# University of Moratuwa

## Department of Electronic and Telecommunication Engineering


# EN3160 - Image Processing and Machine Vision

## A01: Intensity Transformations and Neighborhood Filtering

**Index No: 200396U**

**Name: Miranda C.M.C.C.**

## Question 1

- Important part of the code

```
# Intensity Transformation
c = np.array([(50,50), (50,100),
(150,255), (150,150)])
t1 = np.linspace(0, c[0,1], c[0,0] + 1).
astype("uint8")
t2 = np.linspace(c[1,1] + 1, c[2,1], c[2,0]
- c[1,0]).astype("uint8")
t3 = np.linspace(c[3,1] + 1, 255, 255 -
c[3,0]).astype("uint8")

transform = np.concatenate((t1, t2),
axis=0). astype("uint8")
transform = np.concatenate((transform, t3),
axis=0).astype("uint8")

img_orig = cv.imread('emma.jpg',
cv.IMREAD_GRAYSCALE)
img_transformed = cv.LUT(img_orig,
transform)
```

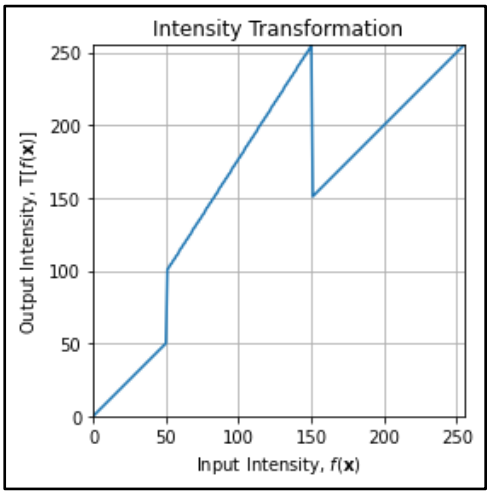


*Figure 1: Intensity Transformation*

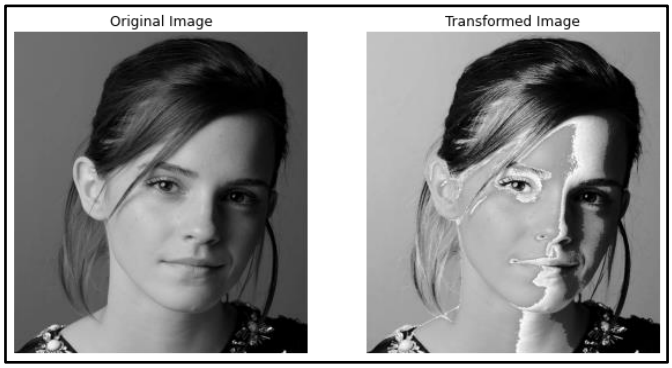


*Figure 1: Original and Transformed Images*

The objective is to enhance the visual appeal of an input image by implementing a change in pixel values through an intensity transformation. This alteration aims to create a more visually pleasing outcome for the image.

## Question 2

**(a)** Important part of the code

```
# Enhancing white matter
c= np.array([(50,25), (50,25), (180,90),
(180,220)])
t1 = np.linspace(0, c[0,1], c[0,0] + 1).
astype("uint8")
t2 = np.linspace(c[1,1] + 1, c[2,1], c[2,0] -
c[1,0]).astype("uint8")
t3 = np.linspace(c[3,1] + 1, 255, 255 -
c[3,0]).astype("uint8")

transform = np.concatenate((t1, t2),
axis=0).astype("uint8")
transform = np.concatenate((transform, t3),
axis=0).astype("uint8")
```

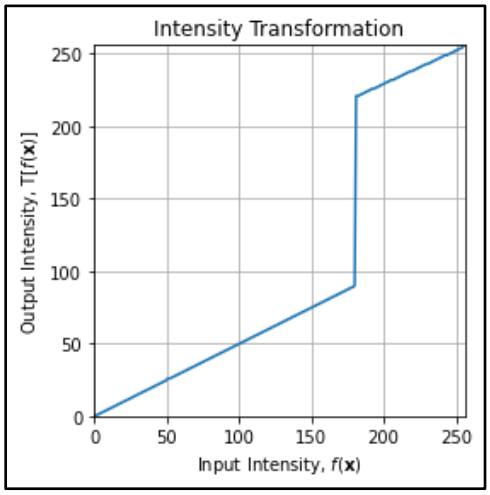


*Figure 3: Intensity Transformation (White Matter)*

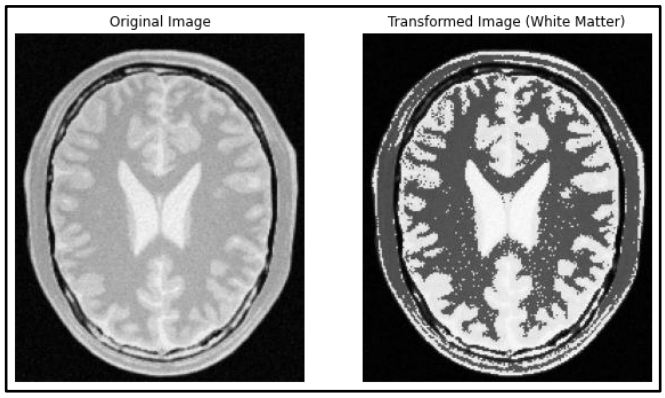


*Figure 4: Original & Transformed Images (White Matter)*

**(b)** Important part of the code

```
# Enhancing gray matter
c= np.array([(50,50), (50,100), (180,255),
(180,50)]
t1 = np.linspace(0, c[0,1], c[0,0] + 1).
astype("uint8")
t2 = np.linspace(c[1,1] + 1, c[2,1], c[2,0] -
c[1,0]).astype("uint8")
t3 = np.linspace(c[3,1] + 1, 0, 255 -
c[3,0]).astype("uint8")
```

```
transform = np.concatenate((t1, t2),
axis=0).astype("uint8")
transform = np.concatenate((transform, t3),
axis=0).astype("uint8")
```

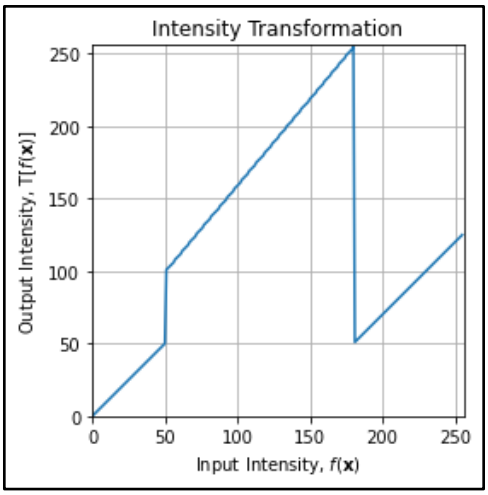


*Figure 5: Intensity Transformation (Gray Matter)*

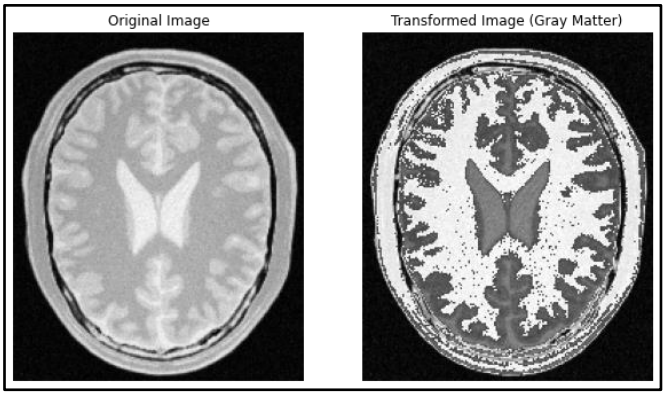


*Figure 2: Original & Transformed Images (Gray Matter)*

In pursuit of enhancing the visual characteristics of a proton density brain image, our objective is to emphasize the white matter and gray matter within the image. To achieve this, we formulated distinct intensity transformations tailored to each type of tissue.

By designing separate intensity transformations for white matter and gray matter, we can target and enhance the unique features of these brain tissues, thereby improving their visibility and making them more pronounced in the final image.

## Question 3

**(a)** Important part of the code

```
# Gamma Correction
def apply_gamma_correction(L_channel, gamma):
    L_Channel_corrected = np.array([(i/255.0)
**(gamma) * 255.0 for i in L_channel]).
astype("uint8")
    L_corrected = np.clip (L_Channel_corrected , 0,
255)
    return L_corrected
```


*Figure 7: Original & L-plane Corrected Images*

In this task, we are implementing gamma correction on the L plane within the L∗a∗b color space. Gamma correction serves as a method to fine-tune the intensity levels of an image. Here, our focus is on applying this technique to the luminance channel (L) of the color space. The degree of correction is governed by a particular γ value, which determines the extent of the adjustments made.

**(b)** Important part of the code

```
# Gamma correction to the L channel
L, a, b = cv.split(original_image_Lab)
gamma_value = 0.5
L_corrected = apply_gamma_correction(L,
gamma_value)
lab_corrected = cv.merge((L_corrected, a, b))
rgb_corrected = cv.cvtColor(lab_corrected,
cv.COLOR_Lab2RGB)
```

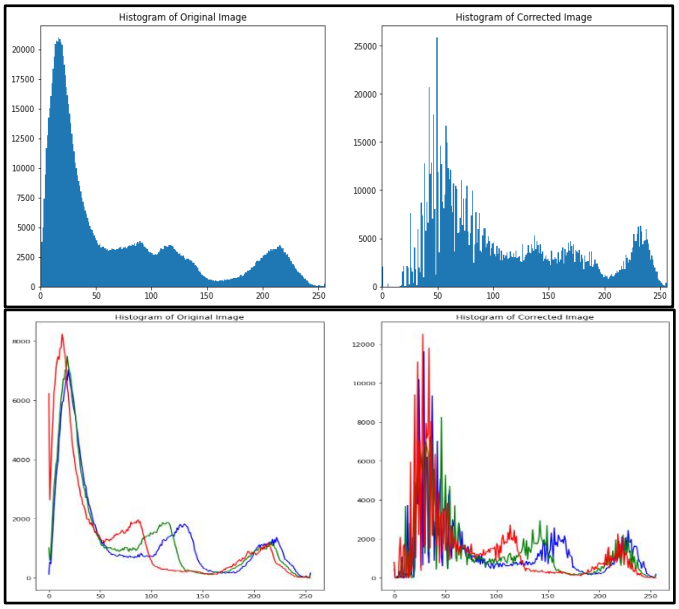


*Figure 3: Histogram of Original & Corrected Images*

Furthermore, histograms were generated to visually compare the intensity distribution of the initial L channel with the modified version post-correction. These histograms provide a clear view of the level of enhancement attained through the gamma correction procedure, offering valuable insights into the effectiveness of the process.

## Question 4

**(a)**

```
# Split the HSV image into hue, saturation, and value planes
hue, saturation, value = cv.split(hsv_image)
```

**(b)**

```
# Intensity transformation function
def intensity_transformation(x, a, sigma):
    return np.minimum(x + (a * 128) * np.exp(-((x - 128)**2) / (2 * sigma**2)), 255)
```

**(c)**

```
a = 0.3
sigma = 70
```

**(d)** Important part of the code

```
transformed = np.vectorize(intensity_transformation)(saturation, a, sigma)
transformed = np.clip(transformed, 0, 255).astype(np.uint8)
# Merge the transformed saturation back into the HSV image
transformed_hsv_image = cv.merge((hue, transformed, value))
```


*Figure 4: Original & Vibrance Enhanced Images*

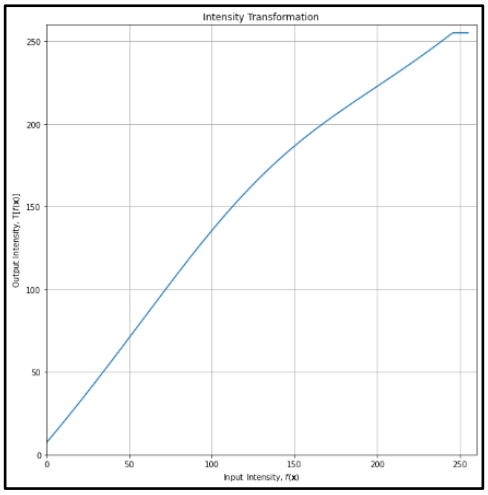


*Figure 5: Intensity Transformation (a = 0.3)*

When enhancing the vibrance of a photograph, we utilize a tailored saturation plane transformation through the function f(x). This method entails dividing the image into its hue, saturation, and value constituents. The saturation plane is subjected to transformation, involving the fine-tuning of parameter 'a' for the best possible outcome.

$$f(x) = \min\left(x + a \times 128 e^{-\frac{(x-128)^2}{2\sigma^2}}, 255\right)$$

## Question 5

- Important part of the code

```
# Compute Histogram
def compute_histogram(image):
    histogram = np.zeros(256, dtype=int)
    for pixel_value in image.flatten():
        histogram[pixel_value] += 1
    return histogram

# Compute Cumulative Histogram
def compute_cumulative_histogram(histogram):
    cumulative_histogram = np.zeros(256, dtype=int)
    cumulative_histogram[0] = histogram[0]
    for i in range(1, 256):
        cumulative_histogram[i] = cumulative_histogram[i - 1] + histogram[i]
    return cumulative_histogram

# Histogram Equalization
def histogram_equalization(image, histogram, cumulative_histogram, num_pixels):
    equalized_image = np.zeros_like(image)
    for y in range(image.shape[0]):
      for x in range(image.shape[1]):
        pixel_value = image[y, x]
        equalized_value = int(255 * cumulative_histogram[pixel_value] / num_pixels)
        equalized_image[y, x] = equalized_value
    return equalized_image

# Compute histogram
histogram = compute_histogram(image)
cumulative_histogram = compute_cumulative_histogram(histogram)
cdf = cumulative_histogram / num_pixels
# Compute equalized image
equalized_image = histogram_equalization(image, histogram, cumulative_histogram, num_pixels)
equalized_histogram = compute_histogram(equalized_image)
equalized_cdf = compute_cumulative_histogram(equalized_histogram) / num_pixels
```

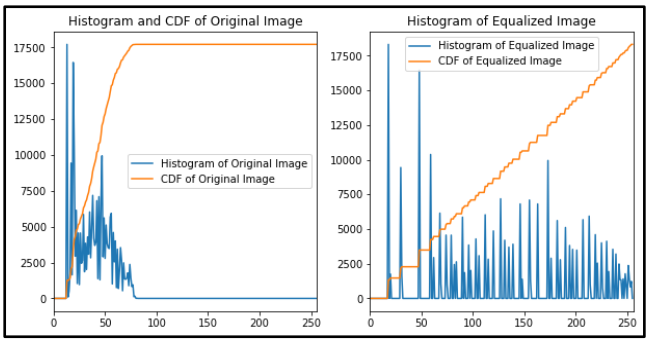


*Figure 6: Histograms and CDFs of Original and Equalized Images*

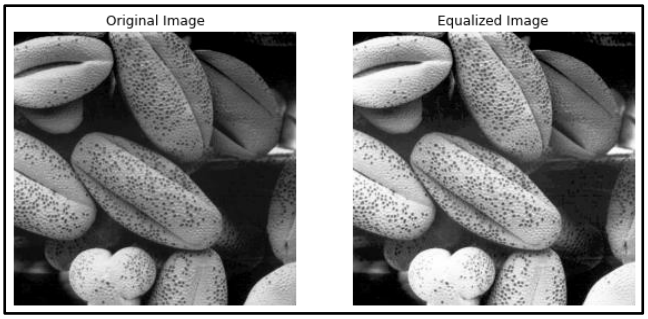


*Figure 7: Original & Histogram Equalized Images*

The task is to craft a personalized Python function for histogram equalization. This approach avoids relying on built-in functions such as *cv2.equihist()*. The function commences by loading the image as a *numpy* array, followed by the computation of its histogram and subsequent cumulative distribution function (CDF). Normalization of the CDF follows, ensuring it stays within the desired range. Applying histogram equalization entails redistributing pixel intensities based on the CDF. The pre-equalization and post-equalization histograms are then displayed, revealing the transformation's effect on the image's intensity distribution.

## Question 6

**(a)**

```
# Load the image
image = cv.imread("jeniffer.jpg",
cv.IMREAD_COLOR)
# Convert the image to HSV color space
hsv_image = cv.cvtColor(image,
cv.COLOR_BGR2HSV)
# Split into hue, saturation, and value
hue, saturation, value = cv.split(hsv_image)
```


*Figure 8: Images of HSV planes*

**(b)**

```
# Saturation plane can be used to extract the
foreground
threshold = 12
mask = (saturation > threshold).astype(np.uint8)
* 255
mask_3d = np.repeat(mask[:, :, None], 3,
axis=2)
foreground_hsv = cv.bitwise_and(hsv_image,
mask_3d)
```


*Figure 9: Mask and Foreground Mask*

**(c), (d), (e)** Important part of the code

```
equalized_foreground = foreground_rgb.copy()
colors = ('r', 'g', 'b')
total = mask.sum() // 255

# Loop over color channels and calculate
histograms
for i, color in enumerate(colors):
    hist = cv.calcHist([foreground_rgb], [i],
mask, [256], [0, 256])
    ax[0].plot(hist, color=color)
    ax[0].set_xlim([0, 256])

    cumulative = np.cumsum(hist)
    ax[1].plot(cumulative, color=color)
    ax[1].set_xlim([0, 256])

    transform = cumulative * 255 / total
    equalized_foreground[:, :, i] =
transform[foreground_rgb[:, :, i]]

# Remove background again after equalization
equalized_foreground =
cv.bitwise_and(equalized_foreground, mask_3d)
```

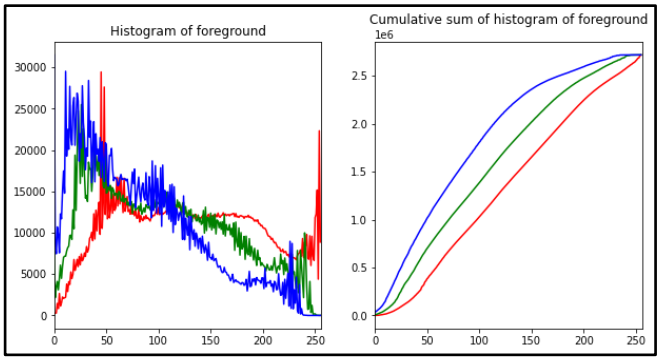


*Figure 10: Histogram & Cumulative Sum of Histogram*

**(f)**

```
# Extract the background and add with the
histogram equalized foreground.
background_mask_3d = 255 - mask_3d
background_hsv = np.bitwise_and(hsv_image,
background_mask_3d)
background_rgb = cv.cvtColor(background_hsv,
cv.COLOR_HSV2RGB)
final_image = background_rgb +
equalized_foreground
```

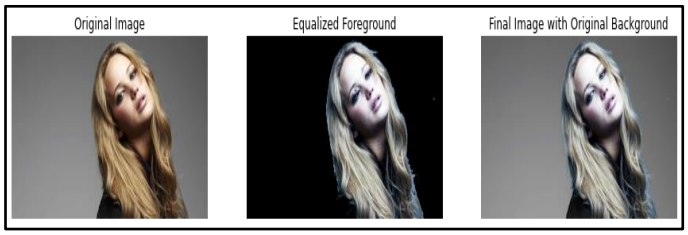


*Figure 11: Original image & the result with the histogram equalized.*

This task focuses on improving the histogram of an image's foreground, resulting in a histogram-equalized foreground image. The process involves dividing the image into its hue, saturation, and value components, extracting a foreground mask through thresholding, and isolating the foreground using operations like *cv.bitwise_and*. The foreground's histogram is calculated, and its cumulative sum is computed. Applying histogram equalization formulas enhances the foreground. Ultimately, the background is reintegrated to achieve the desired result.

## Question 7

**(a)**

```
# Sobel Filter using filter2D
kernel = np.array([[1, 0, -1], [2, 0, -2],
[1, 0, -1]])
image_a = cv.filter2D(image, -1, kernel)
```

**(b)**

```
def filter(image, kernel):
    assert kernel.shape[0] % 2 == 1 and
kernel.shape[1] % 2 == 1
    k_hh = kernel.shape[0] // 2
    k_hw = kernel.shape[1] // 2
    h, w = image.shape
    # Normalize the input image
    image_normalized = cv.normalize
(image.astype('float'), None, 0, 1,
cv.NORM_MINMAX)
    result = np.zeros(image.shape,
dtype='float')
    for m in range(k_hh, h - k_hh):
      for n in range(k_hw, w - k_hw):
        result[m, n] = np.dot
      (image_normalized[m - k_hh : m + k_hh +
      1, n - k_hw : n + k_hw + 1].flatten(),
      kernel.flatten())
# Scale the result to the range [0, 255]
    result = result * 255
    result = np.minimum(255, np.maximum(0,
result)).astype(np.uint8)
    return result
```

**(c)**

```
def filter_in_steps(input_image,
first_kernel, second_kernel):
    # Define filtering for an already
normalized image without any rounding
    def filter_step(image, kernel):
        assert kernel.shape[0] % 2 == 1 and
kernel.shape[1] % 2 == 1
        k_hh = kernel.shape[0] // 2
        k_hw = kernel.shape[1] // 2
        h, w = image.shape
        result = np.zeros(image.shape,
dtype='float')
        for m in range(k_hh, h - k_hh):
          for n in range(k_hw, w - k_hw):
            result[m, n] = np.dot(image[m -
k_hh: m + k_hh + 1, n - k_hw: n + k_hw +
1].flatten(), kernel.flatten())
        return result

    image_float = cv.normalize
(input_image.astype('float'), None, 0, 1,
cv.NORM_MINMAX)
    result = filter_step(filter_step
(image_float, first_kernel), second_kernel)
    result = result * 255
    result = np.minimum(255, np.maximum(0,
result)).astype(np.uint8)
    return result
```

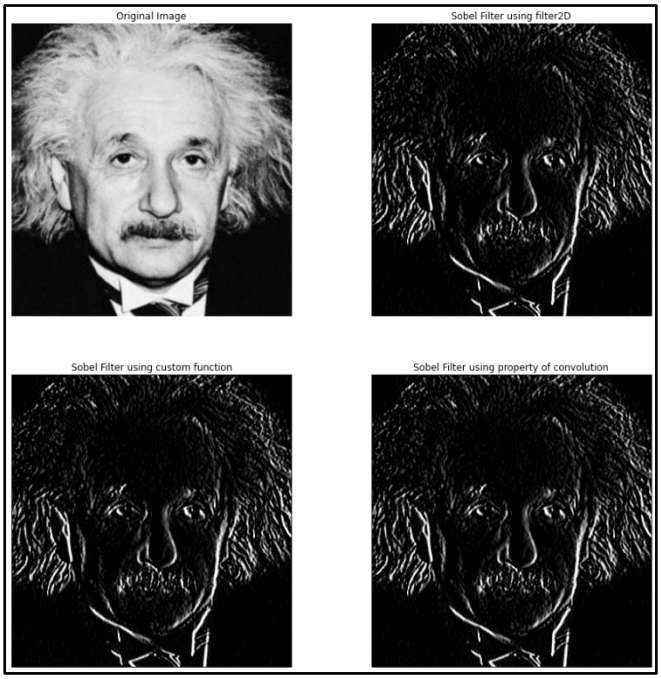


*Figure 12: Original Image and Filtered Images*

In this section, various techniques were used for applying Sobel filtering to an image:

- In the first approach, the 'filter2D' function was used, which is readily available for Sobel filtering.
- The second method involves crafting custom code for Sobel filtering, giving us greater control over the process.
- Lastly, a convolution property was used that entails a specific matrix configuration. By convolving this matrix with the image, we attain the Sobel filtering effect, effectively accentuating edge features within the image.

## Question 8

**(a)**

```
def nearest_neighbors(image, indices):
    indices[0] =
np.minimum(np.round(indices[0]),
image.shape[0] - 1)
    indices[1] =
np.minimum(np.round(indices[1]),
image.shape[1] - 1)
    indices = indices.astype(np.uint64)
    return image[indices[0], indices[1]]
```

**(b)**

```
def bilinear_interpolation(image, indices):
    floors =
np.floor(indices).astype(np.uint64)
    ceils = floors + 1
    ceils_limited = [np.minimum(ceils[0],
image.shape[0] - 1), np.minimum(ceils[1],
image.shape[1] - 1)]

    p1 = image[floors[0], floors[1]]
    p2 = image[floors[0], ceils_limited[1]]
    p3 = image[ceils_limited[0], floors[1]]
    p4 = image[ceils_limited[0],
ceils_limited[1]]
    # Repeat indices for the 3 color planes
    indices = np.repeat(indices[:, :, :,
None], 3, axis=3)
    ceils = np.repeat(ceils[:, :, :, None],
3, axis=3)
    floors = np.repeat(floors[:, :, :, None],
3, axis=3)
    # Find the horizontal midpoints
    m1 = p1 * (ceils[1] - indices[1]) + p2 *
(indices[1] - floors[1])
    m2 = p3 * (ceils[1] - indices[1]) + p4 *
(indices[1] - floors[1])
    # Find the vertical midpoint of
horizontal midpoints
    m = m1 * (ceils[0] - indices[0]) + m2 *
(indices[0] - floors[0])
    return m.astype(np.uint8)
```

```
def zoom(image, factor, interpolation):
    h, w, _ = image.shape
    zoom_h, zoom_w = round(h * factor),
round(w * factor)
    zoomed_image = np.zeros((zoom_h, zoom_w,
3)).astype(np.uint8)
    zoomed_indices = np.indices((zoom_h,
zoom_w)) / factor
    if interpolation == "nn":
        zoomed_image =
nearest_neighbors(image, zoomed_indices)
    elif interpolation == "bi":
        zoomed_image =
bilinear_interpolation(image, zoomed_indices)
    return zoomed_image

def normalized_ssd(image1, image2):
    ssd = np.sum((image1 - image2)**2)
    return ssd / (image1.size * 255 * 255)
```

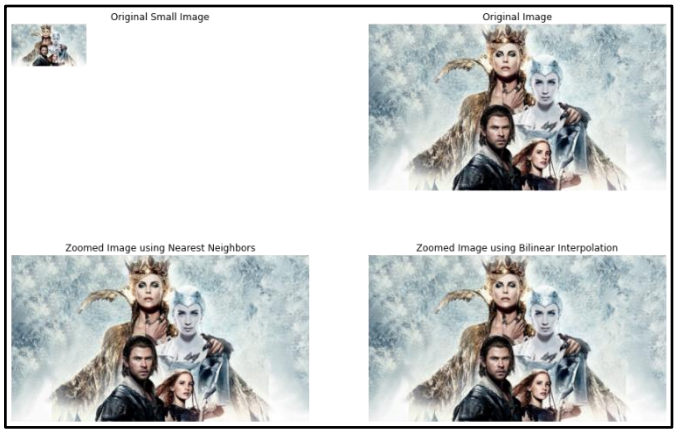


*Figure 13: Original Images & Zoomed Images*

```
Image 01: NN = 0.000616, BL = 0.000603
Image 02: NN = 0.000258, BL = 0.000249
Image 04: NN = 0.001257, BL = 0.001255
Image 05: NN = 0.000839, BL = 0.000825
Image 06: NN = 0.000537, BL = 0.000546
Image 07: NN = 0.000470, BL = 0.000464
Image 09: NN = 0.000431, BL = 0.000410
```

This program is to zoom in on an image using a given factor 's' within the range of (0, 10]. It offers two distinct zooming techniques: nearest-neighbor and bilinear interpolation. In the nearest-neighbor method, the program enlarges the image by replicating the nearest pixel values, maintaining a blocky appearance. On the other hand, the bilinear interpolation method calculates pixel values by considering weighted averages of nearby pixels, resulting in smoother zoomed images. To assess the accuracy of the zooming, the program computes the normalized sum of squared differences (SSD) when scaling up small images by a factor of 4, and then compares it to the original images.

## Question 9

**(a)**

```
# Create a mask and initialize it
mask = np.zeros(image.shape[:2], np.uint8)

# Define rectangular ROI
rect = (50, 100, image.shape[1],
image.shape[0] - 400)

# Apply GrabCut algorithm
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)
cv.grabCut(image, mask, rect, bgdModel,
fgdModel, 15, cv.GC_INIT_WITH_RECT)

# Select pixels that are background or
probably background as 0 and others as 1
foreground_mask =
np.where((mask==2)|(mask==0), 0,
1).astype('uint8')
background_mask = 1 - foreground_mask

# Create foreground and background images
foreground_image = image *
foreground_mask[:, :, np.newaxis]
background_image = image *
background_mask[:, :, np.newaxis]
```

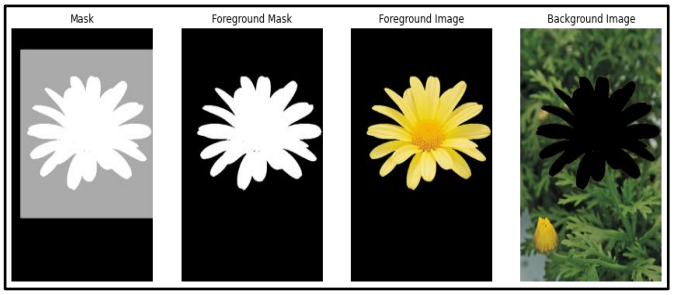


*Figure 14: Mask, Foreground and Background Images*

**(b)**

```
# Image with blurred background
kernel = 51
sigma = 5
blurred_background =
cv.GaussianBlur(background_image, (kernel,
kernel), sigma)
blurred_background = blurred_background *
background_mask[:, :, np.newaxis]
blurred_image = blurred_background +
foreground_image
```


In the provided flower image, where both the foreground and background are sharply focused, the task has two main objectives:

In the first part, we use the 'grabCut' algorithm for image segmentation. This results in two important outcomes: a segmentation mask that separates the foreground and background and isolated foreground and background images.

In the second part, the aim is to improve the image by strongly blurring the background, creating a visually striking separation between the subject and the background. We present both the original and enhanced images side by side for a clear view of the enhancement achieved.

**(c)**

The reason behind the notably darkened background just beyond the flower's edge in the enhanced image is a deliberate outcome of the blurring process employed to achieve a distinct visual separation between the subject and its surroundings.

This blurring technique, often used to create a captivating "bokeh" effect, intentionally reduces the sharpness of elements located farther away from the main subject. Consequently, the background immediately beyond the flower's edge is intentionally rendered in a softened and darker manner. This artistic choice serves to accentuate the flower as the central focal point, enhancing its presence and creating a visually pleasing contrast between the subject and its environment.

- Python Code: [GitHub Link]