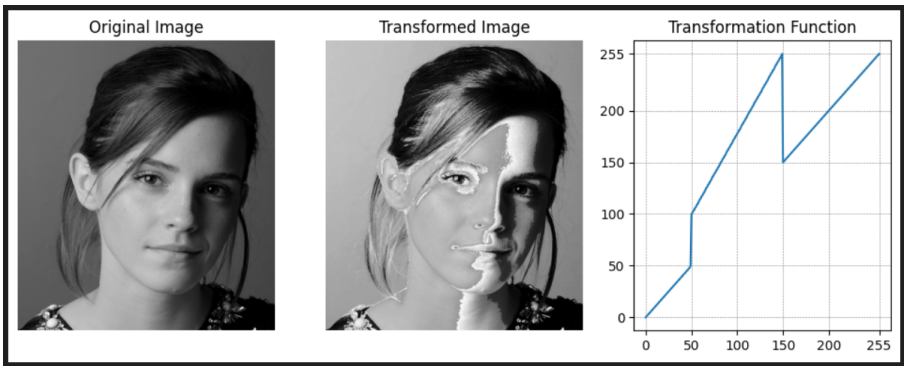
Intensity-Transformations-and-Neighborhood-Filtering

Walgampaya H.K.B 220675T
GitHub Link: GitHub

(https://github.com/HimethW/Intensity-Transformations-and-Neighborhood-Filtering/blob/main/Assignment1.ipynb)

1 Question 01

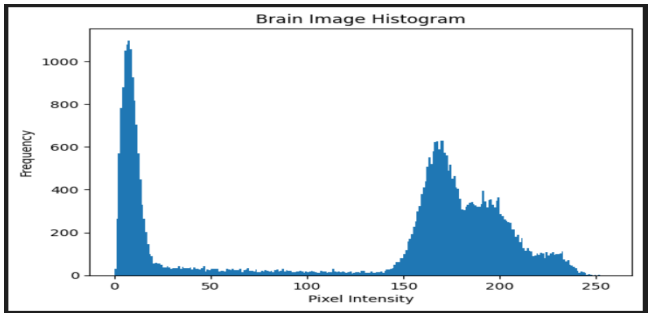
In this, an intensity transformation was applied based on the given transformation function. It can be seen



that while the lower intensities(0-50) and higher intensities(150-255) remains the same, the middle range intensities(50-150) are stretched out over a wider range(100-255). So in the image we see that while the most dark and bright areas(shadows and highlights) remain intact, the middle range intensity areas have amplified making those textures more visible. Furthermore, there is no clipping do to the output always remaining within 0-255

```
#transformation points
c = np.array([(50, 50), (50, 100), (150, 255), (150, 150), (255, 255)])
#from 0 to 49 = 50 parts
t1 = np.linspace(0, c[0,1], c[0,0]).astype('uint8')
#from 50 to 149 = 100 parts
t2 = np.linspace(c[1,1], c[2,1], c[2,0] - c[1,0]).astype('uint8')
#from 150 to 255 = 106 parts
t3 = np.linspace(c[3,1], c[4,1], c[4,0] - c[3,0] + 1).astype('uint8')
#Concatenate the three parts to form the complete transformation
transform = np.concatenate((t1, t2, t3), axis=0).astype('uint8')
#transform the image using the transformation function
f = cv.imread('a1images/emma.jpg', cv.IMREAD_GRAYSCALE)
g = transform[f]
```

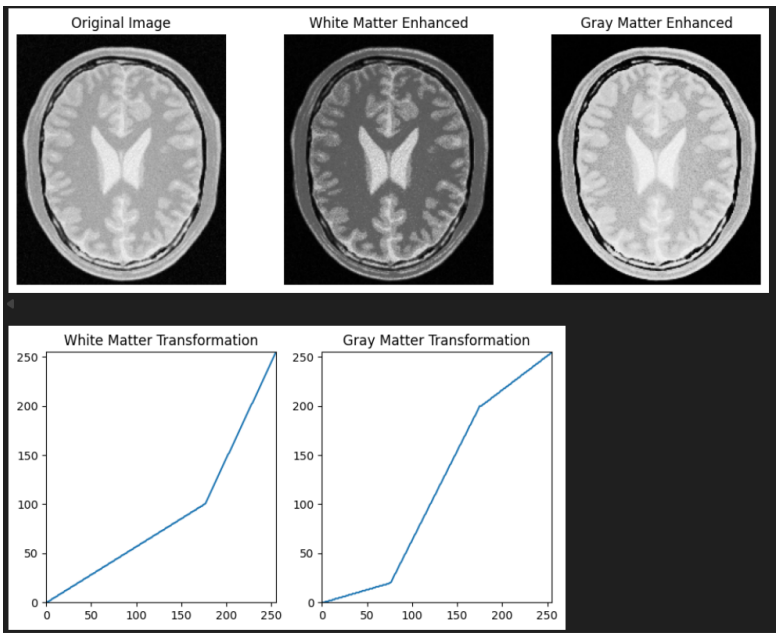
2 Question 02



A similar operation was done to accentuate the white matter and gray matter of a brain proton. First I did a histogram analysis to get an idea about how the intensities are distributes in the image in order to accentuate certain areas. So based on the histogram, I applied the following transformations

- White matter transformation: Compress the 0-175 region and stretch out the 175 to 255 region to show the higher intensities better
- Gray matter transformation: Compress region from 0-75 and 175-255(the extremes) and stretch out the middle intensities (75-175)

After applying the transformation, in the second image, the dark regions have become more darker (compressed) while the white matter has enhanced due to the stretch in higher intensities (darkening CSF and gray matter). Similarly, in the third image, we see that the mid tones have amplified thus making th e gray matter more visible. The question asks to accentuate certain areas so I have compresses the unwanted regions and amplified the wanted areas rather than removing the unwanted regions




```python
# White Matter Transformation
t1_wm = np.linspace(0, 100, 176).astype('uint8')  # from 0 to 175 = 176 parts
t2_wm = np.linspace(100, 255, 80).astype('uint8')  # from 176 to 255 = 80 parts
transform_wm = np.concatenate((t1_wm, t2_wm), axis=0).astype('uint8')
# Gray Matter Transformation
t1_gm = np.linspace(0, 20, 76).astype('uint8') # from 0 to 75 = 76 parts
t2_gm = np.linspace(20, 200, 100).astype('uint8')  # from 76 to 175 = 100 parts
t3_gm = np.linspace(200, 255, 80).astype('uint8')  # from 176 to 255 = 80 parts
transform_gm = np.concatenate((t1_gm, t2_gm, t3_gm), axis=0).astype('uint8')
# Load and transform the image
f = cv.imread('a1images/brain_proton_density_slice.png', cv.IMREAD_GRAYSCALE)
g_wm = transform_wm[f]  # white matter transformation
g_gm = transform_gm[f]  # gray matter transformation
```

3 Question 03

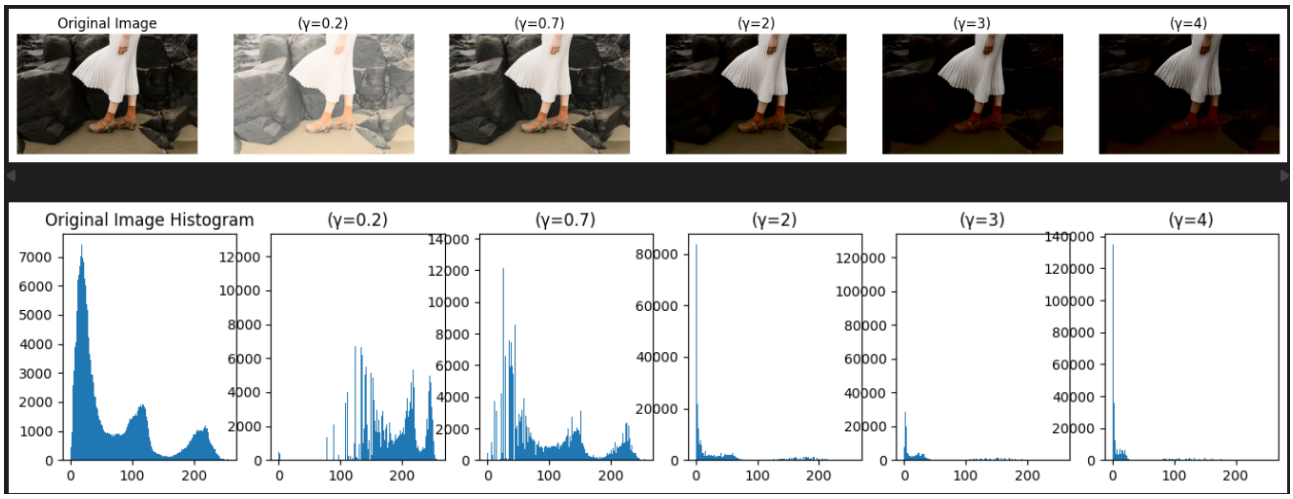
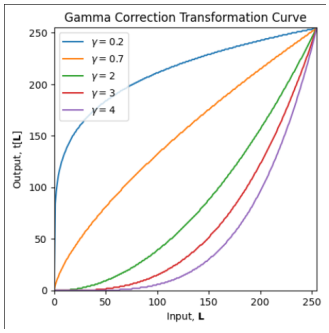


Figure 1: How different gamma values affect the L plane and image



Gamma correction is a non-linear intensity transformation that can be used to adjust the image brightness and contrast. When $\gamma < 1$, darker regions are brightened, showing hidden details in shadows, while $\gamma > 1$ darkens mid-tones and enhances contrast in brighter areas. The L channel was isolated as it contains intensity information independent of colour. By comparing results we can see that:

- when $\gamma < 1$, the intensities in the darker regions have increased(non-linear stretch), giving the image a brighter look and enhancing textures, whereas when $\gamma > 1$ the mid range intensities darkened (compressed) while the higher values remained. Also the color channels are not affected.
- This can be conformed by looking at the histograms. In the original image the intensities have spread in the full region. But when $\gamma < 1$ there is a right shift indicating the intensity enhancement in darker regions. When $\gamma > 1$, the histogram has shifted to the left showing that many intensities have reduced, giving the image a darker look.

From the results we can say that for this image $\gamma = 0.7$ is good fit as it enhances the image intensity while increasing the visibility of hidden details and textures (eg: details on rocks, etc.)

```
#Apply gamma correction to the L plane in the L*a*b color space and state the gamma value.
image_lab = cv.cvtColor(image, cv.COLOR_BGR2Lab)
L,a,b = cv.split(image_lab)
gamma_vals = [0.2,0.7,2,3,4]
for gamma in gamma_vals:
  t = np.array([((i/255)**gamma)*255 for i in np.arange(0,256)]).astype(np.uint8)  # Gamma
    correction transformation
  #plot the curve
  ax_gamma.plot(t, label=fr'$\gamma = {gamma}$')
  ax_gamma.set_title('Gamma Correction Transformation Curve')
  L_corrected = cv.LUT(L, t)  # Apply gamma correction
  image_corrected = cv.merge((L_corrected, a, b))
  # Convert back to BGR color space
  image_corrected_bgr = cv.cvtColor(image_corrected, cv.COLOR_Lab2BGR)
```

4 Question 04

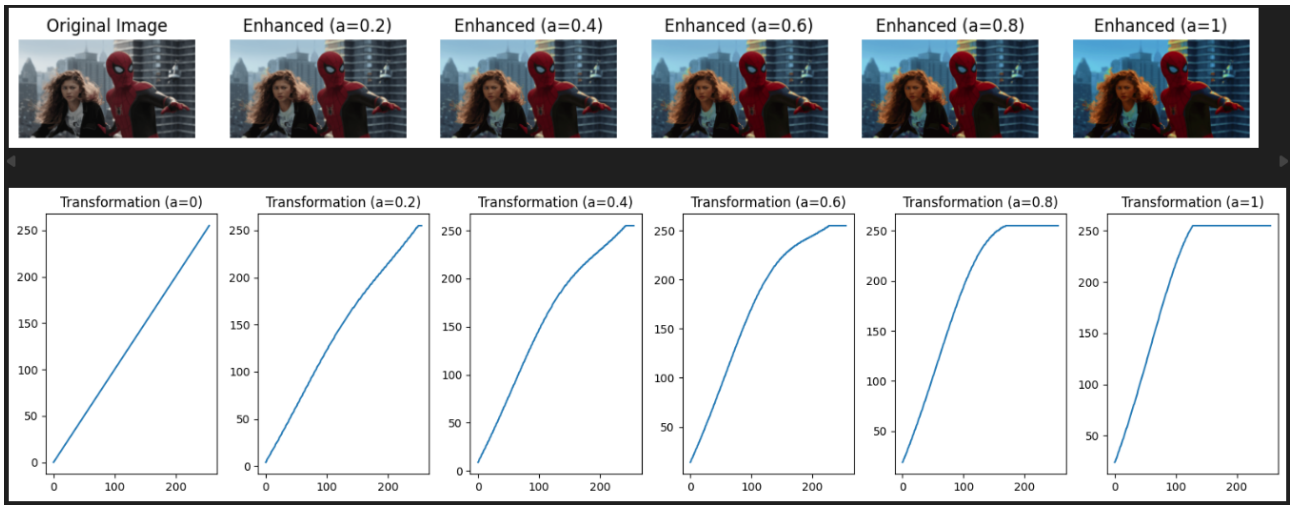


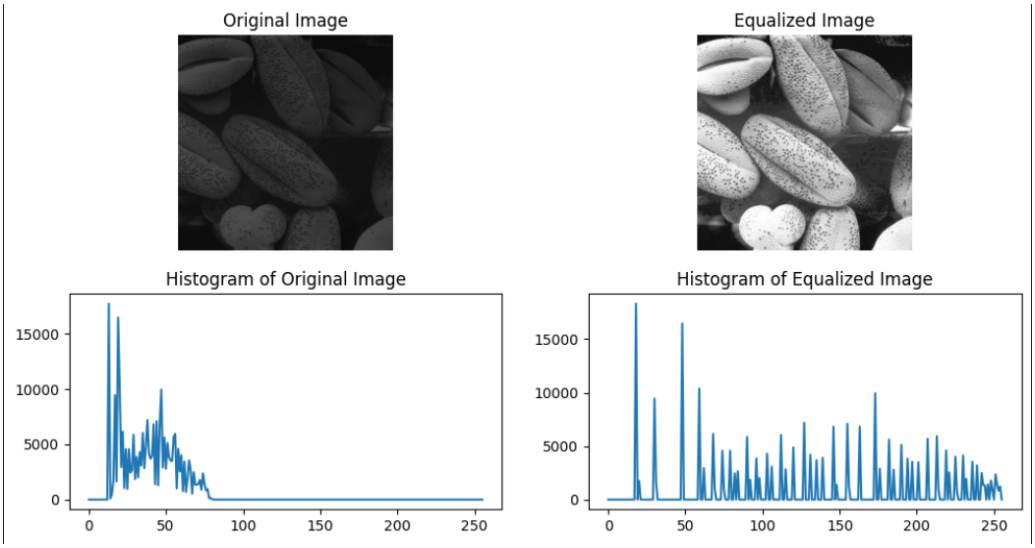
Figure 2: How different "a" values affect the vibrance of the image

The gaussian term in the equation gives the mid tones the strongest vibrancy boost compared to the high and ow extremes. Looking at the graphs it can be seen that when $a >= 0.6$ there appears to be a significant clipping of higher values indicating over-saturation. For these higher values($a >= 0.6$) the image appears to be unnatural (specially skin tones) **a = 0.4** gives the best balance in this case as there are no severe clippings and the image hasn't lost it's natural look (when a = 0.4, the sky has more blue colours and the skin tones are not over saturated)

To achieve this in the code I have first converted the image to HSV and then split it to separate channels. Then the transformation was added only to the saturation plane

```python
# (a) Split the image into hue, saturation, and value planes
hsv_image1 = cv.cvtColor(image, cv.COLOR_BGR2HSV)
h, s, v = cv.split(hsv_image)
#print(np.max(s), np.min(s))  # Check the range of saturation channel
fig_images, ax_images = plt.subplots(2, 3, figsize=(13, 6))
fig_transformation, ax_transformation = plt.subplots(2, 3, figsize=(12, 7))

def vibrance_transformation(x, a, sigma=70):
  gaussian_part = a*128*np.exp(-((x-128)**2)/(2*sigma**2))
  return np.minimum(x + gaussian_part, 255).astype(np.uint8)

# (b) Apply the intensity transformation to the saturation plane
a_vals = [0,0.2,0.4,0.6,0.8,1]  # (c) Adjust a to get a visually pleasing output
s_transformed = [vibrance_transformation(s, a) for a in a_vals]
# (d) Recombine the three planes
index = 0
for (a_value, s_transformed_value) in zip(a_vals, s_transformed):
    hsv_transformed = cv.merge((h, s_transformed_value, v))
    # Convert back to BGR color space
    bgr_transformed = cv.cvtColor(hsv_transformed, cv.COLOR_HSV2BGR)
    # (e) Display the original image, vibrance-enhanced image, and the intensity
    transformation
    ax_images[index//3, index%3].imshow(cv.cvtColor(bgr_transformed, cv.COLOR_BGR2RGB))
    ax_images[index//3, index%3].set_title('Original Image' if index == 0 else f'Vibrance
    Enhanced (a={a_value})')
    # Plot the intensity transformation
    x = np.arange(0, 256)
    y = vibrance_transformation(x, a_value)
    ax_transformation[index//3, index%3].plot(x, y, label=f'a={a_value}')
    ax_transformation[index//3, index%3].set_title(f'Intensity Transformation (a={a_value})')
    index+=1
```

5 Question 05

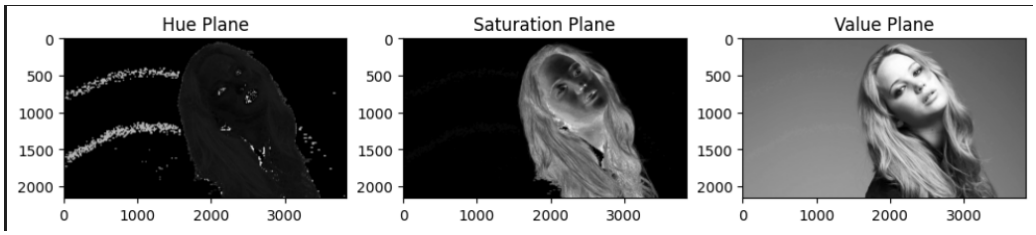


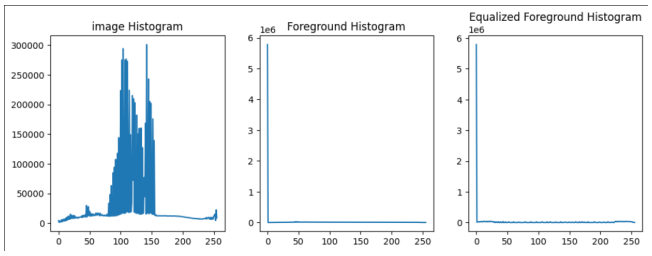
The original image is dark and hence the histogram is weighted towards the left side(it is skewed). Histogram equalization can be used to redistribute the pixel intensities in the image. The equation used is $T(k) = \frac{(L-1)}{M \cdot N} \sum_{j=0}^{k} n_j$ where MN is the total no. of pixels and L is the number of levels. This uses the CDF of the original histogram and calculates the equalized histogram transformation. After the equalization process we can see that the image is brighter and the intensities are spread in the full span of the 0-255 range. In the intensity mapping several intensities can get mapped to a single value and some intensities in the output may not exist as we see in the equalized histogram (the input has many unused bins in the high intensity range)

```python
image = cv.imread('a1images/shells.tif', cv.IMREAD_GRAYSCALE)
M,N = image.shape[:2]
bits_per_pixel = 8
L = 2**bits_per_pixel
fig, ax = plt.subplots(2, 2, figsize=(12, 12))

def equalize_histogram(image):
```

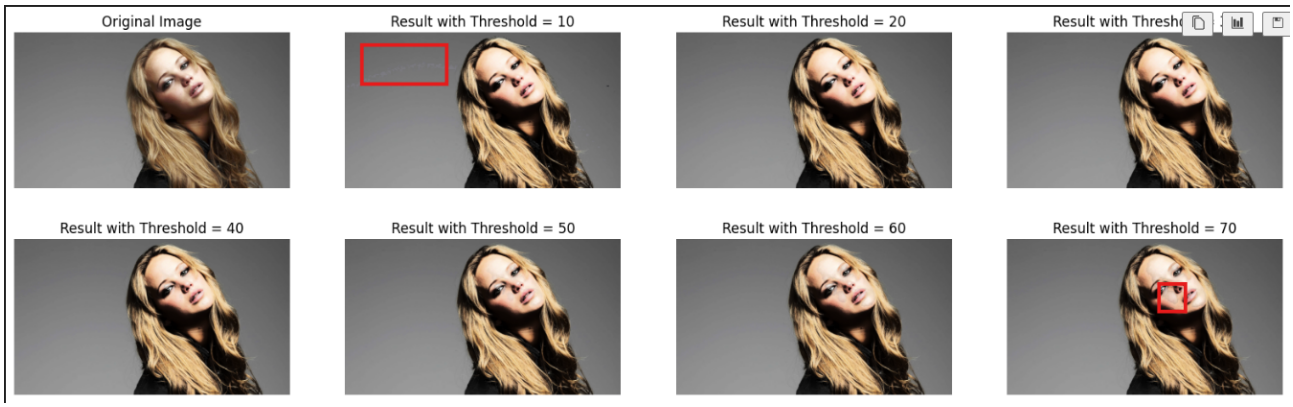
```
  """  Function to perform histogram equalization on the input image."""
  hist , bins = np.histogram(image.flatten(), bins=L, range=(0, L-1))
  cdf = hist.cumsum()  # Cumulative distribution function
  t = np.array([(L-1)/(M*N)*cdf[i] for i in range(256)], dtype=np.uint8)
  #t = [(L-1)/(M*N)*cdf[i] for i in np.arange(0,256)].astype(unit8)
  return (t, hist, cdf)

t, hist, cdf = equalize_histogram(image)
g = t[image]
```

6 Question 06





(b)After dividing the image into H,S,V channels, it is clear that the saturation plane is the most appropriate to threshold in order to get the foreground mask. This reason is that the saturation measures the purity of colour so it can be used to distinguish between the colourful foreground and the de-saturated background. The mask is created thresholding this plane. (c)The mask is applied to the value plane(bitwise operations) in order to extract the foreground values (d,e)the the histogram of the image is obtained and the transformation function is taken. After that, the equalization is applied to the foreground (f) Finally the background is extracted using the inverse of the mask and the two are combined to get the result.
Looking at the histograms we can see that the forground has a lot of values that are 0 due to the bitwise operation with the mask. After equalization the 0 values stay at zero(background is not changed) while the other values have re-distributed.



By comparing the images we can see that when the threshold is too low (10), parts of the background also gets selected(shown in the red box) Furthermore when the threshold is too high(60 or 70) then parts of the foreground is not selected, resulting in those pixels not getting equalized(shown in the red box). Considering these, the best threshold for this image is around 30-40.

```
# (a)Split the image into hue, saturation, and value planes
M,N = image.shape[:2]
bits_per_pixel = 8
L = 2**bits_per_pixel
hsv_image = cv.cvtColor(image, cv.COLOR_BGR2HSV)
h, s, v = cv.split(hsv_image)
```

```
threshold_values = [10,20,30,40,50,60,70]
fig_img, ax_img = plt.subplots(2, 4, figsize=(20, 6))
for index, threshold_value in enumerate(threshold_values,1):
  # (b) Select the appropriate plane to threshold in extract the foreground mask.
  th, mask = cv.threshold(s, threshold_value, 255, cv.THRESH_BINARY)  # Thresholding the
    saturation plane
  # (c) Now obtain the foreground only using cv.bitwise_and and compute the histogram.
  foreground = cv.bitwise_and(v, v, mask=mask)
  hist_v, bins_v = np.histogram(v.flatten(), bins=256, range=(0, 255))
  # (d) Obtain the cumulative sum of the histogram using np.cumsum.
  cdf = np.cumsum(hist_v)
  # (e) Use the formulas in slides to histogram-equalize the foreground.
  t = np.array([(L-1)/(M*N)*cdf[i] for i in range(256)], dtype=np.uint8)
  foreground_eq = t[foreground]
  hist_foreground_eq, bins_foreground_eq = np.histogram(foreground_eq.flatten(), bins=256,
    range=(0, 255))
  background = cv.bitwise_and(v, v, mask=cv.bitwise_not(mask))
  result_v = cv.add(background, foreground_eq)
  result_hsv = cv.merge((h, s, result_v))
  result_bgr = cv.cvtColor(result_hsv, cv.COLOR_HSV2BGR)
```

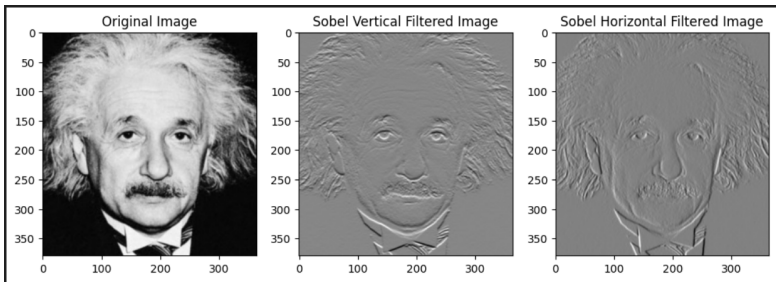
7 Question 07

(a)Using the cv.filter2D command

```
# (a) Using the existing filter2D to Sobel filter the image.
sobel_vertical = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
sobel_horizontal = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
image_sobel_vertical = cv.filter2D(image, cv.CV_32F, sobel_vertical)
image_sobel_horizontal = cv.filter2D(image, cv.CV_32F, sobel_horizontal)
```

(b)Using my own function

```
#(b) Write your own code to Sobel filter the image.
def sobel_filter(image, kernel):
    """Apply Sobel filter to the image using the given kernel."""
    height, width = image.shape
    result = np.zeros_like(image, dtype=np.float32)

    #pad the image to handle borders using zero padding
    padded_image = np.pad(image, ((1, 1), (1, 1)), mode='constant', constant_values=0)
    for y in range(height):
        for x in range(width):
            # Apply the kernel
            padded_image_region = padded_image[y:y+3, x:x+3]
            result[y, x] = np.sum(padded_image_region * kernel)
    return result

image_sobel_vertical = sobel_filter(image, sobel_vertical)
image_sobel_horizontal = sobel_filter(image, sobel_horizontal)
```

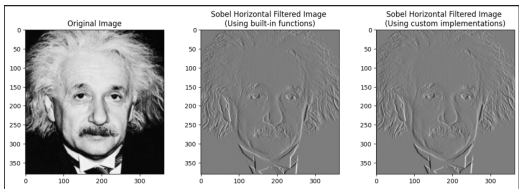


in both part (a) and part (b) I got similar results as shown on the left side. Sobel filter is used for edge detection of an image using intensity gradients. As shown in the figure, using the vertical and horizontal Sobel kernels we are able to detect the horizontal and vertical edges of the given image respectively. Furthermore the Sobel kernel we used are weighted, which combines edge detection with noise suppression of the image.

(c)Use the given property

```
vertical_kernal = np.array([1,2,1], dtype=np.float32)
horizontal_kernal = np.array([1,0,-1], dtype=np.float32)

#compute using cv.filter2D
temp_image = cv.filter2D(image, cv.CV_32F, vertical_kernal.reshape(3, 1))
sobel_image = cv.filter2D(temp_image, cv.CV_32F, horizontal_kernal.reshape(1, 3))
```

```
ax[1].imshow(sobel_image, cmap='gray', vmin=0, vmax=255)
ax[1].set_title('Sobel Horizontal Filtered Image \n(Using built-in functions)')

#compute using custom function
height, width = image.shape
padded_image = np.pad(image, ((1, 1), (0, 0)), mode='constant', constant_values=0)
result = np.zeros_like(image, dtype=np.float32)
for y in range(height):
    for x in range(width):
        # Apply the vertical kernel
        padded_image_region = padded_image[y:y+3, x+1:x+2]
        result[y,x] = np.sum(padded_image_region * vertical_kernal.reshape(3, 1))

padded_result = np.pad(result, ((0, 0), (1, 1)), mode='constant', constant_values=0)
for y in range(height):
    for x in range(width):
        # Apply the horizontal kernel
        padded_image_region = padded_result[y+1:y+2, x:x+3]
        result[y, x] = np.sum(padded_image_region * horizontal_kernal.reshape(1, 3))
```



Here I have done the Sobel filtering by separating the 2D kernel into two 1D kernels. The results are the same as before. (the flipping of the result is due to the given kernel being flipped from the ne used for part (a) and (b)). The computational efficiency of this is high compared to using the 2D kernel(Complexity reduces from $O(n^2m^2)$ to $O(n^2m)$

8 Question 08

```
def zoom_image_custom(image, factor,method):
    """Zoom the image by a given factor using nearest-neighbor or bilinear interpolation."""
    if factor <= 0 or factor > 10:
        raise ValueError("Scale factor must be in (0, 10]")
    height, width = image.shape[:2]
    new_height = int(height * factor)
    new_width = int(width * factor)
    result = np.zeros((new_height, new_width), dtype=image.dtype)
    if method == 'nearest':
        for y in range(new_height):
            for x in range(new_width):
                orig_y,orig_x = int(y / factor),int(x / factor)
                result[y, x] = image[min(orig_y,height-1), min(orig_x,width-1)]
    elif method == 'bilinear':
        for y in range(new_height):
            for x in range(new_width):
                orig_y = y / factor
                orig_x = x / factor
                #get the four surrounding pixels
                x1,x2,y1,y2 = int(np.floor(orig_x)),int(np.ceil(orig_x)),int(np.floor(orig_y))
    ,int(np.ceil(orig_y))
                x2,y2 = min(x2, width - 1),min(y2, height - 1)
                dx,dy = (orig_x - x1),(orig_y - y1)    #calculate the gap
                # Perform bilinear interpolation
                result[y, x] = (image[y1, x1] * (1 - dx) * (1 - dy) +
                                image[y1, x2] * dx * (1 - dy) +
                                image[y2, x1] * (1 - dx) * dy +
                                image[y2, x2] * dx * dy)
        result = np.clip(result, 0, 255).astype(image.dtype)  # Ensure pixel values are in
    valid range
    return result
def normalized_ssd(img1, img2):
    """Compute the normalized sum of squared differences (SSD) between two images."""
    # Get minimum dimensions
    h = min(img1.shape[0], img2.shape[0])
    w = min(img1.shape[1], img2.shape[1])
    # Crop both images to match
    img1_cropped = img1[:h, :w]
    img2_cropped = img2[:h, :w]
    diff = img1_cropped.astype(np.float32) - img2_cropped.astype(np.float32)
    ssd = np.sum(diff**2)
    return ssd / (img1_cropped.shape[0] * img1_cropped.shape[1])
```

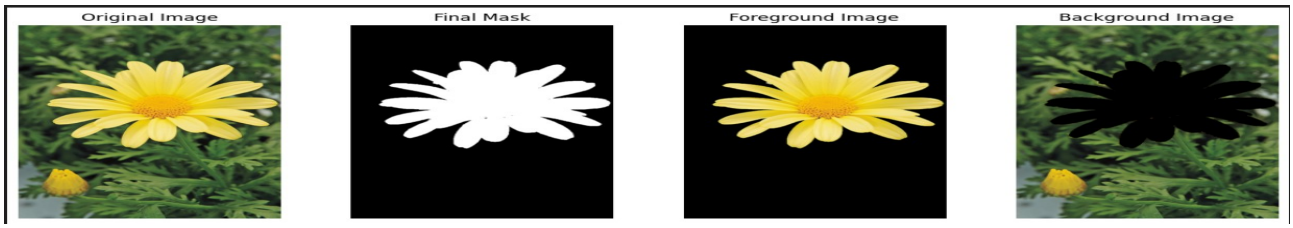
In the code I have implemented the nearest neighbor and bilinear interpolation algorithms to obtain the zoomed image. The SSD is divided by the total no. of pixels in the image to normalize it. The following are the results.

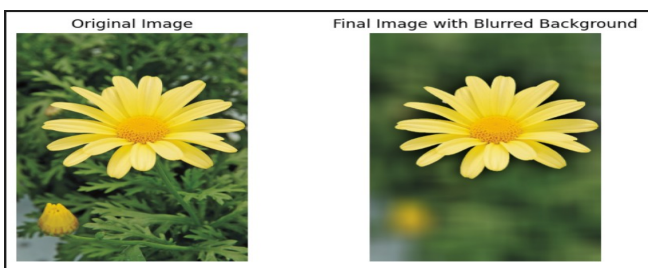
Image	SSD(Nearest)	SSD(Bilinear)
Image1	137.209	115.657
Image2	26.686	18.427
Image3	66.756	50.544
Image4	219.928	190.299
Image5	66.756	255.553

So we can see that the bilinear Interpolation method has given a lower normalized SSD compared to the nearest neighbor method providing a more accurate outcome. This is due to the fact that in the nearest neighbor method we simply get the pixel value of the nearest neighbor whereas in the bilinear method we calculate the distance to each neighbor and add weights to the neighboring intensities before getting a value. This helps in reducing jagged edges of the resulting image

9 Question 09

```python
image_rgb = cv.cvtColor(image, cv.COLOR_BGR2RGB)
height, width = image_rgb.shape[:2]
mask = np.zeros(image.shape[:2], dtype=np.uint8) #create an expty mask
foreground_model = np.zeros((1, 65), dtype=np.float64)
background_model = np.zeros((1, 65), dtype=np.float64)
rect = (0,height//10,int(width),int(height/1.5)) #(x, y, width, height) #ROI
cv.grabCut(image_rgb, mask, rect, background_model, foreground_model, 5, cv.GC_INIT_WITH_RECT)
final_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8') #background = 0
dimension_adjusted_mask = final_mask[:,:, np.newaxis]  # Adjust mask dimensions
#apply the mask to create the forground and background image
foreground_image = image_rgb * dimension_adjusted_mask
background_image = image_rgb * (1 - dimension_adjusted_mask)
kernal_blur = np.ones((71, 71), np.float32) / (71 * 71) #blur the background
blurred_background = cv.filter2D(background_image, cv.CV_32F, kernal_blur).astype(np.uint8)
# Apply the mask to the blurred background to remove the foreground areas of the blurred image
blurred_background = blurred_background* (1-dimension_adjusted_mask)
```





(c) When blurring the background it is a spatial operation rather than a point operation. When removing the flower we make the background black in that area. So when the kernel is near the edge of the flower, it is averaging the original background with the pure back area, resulting in a reduction in brightness in that area. Since the kernel is averaging a large no. of black pixels, the background just beyond the edge of the flower is quite dark.