IMAGE ENHANCEMENT

1. Write a program to slice an gray image in different bit plane. Reconstruct the image using different bit plane
2. Perform Piecewise Linear Transformation of an gray image.
3. Perform a contrast starching using min-max and other function.
4. Perform image segmentation using thresholding

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

Bit-plane Slicing

Bit-plane slicing is a technique used in Computer Vision and Image Processing (CVIP) to extract specific bit-planes from the binary representation of an image. This technique can be used for various applications such as image compression, feature extraction, and image enhancement.

The formula to extract a specific bit-plane from an image is given by:

$B_{ij} = (I_{ij} \& 2^k) >> k$

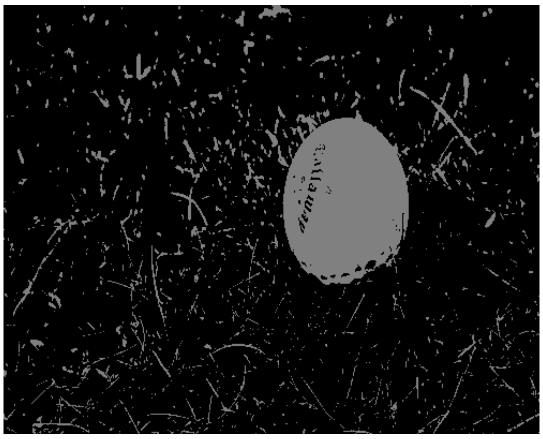
where:

- B_{ij} is the pixel value in the bit-plane at position (i, j)
- I_{ij} is the original pixel value at position (i, j)
- k is the bit-plane level (0 for the least significant bit)

In bit-plane slicing, each bit-plane represents a different level of detail in the image, with the 0th bit-plane representing the least significant bit (LSB) and higher bit-planes representing more significant bits.

By extracting and manipulating specific bit-planes, we can perform operations like image thresholding, noise removal, and edge detection to enhance the visual quality of the image or extract useful information for further processing.

```
import numpy as np
import cv2
# Read the image in greyscale
cherry = cv2.imread('/content/drive/MyDrive/Cvip_Lab/images/golf.png', cv2.IMREAD_GRAYSCALE)

#Iterate over each pixel and change pixel value to binary using np.binary_repr() and store it in a list.
lst = []
nlst = []
for i in range(cherry.shape[0]):
    for j in range(cherry.shape[1]):
      nlst.append(cherry[i][j])
      lst.append(np.binary_repr(cherry[i][j] ,width=8)) # width = no. of bits

# We have a list of strings where each string represents binary pixel value. To extract bit planes we need to iterate over the strings and store the characters corresponding to bi
# Multiply with 2^(n-1) and reshape to reconstruct the bit image.
eight_bit_img = (np.array([int(i[0]) for i in lst],dtype = np.uint8) * 128).reshape(cherry.shape[0],cherry.shape[1])
seven_bit_img = (np.array([int(i[1]) for i in lst],dtype = np.uint8) * 64).reshape(cherry.shape[0],cherry.shape[1])
six_bit_img = (np.array([int(i[2]) for i in lst],dtype = np.uint8) * 32).reshape(cherry.shape[0],cherry.shape[1])
five_bit_img = (np.array([int(i[3]) for i in lst],dtype = np.uint8) * 16).reshape(cherry.shape[0],cherry.shape[1])
four_bit_img = (np.array([int(i[4]) for i in lst],dtype = np.uint8) * 8).reshape(cherry.shape[0],cherry.shape[1])
three_bit_img = (np.array([int(i[5]) for i in lst],dtype = np.uint8) * 4).reshape(cherry.shape[0],cherry.shape[1])
two_bit_img = (np.array([int(i[6]) for i in lst],dtype = np.uint8) * 2).reshape(cherry.shape[0],cherry.shape[1])
one_bit_img = (np.array([int(i[7]) for i in lst],dtype = np.uint8) * 1).reshape(cherry.shape[0],cherry.shape[1])

#Concatenate these images for ease of display using cv2.hconcat()
finalr = cv2.hconcat([eight_bit_img,seven_bit_img,six_bit_img,five_bit_img])
finalv =cv2.hconcat([four_bit_img,three_bit_img,two_bit_img,one_bit_img])

# Vertically concatenate
final = cv2.vconcat([finalr,finalv])

# Display the images
cv2_imshow(eight_bit_img)
```



Piecewise Linear Transformation

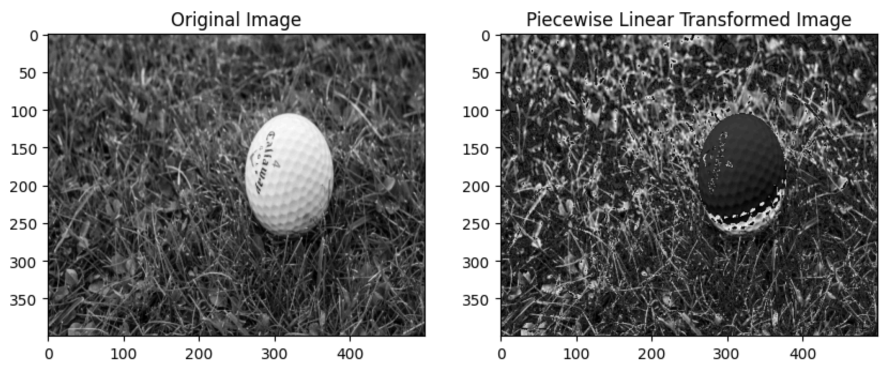
In computer vision and image processing, a Piecewise Linear Transformation is a method used to enhance or modify the contrast of an image by applying different linear functions to different regions of the image. This technique is particularly useful in adjusting the brightness and contrast of an image.

The general form of a piecewise linear transformation function can be defined as:

$$s = T(r) = \begin{cases} a_1 r + b_1 & \text{if } r < r_1 \\ a_2 r + b_2 & \text{if } r_1 \le r < r_2 \\ \vdots & \\ a_n r + b_n & \text{if } r \ge r_n \end{cases}$$

where (r) is the input pixel intensity, (s) is the output pixel intensity, and (a_i), (b_i), and (r_i) are the slope, intercept, and threshold values for each segment of the piecewise function.

```
# Define the piecewise linear transformation function
def piecewise_linear_transform(img, breakpoints, slopes):
    result = np.copy(img)
    for i in range(len(breakpoints) - 1):
        low, high = breakpoints[i], breakpoints[i + 1]
        mask = np.logical_and(img >= low, img < high)
        result[mask] = slopes[i] * (img[mask] - low)
    return np.clip(result, 0, 255).astype(np.uint8)

# Define breakpoints and slopes for the transformation
breakpoints = [0, 50, 150, 255]
slopes = [1, 2, 0.5]

# Apply the piecewise linear transformation
result_image = piecewise_linear_transform(cherry, breakpoints, slopes)

# Display the original and transformed images
plt.figure(figsize=(10, 5))

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

plt.subplot(1, 2, 2)
plt.imshow(result_image, cmap='gray')
plt.title('Piecewise Linear Transformed Image')

plt.show()
```



contrast stretching using min-max

In computer vision and image processing, contrast stretching using min-max normalization is a technique used to enhance the contrast of an image by stretching the intensity values to cover the full dynamic range. This method helps to improve the visual appearance of the image by increasing the distance between the darkest and brightest pixels.

The formula for contrast stretching using min-max normalization is given by:

$$s = \frac{s_{max} - s_{min}}{r_{max} - r_{min}}(r - r_{min}) + s_{min}$$

where (r) is the input pixel intensity, (s) is the output pixel intensity after contrast stretching, (r_{min}) and (r_{max}) are the minimum and maximum intensity values in the original image, and (s_{min}) and (s_{max}) are the desired minimum and maximum intensity values for the stretched image.

```
img1 = cv2.imread('/content/drive/MyDrive/Cvip_Lab/images/golf.png',0)

# Create zeros array to store the stretched image
minmax_img = np.zeros((img1.shape[0],img1.shape[1]),dtype = 'uint8')

# Loop over the image and apply Min-Max formulae
for i in range(img1.shape[0]):
    for j in range(img1.shape[1]):
        minmax_img[i,j] = 255*(img1[i,j]-np.min(img1))/(np.max(img1)-np.min(img1))

# Displat the stretched image
cv2_imshow(minmax_img)
```

Image segmentation using thresholding

Image segmentation using thresholding is a fundamental technique in computer vision and image processing that involves partitioning an image into regions based on pixel intensity values. Thresholding is a simple yet effective method for separating objects from the background in an image.

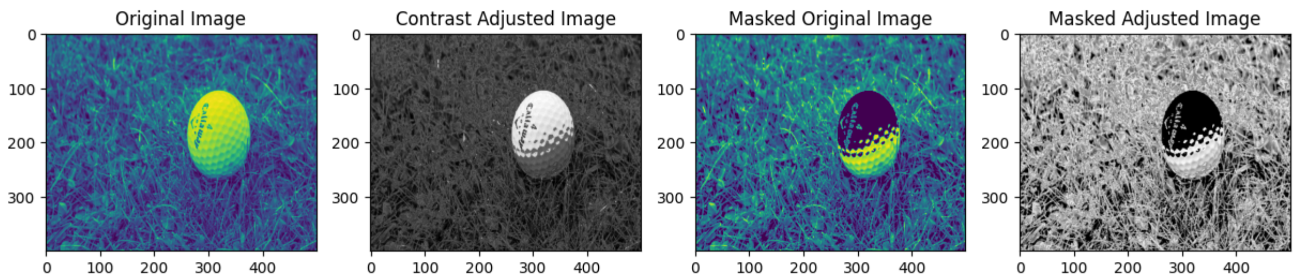
The basic concept of thresholding involves comparing each pixel intensity value in the image to a predefined threshold value. If the pixel intensity is above the threshold, it is assigned to one segment (foreground), and if it is below the threshold, it is assigned to another segment (background).

The formula for thresholding can be expressed as:

$$s(x, y) = \begin{cases} 1 & \text{if } I(x, y) > T \\ 0 & \text{if } I(x, y) \leq T \end{cases}$$

where (I(x, y)) represents the intensity value of the pixel at coordinates ((x, y)), (s(x, y)) is the segmented pixel value (either 0 or 1), and (T) is the threshold value.

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

path = '/content/drive/MyDrive/Cvip_Lab/images/golf.png'

# Load the image
cherry = cv2.imread(path)

# Convert BGR to grayscale
cherry_gray = cv2.cvtColor(cherry, cv2.COLOR_BGR2GRAY)

# Normalize pixel values to the range [0, 1]
img_normalized = cherry_gray / 255.0

# Apply contrast adjustments
img_normalized = 0.5 * img_normalized
img_normalized[img_normalized < (50 / 255)] = 2 * img_normalized[img_normalized < (50 / 255)]
img_normalized[img_normalized > (100 / 255)] = 2 * img_normalized[img_normalized > (100 / 255)]

# Create a mask based on intensity range
lower_threshold = 30
upper_threshold = 200
mask = cv2.inRange(cherry_gray, lower_threshold, upper_threshold)

# Apply the mask to the original and adjusted images
masked_original = cv2.bitwise_and(cherry_gray, cherry_gray, mask=mask)
masked_adjusted = cv2.bitwise_and((img_normalized * 255).astype(np.uint8), (img_normalized * 255).astype(np.uint8), mask=mask)

# Display the original, adjusted, masked original, and masked adjusted images
plt.figure(figsize=(15, 5))

plt.subplot(1, 4, 1)
plt.imshow(cherry_gray)
plt.title('Original Image')

plt.subplot(1, 4, 2)
plt.imshow(img_normalized, cmap='gray')
plt.title('Contrast Adjusted Image')

plt.subplot(1, 4, 3)
plt.imshow(masked_original)
plt.title('Masked Original Image')

plt.subplot(1, 4, 4)
plt.imshow(masked_adjusted, cmap='gray')
plt.title('Masked Adjusted Image')

plt.show()
```


Original Image
Contrast Adjusted Image
Masked Original Image
Masked Adjusted Image
0
100
200
300
400