CSF13: Fundamentos de Programação 1

Projeto Prático #2 (Parte1: Vetores)

Professoras: Leyza Dorini e Myriam Delgado *Estudante:* _____

1. Equipes: o trabalho deve ser feito em equipe de 3 alunos. A discussão entre colegas para compreender a estrutura do trabalho e esta especificação é recomendada e estimulada, mas cada equipe deve apresentar suas próprias soluções para os problemas propostos. Indícios de fraude (cópia) podem levar à avaliação especial[1]. Não compartilhe códigos (fonte nem pseudocódigos)!!!

2. Prazo final de entrega (Parte1+Parte2): 06/06/2019 - 23:55 (exclusivamente via Moodle).

3. Formato para entrega: cada equipe deve entregar, através da página da disciplina no Moodle, uma pasta comprimida (arquivo `.zip`) com:

   - Um arquivo no formato PDF chamado `pp02-x-y.pdf`, em que `x` e `y` são os números de matrícula dos alunos. Este arquivo deve conter um relatório breve (em torno de 2 a 3 páginas), descrevendo (a) a contribuição de cada membro da equipe, (b) detalhes extras sobre o funcionamento das funções entregues que não tenham ficado claros nos comentários, (c) os desafios encontrados, e (d) a forma como eles foram superados. Use o template de artigos da SBC.
   - Diversos arquivos chamados `pp02-Q-x-y.c`, em que `x` e `y` são os números de matrícula dos alunos e Q a questão específica. O arquivo deve conter as implementações das funções pedidas (com cabeçalhos idênticos aos especificados, quando a questão fornecer). As funções pedidas podem fazer uso de outras funções, sejam funções da biblioteca-padrão, funções criadas pelos autores, ou as próprias funções pedidas (i.e. uma função pedida pode invocar outras funções pedidas!). Os autores do arquivo devem estar identificados no início, através de comentários.

4. Fique atento aos demais requisitos especificados na disciplina (documentação, nomes de funções e variáveis, etc).

Este projeto prático 02 envolverá a implementação de funções para processamento de sinais de áudio. Os enunciados, bem como os códigos-fonte para manipulação de arquivos de áudio, foram gentilmente cedidos pelo professor Bogdan T. Nassu.

Instruções para "montar" o projeto

Serpa disponibilizado um "pacote" contendo os seguintes arquivos:

- Um arquivo `trabalho2.h`, contendo as declarações das funções, que deve ser incluído no seu arquivo `.c` através da diretiva `#include`. 2) Arquivos contendo declarações e funções para a manipulação de arquivos no formato Microsoft Wave (`.wav`): `wavfile.c` e `wavfile.h`.
- Exemplos de programas para teste e arquivos de áudio demonstrando os resultados produzidos. Os arquivos de áudio podem ser obtidos no moodle.
- Para compilar e usar no seu programa as funções presentes no arquivo `wavfile.c`, crie na IDE usada um projeto, e adicione o arquivo ao mesmo. Crie outro arquivo `.c` para as suas funções. Cada arquivo contendo um exemplo de programa contém uma função main, portanto apenas um dos exemplos deve ser incluído no projeto por vez. Recomenda-se também a criação de outros programas para testar as funções.

[1] Indícios de fraude ou de divisão desigual do trabalho podem levar a uma avaliação especial, com os alunos sendo convocados e questionados sobre aspectos referentes aos algoritmos usados e à implementação. Além disso, alguns alunos podem ser selecionados para a avaliação especial, mesmo sem indícios de fraude, caso exista uma grande diferença entre a nota do trabalho e de uma prova, ou se a explicação sobre o funcionamento de uma função for pouco clara.

Sobre as funções do módulo `wavfile`

O módulo `wavfile` implementa o tipo `WavHeader` e um conjunto de funções que podem ser usadas para testar as funções do trabalho. O tipo `WavHeader` é usado para armazenar informações sobre um arquivo no formato Microsoft Wave (`.wav`). Ele é implementado como uma `struct`, um conceito que ainda não vimos em aula.

Por enquanto, apenas trate este tipo como um dos tipos primitivos da linguagem C (`int`, `float`, etc.), seguindo os exemplos. Para usar o tipo `WavHeader` e as funções auxiliares, você deve incluir o arquivo `wavfile.h`, através da diretiva `#include`, no arquivo `.c` que usa o tipo ou as funções declaradas. As funções para manipulação de arquivos trabalham apenas com um sub-conjunto do formato Microsoft Wave. Use-as sempre com áudio não comprimido, no formato PCM, com 16 bits por amostra.

(A) `readWavFile()`

```
int readWavFile (char* filename, WavHeader* header, double** data_l, double** data_r);
```

Abre um arquivo wav e lê o seu conteúdo.

```
Parâmetros:
- char* filename: arquivo a ser lido.
- WavHeader* header: parâmetro de saída para os dados do cabeçalho. double** data_l:
  parâmetro de saída, é um ponteiro para um vetor dinâmico, onde manteremos os dados
  lidos para o canal esquerdo. O vetor será alocado nesta função, lembre-se de desalocá-lo
  quando ele não for mais necessário! Se ocorrerem erros, o vetor NÃO estará alocado quando
  a função retornar.
- double** data_r: igual ao anterior, mas para o canal direito. Se o arquivo for mono, o
  vetor será == NULL.
Valor de Retorno: o número de amostras do arquivo, 0 se ocorrerem erros.
```

(B) `writeWavFile()`

```
int writeWavFile (char* filename, WavHeader* header, double* data_l, double* data_r);
```

Escreve os dados de áudio em um arquivo wav. Esta função NÃO testa os dados do cabeçalho, então dados inconsistentes não serão detectados. Cuidado!

```
Parâmetros:
- char* filename: nome do arquivo a ser escrito.
- WavHeader* header: ponteiro para os dados do cabeçalho.
- double* data_l: dados do canal esquerdo.
- double* data_r: dados do canal direito. Ignorado se o arquivo for mono.
Valor de Retorno: 1 se não ocorreram erros, 0 do contrário.
```

(C) `generateSignal()`

```
WavHeader generateSignal (unsigned long* n_samples, unsigned short num_channels,
                          unsigned long sample_rate, double** data_l, double** data_r);
```

Gera um sinal de conteúdo indeterminado. Use esta função se, em vez de ler os dados de um arquivo, você quiser gerar um sinal.

```
Parâmetros:
- unsigned long* n_samples: ponteiro para uma variável contendo o número de amostras a gerar
  para cada canal. Se for maior que o permitido, a função gera um sinal menor que o pedido e
  modifica o valor da variável.
- unsigned short num_channels: número de canais a gerar. Precisa ser 1 ou 2, qualquer outro
  valor é automaticamente tratado como 1.
- unsigned long sample_rate: taxa de amostragem. Se for == 0, é automaticamente ajustada para 44.1kHz.
- double** data_l: parâmetro de saída, é um ponteiro para um vetor dinâmico, onde manteremos os
  dados para o canal esquerdo. O vetor será alocado nesta função, lembre-se de desalocá-lo
  quando ele não for mais necessário!
- double** data_r: igual ao anterior, mas para o canal direito. Se num_channels==1, o vetor será ==NULL.
Valor de Retorno: o cabeçalho para o sinal gerado.
```

(D) `generateRandomData()`

```
void generateRandomData (double* data, unsigned long n_samples);
```

Preenche um vetor dado com dados aleatórios (ruído). A função NÃO inicia a semente aleatória.

```
Parâmetros:
double* data: vetor a preencher.
unsigned long n_samples: número de amostras no vetor.
Valor de Retorno: NENHUM
```

(E) `writeSamplesAsText()`

```
int writeSamplesAsText (char* filename, double* data, unsigned long n_samples);
```

Salva as amostras em um arquivo de texto, uma amostra por linha. Esta função é útil para visualizar os dados e depurar o trabalho.

```
Parâmetros:
- char* filename: nome do arquivo a ser escrito.
- double* data: vetor de dados contendo as amostras.
- unsigned long n_samples: número de amostras no vetor.
Valor de Retorno: 1 se não ocorreram erros, 0 do contrário.
```

Tarefa

Sua tarefa consiste em

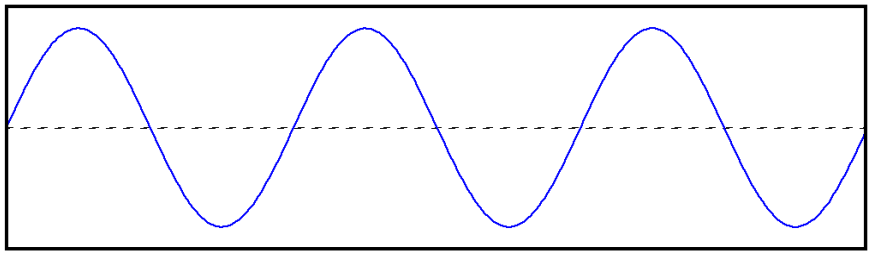
1. Implementar as seguintes funções:
   - `void mudaGanho (double* dados, unsigned long n_amostras, double ganho);`
   - `void ruidoPeriodico (double* dados, unsigned long n_amostras, unsigned short intervalo);`
   - `void removeRuido (double* dados, unsigned long n_amostras);`
   - `void simulaSubAmostragem (double* dados, unsigned long n_amostras, unsigned short n_constantes);`
2. Realizar algumas modificações nos arquivos testexxx.c conforme solicitado nos comentários ao longo dos códigos contidos nestes arquivos.

Função 1

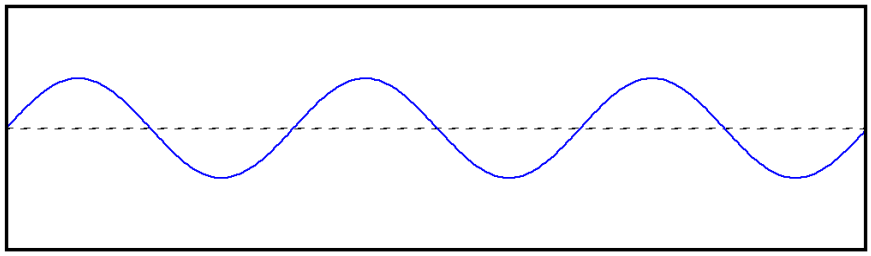
```
void mudaGanho (double* dados, unsigned long n_amostras, double ganho);

Parâmetros: double* dados: vetor de dados.
            unsigned long n_amostras: número de amostras no vetor.
            double ganho: modificador do ganho.
```

Esta função deve modificar o ganho (volume) de um sinal. Para isso, basta multiplicar cada amostra por um parâmetro `ganho` dado. Se `ganho` = 1, o sinal não é alterado; se `ganho` > 1, o volume do sinal será aumentado; se 0 < `ganho` < 1, o volume do sinal será reduzido; se `ganho` = 0, o sinal será silenciado. Valores negativos para o ganho terão o mesmo efeito, mas invertendo a fase do sinal. Não é preciso tratar de casos nos quais o valor das amostras extrapola o limite máximo permitido pela representação usada (saturação, ou *clipping*). A transformação deve ser feita *in place*, ou seja, o próprio vetor de entrada é usado como saída.

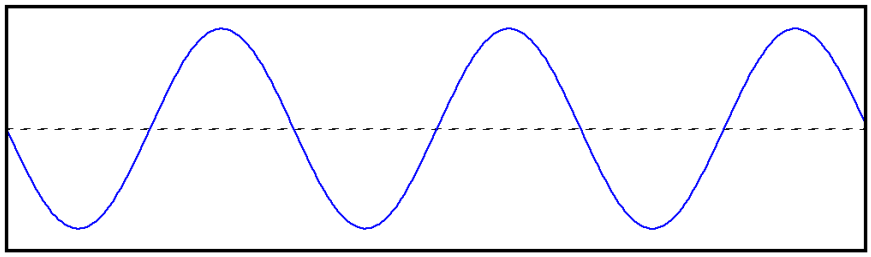
Exemplo:
Sinal original:



Resultado com `ganho` = 0.5:



Resultado com `ganho` = -1. Note que todas as amostras continuam com o mesmo valor absoluto, mas com o sinal invertido.

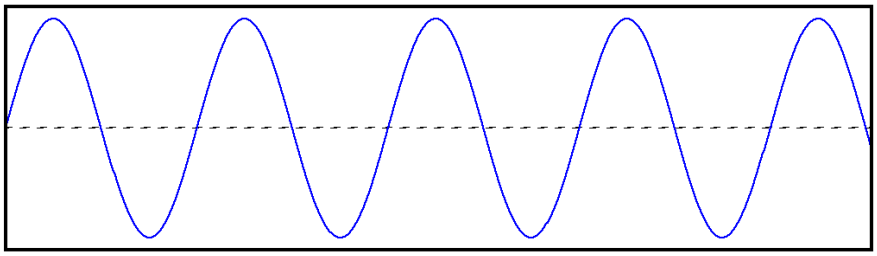


Função 2

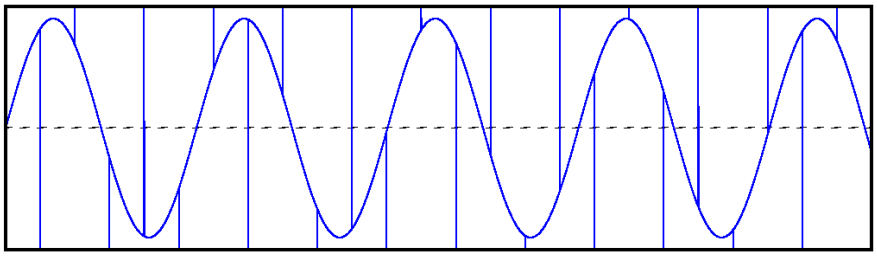
```
void ruidoPeriodico (double* dados, unsigned long n_amostras,
                     unsigned short intervalo);

Parâmetros: double* dados: vetor de dados.
            unsigned long n_amostras: número de amostras no vetor.
            unsigned short intervalo: o número de amostras entre dois
              pontos de ruído é intervalo-1.
```

Esta função adiciona "estalos" periódicos a um sinal. O ruído começa com o valor 1 na posição 0, e vai intercalando 1 e -1 em um intervalo dado. O número de amostras entre dois pontos de ruído é igual a `intervalo-1`. Ou seja, se `intervalo` tiver valor 10, a amostra na posição 0 é substituída por 1, a amostra na posição 10 é substituída por -1, a amostra na posição 20 é substituída por 1 e assim por diante. O efeito desta função será um som com "estalos", do tipo que eventualmente ocorre quando se usam equipamentos de baixa qualidade ou quando existem problemas elétricos.

Exemplo:
Sinal original:



Sinal com ruído periódico:



Função 3

```
void removeRuido (double* dados, unsigned long n_amostras);

Parâmetros: double* dados: vetor de dados.
            unsigned long n_amostras: número de amostras no vetor.
```

Esta função deve tomar um sinal com "estalos" (do tipo gerado pela função `ruidoPeriodico`) e produzir uma versão aproximada do sinal original. Para isso, deve ser usado um *filtro da mediana de largura 3*: cada amostra é substituída pela mediana dos valores em uma vizinhança que inclui a amostra, sua antecessora e sua sucessora. Por exemplo, o valor na posição 10 do vetor de dados é substituído pela mediana dos valores nas posições 9, 10 e 11. Como a primeira amostra não tem antecessora e a última amostra não tem sucessora, os seus valores permanecem inalterados. É importante lembrar que as medianas são feitas sobre os valores do sinal original, e não sobre o sinal modificado: ou seja, a amostra na posição 11 deve considerar o valor original da amostra na posição 10, e não a mediana dos valores nas posições 9, 10 e 11.

Explique em seu relatório por que o filtro da mediana tem como efeito a eliminação dos ruídos do tipo "estalo".

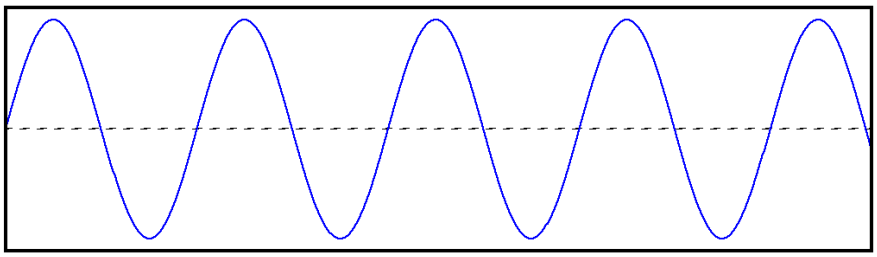
Função 4

```
void simulaSubAmostragem (double* dados, unsigned long n_amostras,
                          unsigned short n_constantes);

Parâmetros: double* dados: vetor de dados.
            unsigned long n_amostras: número de amostras no vetor.
            unsigned short n_constantes: largura de cada "degrau".
```

Esta função deve simular uma sub-amostragem do sinal. Para isso, ela deve substituir cada grupo de `n_constantes` amostras pelo valor da primeira amostra do grupo. Por exemplo, se a primeira amostra tiver valor 0.4 e `n_constantes` for 10, as 10 primeiras amostras terão todas o valor 0.4, as 10 amostras seguintes terão todas o valor da 11ª amostra, e assim por diante.

O efeito desta função é deixar o sinal com "degraus", com a qualidade degradada, gerando distorções e componentes que não existem no sinal original (chamados de *artefatos*). Quanto maior o valor de `n_constantes`, mais perceptível é a degradação. Alguns audiófilos com conhecimento limitado sobre processamento de sinais costumam associar este aspecto de "degraus" ao áudio digital em geral. Esta associação é equivocada, porque a saída de um conversor digial-analógico atenua esses "degraus" (na verdade, produzir degraus ideais é tecnicamente inviável!). O efeito de se ter um sinal com baixa taxa de amostragem não é o mesmo desta função, mas sim a percepção de um som "abafado".

Exemplo:
Sinal original:



Sinal com sub-amostragem simulada: