LAB 5:

1. Write a program that is used to calculate and display the histogram of a grayscale image.

What is a histogram in an image?

A histogram shows how many pixels in an image have each possible intensity value. It helps analyze the contrast, brightness, and exposure of an image.

a. A balanced histogram → good contrast.
b. A shifted histogram → too dark or too bright image.

What does this do practically?

It visualizes the distribution of pixel intensities and is useful for image processing, like:

a. Contrast adjustment.
b. Histogram equalization.
c. Thresholding.

Code Implementation:

```
import cv2

import matplotlib.pyplot as plt

#load img in GrayScale

image_path = "D:\Kunaa_____l\multi-media\lenna.jpg"

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

plt.hist(image.ravel(), bins = 256, range=[0,256], color='gray', alpha=0.7)

plt.title("Histogram of Lenna.jpg")

plt.xlabel("Pixel Value")

plt.ylabel("Frequency")

plt.show()
```

2. Write a program to create a random grayscale image of size 400 x 400 pixels and save it as a `.jpg` file.

Why is this useful ?

- To test image processing algorithms.
- To generate random textures.
- To check filters or transformations on noisy data.
- Just for fun! (Random art 😄)

Code Implementation:

```
import numpy as np
import cv2
# Generate random image (400x400) with values in range [0, 256)
random_image = np.random.randint(0, 256, (400, 400), dtype=np.uint8)
# Save the image
output_path = "D:\\Kunaa_____l\\multi-media\\random_image.jpg"

cv2.imwrite(output_path, random_image)

print(f"Random image saved at {output_path}")
```

From the code implementation above, the image will appear as a static noise or TV static kind of image, because every pixel has a random gray value.

3. Write a program to demonstrate the concept of sampling a continuous sine wave and visualize how discrete samples represent the continuous signal.

What is a sine wave?

A sine wave is a smooth, periodic oscillation. It's defined by:

y(t)=A⋅sin(2πft+ϕ)y(t) = A \cdot \sin(2\pi f t + \phi)y(t)=A⋅sin(2πft+ϕ)

Where:

- A = Amplitude (height of the wave)
- f = Frequency (how many cycles per second, in Hz)
- t = Time
- φ (phi) = Phase shift (horizontal shift)

Sampling:
Sampling means taking measurements of a continuous signal at regular time intervals. The time between samples is called the sampling interval.The number of samples taken per second is called the sampling rate.

Why is this important?

- This is the core of digital signal processing (DSP).
- In audio, video, and communication, real-world signals are continuous, but computers handle discrete data.
- Correct sampling ensures we can accurately reconstruct the original signal.

Important theory link: Nyquist Theorem

The Nyquist Theorem says that to capture a signal correctly, the sampling rate must be at least twice the highest frequency in the signal.

For a 3 Hz wave, sampling should be 6 Hz or higher to avoid aliasing (distortion caused by low sampling rates).

Code Implementation:

```
import numpy as np

import matplotlib.pyplot as plt

frequency = 3  # Hz

amplitude = 1

phase = 0

interval = 0.3  # Sampling interval

samples = 10  # Number of samples

time_samples = np.arange(0, samples * interval, interval)

sine_wave_values = amplitude * np.sin(2 * np.pi * frequency * time_samples + phase)

plt.scatter(time_samples, sine_wave_values, color='red', label="Samples")

plt.plot(time_samples, sine_wave_values, linestyle='dashed', color='blue', label="Sine Wave")

plt.xlabel("Time (seconds)")
```

```
plt.ylabel("Amplitude")

plt.title("Sine Wave Samples at 3Hz")

plt.legend()

plt.show()

import numpy as np

import matplotlib.pyplot as plt

frequency = 3  # Hz

amplitude = 1

phase = 0

interval = 0.01  # Sampling interval

samples = 10  # Number of samples

time_samples = np.arange(0, samples * interval, interval)

# Compute sine wave values

sine_wave_values = amplitude * np.sin(2 * np.pi * frequency * time_samples + phase)

# Generate continuous sine wave for comparison

continuous_time = np.linspace(0, samples * interval, 1000)

continuous_sine_wave = amplitude * np.sin(2 * np.pi * frequency * continuous_time + phase)

# Plot samples and continuous sine wave

plt.scatter(time_samples, sine_wave_values, color='red', label="Samples")

plt.plot(time_samples, sine_wave_values, linestyle='dashed', color='blue', label="Interpolated
Sine Wave")

plt.plot(continuous_time, continuous_sine_wave, color='green', label="Continuous Sine Wave")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")
```

```
plt.title("Sine Wave Samples at 3Hz")

plt.legend()

plt.grid(True)

plt.show()
```

4. Write a program that generates and plots a sine wave and its sampled points at a given sampling rate.

1. Sine Wave:

A sine wave is a mathematical curve that represents periodic oscillations. It is commonly used to model oscillatory behaviors such as sound waves and electrical signals.

The general equation of a sine wave is:

y(t)=A⋅sin(2πft+ϕ)y(t) = A \cdot \sin(2 \pi f t + \phi)y(t)=A⋅sin(2πft+ϕ)

Where:

- A = Amplitude (how tall the wave is, 1 in this case).
- f = Frequency (how many cycles per second, 3 Hz here).
- t = Time variable.
- φ = Phase shift (0 in this case, meaning no horizontal shift).

2. Sampling:

Sampling is the process of converting a continuous signal (like a sine wave) into a discrete signal by measuring its value at specific intervals.

- Sampling rate determines how often the signal is sampled per second, expressed in Hz (samples per second).

For example, a sampling rate of 2 Hz means that the signal will be sampled every 0.5 seconds (1/2 Hz).

Sampling The Sine Wave:

<u>Parameters:</u>

- Frequency of the sine wave (f) = 3 Hz: This means that in one second, the sine wave completes 3 full cycles.\

- Amplitude (A) = 1: This controls the height of the wave, but the sine wave oscillates between +1 and -1.
- Sampling Rate = 2 Hz: This means we take samples at 2 points per second, i.e., once every 0.5 seconds.

<u>Sampling Process:</u>

The program computes the sine wave for each time step in time = np.arange(0, 10, 1/sampling_rate), which creates an array of time values from 0 to 10 seconds, sampled every 0.5 seconds (since 1/2 Hz = 0.5s).

The sine wave values are then calculated at these specific time points, and the program plots them.

<u>Visualizing the Sampling:</u>

- Red dots: These represent the sampled points of the sine wave at the specified sampling rate of 2 Hz. This shows how the sine wave is sampled at intervals of 0.5 seconds.
- Blue dashed line: This is the continuous sine wave which would be the full wave if you were to measure it continuously without any sampling.

What Happens with 2 Hz Sampling Rate?

- Since the sampling rate is lower than the sine wave frequency (3 Hz), the program is undersampling the sine wave. This means we don't capture enough data points to fully represent the wave.
- Aliasing: With a sampling rate that is too low, you risk missing crucial information about the signal, resulting in a phenomenon called aliasing, where the sampled points can incorrectly represent a lower-frequency signal.
- In the plot, the red dots appear to follow the sine wave, but they do not capture the full oscillation of the wave between samples, which distorts the true representation of the continuous wave.

Why is Sampling Rate Important?

To accurately represent a signal, you need a sampling rate that is at least twice the highest frequency of the signal (this is called the Nyquist-Shannon sampling theorem). If the sampling rate is too low, aliasing can occur, and the signal will not be faithfully reconstructed.

- For a 3 Hz sine wave, the minimum required sampling rate is 6 Hz (i.e., at least 6 samples per second). With a 2 Hz sampling rate, we are below the Nyquist rate, resulting in poor representation of the wave.

Program implementation:

```
# The program visually demonstrates the effects of undersampling a sine wave, showing how
#the sampling rate affects the accuracy and fidelity of the signal representation.

import numpy as np
import matplotlib.pyplot as plt\
# Define sine wave parameters
frequency = 3  # Hz
amplitude = 1  # Amplitude of the sine wave
phase = 0      # Phase shift
sampling_rate = 2  # Hz (Sampling rate)
time = np.arange(0, 10, 1/sampling_rate)  # From 0 to 10 seconds with 2Hz sampling
sine_wave_values = amplitude * np.sin(2 * np.pi * frequency * time + phase)
# Plot the sampled points
plt.scatter(time, sine_wave_values, color='red', label="Samples")
plt.plot(time, sine_wave_values, linestyle='dashed', color='blue', label="Sine Wave")
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude")
plt.title("Sine Wave with 2Hz Sampling Rate")
plt.legend()
plt.show()
```

5. WAP that demonstrates the concept of sampling a sine wave at the Nyquist rate to avoid aliasing.

1. Nyquist Theorem:

The Nyquist-Shannon Sampling Theorem states that in order to accurately sample a signal without losing information, the sampling rate must be at least twice the frequency of the signal. This rate is referred to as the Nyquist rate.

- Nyquist rate = 2 × frequency of the signal
- If a sine wave has a frequency of 3 Hz, the minimum sampling rate needed to avoid aliasing is 6 Hz.

2. Sine Wave:

A sine wave is a smooth, periodic oscillation that is mathematically described by the equation:

$y(t)=A \cdot \sin(2\pi ft+\phi)$y(t) = A \cdot \sin(2 \pi f t + \phi)$y(t)=A \cdot \sin(2\pi ft+\phi)$

Where:

- A = Amplitude (1 in this case)

- f = Frequency (3 Hz here)
- t = Time variable (in seconds)
- φ = Phase shift (0 in this case)

For a 3 Hz sine wave:

- The wave completes 3 full cycles per second.
- Its period (time to complete one full cycle) is 1/3 seconds.

Explanation of Code:

1. Parameters:

- frequency = 3 Hz: The sine wave has a frequency of 3 Hz, meaning it completes 3 cycles per second.
- amplitude = 1: The peak value of the sine wave is 1 (it oscillates between +1 and -1).
- phase = 0: There is no phase shift; the wave starts at t = 0.
- sampling_rate = 6 Hz: The sampling rate is set to 6 Hz, which is twice the frequency of the sine wave (as per the Nyquist theorem).
- time: This is the time vector, ranging from 0 to 10 seconds, with a step size of 1/6 seconds (since the sampling rate is 6 Hz).

2. Compute the Sine Wave: The sine wave values are computed using the formula y(t)=sin(2πft)y(t) = \sin(2 \pi f t)y(t)=sin(2πft), where the time vector is calculated using the given sampling rate.

3. Plotting the Wave:

- plt.plot(time, sine_wave_values, label="Sine Wave (3 Hz)", color='b'): This plots the continuous sine wave using a blue line.
- plt.scatter(time, sine_wave_values, color='r', label="Sampled Points", zorder=5): This marks the sampled points (the points where the sine wave is sampled at the 6 Hz rate) with red dots.

4. Plot Labels and Title:

- The x-axis is labeled as "Time (seconds)".
- The y-axis is labeled as "Amplitude".
- The plot is titled "Sine Wave with 6Hz Sampling Rate (Nyquist Theorem)".
- The legend is added to differentiate between the sine wave and the sampled points.

Visualizing the Plot:

1. Continuous Sine Wave: The blue line represents the continuous sine wave, showing how the wave oscillates between +1 and -1 over time.
2. Sampled Points: The red dots represent the sampled points at intervals determined by the 6 Hz sampling rate. Since the sampling rate is exactly twice the frequency of the sine wave, the red dots align well with the continuous sine wave. This is a proper sampled representation, showing no aliasing.

Code implementation:

```
import numpy as np
import matplotlib.pyplot as plt
# Parameters for the sine wave
frequency = 3  # Hz
amplitude = 1  # Amplitude
phase = 0      # Phase
sampling_rate = 6  # Nyquist theorem: Sampling rate should be >= 2 * frequency
time = np.arange(0, 10, 1/sampling_rate)  # Time vector from 0 to 10 seconds
sine_wave_values = amplitude * np.sin(2 * np.pi * frequency * time + phase)

plt.plot(time, sine_wave_values, label="Sine Wave (3 Hz)", color='b')
plt.scatter(time, sine_wave_values, color='r', label="Sampled Points", zorder=5)

# Label the plot
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude")
plt.title("Sine Wave with 6Hz Sampling Rate (Nyquist Theorem)")
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

LAB 6:

<u>First:</u>

Write a program that demonstrates how to play audio files in WAV and MIDI formats using Pydub and Pygame libraries. Here's the theory and explanation of each part of the program:

1. WAV and MIDI File Formats:

- WAV (Waveform Audio File Format): A common uncompressed audio file format that stores waveform data. WAV files are usually larger but provide high-quality sound without compression artifacts.
- MIDI (Musical Instrument Digital Interface): A protocol used for sending music data between devices. Unlike WAV, MIDI files don't store actual audio data but instead contain information on notes, timing, and instrument control.

2. Required Libraries:

- Pydub: A library that allows you to work with audio files in different formats. It is used here to load and play WAV files.
- Pygame: A library used for writing video games, but it also has audio capabilities. It is used to load and play MIDI files.

This allows you to play both WAV and MIDI files. Here's a summary of its workflow:

1. The program attempts to find and play a WAV file using Pydub.
2. If the WAV file is not found, it attempts to find and play a MIDI file using Pygame.
3. The Pydub library handles playing uncompressed audio (WAV), while Pygame handles playing MIDI (musical instructions).

This can be useful for applications such as music players, game audio, or sound testing tools where you need to play different types of audio files.

Code Implementation:

```
pip install pydub pygame

import os

from pydub import AudioSegment

from pydub.playback import play

import pygame
```

```
def play_wav(file_path):

audio = AudioSegment.from_wav(file_path)

# Play the WAV file

play(audio)

def play_midi(file_path):

# Initialize pygame mixer

pygame.mixer.init()

# Load the MIDI file

pygame.mixer.music.load(file_path)

# Play the MIDI file

pygame.mixer.music.play()

# Wait until the music finishes playing

while pygame.mixer.music.get_busy():

pygame.time.Clock().tick(10)

def main():

# Example file paths

wav_file = &quot;example.wav&quot;

midi_file = &quot;example.mid&quot;

if os.path.exists(wav_file):

print(f&quot;Playing WAV file: {wav_file}&quot;)

play_wav(wav_file)

else:

print(f&quot;WAV file not found: {wav_file}&quot;)
```

```
if os.path.exists(midi_file):

print(f&quot;Playing MIDI file: {midi_file}&quot;)

play_midi(midi_file)

else:

print(f&quot;MIDI file not found: {midi_file}&quot;)

if __name__ == &quot;__main__&quot;:

main()
```

Second:
Write a program to simulate the game of pool table.

```
pip install pygame
import pygame
import math
# Constants
WIDTH, HEIGHT = 800, 400
TABLE_COLOR = (39, 119, 20)
BALL_COLOR = (255, 255, 255)
HOLE_COLOR = (0, 0, 0)
BALL_RADIUS = 10
BALL_MASS = 1
FRICTION = 0.01
# Initialize Pygame
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption(&quot;Pool Table Simulation&quot;)
clock = pygame.time.Clock()

class Ball:
def __init__(self, x, y, radius, color):
self.x = x
self.y = y
self.radius = radius
self.color = color
self.vx = 0
self.vy = 0
def draw(self, screen):
```

```
pygame.draw.circle(screen, self.color, (int(self.x), int(self.y)),
self.radius)
def update(self):
self.x += self.vx
self.y += self.vy
# Apply friction
self.vx *= (1 - FRICTION)
self.vy *= (1 - FRICTION)
# Bounce off walls
if self.x &lt;= self.radius or self.x &gt;= WIDTH - self.radius:
self.vx = -self.vx
if self.y &lt;= self.radius or self.y &gt;= HEIGHT - self.radius:
self.vy = -self.vy
def collide(self, other):
dx = other.x - self.x
dy = other.y - self.y
distance = math.hypot(dx, dy)

if distance &lt; self.radius + other.radius:
angle = math.atan2(dy, dx)
self.vx, other.vx = other.vx, self.vx
self.vy, other.vy = other.vy, self.vy
def main():
ball1 = Ball(WIDTH // 2, HEIGHT // 2, BALL_RADIUS,
BALL_COLOR)
ball2 = Ball(WIDTH // 2 + 50, HEIGHT // 2, BALL_RADIUS,
BALL_COLOR)
balls = [ball1, ball2]
running = True
while running:
for event in pygame.event.get():
if event.type == pygame.QUIT:
running = False
elif event.type == pygame.MOUSEBUTTONDOWN:
if event.button == 1: # Left mouse button
mx, my = pygame.mouse.get_pos()
for ball in balls:
dx = mx - ball.x
dy = my - ball.y
distance = math.hypot(dx, dy)
if distance &lt; ball.radius:
ball.vx = dx * 0.1
```

```
ball.vy = dy * 0.1
# Update ball positions
for ball in balls:
ball.update()
# Handle collisions
for i in range(len(balls)):
for j in range(i + 1, len(balls)):
balls[i].collide(balls[j])
# Drawing
screen.fill(TABLE_COLOR)
for ball in balls:
ball.draw(screen)
pygame.display.flip()
clock.tick(60)
pygame.quit()
if __name__ == &quot;__main__&quot;:
main()
```

<u>Third:</u>

Q. Write a program to simulate the game minesweeper

```
pip install pygame
import pygame
import random
# Constants
WIDTH, HEIGHT = 600, 600
ROWS, COLS = 10, 10
MINES = 10
CELL_SIZE = WIDTH // COLS
FONT_SIZE = 24
# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (192, 192, 192)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
# Initialize Pygame
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption(&quot;Minesweeper&quot;)
font = pygame.font.SysFont(None, FONT_SIZE)
class Cell:
```

```
def __init__(self, row, col):
self.row = row
self.col = col
self.is_mine = False
self.is_revealed = False
self.is_flagged = False
self.adjacent_mines = 0
def draw(self, screen):
x = self.col * CELL_SIZE
y = self.row * CELL_SIZE
rect = pygame.Rect(x, y, CELL_SIZE, CELL_SIZE)
pygame.draw.rect(screen, WHITE, rect)
pygame.draw.rect(screen, BLACK, rect, 1)
if self.is_revealed:
if self.is_mine:

pygame.draw.circle(screen, BLACK, (x + CELL_SIZE //
2, y + CELL_SIZE // 2), CELL_SIZE // 3)
else:
text = font.render(str(self.adjacent_mines), True,
BLACK)
screen.blit(text, (x + CELL_SIZE // 3, y + CELL_SIZE
// 3))
elif self.is_flagged:
pygame.draw.rect(screen, RED, rect)
def create_grid(rows, cols, mines):
grid = [[Cell(row, col) for col in range(cols)] for row in
range(rows)]
mine_positions = set()
while len(mine_positions) &lt; mines:
row = random.randint(0, rows - 1)
col = random.randint(0, cols - 1)
if (row, col) not in mine_positions:
mine_positions.add((row, col))
grid[row][col].is_mine = True
for row in range(rows):
for col in range(cols):
if not grid[row][col].is_mine:
grid[row][col].adjacent_mines = sum(
grid[r][c].is_mine
for r in range(row - 1, row + 2)
for c in range(col - 1, col + 2)
```

```
if 0 &lt;= r &lt; rows and 0 &lt;= c &lt; cols
return grid
def reveal_cell(grid, row, col):
if grid[row][col].is_revealed or grid[row][col].is_flagged:
return
grid[row][col].is_revealed = True
if grid[row][col].adjacent_mines == 0:
for r in range(row - 1, row + 2):
for c in range(col - 1, col + 2):
if 0 &lt;= r &lt; ROWS and 0 &lt;= c &lt; COLS and not
grid[r][c].is_mine:
reveal_cell(grid, r, c)
def main():
grid = create_grid(ROWS, COLS, MINES)
running = True
game_over = False
while running:
for event in pygame.event.get():
if event.type == pygame.QUIT:
running = False
elif event.type == pygame.MOUSEBUTTONDOWN and
not game_over:
x, y = pygame.mouse.get_pos()
row, col = y // CELL_SIZE, x // CELL_SIZE
if event.button == 1: # Left click
if grid[row][col].is_mine:
game_over = True
for row in range(ROWS):
for col in range(COLS):
grid[row][col].is_revealed = True
else:
reveal_cell(grid, row, col)
elif event.button == 3: # Right click
grid[row][col].is_flagged = not
grid[row][col].is_flagged
screen.fill(GRAY)
for row in range(ROWS):
for col in range(COLS):
grid[row][col].draw(screen)
pygame.display.flip()
pygame.quit()
```

```
if __name__ == &quot;__main__&quot;:
main()
.
```

LAB 7:

1. Given the wave file, write a python program to get different parameters of the wave file such as number of channels, sampling width (bit depth), sampling rate, number of samples. Use a wave library.

Code Implementation:

```
# Lab 7a
import wave
# Open the WAV file in read mode
wav_file = wave.open(r"D:\Kunaa_____l\multi-media\lab7\example.wav", "r")
# Get the parameters of the WAV file
num_channels = wav_file.getnchannels()  # Corrected method
sample_width = wav_file.getsampwidth()
sample_rate = wav_file.getframerate()
num_samples = wav_file.getnframes()
# Close the WAV file
wav_file.close()
# Print the parameters
print("Number of channels: ", num_channels)
print("Sampling width (bit depth): ", sample_width)
print("Sampling rate: ", sample_rate)
print("Number of samples: ", num_samples)
```

2. Let us consider a sine wave with frequency 4400hz, 400m amplitude and phase0. Generate a sample from this sine wave at the rate of 44100hz for 1sec. Save the samples in the form of a wave file in python. Use a wave library.

Code Implementation:

```
# Lab 7b
import wave
import math
frequency = 440  # Hz (or keep 4400 if you like)
amplitude = 32767  # Max for 16-bit audio
phase = 0
duration = 1  # seconds
sample_rate = 44100
num_samples = int(sample_rate * duration)
with wave.open('sine_wave.wav', 'w') as file:
    file.setnchannels(1)  # Mono
    file.setsampwidth(2)  # 2 bytes = 16-bit
    file.setframerate(sample_rate)
    for i in range(num_samples):
```

```
        time = i / sample_rate
        sample = amplitude * math.sin(2 * math.pi * frequency * time + phase)
        sample_int = int(sample)
        sample_bytes = sample_int.to_bytes(2, byteorder='little', signed=True)
        file.writeframes(sample_bytes)
```

3. Implement 1D DCT (discrete cosine transform) in python.

Code Implementation:

```
# Lab 7c
import numpy as np
def dct(signal):
    N = len(signal)
    dct_coef = np.zeros(N)
    for k in range(N):
        sum = 0
        for n in range(N):
            sum += signal[n] * np.cos(np.pi * k * (2 * n + 1) / (2 * N))
        dct_coef[k] = sum * np.sqrt(2 / N)
        if k == 0:
            dct_coef[k] *= np.sqrt(1 / 2)
    return dct_coef
# Example usage
signal = [1, 2, 3, 4]
dct_result = dct(signal)
print("DCT result:", dct_result)
```

4. Implement run length encoding.

Code Implementation:

```
def run_length_encoding(input_string):
    count = 1
    prev_char = input_string[0]
    output = ""
    for char in input_string[1:]:
        if char == prev_char:
            count += 1
        else:
            output += str(count) + prev_char
```

```
            count = 1
            prev_char = char
    output += str(count) + prev_char

    return output

input_string = "aaabbcccdaa"
encoded_string = run_length_encoding(input_string)
print(encoded_string)
```