Assignment 1:Speech Signal Processing

·Student name: Lin Juekai

·Student id: 2253744

·Tutor: Ying Shen

1.Project Introduction

MFCC (Mel-Frequency Cepstral Coefficients, MEL frequency inversion coefficient) is a widely used feature extraction method for speech signal processing, especially in speech recognition and audio analysis.It consists contains the following processes:Pre-emphasis,Frame-divide,Windowing,STFT,Mel-filterbank,Log(),DCT,Dynamic feature extraction and Feature transformation.

This report describes the extraction process of MFCC (MER frequency inversion coefficient) acoustic features from the speech segment, and compares them with the MFCC calculation function in the python library.

2.Project Files

·Project Structure

```
2253744_林觉凯_Assignment1
        |_Assignment1_report
        |_Assignment1_code
                |_image
                |_Assignment1.py
                |_Assignment1_voice.wav
```

Assignment1_report: Description and analysis report of this project.

Assignment1.py: The main source code for this project.

Assignment1_voice.wav: Audio files for this project.

Iamge: Relevant pictures generated by the MFCC processing procedure.

·Project Environment

Make sure that the following python library is already installed on your computer.

```
pip install numpy librosa matplotlib
```

Run this project code:

```
python Assignment1.py
```

3.Project Process

The MFCC processing process can be roughly divided into the following stages:

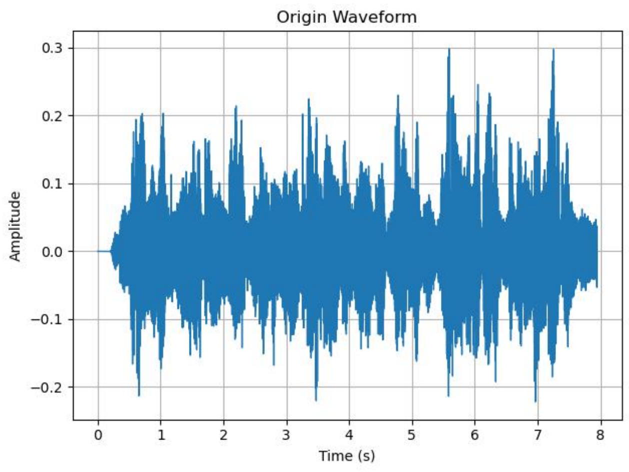
·Pre-emphasis

Pre-emphasis is used to increase the high frequency energy of the speech signal, so that the high frequency part of the speech signal has a similar amplitude to the low frequency part. This is to compensate for the loss of high frequencies in the speech signal and to improve the performance of MFCC.

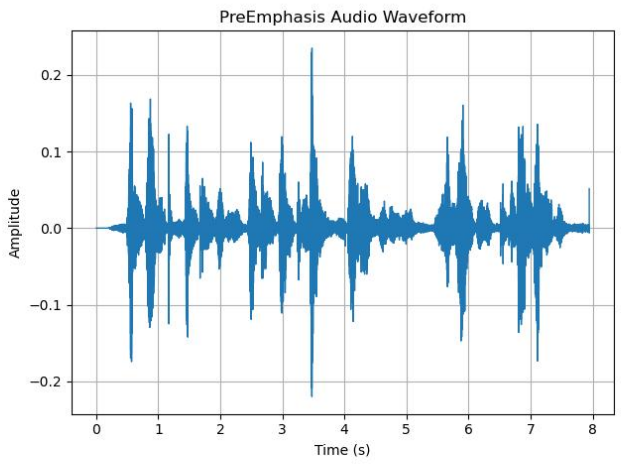
The Pre-emphasis factor alpha is 0.97.

```python
# 对信号进行预加重处理
def preEmphasis(signal, alpha=0.97):
    # alpha: 预加重系数为0.97
    emphasizedSignal = np.append(signal[0], signal[1:] - alpha * signal[:-1])
    return emphasizedSignal
```

The original audio amplitude:



The audio amplitude after Pre-emphasis:

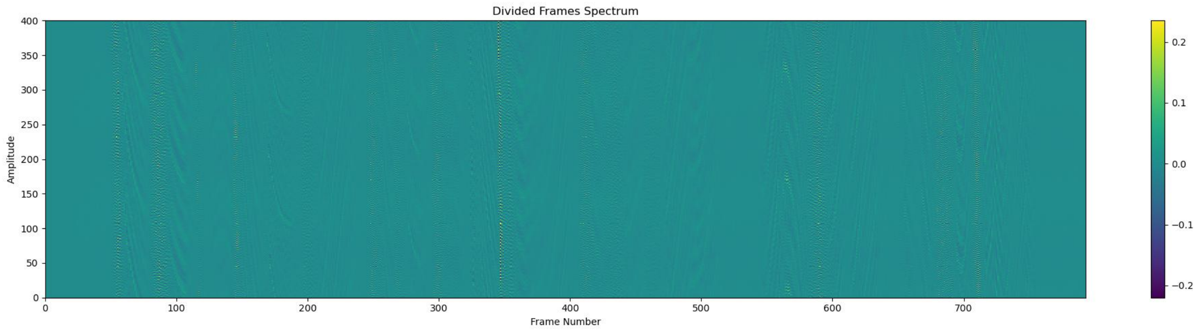


·Frame-divide

The function of frames is to divide continuous voice signals into short time periods to better capture the changes of voice signals over time.Frame length is typically set from 20 ms to 40 ms, with coverage between adjacent frames being 50%. For ASR, the frame width is usually 25 ms and the displacement is 10 ms.

```
# 信号需要被分成短时间帧
def frameDivide(data, frameLen, frameMov):
    sigLen = len(data)
    # 计算帧数，向上取整，保证信号能被完整分帧
    frameNum = int(np.ceil((sigLen - frameLen) / frameMov))

    zeroNum = (frameNum * frameMov + frameLen) - sigLen
    zeros = np.zeros(zeroNum)
    # 将信号填充零后的完整信号
    filledSignal = np.concatenate((data, zeros))
    indices = np.tile(np.arange(0, frameLen), (frameNum, 1)) + \
      np.tile(np.arange(0, frameNum * frameMov, frameMov), (frameLen, 1)).T
    # 根据索引提取分帧后的信号
    indices = np.array(indices, dtype=np.int32)
    divided = filledSignal[indices]
    return divided
```

Time frame series of signals after framing processing:



·Windowing

The purpose of windowing is to improve the accuracy of frequency domain analysis by reducing signal discontinuity between frames and reducing spectral leakage during short-time Fourier transform and other time domain processing.

Commonly used Windows are: rectangular window, Hanming window, Hanning window. This allocation uses the Hamming window.

The window function of the Hanming window is a smooth window with the formula of:

$$w[n] = (1-\alpha) - \alpha cos(\frac{2\pi n}{L-1})$$

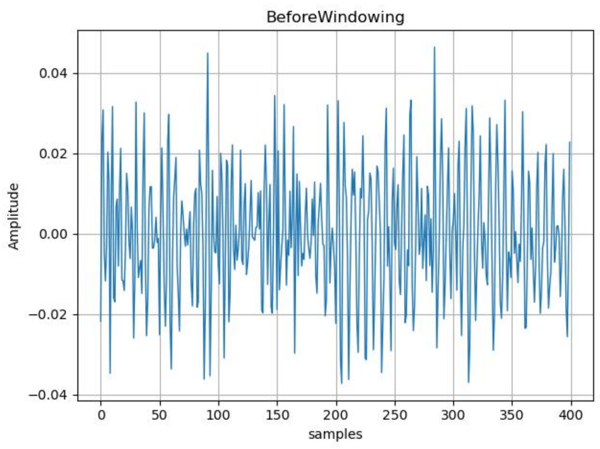
L is the length of the window.

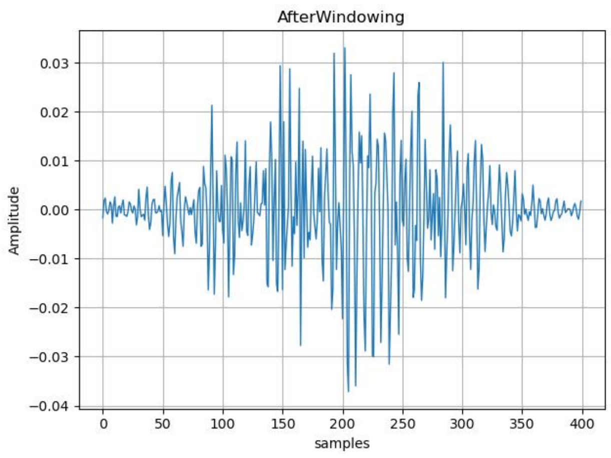
n is the index of the samples in the current window.

α is the parparameter for the window function with a common value of 0.46164, also used for this project.

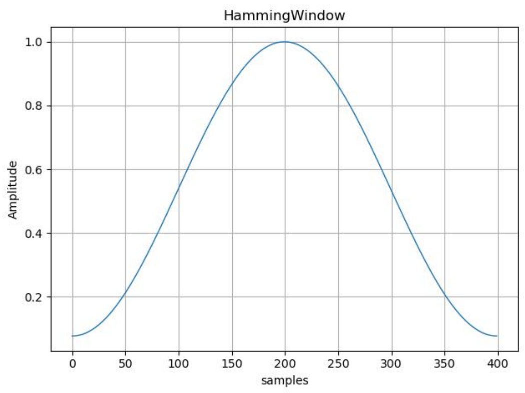
```
# 使用汉明窗进行加窗操作
def hammingWindow(audio,frame_len,alpha=0.46164):
    saveImage(audio[300],"BeforeWindowing",'samples','Amplitude')

    # 创建一个帧长为 frame_len 的汉明窗函数
    n = np.arange(frame_len)
    #  Hamming 窗的公式
    window = 1-alpha - alpha * np.cos(2 * np.pi * n / (frame_len - 1))
    saveImage(window,"HammingWindow",'samples','Amplitude')
    # 将每一帧的音频信号与汉明窗函数逐点相乘，得到加窗后的信号
    windowed_audio = audio * window
    saveImage(windowed_audio[300],"AfterWindowing",'samples','Amplitude')
    return windowed_audio
```

The audio signal before the windowing and after the windowing:





The shape of the Hamming window function:



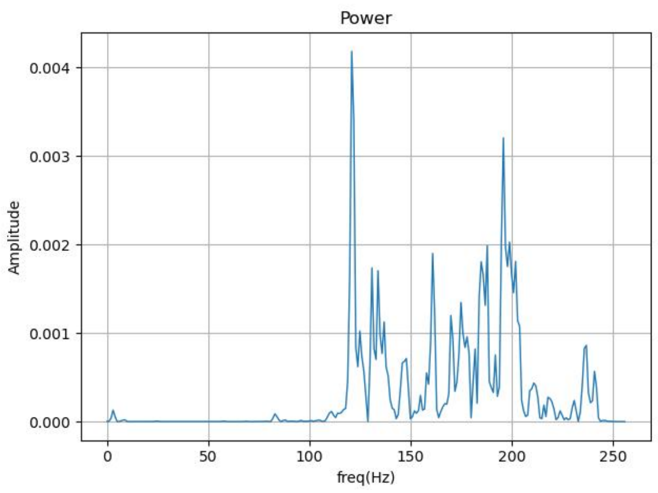
·STFT

Short-time Fourier Transform is a conversion tool from time domain to frequency domain, used to analyze the frequency composition of audio signals. By splitting the continuous signals into short frames and applying the Fourier transform to them separately, we can obtain the frequency spectrum for each frame.

For each frame of the window signal, a N point FFT transform.

```
# 进行快速傅里叶变换
def stft(audioFrame, nFft):
    # 使用 rfft 得到每个频率的结果
    magnitudeFrame = np.absolute(np.fft.rfft(audioFrame, nFft))
    powerFrame = (1.0 / nFft * (magnitudeFrame ** 2))
    saveImage(powerFrame[300], "Power", 'freq(Hz)', 'Amplitude')
    return powerFrame
```

Outputs the power distribution of one frame signal in the frequency domain:



·Mel-filterbank

The Meer filter is used to convert the signal from the frequency domain to the Mer frequency domain. The MEer frequency is designed based on the perception of the human ear to the frequency, which approximates the logarithmic scale and can better capture the different perception abilities of the human ear to low and high frequencies.

Frequency to mel scale mapping: convert the frequency of the audio signal from Hertz to mel scale with the following formula:

$$m = 2595 \times log_{10}(1 + \frac{f}{700})$$

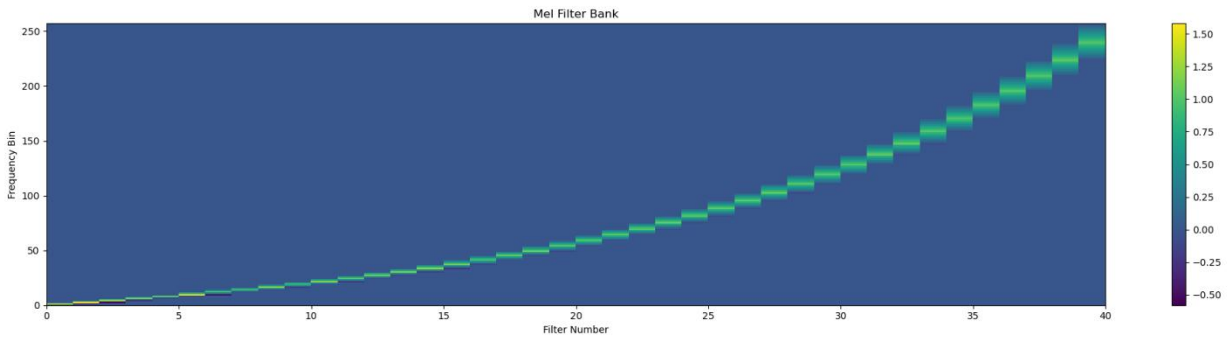
Inverse mapping of Mayer scale to frequency: reverse convert Mayer frequency to Hertz frequency with the following formula:

$$f = 700 \times (10^{\frac{m}{2595}} - 1)$$

The final output is the Merer filter set (fbank), each of which captures signal features in different frequency ranges.

```
# 梅尔滤波器处理
def melFilter(sampleRate, nFft):
    # 将频率范围映射到梅尔刻度上
    lowFreqMel = 0
    highFreqMel = 2595 * np.log10(1 + (sampleRate / 2) / 700)

    nfilt = 40
    melPoints = np.linspace(lowFreqMel, highFreqMel, nfilt + 2)
    hzPoints = 700 * (10 ** (melPoints / 2595) - 1)
    # 创建滤波器组，每个滤波器有 nFft/2 + 1 个频率点
    fbank = np.zeros((nfilt, int(nFft / 2 + 1)))
    bin = (hzPoints / (sampleRate / 2)) * (nFft / 2)
     # 生成每个梅尔滤波器的三角滤波器
    for m in range(1, nfilt + 1):
        fMMinus = int(bin[m - 1])
        fM = int(bin[m])
        fMPlus = int(bin[m + 1])
        for k in range(fMMinus, fM):
            fbank[m - 1, k] = (k - bin[m - 1]) / (bin[m] - bin[m - 1])
        for k in range(fM, fMPlus):
            fbank[m - 1, k] = (bin[m + 1] - k) / (bin[m + 1] - bin[m])

    plotSpectrogram(fbank.T, "Mel Filter Bank", "Filter Number", "Frequency Bin")
    # 返回梅尔滤波器组
    return fbank
```

The frequency response diagram of the Mayer filter bank is as follows:

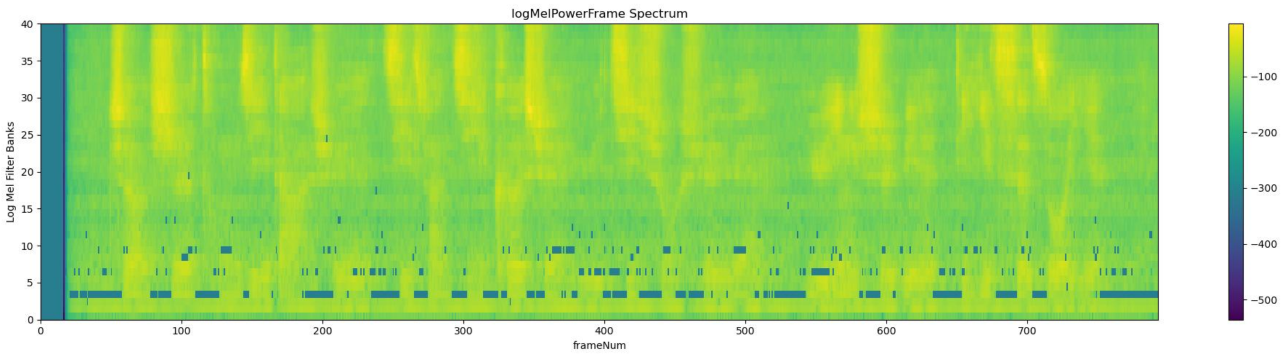


·Log()

The logarithmic transformation is applied to the output energy of the MEer filter set, mainly to simulate the perception characteristics of the human ear to sound. The human ear senses sound logarithmically, meaning that the human ear is insensitive to changes in sound amplitude and has a nonlinear response to changes in energy amplitude. Log conversion can make the amplitude of the spectrum more in line with the law of human auditory perception.

```
# 进行对数转换
def applyLogFilterBankEnergy(filterBanksEnergy):
    filterBanksEnergy = np.where(filterBanksEnergy <= 0, np.finfo(float).eps,
filterBanksEnergy)
    logMelEnergy = 20 * np.log10(filterBanksEnergy)
   return logMelEnergy
```

The output energy of the Mayer filter is log-transformed to reflect the log-mel energy distribution at different time frames:



·DCT

DCT is used to transform the log-mel energy spectrum into a set of discrete cosine coefficients by preserving the first few significant coefficients, squeezing the feature dimensions and removing redundant information. This makes the feature vectors more compact, reducing computational complexity, while retaining critical information about the audio signal, thus improving the effectiveness of the task.

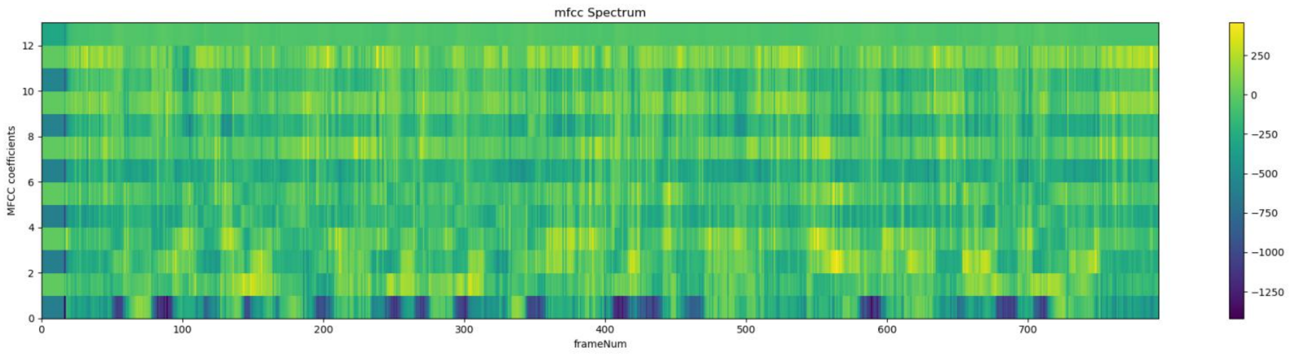
Since log power spectrum is real and symmetric the inverse DFT is equivalent to a discrete cosine transform.

$$c_t[k] = \sum_{m=0}^{M-1} log(Y_t[m])cos((m+0.5)\frac{k\pi}{M}), k = 0, \cdots, C-1$$

C is the number of MFCCs.

```
# 离散余弦变换
def dct(logMel, nMfcc=26, nCeps=12):
    transpose = logMel.T
    lenData = len(transpose)

    # 对每个 MFCC 系数进行计算
    dctAudio = []
    for j in range(nMfcc):
        temp = 0
        # 计算 DCT 系数
        for m in range(lenData):
            temp += (transpose[m]) * np.cos(j * (m - 0.5) * np.pi / lenData)
        dctAudio.append(temp)
    # 只保留前 nCeps 个系数
    ret = np.array(dctAudio[1:nCeps + 1])
    return ret
```

The following is a combined view of MFCC features and frame energy, showing the speech features and energy changes over time.



·Dynamic feature extraction

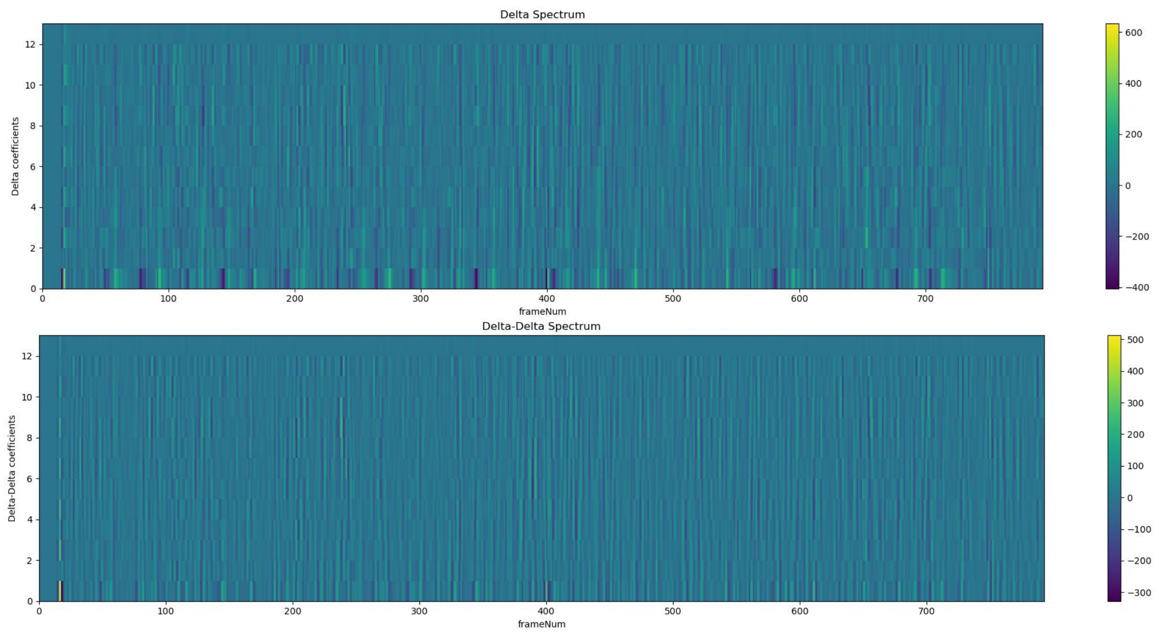
Dynamic feature extraction is used to describe the temporal trend of speech signals and compensate for the defect that static MFCC features cannot reflect dynamic information. By calculating the first and second derivatives of the MFCC features, we can capture the rate of change in time as well as the acceleration.

“Standard” ASR features (for GMM-based systems) are 39 dimensions:

- 12 MFCCs, and energy
- 12 ΔMFCCs, Δenergy
- 12 Δ2MFCCs, Δ^2energy

```
# 动态特征提取
def delta(data, k=1):
    # deltaFeat 用于存储每一帧的导数特征
    deltaFeat = []
    transpose = data.T
    q = len(transpose)
    for t in range(q):
        # 对于前 k 帧，使用后一帧与当前帧的差值计算
        if t < k:
            deltaFeat.append(transpose[t + 1] - transpose[t])
        # 对于最后 k 帧，使用当前帧与前一帧的差值计算
        elif t >= q - k:
            deltaFeat.append(transpose[t] - transpose[t - 1])
        # 对于中间帧，使用对称的加权差分公式计算
        else:
            denominator = 2 * sum([i ** 2 for i in range(1, k + 1)])
            numerator = sum([i * (transpose[t + i] - transpose[t - i]) for i in range(1, k + 1)])
            deltaFeat.append(numerator / denominator)
    return np.array(deltaFeat)
```

Visualization display of two dynamic features (first derivative and second derivative):



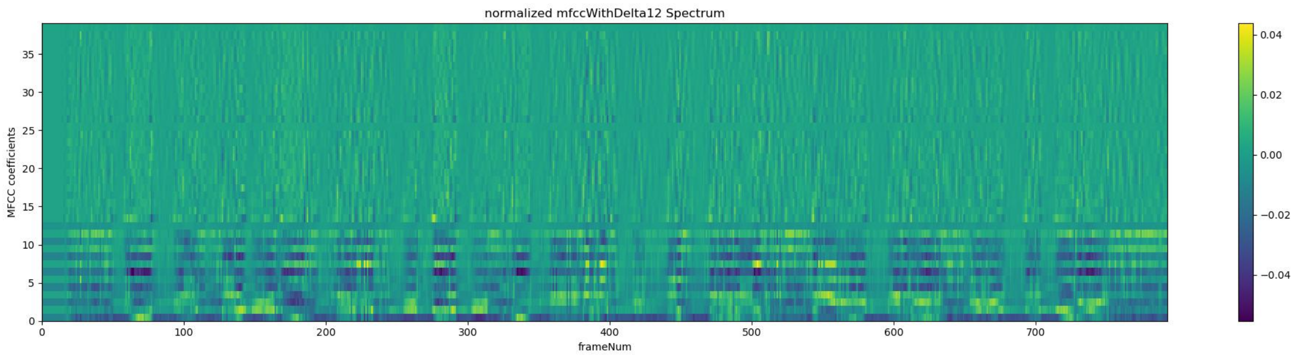
·Feature transformation

In MFCC, feature transformation, such as normalization or standardization, is mainly used to adjust the scale of features to avoid the impact of excessive numerical differences of features in different dimensions on subsequent processing.

CMN/CVN: Usually, we may want to calculate and apply mean and variance statistics on the same speaker/channel to ensure that data from the same speaker or the same channel has a similar distribution.

```
# 特征变换，归一化或标准化
def normalization(data):
    dataMean = np.mean(data, axis=0, keepdims=True)
    dataVari = np.var(data, axis=0, keepdims=True)
     # 归一化操作，将每个特征值减去均值并除以方差
    return (data - dataMean) / dataVari
```

MFCC coefficient plot after normalization:

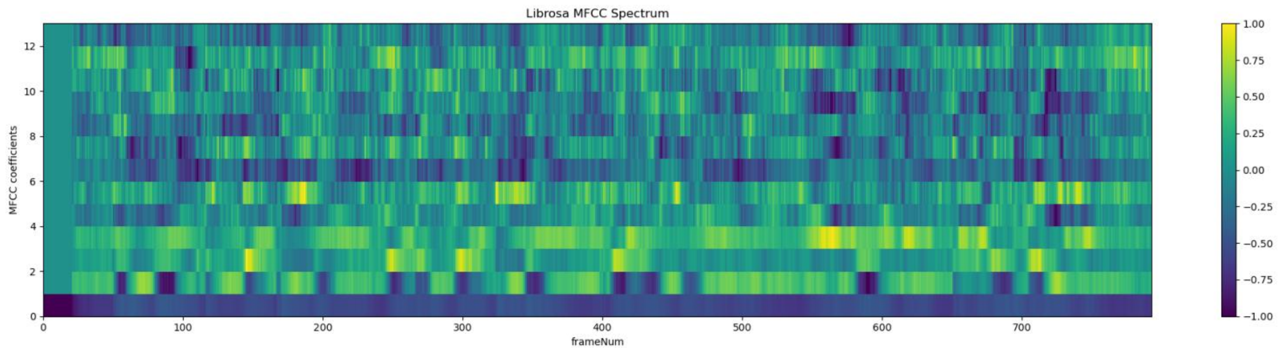


4.Project Comparison and Analysis

The Librosa library in Python also provides the function to generate MFCC.In this project, I also called the library to generate the MFCC of the audio files.

```
# 使用 Librosa 库生成的 MFCC 结果
def compareWithLibrosa(audio, sr, mfccHandwritten):
    # 设置 hop_length 更小，增加每帧的精度
    librosaMfcc = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13,
hop_length=frameMov // 1, n_fft=512)
    minFrames = min(mfccHandwritten.shape[1], librosaMfcc.shape[1])
    mfccHandwritten = mfccHandwritten[:, :minFrames]
    librosaMfcc = librosaMfcc[:, :minFrames]

    # 归一化 Librosa 生成的 MFCC 系数
    librosaMfcc = librosa.util.normalize(librosaMfcc, axis=1)
    # 显示 Librosa 的 MFCC 计算结果
    plotSpectrogram(librosaMfcc, 'Librosa MFCC Spectrum', 'frameNum', 'MFCC
coefficients')
```

The final results are plotted below the figure:



We found that the results of the handwritten MFCC and the MFCC results generated in the Librosa library were similar in the range 0-13, with the energy spectrum generally consistent with frequency, and their spectral structure was similar.

However, the MFCC results generated by the obvious Librosa library have high processing accuracy and better stability, that is, the data in the detection and processing interval is more detailed, and the changes are more stable.

After analysis, I think the reasons for these differences are as follows:

• When performing DCT and log operations, handwritten code may have a small deviation from the output of Librosa because of the different way and accuracy of floating point processing. This deviation is especially evident in the high or smaller energies.

• In the number of Mayer filters and design, numFilters is a fixed value, but in fact the number of Mayer filters needs to be adjusted dynamically, according to the characteristics of frequency band distribution and audio. In general, lower frequency filters require larger bandwidth, and higher frequency filters require tighter bandwidth.

• Stability of log processing: although np.finfo (float) was used. And eps to avoid negative and zero values in log operations, but the method is not always optimal. For some low-amplitude signals, the np.finfo (float). The values of eps may not be small enough to cause numerical instability.

5.Project Summary

Through experiments, I successfully completed the task of speech feature extraction, and also had a deep understanding of the calculation process of MFCC through handwriting implementation and comparison with the existing library(Librosa). By gradually realizing the steps of speech segment preprocessing, window function, Fourier transform, Mayer filter, logarithmic energy, discrete cosine transformation (DCT), Dynamic feature extraction and Feature transformation.I not only master the basic principle of MFCC feature extraction, but also better understand the role of each step in feature extraction.