Assignment: Image Formation

1: Image Formation

- Objective: Understand how geometric transformations affect images.
- Instructions:
    1. Write a function to perform translation, rotation, and scaling on an image without using any library functions.
    2. Use a sample image and apply each transformation.
    3. Display the original and transformed images side by side.

2: Photometric Models

- Objective: Understand how lighting conditions affect pixel intensities.
- Instructions:
    1. Write a function to simulate different lighting conditions (e.g., increase brightness, decrease brightness).
    2. Apply these changes to a sample image.
    3. Display the original and altered images.

3: Sampling and Quantization

- Objective: Understand the effects of sampling and quantization on image quality.
- Instructions:
    1. Write a function to downsample and upsample an image.
    2. Write a function to quantize an image to different levels (e.g., 2-bit, 4-bit).

3. Apply these functions to a sample image.
4. Display the results on image quality.

4: Image Definition and Neighbourhood Metrics

- Objective: Understand how images are defined and represented, and explore neighborhood metrics.
- Instructions:
    1. Define an image as a 2D matrix and implement functions to compute basic neighborhood metrics (e.g., mean, median, standard deviation within a neighborhood).
    2. Apply these metrics to a sample image.
    3. Display the results and discuss their significance.

Assignment 4

Sampling and Quantization

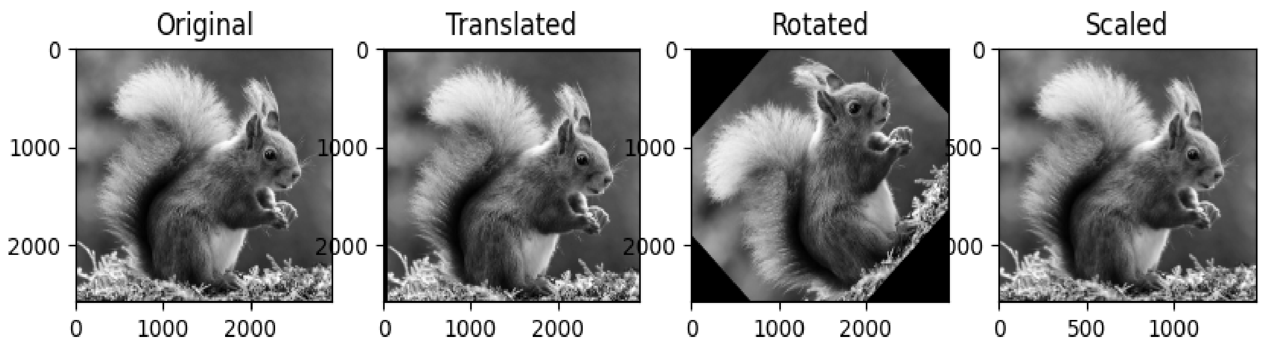
1: Image Formation

● Objective: Understand how geometric transformations affect images.
● Instructions:
1. Write a function to perform translation, rotation, and scaling on an image without using any library functions.
2. Use a sample image and apply each transformation.
3. Display the original and transformed images side by side.

```
import numpy as np
import matplotlib.pyplot as plt
import cv2

# Function to apply translation to an image
def translate(image, tx, ty):
    rows, cols = image.shape
    M = np.float32([[1, 0, tx], [0, 1, ty]])  # Translation matrix
    translated_image = cv2.warpAffine(image, M, (cols, rows))
    return translated_image

# Function to apply rotation to an image
def rotate(image, angle):
    rows, cols = image.shape
    M = cv2.getRotationMatrix2D((cols/2, rows/2), angle, 1)  # Rotation
matrix
    rotated_image = cv2.warpAffine(image, M, (cols, rows))
    return rotated_image

# Function to apply scaling to an image
def scale(image, fx, fy):
    scaled_image = cv2.resize(image, None, fx=fx, fy=fy,
interpolation=cv2.INTER_LINEAR)
    return scaled_image
# Load the sample image
image = cv2.imread('sample_image.jpg', 0)  # Load in grayscale

# Apply transformations
translated_image = translate(image, 50, 50)  # Translate by (50, 50)
rotated_image = rotate(image, 45)  # Rotate by 45 degrees
scaled_image = scale(image, 0.5, 0.5)  # Scale by 0.5

# Display original and transformed images side by side
```

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 4, 1), plt.imshow(image, cmap='gray'),
plt.title('Original')
plt.subplot(1, 4, 2), plt.imshow(translated_image, cmap='gray'),
plt.title('Translated')
plt.subplot(1, 4, 3), plt.imshow(rotated_image, cmap='gray'),
plt.title('Rotated')
plt.subplot(1, 4, 4), plt.imshow(scaled_image, cmap='gray'),
plt.title('Scaled')
plt.show()
```

Output:



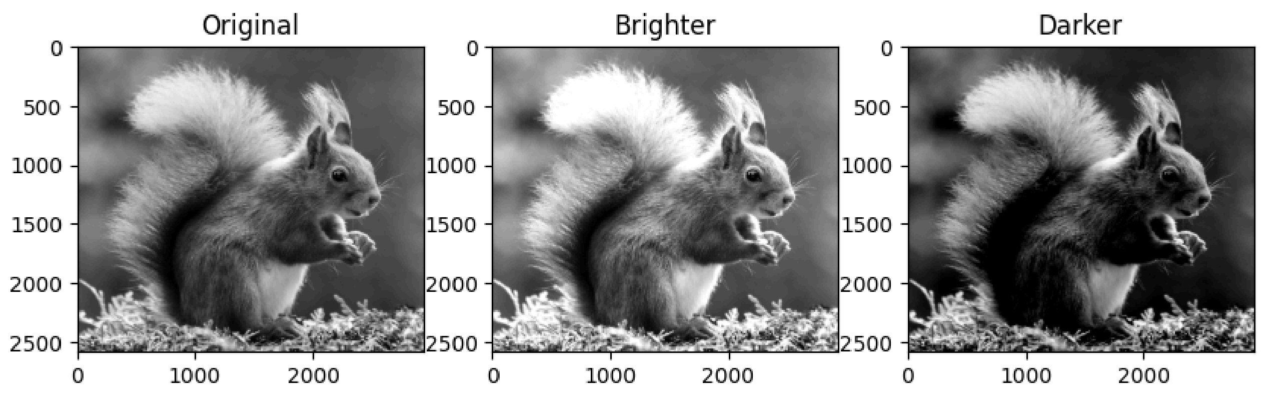
2: Photometric Models

● Objective: Understand how lighting conditions affect pixel intensities.
● Instructions:
1. Write a function to simulate different lighting conditions (e.g. increase brightness, decrease brightness).
2. Apply these changes to a sample image.
3. Display the original and altered images.

```
# Function to increase brightness
def increase_brightness(image, value):
    bright_image = cv2.add(image, value)
    return bright_image

# Function to decrease brightness
def decrease_brightness(image, value):
    dark_image = cv2.subtract(image, value)
    return dark_image

# Load the sample image
image = cv2.imread('sample_image.jpg', 0)  # Load in grayscale
# Apply lighting transformations
brighter_image = increase_brightness(image, 50)  # Increase brightness by
50
```

```python
darker_image = decrease_brightness(image, 50)  # Decrease brightness by 50

# Display original and altered images
plt.figure(figsize=(10, 5))
plt.subplot(1, 3, 1), plt.imshow(image, cmap='gray'),
plt.title('Original')
plt.subplot(1, 3, 2), plt.imshow(brighter_image, cmap='gray'),
plt.title('Brighter')
plt.subplot(1, 3, 3), plt.imshow(darker_image, cmap='gray'),
plt.title('Darker')
plt.show()
```

Output:

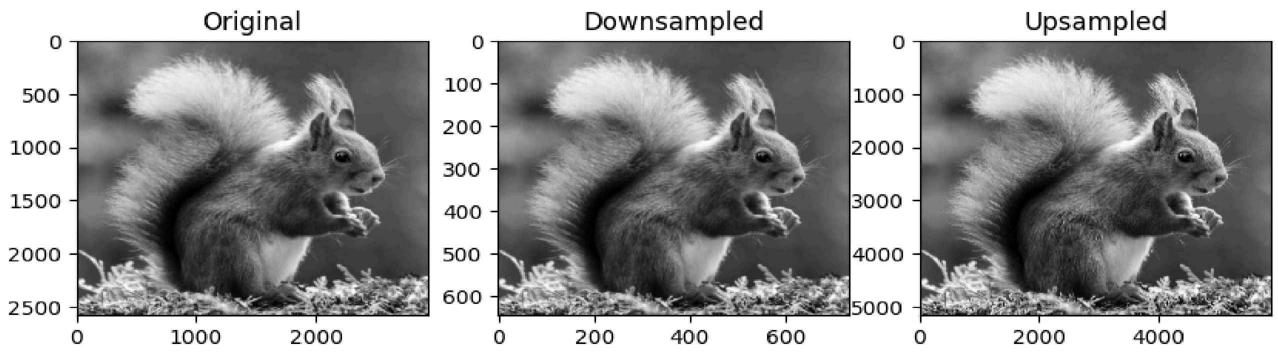


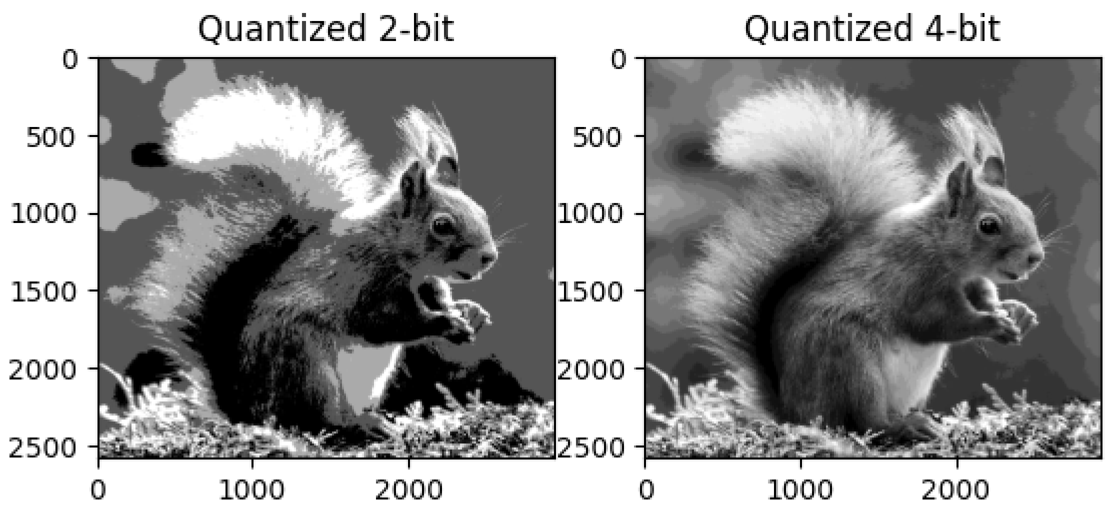
3: Sampling and Quantization

● Objective: Understand the effects of sampling and quantization on image quality.
● Instructions:
1. Write a function to downsample and upsample an image.
2. Write a function to quantize an image to different levels (e.g., 2-bit, 4-bit).
3. Apply these functions to a sample image.
4. Display the results on image quality.

```python
# Function to downsample an image (reduce resolution)
def downsample(image, factor):
    height, width = image.shape
    new_height, new_width = height // factor, width // factor
    downsampled_image = cv2.resize(image, (new_width, new_height),
interpolation=cv2.INTER_LINEAR)
    return downsampled_image

# Function to upsample an image (increase resolution)
def upsample(image, factor):
    height, width = image.shape
    new_height, new_width = height * factor, width * factor
    upsampled_image = cv2.resize(image, (new_width, new_height),
interpolation=cv2.INTER_LINEAR)
    return upsampled_image

# Function to quantize an image to n-bit
def quantize(image, n_bits):
    # Create the quantization factor
    quantization_level = 2 ** n_bits
    quantized_image = np.floor(image / (256 / quantization_level)) * (256
/ quantization_level)
    return quantized_image.astype(np.uint8)

# Load the sample image
image = cv2.imread('sample_image.jpg', 0)  # Load in grayscale
# Apply sampling transformations
downsampled_image = downsample(image, 4)  # Downsample by a factor of 4
upsampled_image = upsample(image, 2)  # Upsample by a factor of 2
# Apply quantization transformations
quantized_image_2bit = quantize(image, 2)  # Quantize to 2-bit
quantized_image_4bit = quantize(image, 4)  # Quantize to 4-bit

# Display results
plt.figure(figsize=(10, 10))
plt.subplot(2, 3, 1), plt.imshow(image, cmap='gray'),
plt.title('Original')
plt.subplot(2, 3, 2), plt.imshow(downsampled_image, cmap='gray'),
plt.title('Downsampled')
plt.subplot(2, 3, 3), plt.imshow(upsampled_image, cmap='gray'),
plt.title('Upsampled')
plt.subplot(2, 3, 4), plt.imshow(quantized_image_2bit, cmap='gray'),
plt.title('Quantized 2-bit')
plt.subplot(2, 3, 5), plt.imshow(quantized_image_4bit, cmap='gray'),
plt.title('Quantized 4-bit')
plt.show()
```

Output:





4: Image Definition and Neighbourhood Metrics

● Objective: Understand how images are defined and represented, and explore neighbourhood metrics.
● Instructions:
1. Define an image as a 2D matrix and implement functions to compute basic neighbourhood metrics (e.g., mean, median, standard deviation within a neighbourhood).
2. Apply these metrics to a sample image.
3. Display the results and discuss their significance.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import median_filter, convolve

# Load the image in grayscale
image = cv2.imread('sample_image.jpg', cv2.IMREAD_GRAYSCALE)

# Compute neighborhood metrics using efficient methods
mean_image = convolve(image, np.ones((3, 3))/9)  # Mean using convolution
median_image = median_filter(image, size=3)  # Median using filter
std_image = convolve(image, np.ones((3, 3))/9)  # Standard deviation
(simple approximation)

# Display the results
plt.figure(figsize=(10, 10))
plt.subplot(2, 2, 1), plt.imshow(image, cmap='gray'), plt.title('Original
Image')
plt.subplot(2, 2, 2), plt.imshow(mean_image, cmap='gray'), plt.title('Mean
Neighborhood')
```

```
plt.subplot(2, 2, 3), plt.imshow(median_image, cmap='gray'),
plt.title('Median Neighborhood')
plt.subplot(2, 2, 4), plt.imshow(std_image, cmap='gray'),
plt.title('Standard Deviation Neighborhood')

plt.show()
```

Output: