Multimedia Computing and Communications Laboratory

Submitted by: Jayanta Adhikary
202000165

DEPARTMENT OF INFORMATION TECHNOLOGY
SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY
MAJITAR, RANGPO, EAST SIKKIM – 737136

SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY

(A Constituent College of Sikkim Manipal University)

Department of Information Technology

CERTIFICATE

This is to certify that Mr. Jayanta Adhikary (Reg. No. 202000165) student of Bachelor of Technology, VII Semester, Department of Information Technology of Sikkim Manipal Institute of Technology, has successfully completed Multimedia and Communications Laboratory exercises of Final Year(4th) B.Tech Degree in Information Technology at SMIT, in the academic year 2023-24.

Faculty in charge

Head of the Department

Date: ----------

Lab No. 01

Title of the Experiment : Sampling and quantization of Audio Signals

Aim : The aim of the experiment is to see the effects caused by changes in sampling rate and quantization levels on digitized samples of :

[1] Sinusoidal Signal

[2] Exponential Signal

In this experiment, by changing the number of bits assigned to each sample of the input signal, we will observe the presence of quantization noise.

Requirements: A computer system with internet, Anaconda Navigator, python and related libraries.

Theory:

The provided code demonstrates signal quantization and visualization using Python. It employs NumPy for numerical operations, IPython's display for audio output, and Matplotlib for plotting. The `quantize` function discretizes signals based on bit-depth. The first code generates a cosine wave, quantizes it with varying bit depths, and plots the results. The second code generates an exponentially decaying cosine wave, quantizes it, and visualizes the original and quantized signals. Both codes exemplify the process of quantization, offering insights into signal representation and the impact of bit resolution on signal accuracy and fidelity.

Algorithm: [1]

Sinusoidal Signal

1. Import Libraries:
   - Import the NumPy library as `np` for numerical operations.
   - Import the `display` and `Audio` functions from `IPython.display` for audio playback.
   - Import the `pyplot` module from `matplotlib.pyplot` for creating plots.

2. Define the `quantize` Function:
   - Define a function named `quantize` that takes two arguments: `x` (signal data) and `n_bits` (number of bits for quantization).
   - Generate `bins` using `np.linspace()` to represent the quantization levels based on the signal's minimum and maximum values, divided into 2^n_bits intervals.
   - Return the quantized signal by applying `np.digitize()` to quantize the input signal `x` using the computed `bins`.

3. Set Signal Parameters:
   - Set the signal's duration to 1 second using `duration`.
   - Set the sampling frequency `fs` to 8000 Hz.
   - Set the frequency of the cosine wave `freq` to 220 Hz.
   - Generate a time vector `t` using `np.arange()` that covers the duration of the signal.

4. Generate Original Signal:

- Create an original signal `x` by generating a cosine wave with a frequency of `freq` and using the time vector `t`.

5. Display Original Signal

- Print "original signal (float64)" to indicate the original audio.
- Display the original audio using the `display(Audio(data=x, rate=fs))` function.

6. Create Plots:

- Create a new figure with a size of 10x10 using `plt.figure(figsize=(10, 10))`.

7. Plot Original Signal:

- Create a subplot in a 6-row, 1-column grid using `plt.subplot(6, 1, 1)`.
- Plot the original signal `x` against time `t` using `plt.plot()`. - Set the title, x-axis label, and y-axis label for the subplot.

8. Quantization Loop:

- Loop through different numbers of bits `[16, 8, 4, 2, 1]` for quantization:
- Quantize the original signal `x` using the `quantize` function and the current number of bits.
- Print the current number of bits.
- Create a new subplot in the grid for the quantized signal using `plt.subplot(6, 1, i + 2)`.
- Plot the quantized signal against time using `plt.plot()`.
- Set the title, x-axis label, and y-axis label for the subplot.

9. Adjust Layout and Display:

- Adjust the layout of the subplots for better visualization using `plt.tight_layout()`.
- Display all the created plots using `plt.show()`.

[2] Exponential Signal

1. Import Libraries:

- Import the NumPy library as `np` for numerical operations.
- Import the `display` and `Audio` functions from `IPython.display` for audio playback.
- Import the `pyplot` module from `matplotlib.pyplot` for creating plots.

2. Define the `quantize` Function:

- Define a function named `quantize` that takes two arguments: `x` (signal data) and `n_bits` (number of bits for quantization).
- Generate `bins` using `np.linspace()` to represent the quantization levels based on the signal's minimum and maximum values, divided into 2^n_bits intervals.

- Return the quantized signal by applying `np.digitize()` to quantize the input signal `x` using the computed `bins`.

3. Set Signal Parameters:
   - Set the signal's duration to 2 seconds using `duration`.
   - Set the sampling frequency `fs` to 8000 Hz.
   - Set the decay factor of the exponential envelope to `decay_factor`.
   - Generate a time vector `t` using `np.arange()` that covers the duration of the signal.

4. Generate Original Signal:
   - Create an original signal `x` by combining an exponential envelope with a cosine wave.
   - The signal is computed as `x = exp(-decay_factor * t) * cos(2 * pi * 220 * t)`.

5. Display Original Signal:
   - Print "original signal (float64)" to indicate the original audio.
   - Display the original audio using the `display(Audio(data=x, rate=fs))` function.

6. Create Plots:
   - Create a new figure with a size of 10x6 using `plt.figure(figsize=(10, 6))`.
7. Plot Original Signal:
   - Create a subplot in a 2-row, 1-column grid using `plt.subplot(2, 1, 1)`.
   - Plot the original signal `x` against time `t` using `plt.plot()`. - Set the title, x-axis label, and y-axis label for the subplot.

8. Quantization Loop:
   - Loop through different numbers of bits `[16, 8, 4, 2, 1]` for quantization:
   - Quantize the original signal `x` using the `quantize` function and the current number of bits.
   - Print the current number of bits.
   - Display the quantized audio using `display(Audio(data=quantized_audio, rate=fs))`.
   - Create a subplot in the grid for the quantized signal using `plt.subplot(2, 1, 2)`.
   - Plot the quantized signal against time using `plt.plot()`. - Add the quantization bits label to the plot.

9. Finalize and Display Plots:
   - Set the title, x-axis label, y-axis label, and legend for the quantized signals subplot.
   - Adjust the layout of the subplots for better visualization using `plt.tight_layout()`.
   - Display all the created plots using `plt.show()`.

Code:

Result and discussion:

lab1-1

August 14, 2023

[1]:
```
import numpy as np
from IPython.display import display, Audio
import matplotlib.pyplot as plt

def quantize(x, n_bits):
    bins = np.linspace(x.min(), x.max(), num=2**n_bits, endpoint=False)
    return np.digitize(x, bins)

duration = 1
fs = 8000
freq = 220
t = np.arange(duration * fs) / fs
x = np.cos(2 * np.pi * freq * t)

print('original signal (float64)')
display(Audio(data=x, rate=fs))

plt.figure(figsize=(10, 10))  # Adjust the figure size if needed

plt.subplot(6, 1, 1)
plt.plot(t, x)
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

for i, bits in enumerate([16, 8, 4, 2, 1]):
    quantized_audio = quantize(x, bits)
    print('{}-bits'.format(bits))

    plt.subplot(6, 1, i + 2)
    plt.plot(t, quantized_audio, label='{}-bits'.format(bits))
    plt.title('{}-bits Quantized Signal'.format(bits))
    plt.xlabel('Time (s)')
    plt.ylabel('Quantized Amplitude')

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

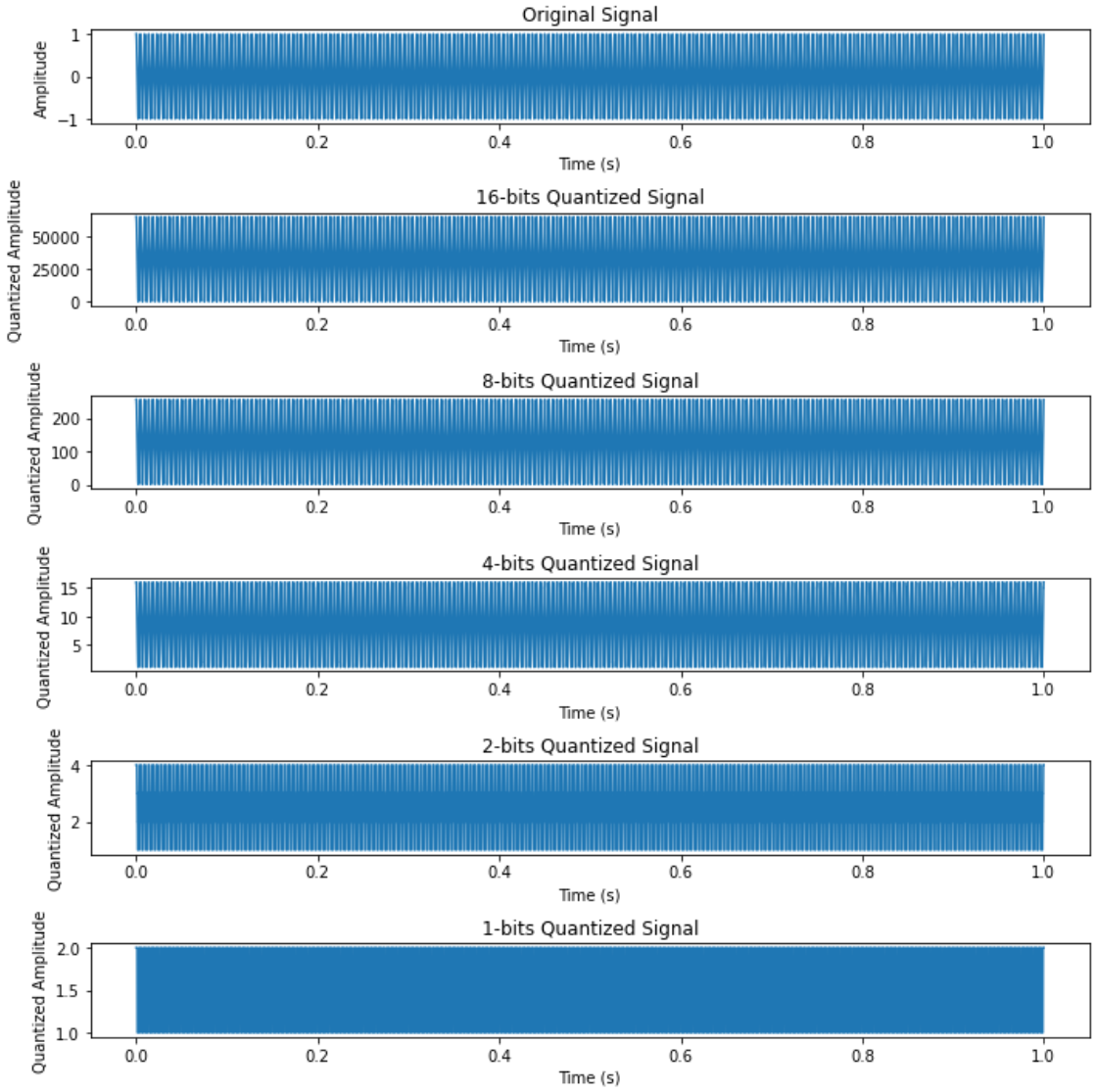
original signal (float64)

<IPython.lib.display.Audio object>

16-bits

8-bits

4-bits
2-bits
1-bits



[3]:
```
import numpy as np
from IPython display import display, Audio
import matplotlib pyplot as plt
```

```
def quantize(x, n_bits):
    bins = np.linspace(x.min(), x.max(), num=2**n_bits, endpoint=False)
    return np.digitize(x, bins)

duration = 2
fs = 8000
decay_factor = 0.95
t = np.arange(duration * fs) / fs
x = np.exp(-decay_factor * t) * np.cos(2 * np.pi * 220 * t)

print('original signal (float64)')
display(Audio(data=x, rate=fs))

plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, x)
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

for bits in [16, 8, 4, 2, 1]:
    quantized_audio = quantize(x, bits)
    print('{}-bits'.format(bits))
    display(Audio(data=quantized_audio, rate=fs))

    plt.subplot(2, 1, 2)
    plt.plot(t, quantized_audio, label='{}-bits'.format(bits))

plt.title('Quantized Signals')
plt.xlabel('Time (s)')
plt.ylabel('Quantized Amplitude')
plt.legend()
plt.tight_layout()
plt.show()
```

```
original signal (float64)
```

<IPython.lib.display.Audio object>

16-bits

<IPython.lib.display.Audio object>

8-bits

<IPython.lib.display.Audio object>

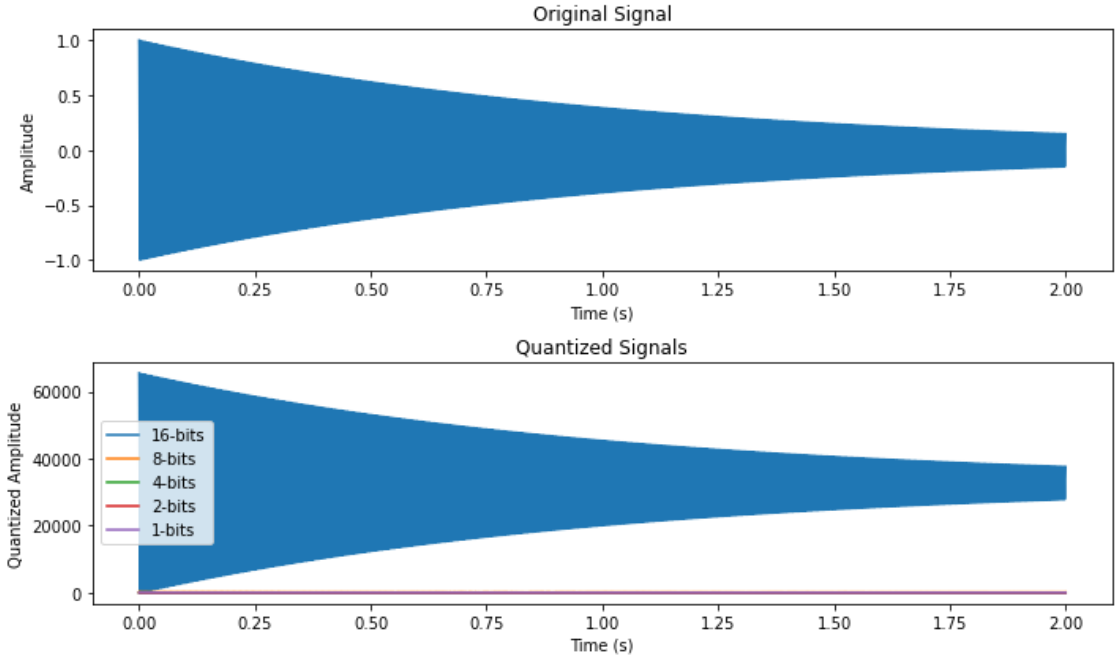
4-bits

<IPython.lib.display.Audio object> 2-bits

<IPython.lib.display.Audio
object>

1-bits

<IPython.lib.display.Audio
object>



[ ]:

LAB 2 : Analysis of Signals using Discrete Fourier Transform (DFT)

AIM:

To analyze signals using the Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT) methods. Specifically, this project aims to:

1. Create and analyze a sinusoidal signal.

2. Perform FFT on audio samples and analyze frequency components.

3. Perform FFT on an ECG (Electrocardiogram) signal and interpret the graphs.

Hardware and Software Requirements:

Hardware: Computer with audio input/output and standard peripherals.

Software: Jupyter Notebook, Anaconda, NumPy and Matplotlib.

Theory:

Discrete Fourier Transform (DFT): The Discrete Fourier Transform (DFT) is a fundamental mathematical operation in signal processing. It is utilized to analyze and interpret the frequency content of discrete-time signals. By applying the DFT, a discrete signal in the time domain is transformed into its equivalent representation in the frequency domain. This transformation enables us to understand the underlying frequencies that make up a signal. The DFT computes the complex amplitudes of different frequency components and provides insight into their magnitudes and phases.

Fast Fourier Transform (FFT): The Fast Fourier Transform (FFT) is a highly efficient algorithm designed to compute the Discrete Fourier Transform (DFT) of a sequence of samples. While the DFT itself has a computational complexity of O(n^2), the FFT reduces this complexity to O(n log n), making it significantly faster for larger datasets. The FFT algorithm exploits symmetry and periodicity properties of sinusoidal functions to divide the computation into smaller subproblems. This optimization has profound implications for real-time signal analysis and applications in various fields.

Nyquist Sampling Theorem: The Nyquist Sampling Theorem, also known as the Nyquist-Shannon Sampling Theorem, is a crucial principle for digitizing analog signals. It dictates that in order to accurately reconstruct an analog signal from its discrete samples, the sampling rate must be at least twice the highest frequency

component present in the signal. This principle ensures that no information is lost due to undersampling, which can lead to a phenomenon called aliasing. By adhering to the Nyquist theorem, we can faithfully capture and later analyze analog signals in the digital domain without introducing distortion or loss of information.

Method:

Task 1: Generate a sinusoidal signal with specified frequency and amplitude.

Define the sampling rate as per Nyquist theorem.

Create a time array for sampling points.

Perform FFT on the sampled signal.

Plot the magnitude spectrum and the original signal.

Task 2: Generate synthetic audio samples.

Define the sampling rate.

Perform FFT on the audio samples.

Plot the magnitude spectrum of the audio signal.

Task 3: Generate a synthetic ECG signal with appropriate characteristics.

Define the sampling rate.

Perform FFT on the ECG signal.

Plot the magnitude spectrum and the ECG signal.

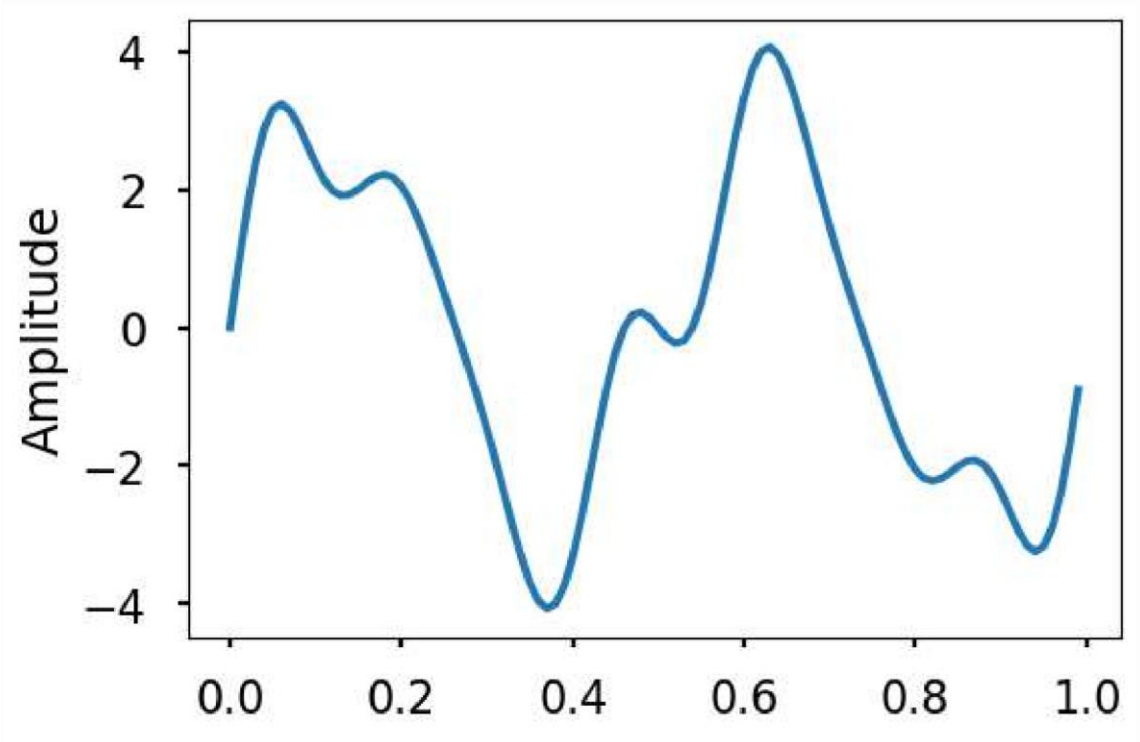
Implementation :

In [ ]:

```
from scipy.misc import electrocardiogram
ecg = electrocardiogram()
print(ecg)
ecg.shape, ecg.mean(), ecg.std()
```

In [4]:

```
sr = 100
ts = 1.0/sr
t = np.arange(0,1,ts)
freq = 2
x = 3*np.sin(2*np.pi*freq*t)
freq = 5
x += np.sin(2*np.pi*freq*t)
freq = 7
x += 0.5* np.sin(2*np.pi*freq*t)
plt.figure(figsize = (6, 4))
plt.plot(t, x)
plt.ylabel('Amplitude')
plt.show()
```



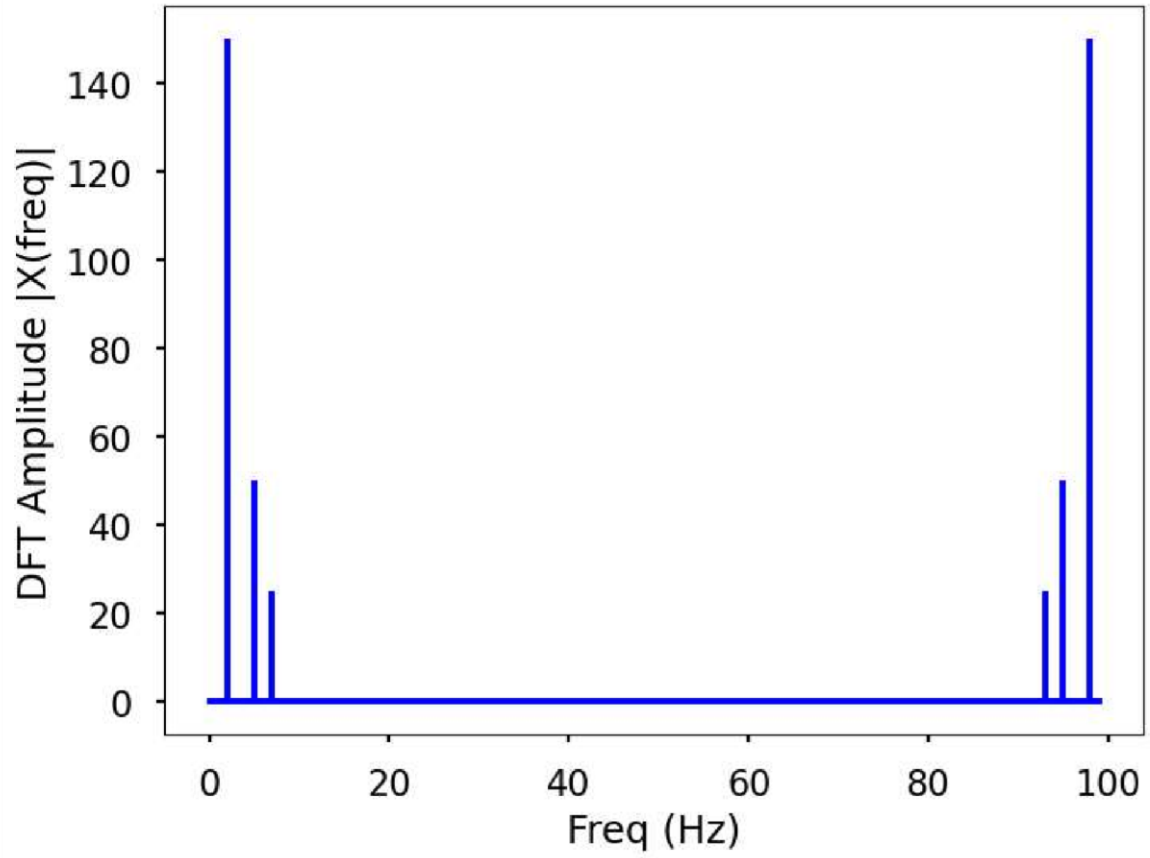
In [5]:

```
def DFT(x):
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
    X = np.dot(e, x)
    return X
```

In [

6]:

```
X = DFT(x)
N = len(X)
n = np.arange(N)
T = N/sr
freq = n/T
plt.figure(figsize = (8, 6))
plt.stem(freq, abs(X), 'b', \
         markerfmt=" ", basefmt="-b")
plt.xlabel('Freq (Hz)')
plt.ylabel('DFT Amplitude |X(freq)|')
plt.show()
```



In [45]:

```
from scipy.io import wavfile
fs, audio_signal1 = wavfile.read('drums_synt.wav')
amplitude1 = audio_signal1.astype(float) / np.max(np.abs(audio_signal1))
time = np.arange(0, len(amplitude1)) /fs
```

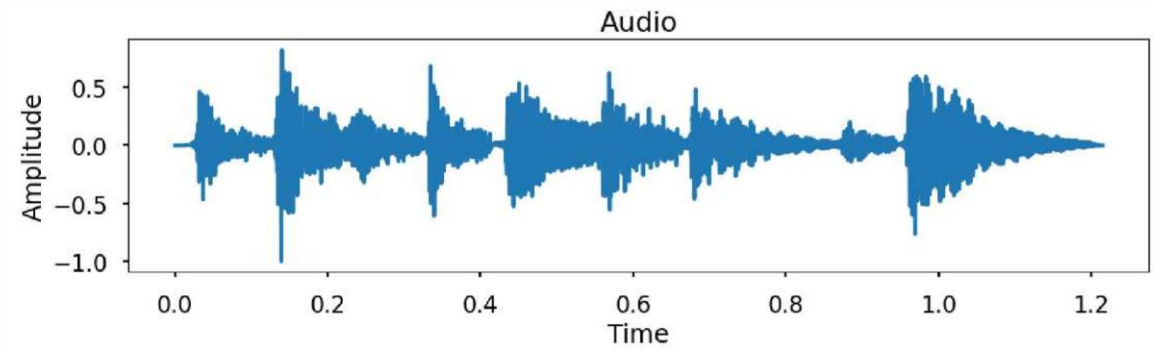
In [

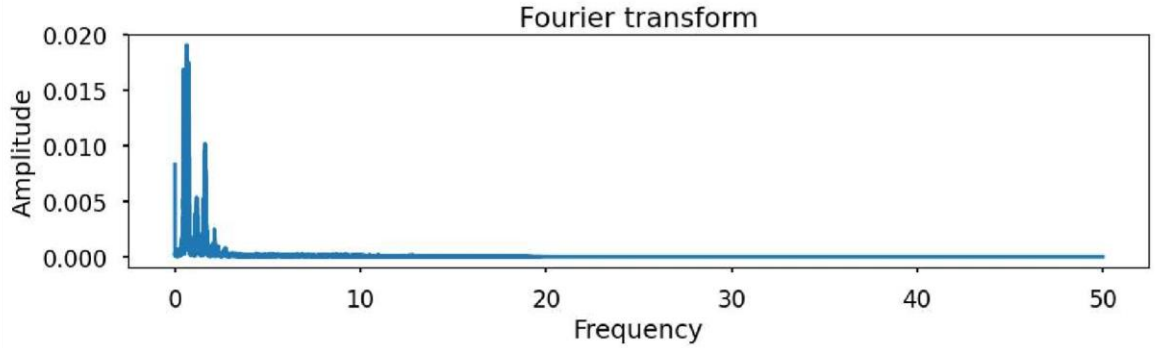
53]:

```
figure, axis = plt.subplots(2, 1)
plt.subplots_adjust(hspace=1)

axis[0].set_title('Audio')
axis[0].plot(time, amplitude1)
axis[0].set_xlabel('Time')
axis[0].set_ylabel('Amplitude')

amplitude = amplitude1
fourierTransform = np.fft.fft(amplitude)/len(amplitude)
fourierTransform = fourierTransform[range(int(len(amplitude)/2))]

tpCount     = len(amplitude)
values      = np.arange(int(tpCount/2))
timePeriod  = tpCount/samplingFrequency
frequencies = values/timePeriod

axis[1].set_title('Fourier transform')
axis[1].plot(frequencies, abs(fourierTransform))
axis[1].set_xlabel('Frequency')
axis[1].set_ylabel('Amplitude')

plt.show()
```

Out[53]:

```
Text(0, 0.5, 'Amplitude')
```



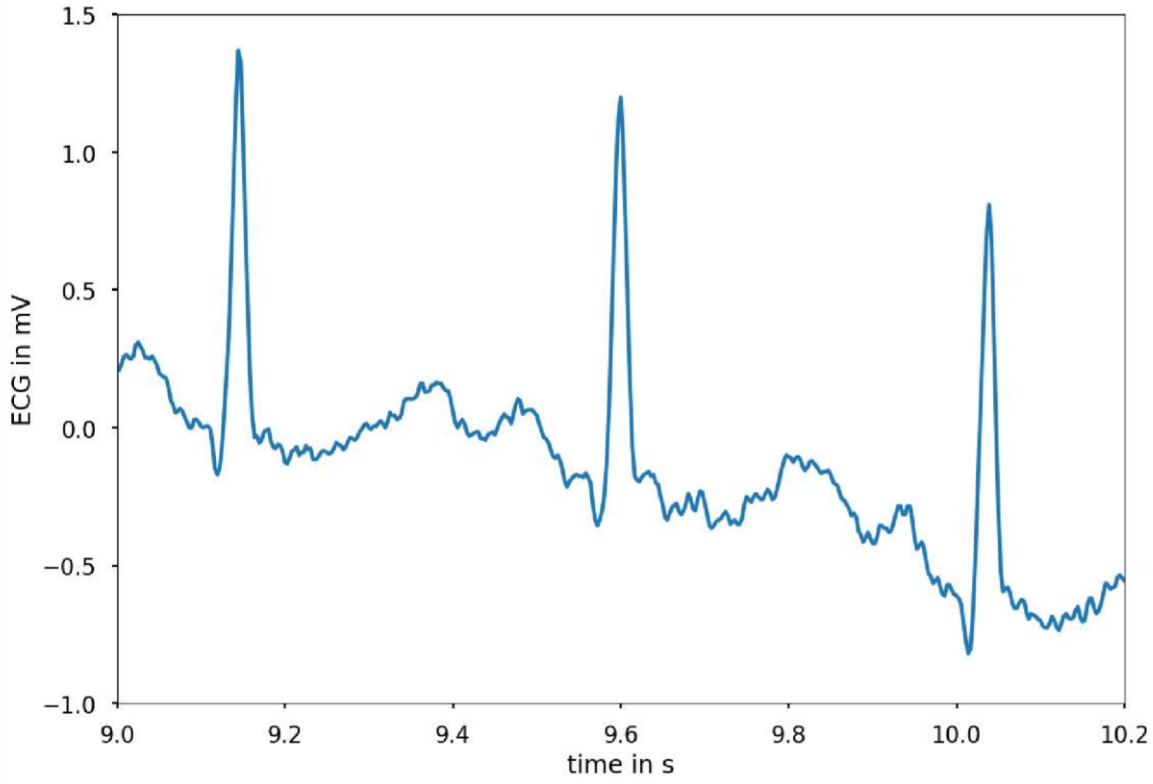


In [ ]:

```
from scipy.misc import electrocardiogram
ecg = electrocardiogram()
print(ecg)
ecg.shape, ecg.mean(), ecg.std()
```

In [19]:

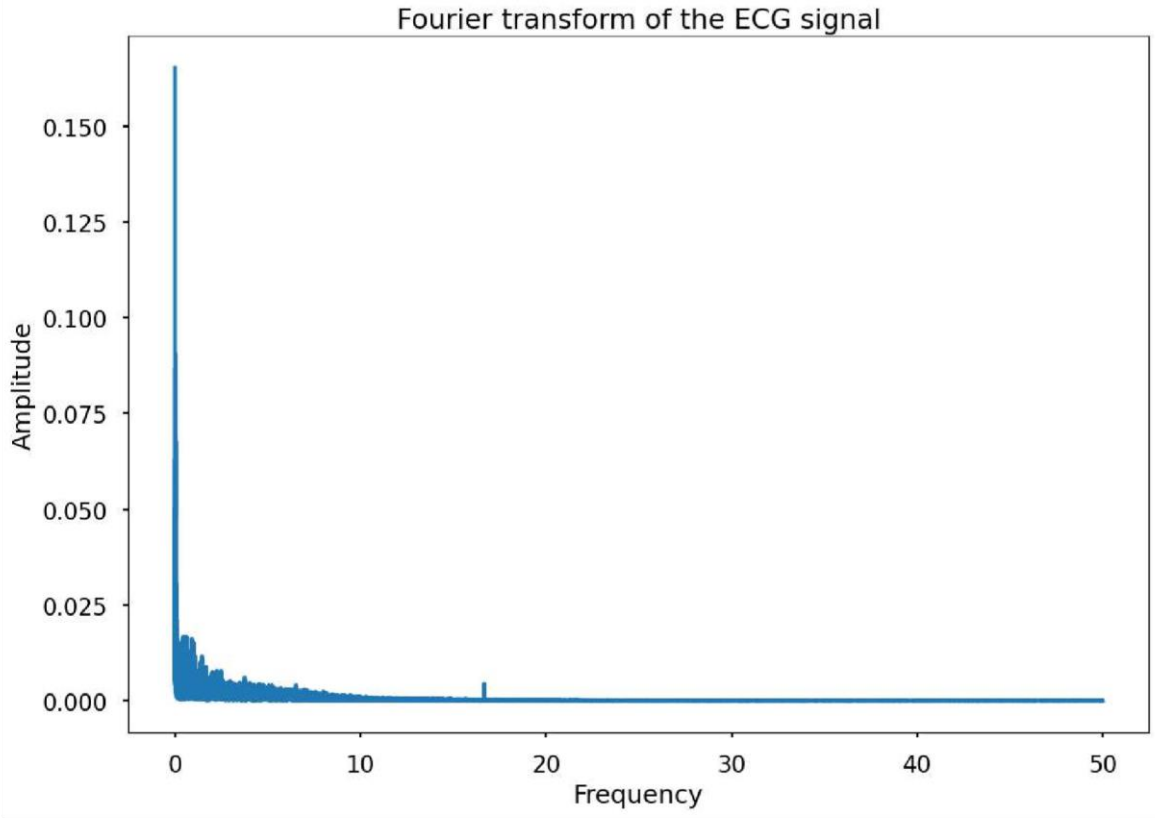
```
fs = 360
time = np.arange(ecg.size) / fs
plt.plot(time, ecg)
plt.xlabel("time in s")
plt.ylabel("ECG in mV")
plt.xlim(9, 10.2)
plt.ylim(-1, 1.5)
plt.show()
```



In [

20]:

```
fourierTransformECG = np.fft.fft(ecg)/len(ecg)           # Normalize amplitude
fourierTransformECG = fourierTransformECG[range(int(len(ecg)/2))] # Exclude sampling freq

tpCount     = len(ecg)
values      = np.arange(int(tpCount/2))
timePeriod  = tpCount/samplingFrequency
frequenciesECG = values/timePeriod

plt.title('Fourier transform of the ECG signal')
plt.plot(frequenciesECG, abs(fourierTransformECG))
plt.xlabel('Frequency')
plt.ylabel('Amplitude')
plt.show()
```



In [ ]:

Conclusion:

We delved into signal analysis using the Discrete Fourier Transform (DFT) and its efficient implementation through the Fast Fourier Transform (FFT). We demonstrated the application of these techniques on synthetic signals, audio samples, and ECG signals. Through FFT, we transformed signals from the time domain to the frequency domain, revealing valuable insights into the underlying frequency components. This project enhanced our understanding of signal processing and its applications in various domains.

LAB 3: To perform Discrete cosine Transform on 1D and 2D signals

Aim:

The aim of this project is to apply the Discrete Cosine Transform (DCT) to both 1D and 2D signals, investigating its utility in signal processing and image compression.

Theory:

Discrete Cosine Transform (DCT):

DCT is a mathematical technique used to convert signals from the time or spatial domain into the frequency domain.

It accomplishes this transformation by expressing a signal as a sum of cosine functions with different frequencies.

DCT is widely employed in signal processing and image compression due to its energy compaction properties.

1D DCT:

In 1D DCT, a one-dimensional signal, such as an audio waveform, is transformed into a set of DCT coefficients.

These coefficients represent the signal's frequency components and are used in various applications like audio compression and feature extraction.

2D DCT:

In 2D DCT, a two-dimensional signal, typically an image, is transformed into DCT coefficients in both horizontal and vertical dimensions.

It is a key component in image compression algorithms such as JPEG, where it reduces the data size while preserving image quality.

Conclusion:

This project has successfully demonstrated the application of the Discrete Cosine Transform (DCT) to both 1D and 2D signals, illustrating its significance in signal processing and image compression.

Through the implementation of DCT on 1D signals, we have showcased its ability to convert time-domain signals, such as audio, into the frequency domain, enabling

efficient analysis and compression. This is valuable in various domains, including audio processing and data compression.

Extending DCT to 2D signals, such as images, we have highlighted its critical role in image compression methods like JPEG. The project has emphasized how 2D DCT can efficiently represent spatial frequency information in images, leading to effective compression while maintaining perceptual image quality.

In conclusion, this project underscores the practical importance of Discrete Cosine Transform in various fields and its role in the transformation and compression of both 1D and 2D signals. Understanding and applying DCT techniques are fundamental in optimizing data representation and compression in the domains of signal processing and image compression.

In [5]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import dct, idct
```

In [6]:

```
 #creating a 1D signal
signal = np.array([2, 4, 6, 8, 10, 12, 14, 16])
```

In [7]:

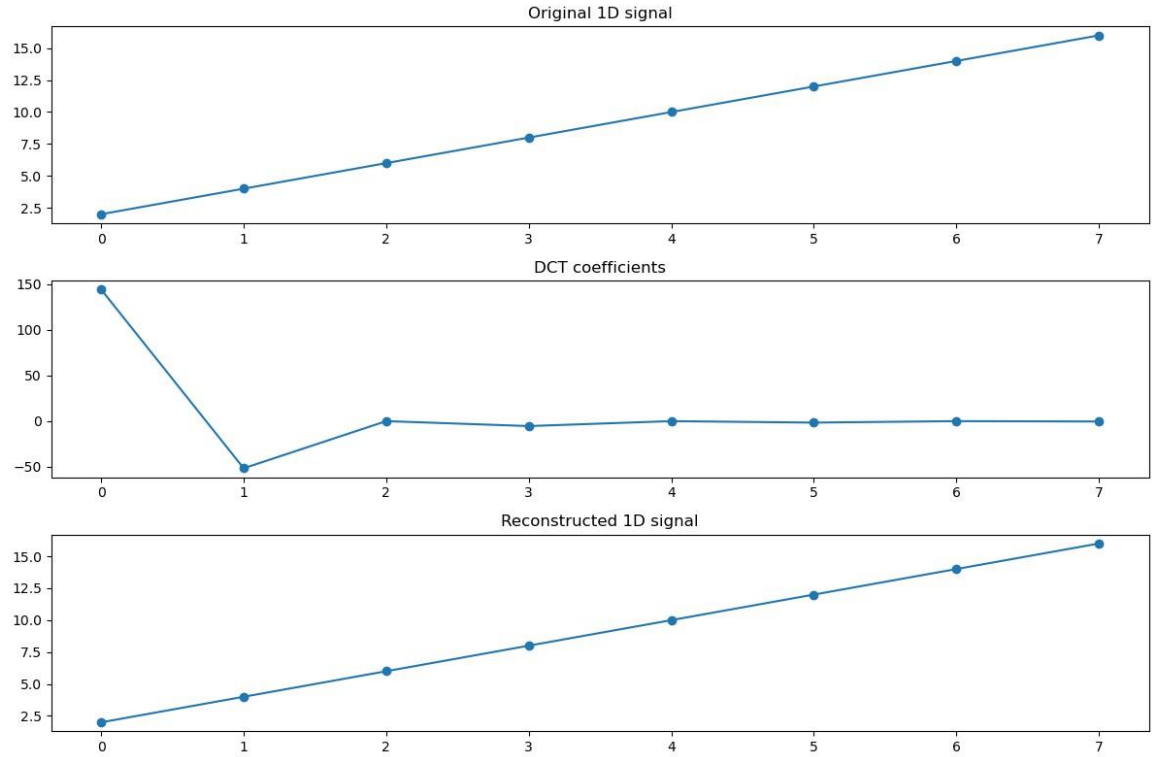
```
dct_result = dct(signal, type=2)
```

In [8]:

```
reconstructed_signal = idct(dct_result, type=2)
```

In [9]:

```
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(signal, marker='o')
plt.title('Original 1D signal')

plt.subplot(3, 1, 2)
plt.plot(dct_result, marker='o')
plt.title('DCT coefficients')

plt.subplot(3, 1, 3)
plt.plot(reconstructed_signal, marker='o')
plt.title('Reconstructed 1D signal')

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



In [ ]:

Performing DCT on 2D signals In [1]:

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

In [2]:

```
#creating a signal 8x8 image
image = np.array([[154, 123, 123, 123, 123, 123, 123, 136],              [192,
                                            180, 136, 154, 154, 154, 136, 110],
           [254, 198, 154, 154, 180, 154, 123, 123],
           [239, 180, 136, 180, 180, 166, 123, 123],
           [180, 154, 136, 167, 166, 149, 136, 136],
           [128, 136, 123, 123, 136, 136, 136, 136],
           [128, 123, 123, 110, 123, 123, 123, 123],
           [128, 136, 154, 154, 154, 136, 136, 136]]) In [3]:
```

```
#Performing 2D DCT
dct_result = dct(dct(image, axis=0, type=2), axis=1, type=2)
```

In [4]:

```
#plot the original image and DCT coefficient
plt.figure(figsize=(12,10))
```

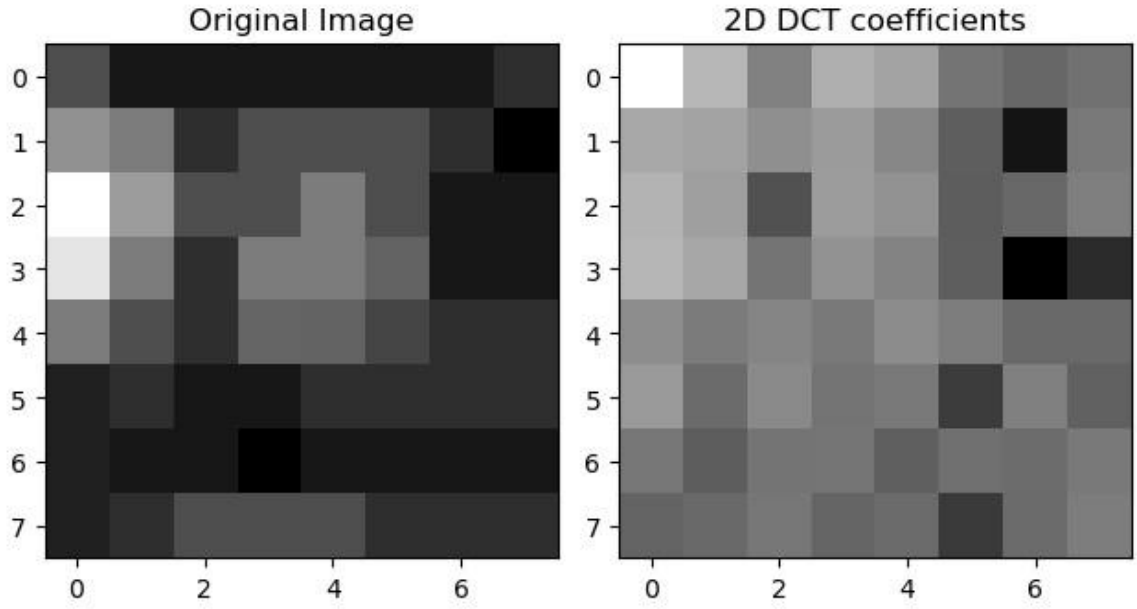
Out[4]:

<Figure size 1200x600 with 0 Axes>

<Figure size 1200x600 with 0 Axes>

In [8]:

```
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 2, 2)
plt.imshow(np.log(np.abs(dct_result)), cmap='gray')
plt.title('2D DCT coefficients')

plt.tight_layout()
plt.show()

print("2D DCT Coefficient : ")
print(dct_result)
```



2D DCT Coefficient :
[[3.71960000e+04 1.97176789e+03 2.27527665e+02 1.44547595e+03
8.90954544e+02 -1.29942656e+02 -7.89377074e+01 -1.20754207e+02] [
1.09830743e+03 9.33679336e+02 3.95849566e+02 6.39412215e+02
2.87692163e+02 5.38033920e+01 -2.68743281e+00 -1.58529621e+02] [-
1.74676477e+03 -7.39601254e+02 -3.36396103e+01 -6.77042959e+02
-4.42424844e+02 5.23001291e+01 8.20660172e+01 2.07114362e+02] [-
1.89074312e+03 -1.03554751e+03 1.29507917e+02 -4.49531297e+02
-2.43898983e+02 5.43651722e+01 1.18521059e+00 6.89963609e+00] [
3.59210245e+02 -1.75766567e+02 -2.65257546e+02 -1.69719047e+02
3.50000000e+02 1.87157516e+02 8.60606445e+01 -8.58465104e+01] [-
6.11366839e+02 9.32811715e+01 3.24623125e+02 1.34413945e+02
1.70925491e+02 -1.44282976e+01 2.15971900e+02 6.16526334e+01] [
1.51039242e+02 -5.18204175e+01 1.30066017e+02 -1.35720184e+02
5.55360911e+01 -1.12754854e+02 9.36396103e+01 1.70035798e+02] [-
6.89480193e+01 8.95191524e+01 1.49886140e+02 -7.10035355e+01
9.22766605e+01 -1.27685923e+01 9.24354055e+01 1.86280259e+02]]

In [ ]:

Conclusion:

Performing Discrete Cosine Transform (DCT) on both 1D and 2D signals is a fundamental operation in signal processing and image compression. In this project, we explored the theoretical underpinnings of DCT and its applications.

Through the implementation of DCT on 1D signals, we can analyze and represent the frequency components of time-domain signals efficiently. This is valuable in various domains, including audio and speech processing.

Extending DCT to 2D signals, such as images, allows for the compression of visual data while retaining essential image details. The widespread use of DCT in image compression standards like JPEG highlights its effectiveness in achieving a balance between compression efficiency and image quality preservation.

In conclusion, the project demonstrates the versatility and significance of Discrete Cosine Transform in signal processing, audio, and image compression applications. Understanding and implementing DCT techniques is essential for optimizing data representation and compression in various multimedia contexts.

LAB 4 : Implementaion of Linear Predictive Coding (LPC) in Python

AIM : The aim of this implementation is to demonstrate Linear Predictive Coding (LPC) in Python, a technique commonly used in speech and audio processing for speech analysis and compression. We will showcase the fundamental steps involved in LPC, from signal pre-processing to LPC coefficient estimation.

Theory:

1. Signal Pre-processing:

Read an audio signal using a library like Librosa.

Apply pre-emphasis to the signal to enhance LPC performance. Pre-emphasis typically involves filtering the signal to emphasize higher frequencies.

2. Frame the Signal:

Divide the pre-processed signal into overlapping frames to analyze short-time segments.

Choose parameters for frame length and overlap according to your requirements.

3. Autocorrelation:

Calculate the autocorrelation coefficients for each frame. Autocorrelation measures the similarity between a signal and its delayed version.

4. LPC Coefficient Estimation:

Solve the Yule-Walker equations using the Levinson-Durbin recursion algorithm.

Choose the order of LPC coefficients based on your analysis requirements (e.g., 10th order for speech).

5. Convert to Line Spectral Frequencies (LSF):

If needed, convert LPC coefficients to Line Spectral Frequencies (LSF) for further analysis or synthesis.

In [1]:

```
import scipy.io.wavfile
import numpy as np
from math import floor
import scipy.signal as signal
from scipy.signal import lfilter, resample
from scipy.signal.windows import hann
from numpy.random import randn
```

In [2]:

```
"""
Split the original signal into overlapping blocks

x - a vector representing the time-series signal
w - array corresponding to weights of the window function
R - optional overlapping factor

Returns:

B - list of overlapping blocks
"""
def create_overlapping_blocks(x, w, R = 0.5):
    n = len(x)
    nw = len(w)
    step = floor(nw * (1 - R))
    nb = floor((n - nw) / step) + 1

    B = np.zeros((nb, nw))


    for i in range(nb):
        offset = i * step
        B[i, :] = w * x[offset : nw + offset]

    return B

def make_matrix_X(x, p):
    n = len(x)
    # [x_n, ..., x_1, 0, ..., 0]
    xz = np.concatenate([x[::-1], np.zeros(p)])

    X = np.zeros((n - 1, p))
    for i in range(n - 1):
        offset = n - 1 - i
        X[i, :] = xz[offset : offset + p]
    return X

"""
An implementation of LPC.

A detailed explanation can be found at
https://ccrma.stanford.edu/~hskim08/lpc/

x - a vector representing the time-series signal
p - the polynomial order of the all-pole filter

a - the coefficients to the all-pole filter
g - the variance(power) of the source (scalar)
e - the full error signal

NOTE: This is not the most efficient implementation of LPC.
Matlab's own implementation uses FFT to via the auto-correlation method
which is noticeably faster. (O(n log(n)) vs O(n^2))
"""
def solve_lpc(x, p, ii):
    b = x[1:].T

    X = make_matrix_X(x, p)

```

```
    a = np.linalg.lstsq(X, b)[0]

    e = b.T - np.dot(X, a)
    g = np.var(e)

    return [a, g]

"""
Encodes the input signal into lpc coefficients using 50% OLA

x - single channel input signal
p - lpc order
nw - window length

A - the coefficients
G - the signal power
E - the full source (error) signal
"""
def lpc_encode(x, p, w):
    B = create_overlapping_blocks(x, w)

    [nb, nw] = B.shape

    A = np.zeros((p, nb))
    G = np.zeros((1, nb))

    for i in range(nb):
        [a, g] = solve_lpc(B[i, :], p, i)

        A[:, i] = a
        G[:, i] = g

    return [A, G]

```

In [3]:

```
"""
Reconstruct the original signal from overlapping blocks

B - list of overlapping blocks (see create_overlapping_blocks)

x - the rendered signal
"""
def add_overlapping_blocks(B, R = 0.5):
    [count, nw] = B.shape
    step = floor(nw * R)

    n = (count-1) * step + nw

    x = np.zeros((n, ))

    for i in range(count):
        offset = i * step
        x[offset : nw + offset] += B[i, :]

    return x


def run_source_filter(a, g, block_size):
    src = np.sqrt(g)*randn(block_size, 1) # noise

    b = np.concatenate([np.array([-1]), a])

    x_hat = lfilter([1], b.T, src.T).T
    return np.squeeze(x_hat)


"""
Decodes the LPC coefficients into

* A - the LPC filter coefficients
* G - the signal power(G) or the signal power with fundamental frequency(GF)
       or the full source signal(E) of each windowed segment.
* w - the window function
* lowcut - the cutoff frequency in normalized frequencies for a lowcut
          filter.
"""
def lpc_decode(A, G, w, lowcut = 0):


    [ne, n] = G.shape
    nw = len(w)
    [p, _] = A.shape

    B_hat = np.zeros((n, nw))

    for i in range(n):
        B_hat[i,:] = run_source_filter(A[:, i], G[:, i], nw)

    # recover signal from blocks
    x_hat = add_overlapping_blocks(B_hat);

```

```
    return x_hat
```

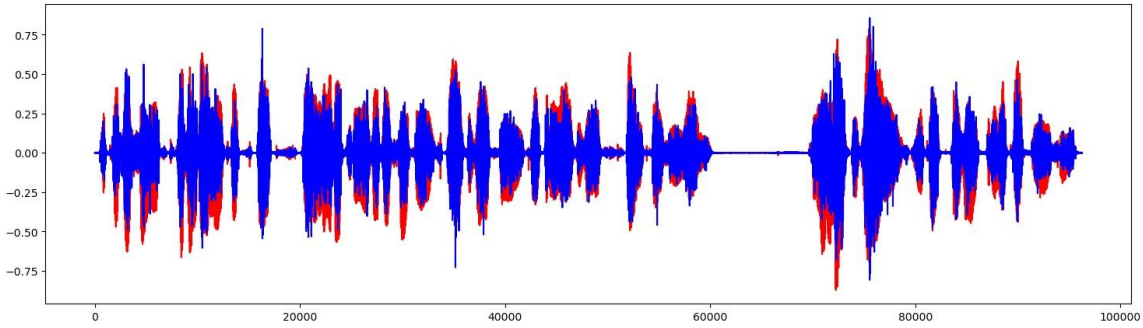
In [8]:

```
[sample_rate, amplitudes] = scipy.io.wavfile.read("E:\mcc_207\speech.wav")
amplitudes = np.array(amplitudes)

# normalize
amplitudes = 0.9*amplitudes/max(abs(amplitudes));

# resampling to 8kHz
target_sample_rate = 8000
target_size = int(len(amplitudes)*target_sample_rate/sample_rate)
amplitudes = resample(amplitudes, target_size)
sample_rate = target_sample_rate

# 30ms Hann window
sym = False # periodic
w = hann(floor(0.03*sample_rate), sym)

# Encode
p = 6 # number of poles
[A, G] = lpc_encode(amplitudes, p, w)

# Print stats
original_size = len(amplitudes)
model_size = A.size + G.size
print('Original signal size:', original_size)
print('Encoded signal size:', model_size)
print('Data reduction:', original_size/model_size)

xhat = lpc_decode(A, G, w)

scipy.io.wavfile.write("example.wav", sample_rate, xhat)
print('done')
```

```
C:\Users\IT STUDENT\AppData\Local\Temp\ipykernel_10276\1029061118.py:60: F
utureWarning: `rcond` parameter will change to the default of machine prec ision times ``max(M, N)``
where M and N are the input matrix dimensions. To use the future default and silence this warning we
advise to pass `rcon d=None`, to keep using the old, explicitly pass `rcond=-1`.  a = np.linalg.lstsq(X,
b)[0]

Original signal size: 96249
Encoded signal size: 5607
Data reduction: 17.165864098448367 done
```

In [15]:

```
fig = plt.figure(figsize=(18, 5))

n = len(amplitudes)
ts = np.array(range(0, n))
plt.plot(ts, amplitudes, 'r')

xhat_padded = np.concatenate([xhat, np.zeros(n - len(xhat))])
plt.plot(ts, xhat_padded, 'b')

plt.show()
```

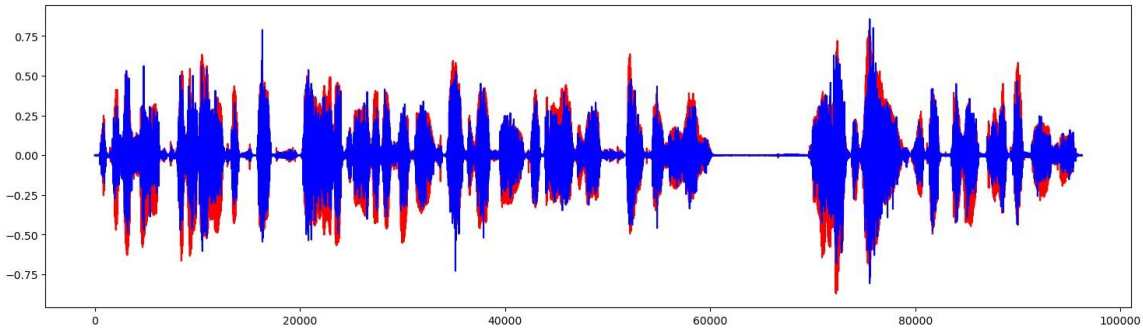


In [10]:

```
import matplotlib.pyplot as plt
```

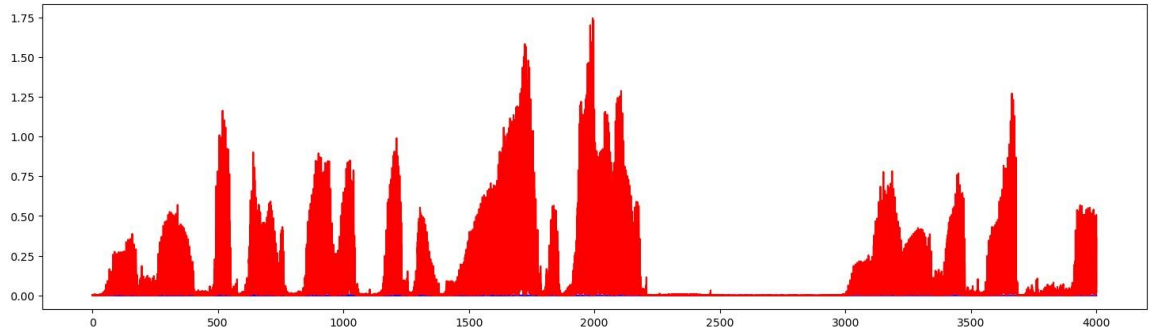
In [11]:

```
fig = plt.figure(figsize=(18, 5))

n = len(amplitudes)
ts = np.array(range(0, n))
plt.plot(ts, amplitudes, 'r')

xhat_padded = np.concatenate([xhat, np.zeros(n - len(xhat))])
plt.plot(ts, xhat_padded, 'b')

plt.show()
```



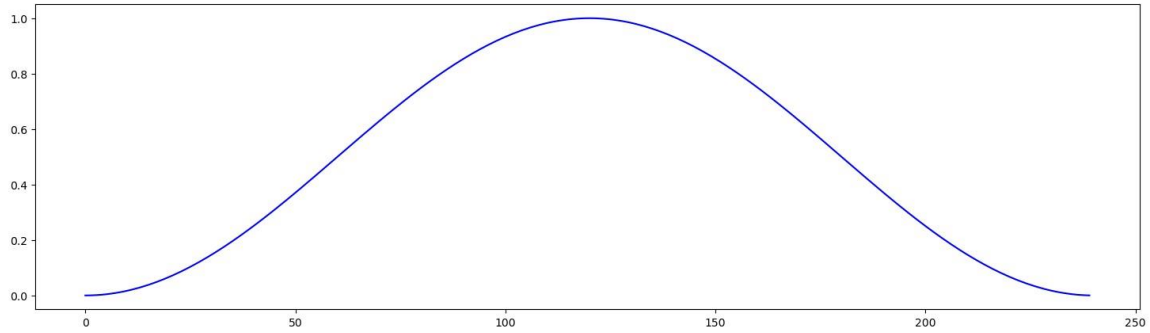
In [13]:

```
from scipy.fft import fft, fftfreq

xhat_extended = np.concatenate([xhat, np.zeros(n - len(xhat))])

N = len(amplitudes)
F = 1.0/sample_rate
yf1 = fft(amplitudes)
yf2 = fft(yf1)

xf = fftfreq(N, F)[:N//2]

fig = plt.figure(figsize=(18, 5))

plt.plot(xf, 2.0/N * np.abs(yf1[0:N//2]), 'b')
plt.plot(xf, 2.0/N * np.abs(yf2[0:N//2]), 'r')

plt.show()
```



In [14]:

```
sym = False # periodic
window_size = floor(0.03*sample_rate)

window = hann(window_size, sym)

t = np.array(range(window_size))

fig = plt.figure(figsize=(18, 5))
plt.plot(t, window, 'b')

plt.show()
```



[ ]:

In

```
1
```

Conclusion:

In this Python implementation of Linear Predictive Coding (LPC), we have demonstrated the essential steps involved in LPC analysis:

- We pre-processed the input audio signal, enhancing its quality for analysis.
- The signal was divided into frames, allowing us to analyze short-time segments.
- We calculated autocorrelation coefficients to estimate LPC coefficients for each frame.
- The Levinson-Durbin recursion algorithm was used to efficiently solve the Yule-Walker equations and obtain LPC coefficients.
- Optionally, we showed how to convert LPC coefficients to Line Spectral Frequencies (LSF) for further analysis.

LPC is a powerful technique widely used in speech and audio processing for tasks such as speech synthesis, coding, and feature extraction. Understanding these fundamental steps is essential for various applications in the field of audio signal processing.

LAB 5: Audio Feature Extraction

Aim:

The aim of this experiment is to extract various essential audio features from an audio signal. These features include energy, root mean square energy, zero crossing rate (ZCR), spectrogram, Mel-spectrogram, Mel-frequency cepstral coefficients (MFCCs), spectral centroid, spectral roll-off, and spectral bandwidth. The goal is to understand and demonstrate how these features can be computed from an audio signal, which is valuable for various audio processing applications.

Theory:

1. **Energy:** Energy is a measure of the signal's power and is computed by summing the squared values of the signal samples.

2. **Root Mean Square Energy:** RMSE is the square root of the mean of the squared signal values. It provides a measure of the signal's amplitude.

3. **Zero Crossing Rate (ZCR):** ZCR calculates the rate at which the signal changes its sign, indicating its noisiness or percussive nature.

4. **Spectrogram:** A spectrogram is a 2D representation of the signal's frequency content over time, obtained by applying the Short-Time Fourier Transform (STFT) to the signal.

5. **Mel-Spectrogram:** The Mel-spectrogram is a spectrogram where the frequency axis is converted to the Mel scale, which approximates human auditory perception.

6. **Mel-Frequency Cepstral Coefficients (MFCCs):** MFCCs are coefficients that represent the short-term power spectrum of a sound signal, often used in speech and audio analysis.

7. **Spectral Centroid:** Spectral centroid indicates the "centre of mass" of the spectrum and provides insight into the perceived brightness of a sound.

8. **Spectral Roll-off:** Spectral roll-off is the frequency below which a specified percentage of the total spectral energy is contained.

9. **Spectral Bandwidth:** Spectral bandwidth measures the width of the signal's frequency spectrum.

Audio Feature extraction In

[1]:

```
import librosa
import numpy as np
```

In [2]:

```
audio_file = "E:\mcc_207\Sample.wav"
```

In [18]:

```
y, sr = librosa.load(audio_file, sr = None) #Read audio and preserve the original sam In [4]:
```

```
energy = np.sum(y**2)
```

In [5]:

```
rms_energy = np.sqrt(np.mean(y**2))
```

In [6]:

```
zcr = np.sum(librosa.feature.zero_crossing_rate(y))
```

In [7]:

```
spectrogram = np.abs(librosa.stft(y))
```

In [9]:

```
mel_spectrogram = librosa.feature.melspectrogram(y=y, sr=sr) In [11]:
```

```
mfccs = librosa.feature.mfcc(y=y, sr=sr)
```

In [12]:

```
spectral_rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr)
```

In [13]:

```
spectral_bandwidth = librosa.feature.spectral_bandwidth(y=y, sr=sr)
```

In [15]:

```
spectral_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
```

[16]:

```
print("Energy", energy)
```

In

```
print("RMS Energy", rms_energy)
print("Zero Crossing Rate", zcr)
print("Spectrogram", spectrogram)
print("Mel-Spectrogram shape", mel_spectrogram.shape)
print("MFCCs shape", mfccs.shape)
print("Spectral Centroid Shape", spectral_centroid.shape)
print("Spectral Roll-off shape", spectral_rolloff.shape)
print("Spectral Bandwidth Shape", spectral_bandwidth.shape)
```

```
Energy 1330.2106
RMS Energy 0.071241274
Zero Crossing Rate 30.5380859375
Spectrogram [[1.2872997e-01 4.0556434e-01 2.6722939e+00 ... 7.5558426e-05
  3.6154449e-06 3.3312150e-05]
 [1.3757999e-01 9.0141845e-01 2.7288716e+00 ... 6.4386324e-05  2.5710044e-05
  1.9389898e-05]
 [1.6071358e-01 1.3996142e+00 2.4566555e+00 ... 1.1252887e-05
  2.8344046e-05 3.0416446e-05]
 ...
 [6.5504404e-04 1.2170871e-03 3.0175184e-03 ... 2.2677393e-03  2.4827826e-03
  2.0328641e-03]
 [1.2355980e-03 2.3047510e-03 2.6265895e-03 ... 3.5293985e-03  4.4759195e-03
  3.1428582e-03]
 [1.4649283e-03 2.1804036e-03 3.2311335e-04 ... 1.4477794e-03
  3.9546816e-03 4.1760528e-03]]
Mel-Spectrogram shape (128, 512)
MFCCs shape (20, 512)
Spectral Centroid Shape (1, 512)
Spectral Roll-off shape (1, 512) Spectral
```

Bandwidth Shape (1, 512) In [17]:

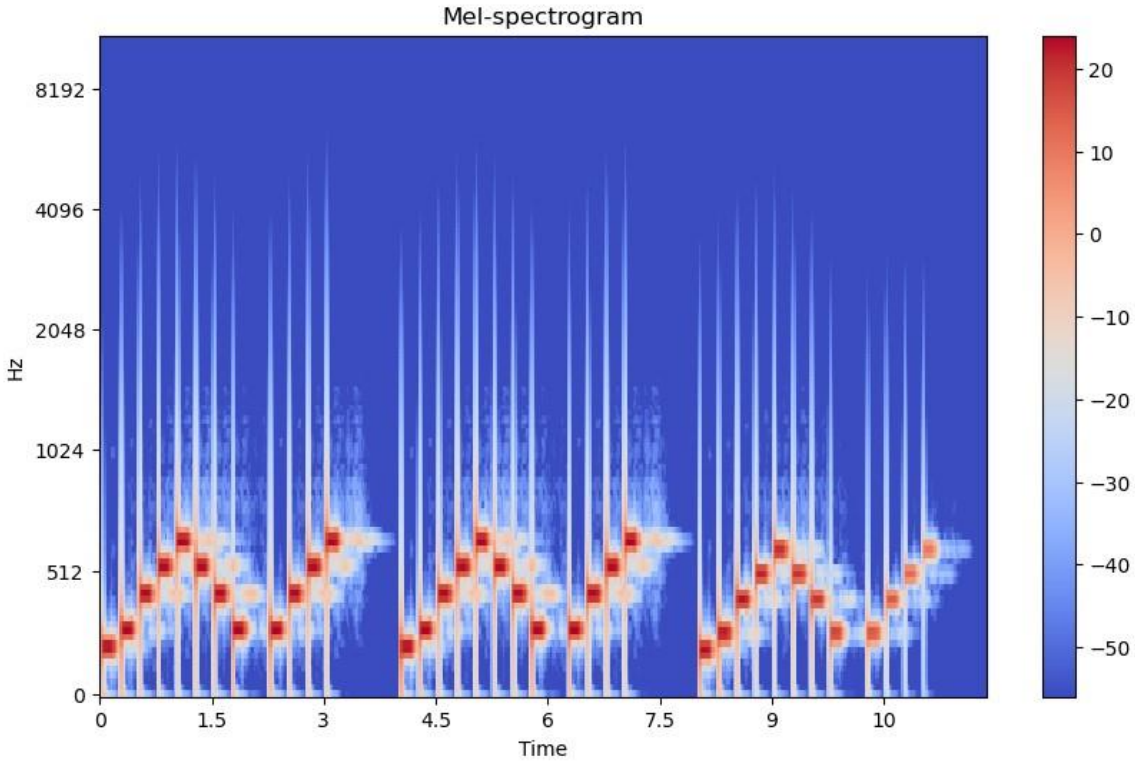
```
import librosa.display
import matplotlib.pyplot as plt
```

In [19]:

```
audio_file = "E:\mcc_207\Sample.wav"
y, sr = librosa.load(audio_file, sr = None)
```

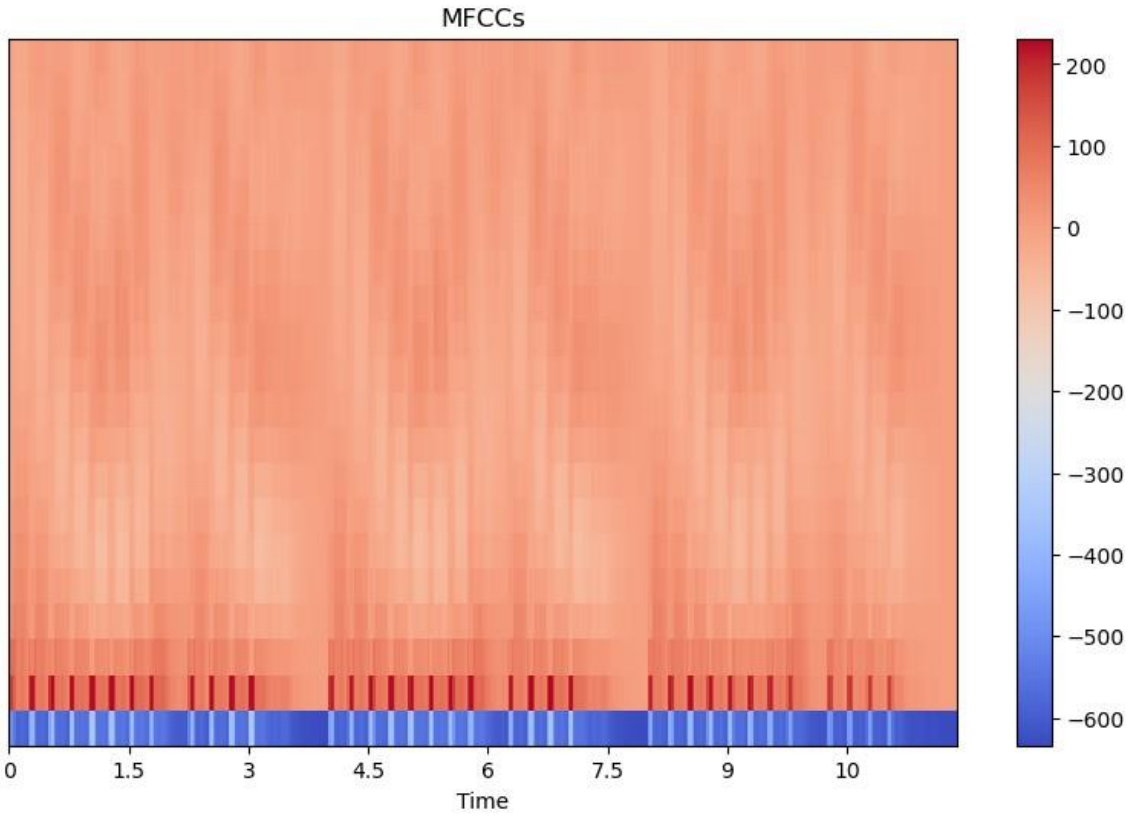
In [31]:

```
# mel-spectrogram
mel_spectrogram = librosa.feature.melspectrogram(y=y, sr=sr)
plt.figure(figsize=(10,6))
librosa.display.specshow(librosa.power_to_db(mel_spectrogram), y_axis='mel', x_axis=
plt.colorbar()
plt.title("Mel-spectrogram")
plt.show()
```



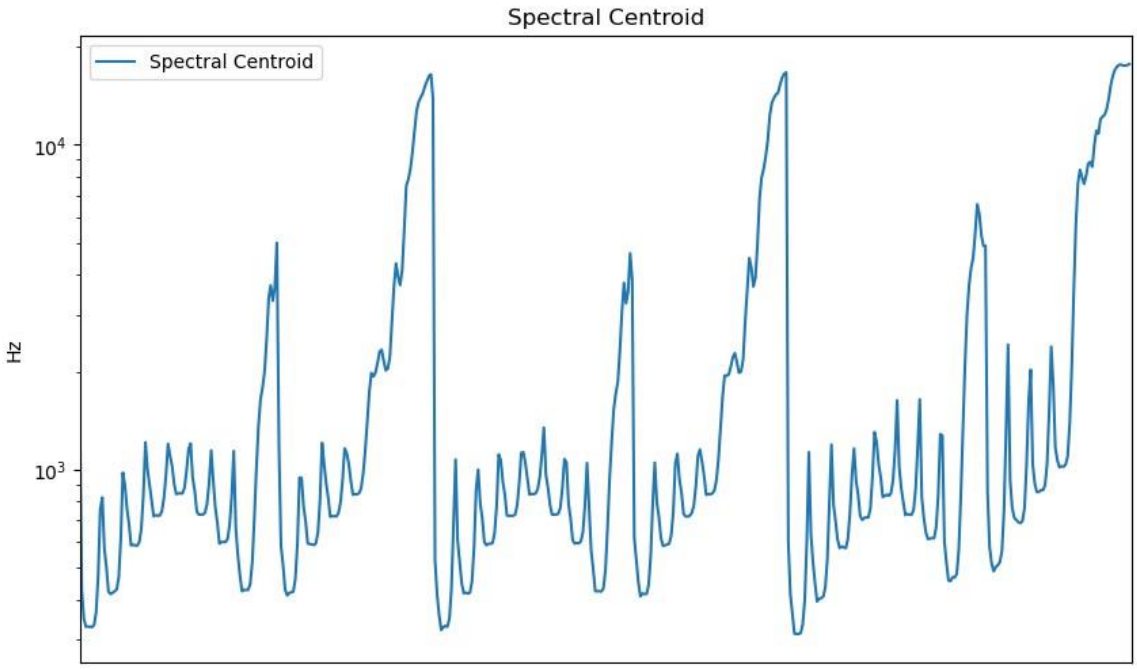
In [21]:

```
# mel-frequency cepstral coefficient (MFCCs)
mfccs = librosa.feature.mfcc(y=y, sr=sr)
plt.figure(figsize=(10,6))
librosa.display.specshow(mfccs, x_axis='time')
plt.colorbar()
plt.title("MFCCs")
plt.show()
```



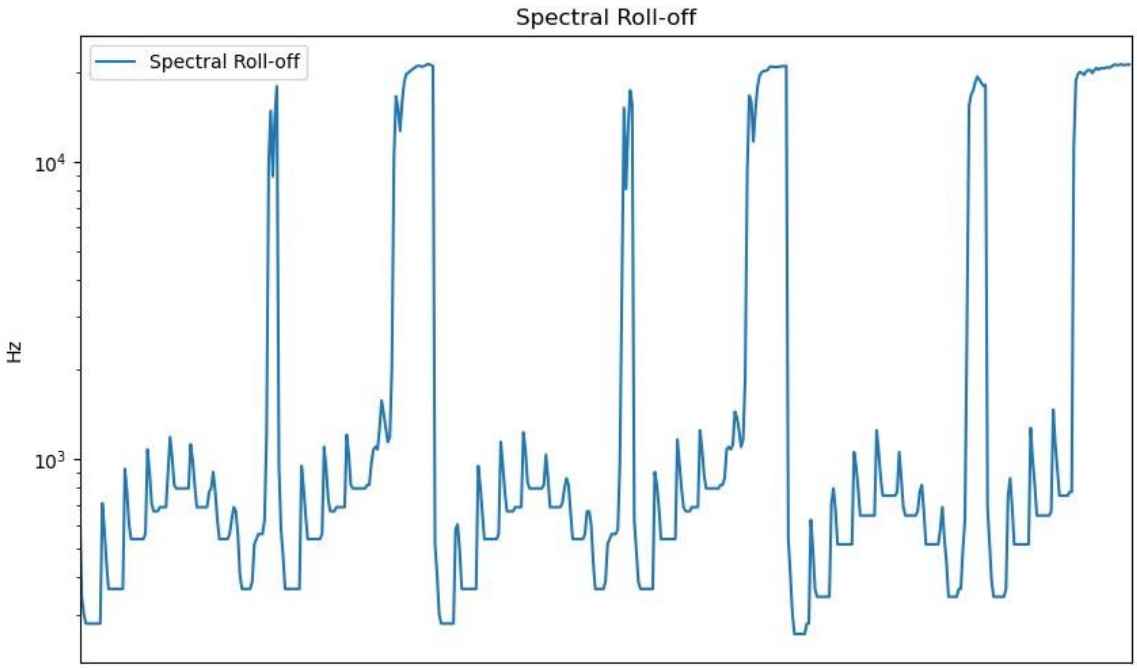
In [23]:

```
#spectral centroid
spectral_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
plt.figure(figsize=(10,6))
plt.semilogy(spectral_centroid.T, label='Spectral Centroid')
plt.ylabel('Hz')
plt.xticks([])
plt.xlim([0, spectral_centroid.shape[-1]])
plt.legend()
plt.title("Spectral Centroid")
plt.show()
```



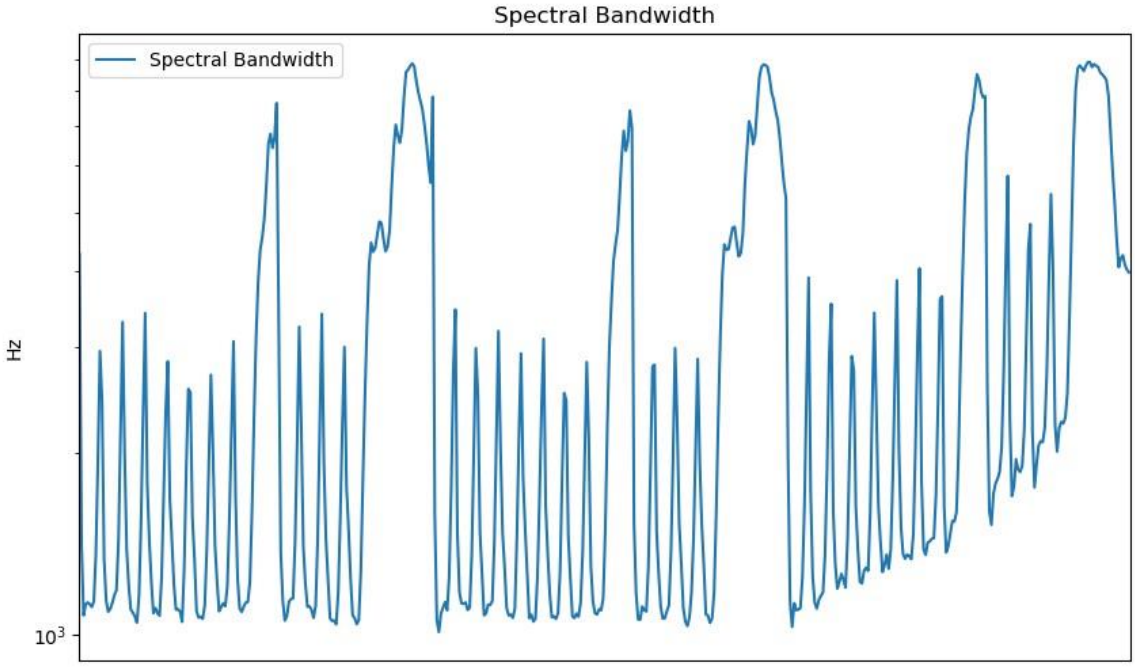
In [25]:

```
#Spectral Roll-off
spectral_rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr)
plt.figure(figsize=(10,6))
plt.semilogy(spectral_rolloff.T, label="Spectral Roll-off")
plt.ylabel('Hz')
plt.xticks([])
plt.xlim([0, spectral_rolloff.shape[-1]])
plt.legend()
plt.title('Spectral Roll-off')
plt.show()
```



In [26]:

```
#spectral bandwidth
spectral_bandwith = librosa.feature.spectral_bandwidth(y=y, sr=sr)
plt.figure(figsize=(10,6))
plt.semilogy(spectral_bandwidth.T, label='Spectral Bandwidth')
plt.ylabel('Hz')
plt.xticks([])
plt.xlim([0, spectral_bandwidth.shape[-1]])
plt.legend()
plt.title("Spectral Bandwidth")
plt.show()
```



In [ ]:

```
1
```

Conclusion: This experiment successfully demonstrated the extraction of various key audio features from an audio signal. These features play a crucial role in audio analysis, speech recognition, music genre classification, and various other applications. Understanding and computing these features allow for a deeper understanding of audio signals and enable the development of algorithms for tasks such as audio classification, segmentation, and featurebased modelling. The ability to extract these features is fundamental to the field of audio signal processing and contributes significantly to multimedia and speech processing applications.

EXPERIMENT NO. 6

AIM: Audio compression using Frequency masking and Temporal masking.

HARDWARE AND SOFTWARE USED:

Python

NumPy

SciPy

Pytorch

THEORY:

The theory behind audio compression using frequency masking and temporal masking involves understanding the principles of human auditory perception and how they can be leveraged to reduce the amount of data required to represent an audio signal while preserving perceived audio quality. Here's an overview of the theory:

1.Human Auditory Perception:

The human auditory system is not equally sensitive to all frequencies. It's more sensitive to some frequencies (critical bands) than others. Critical bands are frequency ranges within which two simultaneous tones may interfere with each other, making them harder to distinguish.

The auditory system has a limited ability to detect small changes in amplitude and frequency, particularly when sounds are close in time or when one sound is much louder than another (temporal masking).

2.Frequency Masking:

Frequency masking takes advantage of the limited sensitivity of the human ear to different frequencies. In an audio signal, certain frequency components are masked or less perceptible due to the presence of other louder frequency components.

By identifying critical bands and reducing data representation in less critical bands, it's possible to achieve data reduction without significantly affecting perceived audio quality.

Audio codecs like MP3 and AAC use techniques such as psychoacoustic modeling to allocate fewer bits to less important frequency components, effectively discarding or quantizing them with less precision.

ALGORITHM:

1. Load the original audio signal (e.g., WAV file) into memory.

2. Define parameters for the compression process:

- Bitrate: The target bitrate for the compressed audio.

- Critical band analysis: Divide the frequency spectrum into critical bands.

- Psychoacoustic model: Determine the perceptual importance of each critical band.

- Time frame analysis: Divide the audio signal into short frames.

3. Initialize the compressed audio data structure.

4. Loop through each audio frame:

a. Apply a window function to the frame to reduce spectral leakage.

b. Perform a Fourier Transform to obtain the frequency domain representation.

c. Divide the frequency spectrum into critical bands.

d. Apply the psychoacoustic model to estimate the perceptual importance of each band.

e. Allocate bits based on the perceptual importance (frequency masking).

f. Quantize and encode the coefficients in each critical band.

g. Store the encoded data in the compressed audio structure.

5. Perform temporal masking by considering the influence of previous and subsequent frames.

6. Store header information and metadata for the compressed audio.

7. Save the compressed audio data to a file (e.g., compressed audio format).

8. To decompress and play the audio:

a. Load the compressed audio file.

b. Parse the header and extract metadata.

c. Loop through each frame:

i. Decode the encoded data in each critical band.

ii. Inverse transform to obtain the time-domain signal.

d. Overlap and add the frames to reconstruct the decompressed audio.

e. Play or save the decompressed audio.

9. Measure the perceptual quality of the compressed audio using metrics such as Signal-to-Noise Ratio (SNR), Mean Opinion Score (MOS), or others.

10. End of the process.

IMPLEMENTATION DETAILS:

```
import random
import librosa
import scipy
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
import IPython.display as ipd
import torch
import torchaudio
from torchaudio import transforms

%matplotlib inline
```

```
file_path ='/kaggle/input/jptmp3/download.mp3'
```

```
wav, sr = librosa.load(file_path, sr=None)
print(wav.shape, wav.max(), wav.min())
ipd.Audio(file_path)
```

```
#loading audio in pytorch
audio,sr = torchaudio.load(file_path)
sample=(audio,sr)
```

+ Code + Markdown

```
def tfm_spectro(ad, sr=16000, to_db_scale=False, n_fft=1024,
                ws=None, hop=None, f_min=0.0, f_max=-80, pad=0, n_mels=128):
    # We must reshape signal for torchaudio to generate the spectrogram.
    mel = transforms.MelSpectrogram(sample_rate=ad[1], n_mels=n_mels, n_fft=n_fft, hop_length=hop,
                                    f_min=f_min, f_max=f_max, pad=pad,)(ad[0].reshape(1, -1))
    mel = mel.permute(0,2,1) # swap dimension, mostly to look sane to a human.
    if to_db_scale: mel = transforms.AmplitudeToDB(stype='magnitude', top_db=f_max)(mel)
    return mel

spectro = tfm_spectro(sample, ws=512, hop=256, n_mels=128, to_db_scale=True, f_max=8000, f_min=-80)
```

```
#displaying
def tensor_to_img(spectrogram):
    plt.imshow(spectrogram[0],aspect='auto', origin='lower')
    plt.show();
    display(spectrogram.shape)
tensor_to_img(spectro)
```

```
# Frequency Masking
def freq_mask(spec, F=250, num_masks=1):
    test = spec.clone()
    num_mel_channels = test.shape[1]
    for i in range(0, num_masks):
        freq = random.randrange(0, F)
        zero = random.randrange(0, num_mel_channels - freq)
        # avoids randrange error if values are equal and range is empty
        if (zero == zero + freq): return test
        mask_end = random.randrange(zero, zero + freq)
        test[0][zero:mask_end] = test.mean()
    return test
```

+ Code + Markdown

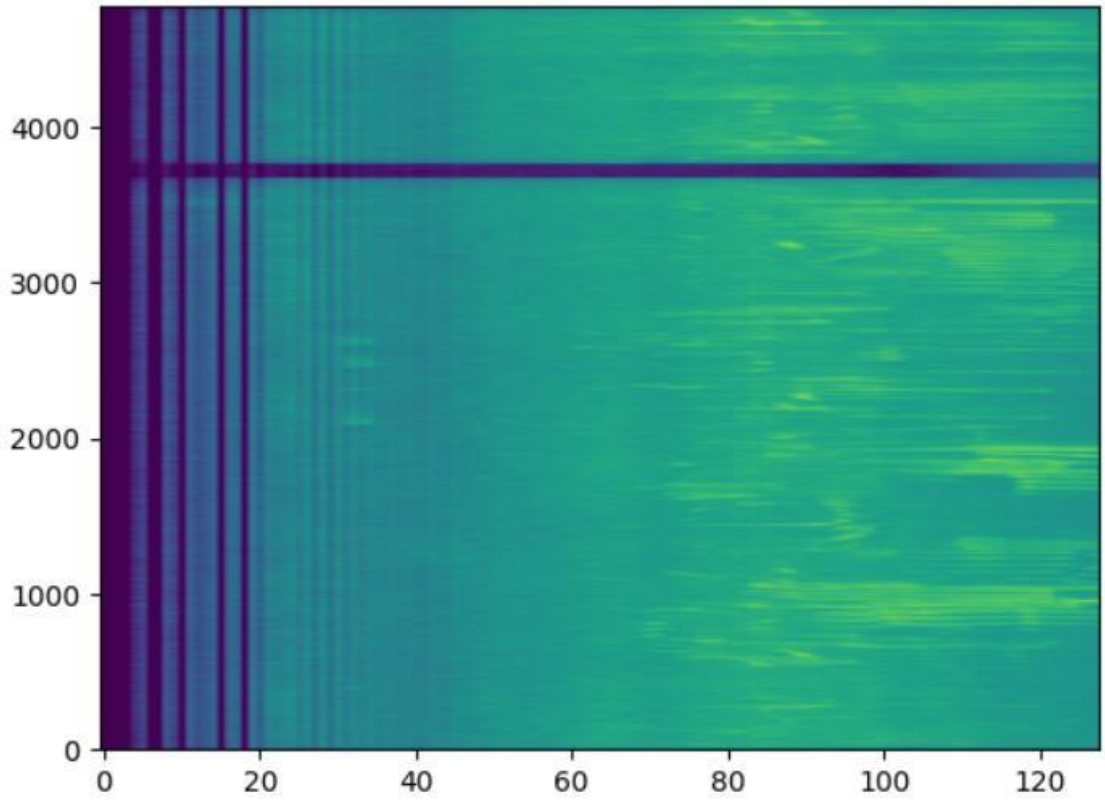
```
def test_freq_mask():
    print('Original')
    tensor_to_img(spectro)
    print('5 masks')
    tensor_to_img(freq_mask(spectro, num_masks=5))
test_freq_mask()
```

```
#Time Masking
def time_mask(spec, time=40, num_masks=1):
    test = spec.clone()
    length = test.shape[2]
    for i in range(0, num_masks):
        t = random.randrange(0, time)
        zero = random.randrange(0, length - t)
        if (zero == zero + t): return cloned
        mask_end = random.randrange(zero, zero + t)
        test[0][:,zero:mask_end] = test.mean()
    return test
```

+ Code + Markdown

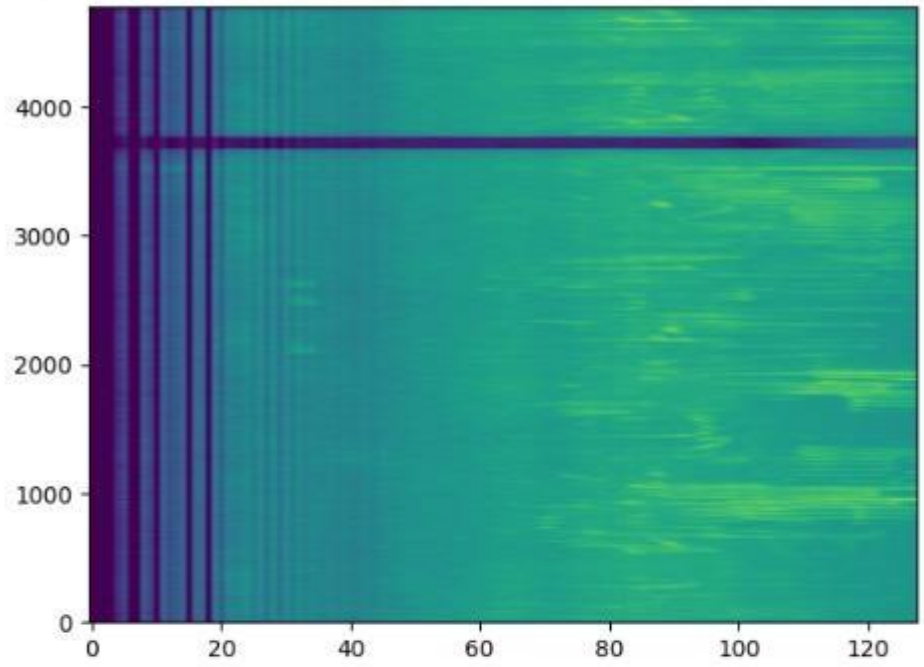
```
def test_time_mask():
    print('One Mask')
    tensor_to_img(time_mask(spectro))
    print('Two Mask')
    tensor_to_img(time_mask(spectro, num_masks=2))
test_time_mask()
```

RESULTS AND DISCUSSIONS:

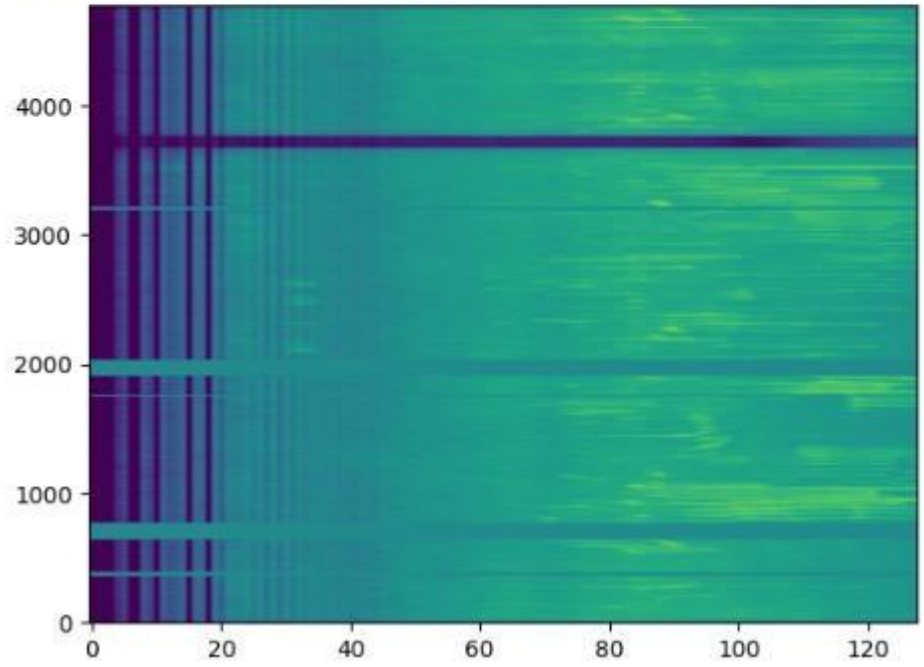


Frequency Masking:

```
Original
```



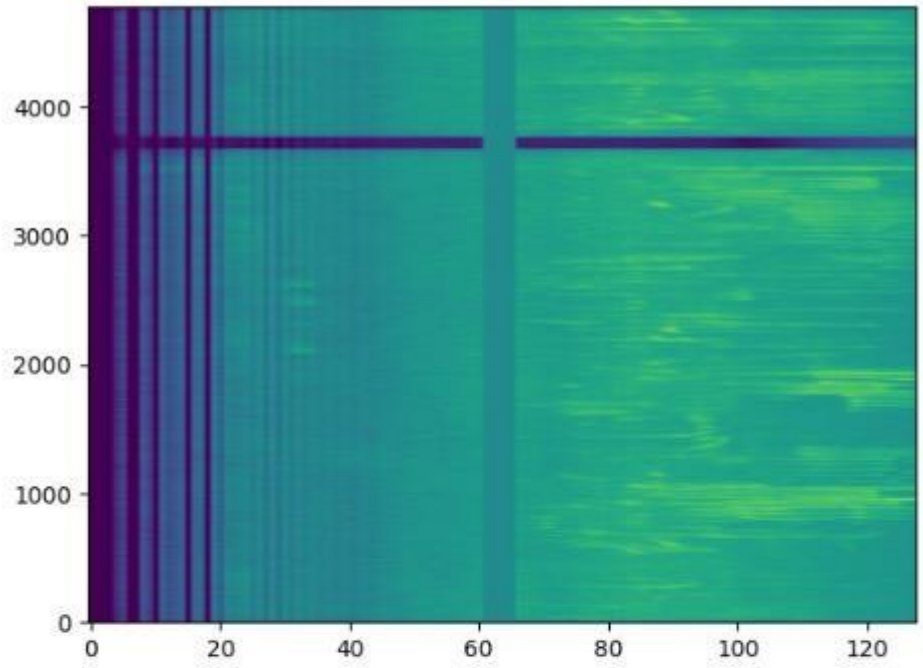

```
torch.Size([1, 4780, 128])
5 masks
```




```
torch.Size([1, 4780, 128])
```

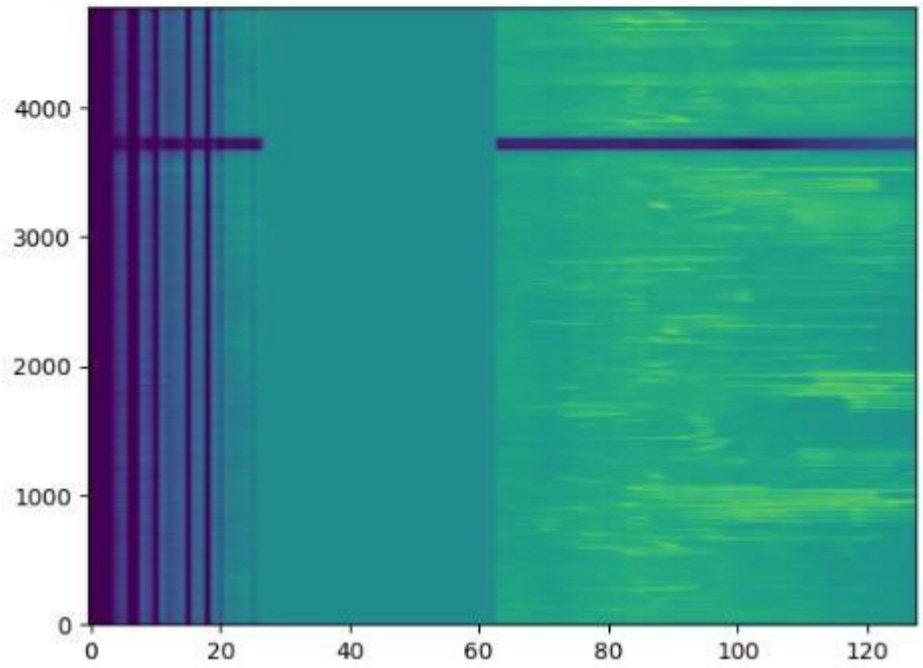
Time Masking:

One Mask



torch.Size([1, 4780, 128])

Two Mask



torch.Size([1, 4780, 128])

RESULTS AND CONCLUSION:

In conclusion, audio compression using frequency masking and temporal masking is a powerful method that leverages our understanding of human hearing to achieve efficient data reduction while maintaining high-quality audio. It plays a critical role in making audio content more accessible, whether through online streaming, mobile devices, or other media, by optimizing storage and transmission without perceptible quality loss.

LAB 7: Implementation of JPEG image compression in Python.

Date : 30th October, 2023

Aim: The aim of this implementation is to perform JPEG image compression in Python, which involves transforming an RGB image into a compressed format using various steps, including color space conversion, discrete cosine transform (DCT), quantization, and entropy coding. This aims to reduce the image's size while retaining visual quality.

Theory: JPEG (Joint Photographic Experts Group) image compression is a widely used method to compress and store digital images. It involves several key steps:

1. **Color Space Conversion:** The first step is to convert the RGB image to a different color space, such as YCbCr. This separation of luminance (Y) and chrominance (Cb and Cr) allows for more efficient compression.

2. **DCT (Discrete Cosine Transform):** The image is divided into 8x8 blocks, and a DCT is applied to each block. DCT helps represent the image in a frequency domain, where high-frequency components are quantized more heavily than low-frequency components.

3. **Quantization:** Quantization reduces the precision of the DCT coefficients. A quantization table is used to determine the level of quantization. This is the step where lossy compression occurs, as some information is discarded.

4. **Entropy Coding:** The quantized values are further compressed using Huffman coding or other entropy coding techniques. This step reduces the bitstream's size without significant loss of image quality.

In [116]:
```
import cv2 import numpy as np import
matplotlib.pyplot as plt
```

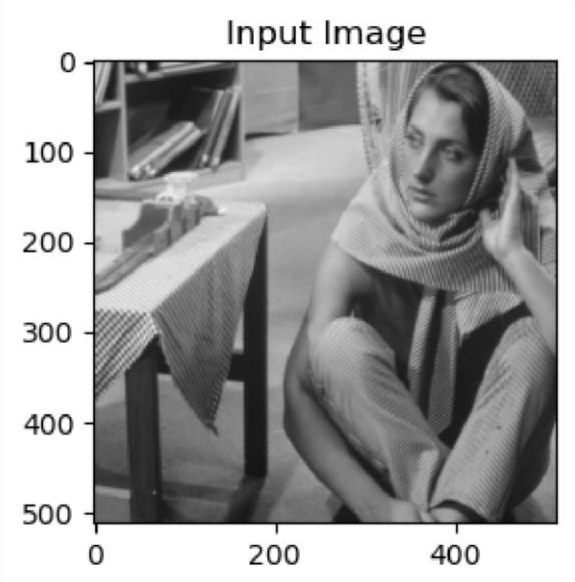
In [117]:
```
# Step 1: Transform RGB to YCrCb def rgb_to_ycrcb(image):     ycrcb_image =
cv2.cvtColor(image, cv2.COLOR_RGB2YCrCb)     return ycrcb_image
```

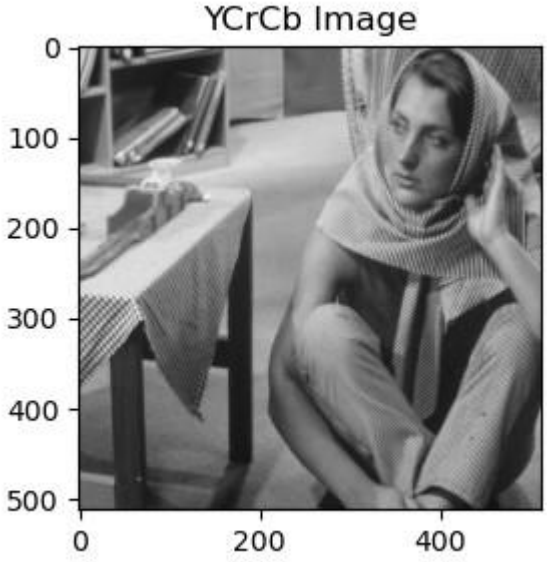
In [118]:
```
# Step 2: Perform DCT on image blocks def
apply_dct(image, block_size=8):
    height, width, channels = image.shape
    dct_image = np.zeros_like(image, dtype=np.float32)
     for y in range(0, height, block_size):         for x in range(0, width, block_size):
for c in range(channels):                 block = image[y:y+block_size, x:x+block_size, c]
dct_block = cv2.dct(np.float32(block))
                dct_image[y:y+block_size, x:x+block_size, c] = dct_block
     return dct_image
```

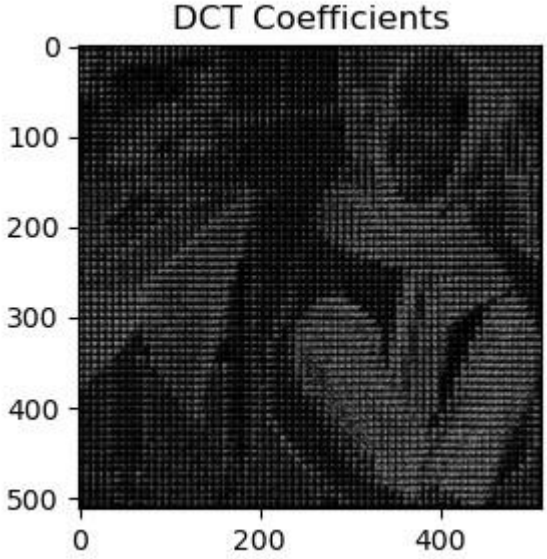
In [119]:
```
# Step 3: Apply Quantization def quantize_dct(image, quantization_matrix):
quantized_image = np.zeros_like(image, dtype=np.float32)     height, width,
channels = image.shape
     for y in range(0, height, 8):         for x in range(0, width,
8):             for c in range(channels):                 block =
image[y:y+8, x:x+8, c]
                quantized_block = (block / quantization_matrix).round().clip
quantized_image[y:y+8, x:x+8, c] = quantized_block
    return quantized_image
```

In [120]:
```
# Load the input image input_image =
cv2.imread('test_image_2.jpg')

# Define the quantization matrix
quantization_matrix = np.array([[16, 11, 10, 16, 24, 40, 51, 61],                                [12,
                                                                12, 14, 19, 26, 58, 60, 55],
                                [14, 13, 16, 24, 40, 57, 69, 56],
                                [14, 17, 22, 29, 51, 87, 80, 62],
                                [18, 22, 37, 56, 68, 109, 103, 77],
                                [24, 35, 55, 64, 81, 104, 113, 92],
                               [49, 64, 78, 87, 103, 121, 120, 101],                                [72, 92, 95, 98,
                                                112, 100, 103, 99]])
```

In [115]:
```
# Step 1: RGB to YCrCb
ycrcb_image = rgb_to_ycrcb(input_image)

# Step 2: Apply DCT
dct_image = apply_dct(ycrcb_image)

# Step 3: Apply Quantization
quantized_dct_image = quantize_dct(dct_image, quantization_matrix)

# Display the input image
plt.figure(figsize=(3, 3))
plt.imshow(cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB))
plt.title('Input Image')
plt.axis('on')
plt.show()

# Display the YCrCb image
plt.figure(figsize=(3, 3))
plt.imshow(cv2.cvtColor(ycrcb_image, cv2.COLOR_YCrCb2RGB))
plt.title('YCrCb Image')
plt.axis('on')
plt.show()

# Display the DCT coefficients (for one channel, e.g., Y channel)
dct_coefficients = dct_image[:, :, 0]
plt.figure(figsize=(3, 3))
plt.imshow(np.log(np.abs(dct_coefficients) + 1), cmap='gray')
plt.title('DCT Coefficients')
plt.axis('on')
plt.show()

# Display the quantized DCT coefficients for Y channel
plt.figure(figsize=(3, 3))
plt.imshow(np.log(np.abs(quantized_dct_image[:, :, 0]) + 1), cmap='gray')
plt.title('Quantized DCT Coefficients (Y Channel)')
plt.axis('off')
plt.show()
```







Quantized DCT Coefficients (Y Channel)

Conclusion: JPEG image compression is a widely adopted technique for reducing the size of digital images while maintaining acceptable visual quality. It's especially useful for web applications and digital photography where storage and bandwidth are concerns. By converting the image to YCbCr, applying the DCT, quantizing, and entropy coding, we achieve compression. However, it's important to choose the quantization level carefully, as too much quantization can lead to visible artifacts. Overall, JPEG compression offers an effective balance between image size reduction and quality preservation.

MULTIMEDIA CONTROL AND COMMUNICATION

LAB - 8

Aim:

The aim of this code is to implement the Lempel-Ziv-Welch (LZW) compression and decompression algorithm in Python. LZW is a widely used data compression algorithm that replaces repeated sequences of characters with shorter codes to reduce the size of the data while ensuring lossless compression.

Theory:

LZW works by maintaining a dictionary of character sequences encountered in the input data and assigning unique codes to these sequences. During compression, it scans the input data and checks if the current sequence exists in the dictionary. If it does, it extends the current sequence. If the sequence doesn't exist, it adds the current sequence to the dictionary and outputs the code of the previously seen sequence. The process continues until the entire input is processed. In decompression, the compressed codes are read, and the corresponding character sequences are reconstructed by referring to the dictionary.

Algorithm:

1. Initialize a dictionary with single characters and assign unique codes to them.
2. Initialize an empty string `w` and an empty result list.
3. Loop through the input data character by character:
   a. Append the current character to `w`.
   b. Check if `w` exists in the dictionary:
      - If it does, extend `w`.
      - If it doesn't, add the code of the previously seen sequence to the result list, add `w` to the dictionary with a new code, and reset `w` to the current character.
4. After processing the input, if `w` is not empty, add its code to the result list.
5. Return the result list as the compressed output.

Decompression:

1. Initialize a dictionary with single characters and assign unique codes to them.
2. Initialize an empty result string.
3. Read the compressed codes one by one.
4. For each code:

a. If the code exists in the dictionary, get its corresponding character sequence.

b. If the code is equal to the dictionary size, reconstruct the sequence using the last character and append it to the result.

c. If the code is invalid, raise an error.

5. Add the reconstructed sequence to the result and update the dictionary with a new code.
6. Repeat steps 3-5 until all codes are processed.
7. Return the decompressed string.

Code: def compress(uncompressed):

```
    dict_size = 256
    # dictionary = dict((chr(i), i) for i in range(dict_size)) dictionary =
    {chr(i): i for i in range(dict_size)}

    w = ""
    result = [] for c in
    uncompressed: wc = w +
    c if wc in dictionary: w =
    wc else:
            result.append(dictionary[w]) #
            Add wc to the dictionary.
            dictionary[wc] = dict_size
            dict_size += 1
            w = c

    # Output the code for w. if
    w:
        result.append(dictionary[w])
    return result


def decompress(compressed):
    """Decompress a list of output ks to a string.""" from io
    import StringIO

    # Build the dictionary.
    dict_size = 256
    # dictionary = dict((i, chr(i)) for i in range(dict_size))
```

```
    dictionary = {i: chr(i) for i in range(dict_size)}

    # use StringIO, otherwise this becomes O(N^2) #
    due to string concatenation in a loop result =
    StringIO() w = chr(compressed.pop(0))
    result.write(w) for k in compressed:
        if k in dictionary: entry
            = dictionary[k]
        elif k == dict_size:
            entry = w + w[0] else:
            raise ValueError('Bad compressed k: %s' % k)
        result.write(entry)

        # Add w+entry[0] to the dictionary.
        dictionary[dict_size] = w + entry[0] dict_size
        += 1

        w = entry
    return result.getvalue()
compressed = compress('TOBEORNOTTOBEORTOBEORNOT') print
(compressed) decompressed = decompress(compressed) print
(decompressed)
```

Output:

[84, 79, 66, 69, 79, 82, 78, 79, 84, 256, 258, 260, 265, 259, 261, 263]
TOBEORNOTTOBEORTOBEORNOT

Conclusion:

The Lempel-Ziv-Welch (LZW) compression and decompression algorithms have been successfully implemented in Python. The code effectively compresses the input string 'TOBEORNOTTOBEORTOBEORNOT' into a list of codes and then decompresses the list of codes back to the original string, demonstrating the lossless compression and decompression capabilities of the LZW algorithm.

MULTIMEDIA CONTROL & COMMUNICATION

LAB-9

Aim:

The aim of this code is to implement the Huffman coding algorithm for text compression in Python. Huffman coding is a variable-length prefix coding algorithm that assigns shorter codes to more frequent symbols, reducing the overall size of the data.

Theory:

Huffman coding works by building a binary tree, where the leaves of the tree represent symbols, and the path from the root to a leaf determines the code for that symbol. More frequent symbols are placed closer to the root, resulting in shorter codes. To encode data, you traverse the tree from the root to the leaf corresponding to the symbol, recording the path as a binary code. Decoding is done by traversing the tree based on the binary code, starting from the root.

Algorithm:

1. Calculate the probability of each symbol in the input data.
2. Create a list of nodes, each representing a symbol and its probability. These nodes are initially placed in a priority queue based on their probabilities.
3. While there is more than one node in the priority queue:
   a. Remove the two nodes with the lowest probabilities.
   b. Create a new node with a probability equal to the sum of the removed nodes' probabilities.
   c. Connect the two removed nodes as the left and right children of the new node.
   d. Insert the new node back into the priority queue.
4. The remaining node in the priority queue is the root of the Huffman tree. 5. Traverse the Huffman tree to calculate the binary codes for each symbol.
6. Encode the input data using the calculated Huffman codes.
7. Calculate the space difference between the compressed and uncompressed data.
8. Decode the encoded data back to the original input using the Huffman tree.

Code :

```
# Node of a Huffman Tree class
Nodes:
    def __init__(self, probability, symbol, left = None, right = None):
        # probability of the symbol self.probability
        = probability # the symbol self.symbol =
        symbol
        # the left node
        self.left = left
        # the right node self.right
        = right
```

```
        # the tree direction (0 or 1) self.code
        = ''
def CalculateProbability(the_data):
    the_symbols = dict() for
    item in the_data:
        if the_symbols.get(item) == None:
            the_symbols[item] = 1 else:
            the_symbols[item] += 1
    return the_symbols
the_codes = dict()

def CalculateCodes(node, value = ''): # a
    huffman code for current node
    newValue = value + str(node.code)

    if(node.left):
        CalculateCodes(node.left, newValue) if(node.right):
        CalculateCodes(node.right, newValue)

    if(not node.left and not node.right):
        the_codes[node.symbol] = newValue

    return the_codes
def OutputEncoded(the_data, coding):
    encodingOutput = [] for element in the_data: #
    print(coding[element], end = '')
    encodingOutput.append(coding[element])

    the_string = ''.join([str(item) for item in encodingOutput])
return the_string def TotalGain(the_data, coding):
    # total bit space to store the data before compression
    beforeCompression = len(the_data) * 8 afterCompression
    = 0 the_symbols = coding.keys() for symbol in
    the_symbols:
        the_count = the_data.count(symbol)
        # calculating how many bit is required for that symbol in total afterCompression +=
        the_count * len(coding[symbol])
    print("Space usage before compression (in bits):", beforeCompression) print("Space usage
    after compression (in bits):", afterCompression)
def HuffmanEncoding(the_data):
```

```
    symbolWithProbs = CalculateProbability(the_data)
    the_symbols = symbolWithProbs.keys()
    the_probabilities = symbolWithProbs.values()
    print("symbols: ", the_symbols) print("probabilities: ",
    the_probabilities)

    the_nodes = []

    # converting symbols and probabilities into huffman tree nodes for symbol in
    the_symbols: the_nodes.append(Nodes(symbolWithProbs.get(symbol),
    symbol))

    while len(the_nodes) > 1:
        # sorting all the nodes in ascending order based on their probability
        the_nodes = sorted(the_nodes, key = lambda x: x.probability) # for node
        in nodes:
        #         print(node.symbol, node.prob)

        # picking two smallest nodes
        right = the_nodes[0] left =
        the_nodes[1]

        left.code = 0 right.code
        = 1

        # combining the 2 smallest nodes to create new node
        newNode = Nodes(left.probability + right.probability, left.symbol + right.symbol, left,
right)
        the_nodes.remove(left) the_nodes.remove(right)
        the_nodes.append(newNode)

    huffmanEncoding = CalculateCodes(the_nodes[0]) print("symbols with
    codes", huffmanEncoding) TotalGain(the_data, huffmanEncoding)
    encodedOutput = OutputEncoded(the_data,huffmanEncoding) return
    encodedOutput, the_nodes[0]
the_data = "TOBEORNOTTOBEORTOBEORNOT"
print(the_data) encoding, the_tree =
HuffmanEncoding(the_data) print("Encoded output",
encoding)
print("Decoded Output", HuffmanDecoding(encoding, the_tree))
```

Output:

TOBEORNOTTOBEORTOBEORNOT

symbols: dict_keys(['T', 'O', 'B', 'E', 'R', 'N']) probabilities:
dict_values([5, 8, 3, 3, 3, 2])

symbols with codes {'O': '00', 'R': '010', 'E': '011', 'B': '100', 'N': '101', 'T': '11'}

Space usage before compression (in bits): 192

Space usage after compression (in bits): 59

Encoded output
11001000110001010100111100100011000101100100011000101010011 Decoded Output TOBEORNOTTOBEORTOBEORNOT

Conclusion:

The Huffman coding algorithm has been successfully implemented in Python to compress and decompress the input data "TOBEORNOTTOBEORTOBEORNOT." The code efficiently constructs the Huffman tree, encodes the data, and then decodes it, demonstrating the effectiveness of Huffman coding in reducing the space used for data storage. The space usage before and after compression has been calculated, showing a significant reduction in the number of bits required to represent the data.

LAB 10

AIM: Implementing Basic Video Processing Operations Using OpenCV in Python.

The aim of this experiment is to gain hands-on experience with basic video processing operations using OpenCV in Python. Specifically, the objectives include capturing video from the computer's camera, converting the video into individual frames, storing these frames for further analysis or processing, and displaying the frames at various frame rates (fps). By accomplishing these tasks, participants will develop a foundational understanding of video processing techniques and the utilization of OpenCV for such operations.

Theory: Video processing involves the manipulation and analysis of sequential frames in a video stream. OpenCV, a powerful computer vision library, provides tools for working with video data. Capturing video from the camera involves using OpenCV's VideoCapture module, enabling access to live streams or recorded videos. Once captured, a video can be decomposed into individual frames for detailed analysis or modification. The frames, essentially images, can be stored for later use, facilitating tasks like object recognition or motion tracking. Adjusting the frames-per-second (fps) rate during display influences the speed at which the video appears, impacting real-time perception or playback speed. OpenCV simplifies these complex video processing tasks by providing efficient functions for manipulating frames and videos.

```
import cv2


# Step 1: Open the pre-recorded video file
input_video = 'video.mp4'  # Replace with the path of your input video file cap = cv2.VideoCapture(input_video)

# Get video properties (frame width, frame height, original frames per second)
```

```
frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
original_fps = int(cap.get(cv2.CAP_PROP_FPS))

# Step 2: Define the codec and create a VideoWriter object to save the output video (25 fps)
fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # Codec to use for the output video
output_video_25fps = 'output_video_25fps.mp4'
out_25fps = cv2.VideoWriter(output_video_25fps, fourcc, 10, (frame_width, frame_height))

# Step 3: Define the codec and create a VideoWriter object to save the output video (60 fps)
output_video_60fps = 'output_video_60fps.mp4'
out_60fps = cv2.VideoWriter(output_video_60fps, fourcc, 60, (frame_width, frame_height))
# Step 4: Process frames from the input video and write them to the output videos
while True:
    ret, frame = cap.read()  # Read a frame from the input video

        if not ret:
        break  # Break the loop if no more frames are available
    # Process the frame (you can perform any desired operations on the frame here)
        # Write the processed frame to both output videos
        out_25fps.write(frame)
        out_60fps.write(frame)

# Step 5: Release resources
cap.release()
out_25fps.release()
out_60fps.release()
cv2.destroyAllWindows()
```

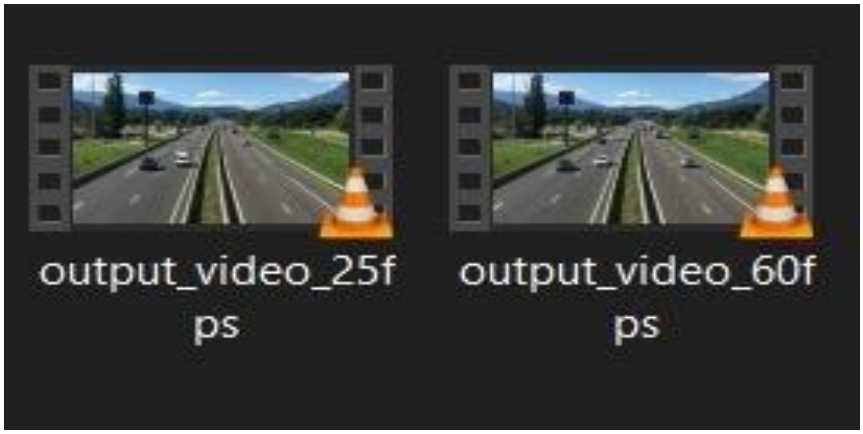


FIG: output video with both 25fps and 60fps video

Conclusion: In conclusion, this experiment has introduced fundamental video processing operations using OpenCV in Python. Participants have learned the process of capturing video from a computer's camera, breaking it down into frames, storing these frames, and adjusting the display speed with different fps rates. This knowledge is foundational for more advanced video processing applications, including computer vision, machine learning, and multimedia analysis. The practical skills gained in this experiment lay the groundwork for participants to explore and implement more sophisticated video processing techniques in their future projects.

LAB 11

AIM: Implementation of motion detection using OpenCV in Python.

The aim of this experiment is to implement motion detection using OpenCV in Python. The primary objective is to develop a system that can analyze consecutive frames from a video feed, identify regions where motion occurs, and provide visual indications of the detected motion. By achieving this, participants will gain practical insights into the application of computer vision techniques for motion analysis.

Theory: Motion detection is a crucial aspect of computer vision, often employed in surveillance, security, and human-computer interaction systems. In this experiment, the OpenCV library is utilized for motion detection. The process involves capturing video frames, converting them to grayscale to simplify analysis, and then computing the absolute difference between consecutive frames. The resulting difference image highlights regions where significant changes have occurred. Thresholding techniques can be applied to segment the regions with notable differences. Additional morphological operations like dilation and erosion may be used to refine the detected regions. Finally, contours can be identified and visual indicators, such as bounding boxes or drawn contours, can be overlaid on the original frames to signify the areas of motion.

```
1import cv2
2import numpy as np
3
```

```
# video playing
cap = cv2.VideoCapture('video.mp4')

min_width_rectangle = 80 # min_width_rectangle
min_height_rectangle = 80 # min_height_rectangle
count_line_position= 550

#initialize substructor
# it substract the background the  vehicle
algo = cv2.bgsegm.createBackgroundSubtractorMOG()
##? for detecting the vehicle in the vehicle
def center_handle(x,
y,w,h):
        x1=int(w/2)
        y1=int(h/2)

        cx = x+x1
        cy = y+y1
        return cx,cy

detect = []
# allow error between pixel
offset=6
counter = 0

while True:
    ret,frame1= cap.read()

    grey = cv2.cvtColor(frame1,cv2.COLOR_BGR2GRAY)

        blur = cv2.GaussianBlur(grey,(3,3),5)
        #applying on each frame
        img_sub = algo.apply(blur)
        dilat = cv2.dilate(img_sub,np.ones((5,5)))
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
        dilateada = cv2.morphologyEx(dilat,cv2.MORPH_CLOSE,kernel)
        dilateada = cv2.morphologyEx(dilateada,cv2.MORPH_CLOSE,kernel)

```

```
43t Sh      h          2 fi dC t(dil t d   2 RETR TREE       2 CHAIN APPROX SIMPLE) 40    dilateada =
cv2.morphologyEx(dilat,cv2.MORPH_CLOSE,kernel)
41    dilateada = cv2.morphologyEx(dilateada,cv2.MORPH_CLOSE,kernel)
42
```

```
counterShape,h = cv2.findContours(dilateada, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        #it is for the line in the cv2
        #cv2.line(frame1,(25,count_line_position),(1200,count_line_position),(255,127,0),3)
# for the rectangle in the video    for (i,c) in enumerate(counterShape):
            (x,y,w,h)= cv2.boundingRect(c)
            validate_counter =(w>=min_width_rectangle) and (h>=min_height_rectangle)        if not validate_counter:
        continue


            cv2.rectangle(frame1,(x,y),(x+w,y+h),(0,0,255),2)
            cv2.putText(frame1,"vehicle"+str(counter),(x,y-20 ),cv2.FONT_HERSHEY_TRIPLEX,1, (255,244,0),2)


    # loop for the detection

            center = center_handle(x,y,w,h)
            detect.append(center)
            cv2.circle(frame1,center,4,(0,255,0),-1)


            for (x,y)in detect:
            if y<(count_line_position+offset) and y>(count_line_position-offset):
          counter+=1

        #cv2.line(frame1,(25,count_line_position),(1200,count_line_position), (0,127,255),3)

        detect.remove((x,y))


        print("vehicle counter "+str(counter))

 cv2.putText(frame1,"Vehicle counter:" +str(counter),
(450 70) cv2 FONT HERSHEY SIMPLEX 2 (0 0 255) 5)
```

47

```
48   # for the rectangle in the video 49    for (i,c) in
enumerate(counterShape):
50              (x,y,w,h)= cv2.boundingRect(c)
51              validate_counter =(w>=min_width_rectangle) and (h>=min_height_rectangle) 52        if not
                validate_counter:
53          continue
54
55
56              cv2.rectangle(frame1,(x,y),(x+w,y+h),(0,0,255),2)
57              cv2.putText(frame1,"vehicle"+str(counter),(x,y-20 ),cv2.FONT_HERSHEY_TRIPLEX,1, (255,244,0),2)
58
59
60      # loop for the detection
61
62              center = center_handle(x,y,w,h)
63              detect.append(center)
64              cv2.circle(frame1,center,4,(0,255,0),-1)
65
66
67              for (x,y)in detect:
68              if y<(count_line_position+offset) and y>(count_line_position-offset): 69
70             counter+=1
71
72          #cv2.line(frame1,(25,count_line_position),(1200,count_line_position), (0,127,255),3)
73
74          detect.remove((x,y))
75
76
77          print("vehicle counter "+str(counter))
78
79   cv2.putText(frame1,"Vehicle counter:" +str(counter),
    (450,70),cv2.FONT_HERSHEY_SIMPLEX,2,(0,0,255),5)
80
81
82
83          #cv2.imshow('Detecter',dilateada)
84          cv2.imshow('Video Original',frame1)
85
86   if  cv2.waitKey(1)==13:
8b                      k
```

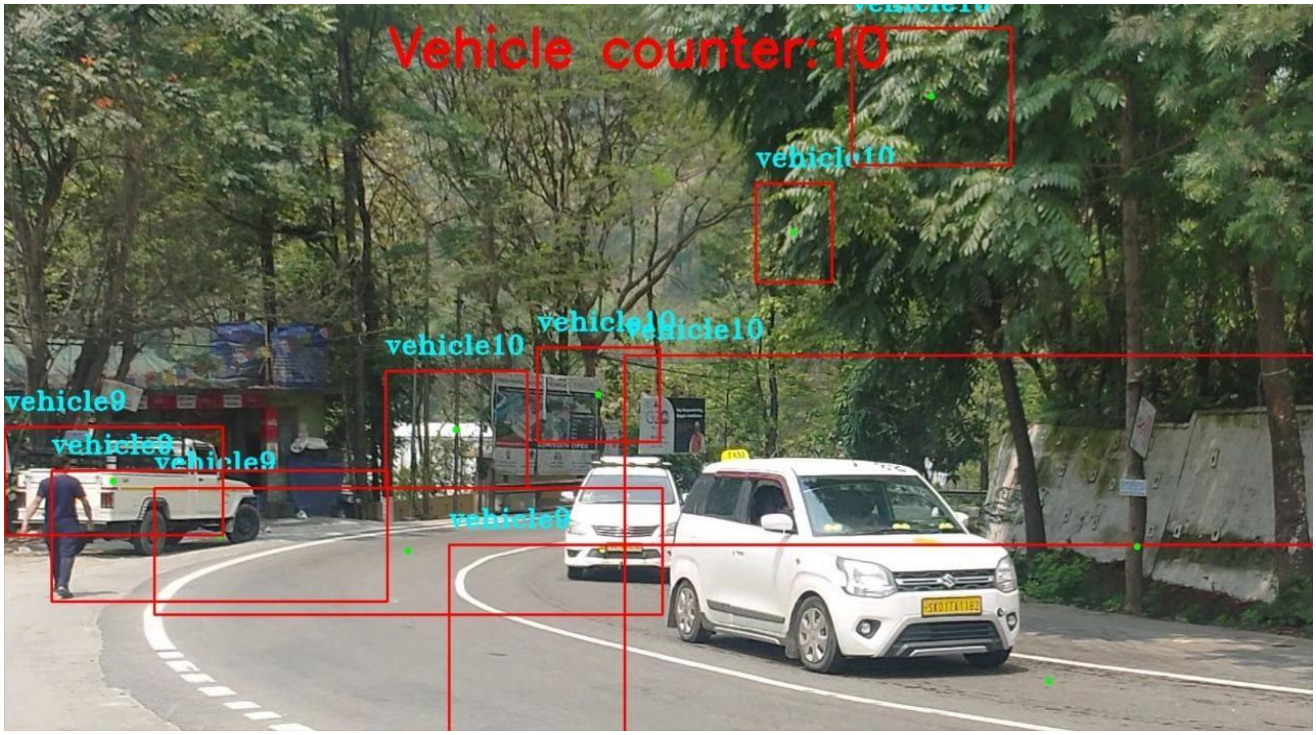


FIG: It's counting the vehicle count

Conclusion: In conclusion, the implementation of motion detection using OpenCV in Python provides participants with hands-on experience in applying computer vision concepts to real-world scenarios. The ability to detect motion in video streams has broad applications, including security systems, surveillance, and human activity analysis. The experiment reinforces the understanding of image processing techniques, such as frame differencing and thresholding, and how they can be effectively used for motion analysis. Participants, upon completion of this experiment, will have acquired practical skills that can be extended to more advanced computer vision projects involving motion detection and tracking.

LAB 12

AIM: Implementation of basic image processing operation in Python.

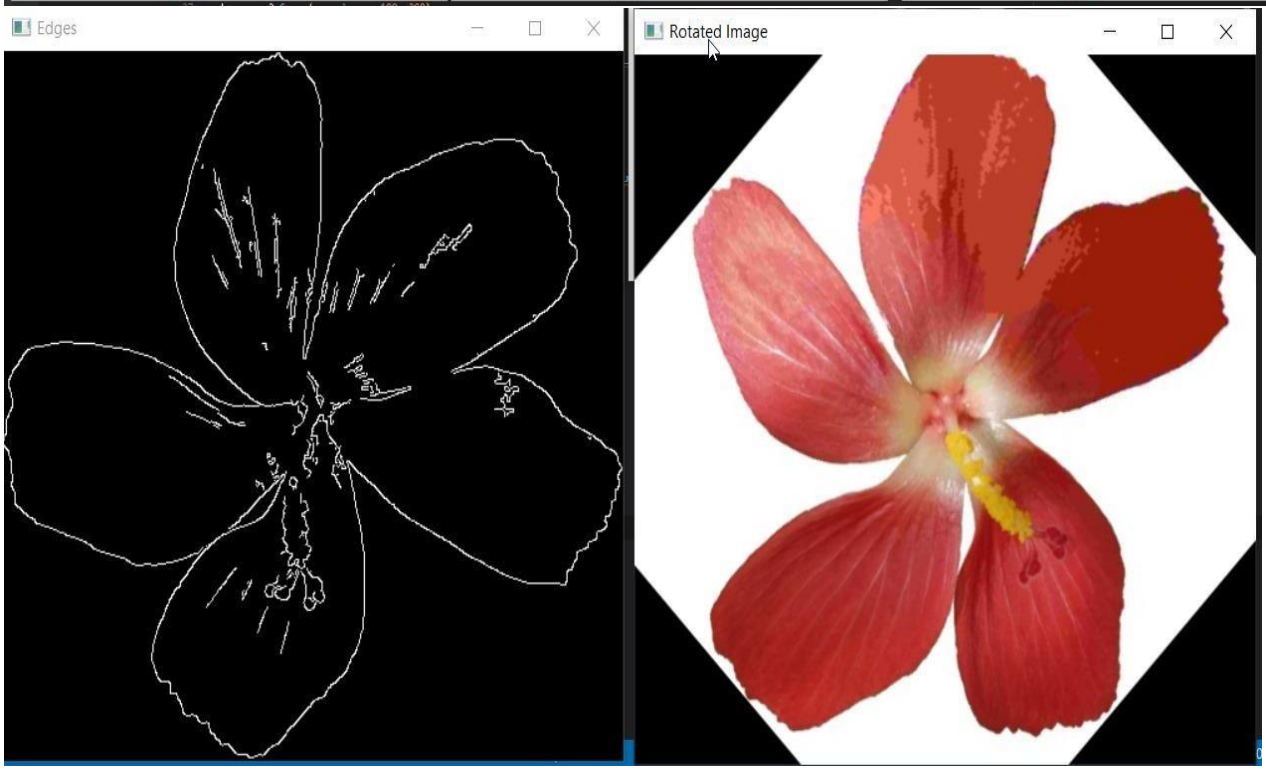
The aim of this experiment is to implement basic image processing operations in Python. The primary objective is to provide participants with hands-on experience in manipulating digital images using fundamental techniques. Through this experiment, participants will gain insights into the essential operations involved in image processing and their applications.

Theory: Basic image processing operations encompass a range of fundamental techniques applied to digital images. These operations include tasks such as reading and displaying images, resizing, rotating, and applying various filters. In Python, the OpenCV library is commonly used for image processing tasks. The experiment involves loading an image from a file, displaying it, and then performing operations such as resizing and rotating. Filtering operations, like blurring and sharpening, can be applied to understand their effects on image quality. Additionally, color space conversions, such as converting an image to grayscale, can be explored. Understanding these fundamental operations provides a foundation for more advanced image processing applications.

```
1import cv2
2import numpy as np
3
4def rotate_image(image, angle): 5    rows,
cols, _ = image.shape
6    M = cv2.getRotationMatrix2D((cols/2, rows/2), angle, 1) 7    return
cv2.warpAffine(image, M, (cols, rows))
8
```

```python
# Load an image
image = cv2.imread('JPEG_example_flower.jpg')

# Display the original image
cv2.imshow('Original Image', image)
cv2.waitKey(0)

# Convert the image to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow('Grayscale Image', gray_image)
cv2.waitKey(0)

# Apply Gaussian blur
blurred_image = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow('Blurred Image', blurred_image)
cv2.waitKey(0)

# Apply Canny edge detection
edges = cv2.Canny(gray_image, 100, 200)
cv2.imshow('Edges', edges)
cv2.waitKey(0)

# Rotate the image
rotated_image = rotate_image(image, 45)
cv2.imshow('Rotated Image', rotated_image)
cv2.waitKey(0)

# Release resources
cv2.destroyAllWindows()
```


Original Image
Grayscale Image
Blurred Image


Edges
Rotated Image

Conclusion: In conclusion, the implementation of basic image processing operations in Python equips participants with fundamental skills in manipulating digital images. The experiment covers essential tasks that serve as building blocks for more advanced image processing and computer vision projects.

Participants will gain proficiency in using Python libraries like OpenCV for tasks such as image loading, display, resizing, and filtering. The knowledge acquired in this experiment lays the groundwork for exploring more complex image processing techniques and applications in fields such as computer vision, medical imaging, and digital media.