D7").
freq2midi	Given a frequency in Hz, returns its MIDI pitch number.
midi2freq	Given a MIDI pitch number, returns its frequency in Hz.
midi2str	Given a MIDI pitch number, returns its note string name (e.g. "C3").
octaves	Given a frequency and a frequency range, returns all frequencies in that range that is an integer number of octaves related to the given frequency.

str2freq(*note_string*)
Given a note string name (e.g. "F#2"), returns its frequency in Hz.

str2midi(**args*, ***kwargs*)
Given a note string name (e.g. "Bb4"), returns its MIDI pitch number.

freq2str(*freq*)
Given a frequency in Hz, returns its note string name (e.g. "D7").

freq2midi(**args*, ***kwargs*)
Given a frequency in Hz, returns its MIDI pitch number.

midi2freq(**args*, ***kwargs*)
Given a MIDI pitch number, returns its frequency in Hz.

midi2str(**args*, ***kwargs*)
Given a MIDI pitch number, returns its note string name (e.g. "C3").

octaves(*freq*, *fmin=20.0*, *fmax=20000.0*)
Given a frequency and a frequency range, returns all frequencies in that range that is an integer number of octaves related to the given frequency.

Parameters

- **freq** – Frequency, in any (linear) unit.
- **[fmin** – ...

- **fmax]** – Frequency range, in the same unit of `freq`. Defaults to 20.0 and 20,000.0, respectively.

Returns A list of frequencies, in the same unit of `freq` and in ascending order.

Examples:

```
>>> from audiolazy import octaves, sHz
>>> octaves(440.)
[27.5, 55.0, 110.0, 220.0, 440.0, 880.0, 1760.0, 3520.0, 7040.0, 14080.0]
>>> octaves(440., fmin=3000)
[3520.0, 7040.0, 14080.0]
>>> Hz = sHz(44100)[1] # Conversion unit from sample rate
>>> freqs = octaves(440 * Hz, fmin=300 * Hz, fmax = 1000 * Hz) # rad/sample
>>> len(freqs) # Number of octaves
2
>>> [round(f, 6) for f in freqs] # Values in rad/sample
[0.062689, 0.125379]
>>> [round(f / Hz, 6) for f in freqs] # Values in Hz
[440.0, 880.0]
```

2.11 `lazy_misc` Module

Common miscellanous tools and constants for general use

Summary of module contents:

Name	Description
DEFAULT_SAMPLE_RATE	`44100`
DEFAULT_CHUNK_SIZE	`2048`
LATEX_PI_SYMBOL	`$\pi$`
blocks	General iterable blockenizer.
chunks	Chunk generator, or a blockenizer for homogeneous data, to help writing an iterable into a file. This implementation is based on the struct module.
array_chunks	Chunk generator based on the array module (Python standard library).
zero_pad	Zero padding sample generator (not a Stream!).
elementwise	Function auto-map decorator broadcaster.
almost_eq_diff	Almost equal, based on the $\lvert a - b \rvert$ value.
almost_eq	Almost equal, based on the amount of floating point significand bits.
multiplication_formatter	Formats a number `value * symbol ** power` as a string, where symbol is already a string and both other inputs are numbers.
pair_strings_sum_formatter	Formats the sum of a and b, where both are numbers already converted to strings.
rational_formatter	Converts a given numeric value to a string based on rational fractions of the given symbol, useful for labels in plots.
pi_formatter	String formatter for fractions of π.
auto_formatter	Automatic string formatter for integer fractions, fractions of π and float numbers with small number of digits.
rst_table	Creates a reStructuredText simple table (list of strings) from a list of lists.
small_doc	Finds a useful small doc representation of an object.
sHz	Unit conversion constants.

`blocks` (*seq*, *size=2048*, *hop=None*, *padval=0.0*)

General iterable blockenizer.

Generator that gets `size` elements from `seq`, and outputs them in a collections.deque (mutable circular queue) sequence container. Next output starts `hop` elements after the first element in last output block. Last block may be appended with `padval`, if needed to get the desired size.

The `seq` can have hybrid / hetherogeneous data, it just need to be an iterable. You can use other type content as padval (e.g. None) to help segregate the padding at the end, if desired.

Note: When hop is less than size, changing the returned contents will keep the new changed value in the next yielded container.

---

chunks(*seq*, *size=2048*, *dfmt='f'*, *byte_order=None*, *padval=0.0*)

Chunk generator, or a blockenizer for homogeneous data, to help writing an iterable into a file. This implementation is based on the struct module.

The dfmt should be one char, chosen from the ones in link:

http://docs.python.org/library/struct.html#format-characters

Useful examples (integer are signed, use upper case for unsigned ones):

- "b" for 8 bits (1 byte) integer
- "h" for 16 bits (2 bytes) integer
- "i" for 32 bits (4 bytes) integer
- "f" for 32 bits (4 bytes) float (default)
- "d" for 64 bits (8 bytes) float (double)

Byte order follows native system defaults. Other options are in the site:

http://docs.python.org/library/struct.html#struct-alignment

They are:

- "<" means little-endian
- ">" means big-endian

array_chunks(*seq*, *size=2048*, *dfmt='f'*, *byte_order=None*, *padval=0.0*)

Chunk generator based on the array module (Python standard library).

Generator: Another Repetitive Replacement Again Yielding chunks, this is an audiolazy.chunks(...) clone using array.array (random access by indexing management) instead of struct.Struct and blocks/deque (circular queue appending). Try before to find the faster one for your machine.

Note: The `dfmt` symbols for arrays might differ from structs' defaults.

---

zero_pad(*seq*, *left=0*, *right=0*, *zero=0.0*)

Zero padding sample generator (not a Stream!).

Parameters

- **seq** – Sequence to be padded.
- **left** – Integer with the number of elements to be padded at left (before). Defaults to zero.
- **right** – Integer with the number of elements to be padded at right (after). Defaults to zero.
- **zero** – Element to be padded. Defaults to a float zero (0.0).

Returns A generator that pads the given `seq` with samples equals to `zero`, `left` times before and `right` times after it.

elementwise(*name=''*, *pos=None*)

Function auto-map decorator broadcaster.

Creates an "elementwise" decorator for one input parameter. To create such, it should know the name (for use as a keyword argument and the position "pos" (input as a positional argument). Without a name, only the positional argument will be used. Without both name and position, the first positional argument will be used.

almost_eq_diff(*a*, *b*, *max_diff=1e-07*, *ignore_type=True*, *pad=0.0*)
Almost equal, based on the $|a - b|$ value.

Alternative to "a == b" for float numbers and iterables with float numbers. See almost_eq for more information.

This version based on the non-normalized absolute diff, similar to what unittest does with its assertAlmostEquals. If a and b sizes differ, at least one will be padded with the pad input value to keep going with the comparison.

Note: Be careful with endless generators!

---

almost_eq(*a*, *b*, *bits=32*, *tol=1*, *ignore_type=True*, *pad=0.0*)
Almost equal, based on the amount of floating point significand bits.

Alternative to "a == b" for float numbers and iterables with float numbers, and tests for sequence contents (i.e., an elementwise a == b, that also works with generators, nested lists, nested generators, etc.). If the type of both the contents and the containers should be tested too, set the ignore_type keyword arg to False. Default version is based on 32 bits IEEE 754 format (23 bits significand). Could use 64 bits (52 bits significand) but needs a native float type with at least that size in bits. If a and b sizes differ, at least one will be padded with the pad input value to keep going with the comparison.

Note: Be careful with endless generators!

---

multiplication_formatter(*power*, *value*, *symbol*)
Formats a number `value * symbol ** power` as a string, where symbol is already a string and both other inputs are numbers.

pair_strings_sum_formatter(*a*, *b*)
Formats the sum of a and b, where both are numbers already converted to strings.

rational_formatter(**args*, ***kwargs*)
Converts a given numeric value to a string based on rational fractions of the given symbol, useful for labels in plots.

Parameters

- **value** – A float number or an iterable with floats.
- **symbol_str** – String data that will be in the output representing the data as a numerator multiplier, if needed. Defaults to an empty string.
- **symbol_value** – The conversion value for the given symbol (e.g. pi = 3.1415...). Defaults to one (no effect).
- **after** – Chooses the place where the symbol_str should be written. If True, that's the end of the string. If False, that's in between the numerator and the denominator, before the slash. Defaults to False.
- **max_denominator** – An int instance, used to round the float following the given limit. Defaults to the integer 1,000,000 (one million).

Returns A string with the rational number written into, with or without the symbol.

Examples:

```
>>> rational_formatter(12.5)
'25/2'
>>> rational_formatter(0.333333333333333)
'1/3'
>>> rational_formatter(0.333)
'333/1000'
>>> rational_formatter(0.333, max_denominator=100)
'1/3'
```

```
>>> rational_formatter(0.125, symbol_str="steps")
'steps/8'
>>> rational_formatter(0.125, symbol_str=" Hz",
...                    after=True) # The symbol includes whitespace!
'1/8 Hz'
```

See Also:

pi_formatter (page 50) Curried rational_formatter for the "pi" symbol.

pi_formatter(*value*, *after=False*, *max_denominator=1000000*)
String formatter for fractions of π.

Alike the rational_formatter, but fixed to the symbol string LATEX_PI_SYMBOL and symbol value `pi`, for direct use with MatPlotLib labels.

See Also:

rational_formatter (page 49) Float to string conversion, perhaps with a symbol as a multiplier.

auto_formatter(*value, order='pprpr', size=[4, 5, 3, 6, 4], after=False, max_denominator=1000000*)
Automatic string formatter for integer fractions, fractions of π and float numbers with small number of digits.

Chooses between pi_formatter, rational_formatter without a symbol and a float representation by counting each digit, the "pi" symbol and the slash as one char each, trying in the given `order` until one gets at most the given `size` limit parameter as its length.

Parameters

- **value** – A float number or an iterable with floats.
- **order** – A string that gives the order to try formatting. Each char should be: - "p" for pi_formatter; - "r" for rational_formatter without symbol; - "f" for the float representation. Defaults to "pprpr". If no trial has the desired size, returns the float representation.
- **size** – The max size allowed for each formatting in the `order`, respectively. Defaults to [4, 5, 3, 6, 4].
- **after** – Chooses the place where the LATEX_PI_SYMBOL symbol, if that's the case. If True, that's the end of the string. If False, that's in between the numerator and the denominator, before the slash. Defaults to False.
- **max_denominator** – An int instance, used to round the float following the given limit. Defaults to the integer 1,000,000 (one million).

Returns A string with the number written into.

Note: You probably want to keep `max_denominator` high to avoid rounding.

---

rst_table(*data*, *schema=None*)
Creates a reStructuredText simple table (list of strings) from a list of lists.

small_doc(*obj*, *indent=''*, *max_width=80*)
Finds a useful small doc representation of an object.

Parameters

- **obj** – Any object, which the documentation representation should be taken from.
- **indent** – Result indentation string to be insert in front of all lines.
- **max_width** – Each line of the result may have at most this length.

Returns For classes, modules, functions, methods, properties and StrategyDict instances, returns the first paragraph in the doctring of the given object, as a list of strings, stripped at right and with indent at left. For other inputs, it will use themselves cast to string as their docstring.

sHz(*rate*)

Unit conversion constants.

Useful for casting to/from the default package units (number of samples for time and rad/second for frequency). You can use expressions like `440 * Hz` to get a frequency value, or assign like `kHz = 1e3 * Hz` to get other unit, as you wish.

Parameters **rate** – Sample rate in samples per second

Returns A tuple `(s, Hz)`, where `s` is the second unit and `Hz` is the hertz unit, as the number of samples and radians per sample, respectively.

2.12 `lazy_poly` Module

Polynomial model

Summary of module contents:

Name	Description
PolyMeta	Poly metaclass. This class overloads few operators to the Poly class. All binary dunders (non reverse) should be implemented on the Poly class
Poly	Model for a polynomial.

class PolyMeta

Bases: `audiolazy.lazy_core.AbstractOperatorOverloaderMeta` (page 20)

Poly metaclass. This class overloads few operators to the Poly class. All binary dunders (non reverse) should be implemented on the Poly class

__operators__ = 'add radd sub rsub mul rmul pow div truediv eq ne pos neg '

__rbinary__(*op_func*)

__unary__(*op_func*)

class Poly(*data=None*, *zero=0*)

Bases: `object`

Model for a polynomial.

That's not a dict and not a list but behaves like something in between. The "values" method allows casting to list with list(Poly.values()) The "terms" method allows casting to dict with dict(Poly.terms()), and give the terms sorted by their power value.

__add__(*other*)

__call__(*value*)

Apply value to the Poly, where value can be other Poly. When value is a number, a Horner-like scheme is done.

__div__(*other*)

__eq__(*other*)

__getitem__(*item*)

__init__(*data=None*, *zero=0*)

Inits a polynomial from given data, which can be a list or a dict.

A list $[a_0, a_1, a_2, a_3, ...]$ inits a polynomial like

$$a_0 + a_1.x + a_2.x^2 + a_3.x^3 + ...$$

If data is a dictionary, powers are the keys and the a_i factors are the values, so negative powers are allowed and you can neglect the zeros in between, i.e., a dict vith terms like `{power: value}` can also be used.

__len__()
Number of terms, not values (be careful).

__metaclass__
alias of `PolyMeta` (page 51)

__mul__(*other*)

__ne__(*other*)

__neg__()

__pos__()

__pow__(*other*)
Power operator. The "other" parameter should be an int (or anything like), but it works with float when the Poly has only one term.

__radd__(*other*)

__repr__()

__rmul__(*other*)

__rsub__(*other*)

__str__()

__sub__(*other*)

__truediv__(*other*)

_compact_zeros()

diff(*n=1*)
Differentiate (n-th derivative, where the default n is 1).

integrate()
Integrate without adding an integration constant.

roots
Returns a list with all roots. Needs Numpy.

terms()
Pairs (2-tuple) generator where each tuple has a (power, value) term, sorted by power. Useful for casting as dict.

values()
Array values generator for powers from zero to upper power. Useful to cast as list/tuple and for numpy/scipy integration (be careful: numpy use the reversed from the output of this function used as input to a list or a tuple constructor).

2.13 `lazy_stream` Module

Stream class definition module

Summary of module contents:

Name	Description
StreamMeta	Stream metaclass. This class overloads all operators to the Stream class, but cmp/rcmp (deprecated), ternary pow (could be called with Stream.map) as well as divmod (same as pow, but this will result in a Stream of tuples).
Stream	Stream class. Stream instances are iterables that can be seem as generators with elementwise operators.
avoid_stream	Decorator to a class whose instances should avoid casting to a Stream when used with operators applied to them.
tostream	Decorator to convert the function output into a Stream. Useful for generator functions.
ControlStream	A Stream that yields a control value that can be changed at any time. You just need to set the attribute "value" for doing so, and the next value the Stream will yield is the given value.
MemoryLeakWarning	A warning to be used when a memory leak is detected.
StreamTeeHub	A Stream that returns a different iterator each time it is used.
thub	Tee or "T" hub auto-copier to help working with Stream instances as well as with numbers.
Streamix	Stream mixer of iterables.

class StreamMeta

Bases: audiolazy.lazy_core.AbstractOperatorOverloaderMeta (page 20)

Stream metaclass. This class overloads all operators to the Stream class, but cmp/rcmp (deprecated), ternary pow (could be called with Stream.map) as well as divmod (same as pow, but this will result in a Stream of tuples).

__binary__(*op_func*)

__operators__ = 'add radd sub rsub mul rmul pow rpow div rdiv mod rmod truediv rtruediv floordiv rfloordiv po

__rbinary__(*op_func*)

__unary__(*op_func*)

class Stream(**dargs*)

Bases: _abcoll.Iterable

Stream class. Stream instances are iterables that can be seem as generators with elementwise operators.

Examples: If you want something like:

```
>>> import itertools
>>> x = itertools.count()
>>> y = itertools.repeat(3)
>>> z = 2*x + y
Traceback (most recent call last):
    ...
TypeError: unsupported operand type(s) for *: 'int' and 'itertools.count'
```

That won't work with standard itertools. That's an error, and not only __mul__ but also __add__ isn't supported by their types. On the other hand, you can use this Stream class:

```
>>> x = Stream(itertools.count()) # Iterable
>>> y = Stream(3) # Non-iterable repeats endlessly
>>> z = 2*x + y
>>> z
<audiolazy.lazy_stream.Stream object at 0x...>
>>> z.take(12)
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
```

If you just want to use your existing code, an "itertools" alternative is already done to help you:

```
>>> from audiolazy import lazy_itertools as itertools
>>> x = itertools.count()
>>> y = itertools.repeat(3)
```

```
>>> z = 2*x + y
>>> w = itertools.takewhile(lambda (idx, el): idx < 10, enumerate(z))
>>> list(el for idx, el in w)
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

All operations over Stream objects are lazy and not thread-safe.

See Also:

thub (page 59) "Tee" hub to help using the Streams like numbers in equations and filters.

tee (page 36) Just like itertools.tee, but returns a tuple of Stream instances.

Stream.tee (page 56) Keeps the Stream usable and returns a copy to be used safely.

Stream.copy (page 56) Same to `Stream.tee`.

In that example, after declaring z as function of x and y, you should not use x and y anymore. Use the thub() or the tee() functions, or perhaps the x.tee() or x.copy() Stream methods instead, if you need to use x again otherwhere.

__add__(*other*)

__and__(*other*)

__call__(**args*, ***kwargs*)
Returns the results from calling elementwise (where each element is assumed to be callable), with the same arguments.

__div__(*other*)

__eq__(*other*)

__floordiv__(*other*)

__ge__(*other*)

__getattr__(*name*)
Returns a Stream of attributes or methods, got in an elementwise fashion.

__gt__(*other*)

__ignored_classes__ = (<class 'audiolazy.lazy_filters.LinearFilter'>, <class 'audiolazy.lazy_filters.ZFilter'>, <cl

__init__(**dargs*)
Constructor for a Stream.

Parameters ***dargs** – The parameters should be iterables that will be chained together. If they're not iterables, the stream will be an endless repeat of the given elements. If any parameter is a generator and its contents is used elsewhere, you should use the "tee" (Stream method or itertools function) before.

All operations that works on the elements will work with this iterator in a element-wise fashion (like Numpy 1D arrays). When the stream sizes differ, the resulting stream have the size of the shortest operand.

Examples: A finite sequence:

```
>>> x = Stream([1,2,3]) + Stream([8,5]) # Finite constructor
>>> x
<audiolazy.lazy_stream.Stream object at 0x...>
>>> tuple(x)
(9, 7)
```

But be careful:

```
>>> x = Stream(1,2,3) + Stream(8,5) # Periodic constructor
>>> x
<audiolazy.lazy_stream.Stream object at 0x...>
>>> x.take(15) # Don't try "tuple" or "list": this Stream is endless!
[9, 7, 11, 6, 10, 8, 9, 7, 11, 6, 10, 8, 9, 7, 11]
```

__invert__()

__iter__()
Returns the Stream contents iterator.

__le__(*other*)

__lshift__(*other*)

__lt__(*other*)

__metaclass__
alias of StreamMeta (page 53)

__mod__(*other*)

__mul__(*other*)

__ne__(*other*)

__neg__()

__nonzero__()
Boolean value of a stream, called by the bool() built-in and by "if" tests. As boolean operators "and", "or" and "not" couldn't be overloaded, any trial to cast an instance of this class to a boolean should be seen as a mistake.

__or__(*other*)

__pos__()

__pow__(*other*)

__radd__(*other*)

__rand__(*other*)

__rdiv__(*other*)

__rfloordiv__(*other*)

__rlshift__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__ror__(*other*)

__rpow__(*other*)

__rrshift__(*other*)

__rshift__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__rxor__(*other*)

__sub__(*other*)

__truediv__(*other*)

__xor__(*other*)

append(**other*)
Append self with other stream(s). Chaining this way has the behaviour:

```
self = Stream(self, *others)
```

blocks(**args*, ***kwargs*)
Interface to apply audiolazy.blocks directly in a stream, returning another stream. Use keyword args.

copy()
Returns a "T" (tee) copy of the given stream, allowing the calling stream to continue being used.

filter(*func*)
A lazy way to skip elements in the stream that gives False for the given function.

map(*func*)
A lazy way to apply the given function to each element in the stream. Useful for type casting, like:

```
>>> from audiolazy import count
>>> count().take(5)
[0, 1, 2, 3, 4]
>>> my_stream = count().map(float)
>>> my_stream.take(5) # A float counter
[0.0, 1.0, 2.0, 3.0, 4.0]
```

classmethod **register_ignored_class**(*ignore*)

take(*n=None*, *constructor=<type 'list'>*)
Returns a container with the n first elements from the Stream, or less if there aren't enough. Use this without args if you need only one element outside a list.

Parameters

- **n** – Number of elements to be taken. Defaults to None.
- **constructor** – Container constructor function that can receie a generator as input. Defaults to `list`.

Returns The first `n` elements of the Stream sequence, created by the given constructor unless `n == None`, which means returns the next element from the sequence outside any container. If `n` is None, this can raise StopIteration due to lack of data in the Stream. When `n` is an integer, there's no such exception.

Examples:

```
>>> Stream(5).take(3) # Three elements
[5, 5, 5]
>>> Stream(1.2, 2, 3).take() # One element, outside a container
1.2
>>> Stream(1.2, 2, 3).take(1) # With n = 1 argument, it'll be in a list
[1.2]
>>> Stream(1.2, 2, 3).take(1, constructor=tuple) # Why not a tuple?
(1.2,)
>>> Stream([1, 2]).take(3) # More than the Stream size, n is integer
[1, 2]
>>> Stream([]).take() # More than the Stream size, n is None
Traceback (most recent call last):
  ...
StopIteration
```

Note: You should avoid using take() as if this would be an iterator. Streams are iterables that can be easily part of a "for" loop, and their iterators (the ones automatically used in for loops) are slightly faster. Use iter() builtin if you need that, instead, or perhaps the blocks method.

---

tee()
Returns a "T" (tee) copy of the given stream, allowing the calling stream to continue being used.

avoid_stream(*cls*)
Decorator to a class whose instances should avoid casting to a Stream when used with operators applied to them.

tostream(*func*)
Decorator to convert the function output into a Stream. Useful for generator functions.

class ControlStream(*value*)
Bases: audiolazy.lazy_stream.Stream (page 53)

A Stream that yields a control value that can be changed at any time. You just need to set the attribute "value" for doing so, and the next value the Stream will yield is the given value.

Examples:

```
>>> cs = ControlStream(7)
>>> data = Stream(1, 3) # [1, 3, 1, 3, 1, 3, ...] endless iterable
>>> res = data + cs
>>> res.take(5)
[8, 10, 8, 10, 8]
>>> cs.value = 9
>>> res.take(5)
[12, 10, 12, 10, 12]
```

__add__(*other*)

__and__(*other*)

__div__(*other*)

__eq__(*other*)

__floordiv__(*other*)

__ge__(*other*)

__gt__(*other*)

__init__(*value*)

__invert__()

__le__(*other*)

__lshift__(*other*)

__lt__(*other*)

__mod__(*other*)

__mul__(*other*)

__ne__(*other*)

__neg__()

__or__(*other*)

__pos__()

__pow__(*other*)

__radd__(*other*)

__rand__(*other*)

__rdiv__(*other*)

__rfloordiv__(*other*)

__rlshift__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__ror__(*other*)

__rpow__(*other*)

__rrshift__(*other*)

__rshift__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__rxor__(*other*)

__sub__(*other*)

__truediv__(*other*)

__xor__(*other*)

exception MemoryLeakWarning
Bases: exceptions.Warning

A warning to be used when a memory leak is detected.

class StreamTeeHub(*data*, *n*)
Bases: audiolazy.lazy_stream.Stream (page 53)

A Stream that returns a different iterator each time it is used.

See Also:

thub (page 59) Auto-copy "tee hub" and helpful constructor alternative for this class.

__add__(*other*)

__and__(*other*)

__del__()

__div__(*other*)

__eq__(*other*)

__floordiv__(*other*)

__ge__(*other*)

__gt__(*other*)

__init__(*data*, *n*)

__invert__()

__iter__()

__le__(*other*)

__lshift__(*other*)

__lt__(*other*)

__mod__(*other*)

__mul__(*other*)

__ne__(*other*)

__neg__()

__or__(*other*)

__pos__()

__pow__(*other*)

__radd__(*other*)

__rand__(*other*)

__rdiv__(*other*)

__rfloordiv__(*other*)

__rlshift__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__ror__(*other*)

__rpow__(*other*)

__rrshift__(*other*)

__rshift__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__rxor__(*other*)

__sub__(*other*)

__truediv__(*other*)

__xor__(*other*)

thub(*data*, *n*)

Tee or "T" hub auto-copier to help working with Stream instances as well as with numbers.

Parameters

- **data** – Input to be copied. Can be anything.
- **n** – Number of copies.

Returns A StreamTeeHub instance, if input data is iterable. The data itself, otherwise.

Examples:

```
>>> def sub_sum(x, y):
...   x = thub(x, 2) # Casts to StreamTeeHub, when needed
...   y = thub(y, 2)
...   return (x - y) / (x + y) # Return type might be number or Stream
```

With numbers:

```
>>> sub_sum(1, 1)
0
```

Combining number with iterable:

```
>>> sub_sum(3., [1, 2, 3])
<audiolazy.lazy_stream.Stream object at 0x...>
>>> list(sub_sum(3., [1, 2, 3]))
[0.5, 0.2, 0.0]
```

Both iterables (the Stream input behaves like an endless [6, 1, 6, 1, ...]):

```
>>> list(sub_sum([4., 3., 2., 1.], [1, 2, 3]))
[0.6, 0.2, -0.2]
>>> list(sub_sum([4., 3., 2., 1.], Stream(6, 1)))
[-0.2, 0.5, -0.5, 0.0]
```

This function can also be used as a an alternative to the Stream constructor when your function has only one parameter, to avoid casting when that's not needed:

```
>>> func = lambda x: 250 * thub(x, 1)
>>> func(1)
250
>>> func([2] * 10)
<audiolazy.lazy_stream.Stream object at 0x...>
>>> func([2] * 10).take(5)
[500, 500, 500, 500, 500]
```

class **Streamix**(*keep=False*, *zero=0.0*)

Bases: audiolazy.lazy_stream.Stream (page 53)

Stream mixer of iterables.

Examples: With integer iterables:

```
>>> s1 = [-1, 1, 3, 2]
>>> s2 = Stream([4, 4, 4])
>>> s3 = tuple([-3, -5, -7, -5, -7, -1])
>>> smix = Streamix(zero=0) # Default zero is 0.0, changed to keep integers
>>> smix.add(0, s1) # 1st number = delta time (in samples) from last added
>>> smix.add(2, s2)
>>> smix.add(0, s3)
>>> smix
<audiolazy.lazy_stream.Streamix object at ...>
>>> list(smix)
[-1, 1, 4, 1, -3, -5, -7, -1]
```

With time constants:

```
>>> from audiolazy import sHz, line
>>> s, Hz = sHz(10) # You probably will use 44100 or something alike, not 10
>>> sdata = list(line(2 * s, 1, -1, finish=True))
>>> smix = Streamix()
>>> smix.add(0.0 * s, sdata)
>>> smix.add(0.5 * s, sdata)
>>> smix.add(1.0 * s, sdata)
>>> result = [round(sm, 2) for sm in smix]
>>> len(result)
35
>>> 0.5 * s # Let's see how many samples this is
5.0
>>> result[:7]
[1.0, 0.89, 0.79, 0.68, 0.58, 1.47, 1.26]
>>> result[10:17]
[0.42, 0.21, 0.0, -0.21, -0.42, 0.37, 0.05]
>>> result[-1]
-1.0
```

See Also:

ControlStream **(page 57)** Stream (iterable with operators)

sHz **(page 51)** Time in seconds (s) and frequency in hertz (Hz) constants from sample rate in samples/second.

__add__(*other*)

__and__(*other*)

__div__(*other*)

__eq__(*other*)

__floordiv__(*other*)

__ge__(*other*)

__gt__(*other*)

__init__(*keep=False*, *zero=0.0*)

__invert__()

__le__(*other*)

__lshift__(*other*)

__lt__(*other*)

__mod__(*other*)

__mul__(*other*)

__ne__(*other*)

__neg__()

__or__(*other*)

__pos__()

__pow__(*other*)

__radd__(*other*)

__rand__(*other*)

__rdiv__(*other*)

__rfloordiv__(*other*)

__rlshift__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__ror__(*other*)

__rpow__(*other*)

__rrshift__(*other*)

__rshift__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__rxor__(*other*)

__sub__(*other*)

__truediv__(*other*)

__xor__(*other*)

add(*delta*, *data*)

Adds (enqueues) an iterable event to the mixer.

Parameters

- **delta** – Time in samples since last added event. This can be zero and can be float. Use "s" object from sHz for time conversion.

- **data** – Iterable (e.g. a list, a tuple, a Stream) to be "played" by the mixer at the given time delta.

See Also:

`sHz` **(page 51)** Time in seconds (s) and frequency in hertz (Hz) constants from sample rate in samples/second.

2.14 `lazy_synth` Module

Simple audio/stream synthesis module

Summary of module contents:

Name	Description
modulo_counter	Creates a lazy endless counter stream with the given modulo, i.e., its values ranges from 0. to the given "modulo", somewhat equivalent to:
line	Finite Stream with a straight line, could be used as fade in/out effects.
fadein	Linear fading in.
fadeout	Linear fading out. Multiply by this one at end to finish and avoid clicks.
attack	Linear ADS fading attack stream generator, useful to be multiplied with a given stream.
ones	Ones stream generator. You may multiply your endless stream by this to enforce an end to it.
zeros	Zeros/zeroes stream generator. You may sum your endless stream by this to enforce an end to it.
zeroes	Zeros/zeroes stream generator. You may sum your endless stream by this to enforce an end to it.
adsr	Linear ADSR envelope.
white_noise	White noise stream generator.
TableLookupMeta	Table lookup metaclass. This class overloads all operators to the TableLookup class, applying them to the table contents, elementwise. Table length and number of cycles should be equal for this to work.
TableLookup	Table lookup synthesis class, also allowing multi-cycle tables as input.
DEFAULT_TABLE_SIZE	`65536`
sin_table	<audiolazy.lazy_synth.TableLookup object at 0x...>
saw_table	<audiolazy.lazy_synth.TableLookup object at 0x...>
sinusoid	Sinusoid based on the optimized math.sin
impulse	Impulse stream generator.
karplus_strong	Karplus-Strong "digitar" synthesis algorithm.

`modulo_counter`(**args*, ***kwargs*)

Creates a lazy endless counter stream with the given modulo, i.e., its values ranges from 0. to the given "modulo", somewhat equivalent to:

Stream(itertools.count(start, step)) % modulo

Yet the given step can be an iterable, and doen't create unneeded big ints. All inputs can be float. Input order remembers slice/range inputs. All inputs can also be iterables. If any of them is an iterable, the end of this counter happen when there's no more data in one of those inputs. to continue iteration.

`line`(**args*, ***kwargs*)

Finite Stream with a straight line, could be used as fade in/out effects.

Parameters

- **dur** – Duration, given in number of samples. Use the sHz function to help with durations in seconds.
- **[begin** – ...

- **end]** – First and last (or stop) values to be yielded. Defaults to [0., 1.], respectively.
- **finish** – Choose if `end` it the last to be yielded or it shouldn't be yield at all. Defauts to False, which means that `end` won't be yield. The last sample won't have "end" amplitude unless finish is True, i.e., without explicitly saying "finish=True", the "end" input works like a "stop" range parameter, although it can [should] be a float. This is so to help concatenating several lines.

Returns A finite Stream with the linearly spaced data.

Examples: With `finish = True`, it works just like NumPy `np.linspace`, besides argument order and lazyness:

```
>>> import numpy as np
>>> np.linspace(.2, .7, 6)
array([ 0.2,  0.3,  0.4,  0.5,  0.6,  0.7])
>>> line(6, .1, .7, finish=True)
<audiolazy.lazy_stream.Stream object at 0x...>
>>> list(line(6, .2, .7, finish=True))
[0.2, 0.3, 0.4, 0.5, 0.6, 0.7]
>>> list(line(6, 1, 4)) # With finish = False (default)
[1.0, 1.5, 2.0, 2.5, 3.0, 3.5]
```

Line also works with Numpy arrays and matrices

```
>>> a = np.mat([[1, 2], [3, 4]])
>>> b = np.mat([[3, 2], [2, 1]])
>>> for el in line(4, a, b):
...   print el
[[ 1.  2.]
 [ 3.  4.]]
[[ 1.5  2. ]
 [ 2.75  3.25]]
[[ 2.   2. ]
 [ 2.5  2.5]]
[[ 2.5  2. ]
 [ 2.25  1.75]]
```

And also with ZFilter instances:

```
>>> from audiolazy import z
>>> for el in line(4, z ** 2 - 5, z + 2):
...   print el
z^2 - 5
0.75 * z^2 + 0.25 * z - 3.25
0.5 * z^2 + 0.5 * z - 1.5
0.25 * z^2 + 0.75 * z + 0.25
```

Note: Amplitudes commonly should be float numbers between -1 and 1. Using line(<inputs>).append([end]) you can finish the line with one extra sample without worrying with the "finish" input.

See Also:

sHz (page 51) Second and hertz constants from samples/second rate.

fadein(*dur*)

Linear fading in.

Parameters **dur** – Duration, in number of samples.

Returns Stream instance yielding a line from zero to one.

fadeout(*dur*)
Linear fading out. Multiply by this one at end to finish and avoid clicks.

Parameters **dur** – Duration, in number of samples.

Returns Stream instance yielding the line. The starting amplitude is is 1.0.

attack(*a*, *d*, *s*)
Linear ADS fading attack stream generator, useful to be multiplied with a given stream.

Parameters

- **a** – "Attack" time, in number of samples.
- **d** – "Decay" time, in number of samples.
- **s** – "Sustain" amplitude level (should be based on attack amplitude). The sustain can be a Stream, if desired.

Returns Stream instance yielding an endless envelope, or a finite envelope if the sustain input is a finite Stream. The attack amplitude is is 1.0.

ones(**args*, ***kwargs*)
Ones stream generator. You may multiply your endless stream by this to enforce an end to it.

Parameters **dur** – Duration, in number of samples; endless if not given.

Returns Stream that repeats "1.0" during a given time duration (if any) or endlessly.

zeros(**args*, ***kwargs*)
Zeros/zeroes stream generator. You may sum your endless stream by this to enforce an end to it.

Parameters **dur** – Duration, in number of samples; endless if not given.

Returns Stream that repeats "0.0" during a given time duration (if any) or endlessly.

zeroes(**args*, ***kwargs*)
Zeros/zeroes stream generator. You may sum your endless stream by this to enforce an end to it.

Parameters **dur** – Duration, in number of samples; endless if not given.

Returns Stream that repeats "0.0" during a given time duration (if any) or endlessly.

adsr(**args*, ***kwargs*)
Linear ADSR envelope.

Parameters

- **dur** – Duration, in number of samples, including the release time.
- **a** – "Attack" time, in number of samples.
- **d** – "Decay" time, in number of samples.
- **s** – "Sustain" amplitude level (should be based on attack amplitude).
- **r** – "Release" time, in number of samples.

Returns Stream instance yielding a finite ADSR envelope, starting and finishing with 0.0, having peak value of 1.0.

white_noise(**args*, ***kwargs*)
White noise stream generator.

Parameters **dur** – Duration, in number of samples; endless if not given.

Returns Stream yielding random numbers between -1 and 1.

class **TableLookupMeta**
Bases: audiolazy.lazy_core.AbstractOperatorOverloaderMeta (page 20)

Table lookup metaclass. This class overloads all operators to the TableLookup class, applying them to the table contents, elementwise. Table length and number of cycles should be equal for this to work.

__binary__(*op_func*)

__operators__ = 'add radd sub rsub mul rmul pow rpow div rdiv mod rmod truediv rtruediv floordiv rfloordiv po

__rbinary__(*op_func*)

__unary__(*op_func*)

class TableLookup(*table*, *cycles=1*)

Bases: `object`

Table lookup synthesis class, also allowing multi-cycle tables as input.

__add__(*other*)

__and__(*other*)

__call__(*freq*, *phase=0.0*)

Returns a wavetable lookup synthesis endless stream. Play it with the given frequency and starting phase. Phase is given in rads, and frequency in rad/sample. Accepts streams of numbers, as well as numbers, for both frequency and phase inputs.

__div__(*other*)

__eq__(*other*)

__floordiv__(*other*)

__getitem__(*idx*)

Gets an item from the table from its index, which can possibly be a float. The data is linearly interpolated.

__init__(*table*, *cycles=1*)

Inits a table lookup. The given table should be a sequence, like a list. The cycles input should have the number of cycles in table for frequency calculation afterwards.

__invert__()

__len__()

__lshift__(*other*)

__metaclass__

alias of `TableLookupMeta` (page 64)

__mod__(*other*)

__mul__(*other*)

__ne__(*other*)

__neg__()

__or__(*other*)

__pos__()

__pow__(*other*)

__radd__(*other*)

__rand__(*other*)

__rdiv__(*other*)

__rfloordiv__(*other*)

__rlshift__(*other*)

__rmod__(*other*)

__rmul__(*other*)

__ror__(*other*)

__rpow__(*other*)

__rrshift__(*other*)

__rshift__(*other*)

__rsub__(*other*)

__rtruediv__(*other*)

__rxor__(*other*)

__sub__(*other*)

__truediv__(*other*)

__xor__(*other*)

harmonize(*harmonics_dict*)
Returns a "harmonized" table lookup instance by using a "harmonics" dictionary with {partial: amplitude} terms, where all "partial" keys have to be integers.

normalize()
Returns a new table with values ranging from -1 to 1, reaching at least one of these, unless there's no data.

table

sinusoid(**args*, ***kwargs*)
Sinusoid based on the optimized math.sin

impulse(**args*, ***kwargs*)
Impulse stream generator.

Parameters **dur** – Duration, in number of samples; endless if not given.

Returns Stream that repeats "0.0" during a given time duration (if any) or endlessly, but starts with one (and only one) "1.0".

karplus_strong(*freq*, *tau=20000.0*, *memory=<function white_noise at 0x21c6aa0>*)
Karplus-Strong "digitar" synthesis algorithm.

Parameters

- **freq** – Frequency, in rad/sample.
- **tau** – Time decay (up to `1/e`, or -8.686 dB), in number of samples. Defaults to 2e4. Be careful: using the default value will make duration different on each sample rate value. Use `sHz` if you need that independent from the sample rate and in seconds unit.
- **memory** – Memory data for the comb filter (delayed "output" data in memory). Defaults to the `white_noise` function.

Returns Stream instance with the synthesized data.

Note: The fractional delays are solved by exponent linearization.

See Also:

sHz **(page 51)** Second and hertz constants from samples/second rate.

white_noise **(page 64)** White noise stream generator.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z