

LAB-III

MESA DE EFEITOS



Equipe

Heverton Reis

Ivon Luiz

John Brian

Matheus Souto

Vitor Cavalcante



Objetivos

Aplicar os conceitos de Processamento Digital de Sinais (PDS) em um ambiente prático.

Utilização de um DSP para implementação de efeitos de áudio em tempo real.

Integração dos conhecimentos teóricos da disciplina com sistemas embarcados.

Consolidação da compreensão sobre técnicas de manipulação de sinais digitais.

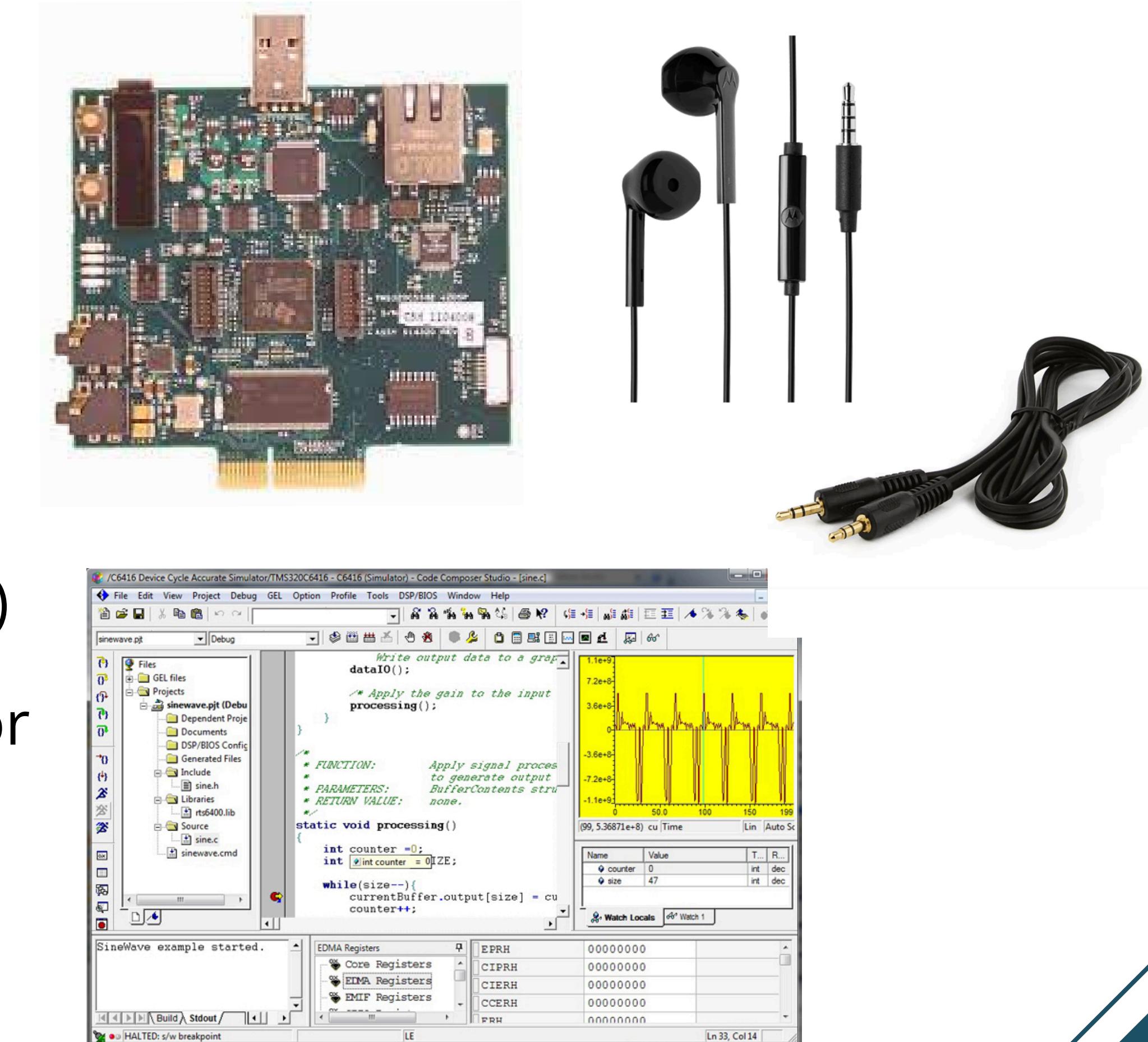
Exploração de aplicações práticas do PDS em projetos reais.

Recursos

- Kit TMS320C5502 eZdsp
- Code Composer Studio (CCS) rodando em um Computador

Pessoal

- Cabo auxiliar P2-P2
- Fone de ouvido estéreo



Processo de implementação

1 - Estudo de como a placa funciona

2 - Implementação em Python dos efeitos em arquivos .wav

3- Estudo da teoria dos efeitos

4 - Implementação e tradução dos códigos de C++ para C

5 - Testes na placa em arquivos .wav

6 - Implementação em tempo real

Efeitos em Python

- »» Efeitos implementados em Python com a lib “SOX”
- »» Classe “AudioEffects”
- »» Cada método configura o Transformer de uma forma

```
class AudioEffects:  
    Qodo Gen: Options | Test this method  
    def __init__(self, audio_path):  
        self.audio_path = audio_path  
        self.audio_name = audio_path[:-4]  
  
    Qodo Gen: Options | Test this method  
    def add_effects(  
        self,  
        reverb_specs=None,  
        delay_specs=None,  
        tremolo_specs=None,  
        flanger_specs=None,  
        pitch_shift=None,  
    ):  
  
        self.tfm = sox.Transformer()  
        effects = []  
  
        if reverb_specs != None:  
            self.add_reverb(reverb_specs)  
        effects.append("reverb")
```

Desenvolvimento dos efeitos

>>> Exemplo de uso:

```
audio = AudioEffects("musica.wav")
audio.add_effects(reverb_specs=[50, 20, 80, 100], pitch_shift=300)
```

Reverb

A reverberação é definida como o efeito combinado de múltiplas reflexões sonoras dentro de uma sala. Esse fenômeno é amplamente utilizado na produção musical, e, ao longo dos anos, muitas técnicas foram desenvolvidas para simular o efeito de reverberação. Um exemplo antigo de técnica para criar som reverberado é o uso de uma câmara de eco. Com o avanço da tecnologia e do processamento digital de sinais, surgiram métodos para emular digitalmente a reverberação, resultando em reverberadores artificiais complexos que produzem um som natural.

O estudo da reverberação artificial foi iniciado na década de 1960 por Manfred Schroeder e Ben Logan. Eles propuseram um reverberador algorítmico, conhecido como Schroeder Reverberator, que resolveu dois problemas principais dos reverberadores da época: a baixa densidade de ecos (número de ecos por segundo) e a resposta de frequência não plana. Para resolver essas questões, Schroeder propôs uma estrutura com quatro filtros comb em paralelo conectados a dois filtros all-pass em série.

Reverb: algoritmo de schroeder

Natural Sounding Artificial Reverberation^{*}

M. R. SCHROEDER

Bell Telephone Laboratories, Incorporated, Murray Hill, New Jersey

Artificial reverberation is added to sound signals requiring additional reverberation for optimum listening enjoyment. This paper describes methods for generating, by purely electronic means, an artificial reverberation which is indistinguishable from the natural reverberation of real rooms. This artificial reverberation can be given any desired characteristics to match different types of music and personal tastes. A method for making the artificial reverberation "ambiophonic" (*i.e.*, three-dimensional) is also described.

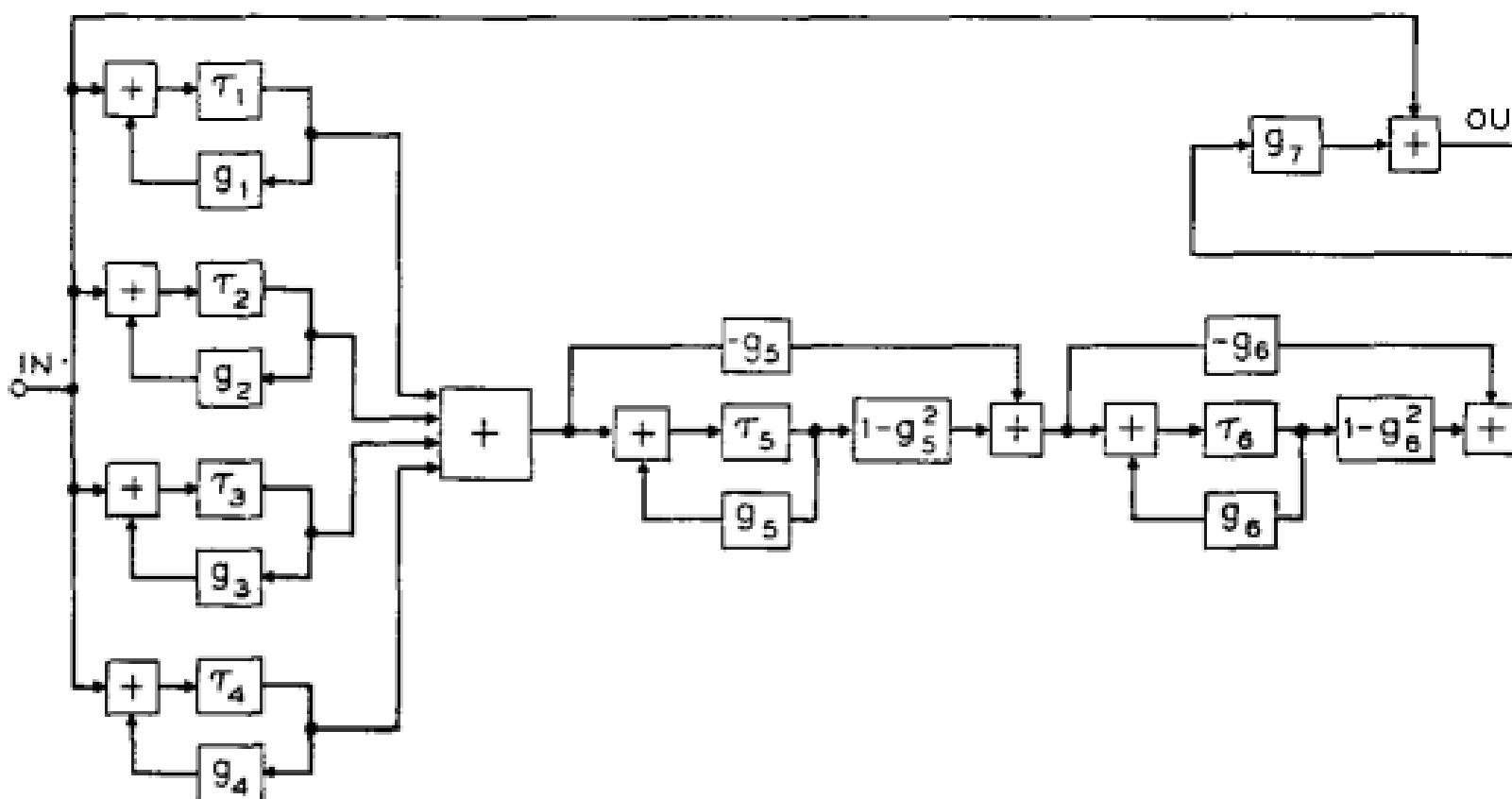


FIG. 6. Block diagram of reverberator using both comb filters (in parallel) and all-pass filters (in series). This type of reverberator allows the widest choice of specifications such as mixing ratios, delay of reverberated sound and kind of decay.

Reverb: schroeder

O filtro comb é o arranjo mais simples de produção de eco com uma linha de atraso e feedback. O ganho, $g < 0$, para que o sistema seja estável.

A resposta ao impulso deste filtro é uma sequência de decaimento exponencial. Isto é semelhante à forma como o som é refletido e decai em uma sala reverberante. Portanto, esta unidade pode ser usada para emular o efeito de um reverb. Mas, como discutido anteriormente, haveria um problema de baixa densidade de eco que causa vibração.

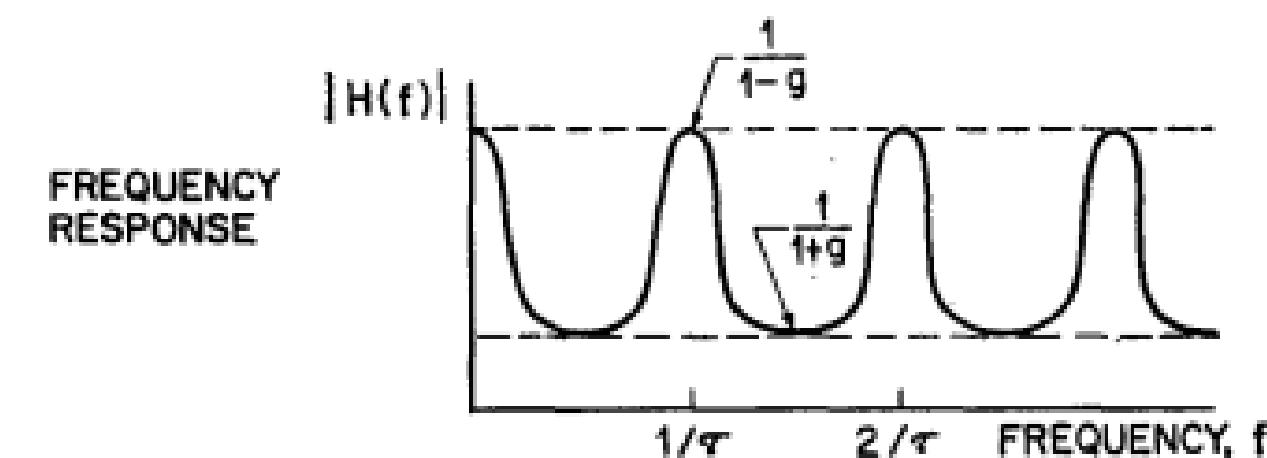
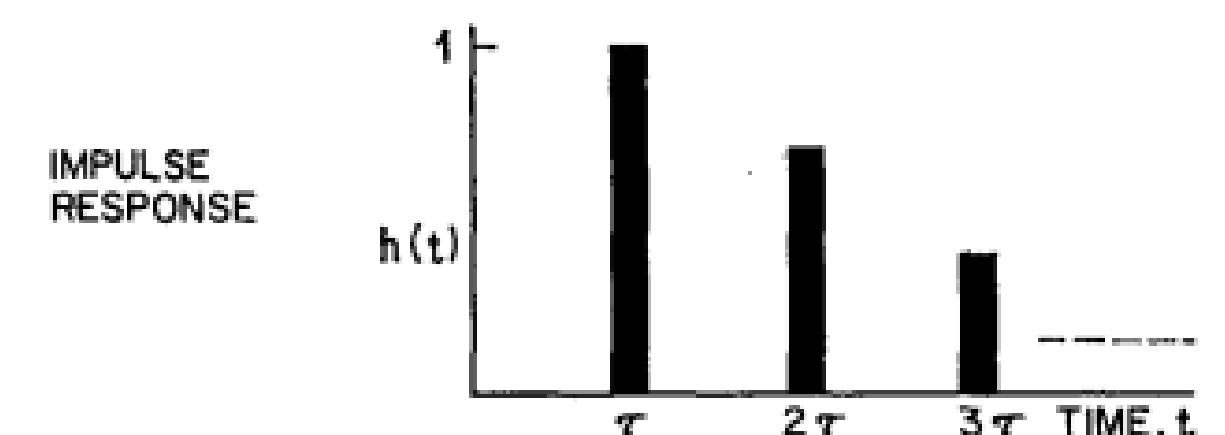
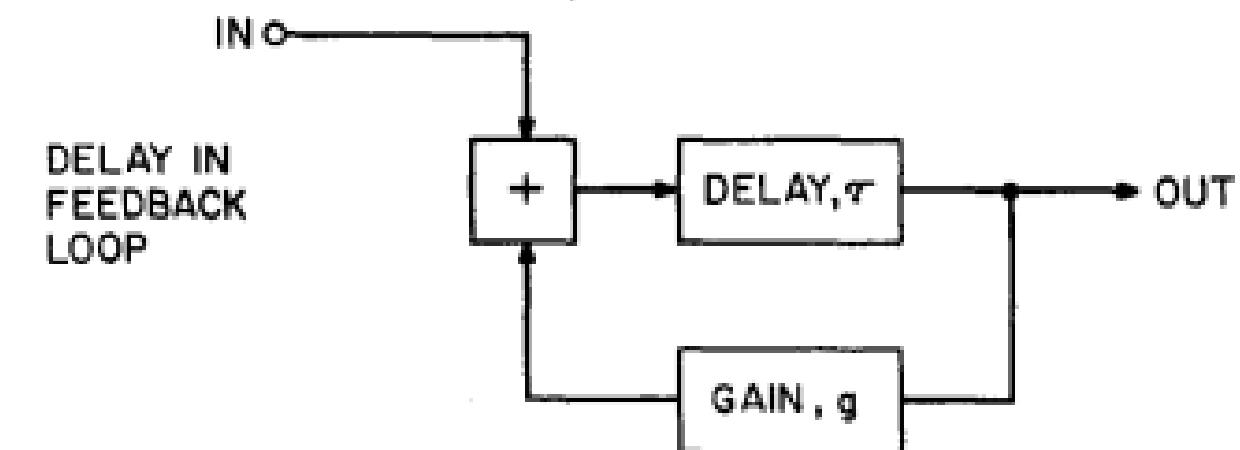


FIG. 1. Simple reverberator with exponentially decaying echo response. Frequency response resembles comb.

Reverb: schroeder

No que diz respeito ao problema da baixa densidade de eco, descobriu-se que são necessários aproximadamente 1000 ecos por segundo para uma reverberação sem vibração.

Assim, para atingir 1000 ecos por segundo, poderíamos colocar unidades de filtro pente em paralelo ou em série. Mas há um problema com qualquer uma dessas abordagens. Para um arranjo paralelo: Considere que temos um filtro comb com uma linha de atraso de 40 ms. Isso produziria 25 ecos por segundo. Então, para obter 1000 ecos por segundo precisaríamos de 40 unidades em paralelo! Isto é impraticável.

Para um arranjo em série: A resposta de frequência de um filtro comb não é plana. Como pode ser visto abaixo, existem picos e vales na resposta de frequência que dão a qualidade "colorida" indesejada ao som reverberado por este filtro. Para atingir a densidade de eco desejada, precisaríamos de 4 a 5 unidades em série, o que produziria uma qualidade de som totalmente inaceitável.

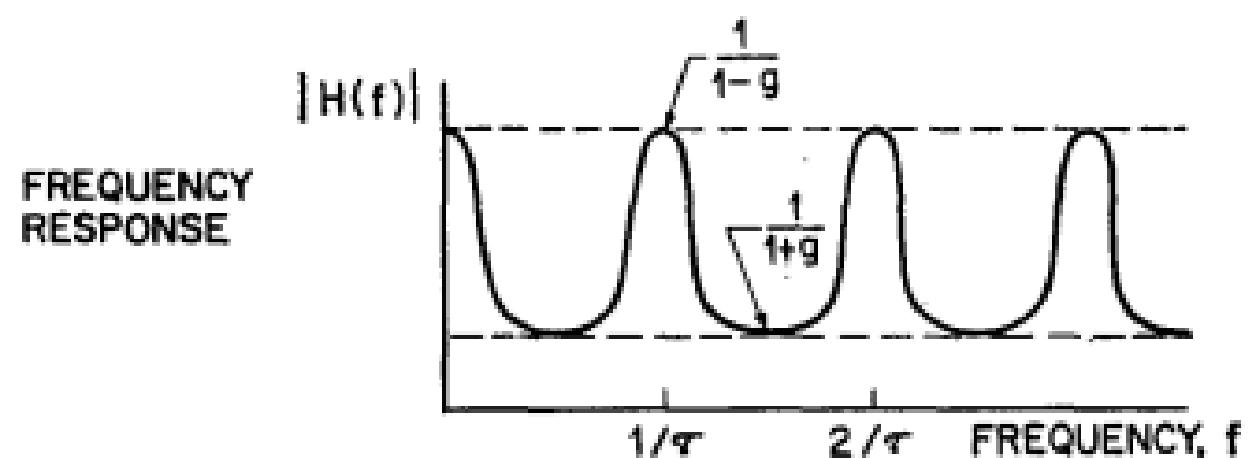
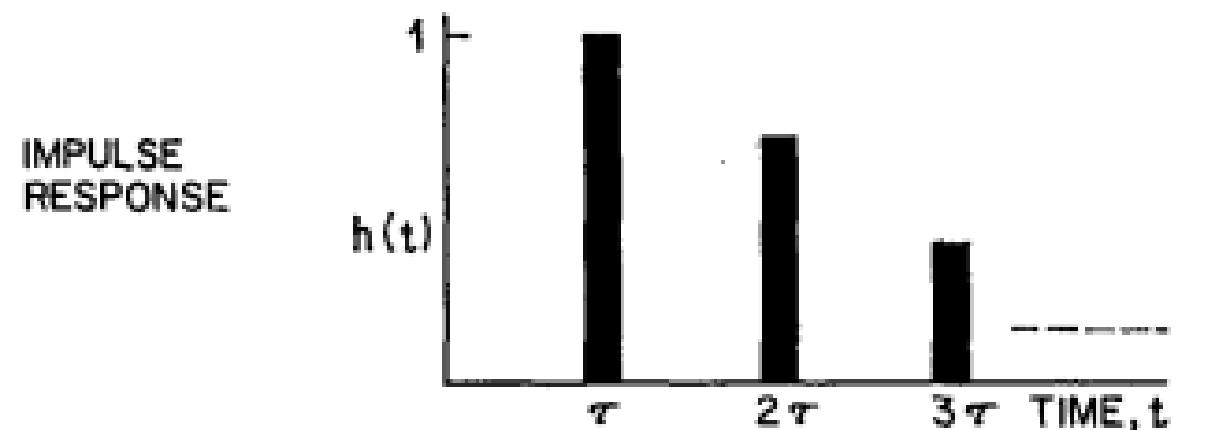
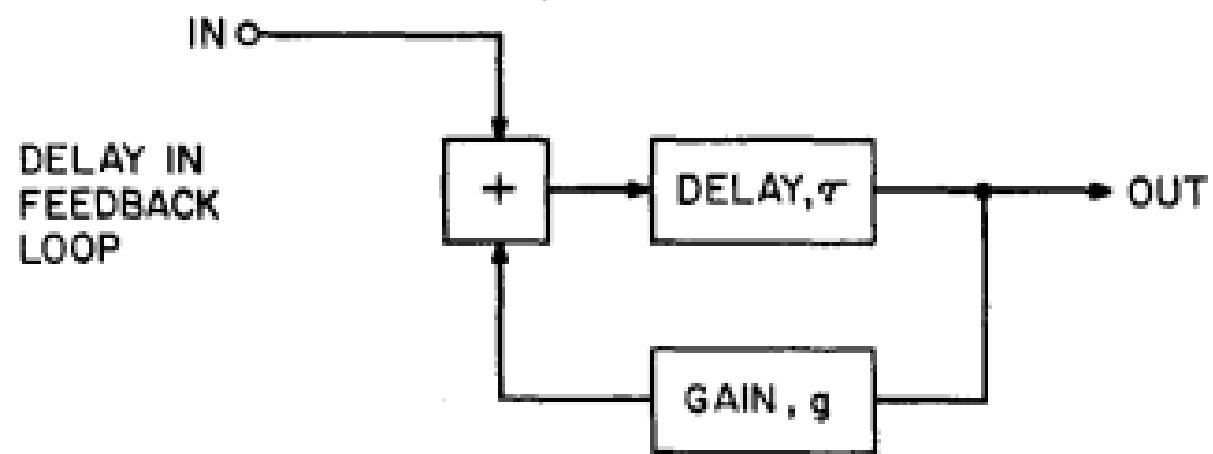


FIG. 1. Simple reverberator with exponentially decaying echo response. Frequency response resembles comb.

Reverb: schroeder

Schroeder observou que um filtro comb modificado, com um componente feed-forward negativo como mostrado, resultaria em uma resposta igual do reverberador para todas as frequências.

A resposta de frequência de um filtro passa-tudo é plana, permitindo assim uma alta densidade de eco sem coloração espectral.

Logan e Schroeder descobriram que as proporções de mixagem para obter uma resposta de frequência plana são $(-g)$ para o som sem atraso e $(1 - g^2)$ para o som com atraso, como pode ser visto na figura.

Conseqüentemente, filtros passa-tudo conectados em série podem ser uma das soluções plausíveis para obter reverberação.

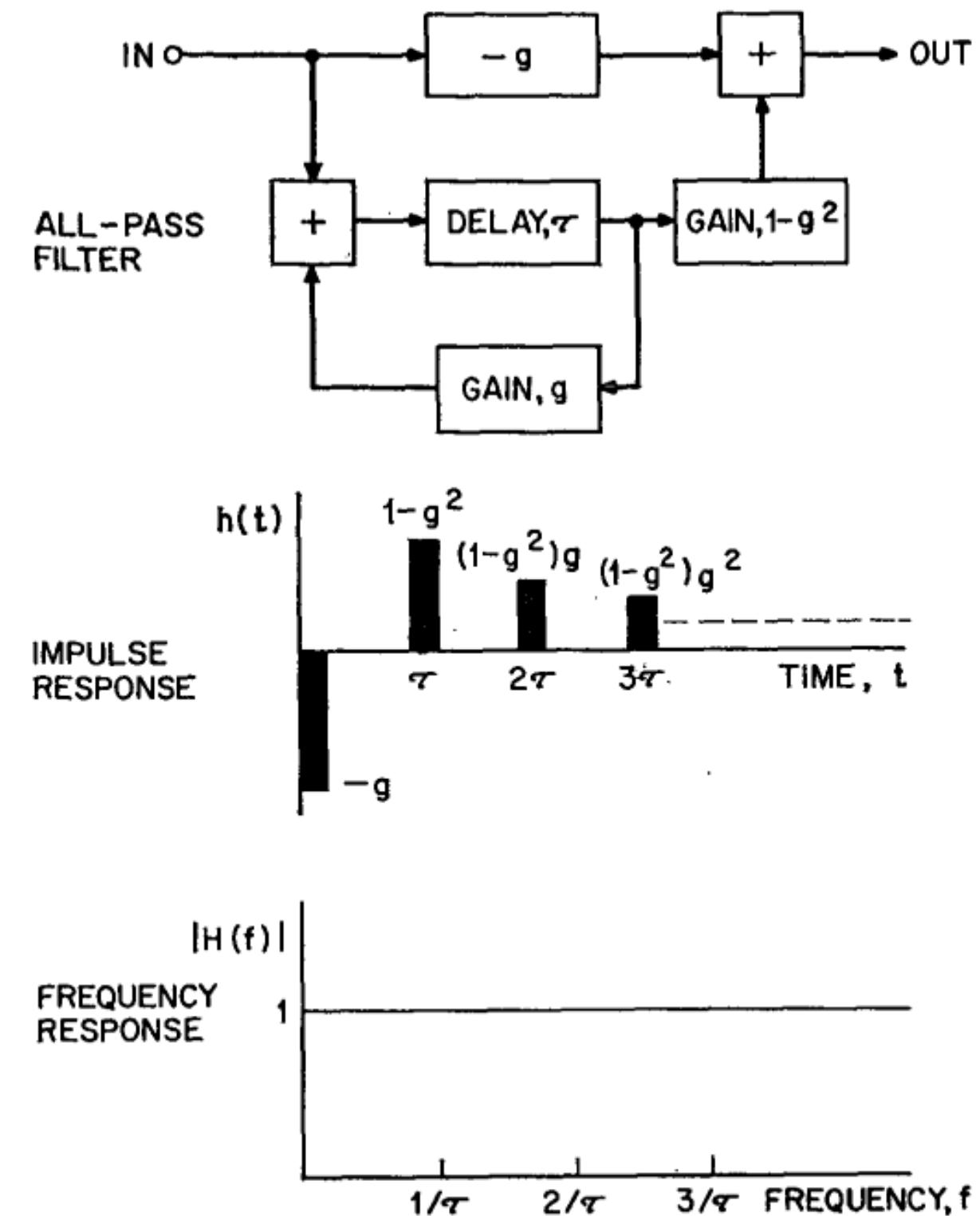


FIG. 2. Modification of simple reverberator. By adding proper amount of undelayed signal, frequency response of the reverberator becomes flat (all-pass reverberator).

Reverb: schroeder

Estrutura de 4 filtros comb e 2 filtros passa-tudo:

Uma sala reverberante tem múltiplos reflexos de som entre suas múltiplas paredes. Um filtro pente de feedback pode simular um par de paredes paralelas, então pode-se escolher o comprimento da linha de atraso em cada filtro pente como o número de amostras necessárias para uma onda plana se propagar de uma parede para a parede oposta e vice-versa.

Schroeder, em seu artigo, apresenta uma abordagem mais motivada psicoacusticamente para explicar o banco de filtros paralelo. A explicação detalhada pode ser encontrada em seu artigo. Em essência, diz que pode-se escolher os comprimentos da linha de atraso do filtro comb mais ou menos arbitrariamente e, em seguida, usar um número suficiente deles em paralelo (com comprimentos de linha de atraso mutuamente primos) para alcançar uma densidade de flutuação perceptualmente adequada na frequência. - magnitude da resposta.

Os filtros passa-tudo em série são adicionados para aumentar a densidade do eco e ter difusão suficiente para evitar vibração sem coloração do som.

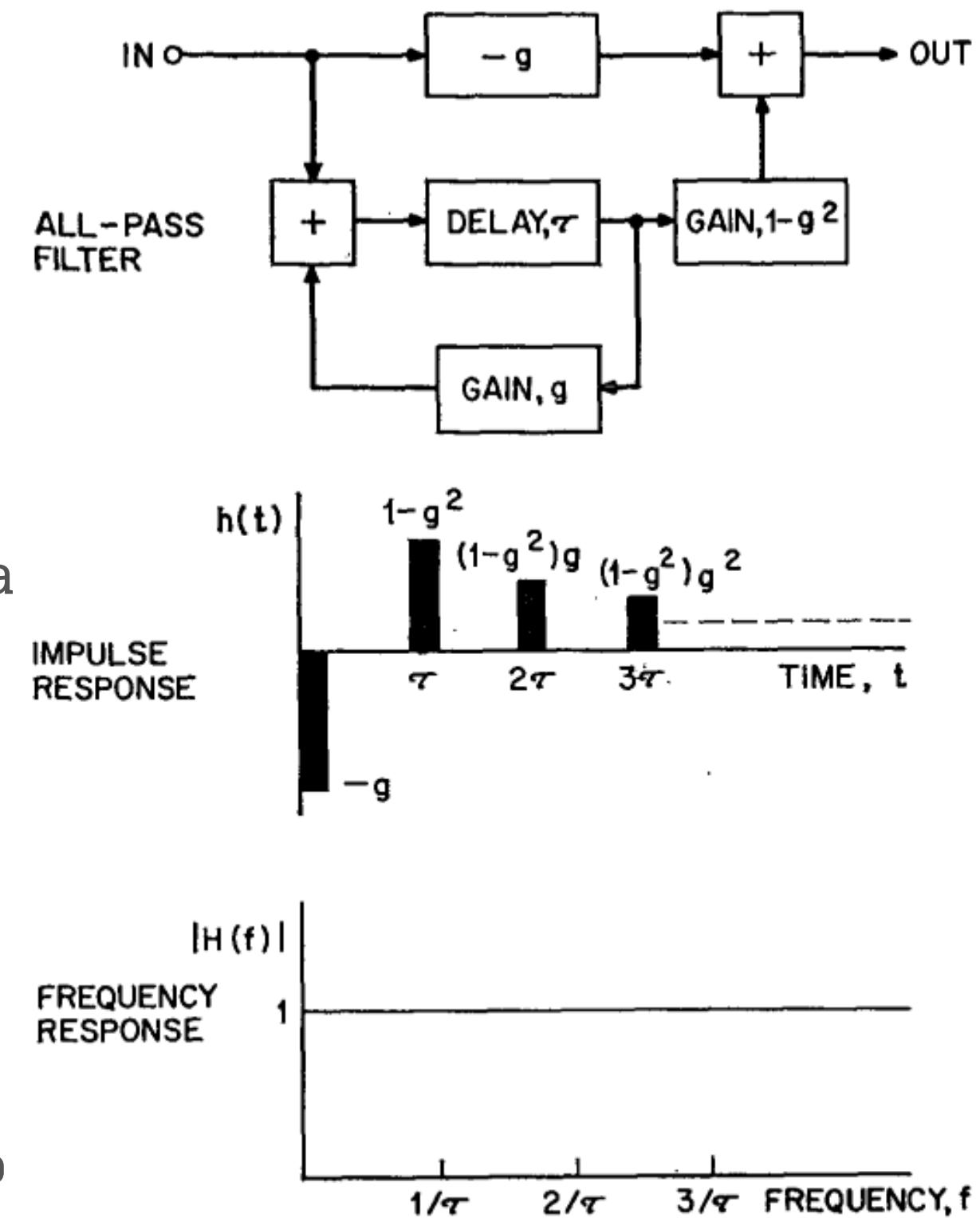


FIG. 2. Modification of simple reverberator. By adding proper amount of undelayed signal, frequency response of the reverberator becomes flat (all-pass reverberator).

Reverb: schroeder

O Tempo de Reverberação (T) é o tempo que o som leva para decair 60 dB.
Para um loop de feedback com ganho g e atraso τ , o tempo de reverberação pode ser calculado usando a fórmula:

$$T = \frac{-60 \cdot \tau}{20 \cdot \log |g|}$$

Essa fórmula permite ajustar o ganho e o atraso para obter o tempo de reverberação desejado.

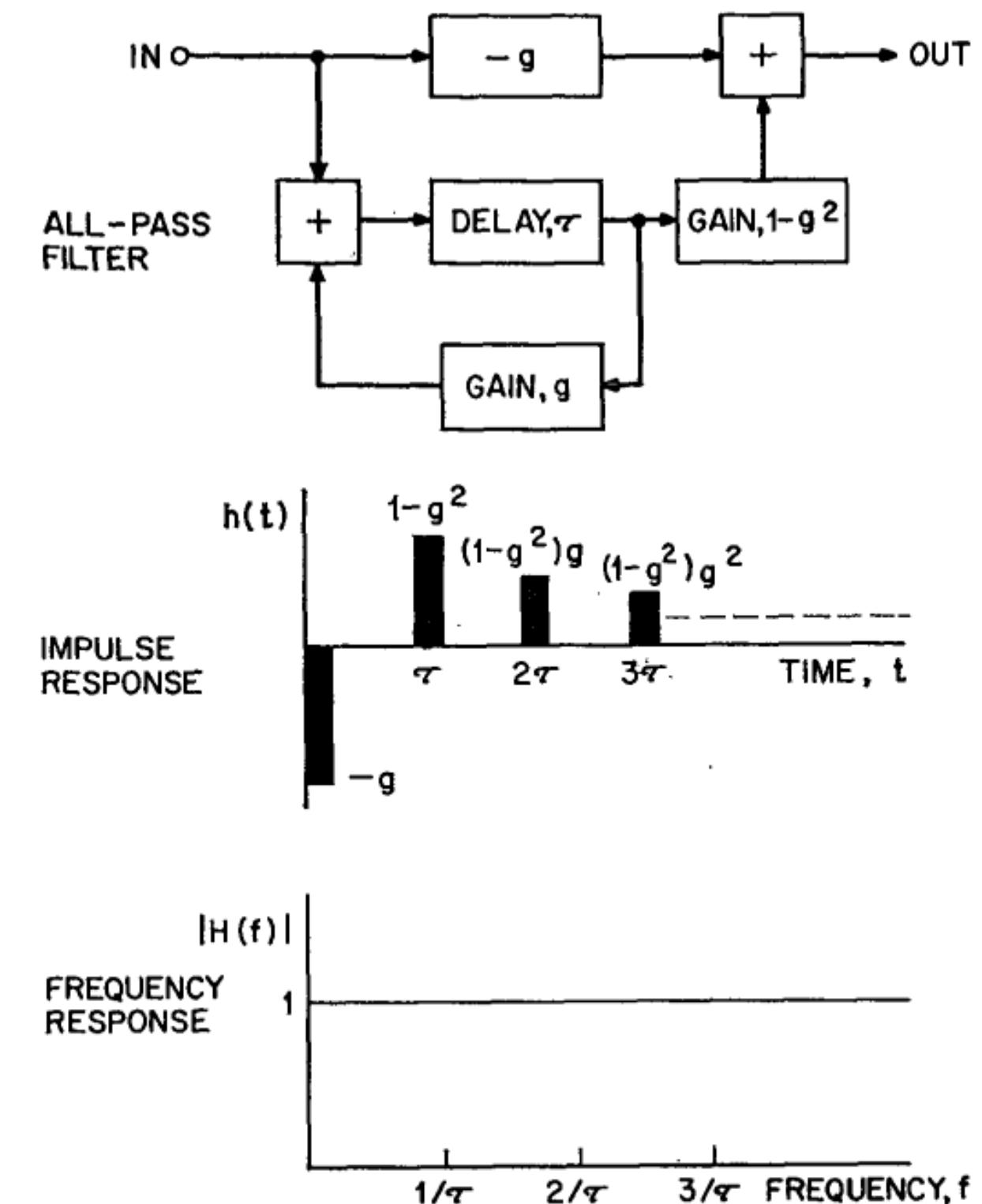
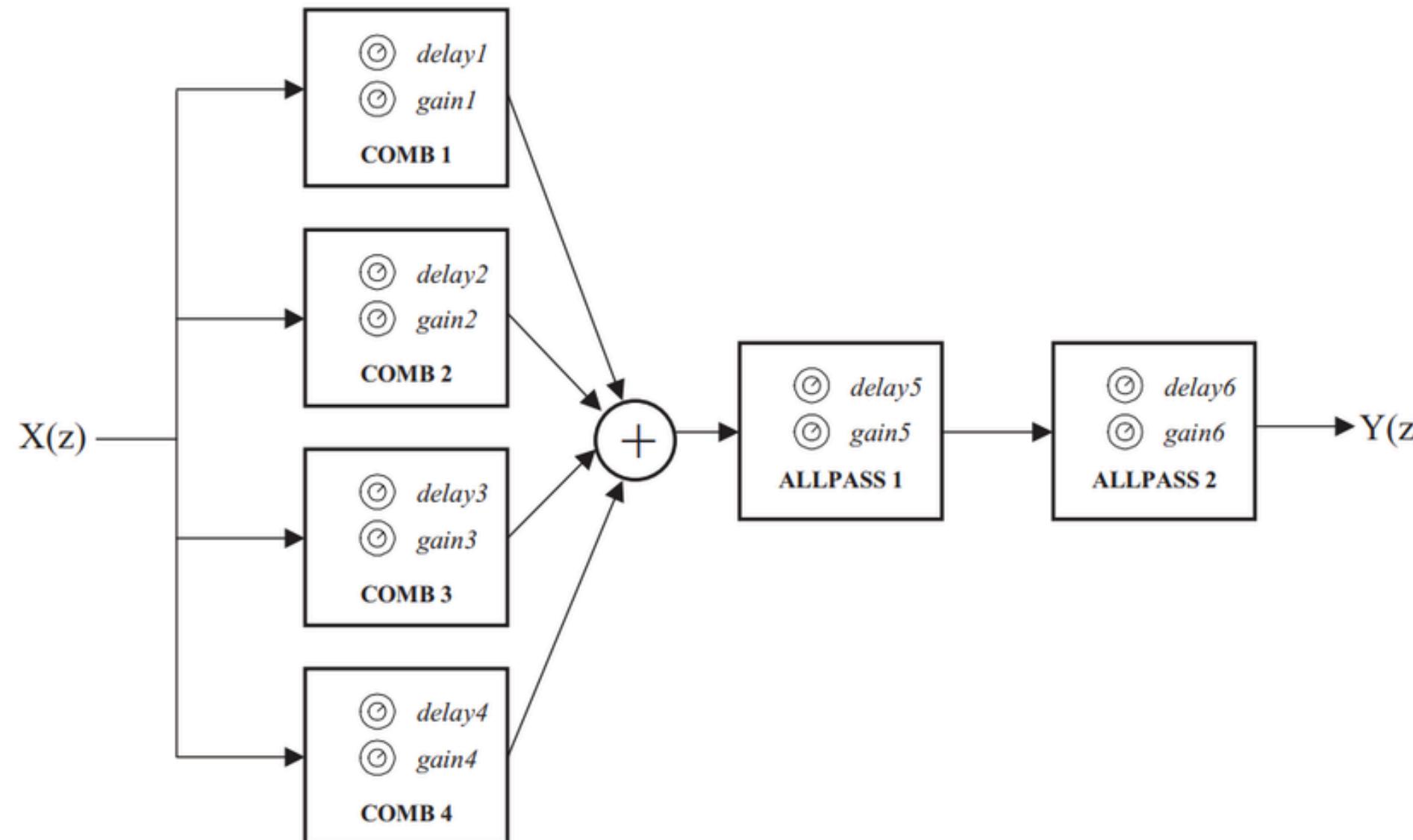


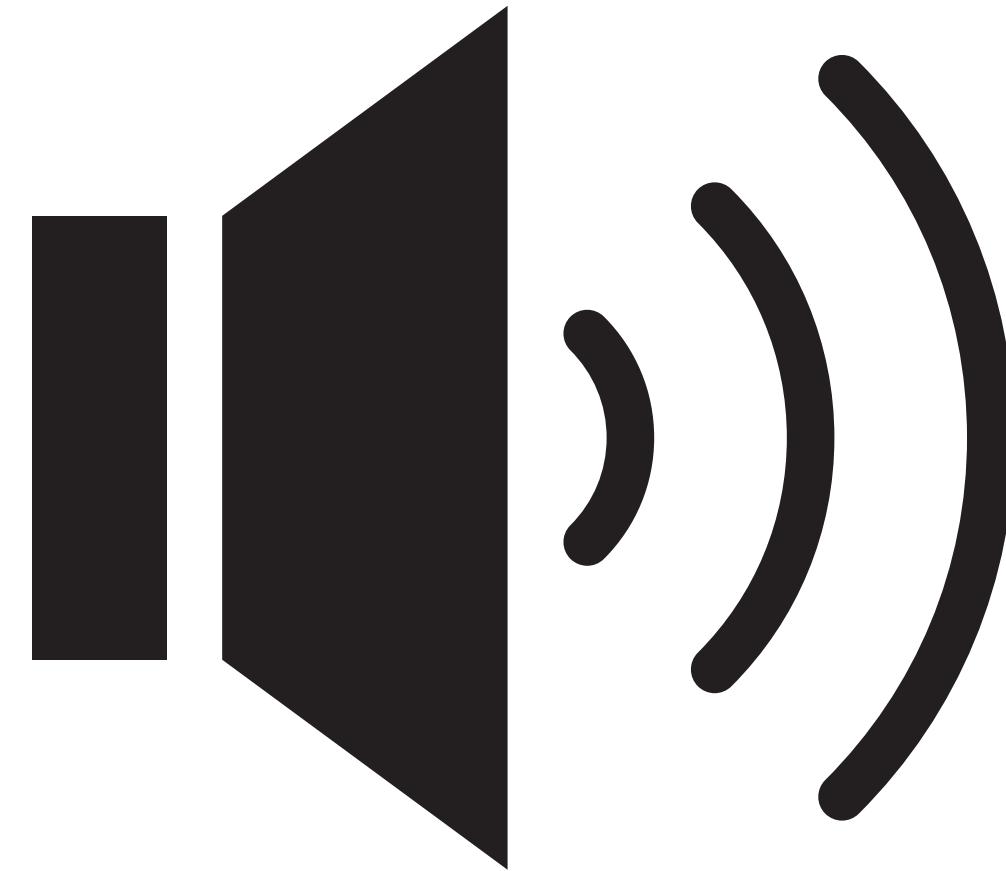
FIG. 2. Modification of simple reverberator. By adding proper amount of undelayed signal, frequency response of the reverberator becomes flat (all-pass reverberator).

Reverb: schroeder



Reverb: schroeder

Aplicação



Pitch shifter

Pitch shifting é uma técnica usada para alterar o tom (frequência) de um som sem mudar sua duração. Por exemplo, você pode aumentar o tom de uma música para que ela soe mais aguda ou diminuir o tom para que ela soe mais grave.

A técnica usa linhas de atraso variáveis para criar o efeito. A ideia básica é:

1. Atraso variável: O som é atrasado por um período de tempo que varia ao longo do tempo.
2. Efeito Doppler: Quando o atraso aumenta, o som parece ficar mais grave (tom mais baixo). Quando o atraso diminui, o som parece ficar mais agudo (tom mais alto).
3. Envelopamento: Para evitar "saltos" ou descontinuidades no som, usamos uma envolvente (fade-in e fade-out) para suavizar as transições.

Pitch shifter

A técnica mais comum para implementar o pitch shifting é baseada em janelas de tempo e sobreposição (time-stretching and overlap), que essencialmente divide o sinal em blocos (janelas) e aplica manipulações específicas, como compressão ou expansão, antes de reordená-los. No caso desta implementação, o efeito é alcançado através de:

- Uso de um Buffer de Atraso Circular (Delay Buffer):
 - Um buffer armazena uma quantidade limitada de amostras de áudio.
 - Os índices do buffer são calculados com base em um atraso variável controlado por uma onda sawtooth (dente de serra).
- Oscilações para Criar Deslocamentos:
 - Duas fases (uma para cada linha de atraso) controlam os atrasos em ciclos desfasados (180 graus fora de fase). Isso gera dois fluxos de áudio com diferentes atrasos.
- Interpolação Linear:
 - Suaviza as transições entre os valores das amostras para evitar artefatos de som indesejados causados por atrasos fracionários.
- Modulação da Envolvente:
 - Envelopes são aplicados aos fluxos de áudio para combinar suavemente as duas linhas de atraso.

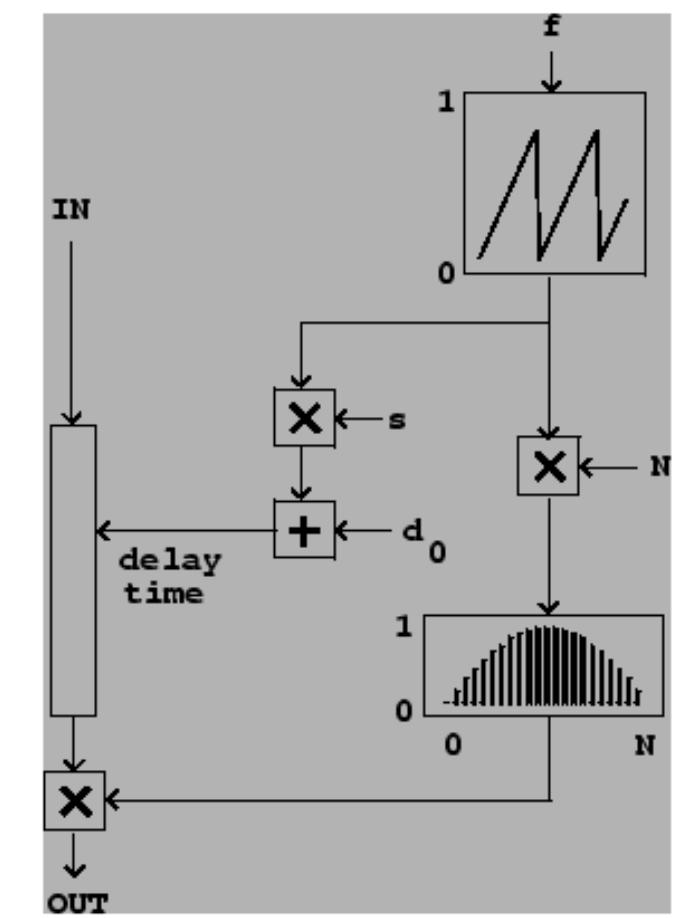


Figure 7.21: Using a variable delay line as a pitch shifter. The sawtooth wave creates a smoothly increasing or decreasing delay time. The output of the delay line is enveloped to avoid discontinuities. Another copy of the same diagram should run 180 degrees (π radians) out of phase with this one.

Pitch shifter

```
// Sawtooth wave is used to generate the delay times
float sawtooth(float phase, float amplitude) {
    return amplitude * (2.0f * (phase - floor(phase)) - 1.0f);
}
```

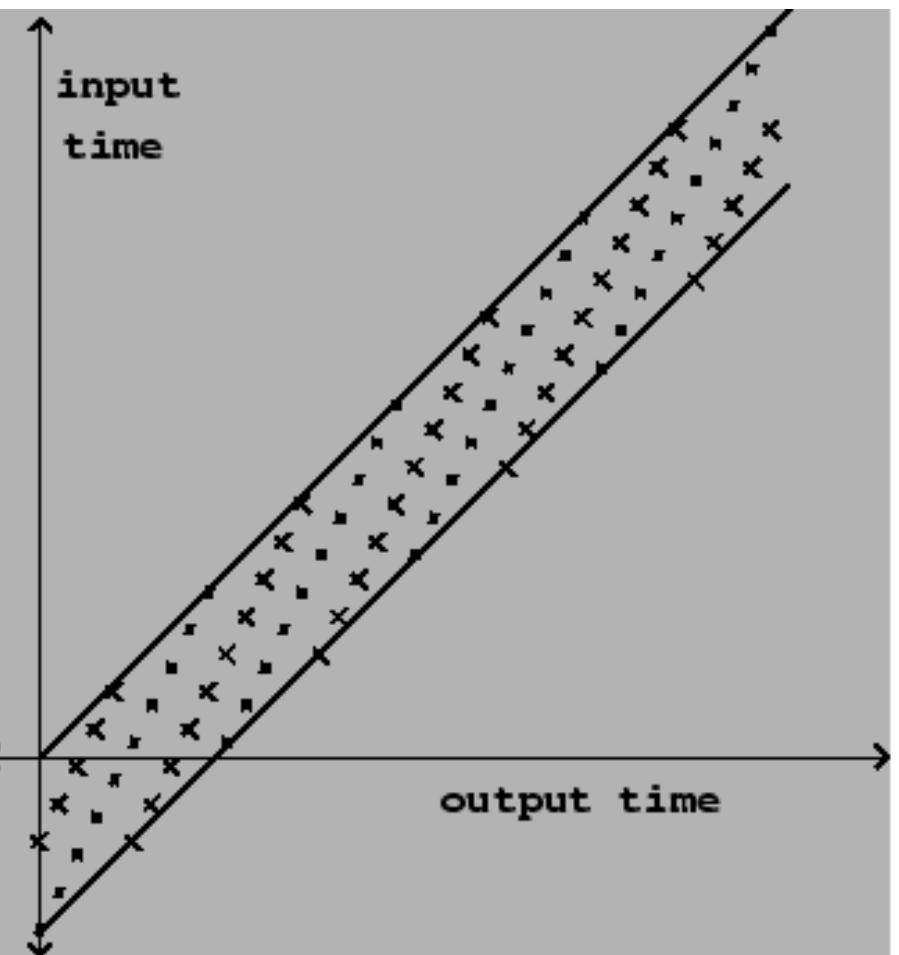


Figure 7.22: The pitch shifter's delay reading pattern using two delay lines, so that one is at maximum amplitude exactly when the other is switching.

```
// Pitch shifting using two variable delay lines with linear interpolation
void shiftPitch(const std::vector<float>& input, std::vector<float>& output, int sampleRate, float pitchshift) {

    // Pitch shifter settings
    const float windowSize = 0.03f; // 30ms window (adjustable)
    const float frequency = sampleRate / windowSize;
    const int delaySamples = static_cast<int>(sampleRate * windowSize);
    const float phaseIncrement = pitchShift / sampleRate;

    std::vector<float> delayBuffer(delaySamples, 0.0f);
    size_t delayIndex = 0;
    float phase1 = 0.0f; // Phase of the first delay line
    float phase2 = 0.5f; // Phase of the second delay line (180 degrees out of phase)

    // Process the input samples
    output.resize(input.size());
    for (size_t i = 0; i < input.size(); ++i) {
        // Generate delay times based on the sawtooth wave
        float delay1 = sawtooth(phase1, delaySamples);
        float delay2 = sawtooth(phase2, delaySamples);

        // calculate the indices in the delay line
        int index1 = static_cast<int>(floor(delayIndex - delay1)) % delaySamples;
        int index2 = static_cast<int>(floor(delayIndex - delay2)) % delaySamples;

        if (index1 < 0) index1 += delaySamples;
        if (index2 < 0) index2 += delaySamples;

        // Linear interpolation to smooth
        float sample1 = delayBuffer[index1];
        float sample2 = delayBuffer[index2];

        // Apply the envelopes
        float env1 = envelope(phase1);
        float env2 = envelope(phase2);

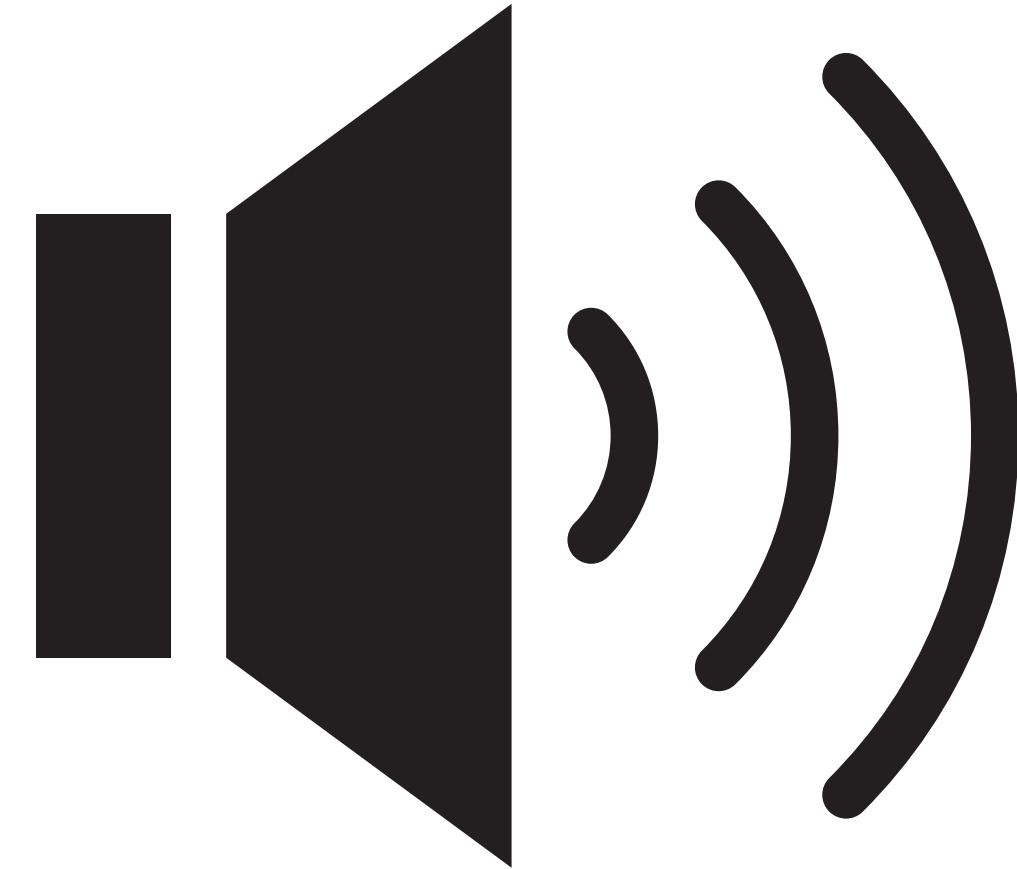
        // Combine the two delays with envelopes
        output[i] = env1 * sample1 + env2 * sample2;

        // Update the delay buffer
        delayBuffer[delayIndex] = input[i];
        delayIndex = (delayIndex + 1) % delaySamples;

        // Increment the phases and normalize
        phase1 = fmod(phase1 + phaseIncrement, 1.0f);
        phase2 = fmod(phase2 + phaseIncrement, 1.0f);
    }
}
```

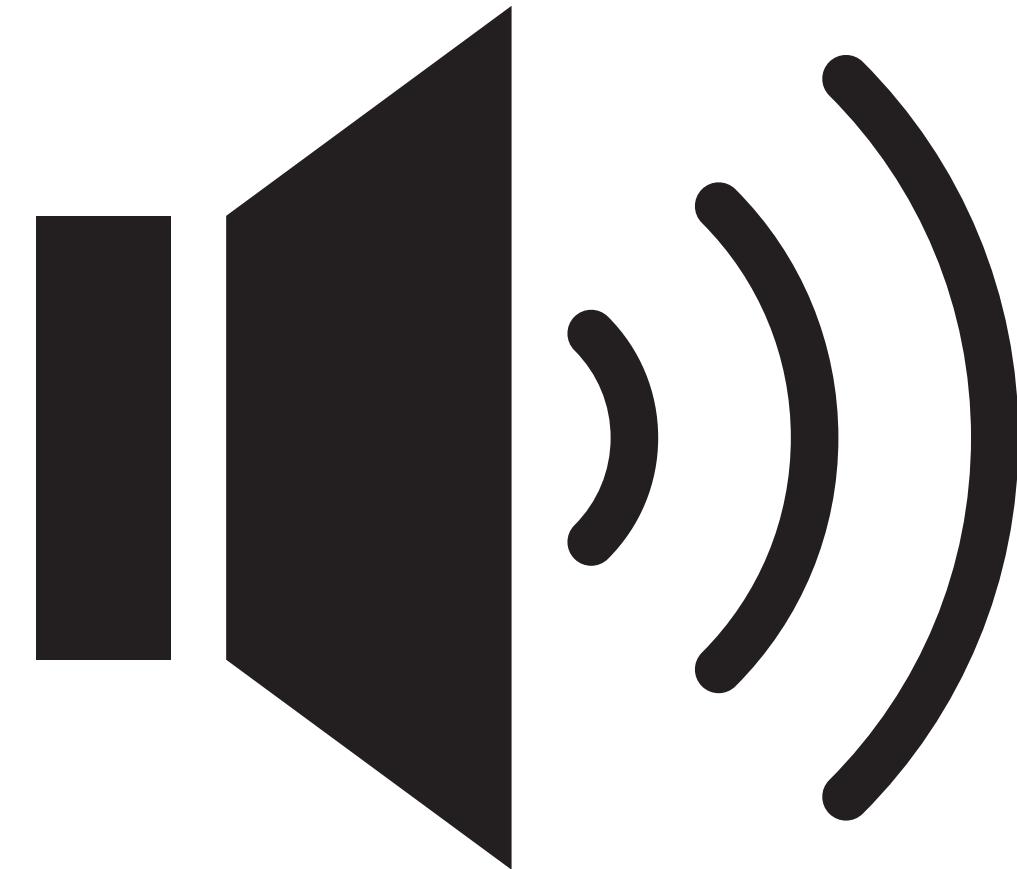
Pitch shifter

Aplicação do shifter para simular uma voz grossa



Pitch shifter

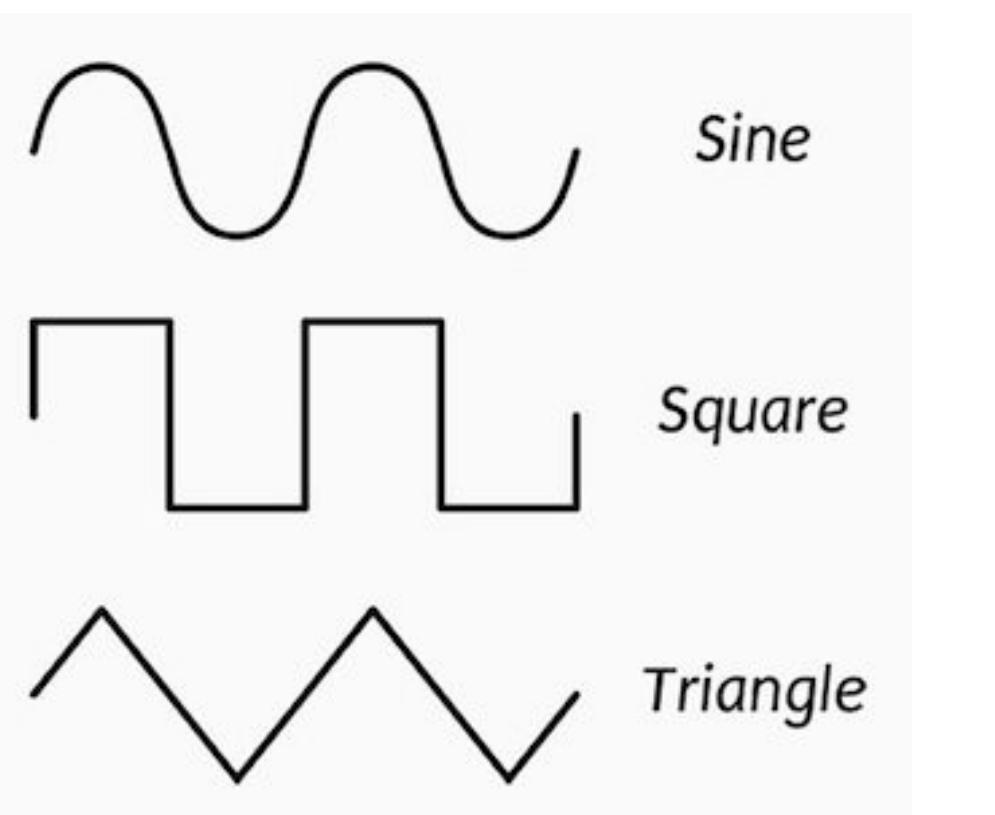
Aplicação do shifter para simular uma voz aguda



Tremolo

O tremolo é um efeito de áudio que modula a amplitude (volume) de um sinal sonoro de forma periódica, criando uma variação rítmica no volume. Esse efeito é amplamente utilizado em música para adicionar textura e movimento ao som.

Implementamos um tremolo combinando diferentes formas de onda (quadrada, triangular e parabólica), cada uma com características sonoras distintas. Dessa forma, foi possível suavizar eventos de transição.



```
float clip(float n, float lower, float upper) {
    return std::max(lower, std::min(n, upper));
}

void tremolo(std::vector<float>& samples, int sampleRate, uint32_t milliseconds) {
    half_cycle_samples = milliseconds * sampleRate / 1000 / 2;
    position = 0;
    square_state = 1;
    clickless_sq_state = 1;
    triangle_state = -1;
    parabolic_state = 0;
    waveform = 2;

    for (size_t i = 0; i < samples.size(); ++i) {
        if (position > half_cycle_samples) {
            square_state = -square_state;
            position = 0;
        }

        // An actual square wave will make clicking sounds at the
        // edges when the audio suddenly cuts in and out. To avoid
        // that, we do a quick fade-in and fade-out instead of a
        // sudden cut.
        if (square_state > clickless_sq_state) {
            clickless_sq_state += 0.02f;
        } else if (square_state < clickless_sq_state) {
            clickless_sq_state -= 0.02f;
        }

        // Clamp the output value between -1 and 1, because float
        // point, discrete math isn't precise and we don't want any
        // little offset from the floats or sampling to add up over
        // time.
        clickless_sq_state = clip(clickless_sq_state, -1.0f, 1.0f);

        // To get a triangle wave, we just add up the square samples,
        // but scale them down so that the positive cycle samples sum
        // to 2, taking our triangle wave from -1 to 1, and subtract 2
        // during the negative to take us back to -1.
        triangle_state += square_state / (half_cycle_samples / 2.0f);
        triangle_state = clip(triangle_state, -1.0f, 1.0f);

        // To get something more like a sine wave, we can add up the
        // triangle wave samples, again scaling so the wave ranges
        // from -1 to 1 (each cycle of the triangle wave has half the
        // area of a square, so we divide by 2 again). This produces a
        // parabolic wave, which is close enough to a sine wave for
        // our purposes.
        parabolic_state += triangle_state / (half_cycle_samples / 4.0f);
        parabolic_state = clip(parabolic_state, -1.0f, 1.0f);

        // Shift the final wave form into the range from 0 to 1, then
        // multiply by the audio sample to scale its volume.
        float waveform_state = clickless_sq_state;
        if (waveform == 1) {
            waveform_state = triangle_state;
        } else if (waveform == 2) {
            waveform_state = parabolic_state;
        }

        samples[i] *= (waveform_state + 1) / 2.0f;
        position++;
    }
}
```

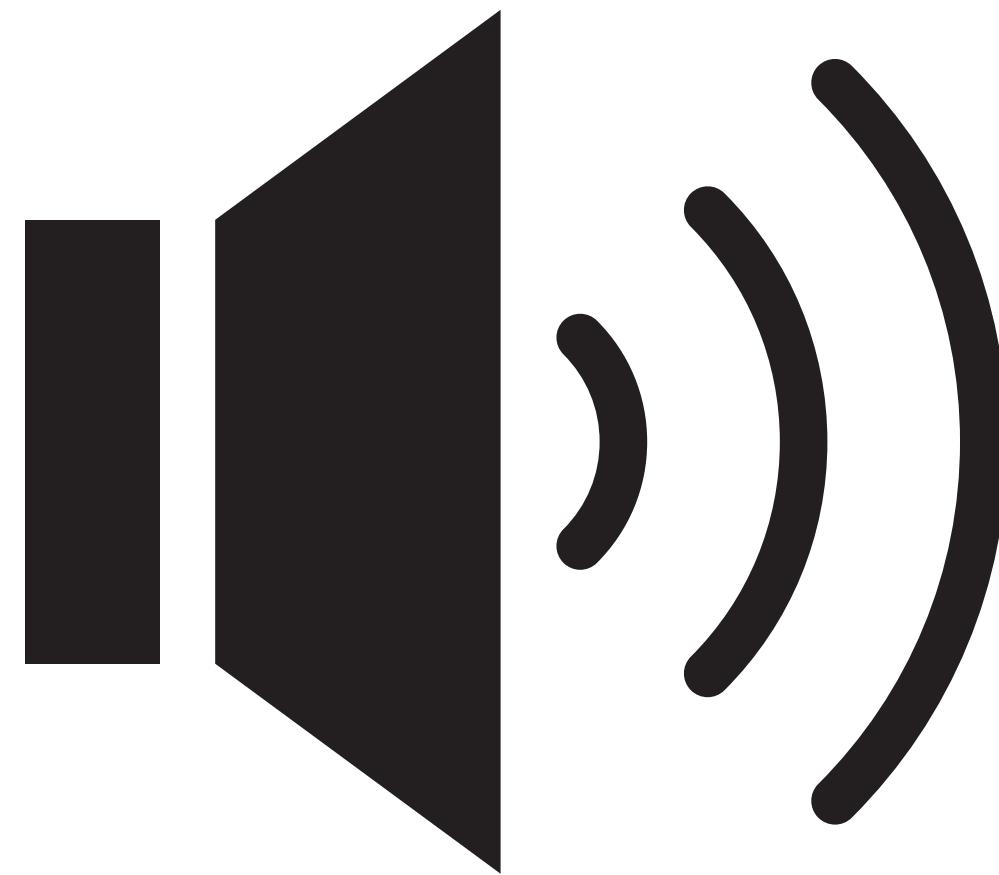
Tremolo

- Alterna entre 1 e -1 a cada meio ciclo.
- Suaviza as transições para evitar cliques.
- Quadrado para Triângulo:
 - O square_state é equivalente a uma derivada constante (ou seja, +1 ou -1).
 - Integrar o square_state cria uma função linear — a forma triangular.
- Triângulo para Parabólico:
 - O triangle_state é uma função linear (que cresce ou decresce linearmente com o tempo).
 - Integrar uma função linear cria uma parábola (parabolic_state).
 - A divisão por half_cycle_samples / 4 escala adequadamente os valores para manter o parabolic_state dentro do intervalo [-1, 1].

```
if (square_state > clickless_sq_state) {  
    clickless_sq_state += 0.02f;  
} else if (square_state < clickless_sq_state) {  
    clickless_sq_state -= 0.02f;  
}
```

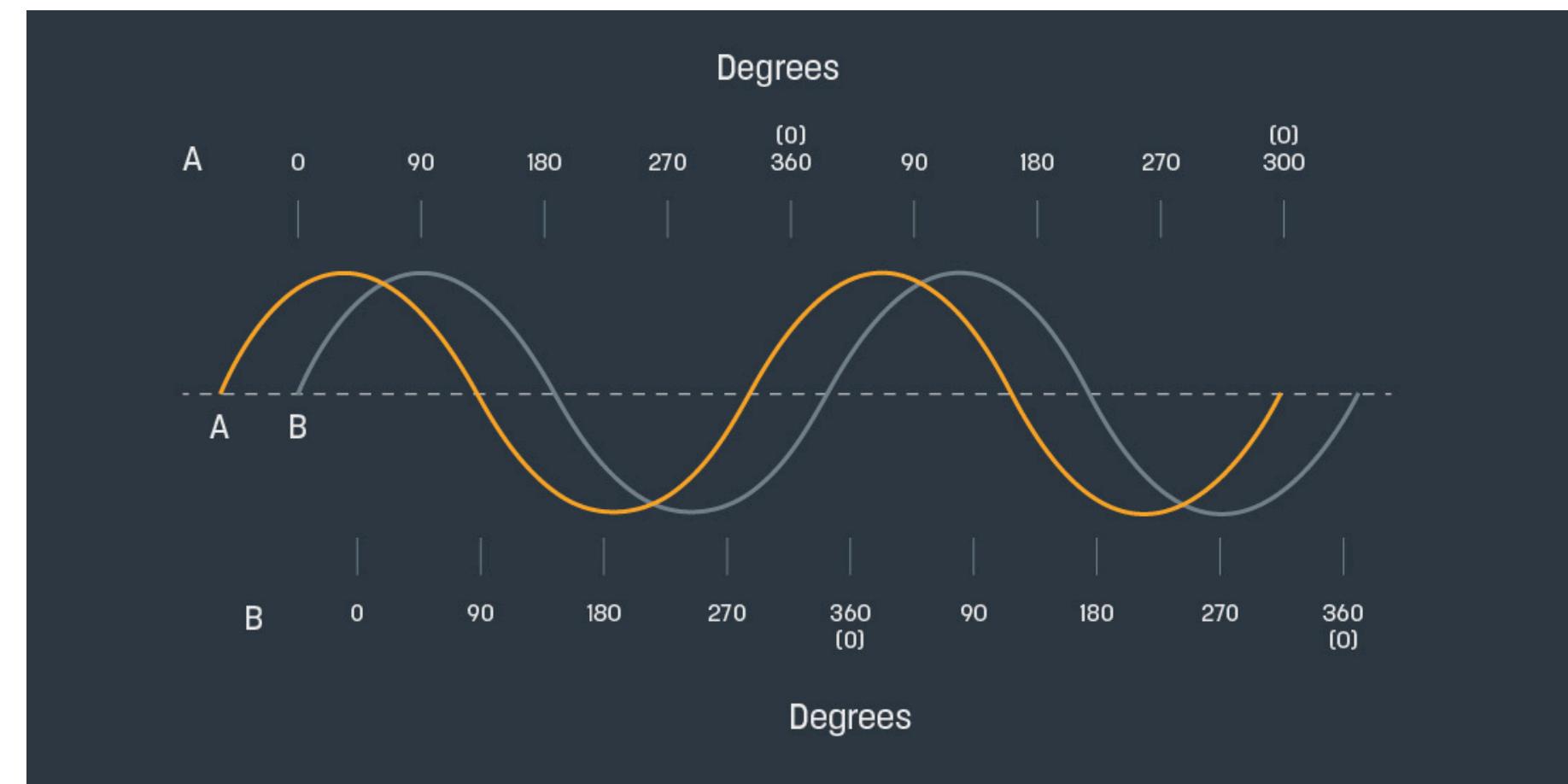
```
triangle_state += square_state / (half_cycle_samples / 2.0f);  
triangle_state = clip(triangle_state, -1.0f, 1.0f);  
  
parabolic_state += triangle_state / (half_cycle_samples / 4.0f);  
parabolic_state = clip(parabolic_state, -1.0f, 1.0f);
```

Tremolo



Flanger

O flanger é um efeito de áudio que cria uma sensação de movimento ou "oscilação" adicionando ao sinal original uma versão dele mesmo atrasada no tempo e modulando esse atraso ao longo do tempo. O efeito é conhecido por criar texturas sonoras semelhantes a "jatos" ou "ondas".

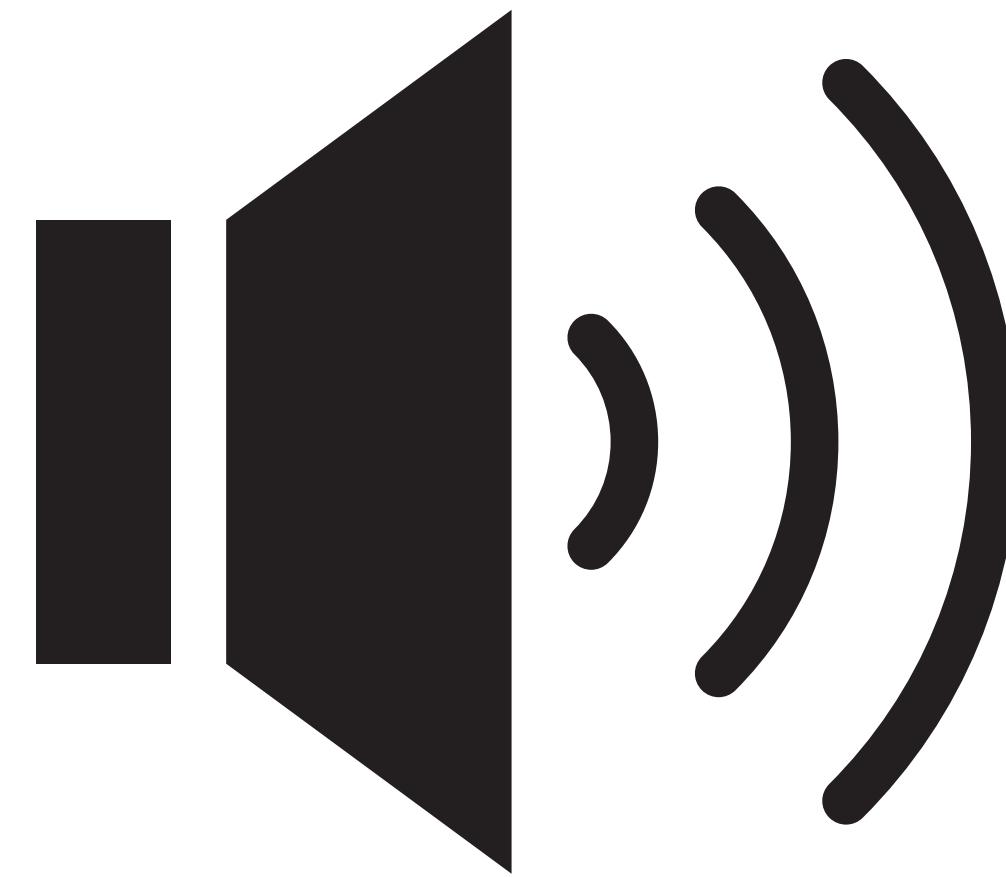


Flanger

- Cálculo do atraso atual
 - O atraso é modulado pelo LFO (função seno)
 - O atraso resultante oscila entre delayMs - depthMs e delayMs + depthMs.
- Leitura com Interpolação Linear
 - Calcula-se a posição de leitura:
 - O valor readPos pode ser fracionário e, para evitar artefatos, é feita uma interpolação linear.
- Escrita no Buffer
 - A amostra de entrada atual é gravada no buffer.
- Combinação dos Sinais
 - A amostra processada é uma combinação do sinal original e do sinal atrasado.
- Atualização dos Índices e Fase
 - O índice de escrita é incrementado circularmente.
 - A fase do LFO é ajustada.

```
void flanger(std::vector<float>& samples, int sampleRate, float delayMs, float depthMs, float rateHz) {  
    // Initialize flanger state  
    FlangerState state;  
    state.phase = 0.0f;  
    float lfoFrequency = rateHz / sampleRate;  
  
    // Calculate buffer size based on maximum possible delay  
    int delayBufferSize = static_cast<int>((delayMs + depthMs) * sampleRate / 1000.0f);  
    // The delay buffer acts as a circular buffer using modulo operations  
    state.delayBuffer.resize(delayBufferSize, 0.0f);  
    state.writeIndex = 0;  
  
    // Process each sample  
    for (size_t i = 0; i < samples.size(); ++i) {  
        // Calculate current delay using LFO  
        float currentDelayMs = delayMs + depthMs * sinf(2.0f * M_PI * state.phase);  
        float currentDelaySamples = currentDelayMs * sampleRate / 1000.0f;  
  
        // Get delay read position with linear interpolation  
        float readPos = state.writeIndex - currentDelaySamples;  
        if (readPos < 0) readPos += delayBufferSize;  
  
        int readIndex = static_cast<int>(readPos);  
        float frac = readPos - readIndex;  
  
        // Ensure indices are within bounds  
        int readIndex2 = (readIndex + 1) % delayBufferSize;  
        readIndex = readIndex % delayBufferSize;  
  
        // Linear interpolation of delayed signal helps avoid artifacts from fractional delay times  
        float delayedSample = state.delayBuffer[readIndex] * (1.0f - frac) + state.delayBuffer[readIndex2] * frac;  
  
        // Write input to buffer  
        state.delayBuffer[state.writeIndex] = samples[i];  
  
        // Mix original and delayed signal  
        samples[i] = 0.5f * (samples[i] + delayedSample);  
  
        // Update indices and phase  
        state.writeIndex = (state.writeIndex + 1) % delayBufferSize;  
        // The phase accumulator wraps between 0 and 1 to create a continuous LFO cycle  
        state.phase += lfoFrequency;  
        if (state.phase >= 1.0f) state.phase -= 1.0f;  
    }  
}
```

Flanger



Efeitos em C

- >>> Desenvolvimento inicial dos efeitos em C++.
- >>> Tradução dos efeitos para C.
- >>> Aplicação do software no CCS.



Efeitos em C

- »»» O programa lê um arquivo de áudio WAV, aplica quatro efeitos de processamento (reverb, tremolo, flanger e pitch shifter), e salva cada resultado em arquivos de saída separados.

Efeitos em C

- >>> WAVHeader: struct que armazena informações do cabeçalho do arquivo
(ex. sampleRate, bitsPerSample, numChannels, subchunk2Size)

```
#include "file.h"

Qodo Gen: Options | Test this function
int16_t *read_wav(const char *filename, WAVHeader *header)
{
    ...

Qodo Gen: Options | Test this function
void write_wav(const char *filename, WAVHeader *header,
    int16_t *data) { ... }
```

Efeitos em C

- >>> O áudio original é copiado em quatro buffers diferentes (um para cada efeito).

```
// copy the data to use in reverb  
int16_t *reverb_data = (int16_t *)malloc(numSamples * sizeof(int16_t));  
memcpy(reverb_data, data, numSamples * sizeof(int16_t));
```

- >>> Aplica as funções de efeitos nas cópias e salva em um arquivo separado.

```
// Apply the tremolo effect  
apply_tremolo(tremolo_data, numSamples, header.sampleRate, 500);  
const char *tremolo_output = "tremolo_output.wav";  
write_wav(tremolo_output, &header, tremolo_data);
```

Efeitos em C

>>> reverb.h

```
float band_pass_filter(float sample, float centerFreq, float q,
                      float sampleRate, float *state);

void all_pass_filter(float *samples, size_t numSamples, float delayMs, float decayGain, int
sampleRate);

void comb_filter(float *samples, size_t numSamples, float delayMs, float decayGain, int
sampleRate);

void reverse_samples(int16_t *data, uint32_t numSamples);

void apply_reverb_simple(int16_t *data, uint32_t numSamples, uint32_t sampleRate, float delay,
float decay);

void apply_reverb_hall(int16_t *data, uint32_t numSamples, uint32_t sampleRate, float delay,
float decay);

void shroeder(int16_t *samples, size_t numSamples, int sampleRate);

void mix_dry_wet(int16_t *dry, int16_t *wet, size_t numSamples, float wetLevel);

void apply_reverb_shroeder(int16_t *dry_samples, size_t numSamples, int sampleRate, float
wetLevel);
```

Efeitos em C

>>> tremolo.h

```
/**  
 * @brief Applies a tremolo effect to an array of audio samples.  
 *  
 * This function modulates the amplitude of the audio samples using  
 * a low-frequency oscillator (LFO) to create a tremolo effect. The  
 * modulation is based on the provided duration in milliseconds.  
 *  
 * @param samples      Pointer to the array of 16-bit audio samples.  
 * @param num_samples   The total number of samples in the array.  
 * @param sampleRate    The sample rate of the audio in Hz (e.g., 44100).  
 * @param milliseconds Duration of the tremolo effect in milliseconds.  
 *  
 * @note The function modifies the original samples in place.  
 */  
  
void apply_tremolo(int16_t *samples, size_t num_samples, int sampleRate, uint32_t milliseconds);  
  
#endif // TREMOLO_H
```

Efeitos em C

>>> flanger.h

```
/**  
 * @brief Applies the flanger effect to the given audio samples.  
 *  
 * @param samples      Array of audio samples to apply the effect to (int16_t).  
 * @param numSamples   Number of samples in the `samples` array.  
 * @param sampleRate   The sample rate of the audio (e.g., 44100 Hz).  
 * @param delayMs     Delay time in milliseconds.  
 * @param depthMs     Depth of the modulation in milliseconds.  
 * @param rateHz      Rate of the LFO in Hertz.  
 */  
void apply_flanger(int16_t *samples, size_t numSamples, int sampleRate, float delayMs, float  
depthMs, float rateHz);  
  
#endif // FLANGER_H
```

Efeitos em C

>>> pitch_shifter.h

```
/**  
 * @brief Applies pitch shifting to an audio signal using two variable delay lines with linear  
 * interpolation.  
 * @param samples - Pointer to the input audio samples.  
 * @param numSamples - Number of samples in the input signal.  
 * @param sampleRate - The sample rate of the audio signal.  
 * @param pitchShift - The pitch shift factor.  
 */  
void apply_pitch_shifter(int16_t *samples, size_t numSamples, int sampleRate, float pitchShift);  
  
#endif // PITCH_SHIFTER_H
```

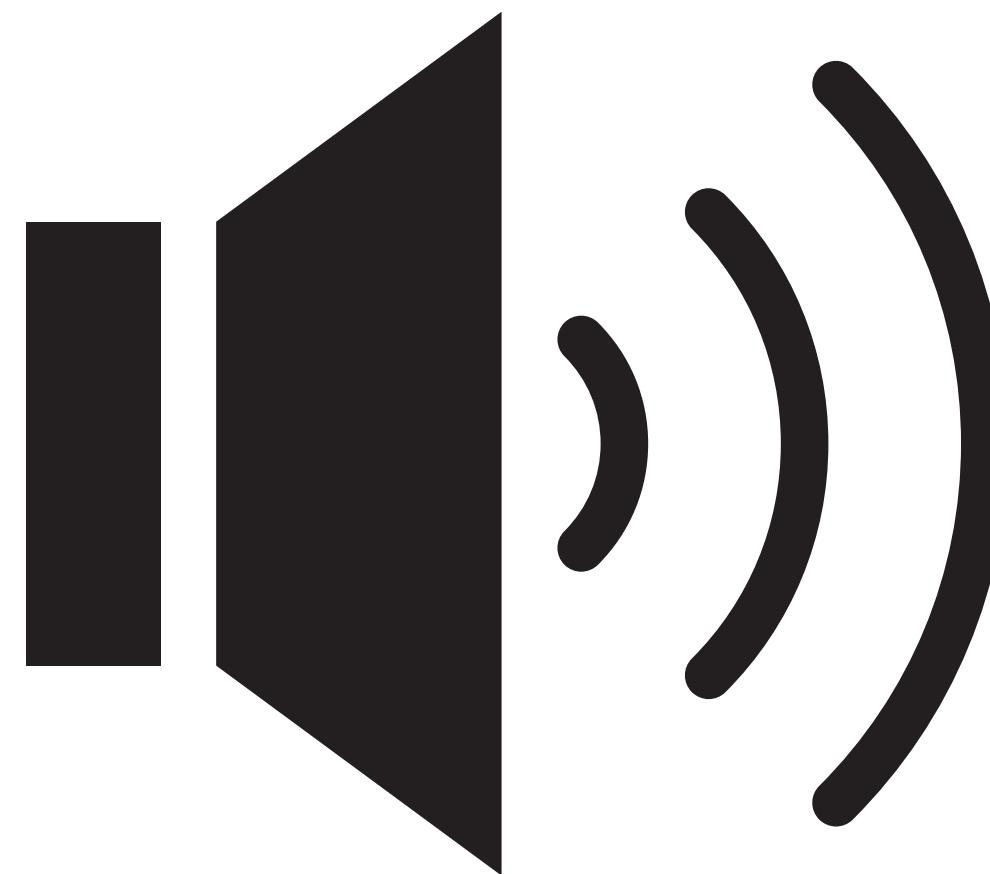
Comparação Python e C

- »»» Medição direta utilizando a biblioteca “time” das linguagens.
- »»» Aplicou-se os mesmos efeitos para ambos os casos (reverb, flanger, tremolo e pitch shifting).
- »»» Cada efeito foi aplicado individualmente em arquivos de saída diferentes.

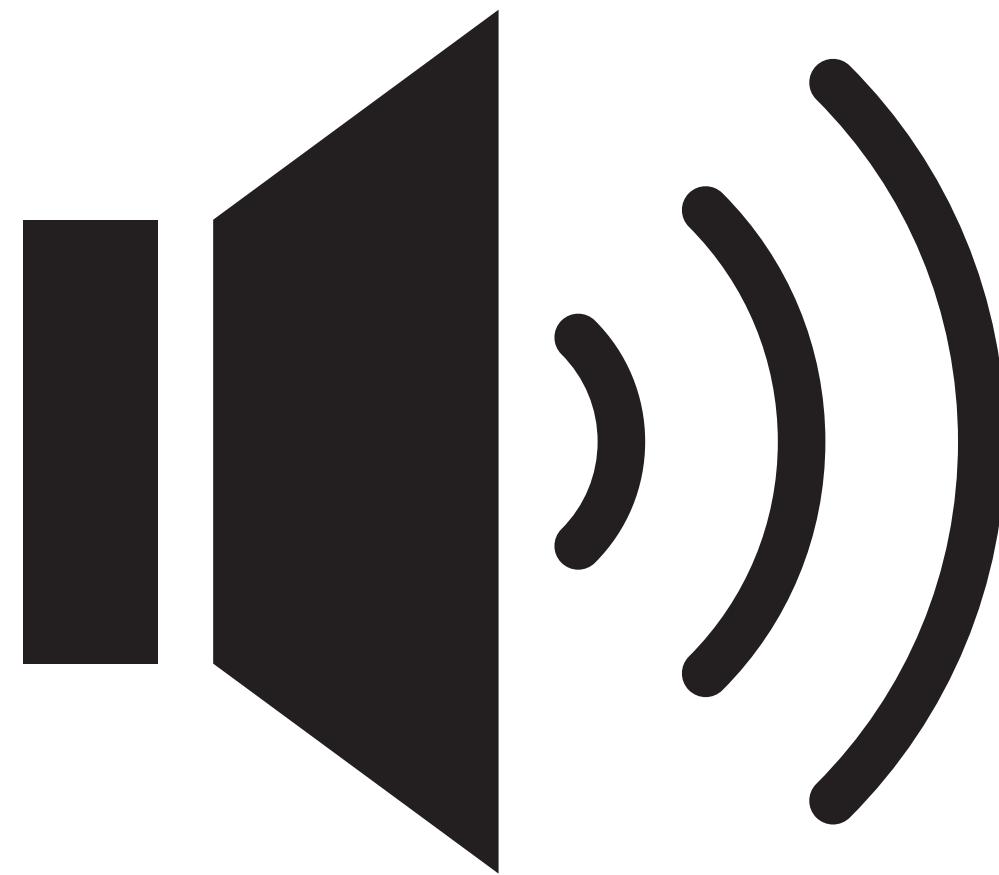
Python: Total processing time: 0.218 seconds

C: Total processing time: 0.116 seconds

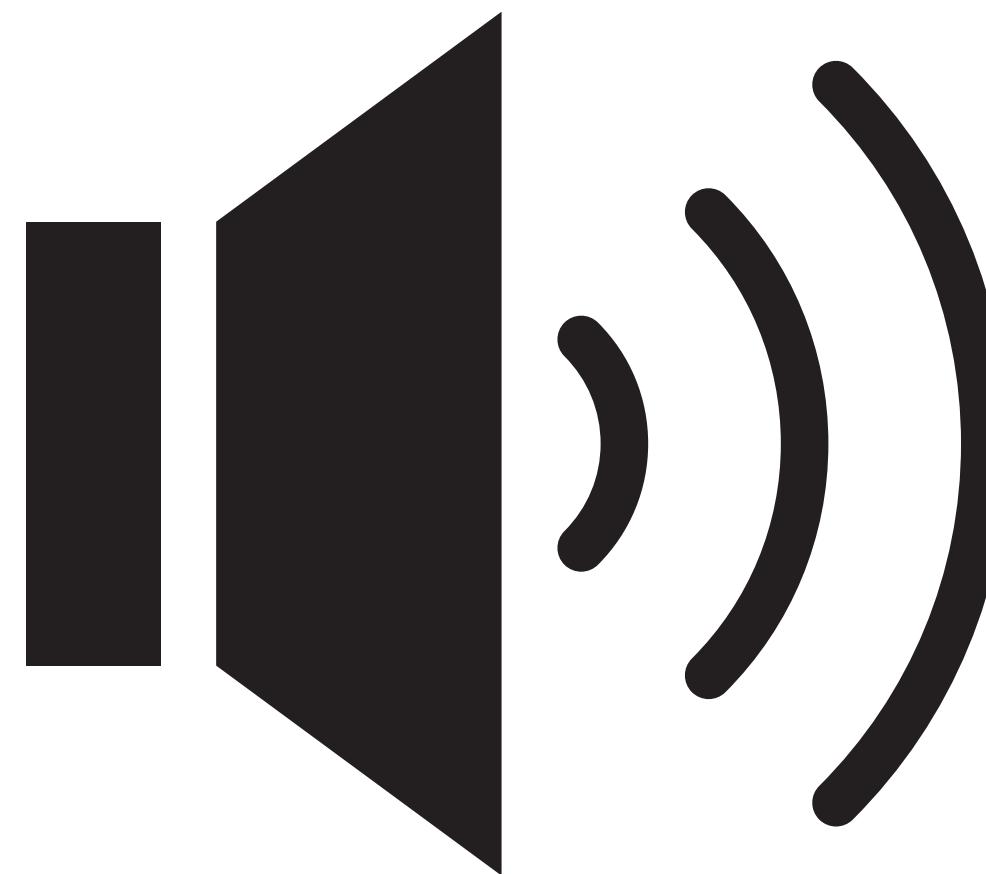
Reverb



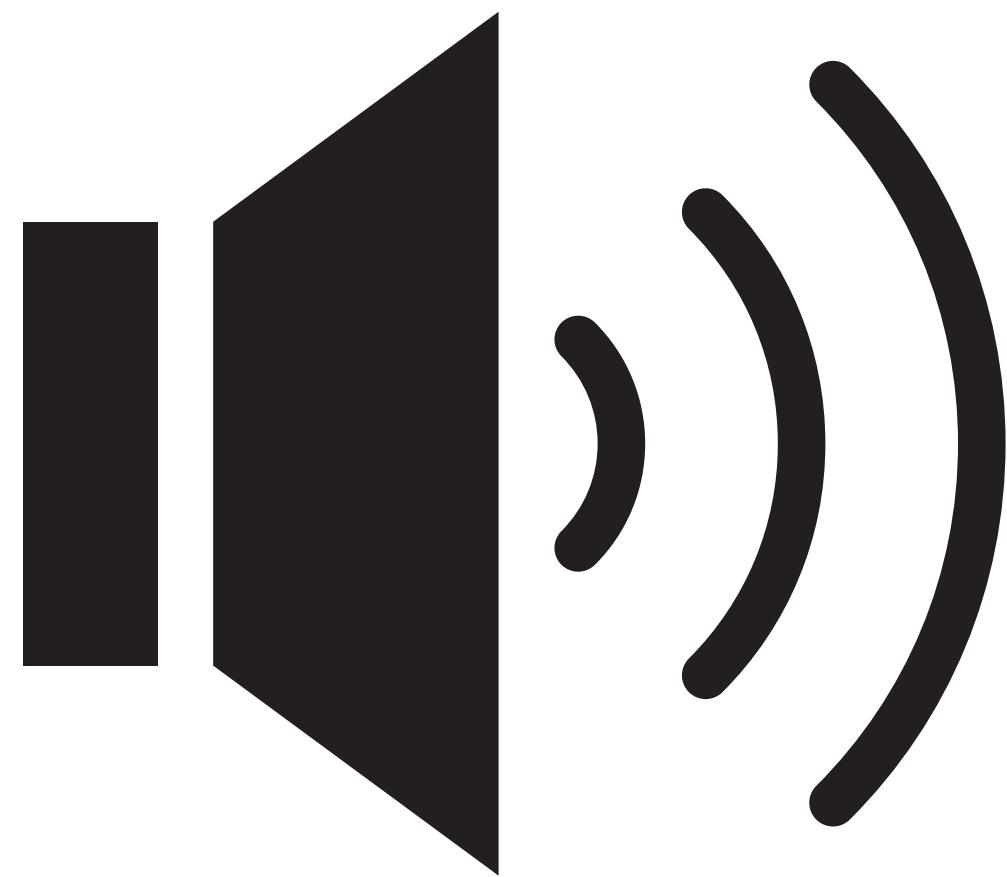
Tremolo



Flanger

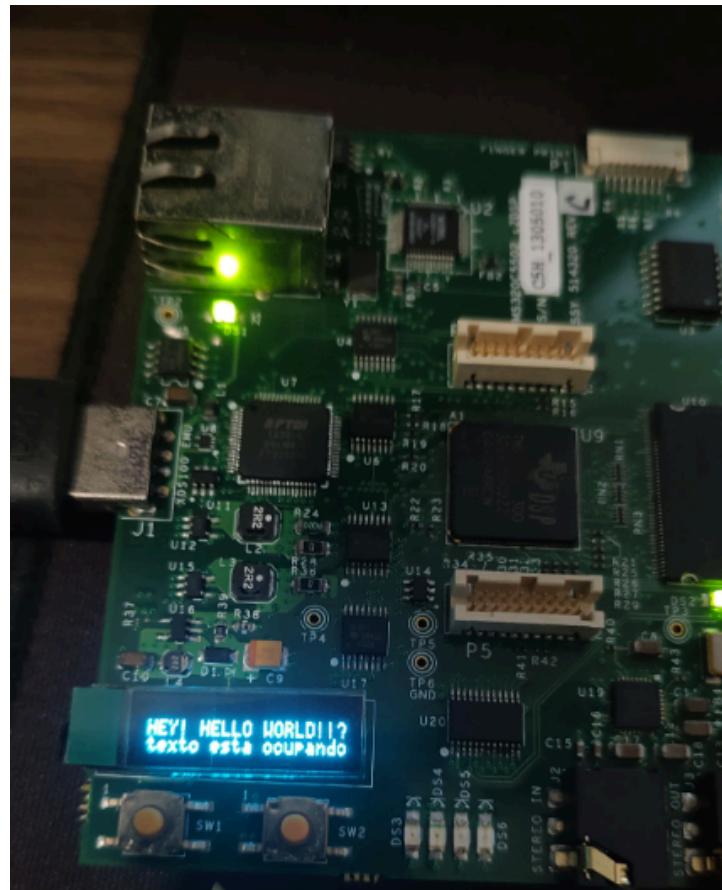


Pitch Shifter

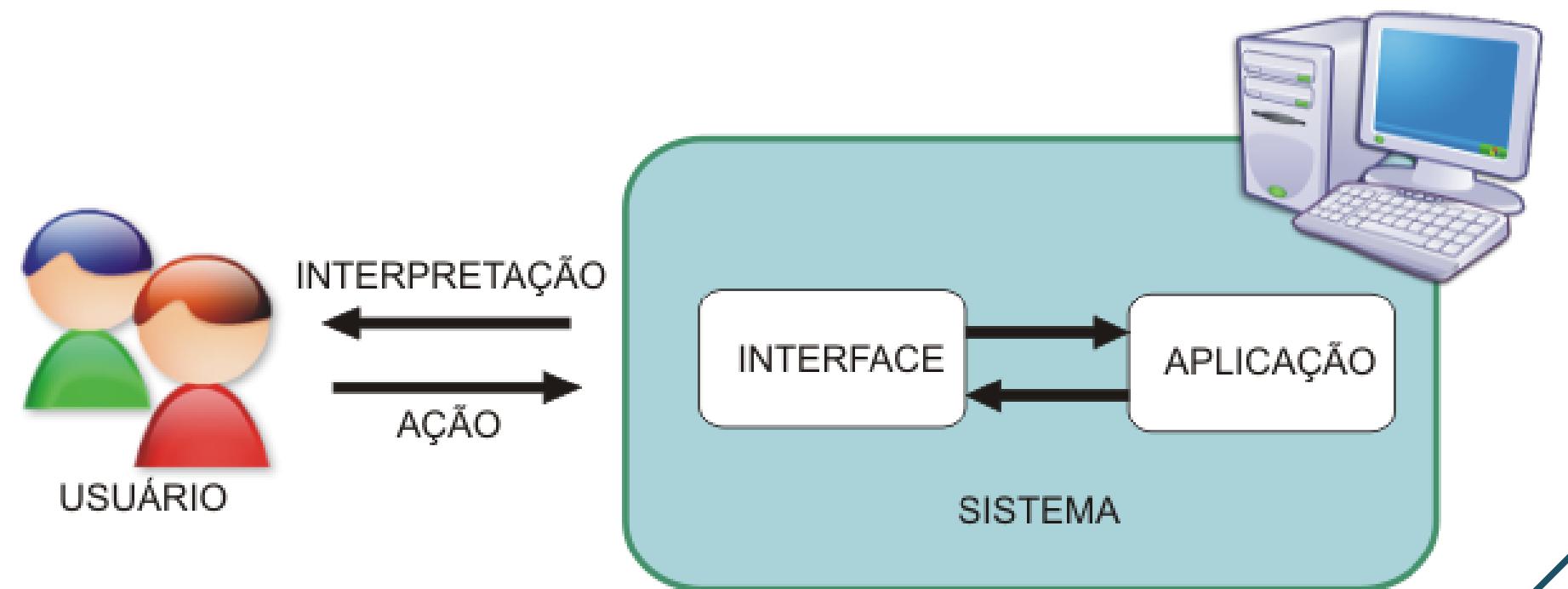


IHM

- Objetivos
- Desafios



```
⌄ ihm
  C display.c
  C display.h
  C font.c
  C lcd.c
  C lcd.h
  C switch.c
  C switch.h
```



font.c

```
#include "ezdsp5502.h"

const Uint16 fonte[65][4] = {
    // Letras maiusculas A-Z
    {0x1F, 0x05, 0x05, 0x1F}, // A
    {0x36, 0x49, 0x49, 0x7F}, // B
    {0x22, 0x41, 0x41, 0x3E}, // C
    {0x3E, 0x41, 0x41, 0x7F}, // D
    {0x41, 0x49, 0x49, 0x7F}, // E
    {0x01, 0x09, 0x09, 0x7F}, // F
    {0x72, 0x51, 0x41, 0x3E}, // G
    {0x7F, 0x08, 0x08, 0x7F}, // H
    {0x00, 0x41, 0x7F, 0x41}, // I
    {0x3F, 0x41, 0x40, 0x20}, // J
    {0x63, 0x14, 0x08, 0x7F}, // K
    {0x40, 0x40, 0x40, 0x7F}, // L
    {0x7F, 0x02, 0x02, 0x7F}, // M
    {0x7F, 0x18, 0x06, 0x7F}, // N
    {0x3E, 0x41, 0x41, 0x3E}, // O
    {0x06, 0x09, 0x09, 0x7F}, // P
    {0x3E, 0x61, 0x41, 0x3E}, // Q
    {0x66, 0x19, 0x09, 0x7F}, // R
    {0x32, 0x49, 0x49, 0x26}, // S
    {0x01, 0x01, 0x7F, 0x01}, // T
    {0x3F, 0x40, 0x40, 0x3F}, // U
    {0x1F, 0x20, 0x20, 0x1F}, // V
    {0x7F, 0x20, 0x20, 0x7F}, // W
    {0x63, 0x08, 0x14, 0x63}, // X
    {0x03, 0x04, 0x78, 0x03}, // Y
    {0x47, 0x49, 0x51, 0x61}, // Z
};

// Letras minusculas a-z
{0x78, 0x54, 0x54, 0x20}, // a
{0x30, 0x48, 0x48, 0x7F}, // b
{0x28, 0x44, 0x44, 0x38}, // c
{0x7F, 0x48, 0x48, 0x30}, // d
{0x18, 0x54, 0x54, 0x38}, // e
{0x01, 0x09, 0x7E, 0x08}, // f
{0x3C, 0x54, 0x54, 0x08}, // g
{0x70, 0x08, 0x08, 0x7F}, // h
{0x40, 0x7A, 0x48, 0x00}, // i
{0x3A, 0x48, 0x40, 0x20}, // j
{0x44, 0x28, 0x10, 0x7F}, // k
{0x00, 0x40, 0x7F, 0x41}, // l
{0x78, 0x04, 0x04, 0x78}, // m
{0x78, 0x04, 0x08, 0x7C}, // n
{0x38, 0x44, 0x44, 0x38}, // o
{0x08, 0x14, 0x14, 0x7C}, // p
{0x7C, 0x14, 0x14, 0x08}, // q
{0x08, 0x04, 0x08, 0x7C}, // r
{0x24, 0x54, 0x54, 0x48}, // s
{0x20, 0x44, 0x3F, 0x04}, // t
{0x7C, 0x40, 0x40, 0x3C}, // u
{0x1C, 0x20, 0x20, 0x1C}, // v
{0x3C, 0x40, 0x40, 0x3C}, // w
{0x44, 0x28, 0x28, 0x44}, // x
{0x3C, 0x50, 0x50, 0x0C}, // y
{0x4C, 0x54, 0x64, 0x44}, // z

// Numeros 0-9
{0x45, 0x49, 0x51, 0x3E}, // 0
{0x40, 0x7F, 0x42, 0x00}, // 1
{0x49, 0x51, 0x61, 0x42}, // 2
{0x3B, 0x45, 0x41, 0x21}, // 3
{0x7F, 0x12, 0x14, 0x18}, // 4
{0x39, 0x45, 0x45, 0x27}, // 5
{0x31, 0x49, 0x4A, 0x3C}, // 6
{0x07, 0x09, 0x71, 0x01}, // 7
{0x36, 0x49, 0x49, 0x36}, // 8
{0x1E, 0x29, 0x49, 0x06}, // 9

// Espaco e caracteres especiais
{0x00, 0x00, 0x00, 0x00}, // ' ' (Espaco)
{0x00, 0x00, 0x5F, 0x00}, // '!' (Exclamacao)
{0x0E, 0x51, 0x01, 0x02} // '?' (Interrogacao)
};
```

lcd.c

```
include "ezdsp5502_i2c.h"
include "ezdsp5502_gpio.h"
include "csl_gpio.h"
include "lcd.h"

#define OSD9616_I2C_ADDR 0x3C // Endereco I2C do display OSD9616

int16 osd9616_send(Uint16 comdat, Uint16 data)
{
    Uint16 cmd[2];
    cmd[0] = comdat & 0x00FF; // Especifica se eh Comando ou Dado
    cmd[1] = data;           // Comando ou dado a ser enviado

    /* Escreve no OSD9616 */
    return EZDSP5502_I2C_write(OSD9616_I2C_ADDR, cmd, 2);
}

int16 osd9616_multiSend(Uint16 *data, Uint16 len)
{
    Uint16 x;
    Uint16 cmd[10];
    for (x = 0; x < len; x++) // Comando ou dado a ser enviado
    {
        cmd[x] = data[x];
    }

    /* Escreve no OSD9616 */
    return EZDSP5502_I2C_write(OSD9616_I2C_ADDR, cmd, len);
}
```

```
/*
 * Inicializa um display OLED (provavelmente o modelo OSD9616) via comandos I2C
 */
Int16 osd9616_init()
{
    Uint16 cmd[10]; // Um array cmd eh definido para armazenar ate 10 comandos, mas a funcao nunca

    /* Inicializacao da Alimentacao do LCD */
    EZDSP5502_GPIO_init(GPIO_GPIO_PIN1);                                // Habilita o pino GPIO
    EZDSP5502_GPIO_setDirection(GPIO_GPIO_PIN1, GPIO_GPIO_PIN1_OUTPUT); // Define como saida
    EZDSP5502_GPIO_setOutput(GPIO_GPIO_PIN1, 1);                         // Ativa 13V

    /* Configuracao Inicial do Display */
    osd9616_send(0x00, 0x00); // Define endereco da coluna de baixo
    osd9616_send(0x00, 0x10); // Define endereco da coluna de cima
    osd9616_send(0x00, 0x40); // Define endereco da linha inicial

    /* Controle de Contraste */
    cmd[0] = 0x00 & 0x00FF;
    cmd[1] = 0x81; // Define o controle de contraste
    cmd[2] = 0x7f; // Nivel medio (127 em decimal).
    osd9616_multiSend(cmd, 3);

    /* Mapeamento e Configuracao de Exibicao */
    osd9616_send(0x00, 0xa1); // Re-mapeia os segmentos de 95 a 0
    //                                         * O comando 0xA1 re-mapeia os segmentos da tela,
    //                                         * provavelmente para inverter a orientacao horizontal da exibicao.
    osd9616_send(0x00, 0xa6); // 0xA6 define a tela em modo normal (nao invertido).

    /* Multiplexacao */
    cmd[0] = 0x00 & 0x00FF;
    cmd[1] = 0xa8; // O comando 0xA8 ajusta a multiplexacao da tela
    cmd[2] = 0x0f; // O valor 0x0F define um fator de multiplexacao especifico (16).
    osd9616_multiSend(cmd, 3);

    osd9616_send(0x00, 0xd3); // Define deslocamento de exibicao
    osd9616_send(0x00, 0x00); // Sem deslocamento
    osd9616_send(0x00, 0xd5); // Define o clock de exibicao
    osd9616_send(0x00, 0xf0); // Define a razao do clock
```

```
/* Periodo de Pre-Carga */
cmd[0] = 0x00 & 0x00FF;
cmd[1] = 0xd9;
cmd[2] = 0x22;
osd9616_multiSend(cmd, 3);

/* Configuracao de Pinos de Comunicacao */
cmd[0] = 0x00 & 0x00FF;
cmd[1] = 0xda;
cmd[2] = 0x02;
osd9616_multiSend(cmd, 3);

/* Configuracao de Referencia de Tensao */
osd9616_send(0x00, 0xdb); // Ajusta vcom
osd9616_send(0x00, 0x49); // 0.83*vref

/* Habilitacao do Display */
cmd[0] = 0x00 & 0x00FF;
cmd[1] = 0x8d;
cmd[2] = 0x14;
osd9616_multiSend(cmd, 3);

osd9616_send(0x00, 0xaf); // Liga o painel OLED

return 0;
}

/* Responsavel por exibir uma letra ou simbolo no display OLED OSD9616 */
Int16 printLetter(Uint16 c4, Uint16 c3, Uint16 c2, Uint16 c1)
{
    osd9616_send(0x40, c4); // Coluna 4
    osd9616_send(0x40, c3); // Coluna 3
    osd9616_send(0x40, c2); // Coluna 2
    osd9616_send(0x40, c1); // Coluna 1
    osd9616_send(0x40, 0x00);

    return 0;
}
```

display.c

```
#include "ezdsp5502.h"
#include "lcd.h"
#include "display.h"

int i = 0;

void printChar(char c)
{
    int index;
    // Converter caractere para indice da tabela (A-Z, a-z, 0-9)
    if (c >= 'A' && c <= 'Z')
    {
        index = c - 'A'; // Indice para letras maiusculas
    }
    else if (c >= 'a' && c <= 'z')
    {
        index = c - 'a' + 26; // Indice para letras minusculas
    }
    else if (c >= '0' && c <= '9')
    {
        index = c - '0' + 52; // Indice para numeros
    }
    else if (c == ' ')
    {
        index = 62; // Indice para o espaco
    }
    else if (c == '!')
    {
        index = 63; // Indice para o caractere '!'
    }
    else if (c == '?')
    {
        index = 64; // Indice para o caractere '?'
    }
    else
    {
        return; // Caractere nao suportado
    }

    // Usar a funcao printLetter com os valores da tabela de fontes
    printLetter(fonte[index][0], fonte[index][1], fonte[index][2], fonte[index][3]);
}
```

```
// Arrays for effect names and descriptions
const char *effect_names_show[] = {
    "EFEITO 1 ",
    "EFEITO 2 ",
    "EFEITO 3 ",
    "EFEITO 4 ",
    "EFEITO 5 ",
    "EFEITO 6 ",
    "EFEITO 7 ",
    "EFEITO 8 "};

const char *effect_names_apply[] = {
    "APLICANDO EFEITO 1 ",
    "APLICANDO EFEITO 2 ",
    "APLICANDO EFEITO 3 ",
    "APLICANDO EFEITO 4 ",
    "APLICANDO EFEITO 5 ",
    "APLICANDO EFEITO 6 ",
    "APLICANDO EFEITO 7 ",
    "APLICANDO EFEITO 8 "};

const char *effect_names_ready[] = {
    "EFEITO PRONTO 1 ",
    "EFEITO PRONTO 2 ",
    "EFEITO PRONTO 3 ",
    "EFEITO PRONTO 4 ",
    "EFEITO PRONTO 5 ",
    "EFEITO PRONTO 6 ",
    "EFEITO PRONTO 7 ",
    "EFEITO PRONTO 8 "};

const char *effect_names_executing[] = {
    "EXECUTANDO EFEITO 1 ",
    "EXECUTANDO EFEITO 2 ",
    "EXECUTANDO EFEITO 3 ",
    "EXECUTANDO EFEITO 4 ",
    "EXECUTANDO EFEITO 5 ",
    "EXECUTANDO EFEITO 6 ",
    "EXECUTANDO EFEITO 7 ",
    "EXECUTANDO EFEITO 8 "};

const char *effect_descriptions[] = {
    "REV HALL1 ",
    "REV ROOM2 ",
    "REV STAGE B ",
    "REV STAGE D ",
    "REV STAGE F ",
    "RET STAGE Gb ",
    "FLANGER ",
    "TREMOLO "};
```

```
// Select the appropriate effect names based on the mode
const char **effect_names;
switch (mode)
{
    case 1:
        effect_names = effect_names_show; // Selection mode
        break;
    case 2:
        effect_names = effect_names_apply; // Applying mode
        break;
    case 3:
        effect_names = effect_names_ready; // Ready mode
        break;
    case 4:
        effect_names = effect_names_executing; // Executing mode
        break;
    default:
        effect_names = effect_names_show; // Default to selection mode
        break;
}

// Check if the effect is valid (between 0 and 7)
if (effect >= 0 && effect <= 7)
{
    printString(effect_names[effect]); // Display effect name
}
else
{
    printString("Invalid effect!"); // Display error message
}

clearLine(1); // Limpa a linha 1
writeOnPage(1);

if (effect >= 0 && effect <= 7)
{
    printString(effect_descriptions[effect]); // Display effect description
}
else
{
    printString("Unknown effect.");
}

clearLine(2); // Limpa a linha 2
return 0;
```

switch.c

```
#include "stdio.h"
#include "ezdsp5502.h"
#include "ezdsp5502_i2cgpio.h"
#include "csl_gpio.h"
#include "switch.h"
#include "display.h"

#define NUM_EFFECTS 8
extern void applyEffect(int effect);
extern void execEffect();
Int16 switchEffect(int currentEffect)
{
    EZDSP5502_I2CGPIO_configLine(SW0, IN);
    EZDSP5502_I2CGPIO_configLine(SW1, IN);

    show_effect(currentEffect, 0);
}
```

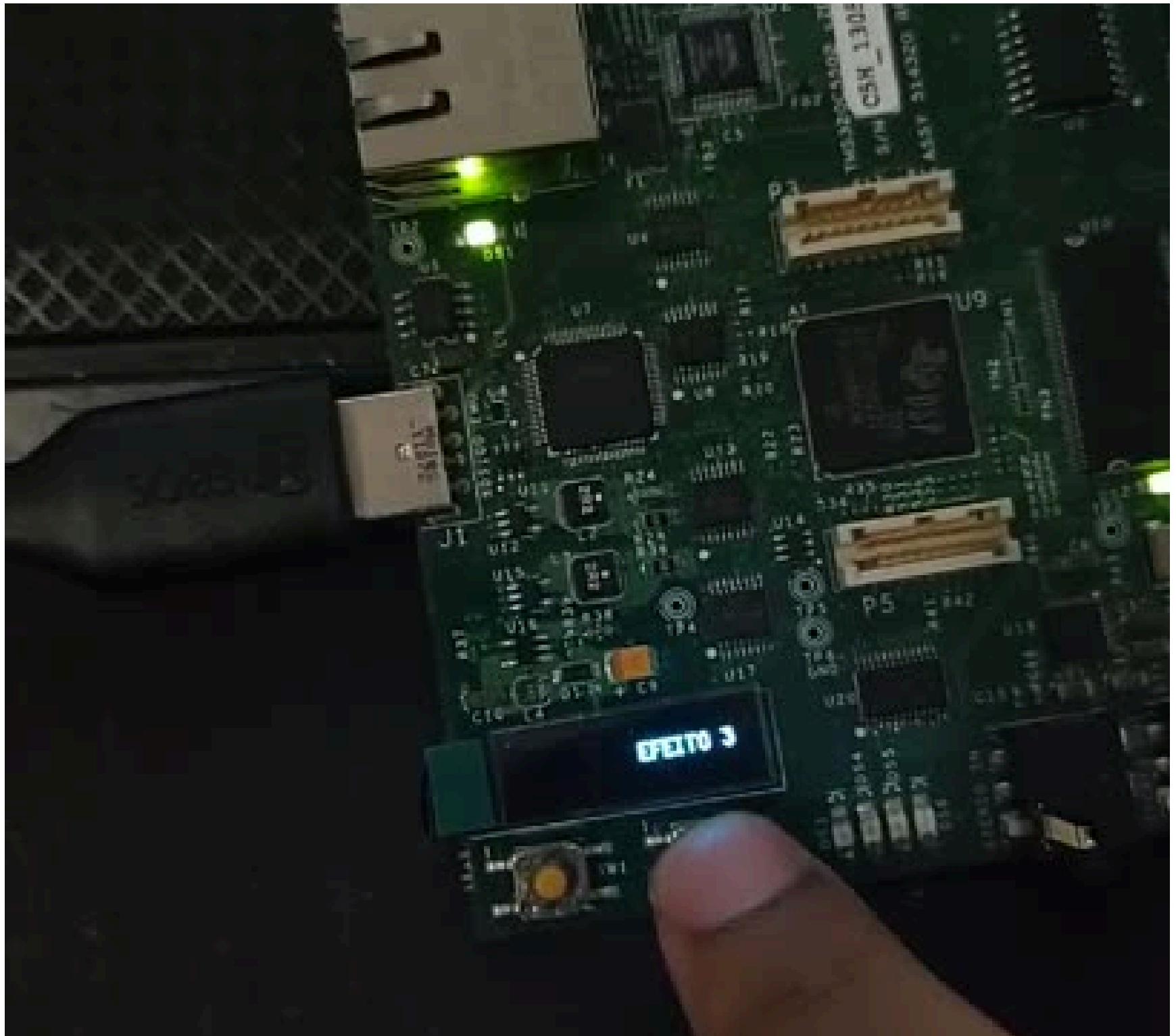
```
while (1)
{
    if (!EZDSP5502_I2CGPIO_readLine(SW0))
    {
        applyEffect(currentEffect);
        show_effect(currentEffect, 3);

        while (1)
        {
            if (!EZDSP5502_I2CGPIO_readLine(SW0))
            {
                while (!EZDSP5502_I2CGPIO_readLine(SW0))
                ;
                execEffect(currentEffect);

                show_effect(currentEffect, 3);
            }
            else if (!EZDSP5502_I2CGPIO_readLine(SW1))
            {
                while (!EZDSP5502_I2CGPIO_readLine(SW1))
                ;
                show_effect(currentEffect, 0);
                break;
            }
        }
    }

    if (!EZDSP5502_I2CGPIO_readLine(SW1))
    {
        currentEffect = (currentEffect + 1) % NUM_EFFECTS;
        show_effect(currentEffect, 0);
        while (!EZDSP5502_I2CGPIO_readLine(SW1))
        ;
    }
}
```

IHM em prática



Referências Bibliográficas

https://hajim.rochester.edu/ece/sites/zduan/teaching/ece472/readings/Schroeder_1962.pdf

