# Speech Signal Processing Project Description

### Code Logic

The logic of the MFCC extraction method in this project can be divided into several steps. Each step simulates common processes in audio feature extraction, including signal preprocessing, feature extraction, and transformation. Below is a detailed explanation of the method's logic:

**Step 1: Pre-emphasis**

pre\_emphasis = 0.97

emphasized\_signal = np.append(signal[0], signal[1:] - pre\_emphasis \* signal[:-1])

**Purpose**: Pre-emphasize the original audio signal to increase the weight of the high-frequency components. This compensates for the attenuation of high frequencies caused by the microphone or sound wave propagation, improving speech clarity.

**Logic**: A simple first-order high-pass filter is used to filter out the low-frequency parts by enhancing the high-frequency components through the difference between the current and previous sample points.

**Step 2: Framing and Windowing**

frame\_size = 0.025 # 25 ms

frame\_stride = 0.01 # 10 ms

frame\_length, frame\_step = frame\_size \* sample\_rate, frame\_stride \* sample\_rate

frame\_length = int(round(frame\_length))

frame\_step = int(round(frame\_step))

signal\_length = len(emphasized\_signal)

num\_frames = int(np.ceil(float(np.abs(signal\_length - frame\_length)) / frame\_step))

**Purpose**: To analyze the signal in short time intervals, the audio signal is divided into multiple small frames (typically 20-40 milliseconds), each reflecting the signal's characteristics in a short time window. Windowing is applied to reduce the impact of frame boundary transitions on spectrum analysis.

**Logic**:

* Each frame is set to 25 milliseconds, and the frame stride (frame interval) is set to 10 milliseconds, which are common settings in speech processing.
* After framing, each frame corresponds to a short time range.

frames \*= np.hamming(frame\_length)

* A Hamming window function is applied to each frame to reduce signal discontinuities at frame boundaries (boundary effects), thereby minimizing artifacts during the Fourier transform.

**Step 3: Short-Time Fourier Transform (STFT)**

NFFT = 512

mag\_frames = np.absolute(np.fft.rfft(frames, NFFT))

pow\_frames = ((1.0 / NFFT) \* (mag\_frames \*\* 2))

**Purpose**: To convert the time-domain signal into the frequency domain using the Fourier transform and analyze the frequency components of each frame.

**Logic**:

* The Fast Fourier Transform (FFT) is used to convert each frame to the frequency domain, obtaining the magnitude spectrum.
* The power spectrum is calculated by squaring the magnitude of the FFT.

**Step 4: Applying Mel Filterbank**

nfilt = 40

mel\_points = np.linspace(low\_freq\_mel, high\_freq\_mel, nfilt + 2)

hz\_points = 700 \* (10\*\*(mel\_points / 2595) - 1)

bin = np.floor((NFFT + 1) \* hz\_points / sample\_rate).astype(int)

**Purpose**: To map the frequency to the Mel scale, which aligns better with human auditory perception. The Mel scale gives more sensitivity to lower frequencies, so more filters are applied in the lower frequency range.

**Logic**:

* First, frequencies are converted from Hertz to the Mel scale, as the Mel scale better reflects the human ear's response.
* Then, filterbanks are distributed evenly on the Mel scale, converting Mel frequencies back to Hertz to determine the boundaries of each filter (FFT bin positions).

for m in range(1, nfilt + 1):

f\_m\_minus = int(bin[m - 1]) # left

f\_m = int(bin[m]) # center

f\_m\_plus = int(bin[m + 1]) # right

for k in range(f\_m\_minus, f\_m):

fbank[m - 1, k] = (k - bin[m - 1]) / (bin[m] - bin[m - 1])

for k in range(f\_m, f\_m\_plus):

fbank[m - 1, k] = (bin[m + 1] - k) / (bin[m + 1] - bin[m])

* The filters are triangular and overlap on the Mel frequency range, with each filter output reflecting the energy of frequency components within that filter's range.

filter\_banks = np.dot(pow\_frames, fbank.T)

filter\_banks = 20 \* np.log10(filter\_banks) # dB

* Mel-weighted spectral energy is extracted for each frame by computing the dot product of the power spectrum and the Mel filterbank. The result is converted to a logarithmic scale (dB), making the features closer to human auditory perception.

**Step 5: Discrete Cosine Transform (DCT)**

mfcc = dct(filter\_banks, type=2, axis=1, norm='ortho')[:, :num\_ceps]

**Purpose**: To further compress the output of the Mel filterbank using the Discrete Cosine Transform (DCT), retaining the most important features.

**Logic**:

* After applying the DCT, the first 13 coefficients are kept as the final MFCC features, while higher-order coefficients (representing high-frequency variations) are discarded, as they usually indicate noise rather than speech characteristics.

**Step 6: Dynamic Feature Extraction (Delta and Delta-Delta)**

delta\_mfcc = librosa.feature.delta(mfcc)

delta2\_mfcc = librosa.feature.delta(mfcc, order=2)

**Purpose**: To capture the dynamic changes in MFCC features over time by extracting the first and second-order differences.

**Logic**:

* The first-order difference represents the rate of change of MFCC features, while the second-order difference represents the change in the rate of change (acceleration). These features better capture the temporal dynamics of speech.

**Step 7: Mean Normalization**

mfcc -= (np.mean(mfcc, axis=0) + 1e-8)

**Purpose**: To eliminate offset in different audio signals, ensuring consistent feature scaling and preventing differences in overall energy from adversely affecting model training.

**Step 8: Result Visualization and Comparison**

fig, axs = plt.subplots(1, 2, figsize=(15, 6))

axs[0].imshow(mfcc.T, aspect='auto', origin='lower')

axs[0].set\_title('Custom MFCC')

librosa\_mfcc = librosa.feature.mfcc(y=signal, sr=sample\_rate, n\_mfcc=num\_ceps)

axs[1].imshow(librosa\_mfcc, aspect='auto', origin='lower')

axs[1].set\_title('MFCC (Librosa)')

**Purpose**: To visualize and compare the custom MFCC features with those provided by the librosa library.

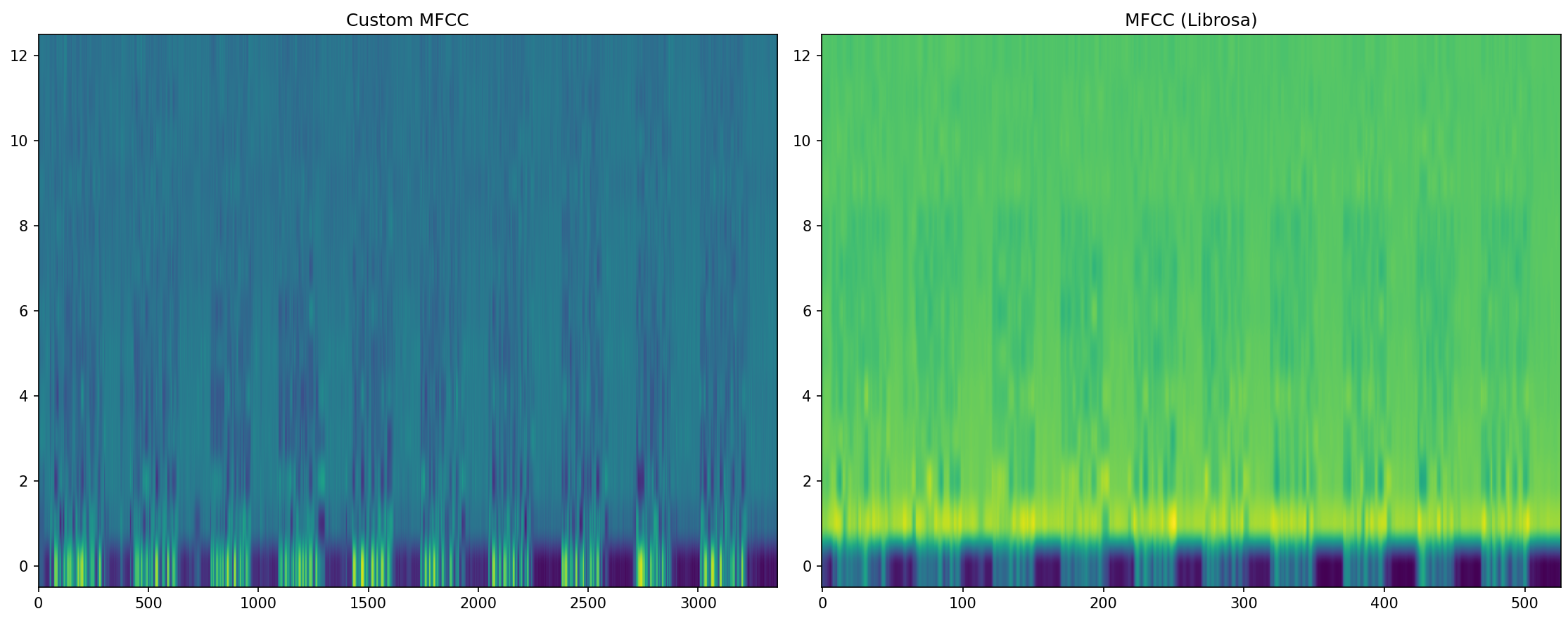
**Logic**:

* matplotlib is used to display two MFCC plots: one from the custom computation and the other from librosa. This helps check the differences between the custom implementation and the standard library.

**Summary**

The custom MFCC extraction process closely follows the standard audio feature extraction procedure, from signal preprocessing, framing, windowing, STFT, applying the Mel filterbank, to DCT and normalization. This implementation closely mimics the functionality of librosa's mfcc function but allows for parameter tuning for specific applications.

### Comparison Results



Comparing the output from the custom method with the Python package's MFCC function, we can observe the following differences:

1. **Color distribution and dynamic range**:
   * **Custom MFCC (left)**: Lower color contrast, with more blue and green, indicating a smaller dynamic range.
   * **Librosa MFCC (right)**: Higher contrast, with more yellow, showing larger values in certain frequencies and a broader dynamic range.
2. **Smoothness of the frequency axis**:
   * **Custom MFCC (left)**: The frequency axis (vertical) appears smoother, with fewer abrupt changes.
   * **Librosa MFCC (right)**: More defined frequency variations, especially in the lower frequencies (bottom bands), showing more high-intensity changes (yellow regions).
3. **Differences in the time axis**:
   * **Custom MFCC (left)**: The time axis (horizontal) is stretched, indicating more frames and a longer analysis window.
   * **Librosa MFCC (right)**: Fewer frames, possibly due to differences in the framing and sampling rate processing between librosa and the custom implementation.

### Analysis of Differences

1. **Filterbank design**:
   * The custom filterbank might differ from librosa's, which has been optimized to better smooth frequency spectrum changes, while the custom implementation might not have applied extra adjustments.
2. **Dynamic range compression**:
   * librosa likely uses a different dynamic range compression and normalization technique, making higher-intensity frequency components more prominent, whereas the custom implementation might have a more conservative dynamic range, leading to lower contrast.
3. **Frame and time processing**:
   * The custom framing and windowing strategy may differ from librosa's default settings, resulting in more frames and a longer analysis time window. This could also be influenced by librosa's default sample rate and framing approach.
4. **Normalization**:
   * librosa may apply more sophisticated normalization or mean subtraction during MFCC extraction, resulting in smoother and higher contrast results. While the custom implementation includes basic mean normalization, it may not match librosa's more advanced processing.

Overall, librosa's MFCC results exhibit more nuanced frequency changes and a broader dynamic range. To make the custom implementation more similar, finer adjustments to filterbank design, dynamic range compression, and consistent framing and normalization techniques could be applied.