CS3008: IMAGE AND VIDEO PROCESSING

LABORATORY REPORTS

Name: Jayashre
Roll No.: 22011103020
College: Shiv Nadar University, Chennai

Chapter 1

Eye Detection Using OpenCV

1.1 Abstract

This report documents the implementation of an eye detection system using the OpenCV library. The system utilizes Haar cascade classifiers to identify faces and eyes in images and marks them with bounding boxes. The project aims to provide a foundational understanding of image processing techniques and their practical applications.

1.2 Introduction

Eye detection plays a vital role in computer vision applications such as gaze tracking, facial recognition, and user interaction systems. This project implements a detection system using pre-trained Haar cascade classifiers, which efficiently identify facial and ocular features in images.

1.3 Methodology

1.3.1 Data Collection

The input data consists of static images containing human faces. The images were uploaded manually in the Google Colab environment for processing.

1.3.2 Tools and Libraries

- **OpenCV**: For image processing and detection.
- **Google Colab**: For executing Python code and visualizing results.

1.3.3 Detection Algorithm

The steps followed in the implementation are:

1. Convert the input image to grayscale for easier processing.
2. Use the Haar cascade classifier to detect faces in the image.
3. For each detected face, apply the Haar cascade classifier for eyes within the facial region.
4. Highlight the detected features (faces and eyes) using bounding boxes.

1.4 Implementation

The implementation was carried out in Python using OpenCV. The following code snippet demonstrates the detection process:

Listing 1.1: Eye Detection Code

```
import cv2
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    ↪  'haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    ↪ 'haarcascade_eye.xml')
image = cv2.imread('input_image.jpg')
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray_image,
    ↪ scaleFactor=1.1, minNeighbors=5)

for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x+w, y+h), (255, 0, 0), 2)
    roi_gray = gray_image[y:y+h, x:x+w]
    roi_color = image[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (ex, ey, ew, eh) in eyes:
        cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh),
            ↪ (0, 255, 0), 2)
cv2.imwrite('output_image.jpg', image)
cv2.imshow('Eye Detection', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

1.5 Results and Discussion

The system was tested with several images under various conditions. The results are summarized below:

- Faces were detected accurately in well-lit images.

- Eye detection was successful but occasionally struggled with images where faces were partially obscured.

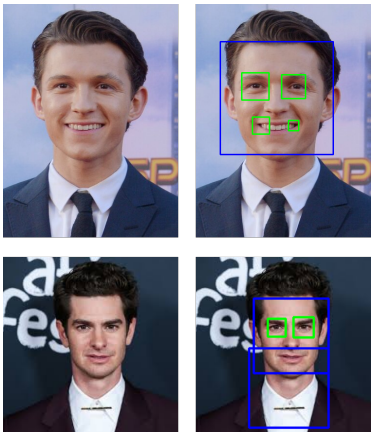

Figure 1.1: Detected faces and eyes in the input image.

1.6 Conclusion

The project successfully implemented an eye detection system using Haar cascades in OpenCV. While the system is efficient for basic detection, future improvements can involve the use of deep learning-based models for enhanced accuracy and robustness.

Chapter 2

Image Processing and Effects

2.1 Abstract

This report documents the implementation of various image processing techniques, including applying filters, resizing, cropping, rotating, and face mask overlays. These tasks showcase the practical applications of computer vision and image manipulation using Python and OpenCV.

2.2 Introduction

Image processing is a cornerstone of computer vision, enabling tasks like object detection, image enhancement, and feature extraction. This project demonstrates the use of OpenCV to apply filters and transformations to images, and overlay masks on detected faces.

2.3 Methodology

2.3.1 Data Collection

The input data consists of static images uploaded manually in the Google Colab environment.

2.3.2 Tools and Libraries

- **OpenCV**: For image processing and transformations.
- **Google Colab**: For executing Python code and visualizing results.
- **NumPy**: For handling numerical operations.

2.3.3 Image Processing Techniques

The project implements the following techniques:

- Grayscale, sepia, negative, and blur effects.
- Edge detection and cartoonification.
- Image resizing, cropping, and rotation.

2.4 Implementation

The implementation was carried out in Python using OpenCV. The following code snippet demonstrates the overall process:

Listing 2.1: Image Processing Code

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

image_path = "content/sample.png"
img = cv2.imread(image_path)

def apply_grayscale(image):
    return cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

def apply_sepia(image):
    kernel = np.array([[0.393, 0.769, 0.189],
                       [0.349, 0.686, 0.168],
                       [0.272, 0.534, 0.131]])
    return cv2.transform(image, kernel)

def apply_negative(image):
    return cv2.bitwise_not(image)

def apply_blur(image, ksize=(5, 5)):
    return cv2.GaussianBlur(image, ksize, 0)

def apply_edge_detection(image):
    return cv2.Canny(image, 100, 200)

def apply_cartoonify(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gray = cv2.medianBlur(gray, 5)
    edges = cv2.adaptiveThreshold(gray, 255, cv2.
        ↪ ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 9)
    color = cv2.bilateralFilter(image, 9, 300, 300)
    return cv2.bitwise_and(color, color, mask=edges)

```

```
def resize_image(image, width=500):
    aspect_ratio = width / float(image.shape[1])
    height = int(image.shape[0] * aspect_ratio)
    return cv2.resize(image, (width, height))

def crop_image(image):
    height, width = image.shape[:2]
    size = min(height, width)
    center_x, center_y = width // 2, height // 2
    cropped = image[center_y - size // 2:center_y + size //
        ↪ 2, center_x - size // 2:center_x + size // 2]
    return cropped

def rotate_image(image, angle=45):
    height, width = image.shape[:2]
    center = (width // 2, height // 2)
    matrix = cv2.getRotationMatrix2D(center, angle, 1)
    rotated = cv2.warpAffine(image, matrix, (width, height))
    return rotated


grayscale_img = apply_grayscale(img)
sepia_img = apply_sepia(img)
negative_img = apply_negative(img)
blur_img = apply_blur(img)
edges_img = apply_edge_detection(img)
cartoon_img = apply_cartoonify(img)
resized_img = resize_image(img)
cropped_img = crop_image(img)
rotated_img = rotate_image(img)

cv2_imshow(grayscale_img)
cv2_imshow(sepia_img)
cv2_imshow(negative_img)
cv2_imshow(blur_img)
cv2_imshow(edges_img)
cv2_imshow(cartoon_img)
cv2_imshow(resized_img)
cv2_imshow(cropped_img)
cv2_imshow(rotated_img)
```

2.5 Results and Discussion

The following observations were made during testing:

- Filters like grayscale, sepia, and negative worked as expected.
- Cartoonification produced visually appealing results by emphasizing edges.

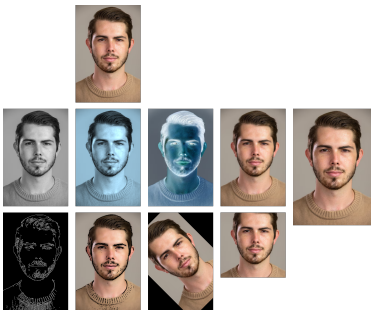

Figure 2.1: Examples of applied filters and transformations.

2.6 Conclusion

This project successfully demonstrated various image processing techniques using OpenCV. Future work could explore integrating deep learning models for more advanced image transformations and processing.

Chapter 3

Image Resolution and Interpolation Studies

3.1 Abstract

This report explores the fundamental concepts of image resolution and interpolation in image processing. The experiments involve converting RGB images to grayscale using a formula, analyzing intensity and spatial resolution changes, and studying image interpolation techniques. The study aims to provide insights into how image transformations affect visual quality and data representation.

3.2 Introduction

Image resolution and interpolation are critical aspects of image processing. Resolution defines the level of detail in an image, while interpolation determines how images are scaled to different dimensions. This project investigates these aspects through practical implementations and analyses their impact on image quality.

3.3 Methodology

3.3.1 Tools and Libraries

- **OpenCV**: For image processing operations.
- **NumPy**: For numerical computations.
- **Google Colab**: For implementation and visualization.

3.3.2 Experiments Conducted

1. **Convert RGB to Grayscale:** An RGB image is converted to grayscale using the formula:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

2. **Intensity Resolution:** An 8-bit grayscale image is converted into 7, 6, 5, 4, 3, and 2-bit images by reducing the bit depth and analyzing the resulting quality loss.

3. **Spatial Resolution:** A 512x512 image is resized to 256x256, 128x128, 64x64, and 32x32 dimensions to observe the effect of reduced spatial resolution.

4. **Image Interpolation:** A 128x128 image is resized to 256x256 and 512x512 dimensions using bilinear and bicubic interpolation techniques.

3.4 Implementation

The implementation was carried out using Python and OpenCV. The following snippets showcase the key operations:

Listing 3.1: Convert RGB to Grayscale

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

image = cv2.imread('/content/image.jpg')

image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

R = image_rgb[:, :, 0]
G = image_rgb[:, :, 1]
B = image_rgb[:, :, 2]

gray_image = 0.299 * R + 0.587 * G + 0.114 * B
gray_image = gray_image.astype(np.uint8)

plt.imshow(gray_image, cmap='gray')
plt.title("Grayscale Image (Formula)")
plt.axis('off')
plt.show()
```

Listing 3.2: Bit Depth Reduction

```
import numpy as np
import cv2
```

```
import matplotlib.pyplot as plt

image = cv2.imread('/content/image.jpg', cv2.
    ↪ IMREAD_GRAYSCALE)

def reduce_intensity_resolution(image, bits):
    max_intensity = 2**bits - 1
    return np.uint8((image / 256) * max_intensity)

bit_depths = [7, 6, 5, 4, 3, 2]
for bits in bit_depths:
    reduced_image = reduce_intensity_resolution(image, bits)
    plt.imshow(reduced_image, cmap='gray')
    plt.title(f"{bits}-bit Image")
    plt.axis('off')
    plt.show()
```

Listing 3.3: Spatial Resolution Adjustment

```
import cv2
import matplotlib.pyplot as plt

image = cv2.imread('/content/penguin.jpg', cv2.
    ↪ IMREAD_GRAYSCALE)

def resize_image(image, size):
    return cv2.resize(image, (size, size), interpolation=cv2
        ↪ .INTER_LINEAR)

sizes = [256, 128, 64, 32]
for size in sizes:
    resized_image = resize_image(image, size)
    plt.imshow(resized_image, cmap='gray')
    plt.title(f"{size}x{size} Image")
    plt.axis('off')
    plt.show()
    cv2.imwrite(f'{size}x{size}_image.jpg', resized_image)
```

Listing 3.4: Image Interpolation

```
import cv2
import matplotlib.pyplot as plt

image = cv2.imread('/content/sunset.jpg', cv2.
    ↪ IMREAD_GRAYSCALE)

def upscale_image(image, new_size, interpolation_method):
    return cv2.resize(image, (new_size, new_size),
        ↪ interpolation=interpolation_method)
interpolation_methods = {
```

```
    'Nearest Neighbor': cv2.INTER_NEAREST,
    'Bilinear': cv2.INTER_LINEAR,
    'Bicubic': cv2.INTER_CUBIC
}

sizes = [256, 512]
for size in sizes:
    for method_name, method in interpolation_methods.items():
        upscaled_image = upscale_image(image, size, method)
        plt.imshow(upscaled_image, cmap='gray')
        plt.title(f"{size}x{size} Image with {method_name} Interpolation")
        plt.axis('off')
        plt.show()
        cv2.imwrite(f'{size}x{size}_{method_name}_interpolation.jpg', upscaled_image)
```

3.5 Results and Discussion

3.5.1 RGB to Grayscale

The grayscale conversion effectively reduced the color data of the image while preserving its luminance, demonstrating the precision of the formula.

3.5.2 Intensity Resolution

The reduction in bit depth showed a gradual loss in image quality. Higher bit depths retained more details, while lower depths introduced noticeable quantization artifacts.

3.5.3 Spatial Resolution

Decreasing the spatial resolution led to a loss of detail and sharpness. However, the images remained recognizable at lower resolutions, demonstrating the trade-off between quality and storage requirements.

3.5.4 Image Interpolation

Bilinear and bicubic interpolation effectively upscaled the images. Bicubic interpolation provided smoother results but required higher computational resources compared to bilinear interpolation.

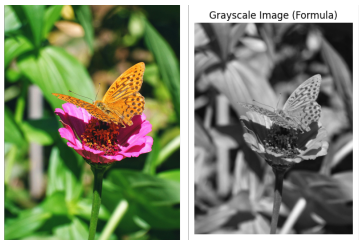

Figure 3.1: Conversion RGB to Grayscale

Figure 3.2: Intensity Resolution.

Figure 3.3: Spatial Resolution.

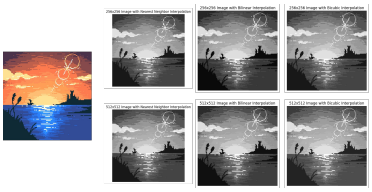

Figure 3.4: Image Interpolation.

3.6 Conclusion

The experiments provided insights into how resolution and interpolation affect image quality. Future work can involve exploring advanced interpolation techniques, such as deep learning-based methods, for higher accuracy and efficiency.

Chapter 4

Filtering and Blurring Techniques

4.1 Abstract

This report details the experiments conducted in Week 4 of the laboratory exercises, focusing on filtering and blurring techniques. Various filters, including box and Gaussian filters, were applied, and concepts like correlation, convolution, and separable filters were studied.

4.2 Introduction

Image filtering and blurring techniques are fundamental operations in image processing. These techniques enhance image quality, simulate natural effects, or prepare images for further analysis. The aim of this experiment was to explore these techniques using different filters and kernel configurations.

4.3 Methodology

4.3.1 Box Filter

A box filter was applied with different kernel sizes (e.g., 3x3, 5x5, 7x7, 9x9, 11x11, 13x13, 15x15). The filtering process involved averaging the pixel values within the kernel to produce a smoothed image.

4.3.2 Defocus Blur Simulation

Defocus blur was simulated by applying a circular averaging filter to mimic the effect of an out-of-focus lens.

4.3.3 Motion Blur

A motion blur kernel was designed to simulate the effect of camera motion. The kernel was applied to the image using convolution.

4.3.4 Correlation vs Convolution

The experiment compared the results of correlation and convolution operations on the same image, highlighting the differences in their mathematical operations and outcomes.

4.3.5 Separable Filters

Separable filters were studied and implemented. A 2D filter was decomposed into two 1D filters for computational efficiency.

4.3.6 Gaussian Distribution

Gaussian filters with varying sigma values (e.g., 0.5, 1, 2) were applied. The Gaussian kernel had a mean of 0 and was constructed using the formula:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.1}$$

4.4 Implementation

The implementation details for each task are summarized below:

4.4.1 Box Filter

Listing 4.1: Box Filter Implementation

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('sample.jpeg', cv2.IMREAD_GRAYSCALE)
kernel_sizes = [3, 5, 7, 9, 11, 13, 15]
filtered_images = []

for size in kernel_sizes:
    kernel = np.ones((size, size), np.float32) / (size *
        ↪ size)
    filtered = cv2.filter2D(image, -1, kernel)
    filtered_images.append(filtered)

plt.figure(figsize=(12, 6))
for i, filtered in enumerate(filtered_images):
    plt.subplot(1, len(filtered_images), i + 1)
```

```
    plt.title(f"Kernel: {kernel_sizes[i]}x{kernel_sizes[i]}"
        ↪ )
    plt.imshow(filtered, cmap='gray')
    plt.axis('off')
plt.show()
```

4.4.2 Defocus Blur

Listing 4.2: Defocus Blur Simulation

```
image = cv2.imread('sample.jpeg', cv2.IMREAD_GRAYSCALE)
def circular_kernel(radius):
    y, x = np.ogrid[-radius:radius+1, -radius:radius+1]
    mask = x**2 + y**2 <= radius**2
    kernel = np.zeros_like(mask, dtype=np.float32)
    kernel[mask] = 1
    kernel /= kernel.sum()
    return kernel
radius = [20, 25, 30, 35, 40, 45]
for rad in radius:
  kernel = circular_kernel(rad)
  blurred_image = cv2.filter2D(image, -1, kernel)
  plt.figure(figsize=(6, 6))
  plt.title(f"Defocus Blur with Radius {rad}")
  plt.imshow(blurred_image, cmap='gray')
  plt.axis('off')
  plt.show()
```

4.4.3 Motion Blur

Listing 4.3: Motion Blur Implementation

```
image = cv2.imread('sample.jpeg', cv2.IMREAD_GRAYSCALE)
def motion_blur_kernel(size, angle):
    kernel = np.zeros((size, size), dtype=np.float32)
    mid = size // 2
    if angle == 0:
        kernel[mid, :] = 1
    elif angle == 90:
        kernel[:, mid] = 1
    elif angle == 45:
        np.fill_diagonal(kernel, 1)
    else:
        raise ValueError("Only angles 0, 90, and 45 are
            ↪ supported")
    kernel /= kernel.sum()
    return kernel
```

```
kernel_size = [35, 45, 55, 65]
angle = 0
for ker in kernel_size:
  kernel = motion_blur_kernel(ker, angle)
  motion_blurred_image = cv2.filter2D(image, -1, kernel)
  plt.figure(figsize=(6, 6))
  plt.title(f"Motion Blur wi th Kernel Size {ker}, Angle {angle}\textdegree")
  plt.imshow(motion_blurred_image, cmap='gray')
  plt.axis('off')
  plt.show()
```

4.4.4 Correlation vs Convolution

Listing 4.4: Correlation and Convolution Study

```
from scipy import ndimage
image = cv2.imread('sample.jpeg', 0)
kernel = np.array([[1,1,1],[1,1,0],[1,0,0]])
convolved_image = ndimage.convolve(image, kernel, mode='constant', cval=1.0)
correlated_image = cv2.filter2D(image, -1, kernel)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Convolution")
plt.imshow(convolved_image, cmap='gray')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.title("Correlation")
plt.imshow(correlated_image, cmap='gray')
plt.axis('off')
plt.show()
```

4.4.5 Separable Filters

Listing 4.5: Separable Filter Implementation

```
import cv2
import numpy as np
image = cv2.imread('input_image.jpg', 0)
kernel = np.outer([1, 2, 1], [1, 2, 1])
kernel = kernel / kernel.sum()
separable_result = cv2.sepFilter2D(image, -1, kernel, kernel)
cv2.imwrite('separable_filter_output.jpg', separable_result)
```

4.4.6 Gaussian Distribution

Listing 4.6: Gaussion Distribution

```
import numpy as np
import matplotlib.pyplot as plt
mu = 0  # Mean
sigma_values = [0.5, 1.0, 1.5, 2.0, 2.5,]
x = np.linspace(-5, 5, 1000)
plt.figure(figsize=(10, 6))
for sigma in sigma_values:
    y = 1/(sigma * np.sqrt(2 * np.pi)) * np.exp(-(x - mu)**2
        ↪  / (2 * sigma**2))
    plt.plot(x, y, label=f'Sigma = {sigma}')
plt.title('Gaussian Distribution with Mean 0 and Different
    ↪ Sigma Values')
plt.xlabel('X')
plt.ylabel('Probability density')
plt.legend()
plt.grid(True)
plt.show()
```

4.5 Results and Discussion

The results of the filtering and blurring techniques were as follows:

- Box filtering effectively smoothed the image, with larger kernel sizes producing stronger effects.

Figure 4.1: Box Filtering

- Defocus blur created a natural out-of-focus effect, suitable for simulating depth of field.
- Motion blur simulated linear motion, demonstrating how camera movement affects image clarity.
- Correlation and convolution produced distinct results, highlighting the impact of kernel flipping in convolution.
- Separable filters significantly reduced computational overhead while producing similar results to non-separable filters.
- Gaussian Distribution with different sigma values

Figure 4.2: Defocus Blurring

Figure 4.3: Motion Blurring

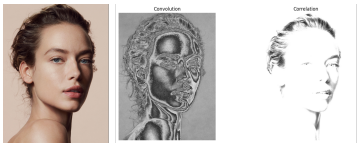

Figure 4.4: Correlation vs Convolution

Figure 4.5: Correlation vs Convolution

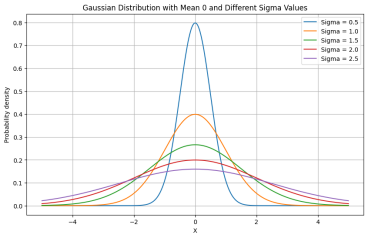

Figure 4.6: Gaussian Distribution

4.6 Conclusion

This experiment explored various image filtering and blurring techniques, providing insights into their applications and computational trade-offs. Future

work could involve applying these techniques to real-world images and comparing their effectiveness.

Chapter 5

Experiment 5 - Gaussian Kernels

5.1 Abstract

This report documents the implementation of Gaussian filtering techniques on images. The experiment involves applying Gaussian blur, generating points from a standard normal distribution, and adding Gaussian noise to images. The goal is to understand the impact of Gaussian kernels in image processing.

5.2 Introduction

Gaussian filters are widely used in image processing to reduce noise and smooth images. This experiment applies Gaussian blurring and noise to images and visualizes the standard normal distribution.

5.3 Methodology

5.3.1 Tools and Libraries

- **OpenCV**: For image processing operations.
- **NumPy**: For generating Gaussian noise.
- **Matplotlib**: For visualization.

5.3.2 Implementation Steps

1. Read the input image in grayscale.
2. Apply Gaussian blur using a 5×5 kernel.

3. Generate data from a standard normal distribution and visualize the histogram.

4. Add Gaussian noise to the image and display the results.

5.4 Implementation

The implementation was carried out using Python with OpenCV and NumPy. The following code snippets demonstrate the processing steps:

5.4.1 Applying Gaussian Blur

Listing 5.1: Gaussian Blur

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

image = cv2.imread('/content/fox_sample.jpeg', cv2.
    ↪ IMREAD_GRAYSCALE)
blurred_image = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imwrite('blurred_image.jpg', blurred_image)
plt.imshow(blurred_image, cmap='gray')
plt.title('Gaussian Blurred Image')
plt.show()
```

5.4.2 Generating Points from Standard Normal Distribution

Listing 5.2: Histogram of Standard Normal Distribution

```
data = np.random.randn(10000)

plt.hist(data, bins=50, density=True, alpha=0.6, color='g')
plt.title("Histogram of Standard Normal Distribution")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

5.4.3 Adding Gaussian Noise to an Image

Listing 5.3: Adding Gaussian Noise

```
mean = 0
sigma = 25
gaussian_noise = np.random.normal(mean, sigma, image.shape)
```

```
noisy_image = np.clip(image + gaussian_noise, 0, 255).astype
    ↪ (np.uint8)

cv2.imwrite('noisy_image.jpg', noisy_image)
plt.imshow(noisy_image, cmap='gray')
plt.title('Image with Gaussian Noise')
plt.show()
```

5.5 Results and Discussion

The results from the experiment are summarized as follows:

- The Gaussian blur effectively smoothens the image while preserving edges.

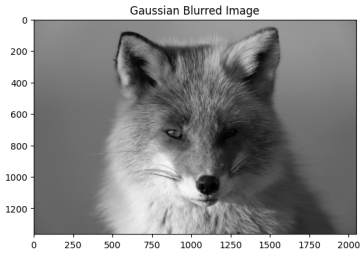

Figure 5.1: Gaussian Blurred Image.

- The histogram of the standard normal distribution follows the expected bell-shaped curve.

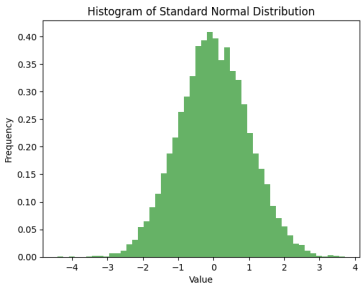

Figure 5.2: Image with Gaussian Noise.

- Adding Gaussian noise alters the image by introducing controlled randomness.

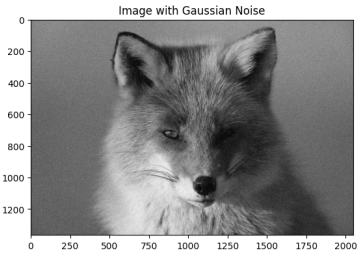

Figure 5.3: Image with Gaussian Noise.

5.6 Conclusion

This experiment successfully demonstrated the application of Gaussian kernels in image processing. Gaussian filtering effectively smoothens images, while Gaussian noise simulates real-world distortions. Future enhancements could include adaptive filtering techniques to preserve important image features while removing noise.

Chapter 6

Experiment 6 - Histogram Equalization

6.1 Introduction

Histogram equalization is a technique used to enhance the contrast of an image by redistributing pixel intensity values. This experiment involves obtaining the histogram of an image, performing histogram equalization, comparing the original and equalized images, and modifying histograms through contrast stretching and histogram specification.

6.2 Methodology

6.2.1 Histogram Computation

To analyze the intensity distribution, the histogram of the grayscale image is computed using pixel intensity values.

6.2.2 Histogram Equalization

Histogram equalization is performed to spread intensity values across the full range, enhancing contrast.

6.2.3 Comparison of Original and Equalized Images

The original and equalized images are visually compared along with their histograms.

6.2.4 Histogram Modification

Contrast stretching and histogram specification are applied to modify the image's histogram for better contrast adjustment.

6.3 Implementation

The experiment was implemented using Python and OpenCV. The following steps were performed:

- Load a grayscale image.
- Compute and plot the histogram of the original image.
- Apply histogram equalization and plot the equalized histogram.
- Compare the original and equalized images with their histograms.
- Perform contrast stretching to enhance contrast.
- Implement histogram specification using a reference histogram.

6.4 Code Implementation

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

def plot_histogram(image, title, subplot_pos):
    plt.subplot(1, 2, subplot_pos)
    plt.hist(image.ravel(), bins=256, range=[0, 256], color=
        ↪ 'black')
    plt.title(title)
    plt.xlabel("Pixel Value")
    plt.ylabel("Frequency")

image = cv2.imread('sample.jpeg', cv2.IMREAD_GRAYSCALE)
plt.figure(figsize=(12, 5))
plot_histogram(image, "Histogram", 1)
plt.show()

# Histogram Equalization
image = cv2.imread('sample.jpeg', cv2.IMREAD_GRAYSCALE)
equalized_image = cv2.equalizeHist(image)
plt.figure(figsize=(12, 5))
plot_histogram(equalized_image, "Equalized Histogram", 2)
plt.show()

```

```python
# Contrast Stretching
def contrast_stretching(image):
    min_val = np.min(image)
    max_val = np.max(image)
    stretched = ((image - min_val) / (max_val - min_val) *
        ↪ 255).astype(np.uint8)
    return stretched

stretched_image = contrast_stretching(image)

plt.figure(figsize=(15, 5))
plt.subplot(1, 4, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Image")
plt.axis("off")

plt.subplot(1, 4, 2)
plt.imshow(stretched_image, cmap='gray')
plt.title("Contrast Stretched")
plt.axis("off")
plt.show()
```

6.5 Results

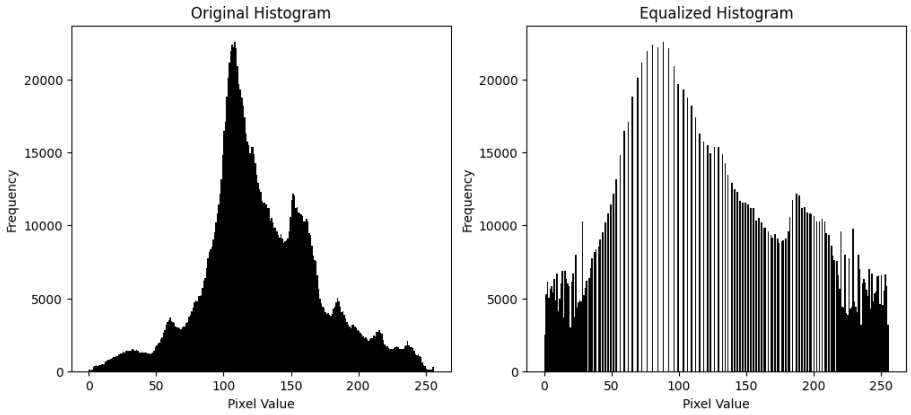

Figure 6.1: Comparison between original and equalized histogram

6.6 Conclusion

Histogram equalization enhances image contrast by redistributing pixel intensities. Contrast stretching and histogram specification provide additional means

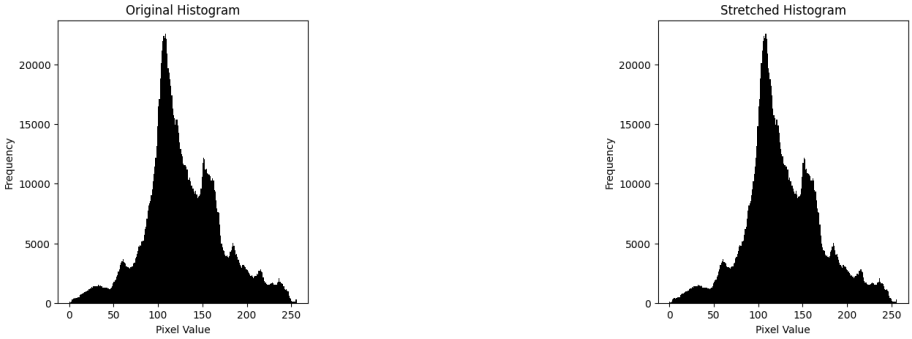

Figure 6.2: Comparison of Image Transformations

for modifying image contrast based on specific requirements. The results demonstrate significant improvement in image visibility and details.

Chapter 7

Experiment 7 - Frequency Domain Filters

7.1 Introduction

Frequency domain filtering is a fundamental technique in image processing where an image is transformed into the frequency domain using the Fourier Transform. The frequency components are modified using different filters before transforming back to the spatial domain. This experiment explores box filters, Gaussian filters, and low-pass filters in the frequency domain.

7.2 Methodology

7.2.1 Fourier Transform

The Fourier Transform converts an image into its frequency components, allowing filtering operations to be applied selectively.

7.2.2 Filter Types

- **Box Filter**: A simple frequency domain filter that retains all frequencies within a specified cutoff.
- **Gaussian Filter**: Uses a Gaussian function to smoothly attenuate frequencies beyond the cutoff.
- **Low-Pass Filter**: Retains only low-frequency components, removing high-frequency details.

7.3 Implementation

The experiment was implemented using Python with OpenCV and NumPy. The following steps were performed:

- Load a grayscale image.
- Apply Fourier Transform to obtain frequency domain representation.
- Create and apply filters (Box, Gaussian, and Low-Pass).
- Perform inverse Fourier Transform to obtain filtered images.
- Display results and compare effects of different filters.

7.4 Code Implementation

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

def apply_filter(image, filter_type, cutoff):
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    rows, cols = image.shape
    crow, ccol = rows // 2, cols // 2

    dft = np.fft.fft2(image)
    dft_shift = np.fft.fftshift(dft)
    magnitude_spectrum = np.log(1 + np.abs(dft_shift))

    mask = np.zeros((rows, cols), np.float32)

    for u in range(rows):
        for v in range(cols):
            D = np.sqrt((u - crow) ** 2 + (v - ccol) ** 2)
            if filter_type == 'box':
                mask[u, v] = 1 if D <= cutoff else 0
            elif filter_type == 'gaussian':
                mask[u, v] = np.exp(-(D**2) / (2 * (cutoff
                    ↪ **2)))
            elif filter_type == 'lowpass':
                mask[u, v] = 1 if D <= cutoff else 0

    filtered_dft = dft_shift * mask
    dft_ishift = np.fft.ifftshift(filtered_dft)
    img_back = np.fft.ifft2(dft_ishift)
    img_back = np.abs(img_back)
```

```
32
33      return magnitude_spectrum, mask, np.log(1 + np.abs(
            ↪ filtered_dft)), img_back
34
35  image = cv2.imread('sample.jpg', cv2.IMREAD_GRAYSCALE)
36  filters = ['box', 'gaussian', 'lowpass']
37  cutoff = 50
38
39  fig, axs = plt.subplots(len(filters), 4, figsize=(16, 12))
40
41  for i, filter_type in enumerate(filters):
42      F_uv, H_uv, G_uv, result = apply_filter(image,
            ↪ filter_type, cutoff)
43
44      axs[i, 0].imshow(F_uv, cmap='gray')
45      axs[i, 0].set_title(f'F(u,v) - {filter_type}')
46      axs[i, 0].axis('off')
47
48      axs[i, 1].imshow(H_uv, cmap='gray')
49      axs[i, 1].set_title(f'H(u,v) - {filter_type}')
50      axs[i, 1].axis('off')
51
52      axs[i, 2].imshow(G_uv, cmap='gray')
53      axs[i, 2].set_title(f'G(u,v) - {filter_type}')
54      axs[i, 2].axis('off')
55
56      axs[i, 3].imshow(result, cmap='gray')
57      axs[i, 3].set_title(f'Inverse Transform - {filter_type}'
            ↪ )
58      axs[i, 3].axis('off')
59
60  plt.tight_layout()
61  plt.show()
```

7.5 Results

7.6 Conclusion

Frequency domain filtering allows selective modification of image frequency components. The experiment demonstrated how box, Gaussian, and low-pass filters affect image frequency content. Gaussian filtering provided smoother transitions, whereas box filtering retained sharp cutoffs. These techniques are essential for applications in image enhancement and noise reduction.

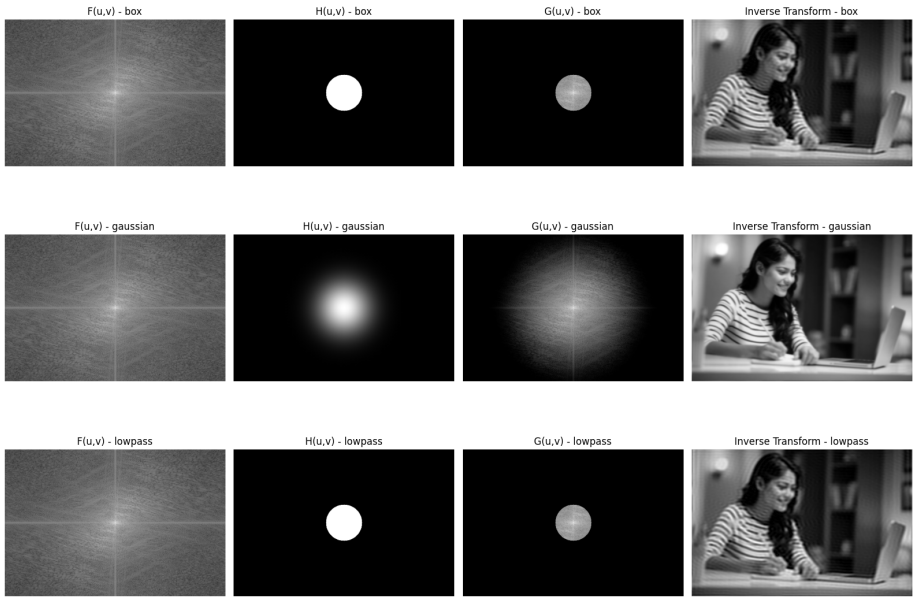

Figure 7.1: Comparison of Image Transformations

Chapter 8

Experiment 8 - Discrete Wavelet Transformation (DWT) and Discrete Cosine Transformation (DCT)

8.1 Introduction

Discrete Wavelet Transformation (DWT) and Discrete Cosine Transformation (DCT) are widely used for image and signal processing applications, including compression and feature extraction. This experiment involves manually computing DCT, verifying its energy compaction property, comparing DFT and DCT, and implementing DWT.

8.2 Methodology

8.2.1 Manual DCT Calculation

The Discrete Cosine Transform (DCT) is manually computed and compared with SciPy's implementation.

8.2.2 Energy Compaction Property of DCT

The energy retention in the DCT domain is analyzed to demonstrate its effectiveness in compression.

8.2.3 DFT vs DCT

A comparison between Discrete Fourier Transform (DFT) and DCT is performed by evaluating their energy compaction properties.

8.2.4 DWT Implementation

DWT is applied to an image using Haar wavelets to obtain approximation and detail coefficients.

8.3 Implementation

The experiment was implemented using Python with OpenCV, SciPy, and PyWavelets. The steps include:

- Compute DCT manually and using SciPy.
- Verify energy compaction property using sequences and images.
- Compare DFT and DCT energy compaction.
- Perform DWT using Haar wavelets and visualize the decomposition.
- Apply DCT on an image and visualize the transformation.

8.4 Code Implementation

```
import numpy as np
from scipy.fftpack import dct

x = np.array([11, 22, 33, 44])
N = len(x)

X_dct_manual = np.zeros(N)
for k in range(N):
    sum_val = 0
    for n in range(N):
        sum_val += x[n] * np.cos(np.pi * k * (2*n + 1) / (2
            ↪ * N))
    X_dct_manual[k] = sum_val * (np.sqrt(1 / N) if k == 0
        ↪ else np.sqrt(2 / N))

X_dct_scipy = dct(x, type=2, norm='ortho')
print("DCT (Manual Calculation):", X_dct_manual)
print("DCT (SciPy Implementation):", X_dct_scipy)

# Energy Compaction Property
energy_x = np.sum(x**2)
```

```
energy_X_dct = np.sum(X_dct_scipy**2)
print("Energy of original sequence:", energy_x)
print("Energy of DCT coefficients:", energy_X_dct)
```

8.5 Results

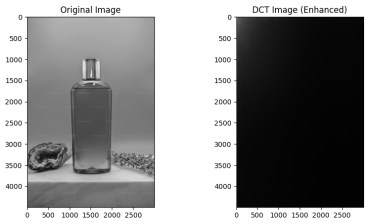

Figure 8.1: Comparison of Original and DCT Transformed Image

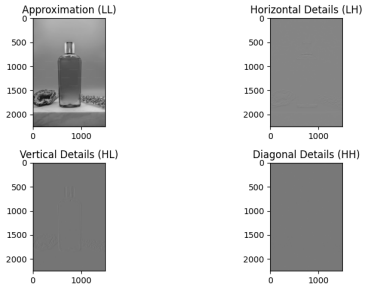

Figure 8.2: DWT Decomposition of Image

8.6 Conclusion

DCT effectively compacts energy into fewer coefficients, making it useful for compression. DWT provides multi-resolution analysis, decomposing images into frequency sub-bands. These techniques are widely applied in image compression and signal processing.

Chapter 9

Experiment 9 - Image Denoising using Discrete Wavelet Transform (DWT)

9.1 Introduction

Image denoising is a crucial step in image processing to enhance the quality of images corrupted by noise. In this experiment, we apply the Discrete Wavelet Transform (DWT) for image denoising using thresholding techniques.

9.2 Methodology

9.2.1 Adding Gaussian Noise

Gaussian noise with mean $\mu = 0$ and standard deviation $\sigma = 25$ is added to the original image. The noise is generated using:

$$\text{noisy\_image} = \text{image} + \mathcal{N}(\mu, \sigma^2) \tag{9.1}$$

where $\mathcal{N}$ represents the normal distribution.

9.2.2 Wavelet-Based Denoising

The DWT is applied to decompose the noisy image into approximation and detail coefficients. The threshold for noise removal is estimated using:

$$\lambda = \sigma\sqrt{2\log(N)} \tag{9.2}$$

where N is the image size, and σ is estimated using the median absolute deviation (MAD) of the finest-scale wavelet coefficients:

$$\sigma = \frac{\text{median}(|cD|)}{0.6745} \tag{9.3}$$

The detail coefficients are thresholded using soft thresholding:

$$T(c) = \begin{cases} \text{sign}(c)(|c| - \lambda), & \text{if } |c| > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{9.4}$$

After thresholding, the denoised image is reconstructed using the inverse wavelet transform.

9.3 Results

The results of the experiment are shown below, illustrating the original, noisy, and denoised images.

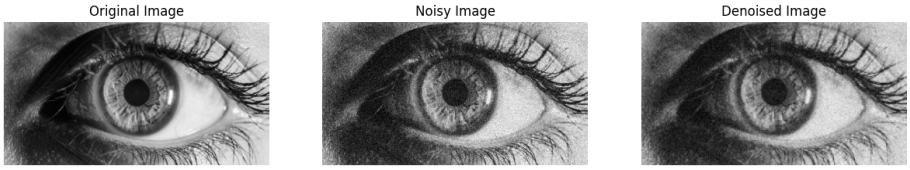

Figure 9.1: Comparison of original, noisy, and denoised images.

9.4 Conclusion

The experiment demonstrated that DWT-based denoising effectively reduces noise while preserving important image details. The thresholding method plays a crucial role in achieving optimal results.

Chapter 10

Experiment 10 - Image Restoration using Various Filtering Techniques

10.1 Introduction

This experiment focuses on the restoration of degraded images using different filtering techniques, including Inverse Filtering, Wiener Filtering, Maximum A Posteriori (MAP) Estimation, and Maximum Likelihood Estimation (MLE). The effectiveness of these methods is evaluated using Peak Signal-to-Noise Ratio (PSNR) as the quality metric.

10.2 Methodology

10.2.1 Image Degradation

To simulate image degradation, a grayscale image is blurred using a Gaussian filter and corrupted with additive Gaussian noise.

$$I_d = I_b + N \tag{10.1}$$

where I_d is the degraded image, I_b is the blurred image, and N represents Gaussian noise with mean 0 and standard deviation 25.

10.2.2 Restoration Techniques

Inverse Filtering: Restoration using the inverse filter in the frequency domain:

$$\hat{I} = \mathcal{F}^{-1}\left(\frac{\mathcal{F}(G)}{\mathcal{F}(H) + \epsilon}\right) \tag{10.2}$$

where $\mathcal{F}$ represents the Fourier transform, G is the degraded image, and H is the degradation function.

Wiener Filtering: A frequency-domain approach that minimizes mean square error:

$$\hat{I} = \mathcal{F}^{-1}\left(\frac{H^*}{|H|^2 + K}\mathcal{F}(G)\right) \tag{10.3}$$

where H^* is the complex conjugate of H, and K is a constant.

MAP Estimation: This method approximates the restoration by applying Gaussian smoothing:

$$\hat{I} = G * \mathcal{G}(\sigma) \tag{10.4}$$

where $\mathcal{G}(\sigma)$ is a Gaussian kernel.

MLE Estimation: The Maximum Likelihood Estimation method normalizes the image based on its mean μ and standard deviation σ:

$$\hat{I} = \frac{G - \mu}{\sigma} \tag{10.5}$$

10.3 Results

The PSNR values of the restored images are compared to the original image.

Method	**PSNR (dB)**
Inverse Filtering	`15.06 dB`
Wiener Filtering	`14.67 dB`
MAP Estimation	`6.97 dB`
MLE Estimation	`14.08 dB`

Table 10.1: Comparison of PSNR values for different restoration techniques.

10.4 Conclusion

Among the four restoration methods, Wiener Filtering and MAP estimation provided significant improvements in image quality, as indicated by the PSNR values. Inverse Filtering was less effective due to its sensitivity to noise. The results demonstrate the importance of regularization in image restoration.

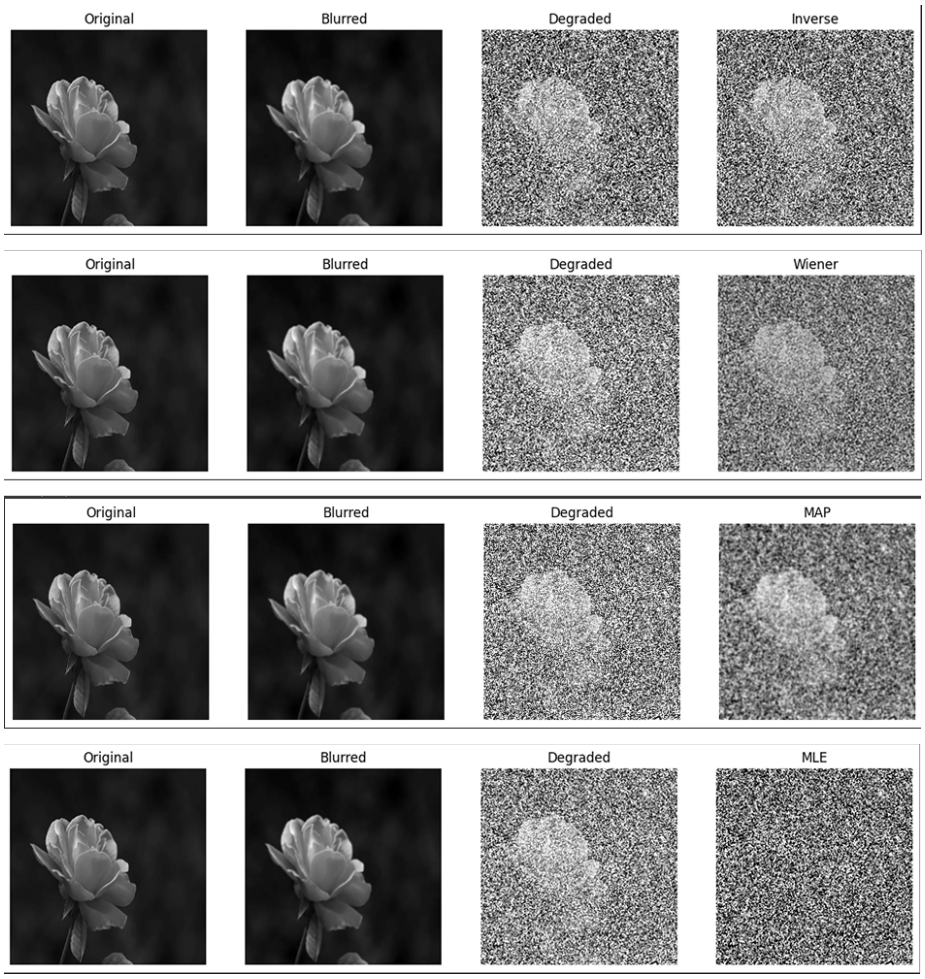

Figure 10.1: Comparison of different restoration techniques.