语音信号预处理

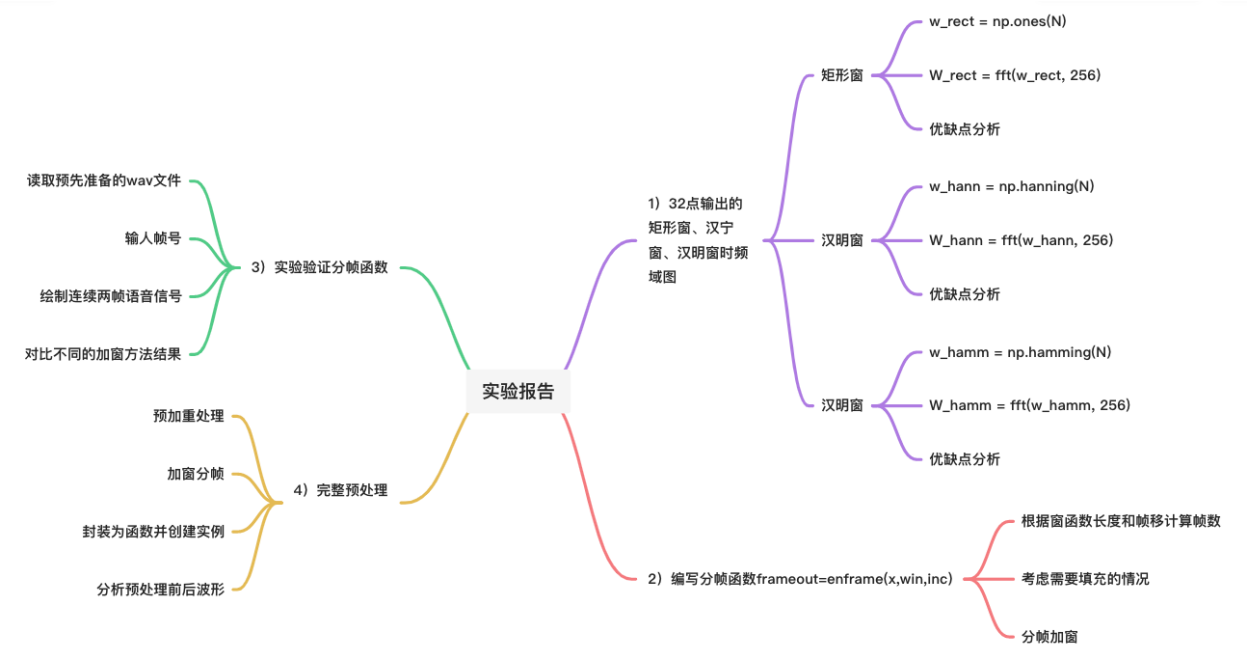
Reviewed ☐

实验报告

实验进展：

已完成1)-6)所有实验内容

实验思维导图：



（实验最后的 5)6)部分在思维导图工作完成后布置，故没有呈现在导图中）

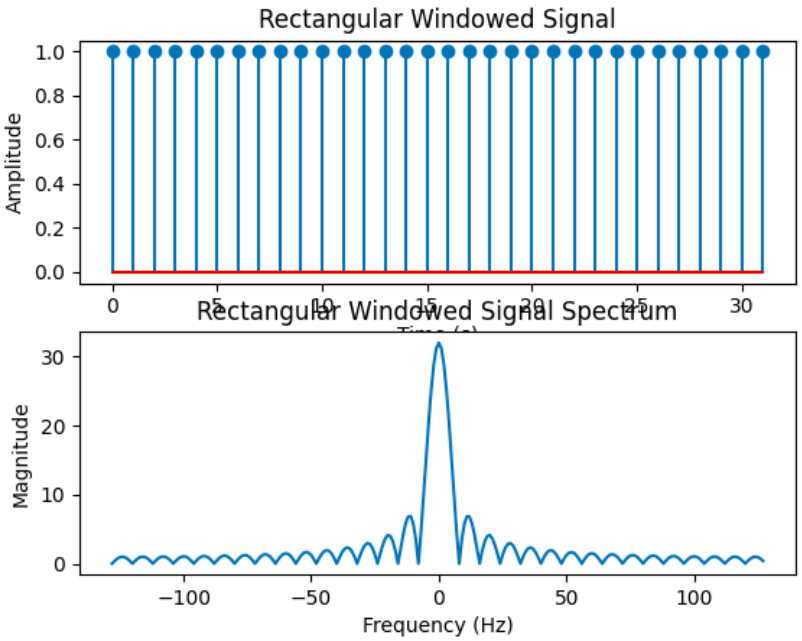
实验内容：

> **1）完成32点输出的矩形窗、汉宁窗、汉明窗时频域图，并对三个窗进行分析。**

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft

# 定义窗
N = 32  # 32点
nn = np.arange(0, N)  # 时域横坐标范围
NN = np.arange(-128, 128)  # 频域横坐标范围
```

```
# 矩形窗
'''
矩形窗是使用 np.ones 函数生成的。
该函数返回一个全为1的数组，其形状与信号 x 相同。表示在信号中选取一个连续的时间窗口，其幅度为1。
'''

w_rect = np.ones(N)  # 矩形窗函数在相应序列点时域上的幅度响应
W_rect = fft(w_rect, 256)  # 矩形窗函数在相应序列点频域上的幅度响应
plt.subplot(2,1,1)
plt.stem(nn,w_rect)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Rectangular Windowed Signal')

plt.subplot(2,1,2)
plt.plot(NN, abs(np.fft.fftshift(W_rect)))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Rectangular Windowed Signal Spectrum')
plt.show()

# 汉宁窗
'''
使用 numpy 库中的 hanning 函数生成汉宁窗。
'''
w_hann = np.hanning(N)  # 汉宁窗函数在相应序列点时域上的幅度响应
W_hann = fft(w_hann, 256)  # 汉宁窗函数在相应序列点频域上的幅度响应
plt.subplot(2,1,1)
plt.stem(nn,w_hann)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Hanning Windowed Signal')

plt.subplot(2,1,2)
plt.plot(NN, abs(np.fft.fftshift(W_hann)))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Hanning Windowed Signal Spectrum')
plt.show()

# 汉明窗
'''
使用 numpy 库中的 hamming 函数生成汉明窗。
'''
w_hamm = np.hamming(N)  # 汉明窗函数在相应序列点时域上的幅度响应
W_hamm = fft(w_hamm, 256)  # 汉明窗函数在相应序列点频域上的幅度响应
plt.subplot(2,1,1)
plt.stem(nn,w_hamm)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Hamming Windowed Signal')

plt.subplot(2,1,2)
plt.plot(NN, abs(np.fft.fftshift(W_hamm)))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Hamming Windowed Signal Spectrum')
plt.show()
```

1. 矩形窗



矩形窗在时域上的表示为一段常数值为1的矩形信号，而在频域上则是一个包含多个峰值的线性谱。

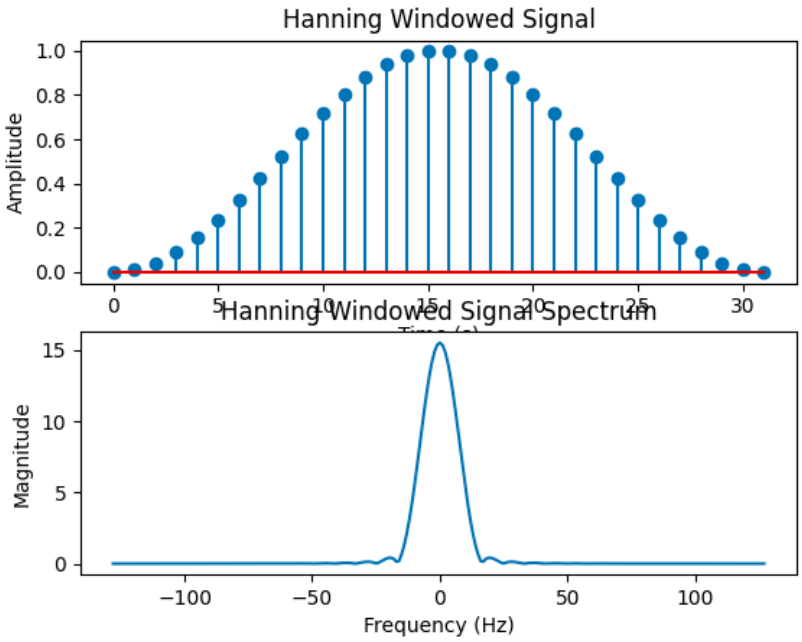
时域上：矩形窗在时域上的坡度为矩形函数，在窗口的两端突然变化，窗口内部变化率为0。

频域上：矩形窗具有较好的分辨率，但容易产生频谱泄漏：频域上矩形窗的主瓣宽度最宽，副瓣能量相对较强。由于其能量在频域上分布比较平均，无法将信号的高频部分和低频部分有效地区分开来，从而导致频谱分析的精度和准确性降低。因此在频谱分析中容易出现频谱泄漏现象。

矩形窗的优点是计算简单，实现容易，但是需要注意在使用时可能会引起频谱泄露问题。

> 主瓣是指频域上最大幅值的成分，而旁瓣则是指在主瓣附近出现的小幅成分。

2. 汉宁窗

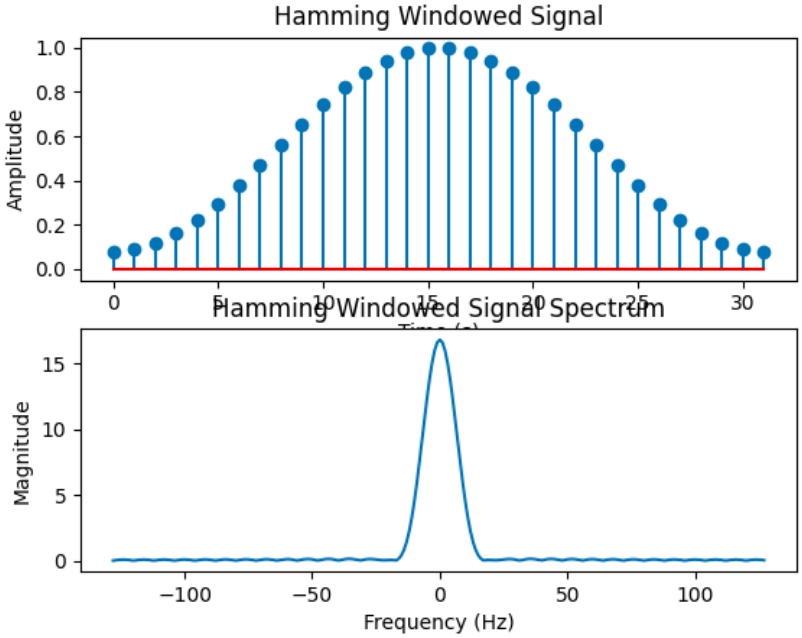


时域上：为一个对称的带有汉宁余弦函数的窗口，是平滑的窗口函数，其坡度较小，能够更好的保留信号的截止和平滑特性。

频域上：主瓣相对比较窄，副瓣能量相对较小，能够有效的减少频谱泄露问题，提高频谱分析的精度和准确性。

汉宁窗的缺点是其副瓣衰减速度较慢，需要更高的动态范围。

3. 汉明窗



时域上：为一个对称的带有汉明函数的窗口，是平滑的窗口函数，其坡度较小，能够更好的保留信号的截止和平滑特性。

频域上：汉明窗和汉宁窗在主瓣和旁瓣方面相似，主瓣相对比较窄，副瓣能量相对较小，能够有效的减少频谱泄露。但是汉明窗的旁瓣衰减速度比汉宁窗更快，能够在更低的动态范围内实现高精度的频谱分析。

> **2）根据语音分帧的思想，编写分帧函数。函数定义如下：**
> **函数格式：frameout=enframe(x,win,inc)**
> **输入参数：x 是语音信号；win 是窗函数，若为窗函数，帧长便取窗函数长；inc是帧移。**
> **输出参数：frameout是分帧后的数组，长度为帧长和帧数的乘积**

`enframe(x,win,inc)函数`

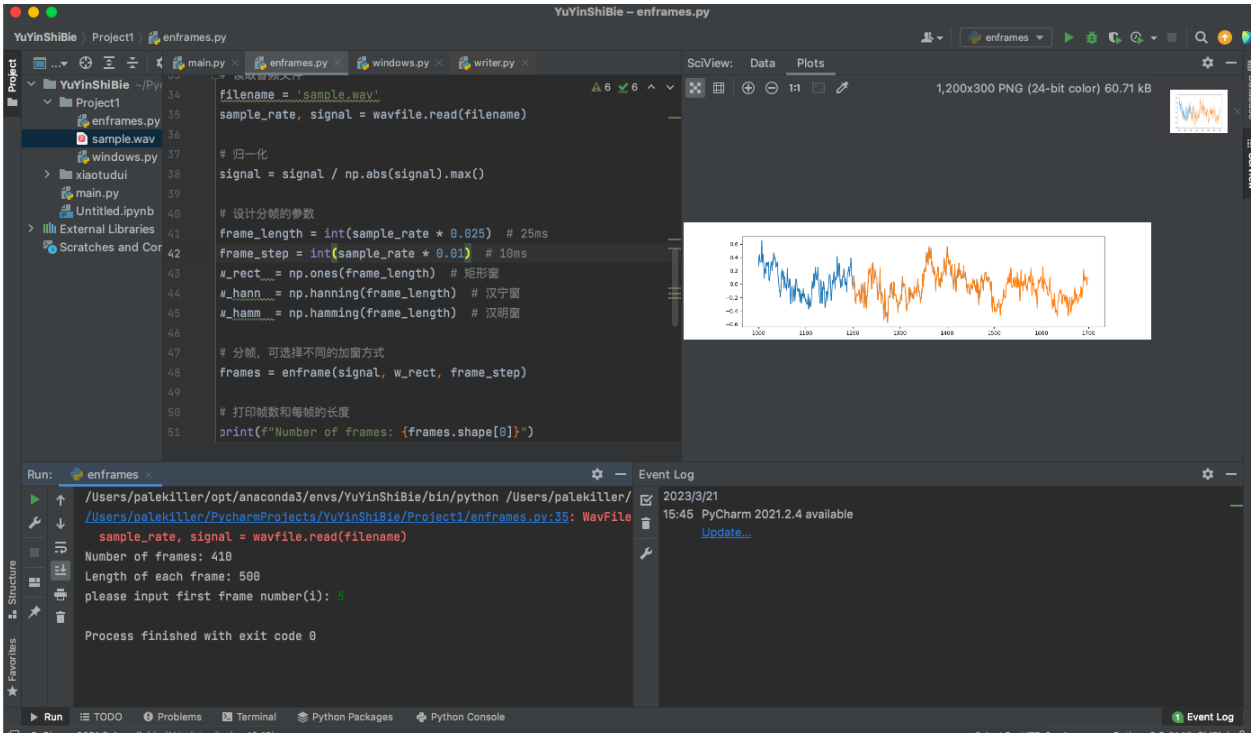
```
import numpy as np

def enframe(x, win, inc):
    """
    将语音信号分帧，并加窗，返回分帧后的数组
    :param x: 语音信号
    :param win: 窗函数
    :param inc: 帧移
    :return: 分帧后的数组，大小为(nframes, len(win))
    """
    x_len = len(x)  # 语音信号的长度
    win_len = len(win)  # 窗函数的长度：在每一帧中，我们需要选择 win_len 个样本点，并对它们进行加窗处理

    # 根据窗函数长度和帧移计算帧数
    frame_num = int(np.ceil((x_len - win_len + inc) / inc))  # 计算分帧后的帧数：np.ceil()对数组进行向上取整【分帧后的帧数可能不是整数】

    pads = ((frame_num - 1) * inc + win_len - x_len)  # 计算需要填充的长度，保证最后一帧长度为win的长度（填充信号长度，使其刚好可以被inc整除）
    if pads > 0:  # 如果需要填充，则在信号两端进行对称填充
        x = np.pad(x, (0, pads), 'constant')

    # 分帧
    frameout = np.zeros((frame_num, win_len))  # 定义一个全0数组，用于存储分帧后的结果
    for i in range(frame_num):  # 依次对原始语音信号进行分帧
        frameout[i] = x[i * inc:i * inc + win_len] * win  # 计算第i帧的结果，并乘以窗函数win

    return frameout  # 返回分帧后的结果数组
```

> **3）实验验证：自己录制1秒语音：根握分帧的语音，输人帧号(或者自己选定一个帧号），绘制连续两帧语音信号**
> **•实验验证提示：**
> **•自行设计窗长和帧移的点数**
> **•读取wav文件scipy.io. wavfile.read()**
> **•音频需要归一化：最大值：np.absolute(wave data).max0**
> **•画第帧的输入提示：i=input('please input first frame number(i)")**

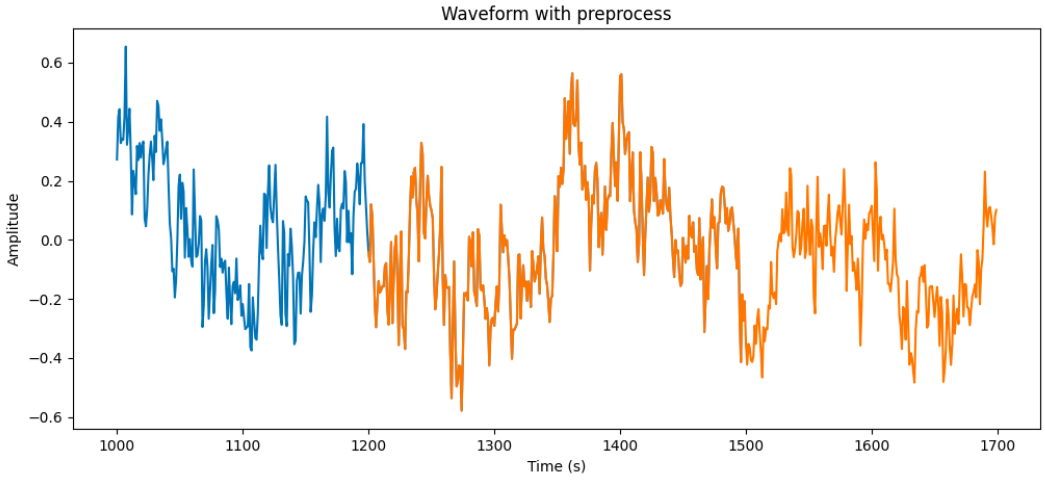
实例：录制语音sample.wav

```
# 实例
from scipy.io import wavfile
import matplotlib.pyplot as plt

# 读取音频文件
filename = 'sample.wav'
sample_rate, signal = wavfile.read(filename)

# 归一化
signal = signal / np.abs(signal).max()

# 设计分帧的参数
frame_length = int(sample_rate * 0.025)  # 25ms
frame_step = int(sample_rate * 0.01)  # 10ms
w_rect  = np.ones(frame_length)  # 矩形窗
w_hann  = np.hanning(frame_length)  # 汉宁窗
w_hamm  = np.hamming(frame_length)  # 汉明窗

# 分帧，可选择不同的加窗方式
frames = enframe(signal, w_hann, frame_step)

# 打印帧数和每帧的长度
print(f"Number of frames: {frames.shape[0]}")
print(f"Length of each frame: {frames.shape[1]}")

# 选择某帧和后一帧，并绘制波形图
frame_idx = int(input('please input first frame number(i): '))
'''
在绘制连续两帧语音信号的波形图时，可以根据自己的需要选择要显示的帧号，
即输入 i 的值，尝试不同的帧号，以观察连续两帧语音信号的差异。
'''
plt.figure(figsize=(12, 3))
# frame_idx帧的波形图，frame_idx表示帧号，frame_step表示帧移，frame_length表示帧长
plt.plot(np.arange(frame_idx*frame_step, frame_idx*frame_step+frame_length), frames[frame_idx])
'''
np.arange()生成了表示时间轴的数组，
该数组的起始值是frame_idx*frame_step，步长是1，长度是frame_length.
frames[frame_idx]表示第frame_idx帧的语音信号。
'''
# frame_idx+1帧的波形图
plt.plot(np.arange((frame_idx+1)*frame_step, (frame_idx+1)*frame_step+frame_length), frames[frame_idx+1])  # 与上同理
plt.show()
```

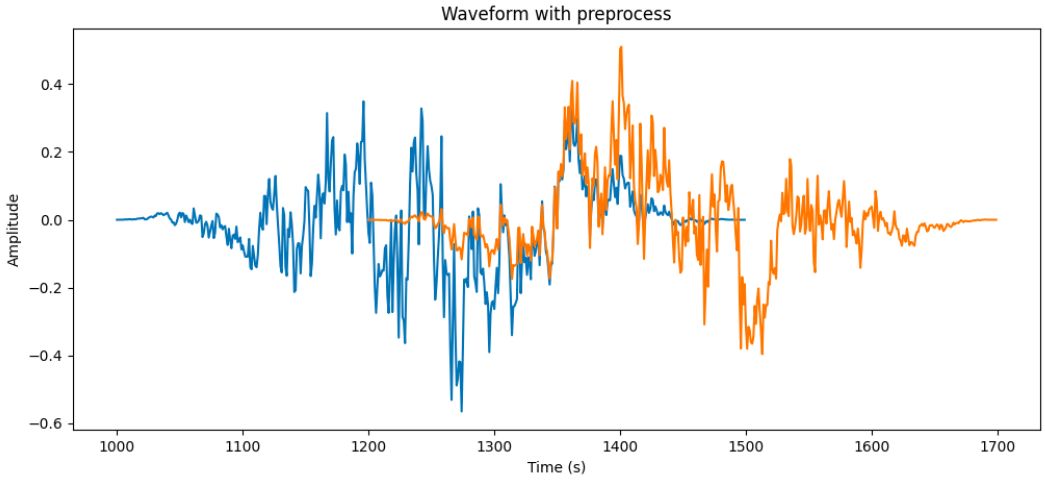
实例：选择第五帧和第六帧。图中蓝色代表第一帧，橙色代表第二帧。


YuYinShiBie – enframes.py
filename = 'sample.wav'
sample_rate, signal = wavfile.read(filename)
# 归一化
signal = signal / np.abs(signal).max()
# 设计分帧的参数
frame_length = int(sample_rate * 0.025) # 25ms
frame_step = int(sample_rate * 0.01) # 10ms
w_rect = np.ones(frame_length) # 矩形窗
w_hann = np.hanning(frame_length) # 汉宁窗
w_hamm = np.hamming(frame_length) # 汉明窗
# 分帧，可选择不同的加窗方式
frames = enframe(signal, w_rect, frame_step)
# 打印帧数和每帧的长度
print(f"Number of frames: {frames.shape[0]}")
SciView: Data Plots
1,200x300 PNG (24-bit color) 60.71 kB
Run: enframes
/Users/palekiller/opt/anaconda3/envs/YuYinShiBie/bin/python /Users/palekiller/
/Users/palekiller/PycharmProjects/YuYinShiBie/Project1/enframes.py:35: WavFile
sample_rate, signal = wavfile.read(filename)
Number of frames: 410
Length of each frame: 500
please input first frame number(i): 8
Process finished with exit code 0
Event Log
2023/3/21
15:45 PyCharm 2021.2.4 available
Update...

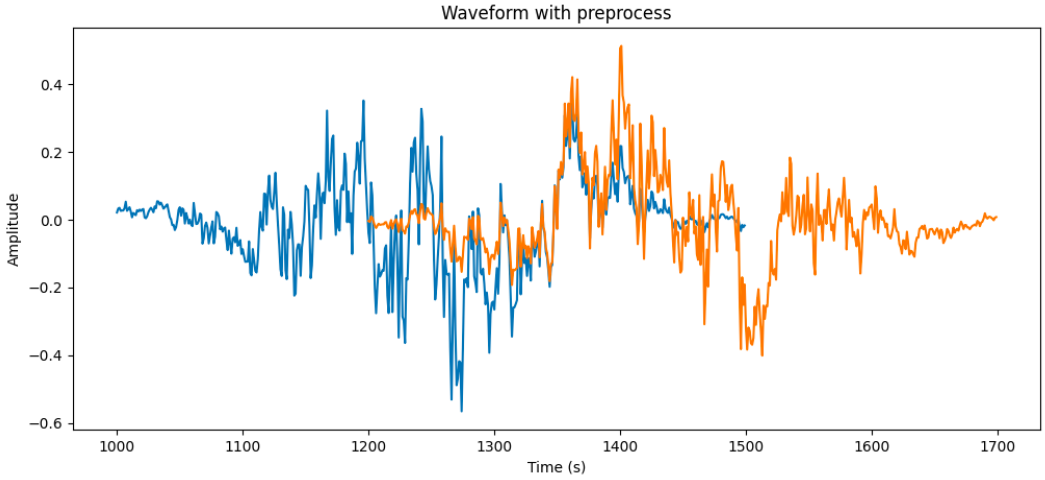
矩形窗


Waveform with preprocess
Amplitude
Time (s)
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
1000
1100
1200
1300
1400
1500
1600
1700

汉宁窗



汉明窗



> **4）以上内容完成是第一阶段，余下是第二阶段：**
> **进行预加重处理**
> **进行加窗（例如汉宁窗）处理**
> **将预加重、分帧、加窗封装为函数**
> **比较分析预加重、加窗步骤前后波形的变化**

1. **预加重处理**

（1）预加重是一种对语音信号进行预处理的方法，用于增强高频分量，减小低频分量，以改善信号质量和语音识别性能。在进行分帧之前，一般需要对语音信号进行预加重处理。

（2）预加重的过程是对原始语音信号x(t)做一个差分运算，得到差分语音信号y(t)，具体计算公式为：

$$y(t) = x(t) - \alpha * x(t-1)$$

其中，α是预加重系数，一般取值为0.95。

```
import numpy as np

def preemphasis(signal, alpha=0.95):
    """预加重"""
    return np.append(signal[0], signal[1:]-alpha*signal[:-1])
```

2. **加窗（例如汉宁窗）分帧处理**

   （1）**加窗**

   加窗是一种在信号分帧时常用的处理方法。由于原始信号在每一帧的两端存在不连续的现象，为了消除这种不连续性，我们需要对每一帧进行加窗处理，消除信号边缘截断引起的频率泄漏问题。窗函数的选择对语音信号处理有较大的影响，不同的窗函数有着不同的性质和应用场景，不同窗函数特性已在上文详细说明。

   （2）**分帧**

   在进行分帧处理时，通常将语音信号按照固定的帧长进行切分，切分后的每一帧通常是重叠的，即前一帧的最后一部分和下一帧的开头部分重叠。这样做的目的是为了避免在帧边界处产生不连续的信号，同时也便于后续的处理和分析。

```
def enframe(x, win, inc):
    """
    将语音信号分帧，并加窗，返回分帧后的数组
    :param x: 语音信号
    :param win: 窗函数
    :param inc: 帧移
    :return: 分帧后的数组，大小为(nframes, len(win))
    """
    x_len = len(x)  # 语音信号的长度
    win_len = len(win)  # 窗函数的长度：在每一帧中，我们需要选择 win_len 个样本点，并对它们进行加窗处理

    # 根据窗函数长度和帧移计算帧数
    frame_num = int(np.ceil((x_len - win_len + inc) / inc))  # 计算分帧后的帧数：np.ceil()对数组进行向上取整【分帧后的帧数可能不是整数】

    pads = ((frame_num - 1) * inc + win_len - x_len)  # 计算需要填充的长度，保证最后一帧长度为win的长度（填充信号长度，使其刚好可以被inc整除）
    if pads > 0:  # 如果需要填充，则在信号两端进行对称填充
        x = np.pad(x, (0, pads), 'constant')

    # 分帧
    frameout = np.zeros((frame_num, win_len))  # 定义一个全0数组，用于存储分帧后的结果
    for i in range(frame_num):  # 依次对原始语音信号进行分帧
        frameout[i] = x[i * inc:i * inc + win_len] * win  # 计算第i帧的结果，并乘以窗函数win

    return frameout  # 返回分帧后的结果数组
```

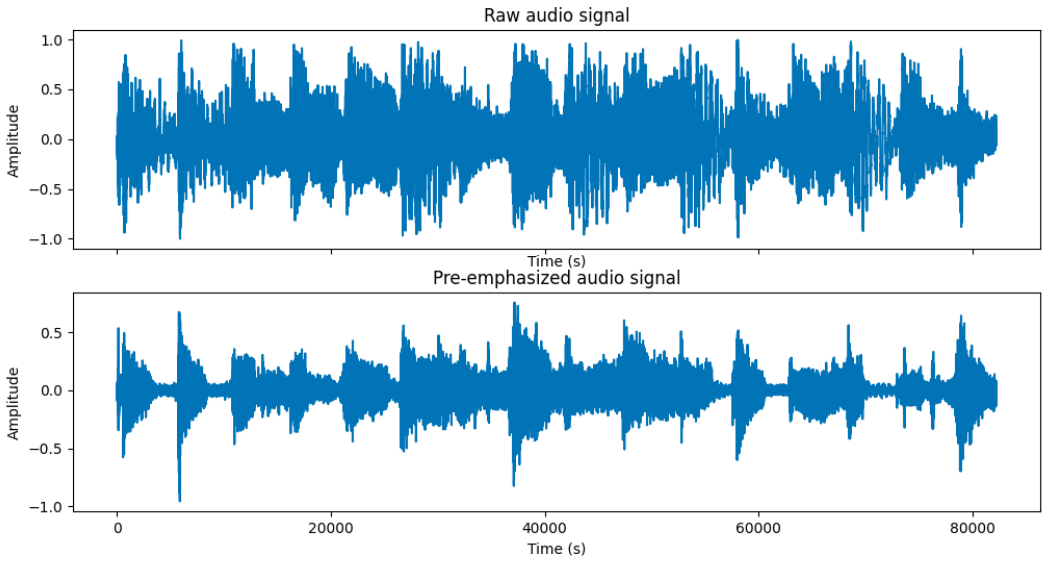
实例：

```
# 实例
# 读取音频文件
filename = 'sample.wav'
sample_rate, signal = wavfile.read(filename)

# 归一化
signal = signal / np.abs(signal).max()

# 预加重
pre_emphasis = preemphasis(signal)

# 设计分帧的参数
frame_length = int(sample_rate * 0.025)  # 25ms
frame_step = int(sample_rate * 0.01)  # 10ms
w_rect  = np.ones(frame_length)  # 矩形窗
w_hann  = np.hanning(frame_length)  # 汉宁窗
w_hamm  = np.hamming(frame_length)  # 汉明窗
```

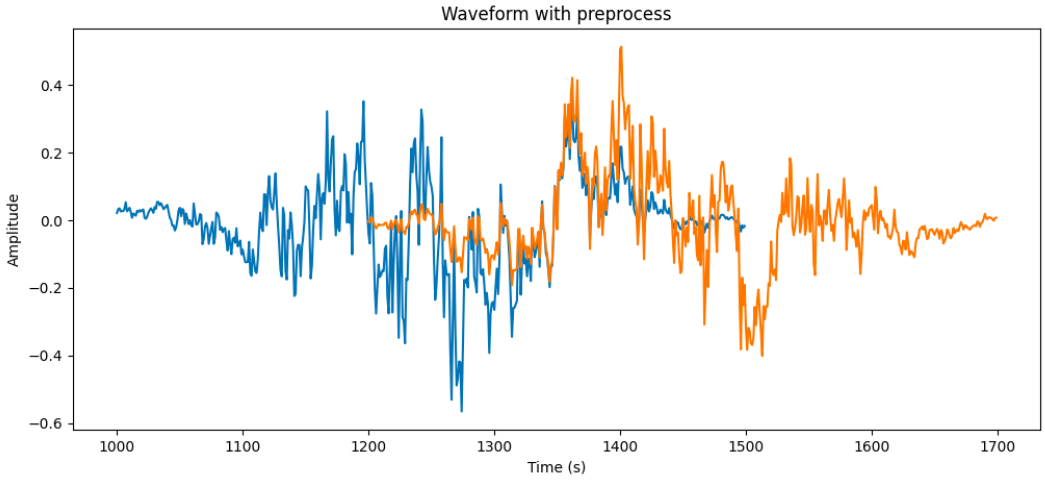
```
# 分帧，可选择不同的加窗方式
frames = enframe(signal, w_hamm, frame_step)

# 绘制原始语音信号和预加重信号
fig, axs = plt.subplots(nrows=2, sharex=True, figsize=(12,6))
axs[0].plot(signal)
axs[0].set_title('Raw audio signal')
axs[1].plot(pre_emphasis)
axs[1].set_title('Pre-emphasized audio signal')

# 打印帧数和每帧的长度
print(f"Number of frames: {frames.shape[0]}")
print(f"Length of each frame: {frames.shape[1]}")

# 选择某帧和后一帧，并绘制波形图
frame_idx = int(input('please input first frame number(i): '))
plt.figure(figsize=(12, 3))
# frame_idx帧的波形图，frame_idx表示帧号，frame_step表示帧移，frame_length表示帧长
plt.plot(np.arange(frame_idx*frame_step, frame_idx*frame_step+frame_length), frames[frame_idx])
# frame_idx+1帧的波形图
plt.plot(np.arange((frame_idx+1)*frame_step, (frame_idx+1)*frame_step+frame_length), frames[frame_idx+1])  # 与上同理
plt.show()
```

选择第五帧和第六帧。图中蓝色代表第一帧，橙色代表第二帧。

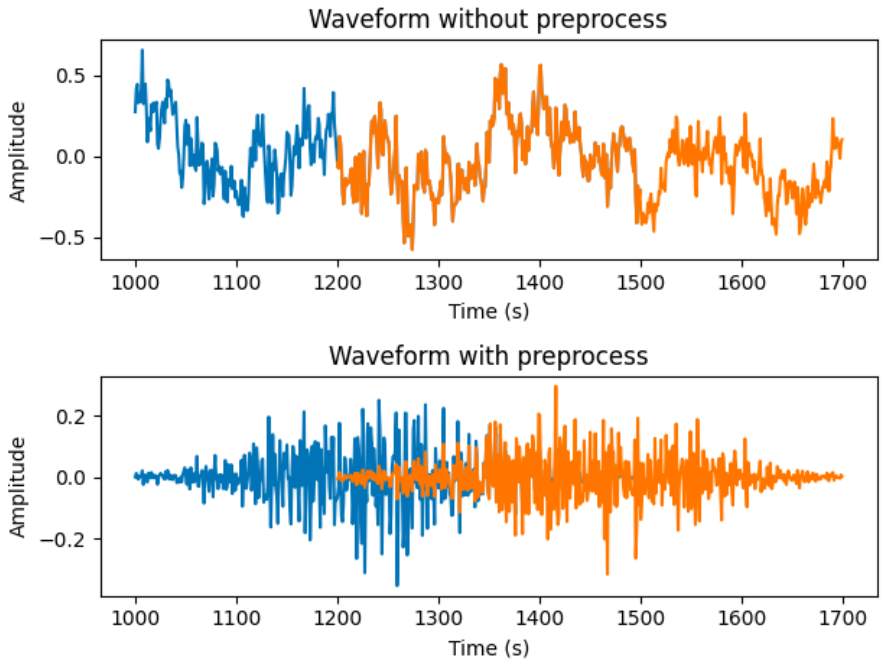
预加重



无预加重处理 直接分帧加窗（此处使用 汉明窗）



预加重、加窗分帧步骤前后波形的变化比较

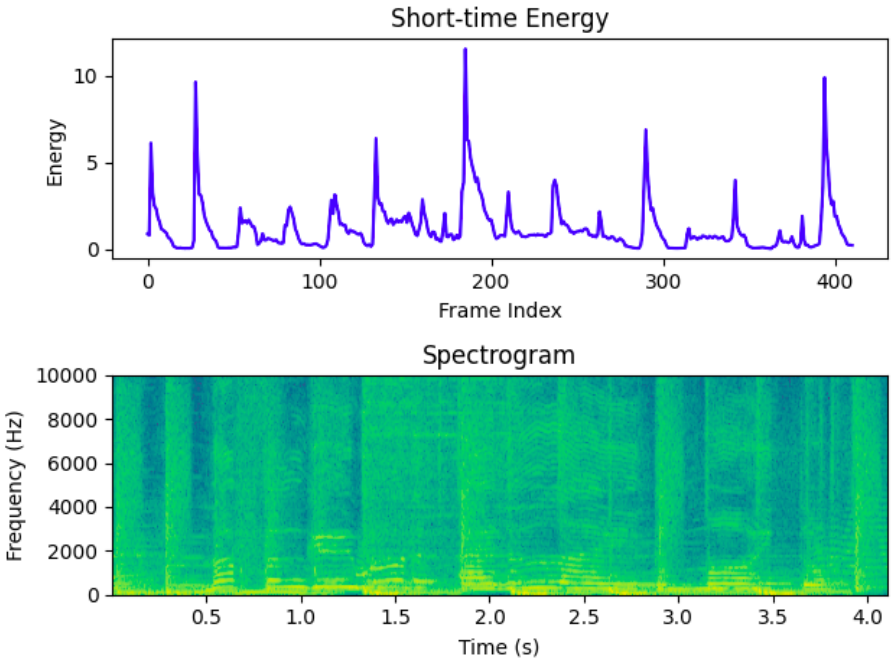


通过 预加重、加窗分帧步骤 ：

- 因为预加重增强高频部分信号的能量，经过预处理后的波形高频部分信号更加明显。同时降低低频信号的能量，减少波形中低频的部分。
- 整体使信号波形变得更平滑，更加连续。这是因为加窗减少不连续性和频谱泄漏。
- 改变波形中信号的能量分布，从而使信号更符合所需的频率特性。
- 减少信号端点处的影响，波形中可观察到减少了信号的过渡效应。

5）调用归一化、分帧等函数，继续实现：
短时能量求取及画图分析
语谱图的绘制及分析

```
# please input first frame number(i): 55
# 短时能量求取及画图分析  &  语谱图的绘制及分析
import numpy as np
import librosa.display
import matplotlib.pyplot as plt
from scipy import signal
from YujiazhongFenzhenWin import preemphasis,enframe


# 短时能量求取及画图分析
# 加载音频文件
filename = 'sample.wav'
y, sr = librosa.load(filename)  # 将音频信号y和采样率sr分别赋值
# 预加重
y_pre = preemphasis(y)
# 设置分帧的参数，其中窗长为25ms，帧移为10ms
win_len = int(sr * 0.025)  # 窗长，25ms
inc = int(sr * 0.01)  # 帧移，10ms
win = np.hamming(win_len)  # 加窗函数
# 使用YujiazhongFenzhenWin中自己编写的分帧函数enframe将信号分帧，并加窗
frames = enframe(y_pre, win, inc)
# 计算每一帧的短时能量
energy = np.sum(frames ** 2, axis=1)
# 画出短时能量图
plt.subplot(2, 1, 1)  # 创建一个带有两个子图的画布，第一个子图
plt.plot(energy, color='blue')  # 绘制能量曲线
plt.ylabel('Energy')  # 设置y轴标签
plt.xlabel('Frame Index')  # 设置x轴标签
plt.title('Short-time Energy')  # 设置图标题


# 语谱图的绘制及分析
# 打开音频文件，获取音频文件参数
f = wave.open(r"sample.wav", "rb")
params = f.getparams()  # 获取音频文件的参数
nchannels, sampwidth, framerate, nframes = params[:4]  # 声道数、每个样本点的字节数、采样率和样本点数

# 读取音频文件数据并进行归一化处理
str_data = f.readframes(nframes)  # 读取音频数据，返回一个字节字符串
wave_data = np.fromstring(str_data, dtype=np.short)
wave_data = wave_data*1.0/(max(abs(wave_data)))  # 归一化处理

plt.subplot(2, 1, 2)
plt.specgram(wave_data,Fs = framerate, scale_by_freq = True, sides = 'default')
'''
其中 wave_data 为归一化后的音频数据，
Fs 为采样率，scale_by_freq 表示是否按比例缩放频谱，
sides 为频谱的取值范围，
'default' 表示频谱的左右两边均绘制
'''
plt.ylabel('Frequency (Hz)')  # 设置y轴标签
plt.xlabel('Time (s)')  # 设置x轴标签
plt.title('Spectrogram')  # 设置图标题
plt.tight_layout()
plt.show()  # 显示图像
```



1. **短时能量分析**

短时能量是指每一帧音频信号的平方和，用来表示该帧的能量大小。

通过计算短时能量可以对音频信号的能量变化进行分析，通常可以用于检测音频中的起止点和语音活动检测等任务。

短时能量图像可以帮助我们找到每个单词的起始和结束位置，以及语音中的停顿和重音位置。上述结果可以推测出说话重音主要位于第35帧、第190帧、第400帧附近。也可以基本推算该音频片段的说话停顿和语速。

同时如果有一些噪音或其他干扰因素，我们也可以在短时能量图像中看到它们的影响，从而进行去噪或其他后处理操作。

2. **语谱图分析**

语谱图是将音频信号在时域和频域上同时表示出来的一种图像，通常可以用于检测音频中的频谱变化和语音识别等任务，用于分析声音的频谱结构和声音的音高、音色等特征。

上面生成的语谱图展示了音频信号的时频特征，横轴表示时间，纵轴表示频率，颜色表示该时间段内该频率的能量强度，颜色越亮表示能量越强。

通过观察语谱图可以了解音频信号的频谱特征，例如频率分布、共振峰等。例如上述结果我们可以看到不同频率上的能量分布情况，以及不同频率之间的相对强度和相位关系。

从图中可以看出该音频信号的频率主要集中在500Hz以下，同时在不同时间段内也出现了频率集中的情况，这可能是因为音频中存在一些重要的声音特征，比如语音中的元音和辅音，乐曲中的音符等。

> **6）简述倒谱分析和美尔倒谱系数的原理2+2、MFCC求解过程2、倒谱分析和美尔倒谱系数的联系及区别2，分析注释一段美尔倒谱系数的代码2（代码可以来源网址）。（注意该代码要有美尔倒谱系数的求解过程，而不能只是一句函数调用）**

1. **倒谱分析和美尔倒谱系数原理**

倒谱分析是一种在信号处理中常用的技术，用于从信号的谱图中提取有关信号的信息。倒谱分析的原理是将信号的对数谱通过傅里叶变换转换为倒谱，倒谱可以用于分析信号的频率轮廓、包络特性和谐波分量等。

在语音识别等领域，这些倒谱系数通常用作声学特征，可以描述语音信号的谐波成分、共振峰等声学特征。（共振峰是指声音信号在声道中受到共振增强的频率区域。在语音信号处理中，共振峰通常与说话人的嗓音特征有关，因为说话人的声道会对发音进行调整，从而在声道中形成一些特定频率区域的共振峰。）

美尔倒谱系数（Mel-frequency cepstral coefficients, MFCCs）是一种通过对信号的频谱图进行梅尔滤波和离散余弦变换得到的特征，常用于语音和音频处理任务中。MFCCs的计算过程主要包括以下两个步骤：

1. **梅尔滤波器组**：将信号的频谱图通过一组梅尔滤波器，这些滤波器在低频区域有较高的分辨率，在高频区域有较低的分辨率，模拟人耳对声音的感知特性。
2. **倒谱系数计算**：对通过梅尔滤波器组的频谱图进行对数运算，然后进行离散余弦变换（DCT），得到倒谱系数。DCT能够将信号在频域中的相关性信息转换到倒谱系数中，通常只保留前几个倒谱系数作为最终的MFCC特征。

MFCCs被广泛应用于语音识别、语音合成、语音增强等领域，它具有对人耳听觉特性的模拟、降低维度、抗噪声等优点。

2. **MFCC求解过程**

美尔倒谱系数（MFCCs）的求解过程通常包括以下步骤：

1. 预处理：将原始信号进行预处理，包括去除静音段、进行语音端点检测、进行语音分帧等操作。
2. 加窗：将每一帧语音信号乘以一个窗函数（例如汉明窗、海宁窗等）以减小频谱泄漏效应。
3. 快速傅里叶变换（FFT）：对加窗后的每一帧语音信号进行FFT，得到频谱图。
4. 梅尔滤波器组：设计一组梅尔滤波器，这些滤波器在梅尔频率尺度上等间隔分布，模拟人耳对声音的感知特性。将频谱图通过这组滤波器，得到每个滤波器的输出能量。
5. 对数运算：对每个滤波器的输出能量取对数，得到对数谱。
6. 离散余弦变换（DCT）：对对数谱进行离散余弦变换，得到倒谱系数。通常只保留前几个倒谱系数作为最终的MFCC特征。
7. 特征处理：对倒谱系数进行一些特征处理操作，例如差分、对数等，以增强特征的稳定性和判别性。
8. 特征归一化：对得到的MFCC特征进行归一化，例如均值方差归一化，将MFCC特征缩放到一定的范围内，以便后续的分类或识别操作。

3. **倒谱分析和美尔倒谱系数的联系及区别**

倒谱分析和美尔倒谱系数（MFCCs）是两种在信号处理领域中用于特征提取的方法，它们之间有一些联系和区别。

联系：

1. 都基于对信号的频谱进行分析。倒谱分析和MFCCs都从信号的频谱信息中提取特征，用于表示信号的频率轮廓、包络特性和谐波分量等。
2. 都可以应用于语音和音频处理。倒谱分析和MFCCs在语音和音频处理领域广泛应用，例如语音识别、语音合成、音频分类等。

区别：

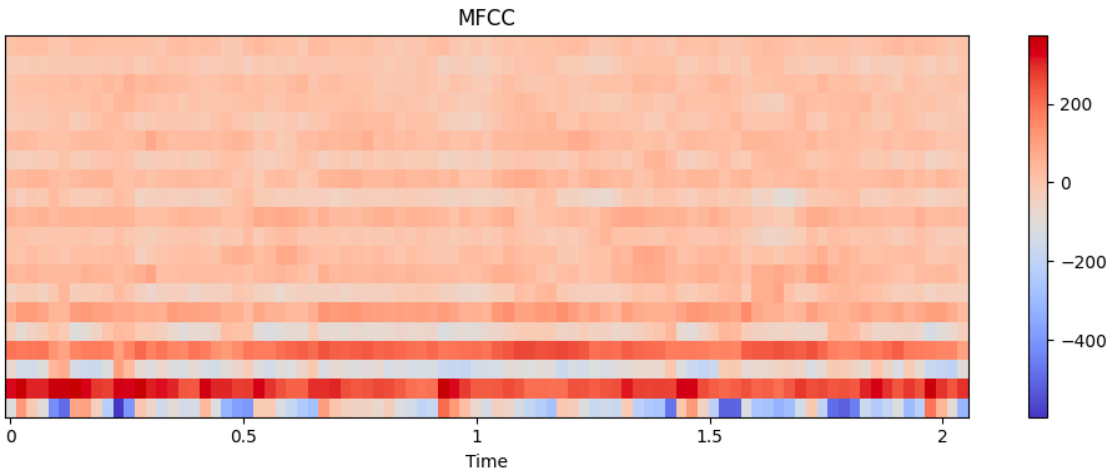
1. 倒谱分析是一种直接的谱分析方法，它通过将信号的对数谱通过傅里叶变换得到倒谱，用于分析信号的频谱特性。而MFCCs则是一种基于梅尔滤波器组和离散余弦变换的特征提取方法，它通过将信号的频谱图经过梅尔滤波器组和离散余弦变换得到倒谱系数，用于表示信号的频谱特性。
2. MFCCs考虑了人耳听觉特性。梅尔滤波器组在MFCCs中的应用可以模拟人耳对声音的感知特性，将频率划分为梅尔频率尺度，使得较低频率区域有较高的分辨率，较高频率区域有较低的分辨率，更符合人耳对声音的感知。而倒谱分析则

没有考虑人耳听觉特性。

3. MFCCs通常只保留少数几个倒谱系数作为最终的特征。在MFCCs中，通常只保留前几个倒谱系数作为最终的特征，这些倒谱系数通常包含了信号的主要频谱特性。而倒谱分析则可以保留更多的倒谱系数，用于表示信号的更详细频谱信息。
4. MFCCs通常包括了梅尔滤波器组的设计。MFCCs中需要设计一组梅尔滤波器，用于将信号的频谱图通过滤波器组得到每个滤波器的输出能量。而倒谱分析则不需要进行滤波器设计。

4. **分析注释一段美尔倒谱系数的代码**

```
import librosa
import numpy as np
import matplotlib.pyplot as plt

# 读取音频文件
audio_path = 'sample.wav'
y, sr = librosa.load(audio_path)  # y：音频信号的时域数据，sr：采样率

# 计算梅尔滤波器组数
n_fft = 2048  # FFT点数，即音频信号被分为多少个FFT窗口
n_mels = 128  # 梅尔滤波器组数
fmin = 0  # 最小频率值
fmax = sr // 2  # 最大频率值（设为采样率的一半，表示不考虑超过采样率一半的频率）
mel_basis = librosa.filters.mel(sr=sr, n_fft=n_fft, n_mels=n_mels, fmin=fmin, fmax=fmax)

# 计算功率谱
S = librosa.stft(y, n_fft=n_fft, hop_length=n_fft // 2)  # 对音频信号进行短时傅里叶变换，得到每个时刻的频谱信息S
power = np.abs(S) ** 2  # 每个时刻的功率谱

# 计算梅尔倒谱系数
n_mfcc = 20
mfcc = librosa.feature.mfcc(S=librosa.power_to_db(power), n_mfcc=n_mfcc, mel_basis=mel_basis)
'''
librosa.power_to_db()：将功率谱转换为分贝尺度的值，
n_mfcc：需要计算的MFCC个数，
mel_basis是上一步计算得到的梅尔滤波器矩阵。
'''

print(mfcc)


# 可视化梅尔倒谱系数
plt.figure(figsize=(10, 4))
librosa.display.specshow(mfcc, x_axis='time')
plt.colorbar()
plt.title('MFCC')
plt.tight_layout()
plt.show()
```


MFCC
200
0
−200
−400
0
0.5
1
1.5
2
Time