LNMIIT
The LNM Institute of
Information Technology

PROJECT
Image Segmentation

NAME : DEVANSH SRIVASTAVA (21UCS061)
DHRUV PATEL (21UCS068)

SUBJECT : DIP

MENTOR : ANUKRITI BANSAL

CONTENTS

Task Description :

Here our main focus is to get the desired(elephant) potion of the image from the RGB Image. So we have done this task by following Steps :

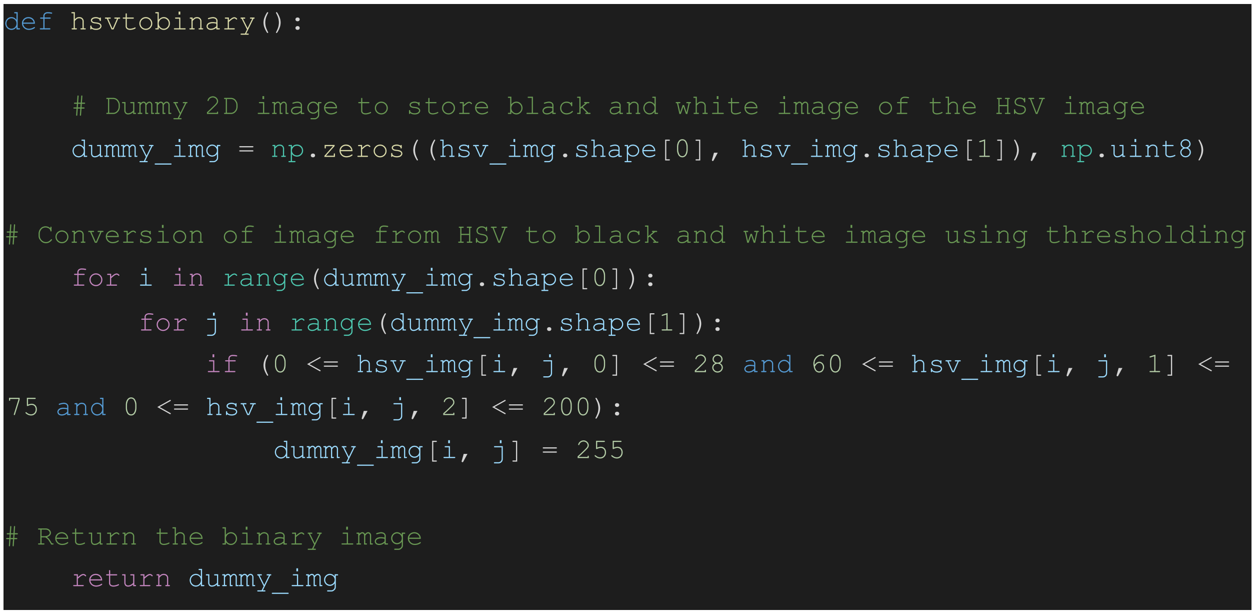
1. Converting RGB to HSV image
2. HSV image to Required portion image
3. Dilation of the image
4. Largest Connected Component from the image
5. Closing Operation
6. AND of the image with Original Image we get final Image

Converting RGB to HSV image :

Here we are simply converting the RGB image to the HSV Image as it is better to modification with the HSV Image as compared with the RGB Image.

HSV image to Black white image

Here by using thresholding we are getting the required portion from the HSV image converted into black and white. The pixels with the Hue range : 0-28
Saturation range : 50-75 and Intensity range : 0-200 are converted to white and others all black.

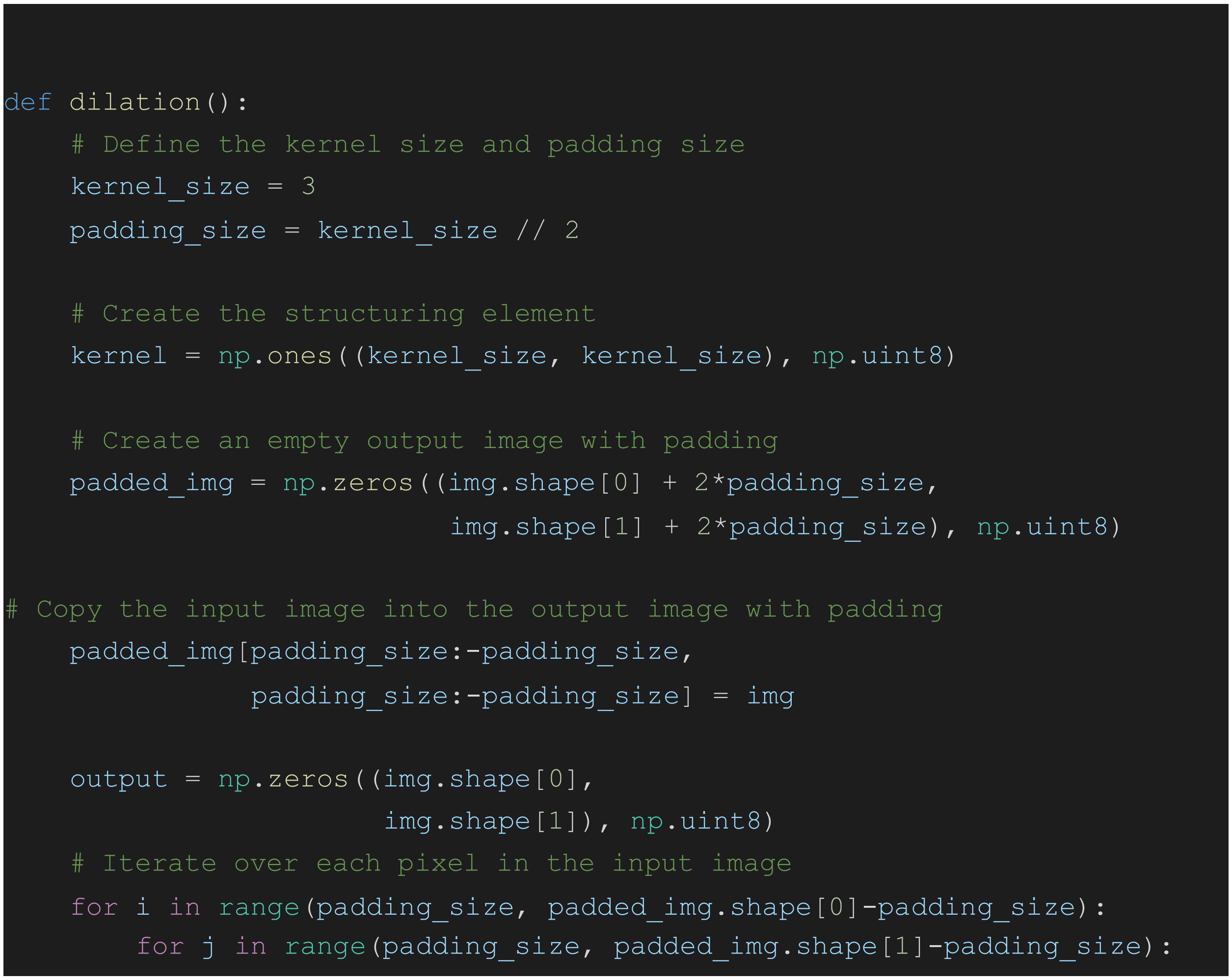


```
def hsvtobinary():

    # Dummy 2D image to store black and white image of the HSV image
    dummy_img = np.zeros((hsv_img.shape[0], hsv_img.shape[1]), np.uint8)

# Conversion of image from HSV to black and white image using thresholding
    for i in range(dummy_img.shape[0]):
        for j in range(dummy_img.shape[1]):
            if (0 <= hsv_img[i, j, 0] <= 28 and 60 <= hsv_img[i, j, 1] <=
75 and 0 <= hsv_img[i, j, 2] <= 200):
                dummy_img[i, j] = 255

# Return the binary image
    return dummy_img
```

LNMIIT
The LNM Institute of
Information Technology

Dilation of Image:

Here we have Dilated the image so as to repair the breaks, and give the defined boundary. Here we have used a 3x3 mask of the square type for dilation.



```
def dilation():
    # Define the kernel size and padding size
    kernel_size = 3
    padding_size = kernel_size // 2

    # Create the structuring element
    kernel = np.ones((kernel_size, kernel_size), np.uint8)

    # Create an empty output image with padding
    padded_img = np.zeros((img.shape[0] + 2*padding_size,
                           img.shape[1] + 2*padding_size), np.uint8)

# Copy the input image into the output image with padding
    padded_img[padding_size:-padding_size,
               padding_size:-padding_size] = img

    output = np.zeros((img.shape[0],
                       img.shape[1]), np.uint8)
    # Iterate over each pixel in the input image
    for i in range(padding_size, padded_img.shape[0]-padding_size):
        for j in range(padding_size, padded_img.shape[1]-padding_size):
            # Check if the kernel can be centered on the current pixel
            if np.max(padded_img[i-padding_size:i+padding_size+1,
j-padding_size:j+padding_size + 1] * kernel) > 0:

                # If so, set the output pixel to white
                output[i-padding_size, j-padding_size] = 255
# Return the dilated image
    return output
```

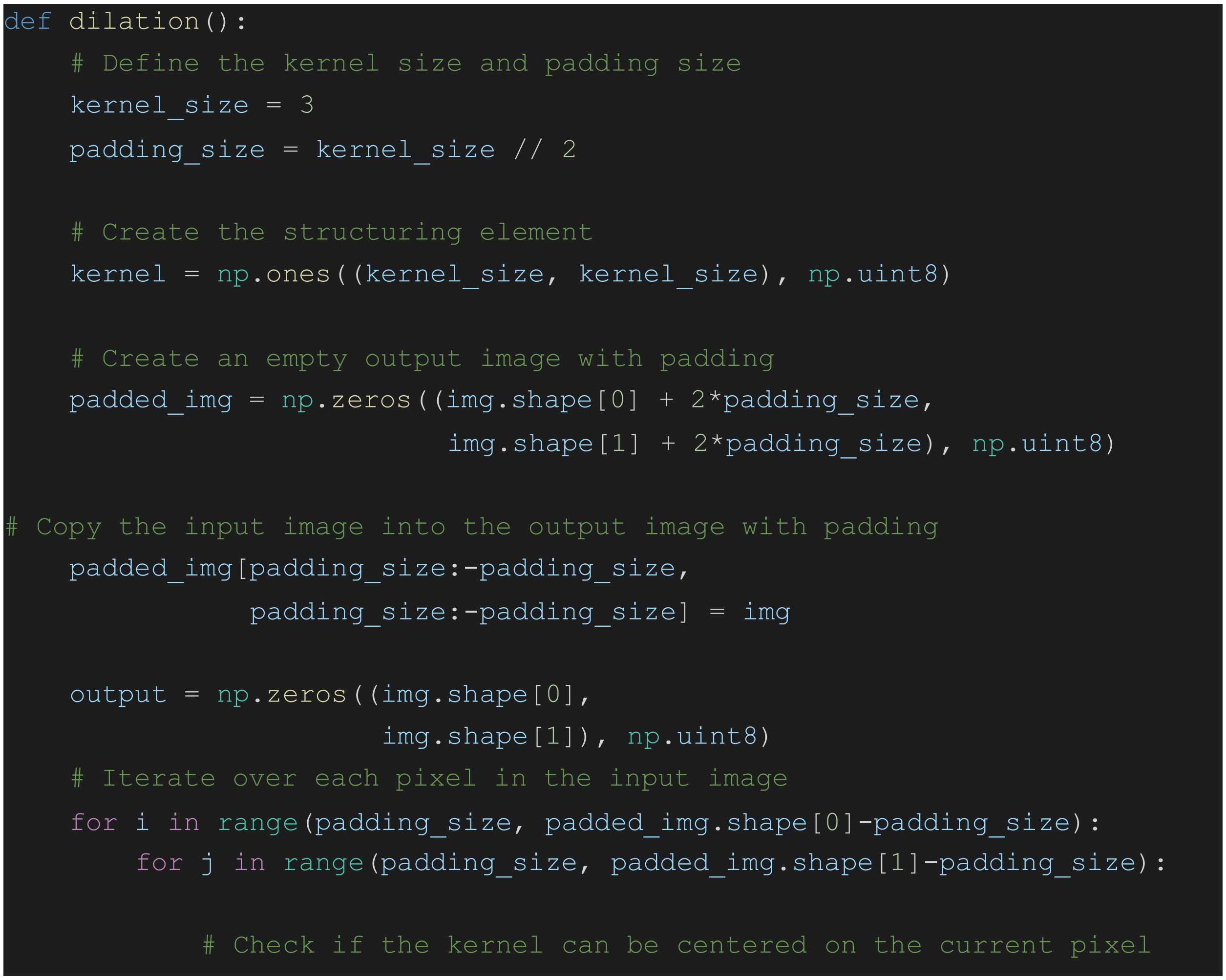
Finding Largest Connected Component

Here we have to find the largest connected component so that we can get the desired part of the image (elephant) from the image.

```
def largest_connected_component():

    # Creating an empty queue
    q = Queue(maxsize=0)

    # Empty 2D image for checking whether pixel already visited or not
    checker = np.zeros_like(img)

    # Empty list for storing connected components
    temp = []

    # Empty list for storing the coordinates inside the largest connected
component
    final = []

    # Stores no of elements in connected components of the image
    max_size = temp_size = 0

    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            if (img[i][j] == 0):
                if (temp_size > max_size):
                    final = temp.copy()
                    max_size = temp_size
                temp_size = 0
                continue
            if (checker[i][j] == 255):
                continue
            if (img[i][j] == 255):
                temp = []
                q.put([i, j])

                # Enqueue the elements only if pixel not visited and pixel
is of white color
                # Dequeue the elements if pixel already visited or color
has been changed from black to white
                while (q.empty() != True):
                    x = q.get()
                    row = x[0]
                    column = x[1]
                    if (checker[row][column] == 255):
                        continue
                    checker[row][column] = 255
                    temp_size = temp_size+1
                    temp.append(x)
                    # Checking the 4 nearest connected pixels
                    if ((row+1) < img.shape[0]-1 and
checker[row+1][column] == 0 and img[row+1][column] == 255):
                        q.put([row+1, column])
                    if ((row-1) > 0 and checker[row-1][column] == 0 and
img[row-1][column] == 255):
                        q.put([row-1, column])
                    if ((column+1) < img.shape[1]-1 and
checker[row][column+1] == 0 and img[row][column+1] == 255):
                        q.put([row, column+1])
                    if ((column-1) > 0 and checker[row][column-1] == 0 and
img[row][column-1] == 255):
                        q.put([row, column-1])

    # 2D image to store the largest connected component in the image
    largest_component = np.zeros_like(img)
    for i in range(0, max_size):
        x = (final[i])[0]
        y = (final[i])[1]
        largest_component[x][y] = 255

    return largest_component
```

Closing of the Image

Enlarge the boundaries for the bright portion of the image and shrink the background color holes in the image, thus giving the smoothing effect in the image.



```
def dilation():
    # Define the kernel size and padding size
    kernel_size = 3
    padding_size = kernel_size // 2

    # Create the structuring element
    kernel = np.ones((kernel_size, kernel_size), np.uint8)

    # Create an empty output image with padding
    padded_img = np.zeros((img.shape[0] + 2*padding_size,
                           img.shape[1] + 2*padding_size), np.uint8)

# Copy the input image into the output image with padding
    padded_img[padding_size:-padding_size,
               padding_size:-padding_size] = img

    output = np.zeros((img.shape[0],
                       img.shape[1]), np.uint8)
    # Iterate over each pixel in the input image
    for i in range(padding_size, padded_img.shape[0]-padding_size):
        for j in range(padding_size, padded_img.shape[1]-padding_size):

            # Check if the kernel can be centered on the current pixel
            if np.max(padded_img[i-padding_size:i+padding_size+1,
j-padding_size:j+padding_size + 1] * kernel) > 0:

                # If so, set the output pixel to white
                output[i-padding_size, j-padding_size] = 255
# Return the dilated image
    return output


def erosion():
    # Define the kernel size and padding size
    kernel_size = 3
    padding_size = kernel_size // 2

    # Create the structuring element
    kernel = np.ones((kernel_size, kernel_size), np.uint8)

    # Create an empty output image with padding
    padded_img = np.zeros((img.shape[0] + 2*padding_size,
                           img.shape[1] + 2*padding_size), np.uint8)

# Copy the input image into the output image with padding
    padded_img[padding_size:-padding_size,
               padding_size:-padding_size] = img

    output = np.zeros((img.shape[0],
                       img.shape[1]), np.uint8)
    # Iterate over each pixel in the input image
    for i in range(padding_size, padded_img.shape[0]-padding_size):
        for j in range(padding_size, padded_img.shape[1]-padding_size):

            # Check if the kernel can be centered on the current pixel
            if np.sum(padded_img[i-padding_size:i+padding_size+1,
j-padding_size:j+padding_size + 1] * kernel) == 2295:

                # If so, set the output pixel to white
                output[i-padding_size, j-padding_size] = 255
# Return the eroded image
    return output
```

Getting Resultant Image

Now we can get the Resultant Image by taking the AND operation with the original image, thus the bright part will be shown in the final image and the rest part will be getting the intensity values as 0.

```
def coloring():
    # Performing AND operation to extract the color image of the elephant from the mask created using largest component
    rows, columns = (img.shape[0], img.shape[1])
    # Dummy color image of black color
    output = np.zeros((rows, columns, 3), np.uint8)
    for i in range(rows):
        for j in range(columns):
            if (img[i][j] == 255):
                output[i][j] = rgb_img[i][j]
    # Return portion of elephant
    return output
```

Conclusion and Results:

1) Here we came to how we can get the desired region portion of the image from the image.
2) Here we also explored practically the usage of concepts like Dilation of image, Conversion between RGB and HSV images etc.
3) Finally we strengthen our team spirit along with the concepts and practically usability of the concepts.