Universidade Tecnológica Federal do Paraná
Departamento Acadêmico de Informática
ICSF13 – Fundamentos de Programação 1
Prof. Bogdan Tomoyuki Nassu
Profa. Leyza Baldo Dorini
Prof. Daniel Fernando Pigatto

Trabalho 2

Esta é a especificação do segundo trabalho da disciplina ICSF13 - Fundamentos de Programação 1, profs. Bogdan T. Nassu, Leyza B. Dorini e Daniel F. Pigatto, para o período 2023/2.

I) Equipes

O trabalho deve ser feito em dupla. Trabalhos individuais só serão aceitos se devidamente justificados e previamente autorizados pelos professores. A justificativa deverá ser um motivo de força maior: questões pessoais como “prefiro trabalhar sozinho” ou “não conheço ninguém da turma” não serão aceitas. Durante as aulas, algumas duplas podem ser convocadas para explicar como está a divisão do trabalho (ver item IV).

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 (ver item IV). Não compartilhe códigos (fonte nem pseudocódigos)!!!

II) Entrega

O prazo de entrega é 30/10/2023. Trabalhos entregues após esta data terão sua nota final reduzida em 0,00025% para cada segundo de atraso. Um (e somente um!) dos membros da equipe deve entregar, através da página da disciplina no Classroom, dois arquivos (separados, sem compressão):

- Um arquivo chamado *t2-x-y.c* (*t2-x.c*, caso o trabalho seja feito individualmente), onde *x* e *y* são os números de matrícula dos alunos. O arquivo deve conter as implementações das funções pedidas (com cabeçalhos idênticos aos especificados). 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.

IMPORTANTE: as funções pedidas não envolvem interação com usuários. Elas não devem imprimir mensagens (por exemplo, através da função `printf`), nem bloquear a execução enquanto esperam entradas (por exemplo, através da função `scanf`). O arquivo também não deve ter uma função `main`.

- Um arquivo no formato PDF chamado *t2-x-y.pdf* (*t2-x.pdf*, caso o trabalho seja feito individualmente), onde *x* e *y* são os números de matrícula dos alunos. Este arquivo deve conter um relatório breve (em torno de 1 a 2 páginas), descrevendo (a) a contribuição de cada membro da equipe, (b) os desafios encontrados, e (c) a forma como eles foram superados. Não é preciso seguir uma formatação específica. Os autores do arquivo devem estar identificados no início.

III) Avaliação (normal)

Todos os testes serão feitos usando a IDE Code::Blocks. Certifique-se de que o seu trabalho pode ser compilado e executado a partir dela.

Os trabalhos serão avaliados por meio de baterias de testes automatizados. A referência será um conjunto de funções implementadas pelos professores, sem “truques” ou otimizações sofisticadas. Os resultados dos trabalhos serão comparados àqueles obtidos pela referência. Cada função será avaliada individualmente. Os seguintes pontos serão avaliados:

III.a) Compilação. Cada erro que impeça a compilação do arquivo implica em uma redução de 50% no peso de uma função caso o erro esteja no seu corpo, ou uma penalidade de 10 pontos em outros casos. Certifique-se que seu código compila!!!

III.b) Corretude. As funções devem produzir o resultado correto para todas as entradas testadas. Uma função que produza resultados incorretos terá sua nota reduzida. Quando possível, o professor corrigirá o código até que ele produza o resultado correto. A cada erro corrigido, a nota da função será multiplicada por um valor: 0.5 se o erro levar a resultados com diferenças óbvias para os exemplos, 0.7 se o erro implicar em erros como repetições infinitas ou acessos inválidos à memória, ou 0.8 em outros casos.

III.c) Atendimento da especificação. Serão descontados até 10 pontos para cada item que não esteja de acordo com esta especificação, como nomes de arquivos e funções fora do padrão.

III.d) Documentação. Comente a sua solução para cada função. Não é preciso detalhar tudo linha por linha, mas forneça uma descrição geral da sua abordagem, assim como comentários sobre estratégias que não fiquem claras na leitura das linhas individuais do programa. Uma função sem comentários terá sua nota reduzida em até 25%.

III.e) Organização e legibilidade. Use a indentação para tornar seu código legível, e mantenha a estrutura do programa clara. Evite construções como *loops* infinitos terminados somente por `break`, ou *loops* `for` com várias inicializações e incrementos, mas sem corpo. Evite também replicar blocos de programa que poderiam ser melhor descritos por repetições. Um trabalho desorganizado ou cuja legibilidade esteja comprometida terá sua nota reduzida em até 30%.

III.f) Variáveis com nomes significativos. Use nomes significativos para as variáveis – lembre-se que `n` ou `x` podem ser nomes aceitáveis para um parâmetro de entrada que é um número, mas uma variável representando uma “soma total” será muito melhor identificada como `soma_total`, `soma` ou `total`; e não como `t`, `aux2` ou `foo`. Uma função cujas variáveis internas não tenham nomes significativos terá sua nota reduzida em até 20%.

III.g) Desempenho. Se a estrutura lógica de uma função for pouco eficiente, por exemplo, realizando muitas operações desnecessárias ou redundantes, a sua nota pode ser reduzida em até 20%.

IV) Avaliação (especial)

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 a qualidade das soluções apresentadas em atividades anteriores, ou se a explicação sobre o funcionamento de uma função for pouco clara.

V) Nota

A nota do trabalho será igual à soma das notas de cada função, mais a nota do relatório. A descrição de cada função indica a sua nota máxima. A nota máxima para o relatório é de 5 pontos.

VI) Apoio

Os professores estarão disponíveis para tirar dúvidas a respeito do trabalho, nos horários de atendimento previstos (e em casos excepcionais, fora deles). A comunicação por e-mail pode ser usada para pequenas dúvidas sobre aspectos pontuais.

Instruções para "montar" o projeto:

Será disponibilizado um "pacote" contendo os seguintes arquivos:

1) 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. Note que as funções do trabalho NÃO PRECISAM acessar o módulo wavfile!

3) Exemplos de programas para teste e arquivos de áudio demonstrando os resultados produzidos.

No Code::Blocks, para compilar e usar no seu programa as funções presentes no arquivo wavfile.c, crie 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.

Sobre as funções do módulo wavfile:

Observação: você <u>não precisa</u> compreender o funcionamento do módulo wavfile, mas conseguir utilizá-lo pode ajudar a realizar testes mais avançados. Em particular, a função `writeSamplesAsText` pode ser útil para depurar o seu programa.

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.

```
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.
```

```
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.
```

```
WavHeader generateSignal (int* 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:
int* 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.
```

```
void generateRandomData (double* data, int n_samples);
```

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

```
Parâmetros:
double* data: vetor a preencher.
int n_samples: número de amostras no vetor.

Valor de Retorno: NENHUM
```

```
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.

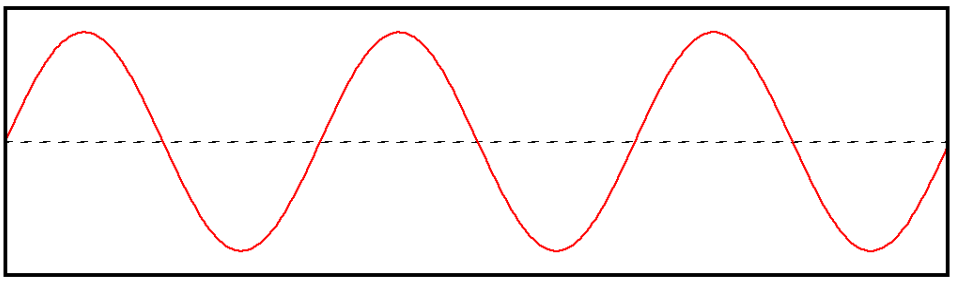
```
Parâmetros:
char* filename: nome do arquivo a ser escrito.
double* data: vetor de dados contendo as amostras.
unsigned long n_samples: número de amostras no vetor.

Valor de Retorno: 1 se não ocorreram erros, 0 do contrário.
```

Função 1 (5 pontos)

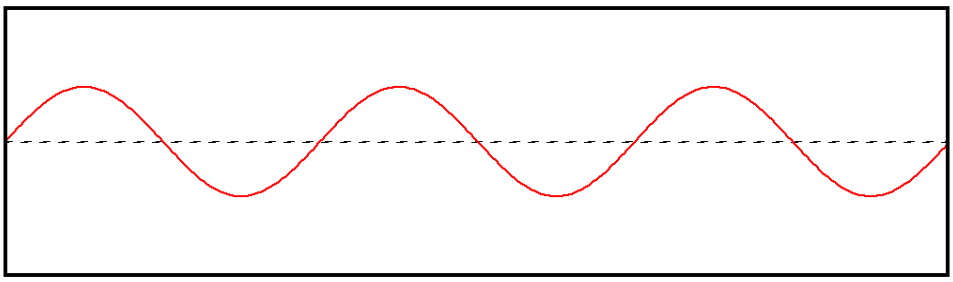
```
void mudaGanho (double* dados, int n_amostras, double ganho);

Parâmetros: double* dados: vetor de dados.
            int 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.

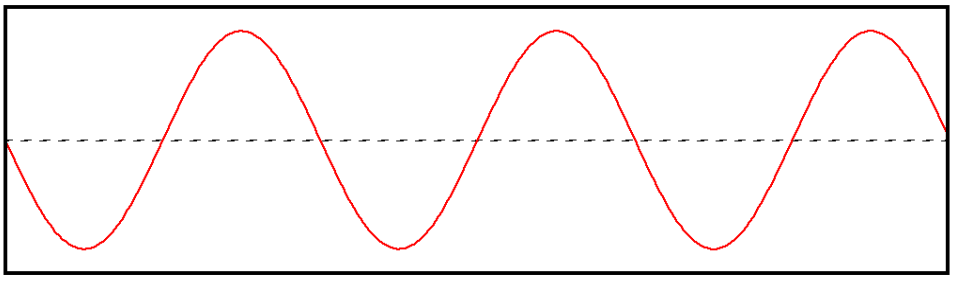
Exemplo:
Sinal original:



Resultado com `ganho` = 0.5:



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



Função 2 (10 pontos)

```
int contaSaturacoes (double* dados, int n_amostras);

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

Um sinal satura (“clipa”) quando o valor de uma amostra extrapola o limite permitido pela representação usada. Neste trabalho, consideramos amostras com valor entre -1 e 1. A função `contaSaturacoes` deve retornar o número de amostras que saturam, ou seja, que têm valor absoluto maior que 1.

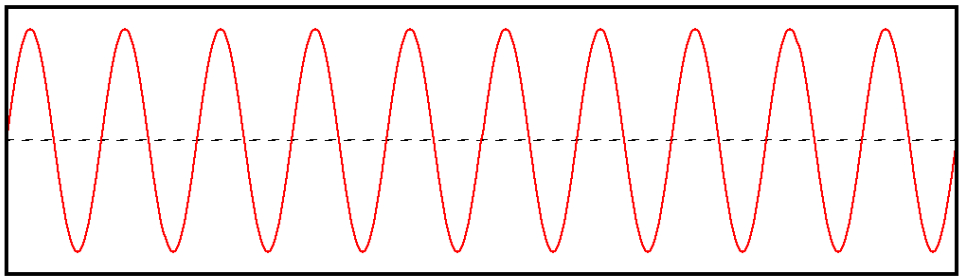
Função 3 (10 pontos)

```
int hardClipping (double* dados, int n_amostras, double limite);

Parâmetros: double* dados: vetor de dados.
            int n_amostras: número de amostras no vetor.
            double limite: limite para a saturação.
```

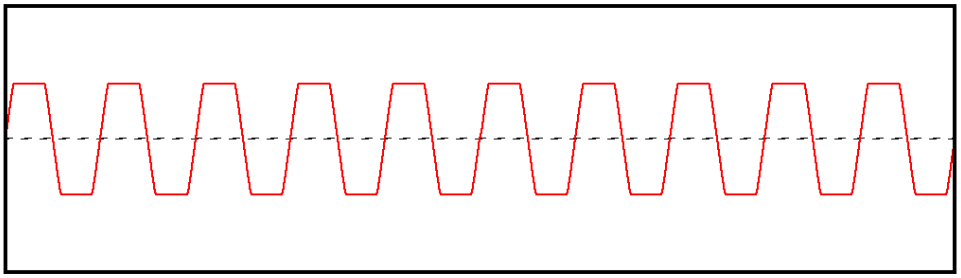
Um componente eletrônico pode “saturar” quando recebe um sinal com intensidade superior à sua faixa normal de operação. Apesar de normalmente ser um efeito indesejado, na música a saturação é usada com fins artísticos – por exemplo, os timbres distorcidos de guitarras são originalmente fruto da saturação das válvulas de amplificadores usados em volumes muito altos. Posteriormente, foram desenvolvidos circuitos que permitem obter efeitos similares em volumes mais baixos, ao saturar válvulas ou diodos em estágios anteriores à amplificação.

Quando um sinal atinge o ponto de saturação, os picos e vales das ondas sofrem alterações diversas. O objetivo desta função é simular a saturação, de uma forma bastante simplificada: todas as amostras do sinal que tiverem valor absoluto maior que um limite dado devem ter sua magnitude modificada para ficar igual ao limite. Isto resulta em sinais com picos e vales “retos”, fenômeno que em inglês é chamado de *hard clipping*. Este é um tipo de saturação agressiva – de fato, a saturação como propomos é tão agressiva que não poderia ser obtida por componentes físicos! A transformação deve ser feita *in place*, ou seja, o próprio vetor de entrada é usado como saída. Ao final, a função deve retornar o número de amostras que foram alteradas.

Exemplo:
Um sinal com 1000 amostras:



Hard clipping com `limite` = 0.5 (660 amostras modificadas). Repare que as amostras com sinal negativo continuam na região inferior do gráfico.



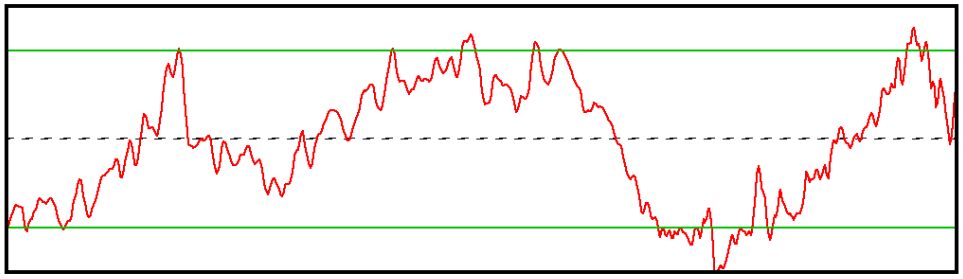
Função 4 (40 pontos)

```
void limitaSinal (double* dados, int n_amostras, int n_passos);

Parâmetros: double* dados: vetor de dados.
            int n_amostras: número de amostras no vetor.
            int n_passos: número de vizinhos atenuados à esquerda e à direita.
```

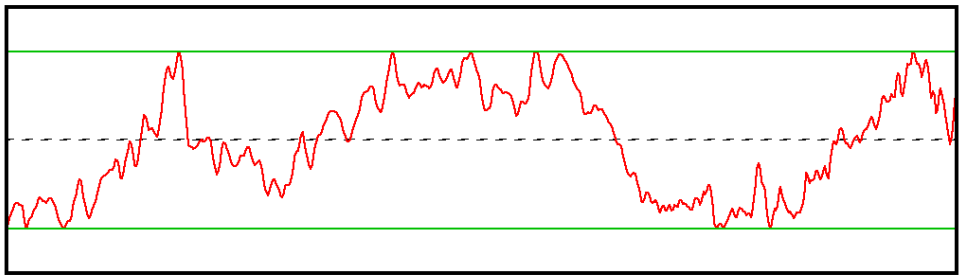
O *alcance dinâmico* é o intervalo de valores entre os pontos de intensidade mínima e máxima de um sinal. No áudio digital com 16 bits por amostra, por exemplo, as amostras costumam estar no intervalo [-32767, +32767], ou seja, existem 65535 valores possíveis para uma amostra. Neste trabalho, o alcance é normalizado para valores reais no intervalo [-1,+1], mas as funções para salvamento de arquivos convertem os valores novamente para a faixa permitida para 16 bits por amostra.

Desde a década de 1970, alguns profissionais da indústria fonográfica notaram que gravações com volume mais alto tendem a se sobressair na percepção dos ouvintes de rádio. Eles então buscaram meios de aumentar o volume percebido de suas produções, com o objetivo de torná-las mais "chamativas" que as concorrentes. O mesmo efeito também passou a ser explorado em comerciais de TV e em plataformas de *streaming* como YouTube. A ferramenta básica para esta tarefa é o que se chama de *limiter*. Um *limiter* limita o alcance dinâmico de um sinal de áudio, "achatando" trechos que extrapolem um limiar dado. Isso implica em uma redução do volume percebido.

Exemplo:
Sinal original:

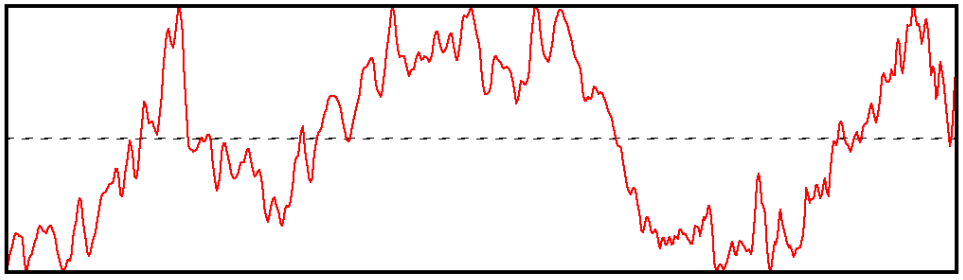


Sinal após uso de um *limiter*:



Mas se o objetivo é aumentar a percepção do volume, por que reduziríamos o alcance dinâmico? O "truque" é, após o uso do *limiter*, aumentar novamente o volume do sinal inteiro, aproveitando todo o alcance dinâmico permitido. Com isso, trechos que não tiverem sido atenuados terão seu volume aumentado. Desta forma, a integral do sinal (visto como uma função de intensidade x tempo) é aumentada, sem que nenhuma amostra "sature", ou seja, ultrapasse o limite permitido pela representação.

Exemplo:
O mesmo sinal do exemplo anterior, após aumento de volume.



A partir da década de 1980, a produção musical em computadores tornou o uso de *limiters* mais acessível, o que levou, a partir da década de 1990, ao uso de *limiters* cada vez mais radicais. Esta busca das gravadoras por níveis de volume cada vez maiores recebeu o apelido de *loudness war* (uma busca na Internet pode revelar muito mais detalhes desta história). Atualmente, quase toda gravação comercial passa por um *limiter*.

O uso exagerado de *limiters* é bastante criticado por certos profissionais. O problema é que, ao "achatar" um sinal, há perda de informações. Ou seja, um sinal muito "achatado" possui menos nuances e sutilezas. Além disso, o uso de um *limiter* reduz as diferenças entre trechos da música, deixando tudo em volumes próximos do máximo. O resultado é que algumas gravações passaram a apresentar distorções perceptíveis, além de causarem fadiga auditiva. Este efeito nocivo é percebido principalmente em gravações de estilos mais populares. Por exemplo, o disco Death Magnetic da banda Metallica, lançado em 2008, sofreu tantas críticas que muitos fãs da banda passaram a compartilhar na Internet uma versão do disco com menos "achatamentos", obtida a partir da trilha sonora do *game* Guitar Hero III!

A função `limitaSinal` deve implementar um *limiter* simples, que mantém todas as amostras dentro do intervalo [-1,1]. Para isso, ela deve percorrer as posições do sinal em ordem crescente. Cada vez que for encontrada uma amostra com valor absoluto maior que 1, a função deve atenuar esta amostra e as `n_passos` amostras à direita e à esquerda, com ganhos progressivamente maiores.

Por exemplo, se `n_passos` for 4 e a amostra na posição 10 tiver valor 2, a amostra na posição 10 terá seu ganho reduzido pela metade (ganho 0.5); as amostras nas posições 9 e 11 terão o ganho ajustado em 0.6; as amostras nas posições 8 e 12 terão o ganho ajustado em 0.7, as amostras nas posições 7 e 13 terão o ganho ajustado em 0.8, e as amostras nas posições 6 e 14 (10-4 e 10+4, respectivamente) terão o ganho ajustado em 0.9. Neste exemplo, a diferença de ganho para cada passo é 0.1, na implementação da função, a diferença deve ser calculada com base na intensidade de cada amostra que estiver fora do limite permitido e no valor de `n_passos`. Após atenuar o sinal a partir da amostra em uma posição *i*, a função deve continuar buscando amostras fora do limite a partir da posição *i+1*.

Note que (por simplicidade) esta função difere dos *limiters* clássicos, pois o limiar é fixado em 1, que é o limite da representação usada. Desta forma, qualquer aumento de volume deve ser realizado *antes* da chamada da função – observe como o programa de exemplo trabalha com a função.

Função 5 (30 pontos):

```
void geraOndaQuadrada (double* dados, int n_amostras, int taxa, double freq);

Parâmetros: double* dados: vetor a preencher.
            int n_amostras: número de amostras no vetor.
            int taxa: taxa de amostragem.
            double freq: frequência da onda.
```

Esta função deve preencher o vetor dado com uma onda quadrada. Uma onda quadrada é um sinal no qual todas as amostras têm valor 1 ou -1. Cada ciclo tem uma série de amostras com valor 1, seguida do mesmo número de amostras com valor -1. A frequência, dada em Hz, indica quantos ciclos completos devem ocorrer em 1 segundo.

A partir da frequência e da taxa de amostragem, é possível obter o período do sinal, dado como um número de amostras. Para controlar a duração dos ciclos, use como base o período de meio ciclo (o meio-período). O meio-período pode ser um número não-inteiro de amostras. Nestes casos, arredondam-se os valores usados para baixo, mas o erro acumulado é mantido e adicionado ao período do próximo meio ciclo. Na prática, isso significa que:

a) Um ciclo pode ter *N* amostras com valor 1, mas *N+1* ou *N*-1 amostras com valor -1 (ou vice-versa); e
b) Pode existir uma diferença no tamanho dos ciclos de um mesmo sinal.

Por exemplo, suponha um sinal com a frequência de 10.7Hz. Se tivermos uma taxa de amostragem de 44.1KHz (padrão de um CD), o meio-período do sinal é de aproximadamente 2060.74766 amostras. O primeiro ciclo terá 2060 amostras com valor 1, o que resulta em um erro de 0.74766 amostras. Desta forma, o primeiro ciclo teria aproximadamente 2061.49532 amostras com valor -1 - valor que é arredondado para 2061, com o erro de 0.49532 sendo propagado para o próximo ciclo. O segundo ciclo terá 2061 amostras com valor 1, seguidas de 2060 amostras com valor -1. O terceiro ciclo terá 2061 amostras com valor 1, seguidas de 2061 amostras com valor -1.

Exemplo:
Uma onda quadrada com 1000 amostras e frequência de 441 Hz, a uma taxa de amostragem de 44.1KHz. Cada ciclo tem 100 amostras, portanto temos 10 ciclos neste sinal.