Processamento de áudio

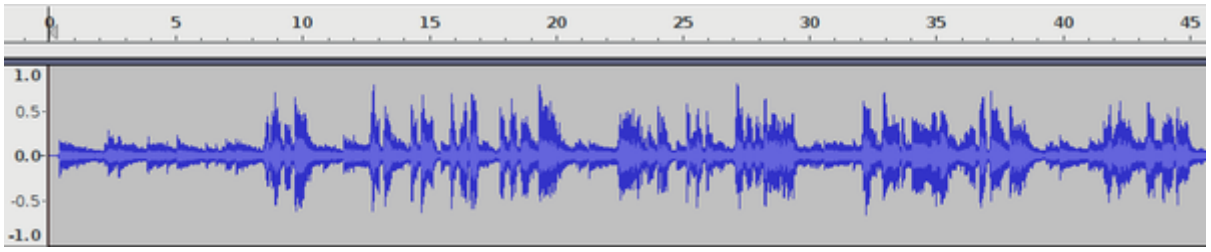
Video deste projeto

Este projeto consiste em criar filtros para arquivos de áudio em formato WAV, implementando operações como ajuste de volume, eco e normalização.

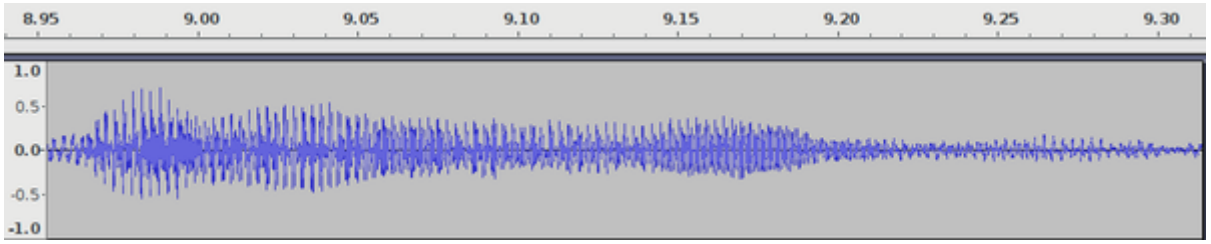
Áudio digital

Em um sistema digital, um som é normalmente codificado como um vetor de amostras (*samples*), onde cada amostra corresponde à amplitude do sinal sonoro em um instante de tempo. As figuras a seguir trazem um exemplos de sinais sonoros codificados dessa forma:

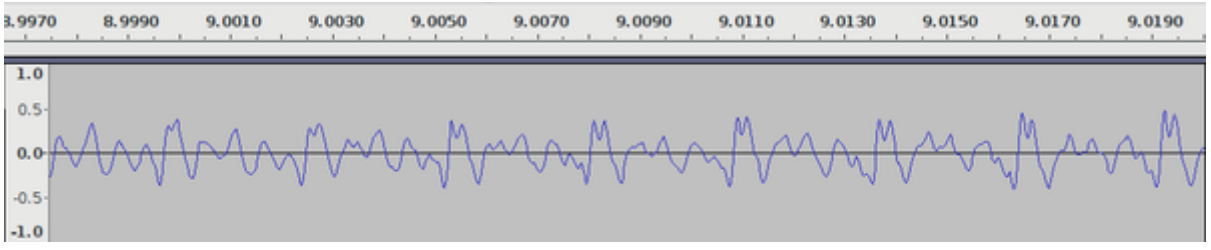
Trecho de áudio (duração 45 s):



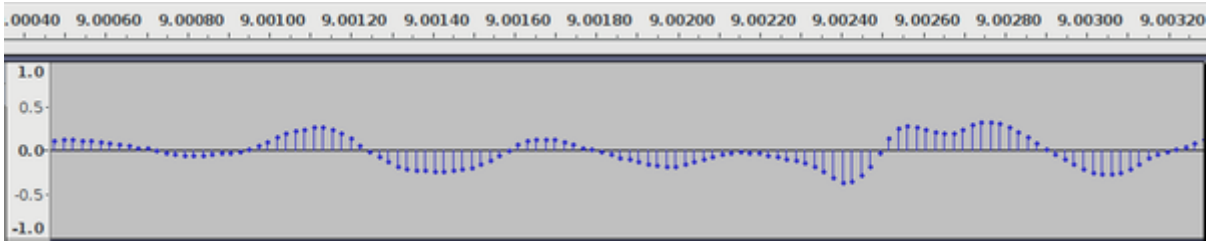
Detalhe do trecho de áudio acima (duração 300 ms):



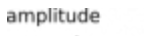
Detalhe do trecho de áudio acima (duração 30 ms):

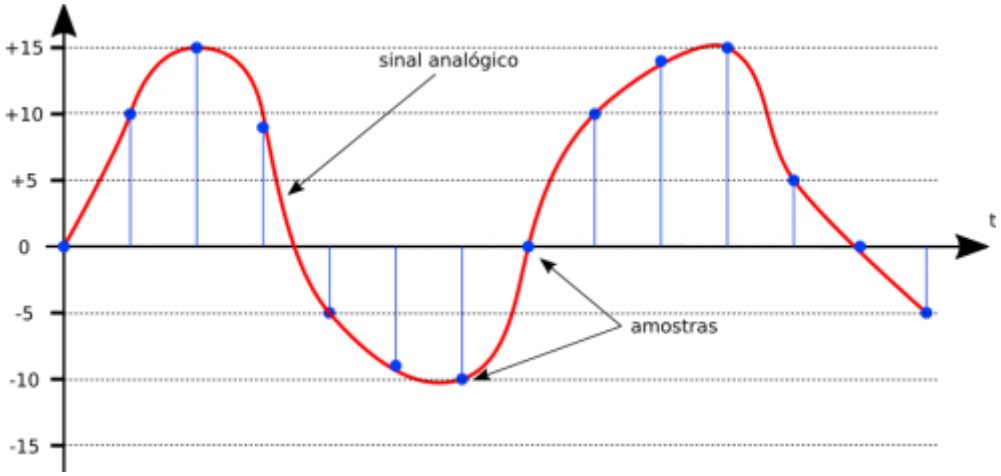


Detalhe do trecho de áudio acima (duração 0,3 ms):



Visão conceitual da amostragem de áudio:





Nessa figura podem ser identificados o **sinal analógico**, proveniente de um microfone ou outro dispositivo de captura, e as **amostras**, que correspondem ao valor do sinal medido em intervalos regulares. O sinal analógico acima seria então representado de forma aproximada pelo seguinte vetor de amostras:

```
[0, 10, 15, 9, -5, -9, -10, 0, 10, 14, 15, 5, 0, -5, ... ]
```

O padrão PCM (Pulse-Code Modulation (https://en.wikipedia.org/wiki/Pulse-code_modulation)) é um dos padrões mais simples para representar áudio em meio digital. Duas informações caracterizam um sinal codificado em PCM:

- **Taxa de amostragem**: é o número de amostras do sinal analógico feitas por segundo. Quanto maior a taxa de amostragem, melhor é a representação digital do som, sobretudo nas frequências mais elevadas (agudos). Por outro lado, taxas de amostragem elevadas implicam em arquivos maiores. Valores típicos de taxas de amostragem são 48 kHz em DVDs e 44,1 KHz em CDs de música (exemplos (https://en.wikipedia.org/wiki/Sampling_(signal_processing))).
- **Resolução**: é o número de bits usado para representar cada amostra. Quanto maior a resolução, mais precisa é a representação de cada amostra e menor a distorção do sinal. CDs de música usam 16 bits de resolução, enquanto sistemas de telefonia costumam usar 8 bits (exemplos (https://en.wikipedia.org/wiki/Audio_bit_depth)).

No caso de sons com vários canais (estéreo ou *surround*), deve existir uma sequência de amostras para cada canal de som, geralmente todas usando a mesma taxa de amostragem e resolução.

O formato WAV

Para facilitar a leitura e escrita dos arquivos de áudio, neste projeto será adotado o formato WAV (https://en.wikipedia.org/wiki/WAV), reconhecido pela maioria dos softwares de geração e reprodução de áudio. **Para este projeto será adotado o padrão WAV com codificação PCM de 16 bits com sinal**. Neste formato, cada amostra é representada por um número inteiro de 16 bits com sinal (ou seja, do tipo `int16_t` , variando entre -32768 e +32767).

Um arquivo WAV típico é organizado nos seguintes *chunks* (pedaços) básicos:

Chunk ID	tamanho (bytes)	conteúdo
`RIFF`	12	cabeçalho RIFF (*Resource Interchange File Format (https://en.wikipedia.org/wiki/Resource_Interchange_File_Format)*)
`fmt`	24	informações sobre o formato do áudio

Chunk ID	tamanho (bytes)	conteúdo
`data`	variável	amostras do(s) canal(is) de áudio

Em alguns arquivos WAV podem haver outros chunks, como LIST e INFO, que não precisam ser considerados neste projeto.

Esses *chunks* têm os seguintes campos internos:

Chunk	nome	tamanho	tipo	valor	significado
`RIFF`	ChunkID	4	`char[4]`	`"RIFF"`	constante, identifica o tipo de arquivo
	ChunkSize	4	`uint32_t`	filesize - 8	tamanho do arquivo em bytes, sem considerar *ChunkID* e *ChunkSize*
	Format	4	`char[4]`	`"WAVE"`	constante, define o formato do restante do conteúdo
`fmt`	SubChunk1ID	4	`char[4]`	`"fmt "`	constante, cabeçalho do *chunk*
	SubChunk1Size	4	`uint32_t`	16 (PCM)	tamanho deste *chunk*
	Audio format	2	`uint16_t`	1 (PCM)	codificação utilizada
	Number of channels	2	`uint16_t`	1, 2, ...	número de canais de áudio
	Sample rate	4	`uint32_t`	44100, etc	Taxa de amostragem (amostras/seg) por canal
	Byte rate	4	`uint32_t`	varia	taxa de bytes por segundo
	Block align	2	`uint16_t`	varia	número de bytes por amostra (soma todos os canais)
	Bits per sample	2	`uint16_t`	8, 16, 32, ...	bits por amostra, por canal
`data`	SubChunk2ID	4	`char[4]`	`"data"`	constante, cabeçalho do *chunk*
	SubChunk2Size	4	`uint32_t`	varia	espaço ocupado pelas amostras, em bytes
	Audio data	variável	-	-	amostras de áudio

Observe que os campos de tipo `char[4]` **não são strings**, pois strings precisam ser terminadas por `\0`, o que não ocorre aqui.

As amostras de áudio são armazenadas no *chunk* de dados (`data`) do arquivo, em sequência por canal e por tempo. Cada amostra tem um tamanho fixo em bytes, definido pelo campo *bits per sample* do cabeçalho. Considerando um áudio com dois canais (*Left* e *Right*), as amostras estarão dispostas da seguinte forma:

```
L[0], R[0], L[1], R[1], L[2], R[2], L[3], R[3], ...
```

Para ler facilmente o cabeçalho de um arquivo WAV, basta definir um *struct* com os mesmos campos, tamanhos e ordem do cabeçalho descrito acima, e então ler os primeiros bytes do arquivo para dentro desse *struct*, usando a função de leitura `fread`. Uma vez lido o *struct*, basta verificar se os campos constantes têm os valores esperados.

Documentação adicional sobre o formato WAV:

- http://soundfile.sapp.org/doc/WaveFormat/ (http://soundfile.sapp.org/doc/WaveFormat/)
- http://unusedino.de/ec64/technical/formats/wav.html (http://unusedino.de/ec64/technical/formats/wav.html)

Alguns exemplos de arquivos de áudio WAV para usar no projeto:

- Trecho de música (estéreo) “One and Only”, de *Audio Idols*
- Batida na mesa, áudio simples com 5s, bom para testar o eco.
- Numbers (mono, 22050 KHz)

Você pode gerar seus próprios arquivos WAV usando programas de processamento de áudio, como o “Audacity”, Sox, etc.

Filtros

Um filtro de áudio é um programa simples, que recebe como entrada um arquivo de áudio (que pode vir da entrada padrão *stdin*), realiza algum tipo de processamento de áudio (https://en.wikipedia.org/wiki/Audio_signal_processing) e entrega na saída um arquivo de áudio (que pode ser *stdout*). Eventuais mensagens de erro devem ser enviadas para a saída de erro (*stderr*).

Lembre-se que o filtro deve atuar em todos os canais do áudio da entrada.

Informações

Este programa não é exatamente um filtro, pois produz como saída uma listagem das principais informações do áudio informado como entrada.

Forma de chamada:

```
wavinfo -i input
```

Exemplo de saída (para o arquivo `music.wav` de exemplo acima):

```
$ wavinfo -i music.wav

riff tag        (4 bytes): "RIFF"
riff size       (4 bytes): 8061776
wave tag        (4 bytes): "WAVE"
form tag        (4 bytes): "fmt "
fmt_size        (4 bytes): 16
audio_format    (2 bytes): 1
num_channels    (2 bytes): 2
sample_rate     (4 bytes): 44100
byte_rate       (4 bytes): 176400
block_align     (2 bytes): 4
bits_per_sample (2 bytes): 16
data tag        (4 bytes): "data"
data size       (4 bytes): 8061740
bytes per sample         : 2
samples per channel      : 2015435
```

Ajuste de volume

O filtro de ajuste de volume permite aumentar ou diminuir o volume de áudio do arquivo, de acordo com um fator de ajuste V (0.0 ≤ V ≤ 10.0, com default em 1.0). Ele consiste basicamente em multiplicar o valor de cada amostra de áudio por V.

Forma de chamada (`-l` : *level*):

```
wavvol -l V -i input -o output
```

Exemplo de uso e saída: music-vol.wav

```
wavvol -l 0.1 -i music.wav -o music-vol.wav
```

Normalização

Este filtro faz a normalização do áudio, ou seja, o ajuste automático de volume. Para tal, é necessário encontrar o valor do maior pico no sinal de áudio (em todos os canais) e usá-lo para calcular um fator de ajuste, de modo que todas as amostras de todos os canais fiquem no intervalo de 16 bits com sinal [-32767 … +32767].

Forma de chamada:

```
wavnorm -i input -o output
```

Reversão

Este filtro produz como saída um áudio invertido, ou seja, “de trás para a frente” em relação ao áudio de entrada.

Forma de chamada:

```
wavrev -i input -o output
```

Exemplo de uso e saída: music-rev.wav

```
wavrev -i music.wav -o music-rev.wav
```

Muitos músicos se divertiram com esse efeito, como nesta música do grupo Pink Floyd (https://youtu.be/CiveqbMsHwQ)!

Eco

Este filtro produz como saída um áudio com eco. O eco é controlado pelos parâmetros *delay* (inteiro ≥ 0, default 1000 ms), que define o atraso do eco em milissegundos, e `level` (0.0 ≤ level ≤ 1.0, default 0.5), que define o nível do eco em relação ao sinal original.

O efeito de eco pode ser definido por esta equação: $sample_t = sample_t + (level \times sample_{t-delay})$

Forma de chamada (`-t` : *time* em ms, `-l` : *level* em %):

```
wavecho -t delay -l level -i input -o output
```

Exemplo de uso e saída (ambos com t=500ms e l=50%): batida-echo.wav music-echo.wav

```
wavecho -t 500 -l 0.5 -i music.wav -o music-echo.wav
```

Experimente aplicar este filtro a um arquivo de voz, com um atraso bem pequeno (10 a 50 ms) 😃

Estéreo ampliado

O filtro de estéreo ampliado permite aumentar a separação de canais em um sinal estéreo, gerando um som mais aberto. Este filtro só pode ser aplicado a sinais estéreo (com 2 canais).

Sendo R(t) e L(t) as amostras de entrada do canal direito e esquerdo em um instante t, a saída R'(t), L'(t) é calculada da seguinte forma:

```
diff  = R(t) - L(t)
R'(t) = R(t) + k * diff
L'(t) = L(t) - k * diff
```

onde `diff` é o sinal de diferença entre os dois canais e `k` é o fator de ampliação do efeito estéreo (0.0 ≤ k ≤ 10.0, default 1.0).

Forma de chamada (`-l` : *level*):

```
wavwide -l k -i input -o output
```

Exemplo de uso e saída: music-wide.wav

```
cat music.wav | wavvol -l 0.5 | wavwide -l 5 | wavnorm > music-wide.wav
```

O exemplo acima mostra o uso de vários filtros concatenados: no início o volume é reduzido para evitar a saturação do áudio, em seguida o efeito *wide* é aplicado e depois o áudio é normalizado e salvo no arquivo de saída. Todas as transferências de dados são feitas por *stdin* e *stdout*.

Concatenação

O filtro de concatenação recebe como entrada **um ou mais arquivos de áudio** e gera uma saída contendo a concatenação das entradas na sequência indicada.

Forma de chamada:

```
wavcat arq1.wav arq2.wav arq3.wav ... -o output
```

A combinação de sinais de áudio com taxas de amostragem diferentes exige um procedimento chamado *reamostragem* (resampling (https://ccrma.stanford.edu/~jos/resample/)), que pode ser complexo. Por isso, restrinja seu programa a arquivos com a mesma taxa de amostragem.

Mistura

O filtro de mistura (mixagem) recebe como entrada um ou mais arquivos de áudio e gera uma saída contendo a mistura (mixagem) das entradas.

Forma de chamada:

```
wavmix arq1.wav arq2.wav arq3.wav ... -o output
```

Ao somar as amostras, cuide para não saturar a saída, o que pode gerar distorção (clipping (https://en.wikipedia.org/wiki/Clipping_(audio))).

Outros filtros (ideias)

- extrair um canal de um áudio stereo
- juntar dois áudios mono em um áudio stereo
- recortar um áudio de t1 a t2
- aplicar um atraso *t* a um áudio (t pode ser negativo, para adiantar o áudio)
- *smoothing*: trocar cada amostra pela média entre ela e suas vizinhas
- efeito *phaser*, *vibrato* e outros
- compressor
- filtro de ruído: reduzir o volume de todas as amostras com nível abaixo de um certo valor (dado em

  % do volume máximo).
- … (outras ideias são bem-vindas)

Mais conteúdo sobre efeitos de áudio:

- 07_Audio_Effects.pdf
- 10_CM0268_Audio_FX.pdf
- 0470665998
- labs-2014.pdf
- …

Atividade

O projeto consiste em implementar os filtros acima definidos.

Requisitos

- Cada filtro é um comando separado. Por exemplo, o filtro de volume deve ser implementado em um arquivo `wavvol.c` que gera um executável `wavvol` , e assim por diante.
- Os filtros devem aceitar como entrada sons no formato WAV PCM 16 bits com sinal e devem gerar como saída sons nesse mesmo formato.
- As rotinas comuns (leitura/escrita de arquivos, tratamento da linha de comando, etc) devem ser implementadas em arquivos separados, cujos cabeçalhos são incluídos nos arquivos de implementação dos filtros.
- Sempre que possível, as informações do som necessárias às funções devem ser transferidas como parâmetros (por valor ou por referência, dependendo da situação). Minimizar o uso de variáveis globais.
- Use alocação dinâmica de memória para os sons, para poder processar arquivos grandes. Só aloque a memória para as amostras após encontrar o número de amostras no cabeçalho.
- Construir um `Makefile` para o projeto:
  - Ao menos os alvos `all` (default), `clean` e `purge` .
  - CFLAGS = `-Wall`
  - Compilar e ligar separadamente (gerar arquivos `.o` intermediários)
- O que deve ser entregue ao professor:
  - arquivos `.c` e `.h`
  - arquivo `Makefile`
  - **não enviar** os sons de teste

Para simplificar a implementação e evitar erros, sugere-o o uso dos tipos inteiros de tamanho fixo (`int16_t` , etc), acessíveis através do arquivo `inttypes.h` .

Linha de comando

- A opção `-i` indica o nome do arquivo de entrada; se não for informado, deve-se usar a entrada padrão (*stdin*).
- A opção `-o` indica o nome do arquivo de saída; se não for informado, deve-se usar a saída padrão (*stdout*).
- Todas as mensagens de erro devem ser enviadas para a saída de erro (*stderr*).

Essas opções podem ser usadas em qualquer combinação, ou seja:

```
// entrada e saída em arquivos
wavvol -i inputfile.wav  -o outputfile.wav
wavvol -o outputfile.wav -i inputfile.wav

// entrada em arquivo, saída em stdout, vice-versa ou ambos
wavvol -i inputfile.wav  > outputfile.wav
wavvol -o outputfile.wav < inputfile.wav
wavvol  < inputfile.wav  > outputfile.wav

// as opções podem estar em qualquer ordem
wavvol -l 0.3 -i inputfile.wav -o outputfile.wav
wavvol -i inputfile.wav -l 0.3 -o outputfile.wav
wavvol -o outputfile.wav -i inputfile.wav -l 0.3
```

Para ler e tratar mais facilmente as opções da linha de comando, sugere-se usar funções já prontas para isso, como `getopt` ou `arg_parse` (link (https://www.gnu.org/software/libc/manual/html_node/Parsing-Program-Arguments.html#Parsing-Program-Arguments))

Como os filtros devem tratar a entrada e saída padrão, é possível combinar filtros usando *pipes* UNIX. Por exemplo, podemos usar pipes para construir o efeito Reverse Echo (https://en.wikipedia.org/wiki/Reverse_echo), muito apreciado por alguns grupos de Rock:

```
wavrev -i input.wav | wavecho -t 500 -l 0.5 | wavrev -o output.wav
```

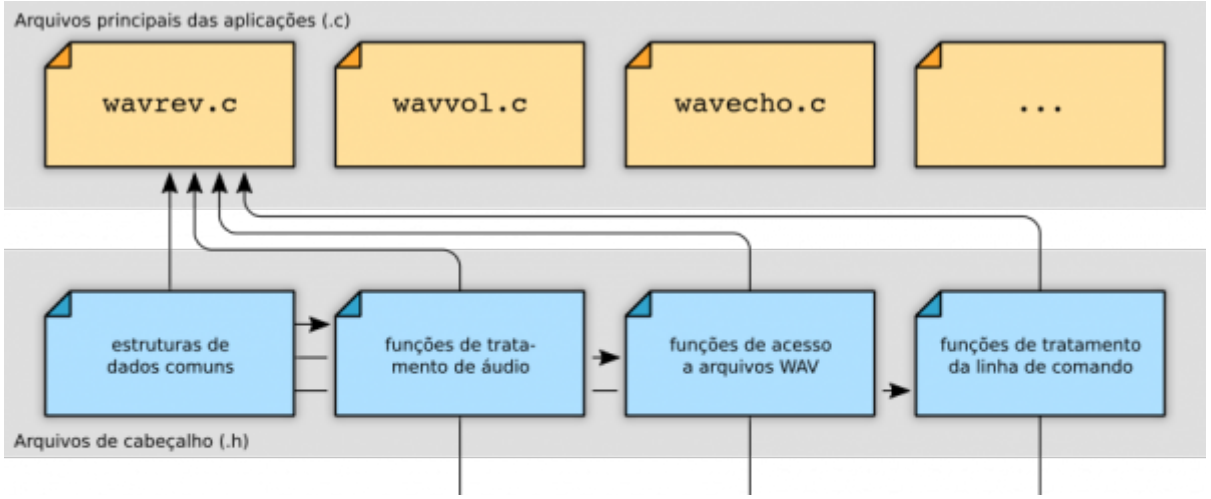
Caso a adição de eco provoque *clipping*, pode-se atenuar o sinal antes de processá-lo:

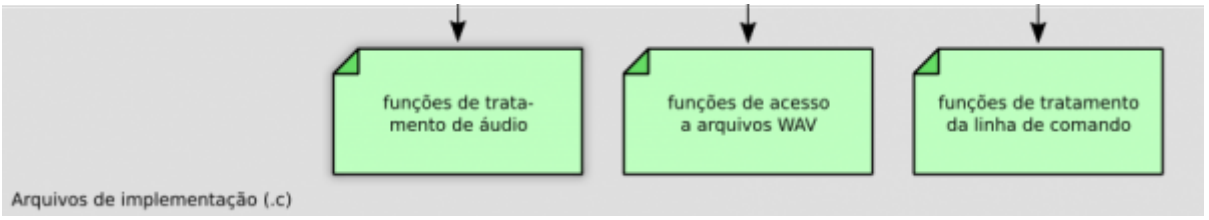
```
wavvol -l 0.5 -i input.wav | wavrev | wavecho -t 500 -l 0.5 | wavrev | wavnorm -o o
utput.wav
```

Para tocar facilmente os arquivos de áudio, pode-se usar os comandos `play`, `aplay` ou `paplay` no terminal do Linux, ou o *preview* de arquivos no gerenciador de arquivos do ambiente gráfico.

Estrutura do código-fonte

O código-fonte deve ser estruturado em diversos arquivos `.c` e `.h` que contenham as funcionalidades a serem implementadas. A figura abaixo traz uma uma **sugestão de estrutura** para o código-fonte (as setas correspondem a `include` s):





prog2/processamento_de_audio.txt Última modificação: 2020/11/16 15:24 por maziero