浙江大学

本科生课程作业报告

题目 基于 STFT 的旋律提取与和弦识别

小组成员 翁超然 3230100470

陈若涵 3230103695

邹依轩 3230104044

周　钰 3230105681

任课教师 赵均/季瑞松

年级专业 2023 级自动化（控制）

所在学院 控制科学与工程学院

完成日期 2025 年 6 月 1 日

目录

摘要

本研究提出一种基于短时傅里叶变换（STFT）的音乐旋律提取与和弦识别方法，旨在解决传统音乐信号处理在多声部、非平稳场景中的局限性。

通过构建“音频预处理—主旋律提取—和弦识别”三级处理框架，结合信号处理与音乐理论，实现了对音乐信号的时频特征解析与结构化和声识别。音频预处理模块通过信号归一化、带通滤波（20-2000Hz）、分帧（2048 样本窗长，70% 重叠率）与汉宁窗加窗处理，系统性提升信号质量，降低时频图基线噪声，提升频谱峰值能量集中度。主旋律提取模块基 STFT 构建时间-频率-能量矩阵，采用 Top-4 峰值频率提取策略（每帧前 4 个高能量频率）与置信度量化模型。和弦识别模块通过综合置信度筛选根音候选（前 3 个高置信度音符），结合 7 种和弦模板的音程完整性校验与时间域降噪机制，实现和弦准确识别，并通过 0.9 惩罚系数抑制三和弦多音符场景的过度匹配。

研究成果为音乐创作、教育（听音训练可视化）及信息检索（个性化推荐）提供了高效工具，未来可通过自适应窗长优化、深度学习融合进一步提升复杂场景性能。

关键词： 短时傅里叶变换（STFT）；基频提取；和弦识别；音乐信号处理；

Abstract

This study proposes a method for melody extraction and chord recognition in music signals based on the Short-Time Fourier Transform (STFT), aiming to address the limitations of traditional music signal processing in polyphonic and non-stationary scenarios. By constructing a three-stage processing framework of "audio preprocessing—melody extraction—chord recognition", the study integrates signal processing and music theory to achieve time-frequency feature analysis of music signals and structured harmonic recognition.

The audio preprocessing module systematically improves signal quality through signal normalization, band-pass filtering (20–2000 Hz), framing (2048-sample window length, 70% overlap rate), and Hanning windowing, reducing baseline noise in time-frequency diagrams and enhancing spectral peak energy concentration. The melody extraction module constructs a time-frequency-energy matrix based on STFT, employing a Top-4 peak frequency extraction strategy (top 4 high-energy frequencies per frame) and a confidence quantification model. The chord recognition module achieves accurate chord identification by screening root note candidates with comprehensive confidence scores (top 3 high-confidence notes), integrating interval integrity verification against 7 chord templates, and temporal noise reduction, while using a 0.9 penalty coefficient to suppress overmatching in triad scenarios with extra notes.

The research provides efficient tools for music creation, education (visualized ear training), and information retrieval (personalized recommendation), with potential for adaptive window optimization and deep learning integration to enhance performance in complex scenarios.

Keywords: Short-Time Fourier Transform (STFT); fundamental frequency extraction; chord recognition; music signal processing

基于 STFT 的旋律提取与和弦识别

1 引言

音乐信号处理作为人工智能与音频技术的交叉领域，近年来在智能音乐创作、数字音乐教育和音乐信息检索等领域展现出显著的应用潜力。旋律作为音乐的核心元素，承载着音乐的主题与情感表达；和弦则构建了音乐的和声框架，决定了音乐的风格与色彩基调。准确提取旋律并识别和弦，对于解析音乐结构、分析音乐风格以及实现音乐的自动化生成与编辑具有关键意义。

传统的音乐信号处理方法高度依赖人工特征设计与复杂的模型架构，难以适应多样化的音乐风格及复杂的声部交织场景。短时傅里叶变换（STFT）作为经典的时频分析工具，能够将时域信号转换为时频域表示，为非平稳音乐信号的分析提供了有效手段。然而，直接应用 STFT 进行音乐特征提取时，面临时频分辨率矛盾、频谱泄漏等问题，需结合音乐信号特性进行针对性优化。基于此，本研究基于 STFT 技术构建了小规模音乐旋律提取与和弦识别模型，通过音频预处理、主旋律提取及和弦识别三个子模块，实现了和弦识别的自动化。

项目框架示意图如图 1.1 所示。

图 1.1 项目框架示意图

2 理论基础

2.1 傅里叶变换（Fourier Transform, FT）

傅里叶变换是信号处理领域的核心工具之一，它能够将时域信号转换为频域表示，揭示信号的频率成分。对于连续时间信号$x(t)$,其傅里叶变换定义为：

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}\mathrm{d}t\#(2.1.1)$$

离散傅里叶变换(D)则是针对离散信号的傅里叶变换，对于长度为N的离散序列$x[n]$,其 DFT 定义为：

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}, \quad k = 0,1,\ldots,N-1\#(2.1.2)$$

快速傅里叶变换(FFT)是 DFT 的高效计算算法，将计算复杂度从$O(N^2)$降低到$O(NlogN)$,极大地提高了傅里叶变换的计算效率。

2.2 短时傅里叶变换（Short-Time Fourier Transform, STFT）

傅里叶变换虽然能够提供信号的频域信息，但无法反映信号的时变特性。为了分析非平稳信号，Gabor 于 1946 年提出了短时傅里叶变换(STFT)。STFT 的基本思想是通过一个滑动窗函数将信号分割成多个短时段，对每个短时段信号进行傅里叶变换，从而得到信号的时频表示。

对于连续时间信号$x(t)$,其 STFT 定义为：

$$\mathrm{STFT}_x(\tau,f) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-j2\pi ft}\mathrm{d}t\#(2.2.1)$$

其中，$w(t)$是窗函数τ是窗函数的中心位置，f 是频率。

对于离散时间信号$x[n]$,其离散 STFT 定义为：

$$\mathrm{STFT}_x[m,k]=\sum_{n=-\infty}^{\infty}x[n]w[n-m]e^{-j\frac{2\pi}{N}kn}\#(2.2.2)$$

其中，$w[n]$是离散窗函数，m表示时间帧，k表示频率bin。

2.3 STFT 在音乐信号处理中的优势

1、时频联合分析能力

音乐信号本质上是非平稳的，本项目主要使用的乐器钢琴的音符的衰减过程中，基频保持不变但泛音能量逐渐降低。STFT 的时频图可直观展示这种动态变化，而传统傅里叶变换仅能提供全局频谱，无法反映时间维度的特征。

2、计算效率与实时性

基于 FFT 的 STFT 实现可在毫秒级处理音频帧，满足实时应用需求。

3、物理可解释性

STFT 的时频单元对应实际的时间-频率局部能量，便于结合音乐理论分析。主旋律的基频通常位于低频区，且在时频图中呈现连续轨迹，可通过峰值追踪提取。

4、作为基础框架的扩展性

许多高级算法依赖 STFT 结果，如**音高检测，**通过频谱峰值确定基频；**声源分离，**利用时频掩码分离主旋律与伴奏；**和弦识别，**分析多基频对应的音程关系。

3 音频预处理模块

3.1 主要难点

音频预处理阶段主要考虑的问题如下：

1、不同音频信号的幅度差异可达数倍甚至数十倍，直接处理可能会出现量纲不一致等问题导致处理结果偏差较大。

2、选择合适的带通滤波器类型以及截止频率需要权衡，如果截止频率设置不当，可能会过滤掉有用的音频信息，或者无法有效抑制噪声。同时，滤波器的阶数选择也很关键，因为阶数太高可能存在数值计算复杂度及精确度的问题，但阶数过低也可能导致滤波效果不佳。

3、分帧和加窗的参数需要根据实际考量，该选择会影响后续处理的准确性。如果帧长过短，可能无法包含完整的音频特征；如果帧长过长，则会降低时间分辨率。加窗处理虽然可以减少频谱泄漏，但不同的窗函数和窗长也需要根据具体音频进行调整。

3.2 算法解决思路

该模块的主要目的是提高后续处理的准确性，主要通过对原始音频信号进行一系列操作，消除不同音频信号幅度的差异、抑制噪声，并将音频信号转换为适合后续处理的形式。依次包括**信号归一化**、**带通滤波**、**分帧处理**、**加窗处理**四个步骤。

1、信号归一化

原始音频信号因录制设备、文件格式不同，幅度可能相差数个数量级。利用公式

$$\mathrm{processed\_audio}=\frac{\mathrm{input\_audio}}{\max(|\mathrm{input\_audio}|)}\#(3.2.1)$$

将输入音频信号的幅度范围统一到[-1,1]之间，从而消除不同音频信号在幅度上的差异，避免因幅度不同而影响后续处理结果。

2、带通滤波

人类听觉范围为 20Hz-20kHz，但音乐中的核心信息集中在中低频段。高频区域多为设备底噪或演奏时的摩擦声，低频区域则可能包含环境振动干扰。为此，模块采 **Butterwort** 或 **Chebyshev I 型滤波器**，设计 20-2000Hz 带通特性：

- **Butterworth 滤波器**以 6 阶实现通带平坦响应，适合保留纯净基频；
- **Chebyshev I 型滤波器**以 4 阶换取更陡峭的过渡带，牺牲轻微通带纹波以强化噪声抑制。

通过 filtfilt 函数实现零相位滤波，避免传统滤波的相位失真，确保信号时域形态不变。

3、**分帧处理**

音频信号具有时变性（如钢琴音符的起振与衰减阶段频谱差异显著），需将其分解为短帧（通常 40-50ms）进行逐帧分析。模块采用 **2048 样本点窗长**（约 42ms，采样率 48kHz 时）与 **70%重叠率**（相邻帧重叠 1434 样本点）的分帧策略：

- **窗长选择**：平衡时间与频率分辨率。短窗适合捕捉瞬态音头，但频率分辨率低；长窗（如 100ms）利于精确基频检测，但时间定位差。42ms 窗长是钢琴等乐器稳态音分析的经验值。
- **重叠设计**：避免帧间边界突变导致的频谱断裂。70%重叠率使相邻帧共享大部分数据，确保基频轨迹在时间轴上连续平滑。

4、加窗处理

直接截断信号会引入吉布斯效应，导致频谱出现虚假峰值。模块采用周期汉宁窗，利用公式

$$w(n) = 0.5\left(1 - cos\frac{2\pi n}{N-1}\right) \#(3.2.2)$$

进行处理，其两端幅值为 0、中间呈余弦曲线，使信号在帧边界处自然过渡。汉宁窗的主瓣宽度适中，旁瓣衰减高可有效抑制泄漏，使频谱峰值更贴近真实频率。代码中通过 windowed_frames = frames .* window 将窗函数逐帧应用，为 STFT 计算提供“干净”的时域片段。

至此音频预处理模块处理完毕，将输出处理好的音频信号到主旋律提取模块。

3.3 创新点

我们设计的音频预处理模块创新点集中体现在**多环节协同处理架构**、**智能噪声抑制策略**与**频谱优化技术**三大维度，通过系统性技术整合与细节突破，构建了高精度的信号预处理框架。具体创新如下：

模块创新性地将信号归一化、带通滤波、分帧处理、加窗优化四大步骤有机串联，形成“标准化—频率筛选—时域分块—频域整形”的系统性处理链路。通过归一化统一音频幅度范围至[-1,1]，消除设备差异导致的量纲偏差；借助分帧与加窗的联动，确保信号时域连续性与频域稳定性；通过带通滤波剔除无效频段。体现“系统性优化优于单点改进”的创新思路。

模块突破固定滤波模式，设计 20-2000Hz 带通滤波区间，精准保留音乐信号核心频率成分同时抑制高频底噪与低频环境振动。更具突破性的是，模块支持 Butterworth 与 Chebyshev I 型滤波器动态切换：前者适用于需保留泛音细节的古典音乐，后者可针对电子乐等强噪声场景强化滤波效果。

针对傅里叶变换的固有缺陷，模块将周期汉宁窗引入预处理环节，通过其余弦加权特性使信号在帧边界处平滑过渡至零，有效抑制吉布斯效应引起的频谱泄漏。该创新使 STFT 时频图中的基频轨迹更锐利、旁瓣干扰更少，为后续频谱分析提供了更高质量的数据基础。

4 主旋律提取模块

4.1 主要难点

主旋律提取阶段主要考虑的问题如下：

1、**STFT 计算复杂度。**短时傅里叶变换需对每帧信号独立进行 FFT 运算，当处理时长超过 3 分钟的音频时，计算量随帧数呈线性增长，难以满足实时分析需求。

2、**峰值频率提取误差。**噪声干扰或多声部信号叠加时，幅度谱中可能出现虚假峰值，导致基频误判。此外，峰值检测参数的经验性设置，可能遗漏真实基频或引入冗余频率成分。

3、**置信度评估合理性。**单纯依赖峰值幅度归一化计算置信度，未考虑频率成分的音乐理论相关性（如泛音与基频的固定音程关系）。当强噪声峰值偶然高于真实基频时，置信度指标可能反向误导后续和弦识别。

4.2 算法解决思路

模块采用“时频分析—峰值筛选—可靠性量化”的分层处理架构，核心流程如下：

1、STFT 时频特征提取

短时傅里叶变换（STFT）作为连接时域与频域的核心桥梁，通过“固定窗滑动+分段 FFT”的方式，将预处理后的连续音频信号转换为具有时间分辨率的频谱序列，具体实现思路如下：

- **窗长与重叠率**：沿用预处理模块的样本窗长与重叠率。
- **频域转换**：计算每帧信号的幅度谱，生成时间-频率-能量三维矩阵$S(t,f)$，呈现各时刻频率成分的能量分布。

2、峰值频率筛选

在 STFT 生成的时频矩阵基础上，峰值频率筛选模块通过“能量排序—数量限制—可靠性标记”三级处理，从海量频率成分中精准提取高价值基频候选，构建后续和弦识别的核心输入特征：

- **Top-4 能量优先筛选**：对每帧幅度谱执行局部峰值检测，按能量幅度降序提取前 4 个峰值频率作为基频候选。
- **置信度量化模型**：定义置信度

$$C_i = \frac{A_i}{A_{\max}} \#(4.2.1)$$

其中A_i为第i个峰值幅度，$A_{\max}$为帧内最大幅度，将基频可靠性转化为$[0,1]$区间的量化指标。

4.3 创新点

模块创新聚焦于**时频分析技术升级**与**数据驱动的特征筛选策略**，通过计算效率优化与可靠性增强，构建高精度的主旋律提取框架：。

我们采用 STFT 对预处理后的音频信号进行计算，得到幅度谱，能够清晰展示不同时刻音频的频率成分分布。而相较于传统的基频提取方法，STFT 也可以更全面、细致地分析音频信号的频谱特征，为后续准确提取基频提供了丰富且可靠的数据基础。

在峰值频率提取方面，我们限定数量从而完成了高效的提取。对每帧的幅度谱进行处理时，提取前 4 个峰值频率作为基频。同时，通过优先选择幅度较大的峰值频率，能够聚焦于音频信号中能量较强、更具代表性的频率成分，更有可能捕捉到主旋律的基频。

此外，我们对提取到的峰值频率作了置信度的量化评估。通过将每个峰值的幅度除以该帧幅度谱的最大值进行归一化处理，得到每个基频的置信度。这一创新点为每个提取的基频赋予了一个量化的可靠性指标，有助于后续对基频的筛选、排序以及和弦识别等处理步骤中，根据置信度的高低进行更合理的决策，提高了整体系统的准确性和稳定性。

5 和弦识别模块

5.1 主要难点

和弦识别模块需要解决的主要问题如下：

1、生成根音候选并遍历所有可能的根音和和弦类型进行匹配，计算量较大。同时，如何准确计算相对于根音的半音间隔、判断是否包含所有必要音程以及计算匹配度分数，都需要考虑多种因素，以提高和弦识别的准确性。

2、多基频转换为 MIDI 音符存在误差，尤其是在基频提取不准确的情况下。这些误差可能会影响后续的和弦匹配和识别结果。

5.2 算法解决思路

模块采用“置信度加权筛选—音程严格匹配—后处理降噪”的三级处理架构，核心流程如下：

1、置信度聚合

- **有效基频筛选：**过滤置信度低于 0.3 的频率成分（基频可靠性低于 30%），保留能量占优的前 4 个基频。
- **MIDI 映射与去重：**通过十二平均律公式

$$\mathrm{MIDI} = 69 + 12\log_2\left(\frac{f}{440}\right) \#(5.2.1)$$

转换为 MIDI 音符，去除重复音符后生成唯一音符集合。

- **综合置信度计算：**对每个唯一音符，取其在原始基频中的最大置信度（举例而言，某 MIDI 音符对应 2 个原始基频，置信度分别为 0.6 和 0.5，则综合置信度为 0.6），作为该音符的可靠性指标。

2、和弦匹配算法

- **根音候选筛选：**按综合置信度降序选取前 3 个唯一音符作为候选根音，从 12 个可能根音降至 3 个，显著减少遍历空间。
- **音程匹配引擎：**对每个候选根音，利用公式

$$\mathrm{interval} = MIDI_{\mathrm{note}} - MIDI_{\mathrm{root}} \bmod 12 \#(5.2.2)$$

计算唯一音符与根音的半音间隔，与预定义和弦的音程模板进行匹配；仅当所有模板音程均被包含，进入匹配度计算。

和弦匹配采用匹配音符的综合置信度均值，利用公式

$$\mathrm{score} = \frac{1}{N}\sum C_{\mathrm{matched}} \#(5.2.3)$$

并对三和弦中超过 3 个音符的情况引入 0.9 的惩罚系数以避免过度匹配。

3、后处理噪声过滤

- **置信度阈值筛选：**将匹配度低于 0.5 的帧标记为“未知”，从而实现过滤低可靠识别结果。
- **时间域连续性校验：**若某帧和弦与前后帧均不同（单帧孤立点），或首/末帧与相邻帧不一致，则标记为“未知”。

5.3 创新点

模块创新聚焦于**置信度驱动的智能筛选**与**结构化音程匹配算法**，通过数据可靠性增强与规则化匹配，构建高精度的和弦识别框架：

模块突破传统根音遍历的盲目性，提出基于综合置信度的动态筛选策略，通过可靠性量化实现根音候选的智能优选。具体而言，算法对每帧多基频检测结果进行去重后，按唯一音符的综合置信度降序排列，

优先选取前 3 个高置信度音符作为根音候选。这一机制通过将数据可靠性转化为可量化的筛选指标，显著降低了多声部信号中次要音高对根音判断的干扰，为后续音程匹配奠定了高可靠的基础输入。

在音程匹配阶段，预设 7 种和弦类型的音程模板，仅当检测到的音符包含所有必要音程时才进行匹配度计算，避免因关键音程缺失导致的误判。针对三和弦中检测到 4 个音符的场景，引入 0.9 的分数惩罚系数抑制过度匹配。在后处理阶段，通过双重降噪机制过滤低可靠结果：一是剔除匹配度<0.5 的帧，二是通过前后帧一致性校验过滤单帧噪声。

6 结果分析

在调试模型代码和测试模型功能的时候，我们选取了众多音频信号（如下图）进行分析，基于分析的结果对代码进行逐步改进，最后得出了理想的结果，现在综合多次的实验效果，对我们的模型结果进行分析。

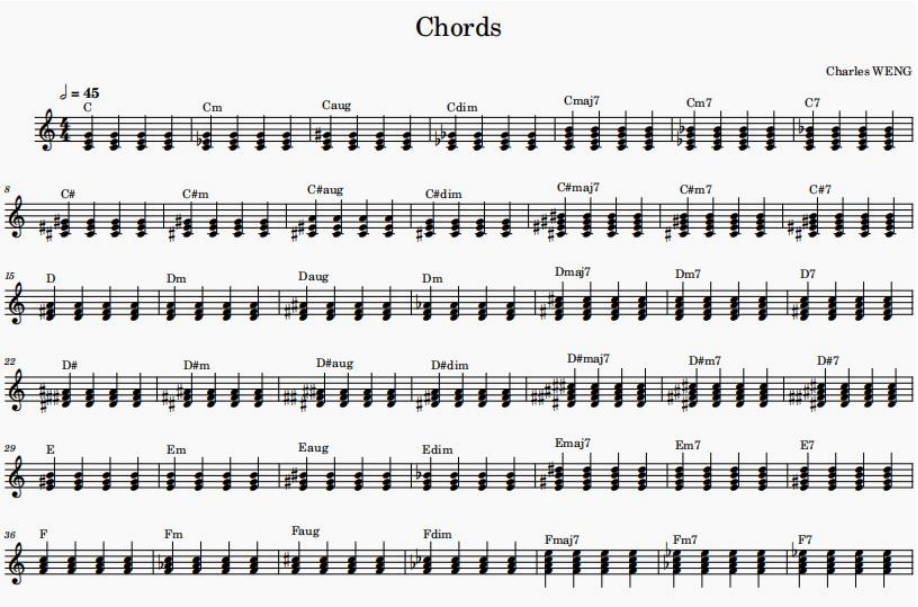

图 6.1 和弦选段 1

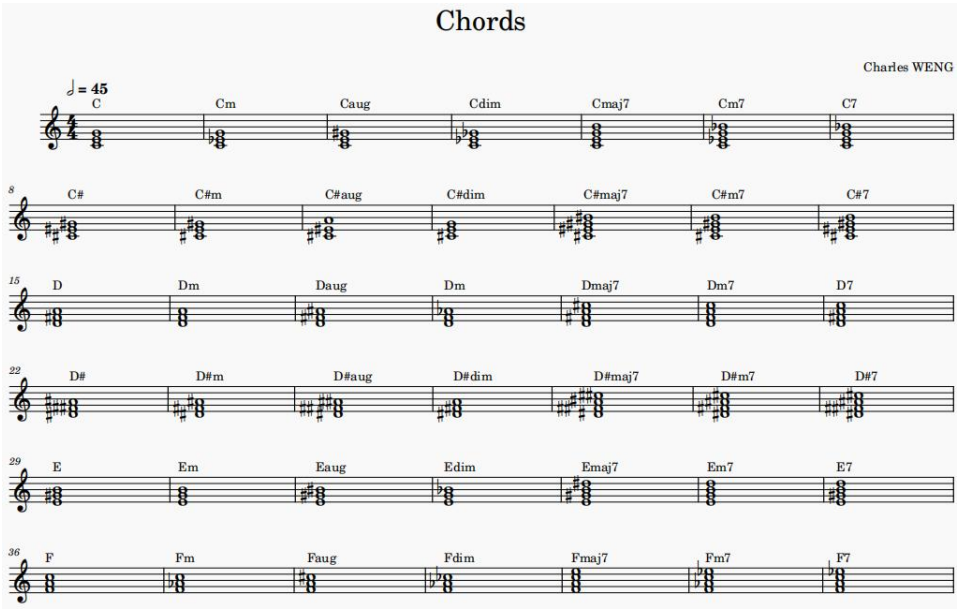

图 6.2 和弦选段 2

6.1 音频预处理结果分析

由实验结果可见，经过归一化处理后，音频信号的幅度范围被统一，不同音频文件的幅度特性不再影响后续处理，而高频噪声也得到有效抑制，信号的质量得到了提高，音频信号被分割成多个短帧，处理也更加灵活，但是有时容易受到极端噪声干扰。

6.2 旋律提取结果分析

实验结果表明，在大多数情况下，我们的旋律提取方法能够准确地提取出音乐中的主旋律，基频估计误差较小，音高检测准确率较高。然而，在一些复杂的音乐场景中，如多声部交织、存在大量噪音的情况下，旋律提取的准确性会受到一定的影响。这主要是因为 STFT 在处理复杂信号时，可能会出现频率分辨率不足、频谱泄漏等问题。

6.3 和弦识别结果分析

实验结果显示，我们的和弦识别方法在大多数情况下能够准确地识别出音乐中的和弦类型，和弦识别准确率较高。但在一些和弦变化较快、和弦结构复杂的音乐片段中，和弦识别的准确性会有所下降。这可能是由于多基频提取的误差以及和弦匹配算法在处理复杂和弦时的局限性导致的。

7 实例演示

为了进一步说明我们的模型，以下基于“chords.csv”进行实际识别效果的展示和说明。

7.1 可视化结果输出

以下是我们编写程序中 Matlab 生成的可视化结果。

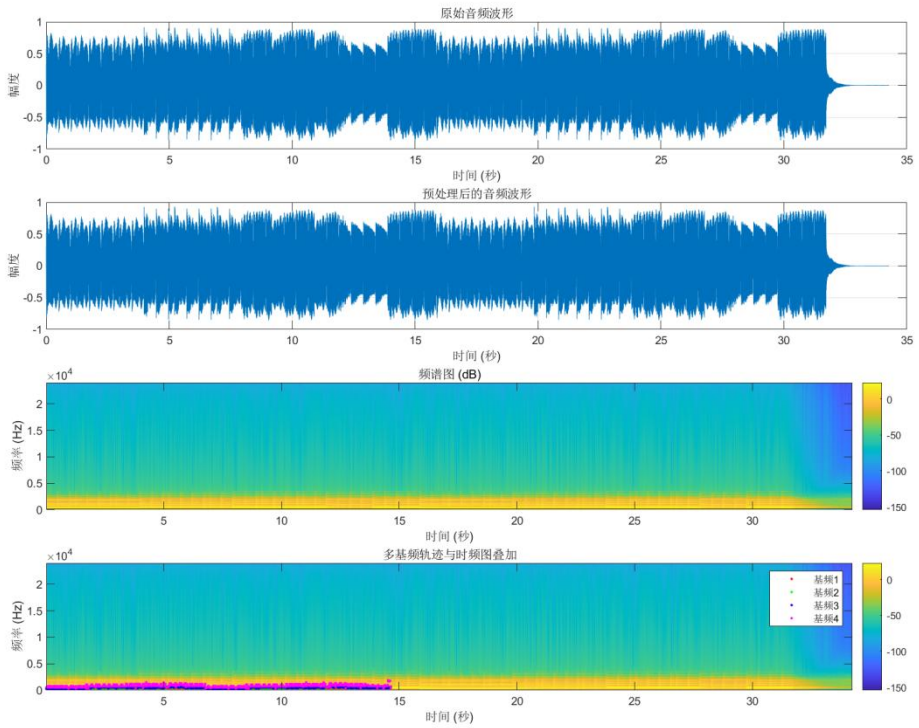

图 7.1.1 音频波形、频谱图和多基频叠加图

由图可见预处理后的音频相比起原始音频少了一些突起，但总体上没有很大的变化，而频谱图和多基频轨迹与时频图叠加图则清晰展示了时域上输入音频经过处理后的状态。

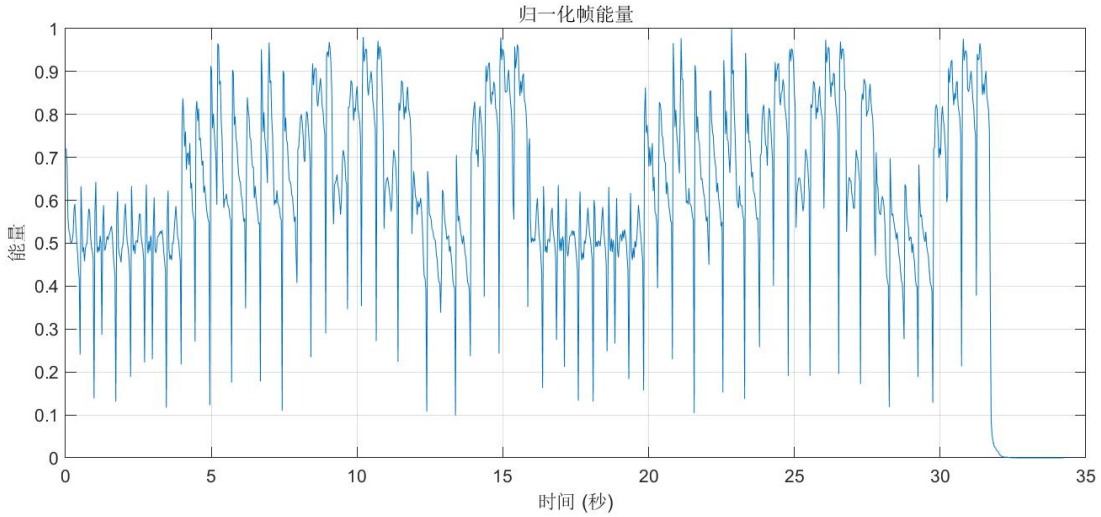

图 7.1.2 归一化能量帧示意图

经过归一化处理后的能量帧虽然仍然有很多凹凸不平的突起，但是可以发现已经隐隐有峰值的产生。

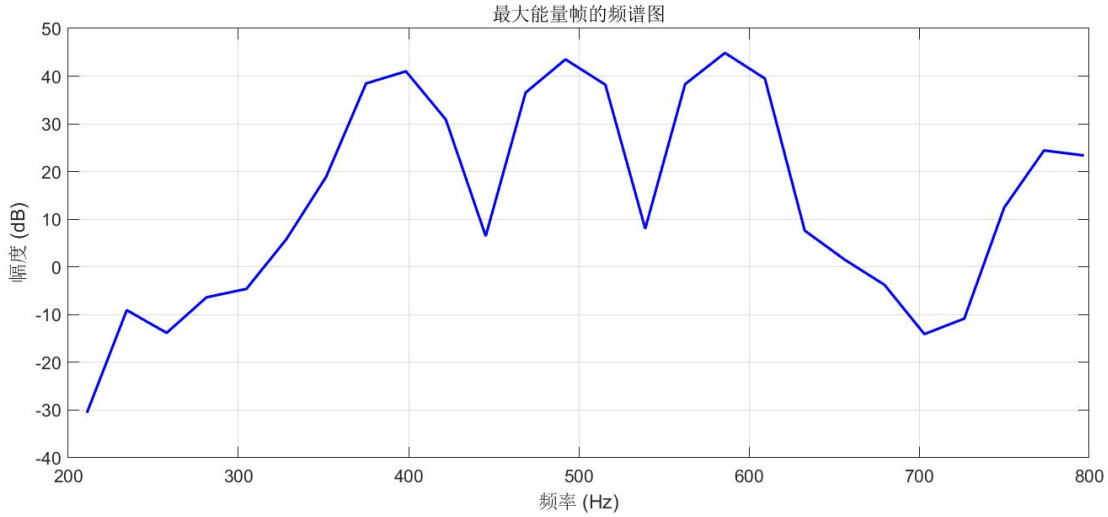

图 7.1.3 最大能量帧频谱图

这是最大能量帧在频域上的示意图，可以清晰发现 3 个峰值，即我们要识别的和弦信号的特征。

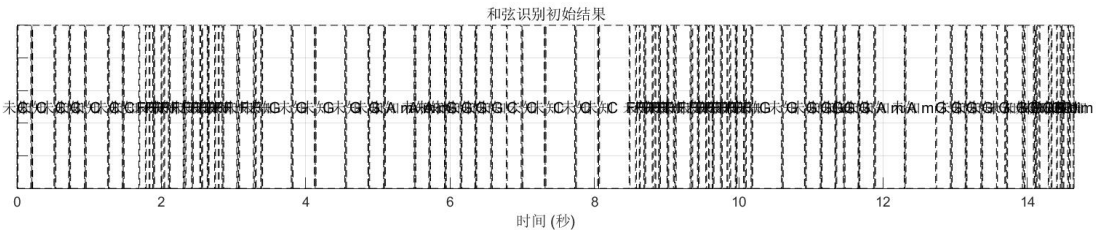

图 7.1.4 和弦识别初始结果示意图

这是我们的最终识别结果输出，即按照相应的时间段输出每一帧能识别到的和弦，如果识别到和弦则输出相应和弦的名称，如果未识别到则输出“未知”，对识别结果有一个可靠的区分。

7.2 交互界面

由于上述输出结果过于重叠，不便于真实效果展示，我们基于整体模型设计了一个美观的可交互用户界面，这个界面能够清晰地展示我们识别的效果。

图 7.2.1 交互界面示意图

这是我们的输入交互界面，用户可以通过输入音频的 csv 文件，从而借助我们的工具对其识别。

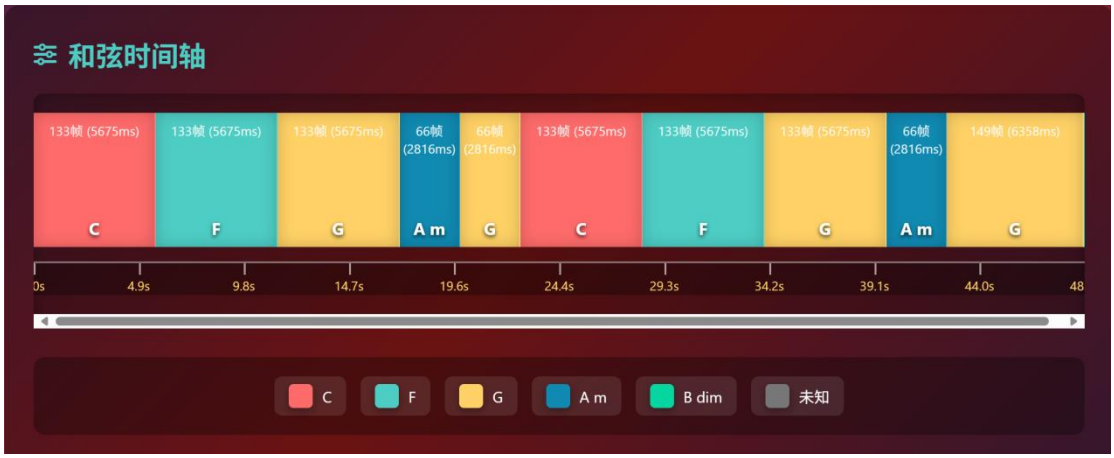

图 7.2.2 识别结果示意图

这是我们的输出识别结果示意图，基于模型的识别结果清晰而美观地展示了我们的识别效果。每一个

色块对应一个不同的和弦，并根据色块的宽度显示了每一个和弦持续的时间，用户可以拖动进度条从而观察每一段时间上和弦的变化。当鼠标置于这个色块之上时，也会显示和弦名、持续时间以及帧数。

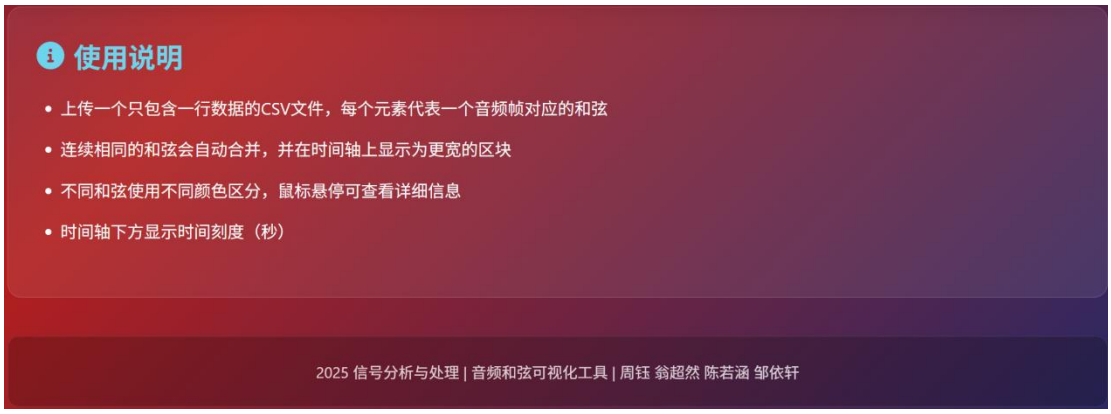

图 7.2.3 使用说明示意图

这是交互界面自带的使用说明，也有助于用户更好地使用我们的交互工具。

7.3 误差分析

在识别的过程中，我们发现第 4 个音有时识别不出来，这将导致模型不能准确区分 Am 和弦与 Am7 和弦。基于此，我们对模型识别过程中可能存在的误差进行了分析，找出可能存在的误差产生原因如下：

1、**归一化误差。**输入音频信号本身可能存在异常的幅度峰值，例如短暂的爆音，归一化可能会使正常音频部分的幅度被过度压缩，影响后续处理。

2、**STFT 计算误差。**计算时窗函数的选择和参数设置会影响频谱的分辨率和精度，而频谱分辨率不足可能导致无法准确区分相邻的频率成分，频谱泄漏会使峰值频率的提取不准确。推测我们的模型不能准确区分 Am 和弦和 Am7 和弦的原因可能正是这个。

3、**峰值频率提取误差。**只提取每帧幅度谱的前 4 个峰值频率作为基频，可能会遗漏一些重要的频率成分，特别是在复杂音乐音频中，某些基频的幅度可能不是最大的，而不准确的基频提取会直接影响后续和弦识别的结果。

4、**根音候选误差。**根音候选是根据音符的综合置信度排序后选取前几个，这种选择方式可能会排除真正的根音，特别是当真正的根音置信度较低时。

5、**和弦匹配误差。**和弦匹配规则是基于计算相对于根音的半音间隔和匹配度分数，但这种规则可能过于严格或宽松，无法准确判断和弦类型。对于一些有转位或省略音的和弦，可能无法准确匹配。

7.4 改进方向

基于对实验结果的分析，我们对模型提出了以下改进方向：

1、**优化预处理参数。**根据不同类型的音频信号，调整归一化、滤波、分帧和加窗的参数，以提高效果。

2、**改进主旋律提取方法。**可以考虑提取更多的峰值频率，或者采用更复杂的基频提取算法，提高基频提取的准确性。

3、**优化 STFT 算法。**改进现有 STFT 算法，从而提高频率分辨率，减少频谱泄漏，以增强旋律提取在复杂信号下的准确性。

4、**引入深度学习方法。**借助深度学习模型，让它反复识别海量的音频，包括用各种乐器演奏出的和弦，利用深度学习强大的特征提取能力，对音频信号进行更深入的分析，提高旋律提取和和弦识别的性能。

5、**改进和弦匹配算法。**现有和弦匹配算法效率比较低、过于简单，对复杂和弦结构的识别效果较差，我们需要优化和弦匹配规则，引入更灵活的匹配规则，如允许一定程度的音程偏差，以提高识别的准确性。

8 应用价值

该模型具备多方面应用价值，以下将从音乐领域和技术领域两个方面展开具体的阐述。

8.1 音乐领域

在**音乐创作**方面，我们的模型能够辅助音乐创作者进行和弦的编排，并为创作者提供灵感的来源。因为对于创作者而言，手动分析音频中的和弦结构十分耗时且需要专业知识，而我们的模型能够自动识别音频中的和弦类型，帮助零基础的创作者快速搭建歌曲的和弦框架，提高创作效率。

在**音乐教育**方面，我们的模型可以作为教学辅助工具，将复杂的音频信号转化为直观的基频序列和和弦类型，帮助无乐理基础的学生更好地理解和分析音乐作品，提高他们的音乐理论水平。同时可以辅助进行艺考的学生进行听音训练，从而提高他们的音乐感知能力和听力水平。

在**音乐产业**方面，我们的模型可以提取音频的主旋律和和弦特征，根据这些特征为用户推荐相似风格的音乐，提高音乐推荐的准确性和个性化程度，从而提升用户体验，增加用户对音乐平台的粘性。除此以外，还可以用于音乐作品的相似度检测，帮助版权方判断是否存在侵权行为，保护音乐创作者的权益。

8.2 技术领域

在算法优化与创新方面，我们的模型可以为后续的音频处理算法研究提供参考，同时也可以在现有模型的算法基础上进行优化和改进，提高主旋律提取和和弦识别的准确性和效率。更进一步，还可以探索新的算法和技术，如深度学习在音频处理中的应用。

在跨学科研究方面，由于音频处理涉及信号处理、机器学习、音乐理论等多个学科领域，因此我们的研究可以促进这些学科之间的交叉融合，推动跨学科研究的发展，同时也帮助我们自身更好地理解这些学科基础知识，对我们未来的学习和研究有一定启发。

9 总结与展望

本模型基于 STFT 技术实现了音乐音频中的旋律提取和和弦识别，通过音频预处理、多基频提取和和弦识别等模块，完成了从音频信号到基频序列和和弦类型的处理。实验结果表明，该方法在大多数情况下能够取得较好的效果，但在复杂音乐场景中仍存在一定的局限性。未来的研究可以从优化 STFT 算法、引入深度学习方法和改进和弦匹配算法等方面入手，进一步提高旋律提取和和弦识别的性能。同时，可以将该模型应用于音乐制作、音乐教育和音乐推荐等领域，为音乐产业的发展提供更多的支持和创新。此外，还可以进一步探索深度学习等新技术在音乐分析中的应用，以实现更高级的音乐处理功能。

参考文献

[1]杨若芳,项顶. 基于Matlab实现音乐识别与自动配置和声的功能[J]. 自动化与仪器仪表,2011(1):126-128. DOI:10.3969/j.issn.1001-9227.2011.01.048.

附录

A. main.m（主程序）

```
close all; clear; clc;

%% 参数配置
% audio_path = '../Test/Sources/Chords/Chords_full/Chords_full_0009.wav';
audio_path = '../Test/Sources/Songs/第一夜.wav';
% 钢琴基频库
c4_midi = 60;               % 中央 C 的 MIDI 编号
g5_midi = 79;               % 高音谱号上加一线 G5 的 MIDI 编号
a4_midi = 69;               % A4 的 MIDI 编号
a4_freq = 440;              % A4 的频率(Hz)
piano_midi = c4_midi:g5_midi;  % 中央 C 到 G5 的范围
piano_freqs = a4_freq * 2.^((piano_midi - a4_midi)/12);  % 十二平均律计算频率

% 创建结果目录
timestamp = datetime('now', 'Format', 'yyyy-MM-dd_HH-mm-ss');
output_dir = fullfile('..', 'Results', char(timestamp));
if ~exist(output_dir, 'dir'), mkdir(output_dir); end
fprintf('成功创建目录: %s\n', output_dir);

%% 加载音频
try
    [audio, fs] = audioread(audio_path);
    if size(audio, 2) > 1
        audio = audio(:, 1);
        fprintf('注: 输入音频为立体声, 已自动转换为单声道\n');
    end
    fprintf('成功加载音频: %s\n 采样率: %d Hz, 长度: %.2f 秒\n', ...
        audio_path, fs, length(audio)/fs);
catch ME
    error('音频加载失败: %s', ME.message);
end

%% 预处理
fprintf('\n===== 开始预处理 =====\n');
[processed_audio, params] = preprocessing(audio, fs);

%% 多基频提取
fprintf('\n===== 开始多基频提取 =====\n');
[multi_pitch, confidences] = melody_extraction(processed_audio, fs, params,
output_dir);

% 保存多基频结果
frame_time = (0:size(multi_pitch,1)-1) * params.hop_length / fs;
output_data = [frame_time', multi_pitch, confidences];
output_data(isnan(output_data)) = 0;
```

```
writematrix(output_data, fullfile(output_dir, 'multi_pitch.csv'), 'Delimiter', ',');
fprintf('多基频结果已保存至: %s\n', fullfile(output_dir, 'multi_pitch.csv'));

%% 和弦识别
fprintf('\n===== 开始和弦识别 =====\n');
chord_labels = chord_recognition(multi_pitch, confidences, piano_midi, piano_freqs);

% 保存和弦结果
chord_data = [cellstr(chord_labels')];
writecell(chord_data, fullfile(output_dir, 'chords.csv'));
fprintf('和弦识别结果已保存至: %s\n', fullfile(output_dir, 'chords.csv'));

%% 可视化结果
fprintf('\n===== 生成可视化结果 =====\n');
generate_visualizations(audio, processed_audio, fs, params, multi_pitch,
chord_labels, output_dir);

fprintf('\n===== 处理完成 =====\n 结果已保存至: %s\n', output_dir);
```

B. preprocessing.m（音频预处理模块）

```
function [processed_audio, params] = preprocessing(input_audio, fs)
% 音频预处理模块
% 输入:
%   input_audio - 输入音频信号（列向量）
%   fs - 采样率（Hz）
% 输出:
%   processed_audio - 预处理后的音频信号（列向量）
%   params - 处理参数结构体

%% 1. 信号归一化
processed_audio = input_audio / max(abs(input_audio)); % 归一化到[-1, 1]

%% 2. 设置处理参数
window_length = 2048;     % 长时窗（42ms@48kHz）
overlap = 1434;           % 70%重叠率
filter_type = 'chebyshev';
cutoff_freqs = [20, 2000]; % 抑制高频噪声
params = struct('window_length', window_length, 'overlap', overlap, ...
    'filter_type', filter_type, 'cutoff_freqs', cutoff_freqs, 'fs', fs);

%% 3. 应用带通滤波器（保留 20-10kHz）
processed_audio = apply_bandpass_filter(processed_audio, fs, filter_type,
cutoff_freqs);

%% 4. 分帧处理（返回帧矩阵，每行是一帧）
[frames, hop_length] = audio_framing(processed_audio, window_length, overlap);
params.frames = frames;
params.num_frames = size(frames, 2); % 总帧数
params.hop_length = hop_length;

%% 5. 加窗处理（汉宁窗）
window = hann(window_length, 'periodic'); % 周期汉宁窗
params.window = window;
params.windowed_frames = frames .* window(:, ones(1, params.num_frames)); % 带窗帧
矩阵

fprintf('预处理完成\n 窗长: %d 样本 (%.1fms), 重叠率: %.1f%%\n 滤波器: %s, 通
带: %d-%dkHz\n', ...
    window_length, window_length/fs*1000, overlap/window_length*100,
filter_type, ...
    cutoff_freqs(1)/1000, cutoff_freqs(2)/1000);

end

%% 子函数：应用带通滤波器
function filtered_audio = apply_bandpass_filter(audio, fs, filter_type, cutoff_freqs)
    Wn = cutoff_freqs / (fs/2); % 归一化截止频率
```

```
    Wn = max(min(Wn, 0.99), 0.01); % 防止接近 0 或 1

    switch lower(filter_type)
        case 'butterworth'
            [b, a] = butter(6, Wn, 'bandpass');
        case 'chebyshev'
            [b, a] = cheby1(4, 1, Wn, 'bandpass'); % 降低阶数避免数值问题
        otherwise
            error('不支持的滤波器类型');
    end

    filtered_audio = filtfilt(b, a, audio); % 零相位滤波
end

%% 子函数：音频分帧
function [frames, hop_length] = audio_framing(audio, window_length, overlap)
    hop_length = window_length - overlap;
    num_frames = floor((length(audio) - window_length) / hop_length) + 1;
    frames = zeros(window_length, num_frames);

    for i = 1:num_frames
        start_idx = (i-1)*hop_length + 1;
        end_idx = start_idx + window_length - 1;
        frames(:, i) = audio(start_idx:end_idx);
    end
end
```

C. melody_extraction.m（基频提取模块）

```
function [multi_pitch, confidences] = melody_extraction(audio, fs, params, output_dir)
% 多基频提取模块
% 输入:
%   audio         - 输入音频信号
%   fs            - 采样率
%   params        - 参数结构体，包含:
%                   .window_length   - 窗长
%                   .hop_length      - 跳长
%                   .window          - 窗函数
%                   .windowed_frames - 加窗后的帧数据(用于可视化)
%   output_dir    - 可视化结果输出目录
% 输出:
%   multi_pitch   - 多基频矩阵(帧数×4，每行最多 4 个频率值)
%   confidences   - 对应基频的置信度矩阵(帧数×4)
    window_length = params.window_length;
    hop_length = params.hop_length;
    window = params.window;

    % 计算 STFT
    [S, F, T] = spectrogram(audio, window, hop_length, window_length, fs, 'yaxis');
    S = abs(S);  % 幅度谱
    F = F(1:window_length/2+1);  % 频率向量
    T = T';  % 时间向量

    % 计算每帧能量
    frame_energy = sum(S.^2, 1)';  % 每帧能量
    [~, max_energy_idx] = max(frame_energy);  % 找到能量最大的帧索引

    % 初始化结果矩阵
    num_frames = size(S, 2);
    multi_pitch = zeros(num_frames, 4);     % 每帧最多 4 个基频
    confidences = zeros(num_frames, 4);     % 对应置信度

    % 对每一帧提取前 4 个峰值频率
    for i = 1:num_frames
        frame_spectrum = S(:, i);

        % 查找局部最大值
        [pks, locs] = findpeaks(frame_spectrum, 'SortStr', 'descend', 'NPeaks', 4);

        % 转换为频率
        peak_freqs = F(locs);

        % 保存结果
        num_peaks = min(4, length(pks));
        if num_peaks > 0
            multi_pitch(i, 1:num_peaks) = peak_freqs(1:num_peaks);
```

```
            confidences(i, 1:num_peaks) = pks(1:num_peaks) / max(frame_spectrum); % 归
一化置信度
        end
    end

    % 可视化帧能量
    visualize_energy(frame_energy, T, output_dir);

    % 可视化最大能量帧的频谱图
    max_energy_frame = params.windowed_frames(:, max_energy_idx);
    visualize_spectrum(max_energy_frame, fs, output_dir);

    fprintf('多基频提取完成 - 总帧数: %d\n', num_frames);
    fprintf('最大能量帧索引: %d (%.2f 秒)\n', max_energy_idx, T(max_energy_idx));
end

%% 子函数：可视化帧能量
function visualize_energy(frame_energy, T, output_dir)
    figure('Position', [100, 100, 1000, 400]);
    plot(T, frame_energy/max(frame_energy));
    title('归一化帧能量');
    xlabel('时间 (秒)');
    ylabel('能量');
    grid on;

    % 保存图像
    saveas(gcf, fullfile(output_dir, 'frame_energy.png'));
end

%% 子函数：可视化指定帧的频谱图
function visualize_spectrum(frame, fs, output_dir)
    N = length(frame);
    fft_frame = fft(frame);
    fft_frame = fft_frame(1:N/2+1);
    f = (0:N/2)*(fs/N);
    mag = abs(fft_frame);

    % 只显示 200-800Hz 范围内的频谱
    freq_range = [200, 800];
    idx_range = (f >= freq_range(1)) & (f <= freq_range(2));

    % 绘制指定频率范围的频谱图
    figure('Position', [100, 100, 1000, 400]);
    plot(f(idx_range), 20*log10(mag(idx_range)), 'b-', 'LineWidth', 1.5);

    % 添加标题和标签
    title('最大能量帧的频谱图');
    xlabel('频率 (Hz)');
    ylabel('幅度 (dB)');
```

```
    grid on;

    % 设置 x 轴范围
    xlim(freq_range);

    % 保存图像
    saveas(gcf, fullfile(output_dir, 'max_energy_frame_spectrum.png'));
end
```

D. chord_recognition.m（和弦识别模块）

```
function chord_labels = chord_recognition(multi_pitch, confidences, piano_midi,
piano_freqs)
% 和弦识别模块
% 输入:
%   multi_pitch   - 多基频矩阵 (帧数×4, 每行最多 4 个频率值)
%   confidences   - 对应基频的置信度矩阵 (帧数×4)
%   piano_midi    - 钢琴 MIDI 编号数组
%   piano_freqs   - 对应钢琴频率数组
% 输出:
%   chord_labels  - 和弦标签细胞数组

%% 初始化参数
num_frames = size(multi_pitch, 1);
chord_labels = cell(num_frames, 1);

%% 定义 7 种和弦类型
chord_types = [
    struct('name', '',    'intervals', [0,4,7],   'min_notes', 3);
    struct('name', 'm',    'intervals', [0,3,7],   'min_notes', 3);
    struct('name', 'aug',    'intervals', [0,4,8],   'min_notes', 3);
    struct('name', 'dim',    'intervals', [0,3,6],   'min_notes', 3);
    struct('name', '7',    'intervals', [0,4,7,10], 'min_notes', 4);
    struct('name', 'maj7',    'intervals', [0,4,7,11], 'min_notes', 4);
    struct('name', 'm7',    'intervals', [0,3,7,10], 'min_notes', 4)];

%% 频率到 MIDI 编号的映射函数
    function midi_note = freq_to_midi(freq)
        if isempty(freq) || isnan(freq) || freq <= 0
            midi_note = NaN;
            return;
        end
        [~, idx] = min(abs(freq - piano_freqs)); % 寻找最接近的钢琴频率
        midi_note = piano_midi(idx); % 获取对应的 MIDI 编号
    end

%% 主循环：逐帧处理
frame_scores = zeros(num_frames, 1); % 存储每帧的最佳分数
for frame = 1:num_frames
    %% 提取当前帧有效基频和置信度
    freqs = multi_pitch(frame, :);
    confs = confidences(frame, :);
    valid_mask = ~isnan(freqs) & freqs > 0;
    valid_freqs = freqs(valid_mask);
    valid_confs = confs(valid_mask);

    %% 转换为 MIDI 音符并去重
    midi_notes = arrayfun(@freq_to_midi, valid_freqs);
```

```
    [unique_notes, ~, note_indices] = unique(midi_notes);
    num_unique = length(unique_notes);
    if num_unique < 2 % 至少需要 2 个音符
        chord_labels{frame} = '未知';
        frame_scores(frame) = 0;
        continue;
    end

    %% 计算每个唯一音符的综合置信度
    unique_confs = arrayfun(@(n) max(valid_confs(note_indices == n)), 1:num_unique);

    %% 生成根音候选（按置信度降序排列）
    [~, sort_idx] = sort(unique_confs, 'descend');
    root_candidates = unique_notes(sort_idx);

    %% 遍历根音候选和和弦类型
    best_score = -inf;
    best_chord = '未知';
    for root_idx = 1:min(3, length(root_candidates)) % 只考虑前 3 个根音候选
        root = root_candidates(root_idx);
        intervals = mod(unique_notes - root, 12); % 计算相对于根音的半音间隔

        %% 遍历所有和弦类型
        for ct = 1:length(chord_types)
            ct_struct = chord_types(ct);
            if num_unique < ct_struct.min_notes % 音符数不足
                continue;
            end

            %% 检查是否包含所有必要音程
            required = ct_struct.intervals;
            missing = false(size(required));
            for i = 1:length(required)
                missing(i) = ~any(intervals == required(i));
            end
            if any(missing)
                continue; % 跳过不匹配的和弦类型
            end

            %% 计算和弦匹配度分数
            matched = ismember(intervals, required);
            matched_confs = unique_confs(matched);
            score = mean(matched_confs); % 匹配音符的置信度均值

            %% 特殊处理：如果是三和弦但检测到 4 个音符，降低分数
            if ct_struct.min_notes == 3 && num_unique > 3
                score = score * 0.9; % 轻微惩罚
            end
```

```
            %% 更新最优解
            if score > best_score
                best_score = score;
                root_name = midi_to_note_name(root);
                best_chord = [root_name, ' ', ct_struct.name];
            end
        end
    end

    %% 保存结果，置信度小于 0.4 的标记为未知
    if best_score < 0.4
        chord_labels{frame} = '未知';
        frame_scores(frame) = 0;
    else
        chord_labels{frame} = best_chord;
        frame_scores(frame) = best_score;
    end
end

%% 后处理：过滤掉短持续时间的和弦（1 帧/2 帧）
if num_frames >= 3
    for frame = 2:num_frames-1
        % 检查当前帧是否与前后帧都不同
        if strcmp(chord_labels{frame}, chord_labels{frame-1}) == 0 && ...
           strcmp(chord_labels{frame}, chord_labels{frame+1}) == 0
            chord_labels{frame} = '未知';
        end
    end

    % 处理第一帧
    if num_frames >= 2 && strcmp(chord_labels{1}, chord_labels{2}) == 0
        % 第一帧与第二帧相同，保留
    else
        chord_labels{1} = '未知';
    end

    % 处理最后一帧
    if num_frames >= 2 && strcmp(chord_labels{num_frames}, chord_labels{num_frames-1})
== 0
        % 最后一帧与倒数第二帧相同，保留
    else
        chord_labels{num_frames} = '未知';
    end
end

%% 辅助函数：MIDI 编号转音符名称
    function note_name = midi_to_note_name(midi)
        notes = {'C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B'};
        note_idx = mod(midi - 60, 12) + 1; % 以中央 C 为基准
```

```
        note_name = notes{note_idx};
    end
end
```

E. generate_visualizations.m（结果可视化）

```
function generate_visualizations(original_audio, processed_audio, fs, params,
multi_pitch, chord_labels, output_dir)
    % 结果可视化模块
    % 输入:
    %   original_audio - 原始音频信号
    %   processed_audio - 预处理后的音频
    %   fs - 采样率
    %   params - 处理参数
    %   multi_pitch - 多基频矩阵
    %   chord_labels - 和弦标签
    %   output_dir - 输出目录

    %% 检查输出目录是否存在
    if ~exist(output_dir, 'dir')
        error('输出目录 %s 不存在，请检查。', output_dir);
    end

    %% 时间轴
    t = (0:length(original_audio)-1) / fs;
    frame_time = (0:size(multi_pitch,1)-1) * params.hop_length / fs;

    %% 创建一个新的图形窗口，用于绘制音频波形、频谱图和多基频叠加图
    figure('Position', [100, 100, 1200, 1000]);

    %% 绘制原始音频波形
    subplot(4, 1, 1);
    plot(t, original_audio);
    title('原始音频波形');
    xlabel('时间 (秒)');
    ylabel('幅度');
    grid on;

    %% 绘制预处理后的音频波形
    subplot(4, 1, 2);
    plot(t, processed_audio);
    title('预处理后的音频波形');
    xlabel('时间 (秒)');
    ylabel('幅度');
    grid on;

    %% 绘制频谱图
    subplot(4, 1, 3);
    [S, F, T] = spectrogram(processed_audio, params.window, params.overlap,
params.window_length, fs);
    imagesc(T, F, 10*log10(abs(S)));
    axis xy;
    title('频谱图 (dB)');
```

```
    xlabel('时间 (秒)');
    ylabel('频率 (Hz)');
    colorbar;

    %% 绘制多基频叠加图
    subplot(4, 1, 4);
    imagesc(T, F, 10*log10(abs(S)));
    axis xy;
    hold on;

    % 绘制多基频轨迹
    colors = {'r', 'g', 'b', 'm'};
    for i = 1:size(multi_pitch, 2)
        valid_idx = multi_pitch(:, i) > 0;
        plot(frame_time(valid_idx), multi_pitch(valid_idx, i), colors{i}, 'Marker',
'.', 'LineStyle', 'none');
    end

    title('多基频轨迹与时频图叠加');
    xlabel('时间 (秒)');
    ylabel('频率 (Hz)');
    legend('基频 1', '基频 2', '基频 3', '基频 4');
    colorbar;

    % 保存图像
    saveas(gcf, fullfile(output_dir, 'multi_pitch_visualization.png'));

    %% 创建一个新的图形窗口，用于绘制和弦识别结果
    figure('Position', [100, 100, 1200, 200]);
    num_frames = length(chord_labels);

    % 合并连续相同的和弦标签
    merged_chords = {};
    merged_start_times = [];
    merged_end_times = [];
    current_chord = chord_labels{1};
    start_time = frame_time(1);
    for i = 2:num_frames
        if ~strcmp(chord_labels{i}, current_chord)
            merged_chords{end+1} = current_chord;
            merged_start_times(end+1) = start_time;
            merged_end_times(end+1) = frame_time(i-1);
            current_chord = chord_labels{i};
            start_time = frame_time(i);
        end
    end
    % 处理最后一个和弦
    merged_chords{end+1} = current_chord;
    merged_start_times(end+1) = start_time;
```

```
    merged_end_times(end+1) = frame_time(end);

    % 绘制合并后的和弦标签
    for i = 1:length(merged_chords)
        mid_time = (merged_start_times(i) + merged_end_times(i)) / 2;
        text(mid_time, 0.5, merged_chords{i}, 'HorizontalAlignment', 'center',
'VerticalAlignment', 'middle');
        % 绘制矩形框表示和弦持续时间
        rectangle('Position', [merged_start_times(i), 0,
merged_end_times(i)-merged_start_times(i), 1], 'EdgeColor', 'k', 'LineStyle', '--');
    end

    xlim([frame_time(1), frame_time(end)]);
    ylim([0, 1]);
    title('和弦识别初始结果');
    xlabel('时间 (秒)');
    yticklabels([]);
    grid on;

    % 保存图像
    saveas(gcf, fullfile(output_dir, 'chord_recognition.png'));
end
```

F. GUI.htm（可视化分析）

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>音频和弦时间轴可视化工具</title>
    <style>
       (略)
    </style>
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css">
</head>
<body>
    <div class="container">
        <header>
            <h1><i class="fas fa-wave-square"></i> 音频和弦时间轴可视化工具</h1>
            <p class="subtitle">智能分析和弦序列，可视化展示音频帧中的和弦变化</p>

            <div class="stats-bar">
                <div class="stat-item">
                    <div>和弦识别</div>
                    <div class="stat-value">等待加载数据</div>
                </div>
            </div>
        </header>

        <div class="upload-section">
            <input type="file" id="csvFile" class="file-input" accept=".csv">
            <label for="csvFile" class="file-label">
                <i class="fas fa-file-upload"></i> 选择 CSV 文件
            </label>
            <p id="fileName">未选择文件</p>
            <div class="progress-bar" id="progressBar">
                <div class="progress-fill" id="progressFill"></div>
            </div>
        </div>

        <div class="visualization-section">
            <h2 class="section-title">
                <i class="fas fa-sliders-h"></i> 和弦时间轴
            </h2>
            <div id="timelineContainer" class="timeline-container">
                <div id="timeline" class="timeline"></div>
                <div id="timeScale" class="time-scale"></div>
            </div>

            <div id="legend" class="legend"></div>
```

```
        </div>

        <div class="instructions">
            <h2><i class="fas fa-info-circle"></i> 使用说明</h2>
            <ul>
                <li>上传一个只包含一行数据的 CSV 文件，每个元素代表一个音频帧对应的和弦
</li>
                <li>连续相同的和弦会自动合并，并在时间轴上显示为更宽的区块</li>
                <li>不同和弦使用不同颜色区分，鼠标悬停可查看详细信息</li>
                <li>时间轴下方显示时间刻度（秒）</li>
            </ul>
        </div>

        <footer>
            <p>2025 信号分析与处理 | 音频和弦可视化工具 | 周钰 翁超然 陈若涵 邹依轩</p>
        </footer>
    </div>

    <script>
        document.addEventListener('DOMContentLoaded', function() {
            const csvFileInput = document.getElementById('csvFile');
            const fileNameDisplay = document.getElementById('fileName');
            const timeline = document.getElementById('timeline');
            const timeScale = document.getElementById('timeScale');
            const legend = document.getElementById('legend');
            const loadSample1Btn = document.getElementById('loadSample1');
            const loadSample2Btn = document.getElementById('loadSample2');
            const progressBar = document.getElementById('progressBar');
            const progressFill = document.getElementById('progressFill');

            // 和弦到颜色的映射
            const chordColors = {};
            const colorPalette = [
                '#FF6B6B', '#4ECDC4', '#FFD166', '#118AB2', '#06D6A0',
                '#EF476F', '#FFC43D', '#1B9AAA', '#9B5DE5', '#F15BB5',
                '#00BBF9', '#00F5D4', '#FEE440', '#9B5DE5', '#00F5D4',
                '#FF7E5F', '#8A2BE2', '#20B2AA', '#FF6347', '#7B68EE'
            ];

            // 帧时长（毫秒）
            const frameDuration = 42.67;

            // 文件选择处理
            csvFileInput.addEventListener('change', function(e) {
                const file = e.target.files[0];
                if (file) {
                    fileNameDisplay.textContent = `已选择: ${file.name}`;
                    showProgressBar();
                    setTimeout(() => {
```

```
                    processFile(file);
                    hideProgressBar();
                }, 600);
            }
        });

        // 显示进度条
        function showProgressBar() {
            progressBar.style.display = 'block';
            progressFill.style.width = '0%';

            setTimeout(() => {
                progressFill.style.width = '30%';
            }, 100);

            setTimeout(() => {
                progressFill.style.width = '70%';
            }, 300);

            setTimeout(() => {
                progressFill.style.width = '100%';
            }, 500);
        }

        // 隐藏进度条
        function hideProgressBar() {
            setTimeout(() => {
                progressBar.style.display = 'none';
            }, 800);
        }

        // 加载示例数据 1
        loadSample1Btn.addEventListener('click', function() {
            fileNameDisplay.textContent = "已加载: 示例数据 1";
            const sampleData = "C,C,未知,C,Am,Am,G,G,未知,G,Em,Em,Dm,Dm,C,C,未知,
未知,G,G,Dm,Dm,未知,Dm,Dm,G,G,C,C,Em,Em,Am,Am,F,F,Dm,Dm,G,G";
            showProgressBar();
            setTimeout(() => {
                processCSVData(sampleData);
                hideProgressBar();
            }, 600);
        });

        // 加载示例数据 2
        loadSample2Btn.addEventListener('click', function() {
            fileNameDisplay.textContent = "已加载: 示例数据 2";
            const sampleData = "F,未知,未知,G,Em,Am,未知,Dm,G,C,未知,Am,未知,F,未
知,C,G,未知,Em,Am,Dm,未知,G,未知,C,Am,F,未知,Dm,G,未知";
            showProgressBar();
```

```
                setTimeout(() => {
                    processCSVData(sampleData);
                    hideProgressBar();
                }, 600);
            });

            // 处理上传的文件
            function processFile(file) {
                const reader = new FileReader();
                reader.onload = function(e) {
                    processCSVData(e.target.result);
                };
                reader.readAsText(file);
            }

            // 处理 CSV 数据
            function processCSVData(csvData) {
                // 清除之前的内容
                timeline.innerHTML = '';
                timeScale.innerHTML = '';
                legend.innerHTML = '';

                // 解析 CSV 数据（假设只有一行）
                let chords = csvData.split(',').map(chord => chord.trim());

                // 处理"未知"和弦
                chords = preprocessUnknownChords(chords);

                // 合并连续相同的和弦
                const mergedChords = mergeConsecutiveChords(chords);

                // 计算总时间
                const totalDuration = mergedChords.reduce((sum, chord) => sum +
chord.duration, 0);

                // 更新统计信息
                updateStats(mergedChords.length, chords.length, totalDuration);

                // 生成和弦块
                createChordBlocks(mergedChords, totalDuration);

                // 生成时间刻度
                createTimeScale(totalDuration);

                // 生成图例
                createLegend();
            }

            // 预处理：处理"未知"和弦
```

```
function preprocessUnknownChords(chords) {
    // 创建新数组以避免修改原始数组
    const processedChords = [...chords];

    for (let i = 0; i < processedChords.length; i++) {
        if (processedChords[i] === "未知") {
            if(i == 0)
                processedChords[i] = processedChords[i+1];
            else
                processedChords[i] = processedChords[i-1];
        }
    }
    return processedChords;
}

// 合并连续相同的和弦
function mergeConsecutiveChords(chords) {
    const merged = [];
    let currentChord = chords[0];
    let count = 1;

    for (let i = 1; i < chords.length; i++) {
        if (chords[i] === currentChord) {
            count++;
        } else {
            merged.push({
                chord: currentChord,
                duration: count * frameDuration,
                frames: count
            });

            currentChord = chords[i];
            count = 1;
        }
    }

    // 添加最后一个和弦
    merged.push({
        chord: currentChord,
        duration: count * frameDuration,
        frames: count
    });

    return merged;
}

// 更新统计信息
function updateStats(mergedCount, totalFrames, totalDuration) {
    const statsBar = document.querySelector('.stats-bar');
```

```
                statsBar.innerHTML = `
                    <div class="stat-item">
                        <div>和弦区块</div>
                        <div class="stat-value">${mergedCount}</div>
                    </div>
                    <div class="stat-item">
                        <div>总帧数</div>
                        <div class="stat-value">${totalFrames}</div>
                    </div>
                `;
            }

            // 创建和弦块
            function createChordBlocks(chords, totalDuration) {
                chords.forEach(chordData => {
                    const chord = chordData.chord;
                    const duration = chordData.duration;

                    // 计算和弦块的宽度（占总宽度的百分比）
                    const widthPercent = (duration / totalDuration) * 100;

                    // 获取和弦颜色
                    const color = getChordColor(chord);

                    // 创建和弦块元素
                    const block = document.createElement('div');
                    block.className = 'chord-block';
                    block.style.width = `${widthPercent}%`;
                    block.style.backgroundColor = color;
                    block.title = `和弦: ${chord}\n持续时间: ${duration.toFixed(2)}ms\n
帧数: ${chordData.frames}`;

                    // 添加和弦信息
                    const info = document.createElement('div');
                    info.className = 'chord-info';
                    info.textContent = `${chordData.frames}帧
(${duration.toFixed(0)}ms)`;
                    block.appendChild(info);

                    // 添加和弦标签
                    const label = document.createElement('div');
                    label.className = 'chord-label';
                    label.textContent = chord;
                    block.appendChild(label);

                    timeline.appendChild(block);
                });
            }
```

```
            // 获取和弦颜色
            function getChordColor(chord) {
                if (chord === "未知") {
                    return '#777777'; // 未知和弦使用灰色
                }

                if (!chordColors[chord]) {
                    // 为新的和弦分配颜色
                    const colorIndex = Object.keys(chordColors).length %
colorPalette.length;
                    chordColors[chord] = colorPalette[colorIndex];
                }
                return chordColors[chord];
            }

            // 创建时间刻度
            function createTimeScale(totalDuration) {
                // 计算总秒数
                const totalSeconds = totalDuration / 1000;

                // 计算刻度间隔（每 500 毫秒一个刻度）
                const tickInterval = totalDuration / 10 ;
                const tickCount = Math.ceil(totalDuration / tickInterval);

                for (let i = 0; i <= tickCount; i++) {
                    const time = i * tickInterval;
                    const positionPercent = (time / totalDuration) * 100;

                    // 创建刻度线
                    const tick = document.createElement('div');
                    tick.className = 'time-tick';
                    tick.style.left = `${positionPercent}%`;

                    // 创建时间标签
                    const label = document.createElement('div');
                    label.className = 'time-label';
                    label.style.left = `${positionPercent}%`;
                    label.textContent = `${(time / 1000).toFixed(1)}s`;

                    timeScale.appendChild(tick);
                    timeScale.appendChild(label);
                }
            }

            // 创建图例
            function createLegend() {
                // 收集所有不同的和弦（包括未知）
                const uniqueChords = [...new Set(Object.keys(chordColors).concat("未
知"))];
```

```
                uniqueChords.forEach(chord => {
                    const item = document.createElement('div');
                    item.className = 'legend-item';

                    const colorBox = document.createElement('div');
                    colorBox.className = 'legend-color';
                    colorBox.style.backgroundColor = getChordColor(chord);

                    const label = document.createElement('span');
                    label.textContent = chord;

                    item.appendChild(colorBox);
                    item.appendChild(label);
                    legend.appendChild(item);
                });
            }
        });
    </script>
</body>
</html>
```