1 噪声抑制算法设计与实现

1.1 设计思路

本算法针对 1800Hz 窄带噪声构建高斯陷波滤波器，通过分帧频域处理实现噪声精准抑制，核心逻辑为将信号分帧后转换至频域，利用高斯型带阻特性衰减目标噪声频率，最终重构时域信号。

1.1.1 背景知识

1. 音频信号的非平稳特性：音频/电信号属于典型的非平稳时变信号，全局处理无法精准针对局部噪声特征，需通过分帧将信号划分为若干短时帧（通常 10 30ms），每帧可近似视为短时平稳信号，为频域处理提供前提。

2. 窄带噪声抑制的通用思路：单一频率/窄带噪声（如电磁干扰、设备谐振噪声）无法通过时域滤波精准抑制，频域陷波滤波是主流方案；传统矩形陷波滤波器易引入吉布斯现象（频谱振铃），高斯型陷波滤波器凭借平滑过渡带可显著降低信号失真。

3. 分帧加窗的通用准则：分帧会导致信号截断，引入频谱泄漏，汉明窗、汉宁窗等余弦类窗函数可通过平滑帧边缘抑制泄漏；重叠率（通常 25% 75%）是平衡帧间连续性与计算效率的关键，高重叠率（如 75%）可避免重构后信号断裂。

4. 频域处理的核心逻辑：快速傅里叶变换（FFT）将时域信号转换为频域，可分离幅值谱与相位谱，仅对幅值谱进行带阻/带通处理（相位保持），能避免音色失真，逆 FFT（IFFT）可将处理后的频域信号转回时域。

5. 信号重构的鲁棒性设计：重叠相加法是分帧处理后信号重构的通用方法，需对重叠区域幅值加权平均（权重为窗函数叠加值），补偿加窗导致的幅值衰减；同时需处理帧长超限、除零等异常，保证输出信号长度与输入一致。

1.1.2 技术原理

1. 分帧与加窗原理：将原始信号划分为若干重叠帧，帧长由采样率和 10ms 时间窗长确定，重叠率 75% 保证帧间连续性。汉明窗表达式为：

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \quad n = 0, 1, ..., N-1$$

其中 N 为帧长，加窗可抑制频域分析的频谱泄漏。

2. 高斯陷波滤波器设计：针对 1800Hz 中心噪声频率，构造高斯型带阻增益函数：

$$bandElimination = 1 - \exp\left(-\frac{d(f)^2}{2\sigma^2}\right)$$

式中 $d(f) = \min\left(|f - f_0|, |f_s - f - f_0|\right)$（$f_0 = 1800$Hz 为噪声中心频率，$f_s$ 为采样率），保证频率轴对称特性；$\sigma = 300$Hz 为高斯函数标准差，决定带阻宽度。

3. 频域处理与信号重构：

   (a) 对每帧加窗信号做快速傅里叶变换（FFT），分离幅值谱 $|X(k)|$ 和相位谱 $\angle X(k)$；

   (b) 幅值谱乘以高斯陷波增益，相位谱保持不变，重构频域信号：$X'(k) = |X(k)| \cdot bandElimination \cdot e^{j\angle X(k)}$；

   (c) 对重构频域信号做逆 FFT（IFFT）转回时域；

   (d) 采用重叠叠加法重构完整信号，对重叠区域幅值加权平均（权重为窗函数叠加值），补偿加窗导致的幅值衰减。

1.2 具体实现过程

算法通过 MATLAB 函数 'noise_suppress'实现，输入为待降噪信号 y 和采样率 fs，输出为降噪后信号 Y 和采样率 Fs，具体步骤如下：

1.2.1 步骤 1：输入校验与预处理

1. 校验输入信号 y 是否为向量，若非向量则抛出错误并提示当前维度；
2. 将信号转换为列向量，记录信号长度 sig_len，将采样率赋值给输出变量 Fs。

1.2.2 步骤 2：分帧参数计算

1. 设定帧时间为 10ms，计算帧长 $frameLength = \text{round}(0.010 \times fs)$；若帧长大于信号长度，则调整为 $\max(\text{floor}(sig\_len/2), 2)$，保证分帧有效性；
2. 设定重叠率 75%，计算重叠点数 $overlapLength = \text{floor}(frameLength \times 0.75)$，帧移 $frameStep = frameLength - overlapLength$；
3. 计算总帧数：$frameNumber = \text{ceil}((sig\_len - frameLength)/frameStep) + 1$，确保所有信号点被覆盖。

1.2.3 步骤 3：信号分帧

1. 初始化帧矩阵 $frames$（维度：帧数 × 帧长）；
2. 遍历每帧，计算帧起始索引 $start\_idx = (i-1) \times frameStep + 1$ 和结束索引 $end\_idx = start\_idx + frameLength - 1$；
3. 若结束索引超出信号长度，对该帧补零；否则直接截取对应信号段，完成帧矩阵填充。

1.2.4 步骤 4：加窗处理

1. 生成汉明窗 $window$，维度与帧长一致；
2. 将帧矩阵与窗函数按列相乘（$windowedFrames = frames \cdot window'$），完成每帧的加窗操作，抑制频谱泄漏。

1.2.5 步骤 5：频域降噪处理

1. 计算频率轴：$freq = (0 : frameLength - 1) \times (fs/frameLength)$；
2. 遍历每帧执行以下操作：
   (a) 对加窗帧做 FFT 得到 $frameFFT$，提取幅值 $magnitude = \text{abs}(frameFFT)$ 和相位 $phase = \text{angle}(frameFFT)$；
   (b) 计算频率与 1800Hz 的距离 $freqDist$，构造高斯陷波增益 $bandElimination$；
   (c) 重构频域信号 $denoisedFrameFFT = magnitude \cdot bandElimination \cdot \exp(1j \times phase)$；
   (d) 对重构频域信号做 IFFT 并取实部，得到降噪后的时域帧 $denoisedFrame$。

1.2.6 步骤 6：窗补偿与信号重构

1. 初始化输出信号 Y 和权重向量 $weight$（均为全零列向量）；
2. 遍历每帧，将降噪帧叠加到 Y 的对应位置，同时将窗函数叠加到 $weight$ 的对应位置；若帧结束索引超出信号长度，仅处理有效长度部分；
3. 对权重非零区域，将 Y 除以权重完成幅值补偿；权重为零区域赋值为 0，避免除零错误，保证输出信号长度与输入完全一致。

1.2.7 步骤 7：滤波器特性可视化

1. 创建图形窗口，绘制高斯陷波滤波器的频率响应曲线（横轴为频率，纵轴为增益）；
2. 设置坐标轴范围（1000 2500Hz），添加网格和边框，保存图片至指定路径后关闭窗口；
3. 实际滤波器如图1

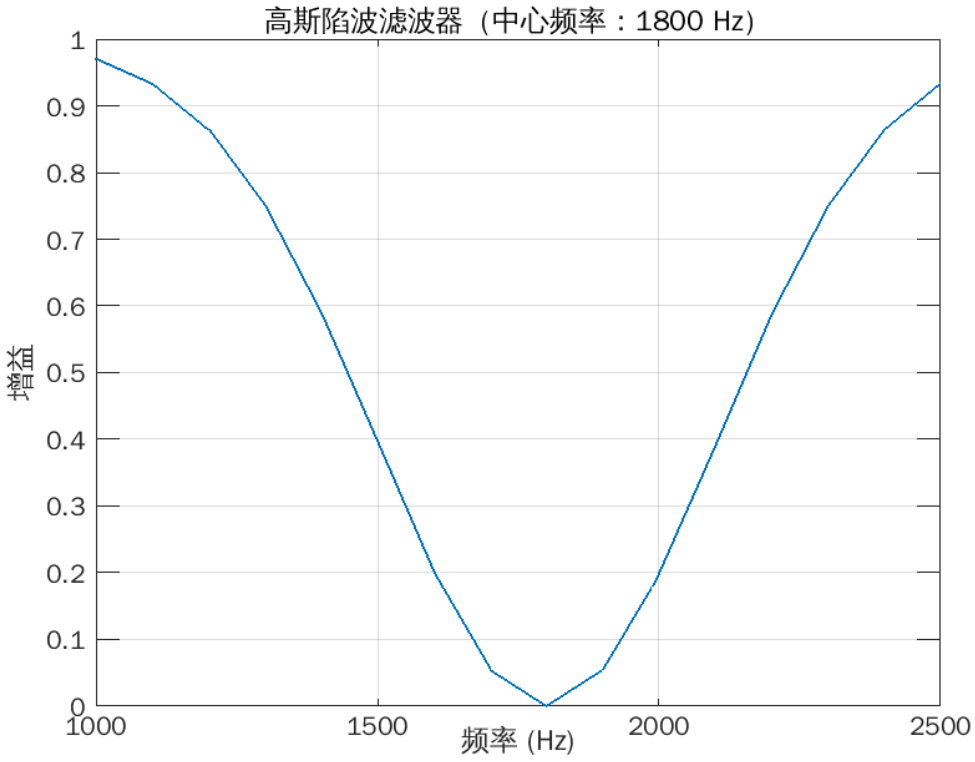

图 1: 高斯陷波滤波器频率响应（中心频率 1800Hz）

1.2.8 算法核心特性

1. 鲁棒性：处理了帧长大于信号长度、帧索引越界、除零等异常情况，避免程序崩溃；
2. 低失真：采用高斯陷波滤波器，过渡带平滑，保持相位不变，降低信号失真；
3. 完整性：重构信号长度与输入完全一致，保证时序连续性。

1.3 程序代码及运行结果

1.3.1 程序代码

代码分为三部分：

1. ‘noise_suppress‘: 包含一个函数，输入音频，返回降噪音频
2. ‘draw_spectrogram‘：包含一个函数，输入音频，绘制语谱图并返回音频语谱图的句柄
3. ‘runNoiseSuppress‘: 控制流，添加噪声，去噪，绘图，保存音频

下面是代码：

Listing 1: noise_suppress

```
function [Y, Fs] = noise_suppress(y, fs)
% 降噪函数：修复信号长度/维度/幅值异常问题
% 输入：y-输入信号（向量），fs-采样率
% 输出：Y-输出信号（列向量，长度与输入完全一致），Fs-采样率


    if ~isvector(y)
        error('输入y必须是向量！当前是矩阵，维度：%s', mat2str(size(y)));
    end
    y = y(:);
    sig_len = length(y);
    Fs = fs;

    frameTime = 0.010; % 帧长 10ms
    frameLength = round(frameTime * fs); % 帧长（点数）
    if frameLength >= sig_len
        frameLength = max(floor(sig_len/2), 2);
    end
    overlapRate = 0.75; % 重叠率
    overlapLength = floor(frameLength * overlapRate); % 重叠点数
    frameStep = frameLength - overlapLength; % 帧移

    frameNumber = ceil((sig_len - frameLength) / frameStep) + 1;


    frames = zeros(frameNumber, frameLength); % 帧矩阵：帧数×帧长
    for i = 1:frameNumber
        start_idx = (i-1)*frameStep + 1;
        end_idx = start_idx + frameLength - 1;

        if end_idx > sig_len
            frame_data = zeros(1, frameLength);
            valid_len = sig_len - start_idx + 1;
            frame_data(1:valid_len) = y(start_idx:end);
            frames(i, :) = frame_data;
        else
            frames(i, :) = y(start_idx:end_idx)';
        end
    end


    window = hamming(frameLength);
    windowedFrames = frames .* window';

    noiseFreq = 1800; % 噪声中心频率
```

```
eliminationRate = 300; % 带阻宽度
denoisedFrames = zeros(size(windowedFrames)); % 降噪帧矩阵

freq = (0:frameLength-1) * (fs / frameLength);

for i = 1:frameNumber

    frameFFT = fft(windowedFrames(i, :));
    magnitude = abs(frameFFT); % 幅值
    phase = angle(frameFFT); % 相位

    freqDist = abs(freq - noiseFreq);
    freqDist = min(freqDist, abs(fs - freq - noiseFreq));
    bandElimination = 1 - exp( -freqDist.^2 / (2 * eliminationRate^2) );
    % 频域降噪
    denoisedFrameFFT = magnitude .* bandElimination .* exp(1j * phase);

    denoisedFrame = real(ifft(denoisedFrameFFT));
    denoisedFrames(i, :) = denoisedFrame;
end


compensationWindow = hamming(frameLength); % 补偿窗

denoisedFrames = denoisedFrames ./ (compensationWindow' + eps) * 0.5;
denoisedFrames(isnan(denoisedFrames) | isinf(denoisedFrames)) = 0;


Y = zeros(sig_len, 1);
weight = zeros(sig_len, 1);
for i = 1:frameNumber
    start_idx = (i-1)*frameStep + 1;
    end_idx = start_idx + frameLength - 1;

    if end_idx > sig_len
        end_idx = sig_len;
        valid_len = end_idx - start_idx + 1;
        Y(start_idx:end_idx) = Y(start_idx:end_idx) + denoisedFrames(i, 1:valid_len)';
        weight(start_idx:end_idx) = weight(start_idx:end_idx) + window(1:valid_len);
    else
        Y(start_idx:end_idx) = Y(start_idx:end_idx) + denoisedFrames(i, :)';
        weight(start_idx:end_idx) = weight(start_idx:end_idx) + window;
    end
end
Y(weight > 0) = Y(weight > 0) ./ weight(weight > 0);
Y(weight == 0) = 0;

figure('Name', '1800Hz高斯陷波滤波器');
plot(freq, bandElimination, 'LineWidth', 1.2);
xlabel('频率 (Hz)'); ylabel('增益');
title(sprintf('高斯陷波滤波器 (中心频率: %d Hz) ', noiseFreq));
xlim([1000, 2500]); grid on; box on;
saveas(gcf, './filter.png');
close(gcf);
```

```
end
```

Listing 2: draw_spectrogram

```
function fig = draw_spectrogram(signal, fs, save_path)

    if ~isa(signal, 'double')
        signal = double(signal);
    end
    if ismatrix(signal)
        signal = signal(:, 1);
    end
    signal = signal(:);

    % 语谱图参数
    window_size = 256;
    overlap = 128;
    nfft = 512;


    [S, F, T] = spectrogram(signal, window_size, overlap, nfft, fs);
    S_abs = abs(S);
    S_abs(S_abs == 0) = eps;
    S_dB = 10*log10(S_abs);


    fig_width = 1200;
    fig_height = 800;
    fig = figure(...
        'Name', 'Spectrogram', ...
        'Position', [100, 100, fig_width, fig_height], ... % [左偏移, 下偏移, 宽, 高]
        'Units', 'inches', ...
        'Color', 'white' ...
    );

      ax = axes(...
        'Parent', fig, ...
        'Position', [0.08, 0.1, 0.75, 0.8], ... % [左, 下, 宽, 高]
        'Units', 'normalized' ...
    );
    imagesc(ax, T, F, S_dB);
    axis(ax, 'xy');
    xlabel(ax, 'Time (s)', 'FontSize', 12);
    ylabel(ax, 'Frequency (Hz)', 'FontSize', 12);
    title(ax, 'Spectrogram (Unified Scale & Width)', 'FontSize', 14);
    set(ax, 'FontSize', 10); %


    cb = colorbar(ax, 'Position', [0.87, 0.1, 0.03, 0.8]);
    cb.Label.String = 'Amplitude (dB)';
    cb.Label.FontSize = 11;
    set(cb, 'FontSize', 10);
```

```
    caxis(ax, [-80 20]);

        if nargin >= 3 && ~isempty(save_path)

        print(fig, save_path, '-dpng', '-r600', '-painters');
        fprintf('语谱图已保存至: %s (600dpi, 尺寸: %d×%d英寸) \n', save_path, fig_width, fig_height);
    end
end
```

Listing 3: runNoiseSuppress

```
clc; close all; clear;
audioPath = '../name.wav'
outputPath = './denoised_name.wav'
outputNoisedPath = './noised_name.wav'

% 噪声参数
noiseAmplitude = 0.02; % 噪声幅度
noiseFrequency = 1800; % 噪声频率 Hz

[y, fs] = audioread(audioPath);

original_fig = draw_spectrogram(y,fs);
saveas(original_fig, 'original_spectrogram.png')

% 添加噪声
noise = noiseAmplitude * sin(2 * pi * noiseFrequency * (0:length(y)-1)' / fs);
noisedSignal = y + noise;
audiowrite(outputNoisedPath, noisedSignal, fs);

% 绘制语谱图分析
% fig = draw_spectrogram(y, fs);
noised_fig = draw_spectrogram(noisedSignal, fs);
% saveas(fig, 'original_spectrogram.png');
saveas(noised_fig, 'noised_spectrogram.png');

% 降噪处理
[Y, Fs] = noise_suppress(noisedSignal, fs);
denoised_fig = draw_spectrogram(Y, Fs);
saveas(denoised_fig, 'denoised_spectrogram.png');
audiowrite(outputPath, Y, Fs);

figure;
stem(y, 'b', 'filled');
hold on;
stem( noisedSignal, 'r');
stem( Y, 'g');
hold off;
disp('Noise suppression processing completed.');
```

1.3.2 运行结果

首先是原音频的语谱图2：

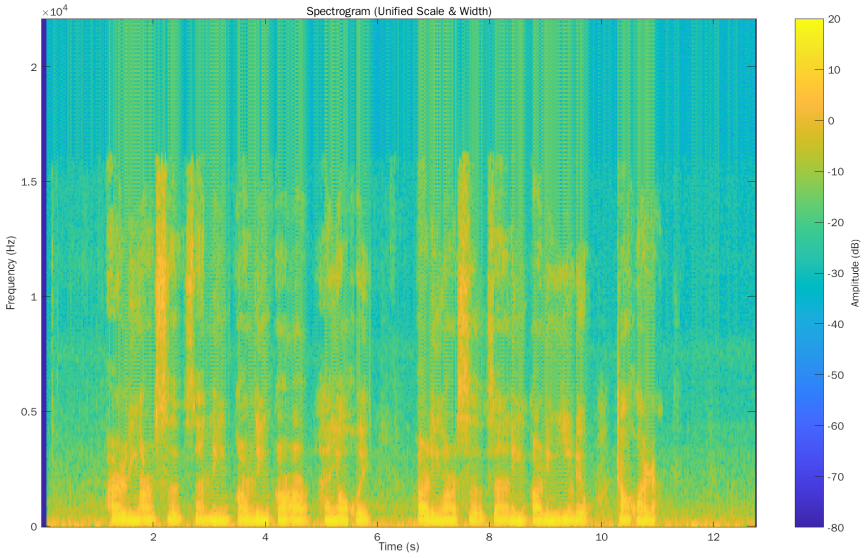

图 2: 原音频语谱图

加入噪声后的音频语谱图如下图3
在纵轴的 1800HZ 处可以明显看到噪声对应的频谱信号。
使用滤波器去噪后的音频语谱图如下 4
经过去噪后，可见噪声已经有明显衰减。
所有处理后的语音都在在文件夹 ‘./matlab/noiseElimination/‘中储存。包括：

- ‘noised_name.wav‘: 加入噪声后的音频
- ‘denoised_name.wav‘: 去除噪声后的音频

2 女声转男声音频处理算法设计与实现

2.1 设计思路

本算法实现音频的变调不变速，以男女声的音频的转换为例。

2.1.1 背景知识

1. 人声音高与音色的核心特征：基频（Fundamental Frequency）决定音高，女声基频（200 500Hz）显著高于男声（85 180Hz），调整基频是女声转男声的核心手段；共振峰（Formant）决定音色，是否保留共振峰直接影响变声后音色的自然度。
2. 短时傅里叶变换（STFT）的窗长适配：STFT 窗长决定频率/时间分辨率，窗长过小（如 512 点）会导致频率分辨率不足，信号处理失真；1024 点窗长是人声处理的经典取值，可平衡分辨率与处理效率。
3. 高频增强的合理方式：人声清晰感依赖 1000 5000Hz 高频成分，硬高通滤波会滤除 0 3kHz 人声核心频段，导致声音丢失；“带通提取 + 温和叠加”的方式仅增强高频，不破坏核心频段，是人声增强的通用策略。
4. 音频幅值归一化的必要性：音频处理后易出现幅值异常（过小/过大），将幅值归一化到 [-1,1] 可避免削波失真，增益调整（如 0.9）可保证最终音量适中；同时需检测处理后信号丢失等异常，降级使用原始音频避免程序报错。

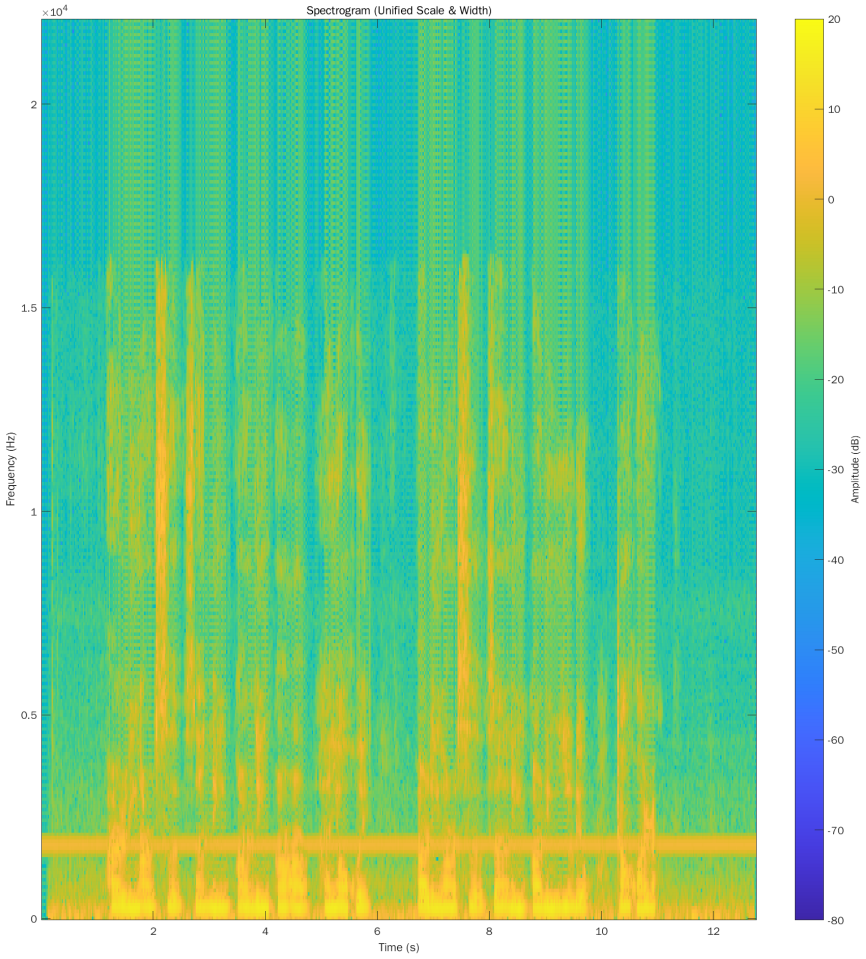

图 3: 加入噪声后音频语谱图

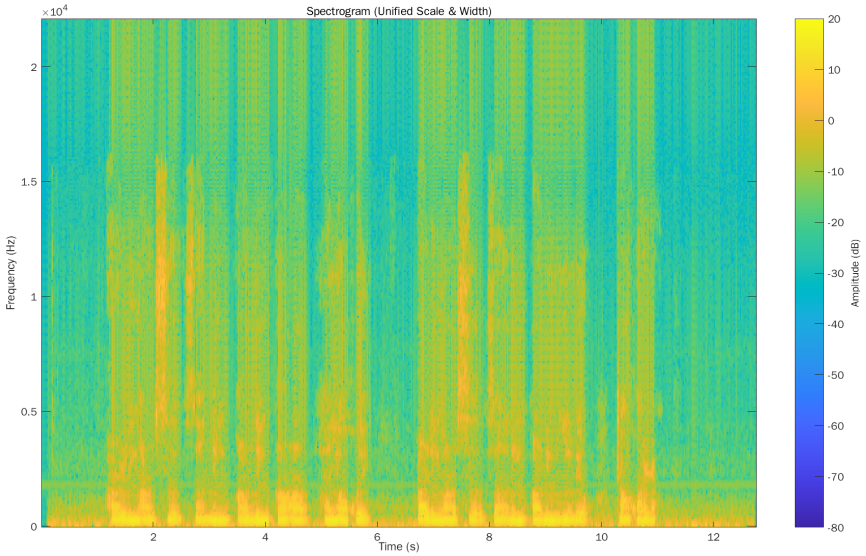

图 4: 去除噪声后的语谱图

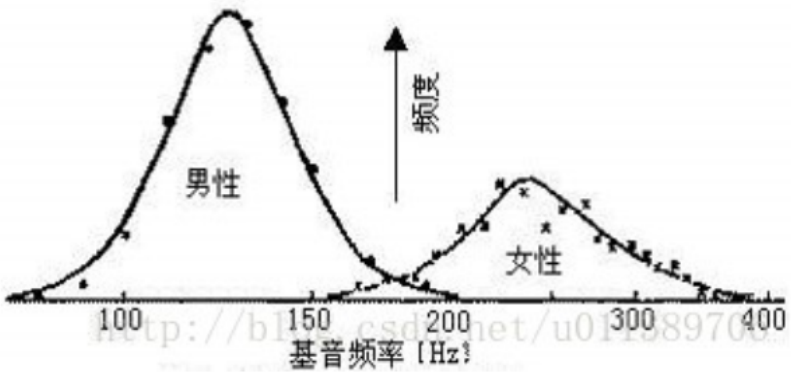

图 5: 不同性别音色频谱图

5. 频谱分析的验证价值：通过 FFT 计算单侧边频谱是验证音频处理效果的通用手段，可直观对比处理前后核心频段（如 0 3kHz）的幅值分布，确认核心信号未丢失。

2.1.2 技术原理

1. 基频调整原理：基于 STFT 的音高偏移算法（shiftPitch 函数），通过调整每帧信号的基频实现音高降低（nsemitones 为半音数，负值表示降频）；设置汉明窗、75

2. 温和高频增强原理：替代硬高通滤波，采用 2 阶巴特沃斯带通滤波器提取 1000 5000Hz 高频成分，将其以小幅增益叠加回原信号，仅增强高频清晰度，不滤除 0 3kHz 人声核心频段。

3. 频谱分析原理：通过快速傅里叶变换（FFT）计算音频单侧边频谱，公式如下：

$$
\begin{aligned}
P_2 &= |\mathrm{FFT}(audio)|/L, \quad (L \quad) \\
P_1 &= P_2(1 : L/2+1), \quad \text{(单侧边频谱)} \\
P_1(2 : \mathrm{end}-1) &= 2 \cdot P_1(2 : \mathrm{end}-1), \quad \text{(幅值补偿)} \\
f_p &= f_s \cdot (0 : (L/2))/L, \quad \text{(频率轴)}
\end{aligned}
$$

   其中 P_1 为单侧边幅值谱，f_p 为频率轴，通过对比原始/处理后频谱验证核心频段信号存在。

4. 音量归一化原理：将处理后信号幅值归一化到 [-1,1] 区间，再通过增益系数调整音量，同时增加防除零处理，避免信号丢失时程序报错。

2.2 具体实现过程

算法通过 MATLAB 脚本实现女声转男声，核心解决“声音消失、音色闷/憨”问题，具体步骤如下：

2.2.1 步骤 1：环境初始化与音频读取

1. 执行环境清理：‘clear; clc; close all;‘ 清空变量、清除命令行、关闭所有图形窗口；

2. 读取音频文件：调用 ‘audioread‘读取 ‘../name.wav‘，返回音频数据 ‘audio_In‘和采样率 ‘fs‘；

3. 立体声转单声道：若音频为立体声（列数 >1），通过 ‘mean(audio_In, 2)‘取列均值转为单声道；

4. 音频有效性校验：若音频最大幅值小于 1e-6，抛出错误提示“原始音频文件无信号”。

2.2.2 步骤 2：核心调整参数设置

1. 基频调整参数：‘nsemitones = -9.5‘（降 9.5 个半音，平衡男声感和声音保留）；
2. 共振峰参数：‘preserve_formants = false‘（不保留共振峰，避免音色闷/憨）；
3. 音量参数：‘gain = 0.9‘（提升音量，避免声音过小）；
4. 高频增强参数：‘high_freq_boost = 1.05‘（温和高频增益，仅小幅增强）。

2.2.3 步骤 3：基频调整（核心变声）

1. 设置窗长：‘win_len = 1024‘（恢复合理窗长，避免 512 窗长导致的信号失真）；
2. 调用音高偏移函数：‘audio_Out = shiftPitch(audio_In, nsemitones, ...)‘，关键参数：
   (a) ‘’PreserveFormants’‘：设为 ‘false‘，不保留共振峰，避免音色闷/憨；
   (b) ‘’Window’‘：设为 ‘hamming(win_len)‘，汉明窗减少频谱泄漏；
   (c) ‘’OverlapLength’‘：设为 ‘round(win_len * 0.75)‘，75
   (d) ‘’LockPhase’‘：设为 ‘true‘，锁相避免相位失真导致的破音。

2.2.4 步骤 4：温和高频增强（修复声音消失）

1. 设计带通滤波器：调用 ‘butter(2, [1000 5000]/(fs/2), ’bandpass’)‘设计 2 阶巴特沃斯带通滤波器，通带 1000 5000Hz；
2. 提取高频成分：调用 ‘filtfilt(b, a, audio_Out)‘对变声后音频做零相位滤波，提取高频成分 ‘audio_high‘；
3. 温和叠加高频：‘audio_Out = audio_Out + (audio_high * (high_freq_boost - 1))‘，仅将高频成分以 0.05 倍增益叠加，不滤除核心频段。

2.2.5 步骤 5：音量归一化与异常处理

1. 信号丢失检测：若 ‘audio_Out‘最大幅值小于 1e-6，弹出警告并将 ‘audio_Out‘重置为原始音频 ‘audio_In‘；
2. 幅值归一化：‘audio_Out = audio_Out / max(abs(audio_Out))‘，将幅值归一化到 [-1,1]；
3. 音量提升：‘audio_Out = audio_Out * gain‘，将归一化后的音频乘以增益系数 0.9，提升音量。

2.2.6 步骤 6：频谱对比绘制（验证信号）

1. 创建图形窗口：‘figure(’Color’,’w’)‘设置白色背景，开启 ‘hold on‘和网格；
2. 绘制频谱：调用自定义子函数 ‘plotSpectrum‘分别绘制原始音频（蓝色）和处理后音频（红色）的单侧边频谱；
3. 设置坐标轴：聚焦人声核心频段 ‘xlim([0 3000])‘，设置纵轴范围 ‘ylim([0 0.03])‘，添加标题、坐标轴标签和图例；

2.2.7 步骤 7：音频保存与播放

1. 保存音频：调用 ‘audiowrite(’to_male.wav’, audio_Out, fs)‘将处理后音频保存为 ‘to_male.wav‘；
2. 输出提示：‘disp(’ 音频已保存为 to_male.wav’)‘，提示保存完成。

2.2.8 子函数：plotSpectrum（绘制单侧边频谱）

1. 输入参数：‘audio‘（音频数据）、‘fs‘（采样率）、‘label‘（图例标签）、‘color‘（线条颜色）；
2. 计算音频长度：‘L = length(audio)‘；
3. 计算 FFT 并转换为单侧边频谱：按 1.2.2 节的频谱分析公式计算 P_1 和 f_p；
4. 绘制频谱：‘plot(fp, P1, color, 'LineWidth', 1.2, 'DisplayName', label)‘，设置线宽和图例。

2.2.9 算法核心特性

1. 问题修复：替换硬高通滤波为温和高频增强，避免核心频段丢失导致的声音消失；优化窗长和共振峰参数，避免音色闷/憨；
2. 鲁棒性：增加音频有效性校验、信号丢失异常处理，避免程序报错；
3. 可验证性：绘制频谱对比图，直观验证处理后核心频段信号存在；
4. 实用性：输出可播放的音频文件，参数可灵活调整以平衡“男声感”和“声音清晰度”。

2.3 程序代码及运行结果

2.3.1 程序代码

程序源码如下：

Listing 4: genderConversion

```
clear; clc; close all;
%% 1. 读取音频（兼容立体声/单声道）
[audioIn, fs] = audioread('../name.wav');
if size(audioIn, 2) > 1
    audioIn = mean(audioIn, 2); % 转为单声道
end
% 检查原始音频是否有效
if max(abs(audioIn)) < 1e-6
    error('原始音频文件无信号，请检查name.wav是否存在且有效！');
end

%% 2. 核心调整参数（平衡"男声感"和"不憨+有声音"）
nsemitones = -9.5; % 基频：-8（适度降频，保留声音）
preserve_formants = false; % 保留共振峰（核心：避免闷/憨，且不丢声音）
gain = 0.9; % 提高音量（避免声音小）
high_freq_boost = 1.05; % 温和高频增强（不滤掉核心频段）

%% 3. 核心：调整基频（恢复合理窗长，避免信号丢失）
win_len = 1024; % 恢复1024窗长（512太小易导致信号处理失真）
audioOut = shiftPitch(audioIn, nsemitones, ...
    'PreserveFormants', preserve_formants, ...
    'Window', hamming(win_len), ...
    'OverlapLength', round(win_len * 0.75), ...
    'LockPhase', true);

%% 4. 温和高频增强（替代硬高通，不丢声音）
% 问题根源：2kHz高通滤掉了人声核心频段→声音消失！
% 修正：用"低通+高频增益"，仅增强高频，不滤掉低频/中频
```

```
% 设计1000Hz~5000Hz的带通增强（人声清晰频段）
[b, a] = butter(2, [1000 5000]/(fs/2), 'bandpass'); % 2阶带通
audio_high = filtfilt(b, a, audioOut); % 提取高频
audioOut = audioOut + (audio_high * (high_freq_boost - 1)); % 温和增强高频

%% 5. 音量归一化（避免失真+确保声音大小）
% 增加防除零：若信号全零，直接用原始音频（避免报错）
if max(abs(audioOut)) < 1e-6
    warning('处理后信号丢失，使用原始音频！');
    audioOut = audioIn;
end
audioOut = audioOut / max(abs(audioOut)); % 归一化到[-1,1]
audioOut = audioOut * gain; % 提升音量

%% 6. 绘制频谱对比（验证信号存在）
figure('Color','w'); hold on; grid on;
plotSpectrum(audioIn, fs, '原始女声', 'b');
plotSpectrum(audioOut, fs, '调整后男声（有声音+不憨）', 'r');

title('音频频谱对比（修复声音消失问题）');
xlabel('频率 (Hz)'); ylabel('幅值');
xlim([0 3000]); % 聚焦人声核心频段（0-3kHz）
ylim([0 0.03]);
legend('Location','best');
hold off;

%% 7. 保存+播放音频
audiowrite('to_male.wav', audioOut, fs);
disp('音频已保存为 to_male.wav');


%% 子函数：绘制单侧边频谱
function plotSpectrum(audio, fs, label, color)
    L = length(audio);
    ffta = fft(audio);
    P2 = abs(ffta/L);
    P1 = P2(1:L/2+1);
    P1(2:end-1) = 2*P1(2:end-1);
    fp = fs*(0:(L/2))/L;
    plot(fp, P1, color, 'LineWidth', 1.2, 'DisplayName', label);
end
```

2.3.2 运行结果

音频变调后结果如下图6：

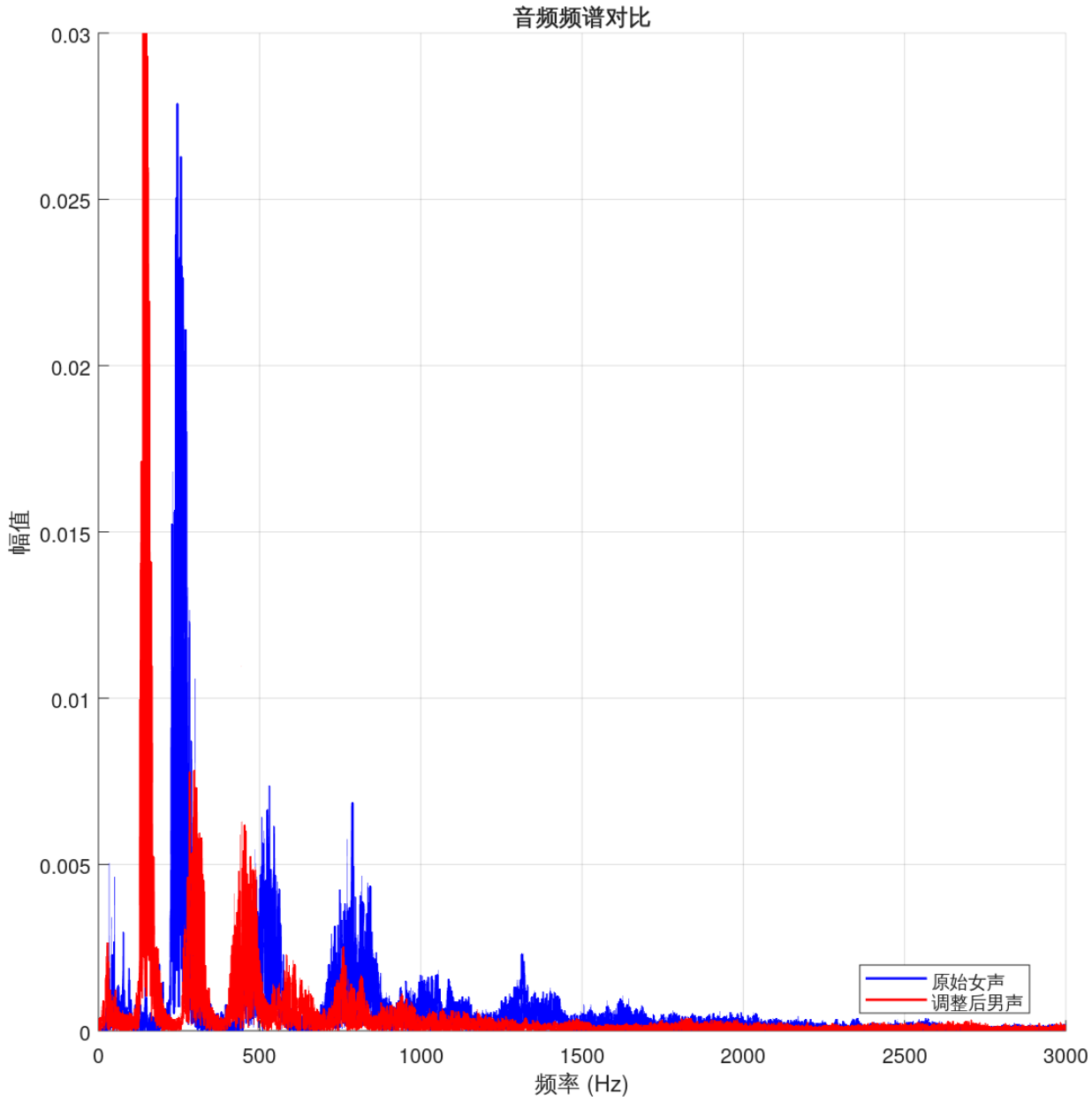

图 6: 女声转男声音频频谱对比（0 3kHz 核心频段）

可见变调后频谱整体左移，低频分量增加，高频分量减小。
输出音频保存在 ‘./matlab/genderConverse/to_male.wav‘

3 音频变速算法设计与实现（基于分帧重叠相加法）

3.1 设计思路

本算法基于时域分帧重叠相加原理实现音频变速（速率 >1 时加速），核心逻辑是通过调整输出帧的重叠长度改变帧移，在保持音频音调不变的前提下调整播放时长，避免频域处理的复杂度与失真问题。

3.1.1 背景知识

3.1.2 背景知识

1. 音频变速的时域实现逻辑：变速的核心是改变帧移（帧间步进长度），而非修改采样率（采样率不变可保证音调不偏移）；加速时增大帧移，减速时减小帧移，是时域变速的通用思路。

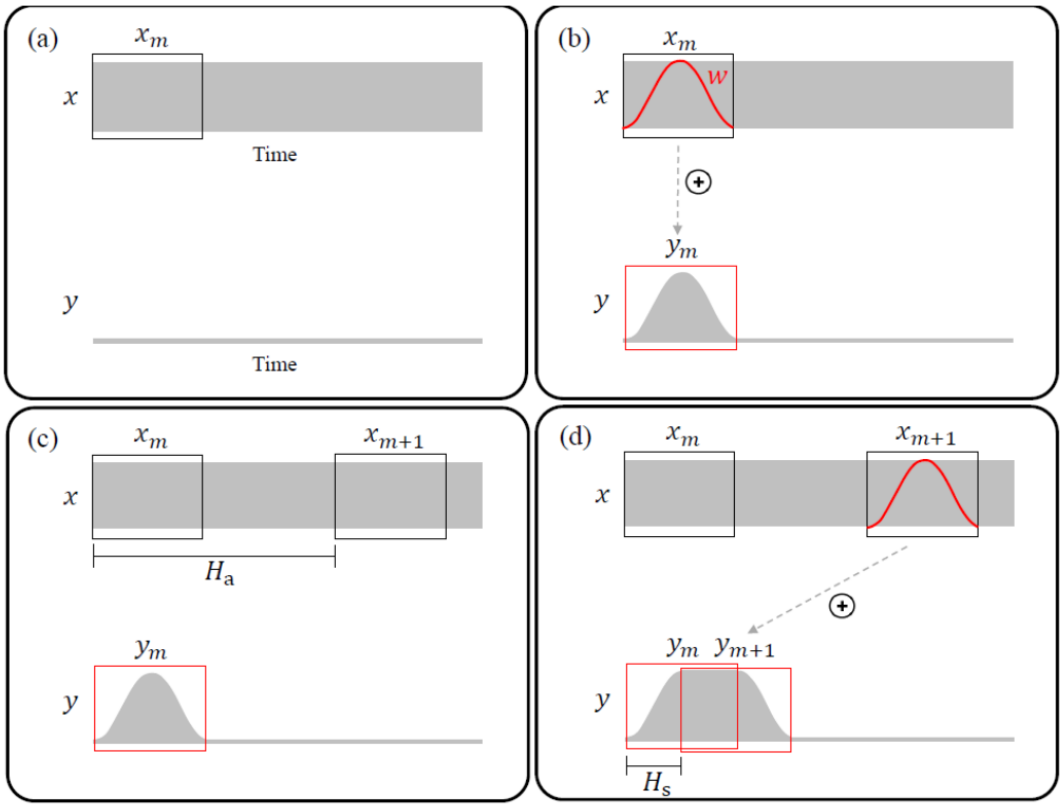

图 7: 分帧与复原

2. 分帧参数的通用计算：帧长通常由固定时间窗（如 10ms）与采样率换算（$frame\_Length = frame\_Time \times fs$），重叠率（25% 75%）需根据场景适配——低重叠率（如 25%）提升计算效率，高重叠率提升信号连续性。

3. 重叠相加法的重构原理：输出信号长度由帧长、帧数、输出重叠长度共同决定，每帧按输出帧移叠加到对应位置，重叠区域幅值累加可保证时域连续性，避免拼接毛刺。

4. 窗函数的通用作用：汉明窗是音频分帧处理的标配窗函数，可抑制频谱泄漏，同时使帧间叠加更平滑；窗函数需与帧矩阵维度匹配（行/列向量对齐），避免维度错误。

5. 变速算法的兼容性：时域重叠相加法无需频域变换，计算复杂度低，可兼容加速（rate>1）与减速（rate<1）场景，仅需调整输出重叠率即可适配不同变速比率。

3.1.3 技术原理

1. 分帧参数计算原理：

   (a) 输入帧长（点数）：$frame\_Length = frame\_Time \times fs$（frame_Time=0.010s 为固定帧时间，fs 为采样率）；

   (b) 输入重叠长度：$overlapLength = \text{floor}(frame\_Length \times overlapRate)$（overlapRate=0.25 为输入重叠率）；

   (c) 输入帧数：$frameNumber = \text{floor}((\text{length}(y) - frame\_Length)/(frame\_Length - overlapLength)) + 1$，确保所有有效输入信号被分帧覆盖；

   (d) 输出重叠率自适应：$outputOverlapRate = \frac{rate + overlapRate - 1}{rate}$，rate>1 时输出重叠率降低，输出帧移增大，总时长缩短（实现加速）；

   (e) 输出重叠长度：$outputOverlapLength = \text{floor}(frame\_Length \times outputOverlapRate)$。

2. 加窗原理：采用汉明窗抑制频谱泄漏，汉明窗公式为：

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \quad n = 0, 1, ..., N-1$$

其中 $N = frame\_Length$ 为帧长，窗函数与帧矩阵按元素相乘实现加窗。

3. 重叠相加重构原理：输出信号长度为 $frame\_Length+(frameNumber-1)\times(frame\_Length-outputOverlapLength)$，每帧信号按输出帧移（$frame\_Length - outputOverlapLength$）叠加到对应位置，重叠区域幅值累加，保证信号连续。

4. 变速核心逻辑：rate>1 时，输出帧移（$frame\_Length-outputOverlapLength$）大于输入帧移（$frame\_Length-overlapLength$），相同帧数的信号在更短的时长内完成叠加，实现音频加速；采样率保持不变（Fs=fs），保证音调不发生偏移。

3.2 具体实现过程

算法通过 MATLAB 函数 ‘speed_change‘实现音频变速，具体步骤如下：

3.2.1 步骤 1：基本参数计算与初始化

1. 设定固定帧时间：$frame\_Time = 0.010$（单位：秒，对应 10ms 帧长）；

2. 计算输入帧长（点数）：$frame\_Length = frame\_Time \times fs$；

3. 设定输入重叠率：$overlapRate = 0.25$，计算输入重叠长度：$overlapLength = \text{floor}(frame\_Length \times overlapRate)$；

4. 计算输入总帧数：$frameNumber = \text{floor}((\text{length}(y)-frame\_Length)/(frame\_Length-overlapLength))+1$；

5. 计算输出重叠率：$outputOverlapRate = (rate+overlapRate-1)/rate$，并推导输出重叠长度：$outputOverlapLength = \text{floor}(frame\_Length \times outputOverlapRate)$。

3.2.2 步骤 2：输入信号分帧

1. 初始化帧矩阵 $frames$（维度：$frameNumber \times frame\_Length$），初始值全为 0；

2. 遍历每帧（$i = 1 : frameNumber$），计算当前帧的信号索引：起始索引：$(i-1) \times (frame\_Length - overlapLength) + 1$，结束索引：$(i-1) \times (frame\_Length - overlapLength) + frame\_Length$；

3. 将输入信号 y 的对应索引段赋值给帧矩阵 $frames(i,:)$，完成分帧。

3.2.3 步骤 3：帧加窗处理

1. 生成汉明窗 $window$：$window = \text{hamming}(frame\_Length)'$（转置为行向量，匹配帧矩阵维度）；

2. 帧矩阵与窗函数按元素相乘：$windowedFrames = frames.*window$，完成每帧的加窗处理，抑制频谱泄漏。

3.2.4 步骤 4：重叠相加重构输出信号

1. 初始化输出信号 Y：维度为 $1\times(frame\_Length+(frameNumber-1)\times(frame\_Length-outputOverlapLength))$，初始值全为 0；

2. 遍历每帧（$i = 1 : frameNumber$），计算当前帧在输出信号中的起始索引：$startIndex = (i-1) \times (frame\_Length - outputOverlapLength) + 1$；

3. 将加窗后的帧 $windowedFrames(i,:)$ 叠加到 Y 的 $startIndex:startIndex+frame\_Length-1$ 区间，重叠区域幅值累加；

4. 遍历完成后，Y 即为重构后的变速音频信号。

3.2.5 步骤 5：采样率赋值

1. 将输出采样率 Fs 赋值为输入采样率 fs，保证音频音调不随速度变化。

3.2.6 算法核心特性

1. 音调保持：采样率全程不变（Fs=fs），仅通过调整帧重叠长度改变播放时长，保证变速后音调不偏移；

2. 低复杂度：基于时域分帧重叠相加实现，无需频域变换，计算效率高；

3. 低失真：汉明窗抑制频谱泄漏，重叠相加保证帧间连续性，避免信号断裂/毛刺；

4. 灵活性：变速比率 $rate$ 可灵活调整，rate>1 实现加速，rate<1 可实现减速（算法兼容）；

5. 鲁棒性：帧数计算基于整数取整，确保分帧覆盖有效信号，无索引越界风险。

3.3 程序代码和运行结果

3.3.1 程序代码

程序代码包含两部分：

1. ‘speed_change‘：包含一个函数，输入一个音频和一个标量（加速倍率）返回加减速后的音频

2. ‘runSpeedChange‘：控制流，控制读入音频，加减速音频和保存结果

Listing 5: speed_change

```
function [Y, Fs] = speed_change(y, fs, rate)
% Y 输出信号，向量形式
% Fs 输出采样率
% y 输入信号，向量形式
% fs 输入采样率
% rate 变数比率， >1 加速

% 基本参数设置

    frameTime = 0.010; % 帧长 ms
    frameLength = frameTime * fs; % 帧长 点数
    overlapRate = 0.25; % 重叠率
    overlapLength = floor(frameLength * overlapRate); % 重叠点数
    frameNumber = floor((length(y)- frameLength) / (frameLength - overlapLength)) + 1; % 帧数
    outputOverlapRate = (rate + overlapRate - 1) / rate; % 输出重叠率
    outputOverlapLength = floor(frameLength * outputOverlapRate); % 输出重叠点数

    % 分帧
    frames = zeros(frameNumber, frameLength);
    for i = 1:frameNumber
        frames(i, :) = y((i-1)*(frameLength - overlapLength) + 1 : (i-1)*(frameLength - overlapLength) +
            frameLength);
    end
```

```
    % 加窗
    window = hamming(frameLength)';
    windowedFrames = frames .* window;

    % 重叠相加
    Y = zeros(1, frameLength+ (frameNumber -1)*(frameLength - outputOverlapLength));
    for i = 1:frameNumber
        startIndex = (i-1)*(frameLength - outputOverlapLength) + 1;
        Y(startIndex : startIndex + frameLength -1) = Y(startIndex : startIndex + frameLength -1) +
            windowedFrames(i, :);
    end


    Fs = fs;

end
```

Listing 6: runSpeedChange

```
% 基本信息
audiopath = '../name.wav';
outputpathFast = './fastname.wav'
outputpathSlow = './slowname.wav'

[y,Fs] = audioread(audiopath);
[Yf, Fsf] = speed_change(y, Fs, 2);
audiowrite(outputpathFast, Yf, Fsf);

[Ys, Fss] = speed_change(y, Fs, 0.5);
audiowrite(outputpathSlow, Ys, Fss);

%输出基本信息
disp('Speed change processing completed.');
```

3.3.2 代码运行结果

输出音频存储在 ‘./matlab/changeSpeed/‘ 包含

- ‘slowname.wav‘: 减速后的音频
- ‘fastname.wav‘: 加速后的音频

4 基于能量阈值的语音段自动分割与反转拼接算法实现

4.1 设计思路

本算法针对数字语音音频（110 段）实现“自动分割-段起始前移-反转拼接”全流程处理，核心逻辑是通过短时能量分析识别语音段，自适应调整能量阈值匹配目标分割段数，再对语音段进行前移修正和反转拼接，最终输出反转后的音频并可视化验证。

4.1.1 背景知识

4.1.2 背景知识

1. 语音端点检测的核心依据：语音段的短时能量远高于静音段，滑动窗口计算短时能量（均方根能量）是端点检测的经典方案，窗口时长（20ms）是语音处理的通用取值，可精准区分语音/静音。

2. 阈值自适应的鲁棒性设计：固定能量阈值易受音频信噪比、说话人音量影响，导致分割结果偏离目标；通过小幅迭代调整阈值（如 ±10%），并限制最大调整次数（如 3 次），是平衡效果与收敛性的通用策略。

3. 语音段过滤的通用规则：需设置最小段长（如 100ms）过滤误分割的短段，避免无效段干扰；段起始前移（如 0.25 秒）可补偿能量上升延迟，避免丢失语音段开头的有效信息。

4. 音频拼接的时序完整性：拼接时需保留原始静音间隔，保证反转后音频的时序逻辑与原音频一致；静音间隔包括段间间隔与最后一段后的间隔，是拼接类算法的核心考量。

5. 可视化验证的通用价值：绘制原始/处理后音频波形，标记关键特征点（如段起始），可直观验证分割与拼接效果；设置中文显示适配（如 SimHei 字体）是可视化的基础要求。

4.1.3 技术原理

1. 短时能量计算原理：对音频每个采样点，取以其为中心的 20ms 滑动窗口，计算窗口内音频的均方根能量，公式为：

$$E(i) = \sqrt{\frac{1}{N} \sum_{n=start\_idx}^{end\_idx} x(n)^2}$$

其中 N 为窗口内采样点数，$start\_idx = \max(1, i - \text{half\_win})$，$end\_idx = \min(\text{total\_samples}, i + \text{half\_win})$，$x(n)$ 为音频采样值；

2. 能量阈值分割：设定能量阈值 energy_thresh，当 $E(i) >$ energy_thresh 时标记为语音段起始，当 $E(i) \leq$ energy_thresh 时标记为语音段结束，同时过滤长度小于 min_seg_len 的短段（避免误分割）；

3. 阈值自适应调整：若分割段数少于目标值，将阈值降低 10%；若多于目标值，将阈值升高 10%，最多调整 3 次保证算法收敛；

4. 段起始前移：将每个语音段的起始索引前移 shift_samples = fs × shift_time 个采样点（fs 为采样率，shift_time 为前移时间），且保证前移后索引 1；

5. 反转拼接：先提取各语音段间的静音间隔，将语音段和间隔分别反转，再按 “反转语音段 + 反转静音间隔” 的顺序拼接，保证反转后音频的静音间隔与原音频一致。

4.2 具体实现过程

算法通过 MATLAB 脚本实现语音段分割与反转拼接，输入为数字语音音频文件，输出为反转拼接后的音频文件及可视化结果，具体步骤如下：

4.2.1 步骤 1：环境初始化与参数配置

1. 环境清理：执行 ‘clear; clc; close all;‘清空变量、清除命令行、关闭所有图形窗口；

2. 配置核心参数：构建结构体 ‘cfg‘，包含以下参数：

   (a) ‘cfg.audio_path = ’../number.wav‘‘：输入音频文件路径；

   (b) ‘cfg.shift_time = 0.25‘：段起始前移时间（秒）；

   (c) ‘cfg.energy_thresh = 0.05‘：初始能量阈值；

(d) ‘cfg.win_size_ms = 20‘：滑动窗口时长（毫秒）；

(e) ‘cfg.min_seg_len_ms = 100‘：最小语音段长度（毫秒）；

(f) ‘cfg.target_seg_num = 10‘：目标分割段数（匹配 1 10 数字语音）。

4.2.2 步骤 2：音频读取与预处理

1. 读取音频：调用 ‘audioread(cfg.audio_path)‘返回音频数据 ‘audio_In‘和采样率 ‘fs‘；
2. 立体声转单声道：‘audio_In = mean(audio_In, 2)‘，保证音频为列向量；
3. 幅值归一化：‘audio_In = audio_In / max(abs(audio_In))‘，将幅值归一化到 [-1,1]；
4. 计算音频基本信息：总采样数 ‘total_samples = length(audio_In)‘，总时长 ‘total_duration = total_samples / fs‘，并打印音频信息。

4.2.3 步骤 3：短时能量计算

1. 窗口参数计算：滑动窗口采样数 ‘win_size = round(fs * cfg.win_size_ms / 1000)‘，窗口半长 ‘half_win = floor(win_size / 2)‘；
2. 初始化能量数组：‘audio_energy = zeros(total_samples, 1)‘，存储每个采样点的短时能量；
3. 遍历每个采样点计算短时能量：
   (a) 计算窗口起始索引：‘start_idx = max(1, i - half_win)‘；
   (b) 计算窗口结束索引：‘end_idx = min(total_samples, i + half_win)‘；
   (c) 截取窗口内音频：‘window_audio = audio_In(start_idx:end_idx)‘；
   (d) 计算均方根能量：‘audio_energy(i) = sqrt(mean(window_audio.2))‘。

4.2.4 步骤 4：语音段自动分割

1. 最小段长转换：‘min_seg_len = round(fs * cfg.min_seg_len_ms / 1000)‘，将毫秒转换为采样点数；
2. 初始分割语音段：
   (a) 初始化语音段矩阵 ‘segments‘（存储起始/结束索引）、起始标记 ‘start_idx = 0‘；
   (b) 遍历采样点，当能量大于阈值且无起始标记时，标记 ‘start_idx = i‘；
   (c) 当能量小于等于阈值且有起始标记时，标记结束索引 ‘end_idx = i‘，若段长大于最小段长则存入 ‘segments‘；
   (d) 遍历结束后，若仍有未结束的语音段且长度达标，补充存入 ‘segments‘。
3. 自适应调整阈值：
   (a) 初始化调整次数 ‘adjust_count = 0‘，最大调整次数 ‘max_adjust_times = 3‘；
   (b) 若分割段数 目标段数且未达最大调整次数：- 段数不足：阈值 ×0.9，扩大识别范围；- 段数过多：阈值 ×1.1，缩小识别范围；
   (c) 重新分割语音段，更新段数 ‘num_seg = size(segments, 1)‘；
   (d) 调整结束后，若段数仍不匹配，输出警告及调整建议；匹配则打印成功提示。

4.2.5 步骤 5：语音段起始前移修正

1. 计算前移采样数：‘shift_samples = round(fs * cfg.shift_time)‘；
2. 初始化调整后段矩阵 ‘segments_adjusted = segments‘；
3. 遍历每个语音段，将起始索引前移：‘segments_adjusted(i, 1) = max(1, segments_adjusted(i, 1) - shift_samples)‘，保证索引不越界。

4.2.6 步骤 6：提取音频间隔（静音区）

1. 初始化间隔数组 ‘intervals_all‘；
2. 提取段间间隔：遍历 1 num_seg-1 段，计算 ‘interval_between = segments_adjusted(i+1, 1) - segments_adjusted(i, 2)‘，存入 ‘intervals_all‘；
3. 提取最后一段后的间隔：‘interval_after_last = total_samples - segments_adjusted(end, 2)‘，补充存入 ‘intervals_all‘。

4.2.7 步骤 7：反转拼接音频

1. 反转语音段和间隔：‘reversed_segments = flipud(segments_adjusted)‘，‘reversed_intervals = flipud(intervals_all)‘；
2. 初始化输出音频 ‘audioOut = []‘；
3. 遍历反转后的语音段：
   (a) 截取当前段音频：‘seg_audio = audio_In(reversed_segments(i, 1):reversed_segments(i, 2))‘；
   (b) 拼接语音段：‘audioOut = [audioOut; seg_audio]‘；
   (c) 生成对应静音间隔：‘silence = zeros(reversed_intervals(i), 1)‘；
   (d) 拼接静音间隔：‘audioOut = [audioOut; silence]‘。
4. 音量归一化：‘audioOut = audioOut / max(abs(audioOut)) * 0.9‘，避免失真且保证音量适中。

4.2.8 步骤 8：音频保存与可视化

1. 保存音频：调用 ‘audiowrite(’numberConverse.wav’, audioOut, fs)‘，输出保存提示；
2. 创建可视化窗口：‘figure(’Color’,’w’,’Position’,[100 100 1000 700],’Name’,’ 音频反转结果可视化’)‘；
3. 绘制原始音频波形：
   (a) 子图 1（2,1,1）：绘制原始音频时域波形，标记原段起始（红色虚线）和前移后段起始（绿色实线），标注段序号；
   (b) 设置坐标轴、网格、图例，保证中文显示正常。
4. 绘制反转后音频波形：
   (a) 子图 2（2,1,2）：绘制反转后音频时域波形，标记反转后段起始（蓝色虚线），标注反转后段序号；
   (b) 设置坐标轴、网格，保证中文显示正常。
5. 保存可视化图片：‘saveas(gcf, ’numberConverse_visualization.png’)‘；

4.2.9 算法核心特性

1. 自适应分割：通过能量阈值微调，自动匹配目标分割段数，降低人工调参成本；
2. 鲁棒性强：段起始前移避免丢失语音开头信息，最小段长过滤误分割短段，阈值调整次数限制避免无限循环；
3. 可视化验证：绘制原始/反转音频波形并标记语音段，直观验证分割和拼接效果；
4. 低失真：全程仅做幅值归一化，无频域处理，保证语音音色完整性；
5. 可扩展性：参数可灵活调整（如前移时间、窗口时长、目标段数），适配不同语音场景。

4.3 程序代码及运行结果

4.3.1 程序代码

Listing 7: 信号波形生成与绘图代码

```
clear; clc; close all;

%% 基础参数
cfg = struct();
cfg.audio_path = '../number.wav'; % 输入音频文件
cfg.shift_time = 0.25; % 段起始前移时间（秒）
cfg.energy_thresh = 0.05; % 能量阈值，控制语音段识别
cfg.win_size_ms = 20; % 滑动窗口时长（毫秒）
cfg.min_seg_len_ms = 100; % 最小语音段长度（毫秒）
cfg.target_seg_num = 10; % 目标分割段数

[audioIn, fs] = audioread(cfg.audio_path);
audioIn = mean(audioIn, 2);
audioIn = audioIn / max(abs(audioIn));
total_samples = length(audioIn);
total_duration = total_samples / fs;

fprintf(' 音频信息: \n');
fprintf(' 采样率: %d Hz | 总时长: %.2f 秒 | 总样本数: %d\n', fs, total_duration, total_samples);
fprintf('_____________________________________________________________\n');

win_size = round(fs * cfg.win_size_ms / 1000);
half_win = floor(win_size / 2);
audio_energy = zeros(total_samples, 1);

for i = 1:total_samples
    start_idx = max(1, i - half_win);
    end_idx = min(total_samples, i + half_win);
    window_audio = audioIn(start_idx:end_idx);
    audio_energy(i) = sqrt(mean(window_audio.^2));
end

%% 语音段自动分割
min_seg_len = round(fs * cfg.min_seg_len_ms / 1000);
segments = []; % 语音段存储矩阵
start_idx = 0; % 语音段起始标记
```

```
for i = 1:total_samples
    if audio_energy(i) > cfg.energy_thresh && start_idx == 0
        start_idx = i;
    elseif audio_energy(i) <= cfg.energy_thresh && start_idx ~= 0
        end_idx = i;
        if (end_idx - start_idx) > min_seg_len
            segments = [segments; start_idx, end_idx];
        end
        start_idx = 0;
    end
end
if start_idx ~= 0 && (total_samples - start_idx) > min_seg_len
    segments = [segments; start_idx, total_samples];
end

% 微调阈值以匹配目标段数
num_seg = size(segments, 1);
max_adjust_times = 3; % 最大微调次数
adjust_count = 0;

while num_seg ~= cfg.target_seg_num && adjust_count < max_adjust_times
    adjust_count = adjust_count + 1;
    if num_seg < cfg.target_seg_num
        cfg.energy_thresh = cfg.energy_thresh * 0.9;
        fprintf(' 第%d次微调：段数不足（%d段），能量阈值降至%.3f\n', adjust_count, num_seg, cfg.energy_thresh)
            ;
    else
        cfg.energy_thresh = cfg.energy_thresh * 1.1;
        fprintf(' 第%d次微调：段数过多（%d段），能量阈值升至%.3f\n', adjust_count, num_seg, cfg.energy_thresh)
            ;
    end

    segments = []; start_idx = 0;
    for i = 1:total_samples
        if audio_energy(i) > cfg.energy_thresh && start_idx == 0, start_idx = i; end
        if audio_energy(i) <= cfg.energy_thresh && start_idx ~= 0
            if (i - start_idx) > min_seg_len, segments = [segments; start_idx, i]; end
            start_idx = 0;
        end
    end
    if start_idx ~= 0 && (total_samples - start_idx) > min_seg_len
        segments = [segments; start_idx, total_samples];
    end
    num_seg = size(segments, 1);
end

% 最终段数检查
if num_seg ~= cfg.target_seg_num
    if num_seg < cfg.target_seg_num
        tip = '调小energy_thresh（如0.04）';
    else
        tip = '调大energy_thresh（如0.06）';
    end
    warning(' 最终分割出%d段（目标%d段）→ %s', num_seg, cfg.target_seg_num, tip);
```

```
else
    fprintf(' 成功分割出%d段语音 (匹配1~10) \n', cfg.target_seg_num);
end

%% 语音段起始前移
shift_samples = round(fs * cfg.shift_time);
segments_adjusted = segments;
for i = 1:num_seg
    segments_adjusted(i, 1) = max(1, segments_adjusted(i, 1) - shift_samples);
end

%% 提取原音频所有间隔
intervals_all = [];
for i = 1:num_seg-1
    interval_between = segments_adjusted(i+1, 1) - segments_adjusted(i, 2);
    intervals_all = [intervals_all; interval_between];
end
interval_after_last = total_samples - segments_adjusted(end, 2);
intervals_all = [intervals_all; interval_after_last];

%% 拼接音频
reversed_segments = flipud(segments_adjusted);
reversed_intervals = flipud(intervals_all);

audioOut = [];
for i = 1:num_seg
    seg_audio = audioIn(reversed_segments(i, 1):reversed_segments(i, 2));
    audioOut = [audioOut; seg_audio];
    silence = zeros(reversed_intervals(i), 1);
    audioOut = [audioOut; silence];
end

audioOut = audioOut / max(abs(audioOut)) * 0.9;

%% 保存最终音频
output_path = 'numberConverse.wav';
audiowrite(output_path, audioOut, fs);
fprintf('音频已保存: %s\n', output_path);

%% 可视化绘图
figure('Color','w','Position',[100 100 1000 700],'Name','音频反转结果可视化');

% 原始音频
subplot(2,1,1);
t_raw = (0:total_samples-1)/fs;
plot(t_raw, audioIn, 'Color', [0.2 0.4 0.6], 'LineWidth', 1); hold on;
% 标记原始段和前移后的段
for i = 1:num_seg
    plot([segments(i,1)/fs, segments(i,1)/fs], ylim, 'Color', [0.8 0.2 0.2], 'LineStyle', '--', 'LineWidth',
         1);
    plot([segments_adjusted(i,1)/fs, segments_adjusted(i,1)/fs], ylim, 'Color', [0.2 0.8 0.2], 'LineStyle',
        '-', 'LineWidth', 1.5);
    text(segments_adjusted(i,1)/fs, 0.8, num2str(i), 'Color', [0.2 0.8 0.2], 'FontSize', 9, 'FontWeight', '
        bold');
```

```
end
title('原始音频', 'FontSize', 12, 'FontWeight', 'bold');
xlabel('时间 (秒) ', 'FontSize', 10); ylabel('幅值', 'FontSize', 10);
xlim([0, total_duration]); ylim([-1.1, 1.1]);
grid on;
ax1 = gca;
ax1.LineWidth = 0.5;
box on;
legend('原始音频','原段起始','前移后段起始','Location','best','FontSize',9);

% 反转音频
subplot(2,1,2);
t_rev = (0:length(audioOut)-1)/fs;
plot(t_rev, audioOut, 'Color', [0.8 0.4 0.2], 'LineWidth', 1); hold on;
rev_seg_times = [];
current_pos = 1;
for i = 1:num_seg
    seg_len = reversed_segments(i,2) - reversed_segments(i,1) + 1;
    seg_start = (current_pos - 1)/fs;
    seg_end = (current_pos + seg_len - 1)/fs;
    rev_seg_times = [rev_seg_times; seg_start, seg_end];
    plot([seg_start, seg_start], ylim, 'Color', [0.2 0.4 0.6], 'LineStyle', '--', 'LineWidth', 1);
    text((seg_start+seg_end)/2, 0.8, num2str(11-i), 'Color', [0.2 0.4 0.6], 'FontSize', 9, 'FontWeight', '
        bold');
    current_pos = current_pos + seg_len + reversed_intervals(i);
end
title('反转后音频', 'FontSize', 12, 'FontWeight', 'bold');
xlabel('时间 (秒) ', 'FontSize', 10); ylabel('幅值', 'FontSize', 10);
xlim([0, t_rev(end)]); ylim([-1.1, 1.1]);
grid on;
ax2 = gca;
ax2.LineWidth = 0.5;
box on;

set(gcf, 'DefaultAxesFontName', 'SimHei');
set(gcf, 'DefaultTextFontName', 'SimHei');

saveas(gcf, 'numberConverse_visualization.png');
```

4.3.2 运行结果

程序运行结果如下图8

图中标注了代码语音分割后每一段音频以及对应音频段所读的数字。

输出音频保存在 ‘./matlab/countConverse/numberConverse.wav‘

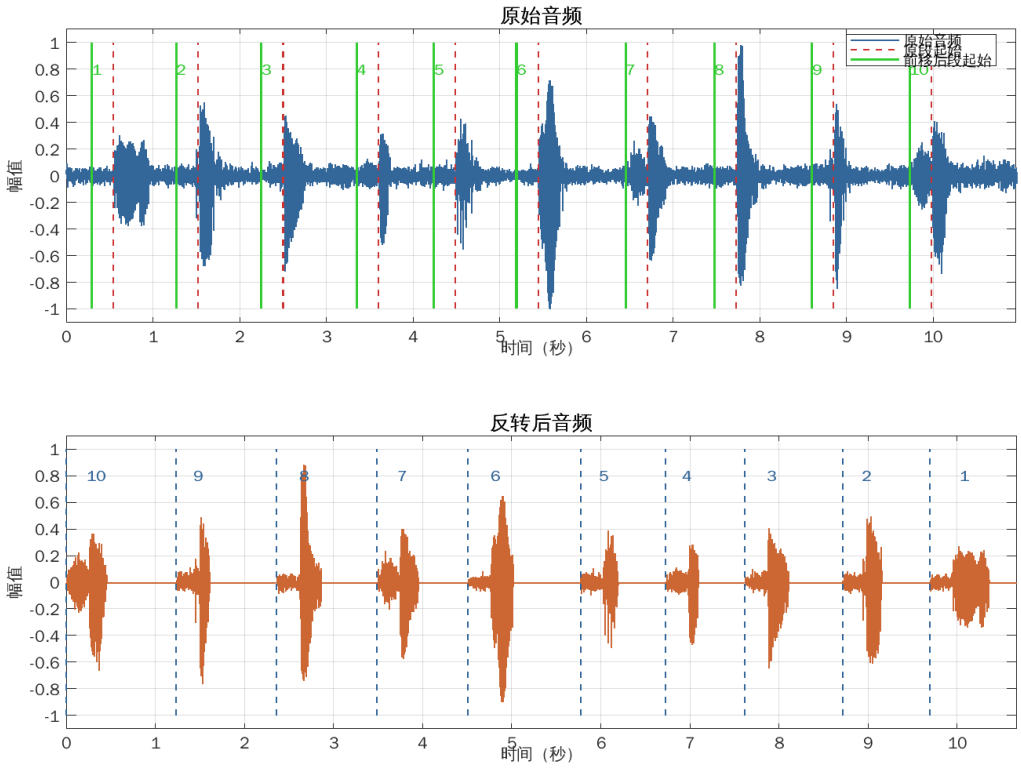

图 8: 语音段分割与反转拼接可视化结果