Windowing:

The simplest way to analyze a signal is in its raw form. For time series signals (signals that evolve with time), windowing allows us to view a short time segment of a longer signal and analyze its frequency content. Mathematically, windowing can be seen as multiplying a signal with a window function that is zero everywhere on the given signal except on the region of interest.

As the window slides over the signal in time, the size is made adaptive by changing according to characteristics of the audio signal. One problem with this is the abrupt changes or jump discontinuities in its shape at the edges of the window. The distortion is the result of the Gibbs phenomenon.

Available Window functions:

Rectangular Window: Simplest window, often used when no smoothing is needed. Impulse-like response, leading to poor frequency resolution.

Hann Window: Balances between main lobe width and side-lobe suppression. Provides better frequency resolution than the rectangular window.

Hamming Window: Similar to Hann but with different weighting. Improved side-lobe suppression compared to Hann.

Blackman Window: Offers a good compromise between main lobe width and side-lobe levels. Suitable for applications where minimizing side lobes is crucial.

Triangular Window (Bartlett): Linearly ramps up and down, forming a triangular shape. Wider main lobe compared to Hann and Hamming.

Bartlett Window: Similar to the triangular window but with a different weighting scheme. Provides a compromise between rectangular and triangular windows.

Kaiser Window:

Controlled by a shape parameter (beta). Allows customization of the trade-off between main lobe width and side-lobe suppression.

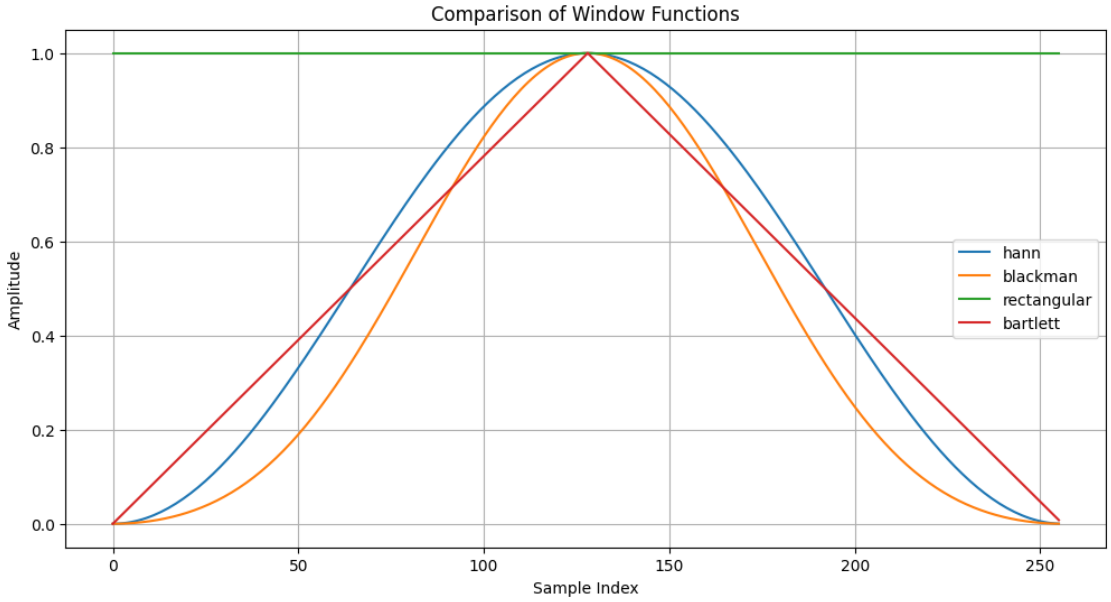
Gaussian Window: Shaped like a Gaussian distribution. Controlled by the standard deviation (std) parameter.

Flat Top Window: Designed to have a flat top in the frequency domain. Suitable for precise frequency analysis with low side-lobe levels.

Code:

```
# Define parameters
window_length = 256  # Adjust the length based on your needs
window_types = ['hann', 'blackman','rectangular','bartlett']  # Choose the window functions

# Generate x-axis values (sample indices)
x = np.arange(window_length)
```

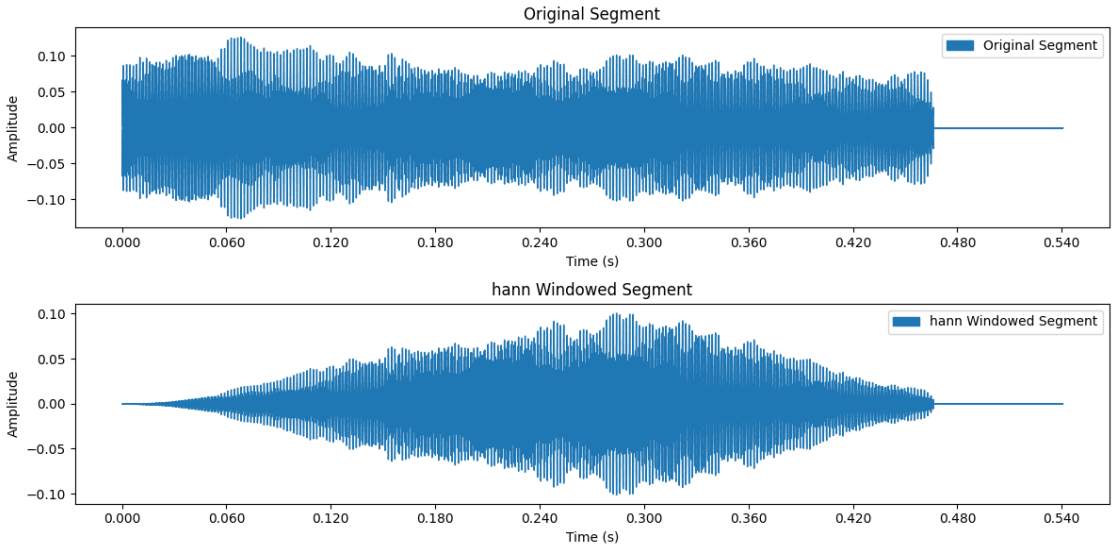
```
# Plot the window functions
plt.figure(figsize=(12, 6))
for window_type in window_types:
    window = signal.get_window(window_type, window_length)
    plt.plot(x, window, label=window_type)

plt.title('Comparison of Window Functions')
plt.xlabel('Sample Index')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)
plt.show()
```



Code to plot the Hann window function for an Audio file:

```
# Load an audio file
y, sr = librosa.load(debussy_file)

# Choose a segment of the audio
start_time = 1  # start time in seconds
duration = 2    # duration in seconds
y_segment = y[int(start_time * sr):int((start_time + duration) * sr)]

# Apply a window function (e.g., Hann) to the segment
window_type = 'hann'
window = signal.get_window(window_type, len(y_segment))
y_windowed = y_segment * window
```

```
# Plot the original and windowed signals
plt.figure(figsize=(12, 6))

# Plot the original segment
plt.subplot(2, 1, 1)
librosa.display.waveshow(y_segment, sr=sr, label='Original Segment')
plt.title('Original Segment')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()

# Plot the windowed segment
plt.subplot(2, 1, 2)
librosa.display.waveshow(y_windowed, sr=sr, label=f'{window_type} Windowed Segment')
plt.title(f'{window_type} Windowed Segment')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()

plt.tight_layout()
plt.show()
```



1)Root-mean-squared energy:

Root Mean Square (RMS) energy is a measure commonly used in audio signal processing to quantify the energy content of a signal. In the context of audio, RMS energy is often used to describe the average power of a signal over a certain time window.

Here's a brief overview of how RMS energy is calculated for an audio signal:

Square the Values: Take each sample of the audio signal and square it. This is done to eliminate any negative values and to emphasize the magnitudes of the samples.

Calculate the Mean: Find the average (mean) of the squared values obtained in step 1.

Take the Square Root: Finally, take the square root of the mean calculated in step 2. This is the RMS value.

RMS energy is often used as a measure of the overall loudness or energy level of a signal. It is more stable than the raw amplitude values and is useful in applications like audio processing, compression, and normalization.

In the context of audio processing, RMS energy can be computed over short time intervals to provide a time-varying measure of energy. This can be useful in tasks such as audio analysis, where you might want to identify changes in energy levels over time, for instance, in speech or music signal processing.

Keep in mind that the choice of time window for calculating RMS energy is important and depends on the specific application. Shorter windows capture more rapid changes in energy, while longer windows provide a more smoothed-out measure over time.

$$RMS = \sqrt{\frac{1}{n}\sum_{i} x_i^2}$$

RMS = root mean square

n = number of measurements

x_i = each value

Applications:

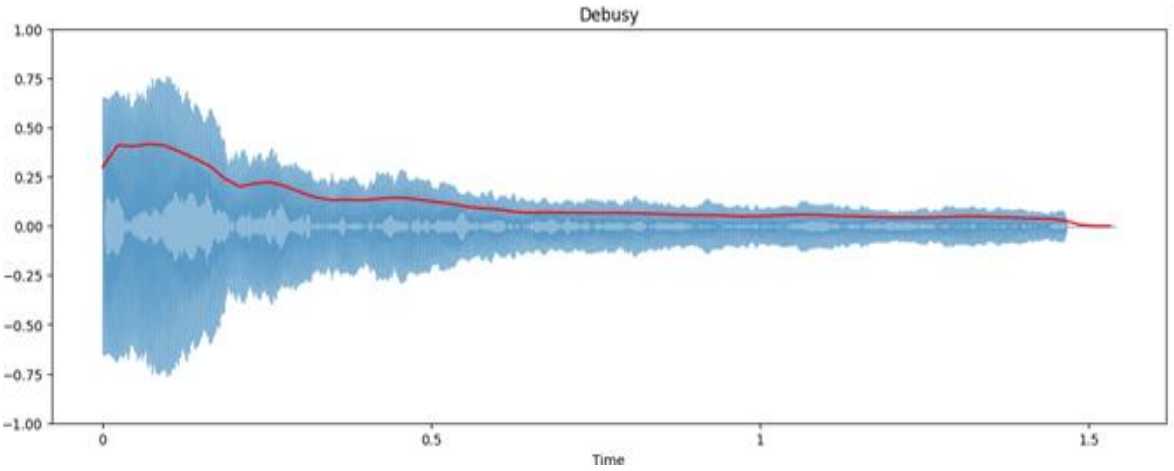
Amplitude Measurement: RMS is commonly used to measure the amplitude or intensity of a signal. Unlike the peak amplitude, which provides the maximum value of a signal, RMS provides a representation of the signal's power or energy over time. This is particularly useful in scenarios where a more representative measure of signal strength is needed.

Vibration Analysis: RMS is extensively used in vibration analysis to assess the energy content of vibration signals. In machinery health monitoring and predictive maintenance, RMS values of vibration signals are employed to detect changes in the mechanical condition of equipment.

Signal Quality Assessment: RMS is often used as a metric for signal quality assessment. For example, in communication systems, RMS can be employed to measure the signal-to-noise ratio (SNR) and assess the fidelity of a transmitted signal.

Code:

```
FRAME_SIZE = 1024
HOP_LENGTH = 512

rms_debussy = librosa.feature.rms(y=debussy, frame_length=FRAME_SIZE,
hop_length=HOP_LENGTH)[0]

frames1 = range(len(rms_debussy))
t1 = librosa.frames_to_time(frames1, hop_length=HOP_LENGTH)

# rms energy is graphed in red
plt.figure(figsize=(15, 17))
ax = plt.subplot(3, 1, 1)
librosa.display.waveshow(debussy, alpha=0.5)
plt.plot(t1, rms_debussy, color="r")
plt.ylim((-1, 1))
plt.title("Debusy")
plt.show()
```



2)RMSE

The Root Mean Squared Error (RMSE) is a metric commonly employed in regression analysis to assess the accuracy of predictive models. It quantifies the disparity between predicted values and actual observations. The process involves calculating the squared difference for each data point, ensuring that both positive and negative errors contribute meaningfully. The Mean Squared Error (MSE) is then determined by averaging these squared errors. To facilitate interpretation and maintain the original data's unit, the square root of the MSE is taken, resulting in the RMSE. This metric penalizes larger errors more significantly due to the squaring operation. A lower RMSE signifies better model performance, indicating smaller discrepancies between predicted and observed values. It's crucial to recognize that while RMSE is widely used, its appropriateness can depend on the specific characteristics of the data, and alternative metrics may be considered in certain situations, such as Mean Absolute Error (MAE) or Mean Absolute Percentage Error (MAPE).

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

Applications:

Peak Signal to Noise Ratio: PSNR is a metric commonly used in image and video processing, but it can also be adapted for audio. It measures the ratio between the maximum possible power of a signal and the power of corrupting noise. Higher PSNR values suggest better signal quality.

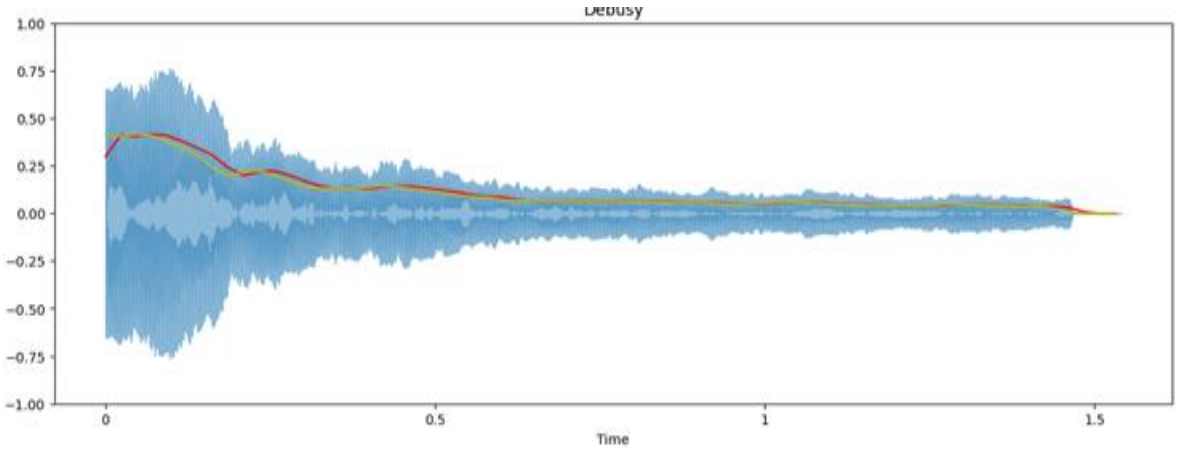
Perceptual Evaluation of Audio Quality: PEAQ is a standardized method for objectively assessing and measuring perceived audio quality. It evaluates various aspects of audio quality, including loudness, sharpness, and annoyance. PEAQ provides a comprehensive approach to audio quality assessment, taking into account perceptual aspects.

Subjective Listening Tests: In many audio processing applications, especially those related to music and entertainment, subjective listening tests are crucial. These tests involve human listeners evaluating the quality of audio signals subjectively. While not a mathematical metric like RMSE, subjective tests provide valuable insights into how humans perceive audio quality.

Code:

```
def rmse(signal, frame_size, hop_length):
    rmse = []

    # calculate rmse for each frame
    for i in range(0, len(signal), hop_length):
        rmse_current_frame = np.sqrt(sum(signal[i:i+frame_size]**2) / frame_size)
```

```
        rmse.append(rmse_current_frame)
    return np.array(rmse)

rms_debussy1 = rmse(debussy, FRAME_SIZE, HOP_LENGTH)

plt.figure(figsize=(15, 17))

ax = plt.subplot(3, 1, 1)
librosa.display.waveshow(debussy, alpha=0.5)
plt.plot(t1, rms_debussy, color="r")
plt.plot(t1, rms_debussy1, color="y")
plt.ylim((-1, 1))
plt.title("Debusy")
```



3)Zero-crossing rate:

Zero Crossing Rate (ZCR) is a feature commonly used in audio signal processing to analyze the rate at which a signal changes its sign. It is particularly useful in tasks such as speech and music analysis. The zero crossing rate is a measure of how often the signal changes its direction, i.e., from positive to negative or vice versa.

The zero crossing rate is calculated by counting the number of times the signal crosses the zero axis within a specified time frame. Here's a basic outline of how you can compute the ZCR for an audio signal:

Frame the Signal: Divide the audio signal into overlapping frames. Each frame typically consists of a fixed number of samples.

Count Zero Crossings: For each frame, count the number of times the signal crosses the zero axis.

Calculate the Zero Crossing Rate: Divide the total number of zero crossings by the total number of frames or by the total duration of the signal.

$$Z_n = \sum_{m=-\infty}^{\infty} \left| \text{sgn}[x(m)] - \text{sgn}[x(m-1)] \right| w(n-m)$$

Applications:

Speech and speaker detection: ZCR is often used as a feature in speech and speaker recognition systems. Different speakers and speech signals exhibit distinct ZCR patterns. Analyzing ZCR can help in distinguishing between different speech segments and identifying speakers.

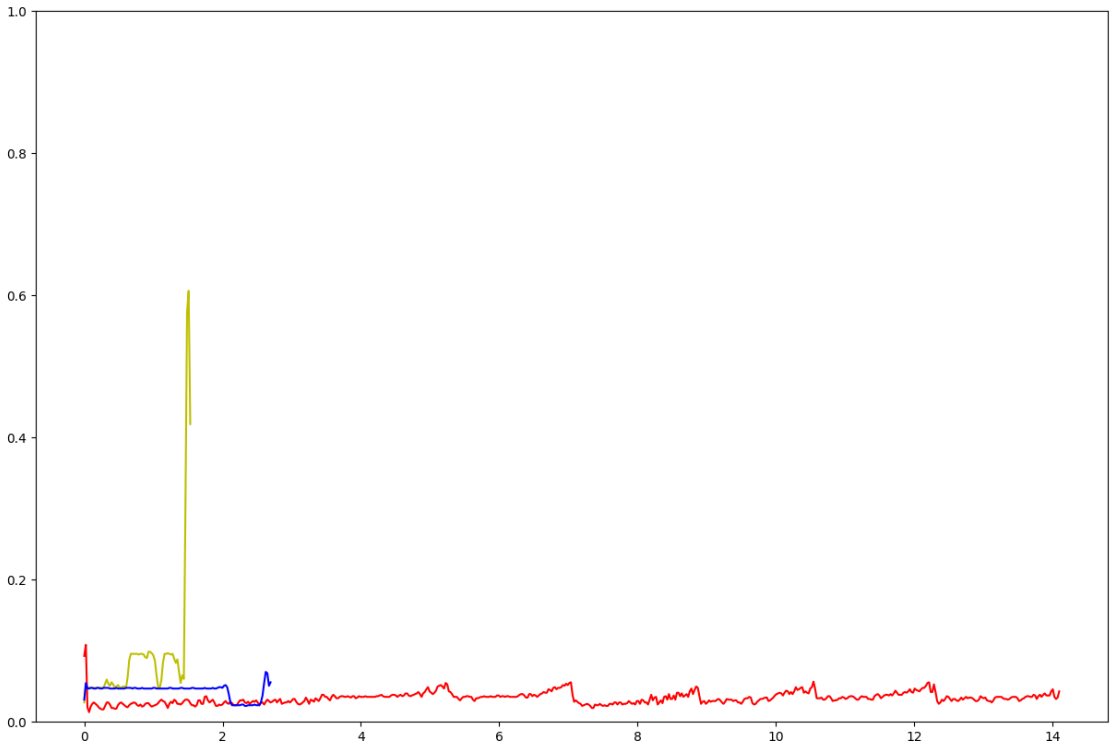
Pitch Detection: ZCR is sometimes used in combination with other features for pitch detection in audio signals. The rate of zero crossings can provide information about the frequency content of the signal and assist in estimating the pitch.

Emotion Detection: ZCR is explored in emotion recognition from speech signals. Changes in ZCR may be associated with variations in prosody and intonation, which can be indicative of different emotional states.

Code:

```
zcr_debussy = librosa.feature.zero_crossing_rate(debussy, frame_length=FRAME_SIZE,
hop_length=HOP_LENGTH)[0]
zcr_redhot = librosa.feature.zero_crossing_rate(redhot, frame_length=FRAME_SIZE,
hop_length=HOP_LENGTH)[0]
zcr_duke = librosa.feature.zero_crossing_rate(duke, frame_length=FRAME_SIZE,
hop_length=HOP_LENGTH)[0]
```

```
plt.figure(figsize=(15, 10))

plt.plot(t1, zcr_debussy, color="y")
plt.plot(t2, zcr_redhot, color="r")
plt.plot(t3, zcr_duke, color="b")
plt.ylim(0, 1)
plt.show()
```



4)Short-Time Energy (STE)

Short-Time Energy (STE) is a signal processing concept used to analyze the energy distribution in a signal over short time intervals. It is particularly useful in applications such as speech processing, music analysis, and other audio signal processing tasks. STE provides a way to observe changes in signal energy over time and is often used in the context of signal segmentation.

The first step involves dividing the input signal into short, overlapping time frames or windows. Each window captures a small portion of the signal, and typically, successive windows overlap to ensure a smooth representation.

Within each time window, the energy of the signal is computed. The energy is often calculated by squaring each sample in the window and summing up the squared values.

The calculated energy for each time window gives the Short-Time Energy for that specific interval. It provides a measure of how the energy of the signal changes over short durations.

Applications:

Voice Activity Detection: VAD is a crucial component in speech processing systems. STE can be used for voice activity detection, helping to identify segments of audio containing speech and distinguish them from non-speech segments.

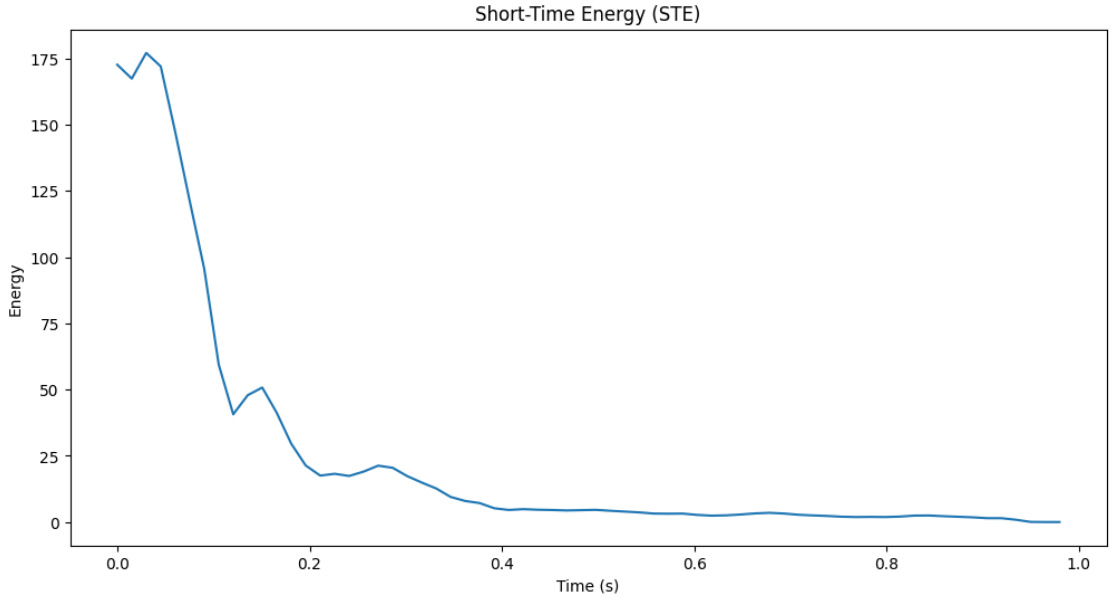
Audio Event Detection: STE is useful in audio event detection, where the goal is to identify specific events or sounds within an audio signal. Changes in energy levels can be indicative of the presence of events such as footsteps, claps, or door slams.

Music Analysis: In music processing, STE can help analyze musical signals. It can be applied to identify the onset of musical notes, segment different musical phrases, or detect changes in musical dynamics.

Code:

```
def calculateSTE(audio_signal, window_type, frame_length, hop_size):
    signal_new = []  # container for signal square
    win = Windowing(type=window_type)  # instantiate window function

    # compute signal square by frame
    for frame in FrameGenerator(audio_signal, frameSize=frame_length, hopSize=hop_size,
startFromZero=True):
        frame_new = frame**2
        signal_new.append(frame_new)

    # output the sum of squares (STE) for each frame
    return np.sum(signal_new, axis=1)

# Example usage and plotting
audio_signal, _ = librosa.load(debussy_file)
```

```
window_type = 'hann'
frame_length = 1024
hop_size = 512

ste_values = calculateSTE(audio_signal, window_type, frame_length, hop_size)

# Plot the Short-Time Energy
plt.figure(figsize=(12, 6))
plt.plot(np.arange(0, len(ste_values)) * hop_size / len(audio_signal), ste_values)
plt.title('Short-Time Energy (STE)')
plt.xlabel('Time (s)')
plt.ylabel('Energy')
plt.show()
```



5)Separation of Harmonic & Percussive Signals

On a very coarse level, many sounds can be categorized as either of the harmonic or percussive sound class. Harmonic sounds are sounds we perceive to have a specific pitch, whereas percussive sounds are often perceived as the result of two colliding objects. Percussive sounds tend to have a clear localization in time moreso than a particular pitch. More granular sound classes can be classified by its harmonic-percussive components ratio. For example, a note played on the piano has a percussive onset (marked by the hammer hitting the strings) preceding a harmonic tone (the result of the vibrating string).

The value of mapping both the time and phase content of a signal in distinguishing between harmonic and percussive components, makes the STFT spectral representation important. The time-frequency bin of the STFT for harmonic component of an input signal is expected to look more horizontal than vertical/time-dependent structure that is a percussive component.

Applications:

Music Transcription: HPSS is valuable in music transcription, the process of converting an audio recording into sheet music. Separating harmonic and percussive components can help in isolating the melody (harmonic) and rhythm (percussive) elements of the music, making it easier to transcribe.

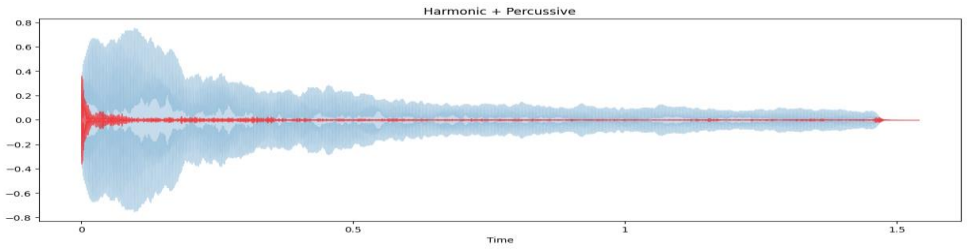
Music remixing and production: In music production and remixing, HPSS enables producers to isolate the melodic and rhythmic elements of a track. This separation provides flexibility in remixing by allowing producers to manipulate and enhance specific components independently.

Audio Compression: Separating harmonic and percussive components can be beneficial in audio compression applications. Different compression strategies can be applied to each component based on their characteristics, optimizing compression efficiency.

Music genre Classification: In singing voice detection, where the goal is to identify and isolate vocal elements in a musical piece, HPSS can be used to separate the harmonic singing voice from the accompanying percussive music.

Code:

```
y_harmonic, y_percussive = librosa.effects.hpss(y)
plt.figure(figsize=(15, 5))
librosa.display.waveshow(y_harmonic, sr=sr, alpha=0.25)
librosa.display.waveshow(y_percussive, sr=sr, color='red', alpha=0.5)
plt.title('Harmonic + Percussive')
plt.show()
```



6)Beat Extraction

Beat extraction is a signal processing technique crucial for identifying and extracting the rhythmic structure, or beats, from an audio signal. Beginning with onset detection, where sudden increases in energy mark significant musical events, the process moves to tempo estimation, determining the beats per minute. Beat tracking then refines beat positions based on the estimated tempo and actual onsets. This timeline of beat positions forms the basis for rhythm analysis, aiding in tasks such as music transcription and DJ applications. Beat information is valuable in automatic playlist generation, as it allows for dynamic playlists based on rhythmic compatibility between tracks. In music genre classification, beats contribute to capturing characteristic rhythmic patterns associated with different genres. Furthermore, beat extraction plays a role in audio effects syncing, ensuring effects align with the rhythmic elements of the music, creating a harmonious listening experience.

Applications:

DJ mixing and remixing: Beat extraction is vital for DJs, ensuring seamless transitions between tracks by synchronizing beats. This allows for smooth and consistent rhythm during live performances and remixing.

Music production and Rhythm analysis: In music production, beat extraction facilitates precise editing and arrangement of rhythmic elements. It also aids in rhythm analysis, providing insights into tempo and timing for tasks like music transcription.

Code:

```
def calculateSTE(audio_signal, window_type, frame_length, hop_size):
    signal_new = []  # container for signal square
    win = Windowing(type=window_type)  # instantiate window function

    # compute signal square by frame
    for frame in FrameGenerator(audio_signal, frameSize=frame_length, hopSize=hop_size,
startFromZero=True):
        frame_new = frame**2
        signal_new.append(frame_new)

    # output the sum of squares (STE) for each frame
    return np.sum(signal_new, axis=1)

audio_signal, _ = audio_signal, _ = librosa.load(debussy_file)

window_type = 'hann'
frame_length = 1024
hop_size = 512

ste_values = calculateSTE(audio_signal, window_type, frame_length, hop_size)

# Plot the Short-Time Energy using seaborn
```

```
plt.figure(figsize=(12, 6))
plt.plot(np.arange(0, len(ste_values)) * hop_size / len(audio_signal), ste_values)
plt.title('Short-Time Energy (STE)')
plt.xlabel('Time (s)')
plt.ylabel('Energy')

# Example seaborn barplot
beat_nums = [1, 2, 3, 4]
beat_time_diff = [0.5, 0.3, 0.7, 0.4]

ax = plt.gca()
ax.set_ylabel("Time difference (s)")
ax.set_xlabel("Beats")

# Use barplot correctly
g = sns.barplot(x=beat_nums, y=beat_time_diff, palette="rocket", ax=ax)
g.set(xticklabels=[])
plt.show()
```

<ipython-input-51-d5007ed90cd4>:42: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

```
g = sns.barplot(x=beat_nums, y=beat_time_diff, palette="rocket", ax=ax)
```