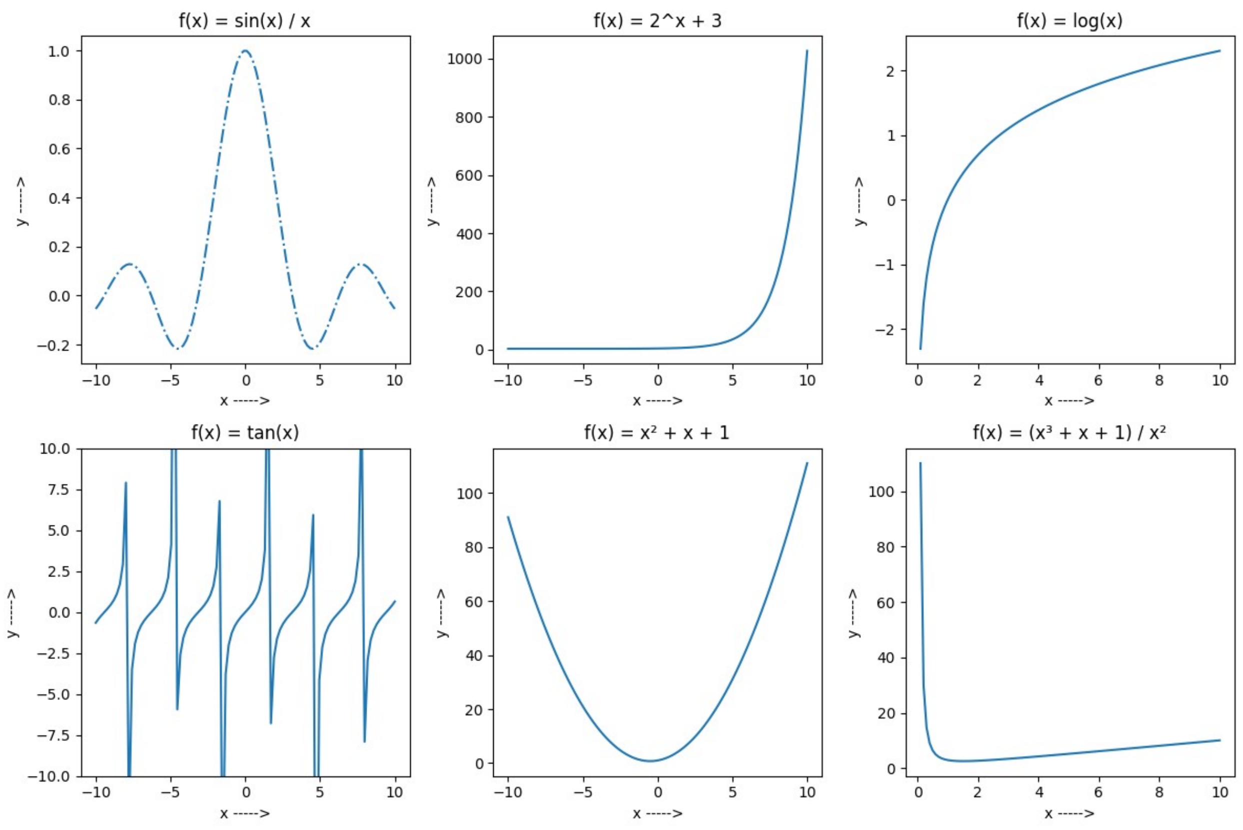
Experiment 1

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 1
Experiment Title: To Study Various Signals in Python

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

x = np.linspace(-10, 10, 100)

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

y1 = np.sin(x) / x
plt.subplot(2, 3, 1)
plt.title('f(x) = sin(x) / x')
plt.xlabel('x ----->')
plt.ylabel('y ----->')
plt.plot(x, y1, '-.')

y2 = 2**x + 3
plt.subplot(2, 3, 2)
plt.title('f(x) = 2^x + 3')
plt.xlabel('x ----->')
plt.ylabel('y ----->')
plt.plot(x, y2)

x_log = np.linspace(0.1, 10, 100)
y3 = np.log(x_log)
plt.subplot(2, 3, 3)
plt.title('f(x) = log(x)')
plt.xlabel('x ----->')
plt.ylabel('y ----->')
plt.plot(x_log, y3)

y4 = np.tan(x)
plt.subplot(2, 3, 4)
plt.title('f(x) = tan(x)')
plt.xlabel('x ----->')
plt.ylabel('y ----->')
plt.plot(x, y4)
plt.ylim(-10, 10)

y5 = x**2 + x + 1
plt.subplot(2, 3, 5)
plt.title('f(x) = x² + x + 1')
plt.xlabel('x ----->')
plt.ylabel('y ----->')
plt.plot(x, y5)

x_rat = np.linspace(0.1, 10, 100)
y6 = (x_rat**3 + x_rat + 1) / (x_rat**2)
plt.subplot(2, 3, 6)
plt.title('f(x) = (x³ + x + 1) / x²')
plt.xlabel('x ----->')
plt.ylabel('y ----->')
plt.plot(x_rat, y6)

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



Conclusion & Result

- Various mathematical signals were successfully generated and visualized.
- The experiment helps understand signal behavior, growth, periodicity, and discontinuities.

Experiment 2

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 2
Experiment Title: Negation of an image.

Theory

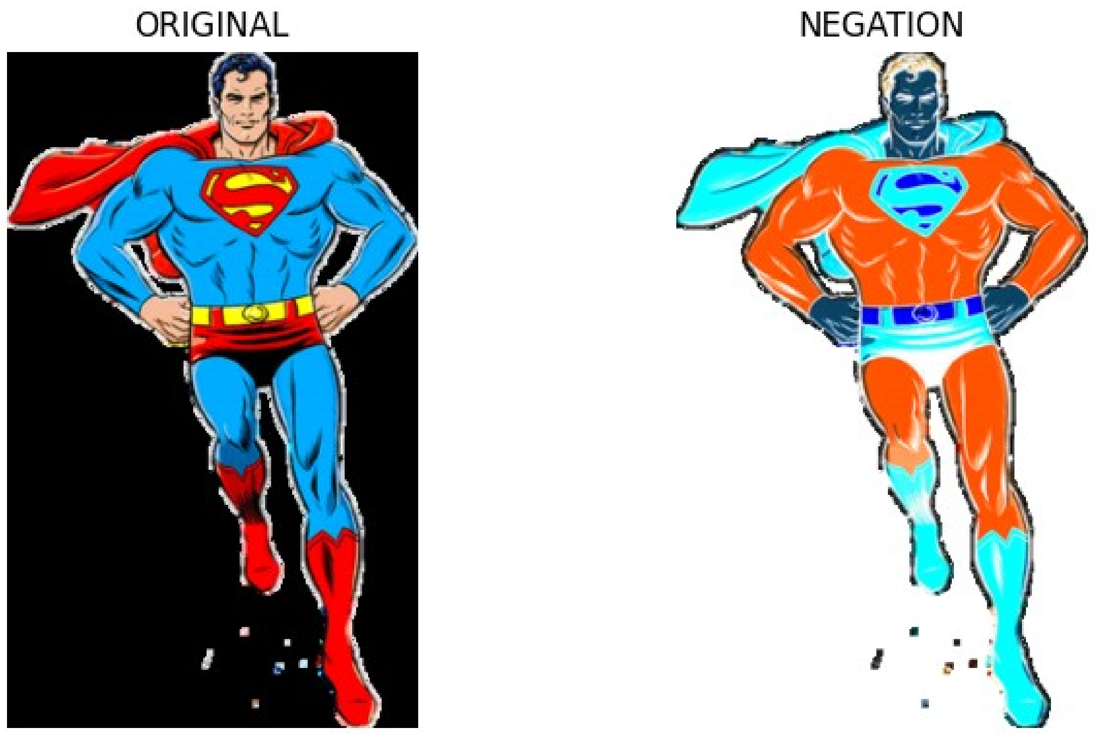
The negative of an image with gray levels in the range ([0, L-1]) is obtained using the negative transformation given by:

[S = L - 1 - r]

This transformation inverts the intensity levels of the image. As the intensity of the input image increases, the intensity of the output image decreases.

Image negation is particularly useful for enhancing white or gray details embedded in dark regions of an image, especially when black areas dominate the scene.

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

img = cv2.imread("test.png")

if img is None:
    raise FileNotFoundError("Image not found. Check file name or
path.")

img = np.array(img)

L = 256
negative = L - 1 - img

plt.figure(figsize=(8, 4))

plt.subplot(1, 2, 1)
plt.title("ORIGINAL")
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis("off")

plt.subplot(1, 2, 2)
plt.title("NEGATION")
plt.imshow(cv2.cvtColor(negative, cv2.COLOR_BGR2RGB))
plt.axis("off")

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



Conclusion & Result

- The negative of the given image was successfully generated using the image negation transformation.
- The experiment demonstrates how intensity inversion enhances light details in dark regions.
- The original and negated images were obtained and visualized as expected.

Experiment 3

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 3
Experiment Title: Thresholding of image .

Theory

Thresholding is a simple image processing technique used to separate an object of interest from the background. The output of thresholding is a binary image.

The thresholding operation is defined as:

[s = \begin{cases} 0, & \text{if } r \le t \ L - 1, & \text{if } r > t \end{cases}]

where (r) is the pixel intensity of the input image, (t) is the threshold value, and (L) is the maximum gray level.

Thresholding is widely used in image segmentation to convert grayscale images into binary images, making object detection easier.

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

img = cv2.imread("test.png", cv2.IMREAD_GRAYSCALE)

img = np.array(img)

plt.subplot(1, 2, 1)
plt.title("GRAYSCALE IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(244.5), np.float64(402.5), np.float64(-
0.5))
```

GRAYSCALE IMAGE

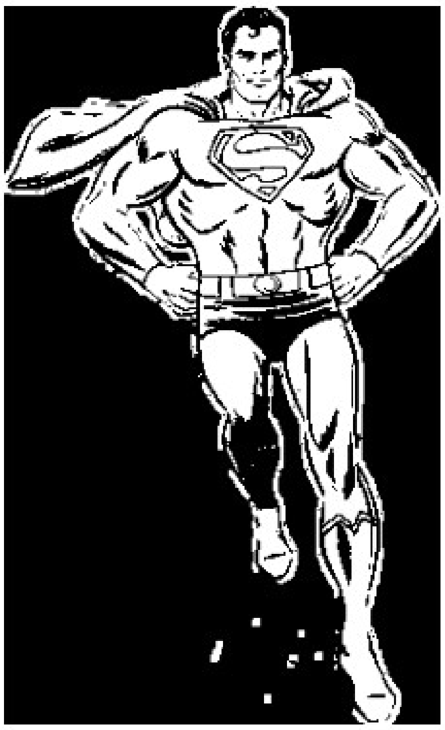
Compute Threshold Value

```
maximum = img.max()
minimum = img.min()
threshold = (maximum + minimum) / 2
```

Apply Thresholding

```
binary = np.zeros(img.shape, dtype=np.uint8)

for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        if img[i][j] >= threshold:
            binary[i][j] = 255
        else:
            binary[i][j] = 0

plt.subplot(1, 2, 2)
plt.title("BINARY IMAGE")
plt.imshow(binary, cmap="gray")
plt.axis("off")
plt.show()
```

BINARY IMAGE



Conclusion & Result

- Thresholding was successfully applied to convert the grayscale image into a binary image.
- The experiment demonstrates how thresholding separates the object from the background based on intensity values.
- The grayscale and binary images were obtained and displayed as expected.

Experiment 4

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 4
Experiment Title: Contrast Stretching of an Image.

Theory

Low contrast images may result from poor illumination conditions, lack of dynamic range in the imaging sensor, or improper exposure. The purpose of contrast stretching is to increase the dynamic range of gray levels in the image, thereby improving its visual appearance.

The contrast stretching transformation function is defined as:

[T(r) = \begin{cases} \alpha \cdot r, & r \le r_1 \ \beta \cdot (r - r_1) + s_1, & r_1 < r \le r_2 \ \ gamma \cdot (r - r_2) + s_2, & r_2 < r \le L - 1 \end{cases}]

where (r) is the input pixel intensity and (T(r)) is the output pixel intensity.
The points ((r_1, s_1)) and ((r_2, s_2)) control the shape of the transformation function.

Contrast stretching improves the visibility of features by expanding the range of intensity values in the image.

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

img = cv2.imread("bird.jpeg", cv2.IMREAD_GRAYSCALE)
plt.subplot(1, 2, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(274.5), np.float64(182.5), np.float64(-
0.5))
```

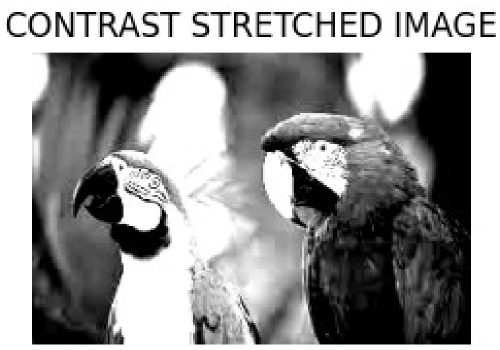
ORIGINAL IMAGE

```
rows, cols = img.shape
contrastedImg = np.zeros((rows, cols))
r1 = int(input("Enter r1: "))
r2 = int(input("Enter r2: "))
s1 = int(input("Enter s1: "))
s2 = int(input("Enter s2: "))

# r1 = 70
# r2 = 140
# s1 = 0
# s2 = 255

alpha = s1 / r1
beta = (s2 - s1) / (r2 - r1)
gamma = (255 - s2) / (255 - r2)

print(alpha, beta, gamma)
```

```
0.0 3.642857142857143 0.0
```

Apply Contrast Stretching

```
for i in range(rows):
    for j in range(cols):
        if img[i][j] <= r1:
            contrastedImg[i][j] = alpha * img[i][j]
        elif img[i][j] <= r2:
            contrastedImg[i][j] = beta * (img[i][j] - r1) + s1
        else:
            contrastedImg[i][j] = gamma * (img[i][j] - r2) + s2

plt.subplot(1, 2, 2)
plt.title("CONTRAST STRETCHED IMAGE")
plt.imshow(contrastedImg, cmap="gray")
plt.axis("off")
plt.show()
```



Conclusion & Result

- Contrast stretching was successfully applied to enhance the dynamic range of the image.
- The experiment demonstrates how pixel intensity transformation improves image visibility.
- The original and contrast stretched images were obtained and displayed as expected.

Experiment 5

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 5
Experiment Title: Bit Plane Slicing.

Theory

Bit Plane Slicing is a technique used to analyze the contribution of each bit in the representation of an image. In an 8-bit grayscale image, each pixel is represented using 8 bits, forming 8 different bit planes ranging from bit plane 0 (least significant bit) to bit plane 7 (most significant bit).

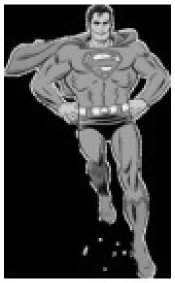
The lower-order bit planes contain finer details and noise, while higher-order bit planes contain the most significant visual information of the image.

Separating an image into its bit planes helps in understanding the relative importance of each bit and is useful in image compression, enhancement, and analysis.

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

img = cv2.imread("test.png", cv2.IMREAD_GRAYSCALE)
img = np.array(img)

plt.subplot(2, 5, 1)
plt.title("ORIGINAL")
plt.imshow(img, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(244.5), np.float64(402.5), np.float64(-
0.5))
```

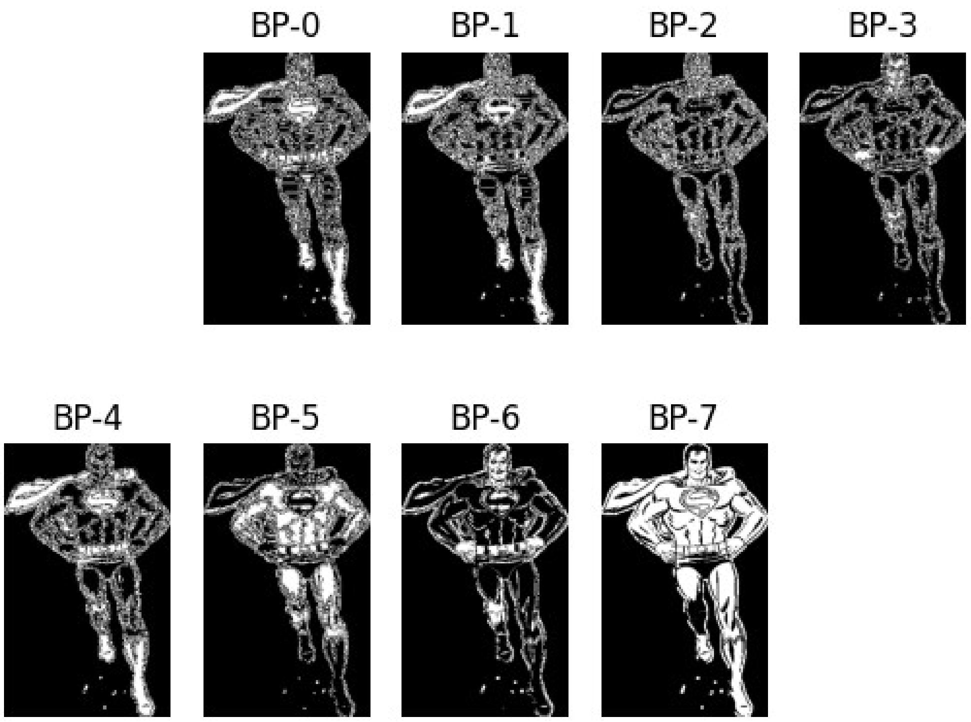
ORIGINAL



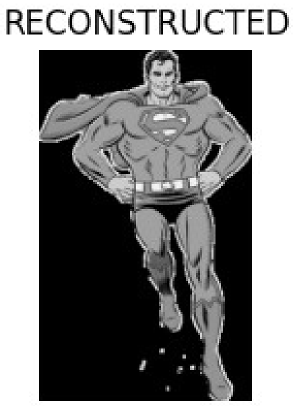
```python
# Initialize Variables
rows, cols = img.shape
finalImage = np.zeros((rows, cols))
temp = img.copy()
```

Extract and Display Bit Planes

```python
for k in range(8):
    bitPlane = np.zeros((rows, cols))

    for i in range(rows):
        for j in range(cols):
            if temp[i][j] % 2 == 1:
                bitPlane[i][j] = 255
                finalImage[i][j] += np.power(2, k)
            else:
                bitPlane[i][j] = 0

            temp[i][j] = temp[i][j] // 2

    plt.subplot(2, 5, k + 2)
    plt.title(f"BP-{k}")
    plt.imshow(bitPlane, cmap="gray")
    plt.axis("off")
```



```python
plt.subplot(2, 5, 10)
plt.title("RECONSTRUCTED")
plt.imshow(finalImage, cmap="gray")
plt.axis("off")

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



Conclusion & Result

- The given grayscale image was successfully decomposed into its individual bit planes.
- The experiment demonstrates that higher-order bit planes contain most of the significant visual information.

- The original image was successfully reconstructed from the extracted bit planes.

Experiment 6

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 6
Experiment Title: Histogram Equalization .

Theory

The histogram of a digital image with gray levels in the range ([0, L-1]) is a discrete function defined as:

[h(r_k) = n_k]

where (r_k) is the k-th gray level and (n_k) is the number of pixels having that gray level.

There are three typical cases of image histograms:

1. Dark images have histogram components concentrated on the lower gray levels.
2. Bright images have histogram components concentrated on higher gray levels.
3. Low contrast images have histogram components clustered in the middle gray levels.

Histogram equalization is a technique used to redistribute the gray levels of an image so that the histogram becomes approximately uniform. This enhances image contrast.

The transformation function used for histogram equalization is:

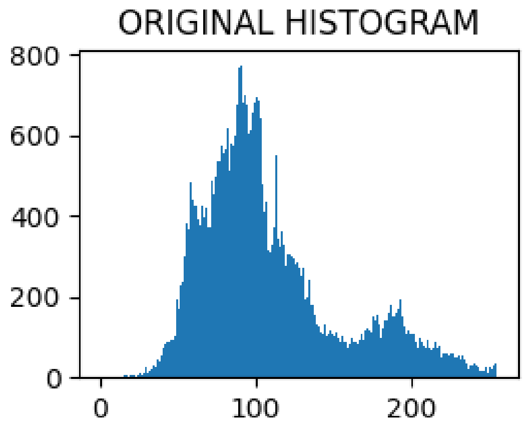
[S_k = \sum_{i=0}^{k} \frac{h(r_i)}{n}]

where (n) is the total number of pixels in the image.

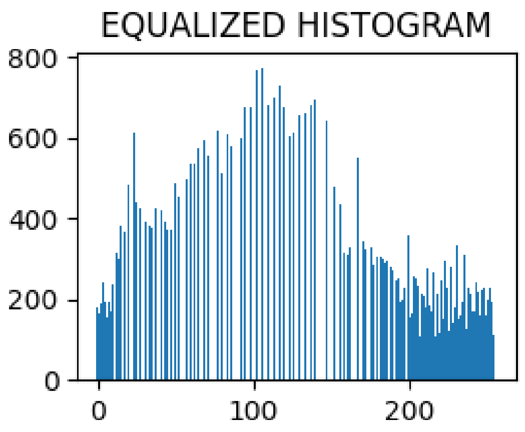
Histogram equalization improves the overall contrast of an image by spreading the most frequent intensity values.

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

img = cv2.imread("bird.jpeg", cv2.IMREAD_GRAYSCALE)
img = np.array(img)
flat = img.flatten()

hist = np.bincount(flat, minlength=256)
```

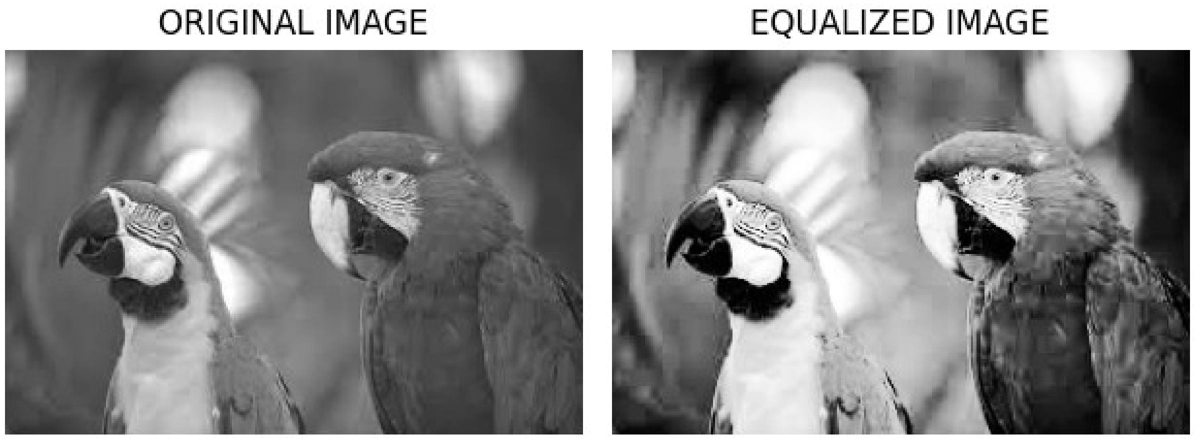
```
plt.subplot(2, 2, 1)
plt.title("ORIGINAL HISTOGRAM")
plt.bar(range(256), hist, width=1)

<BarContainer object of 256 artists>
```



```
def get_histogram(image, bins):
    histogram = np.zeros(bins)
    for pixel in image:
        histogram[pixel] += 1
    return histogram

hist = get_histogram(flat, 256)

def cumsum(a):
    a = iter(a)
    b = [next(a)]
    for i in a:
        b.append(b[-1] + i)
    return np.array(b)

cs = cumsum(hist)
nj = (cs - cs.min()) * 255
N = cs.max() - cs.min()
cs_normalized = nj / N

img_new = cs_normalized[flat]

hist_eq = np.bincount(img_new.astype(np.uint8).flatten(),
minlength=256)

plt.subplot(2, 2, 2)
plt.title("EQUALIZED HISTOGRAM")
plt.bar(range(256), hist_eq, width=1)

<BarContainer object of 256 artists>
```



```
img_new = np.reshape(img_new, img.shape)

plt.subplot(2, 2, 3)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")

plt.subplot(2, 2, 4)
plt.title("EQUALIZED IMAGE")
plt.imshow(img_new, cmap="gray")
plt.axis("off")

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



Conclusion & Result

- Histogram equalization was successfully implemented to enhance the contrast of the image.
- The experiment demonstrates how redistributing gray levels improves visibility in low-contrast images.

- The original and equalized images along with their histograms were obtained as expected.

Experiment 7

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 7
Experiment Title: Zooming by interpolation and replication.

Theory

Image zooming is a process of enlarging an image to make details more visible. Zooming can be achieved using two techniques:

1) Replication

In replication, each pixel of the image is duplicated along rows and columns. As a result, an image of size (n \times n) becomes (2n \times 2n). This method is simple but produces a blocky or patchy appearance because pixel values are directly copied.

2) Interpolation

Interpolation improves upon replication by inserting new pixel values calculated as the average of neighboring pixels. First, interpolation is applied along rows and then along columns. This method reduces the patchiness and produces a smoother zoomed image.

Zooming is widely used in image enhancement and visualization applications.

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

img = cv2.imread("bird.jpeg", cv2.IMREAD_GRAYSCALE)
img = np.array(img)

plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(274.5), np.float64(182.5), np.float64(-
0.5))
```

ORIGINAL IMAGE

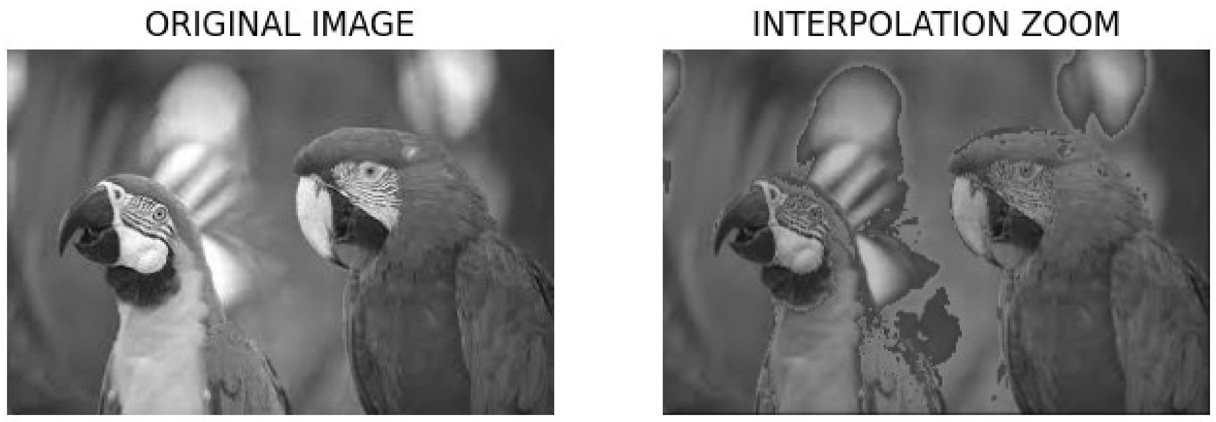
```
#Zooming by Replication
replicated = np.repeat(np.repeat(img, 2, axis=0), 2, axis=1)

plt.subplot(1, 2, 2)
plt.title("REPLICATION ZOOM")
plt.imshow(replicated, cmap="gray")
plt.axis("off")
plt.show()
```

REPLICATION ZOOM

```
img = cv2.imread("bird.jpeg", cv2.IMREAD_GRAYSCALE)
img = np.array(img)
rows, cols = img.shape
row_interp = np.zeros((rows, cols * 2))

for i in range(rows):
    for j in range(cols - 1):
        row_interp[i, 2*j] = img[i, j]
        row_interp[i, 2*j + 1] = (img[i, j] + img[i, j + 1]) / 2
    row_interp[i, -1] = img[i, -1]
```

```
/var/folders/45/2_t604ds0_15wv6b8538tj_00000gn/T/
ipykernel_34767/750808741.py:9: RuntimeWarning: overflow encountered
in scalar add
  row_interp[i, 2*j + 1] = (img[i, j] + img[i, j + 1]) / 2
```

```python
interp = np.zeros((rows * 2, cols * 2))

for i in range(rows - 1):
    interp[2*i] = row_interp[i]
    interp[2*i + 1] = (row_interp[i] + row_interp[i + 1]) / 2

interp[-1] = row_interp[-1]

plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.title("INTERPOLATION ZOOM")
plt.imshow(interp, cmap="gray")
plt.axis("off")

plt.show()
```



Conclusion & Result

- Image zooming was successfully implemented using replication and interpolation techniques.
- Replication produced a zoomed image with noticeable blocky artifacts.
- Interpolation reduced the patchiness and resulted in a smoother enlarged image.
- The results demonstrate the effectiveness of interpolation over replication for image magnification.

Experiment 8

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 8
Experiment Title: Filtering in spatial domain Low pass filtering .

Theory

Low pass filtering is a spatial domain filtering technique used to remove high-frequency components from an image. These high-frequency components generally correspond to noise and sharp intensity variations.

A low pass filter smooths the image by replacing each pixel value with the average of its neighboring pixels. A commonly used low pass filter mask is a 3 × 3 averaging filter given by:

[\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \ 1 & 1 & 1 \ 1 & 1 & 1 \end{bmatrix}]

All the coefficients of the mask are positive, which ensures smoothing of the image. The mask is placed over the image, and the weighted sum of the neighboring pixels is calculated. The center pixel of the output image is replaced by this computed value.

Low pass filtering is widely used for noise reduction and image smoothing.

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

img = cv2.imread("bird.jpeg", cv2.IMREAD_GRAYSCALE)

img = np.array(img)

plt.subplot(1, 3, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(274.5), np.float64(182.5), np.float64(-
0.5))
```

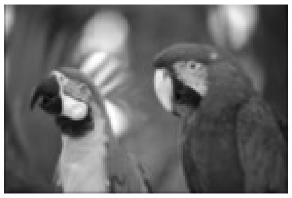
ORIGINAL IMAGE

Apply Low Pass (Mean) Filter

```
rows, cols = img.shape
padded = np.pad(img, pad_width=1, mode="constant", constant_values=0)

mean_filtered = np.zeros((rows, cols))

for i in range(rows):
    for j in range(cols):
        mean_filtered[i][j] = np.sum(padded[i:i+3, j:j+3]) / 9

plt.subplot(1, 3, 2)
plt.title("MEAN FILTER")
plt.imshow(mean_filtered, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(274.5), np.float64(182.5), np.float64(-
0.5))
```

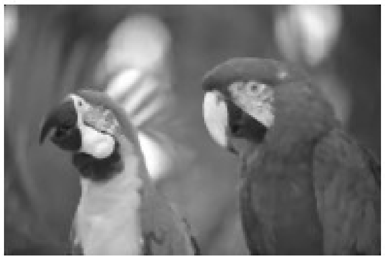
MEAN FILTER



Apply Median Filter

```
median_filtered = np.zeros((rows, cols))

for i in range(rows):
    for j in range(cols):
        median_filtered[i][j] = np.median(padded[i:i+3, j:j+3])

plt.subplot(1, 3, 3)
plt.title("MEDIAN FILTER")
plt.imshow(median_filtered, cmap="gray")
plt.axis("off")

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

MEDIAN FILTER



Conclusion & Result

- Low pass filtering was successfully applied to the given image.
- The mean filter reduced noise by smoothing the image but caused slight blurring.
- The median filter effectively reduced noise while preserving edges better than the mean filter.
- The results demonstrate the usefulness of low pass filtering for image smoothing.

Experiment 9

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 9
Experiment Title: To implement median filtering in spatial domain after adding salt and pepper noise.

Theory

Median filtering is a non-linear spatial domain filtering technique developed by Tukey and is widely used for noise reduction in images. In this method, the value of a pixel is replaced by the median value of the pixels in its neighborhood.

Salt and pepper noise is a type of impulse noise where some pixels take on very high (salt) or very low (pepper) intensity values. Median filtering is particularly effective in removing this type of noise because it ignores extreme values and preserves edges better than linear filters.

By applying median filtering after adding salt and pepper noise, the noise is significantly reduced while maintaining important image details.

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

img = cv2.imread("bird.jpeg", cv2.IMREAD_GRAYSCALE)
img = np.array(img)

plt.subplot(1, 2, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(274.5), np.float64(182.5), np.float64(-
0.5))
```

ORIGINAL IMAGE

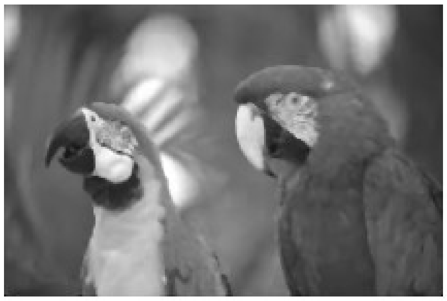
Add Salt and Pepper Noise

```
def add_sp_noise(image, prob):
    output = np.zeros(image.shape, np.uint8)
    threshold = 1 - prob

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            rdn = random.random()
            if rdn < prob:
                output[i][j] = 0
            elif rdn > threshold:
                output[i][j] = 255
            else:
                output[i][j] = image[i][j]
    return output

noisy_img = add_sp_noise(img, 0.01)
```

Apply Median Filter

```
rows, cols = noisy_img.shape
padded = np.pad(noisy_img, pad_width=1, mode="constant",
constant_values=0)

median_filtered = np.zeros((rows, cols))

for i in range(rows):
    for j in range(cols):
        median_filtered[i][j] = np.median(padded[i:i+3, j:j+3])

plt.subplot(1, 2, 2)
plt.title("MEDIAN FILTER WITH S&P NOISE")
plt.imshow(median_filtered, cmap="gray")
plt.axis("off")

plt.show()
```

MEDIAN FILTER WITH S&P NOISE



Conclusion & Result

- Salt and pepper noise was successfully added to the image.
- Median filtering effectively removed the impulse noise while preserving image edges.
- The experiment demonstrates the efficiency of median filtering for noise suppression.
- The noisy and filtered images were obtained and displayed as expected.

Experiment 10

Name: Ankit Kumar
Roll Number: 0901AM231011
Branch: AIML
Semester: 6
Subject: Image Processing Lab

Experiment Number: 10
Experiment Title: Edge Detection .

Theory

Image segmentation can be achieved using edge detection techniques. Edge detection is based on identifying discontinuities in intensity values within an image. An edge represents a boundary between two distinct regions and is characterized by a sudden change in pixel intensity.

Edge detection is broadly based on:

1. Discontinuities in intensity.
2. Similarities in intensity.

Various convolution masks are used to detect edges in different directions.

Prewitt Operator

The Prewitt operator detects edges using horizontal and vertical masks and is simple to compute.

Sobel Operator

The Sobel operator is similar to Prewitt but gives more weight to the center pixels, making it more sensitive to edges.

Laplacian Operator

The Laplacian operator is a second-order derivative operator that highlights regions of rapid intensity change in all directions.

These operators help in extracting meaningful boundaries from images for segmentation and analysis.

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

```
img = cv2.imread("bird.jpeg", cv2.IMREAD_GRAYSCALE)
img = np.array(img)

plt.subplot(1, 3, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")
```

```
(np.float64(-0.5), np.float64(274.5), np.float64(182.5), np.float64(-
0.5))
```

ORIGINAL IMAGE

initialize Output Images and Pad Input

```
rows, cols = img.shape

sobelX = np.zeros((rows, cols))
sobelY = np.zeros((rows, cols))
prewittX = np.zeros((rows, cols))
prewittY = np.zeros((rows, cols))
laplacian = np.zeros((rows, cols))

padded = np.pad(img, pad_width=1, mode="constant", constant_values=0)
```

Define Edge Detection Masks

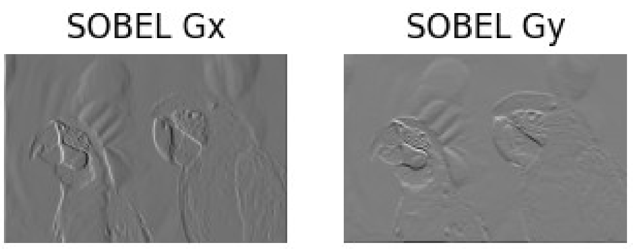
```
sobelGx = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
sobelGy = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]

prewittGx = [[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]
prewittGy = [[-1, -1, -1], [0, 0, 0], [1, 1, 1]]

laplacianG = [[0, -1, 0], [-1, 4, -1], [0, -1, 0]]
```

Apply Masks

```
for i in range(rows):
    for j in range(cols):
        region = padded[i:i+3, j:j+3]

        sobelX[i][j] = np.sum(np.multiply(sobelGx, region))
        sobelY[i][j] = np.sum(np.multiply(sobelGy, region))
        prewittX[i][j] = np.sum(np.multiply(prewittGx, region))
        prewittY[i][j] = np.sum(np.multiply(prewittGy, region))

        laplacian[i][j] = np.sum(np.multiply(laplacianG, region))
```

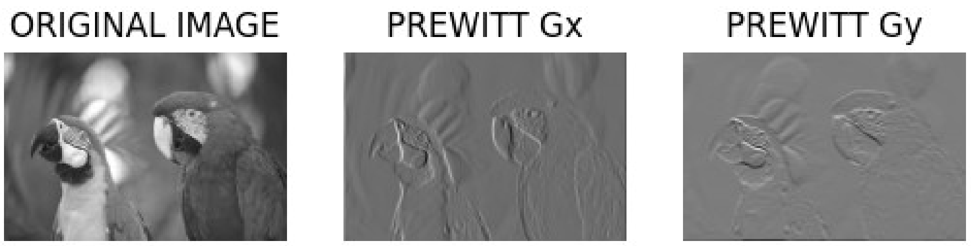
Display Sobel Edge Detection

```
plt.subplot(1, 3, 2)
plt.title("SOBEL Gx")
plt.imshow(sobelX, cmap="gray")
plt.axis("off")

plt.subplot(1, 3, 3)
plt.title("SOBEL Gy")
plt.imshow(sobelY, cmap="gray")
plt.axis("off")

plt.show()
```



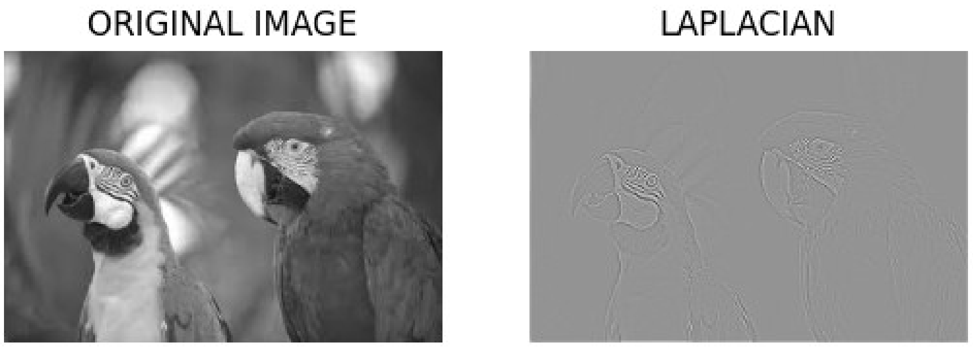
Display Prewitt Edge Detection

```
plt.subplot(1, 3, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")

plt.subplot(1, 3, 2)
plt.title("PREWITT Gx")
plt.imshow(prewittX, cmap="gray")
plt.axis("off")

plt.subplot(1, 3, 3)
plt.title("PREWITT Gy")
plt.imshow(prewittY, cmap="gray")
plt.axis("off")

plt.show()
```



Display Laplacian Edge Detection

```
plt.subplot(1, 2, 1)
plt.title("ORIGINAL IMAGE")
plt.imshow(img, cmap="gray")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.title("LAPLACIAN")
plt.imshow(laplacian, cmap="gray")
plt.axis("off")

plt.show()
```



Conclusion & Result

- Edge detection was successfully implemented using Sobel, Prewitt, and Laplacian operators.
- Sobel and Prewitt operators effectively detected horizontal and vertical edges.
- The Laplacian operator highlighted edges in all directions.
- The experiment demonstrates the importance of edge detection in image segmentation.