**Department of Electronic and Telecommunication Engineering**

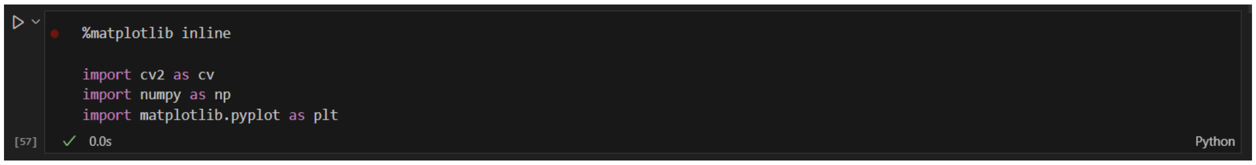
**University of Moratuwa**

EN3160 - Image Processing and Machine Vision

# Assignment 01

Ranasingha A.S.N.        200507N

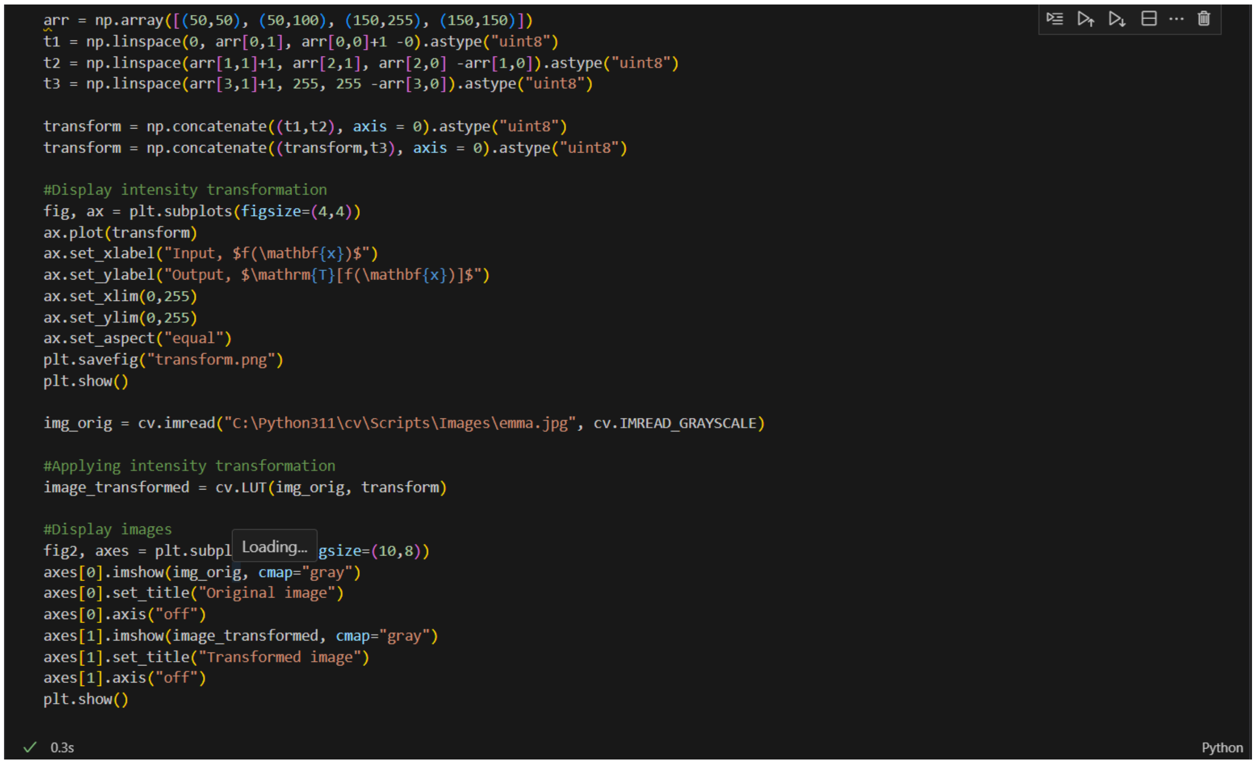
First necessary libraries should be imported.



```
%matplotlib inline

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

[57] ✓ 0.0s Python

## 01. Intensity transformation

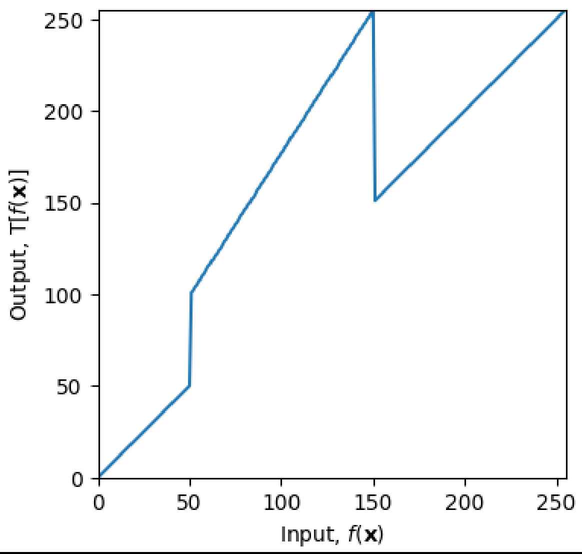


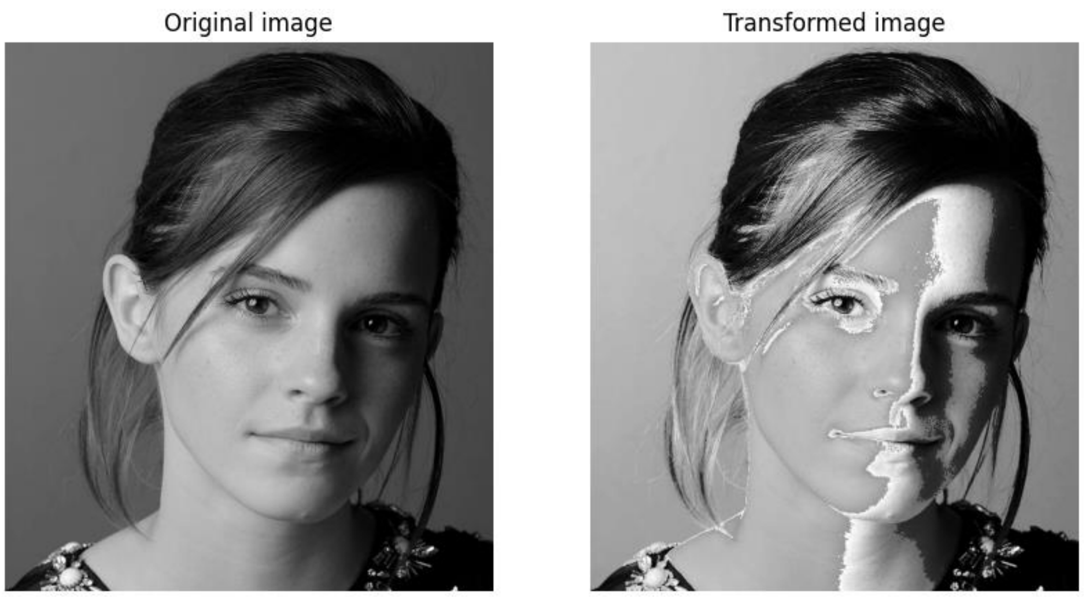
```
arr = np.array([(50,50), (50,100), (150,255), (150,150)])
t1 = np.linspace(0, arr[0,1], arr[0,0]+1 -0).astype("uint8")
t2 = np.linspace(arr[1,1]+1, arr[2,1], arr[2,0] -arr[1,0]).astype("uint8")
t3 = np.linspace(arr[3,1]+1, 255, 255 -arr[3,0]).astype("uint8")

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

#Display intensity transformation
fig, ax = plt.subplots(figsize=(4,4))
ax.plot(transform)
ax.set_xlabel("Input, $f(\mathbf{x})$")
ax.set_ylabel("Output, $\mathrm{T}[f(\mathbf{x})]$")
ax.set_xlim(0,255)
ax.set_ylim(0,255)
ax.set_aspect("equal")
plt.savefig("transform.png")
plt.show()

img_orig = cv.imread("C:\Python311\cv\Scripts\Images\emma.jpg", cv.IMREAD_GRAYSCALE)

#Applying intensity transformation
image_transformed = cv.LUT(img_orig, transform)

#Display images
fig2, axes = plt.subpl Loading... gsize=(10,8))
axes[0].imshow(img_orig, cmap="gray")
axes[0].set_title("Original image")
axes[0].axis("off")
axes[1].imshow(image_transformed, cmap="gray")
axes[1].set_title("Transformed image")
axes[1].axis("off")
plt.show()
```

✓ 0.3s Python

In the above code, the intensity transformation function is defined by concatenating t1, t2 and t3. Then that transformation is applied to the image.

Outputs:



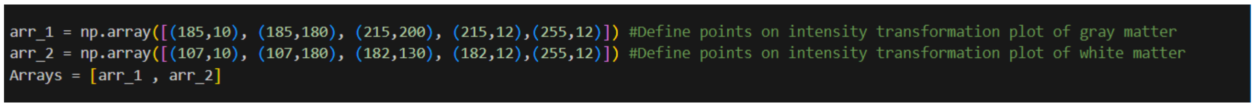


**02. Accentuating gray matter and white matter of brain proton slice.**

Here first I have defined two separate arrays to store important points on the intensity transformation of gray matter accentuation and white matter accentuation respectively. The points are adjusted considering which intensities should be enhanced and which intensities should be reduced.

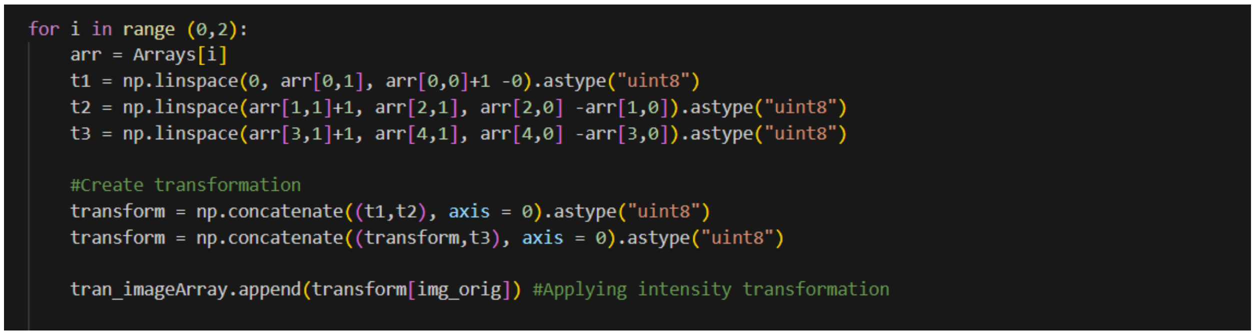
For accentuation of gray matter, the approximate intensity range correspond to gray matter of the original image are lifted to higher values, so that they look brighter, and rest of the intensities are set to lower values so that they look darker.

In the same manner, the intensities corresponding to white matter of the original image are lifted and the rest of the intensities are set to lower values.



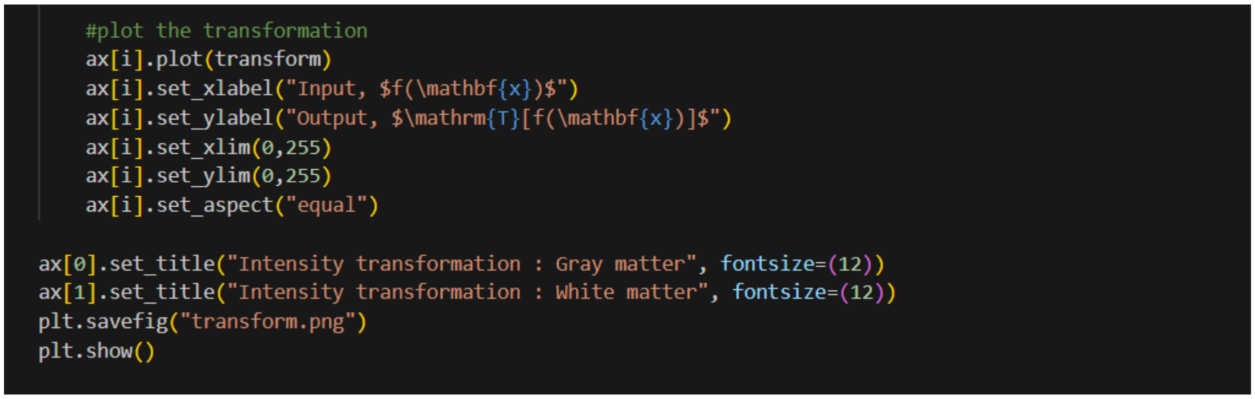
```
arr_1 = np.array([(185,10), (185,180), (215,200), (215,12),(255,12)]) #Define points on intensity transformation plot of gray matter
arr_2 = np.array([(107,10), (107,180), (182,130), (182,12),(255,12)]) #Define points on intensity transformation plot of white matter
Arrays = [arr_1 , arr_2]
```

Then transformation function for each of the two cases is created using the predefined points. Transformation is then applied to the original image.



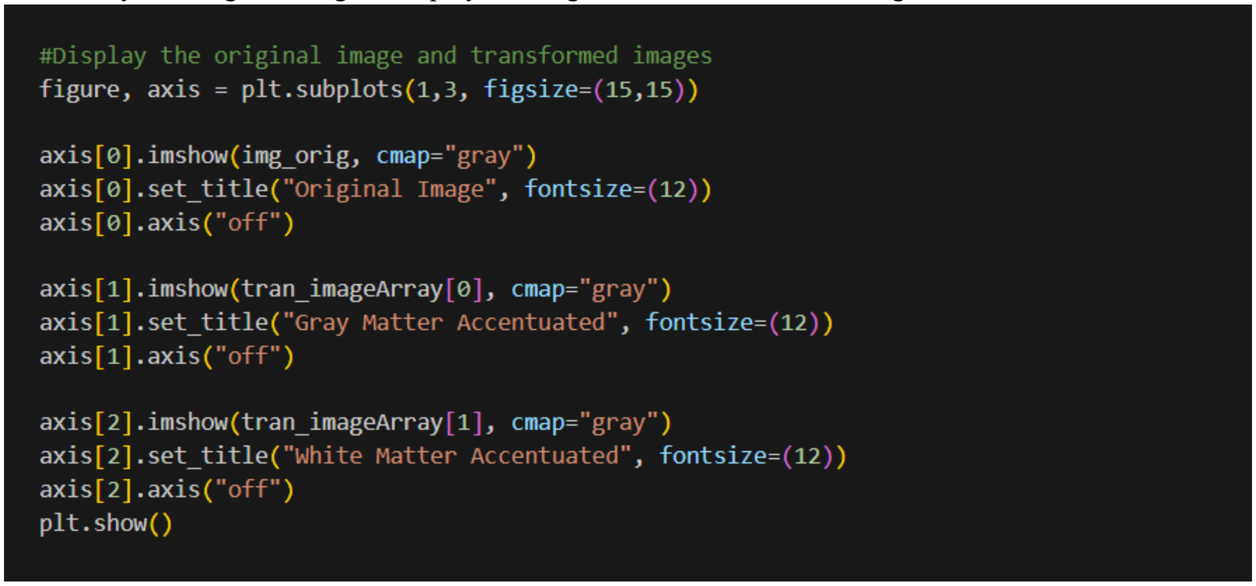
```
for i in range (0,2):
    arr = Arrays[i]
    t1 = np.linspace(0, arr[0,1], arr[0,0]+1 -0).astype("uint8")
    t2 = np.linspace(arr[1,1]+1, arr[2,1], arr[2,0] -arr[1,0]).astype("uint8")
    t3 = np.linspace(arr[3,1]+1, arr[4,1], arr[4,0] -arr[3,0]).astype("uint8")

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

    tran_imageArray.append(transform[img_orig]) #Applying intensity transformation
```

After that the two transformations are displayed using matplot.



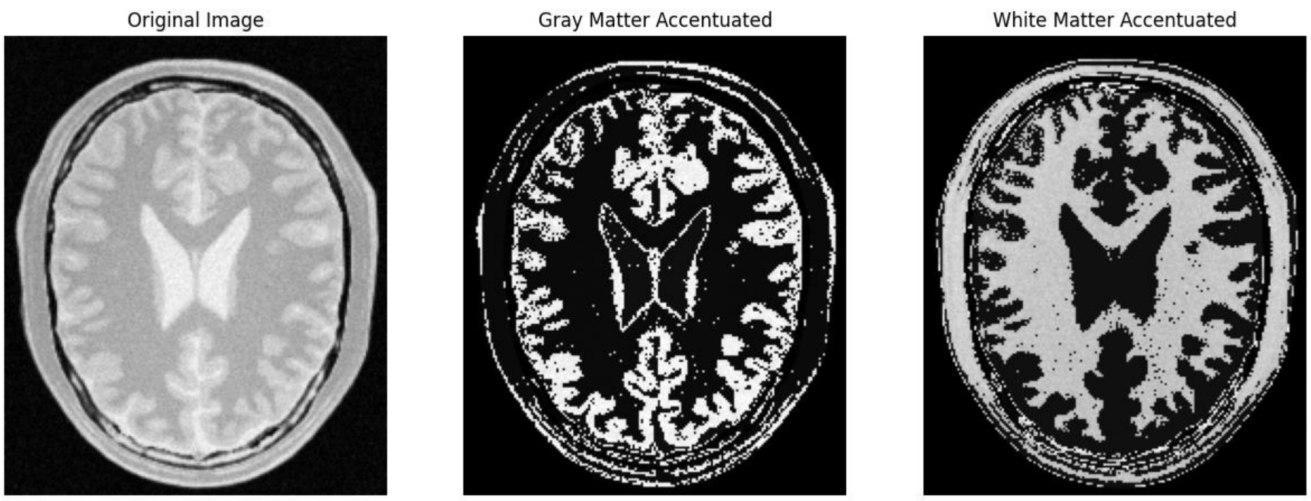
```
    #plot the transformation
    ax[i].plot(transform)
    ax[i].set_xlabel("Input, $f(\mathbf{x})$")
    ax[i].set_ylabel("Output, $\mathrm{T}[f(\mathbf{x})]$")
    ax[i].set_xlim(0,255)
    ax[i].set_ylim(0,255)
    ax[i].set_aspect("equal")

ax[0].set_title("Intensity transformation : Gray matter", fontsize=(12))
ax[1].set_title("Intensity transformation : White matter", fontsize=(12))
plt.savefig("transform.png")
plt.show()
```

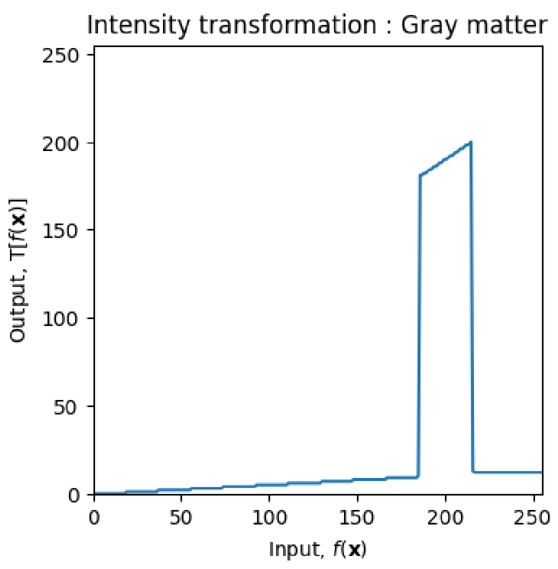
Finally, the original image is displayed along with two transformed images.

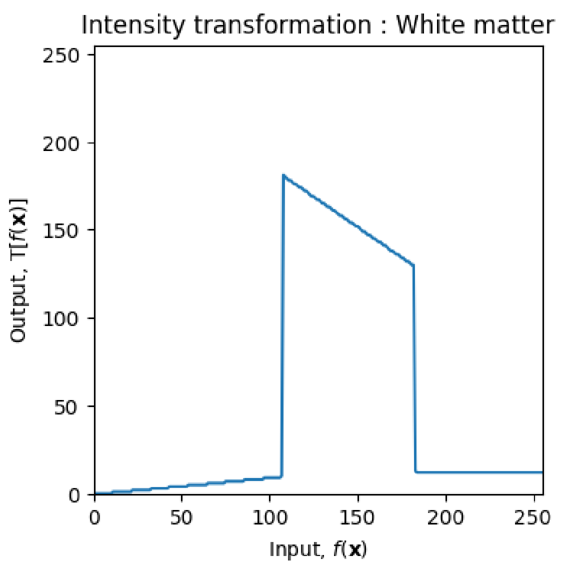


```
#Display the original image and transformed images
figure, axis = plt.subplots(1,3, figsize=(15,15))

axis[0].imshow(img_orig, cmap="gray")
axis[0].set_title("Original Image", fontsize=(12))
axis[0].axis("off")

axis[1].imshow(tran_imageArray[0], cmap="gray")
axis[1].set_title("Gray Matter Accentuated", fontsize=(12))
axis[1].axis("off")

axis[2].imshow(tran_imageArray[1], cmap="gray")
axis[2].set_title("White Matter Accentuated", fontsize=(12))
axis[2].axis("off")
plt.show()
```

The result of the code is as follows.







**03. Gamma Correction**

First, to see how the gamma correction works for different values of gamma, ($\gamma > 1, \gamma < 1$ and $\gamma = 1$), an array containing gamma values covering interested regions are defined.

```
gamma = [0.4,0.8,1,1.2,1.6] #array of gamma values
```

To apply the gamma correction to L plane, the image is split into L*a*b planes.

```
img_orig = cv.cvtColor(img_orig, cv.COLOR_BGR2LAB) # convert image to LAB color space
L_channel, a_channel, b_channel = cv.split(img_orig) # extracting 3 channels separately
img_orig = cv.cvtColor(img_orig, cv.COLOR_LAB2RGB)
```

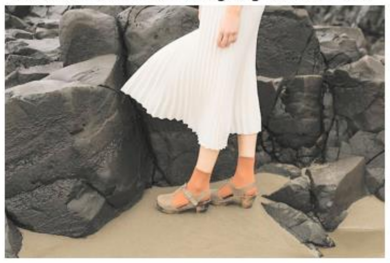
The original image is converted back to RGB format to display it using matplot.
Then a lookup table is created and then gamma correction is done by looking at the corresponding values for L plane using the created lookup table.

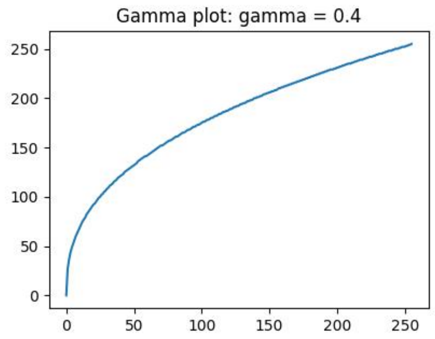
```
for i in range (0,len(gamma)):
    LookUp = np.array([(p/255)**(gamma[i])*255 for p in range(0, 256)]).astype(np.uint8)
    L_channel_gamma_corrected = cv.LUT(L_channel, LookUp) #applying gammma correction to L channel
    img_gamma =cv.merge([L_channel_gamma_corrected, a_channel, b_channel]) # merge the three channels after correction
    img_gamma = cv.cvtColor (img_gamma, cv.COLOR_LAB2RGB)
    image_array.append(img_gamma) #array of gamma corrected images

    #display gamma corrected image
    axes[i,0].imshow(img_gamma)
    axes[i,0].set_title("Gamma corrected image: gamma = " +str(gamma[i]), fontsize = 12)
    axes[i,0].axis("off")
    #display gamma correction plot
    axes[i,1].plot(LookUp)
    axes[i,1].set_title("Gamma plot: gamma = " +str(gamma[i]), fontsize = 12)
```

The outputs are as follows.

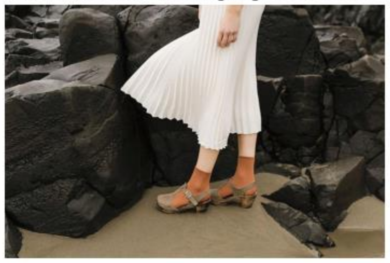
Original image

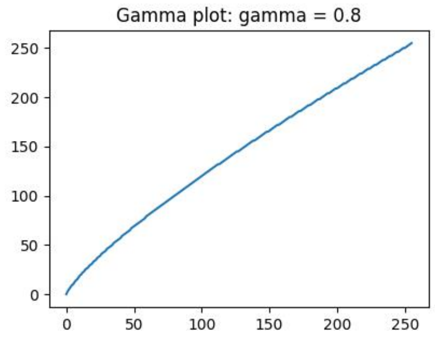
Gamma corrected image: gamma = 0.4



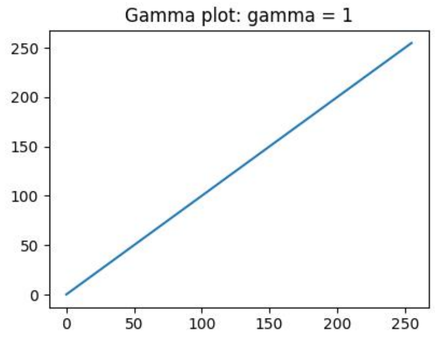


Gamma corrected image: gamma = 0.8

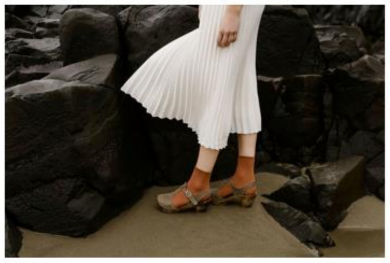


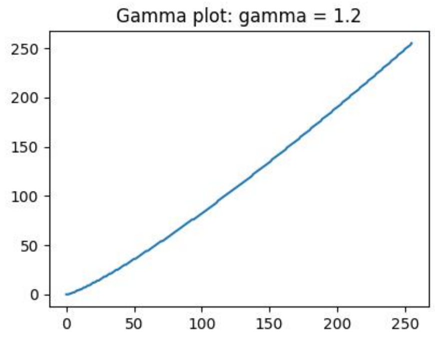


Gamma corrected image: gamma = 1

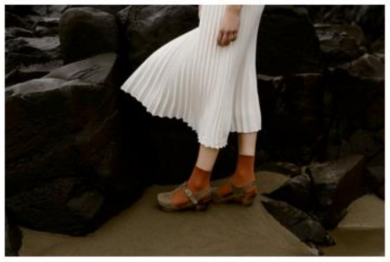


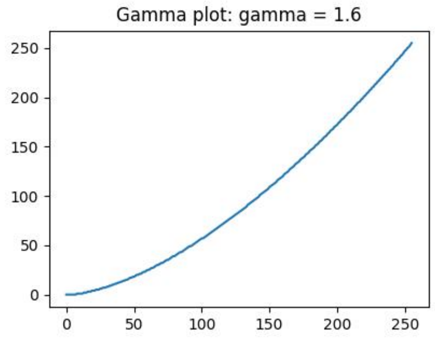
Gamma corrected image: gamma = 1.2



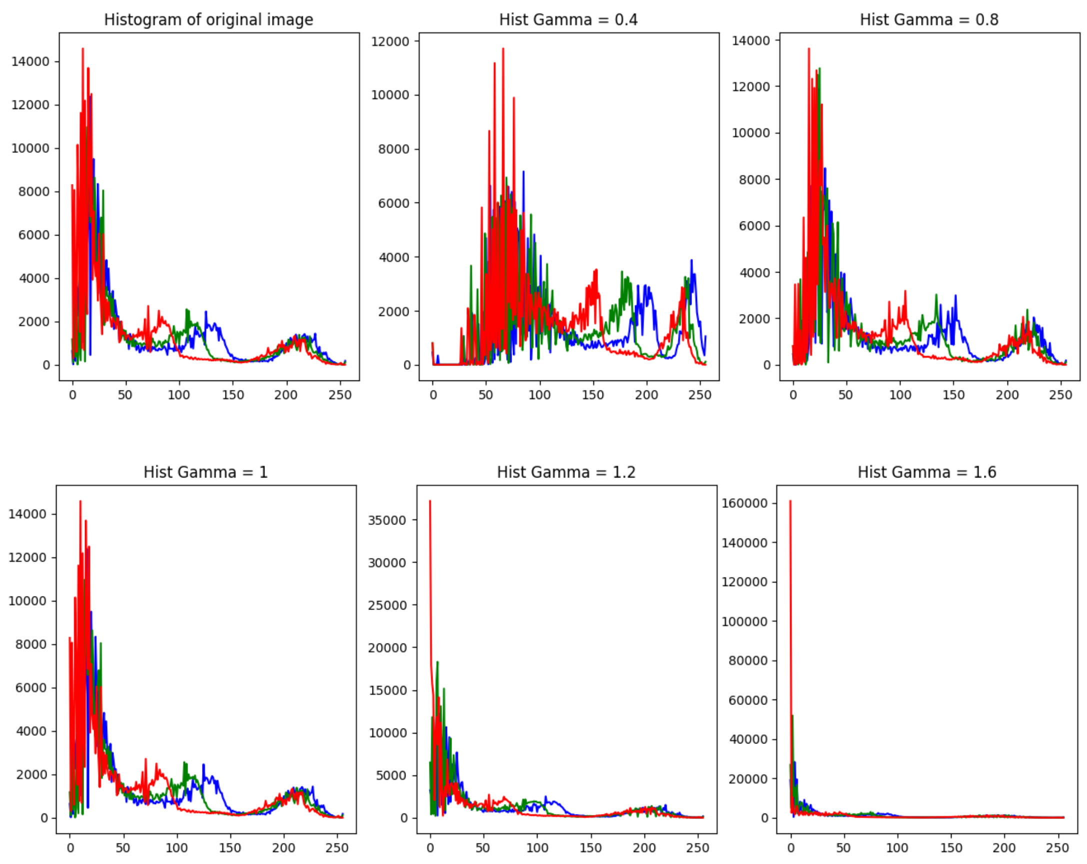


Gamma corrected image: gamma = 1.6



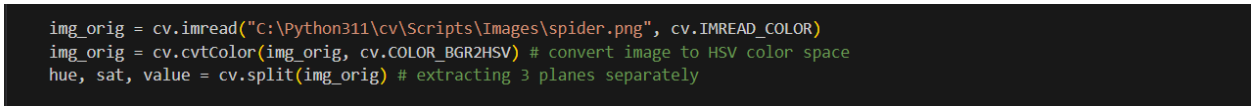


Finally, the histograms for the original image and each gamma corrected image are calculated.



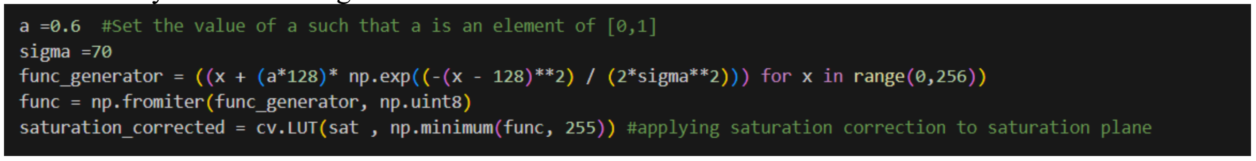
## 04. Increasing vibrance of an image

Here we are applying the intensity transformation to the saturation plane of the image. First the image is split into HSV planes, Hue, Saturation and Value.



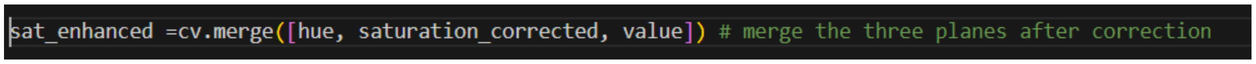
```
img_orig = cv.imread("C:\Python311\cv\Scripts\Images\spider.png", cv.IMREAD_COLOR)
img_orig = cv.cvtColor(img_orig, cv.COLOR_BGR2HSV) # convert image to HSV color space
hue, sat, value = cv.split(img_orig) # extracting 3 planes separately
```

Then intensity transformation is defined, and the saturation plane is updated. No change is done to hue and value planes. In the code, I have set the value of variable a to 0.6, but it can be adjusted as necessary within the range 0 to 1.



```
a =0.6  #Set the value of a such that a is an element of [0,1]
sigma =70
func_generator = ((x + (a*128)* np.exp((-(x - 128)**2) / (2*sigma**2))) for x in range(0,256))
func = np.fromiter(func_generator, np.uint8)
saturation_corrected = cv.LUT(sat , np.minimum(func, 255)) #applying saturation correction to saturation plane
```

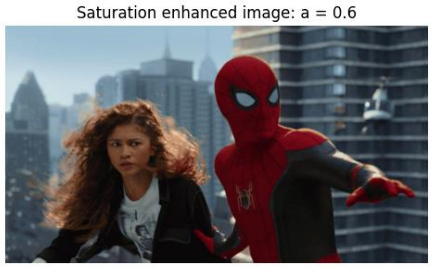
Finally, the updated saturation plane and previous hue and value planes are merged together, and saturation enhanced image is displayed along with the original image and the intensity transformation.

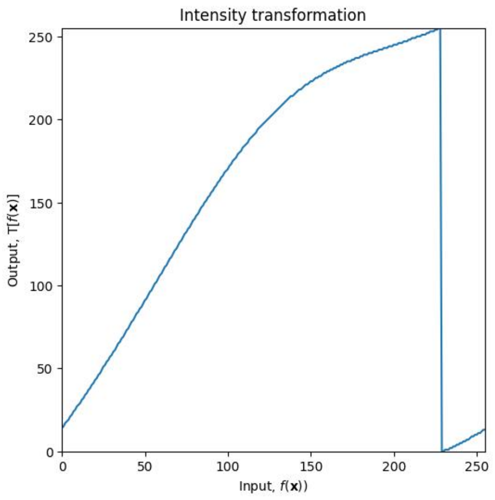


```python
sat_enhanced =cv.merge([hue, saturation_corrected, value]) # merge the three planes after correction
```

The outputs are as follows.

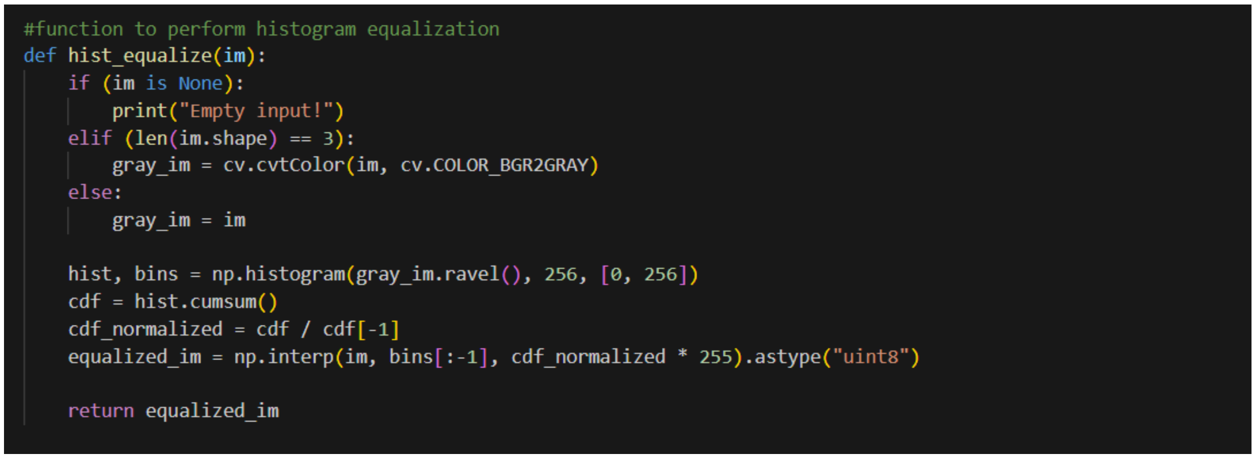
Original image


Saturation enhanced image: a = 0.6



## 05. Histogram equalization

Following is the function defined to perform histogram equalization for an input image.



```python
#function to perform histogram equalization
def hist_equalize(im):
    if (im is None):
        print("Empty input!")
    elif (len(im.shape) == 3):
        gray_im = cv.cvtColor(im, cv.COLOR_BGR2GRAY)
    else:
        gray_im = im

    hist, bins = np.histogram(gray_im.ravel(), 256, [0, 256])
    cdf = hist.cumsum()
    cdf_normalized = cdf / cdf[-1]
    equalized_im = np.interp(im, bins[:-1], cdf_normalized * 255).astype("uint8")

    return equalized_im
```

First an image was imported, and its histogram was calculated. Then the histogram and the cdf of original image were displayed.

```python
# Calculate histogram of original image
hist, bins = np.histogram(image.ravel(), 256, [0, 256])
cdf = hist.cumsum()
normalized_cdf = cdf * hist.max() / cdf.max()

# Display histogram and cdf of original image
plt.figure(figsize=(8, 3))
plt.subplot(121)
plt.plot(normalized_cdf, color="b")
plt.hist(image.flatten(), 256, [0, 256], color="g")
plt.xlim([0, 256])
plt.legend(("cdf", "histogram"), loc="upper right")
plt.title("Histogram of the Original Image")
```

Then histogram equalization is performed on the input image by calling the defined function above.

```python
equ = hist_equalize(image)  # Perform histogram equalization
```

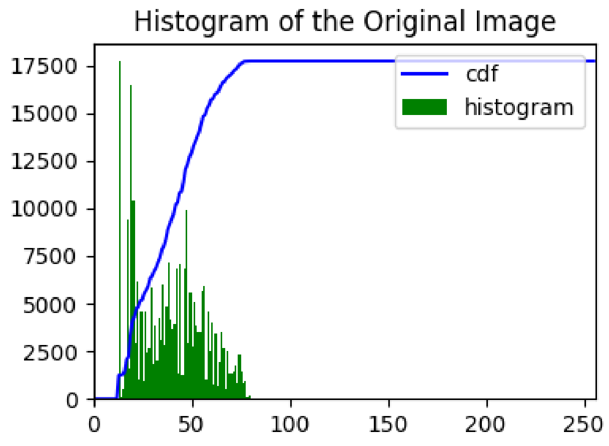
Here the histogram equalized image is assigned to the variable equ. Next, histogram calculation and displaying is done.

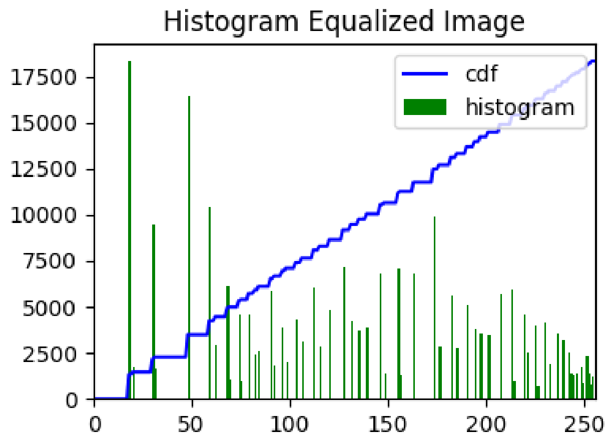
```python
# Calculate histogram of histogram equalized image
hist, bins = np.histogram(equ.ravel(), 256, [0, 256])
cdf = hist.cumsum()
normalized_cdf = cdf * hist.max() / cdf.max()

# Display histogram and cdf of histogram equalized image
plt.subplot(122)
plt.plot(normalized_cdf, color="b")
plt.hist(equ.flatten(), 256, [0, 256], color="g")
plt.xlim([0, 256])
plt.legend(("cdf", "histogram"), loc="upper right")
plt.title("Histogram Equalized Image")

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

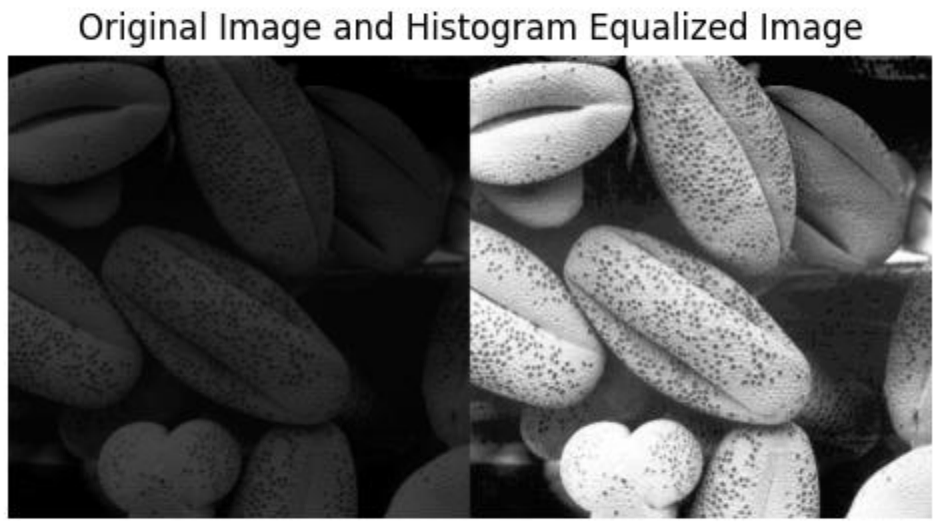
Finally, the original image and the histogram equalized images are displayed.

```python
# Display original image and histogram equalized image
res = np.hstack((image, equ))
plt.figure(figsize=(8, 3))
plt.axis("off")
plt.title("Original Image and Histogram Equalized Image")
plt.imshow(res, cmap="gray")
plt.show()
```

Here are the outputs of the above.

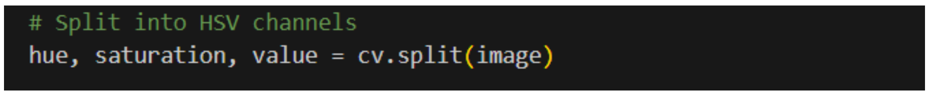






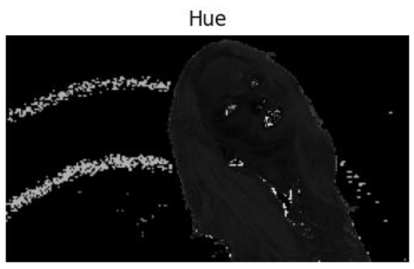
**06. Applying histogram equalization to foreground of an image.**

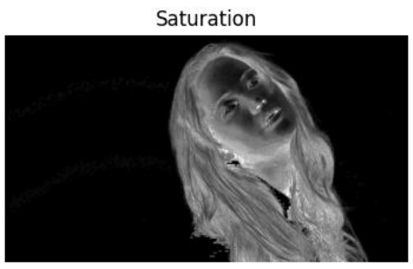
First, we split the image to HSV planes.

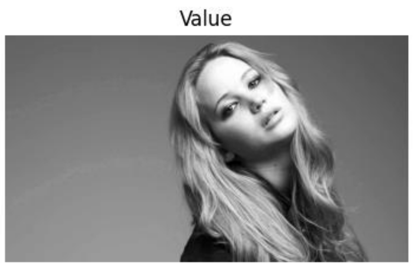


```
# Split into HSV channels
hue, saturation, value = cv.split(image)
```

The extracted saturation, hue and vales planes of the image are as follows.



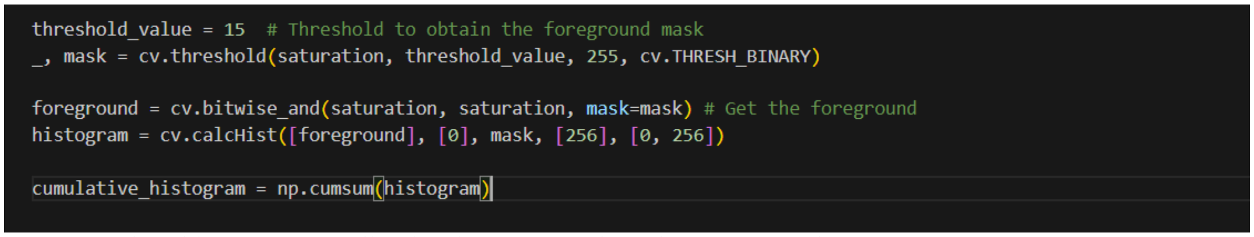




Then a foreground mask is extracted using the saturation plane with threshold set to 15.

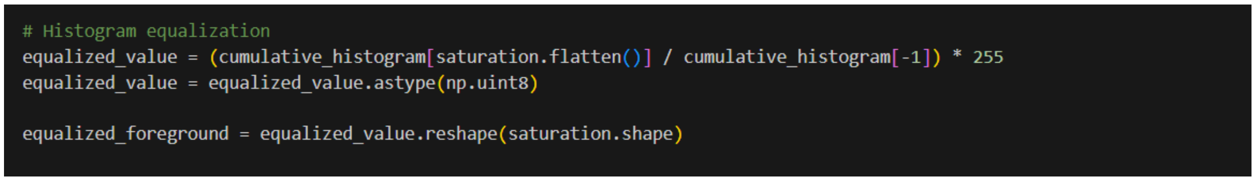
Here threshold was selected as 15 because, for the values above 15, mask was unable to extract some considerably bigger parts of the foreground and for the values above 15, mask included some background parts as well. For threshold equals to 15, mask was better.

Then histogram is calculated.



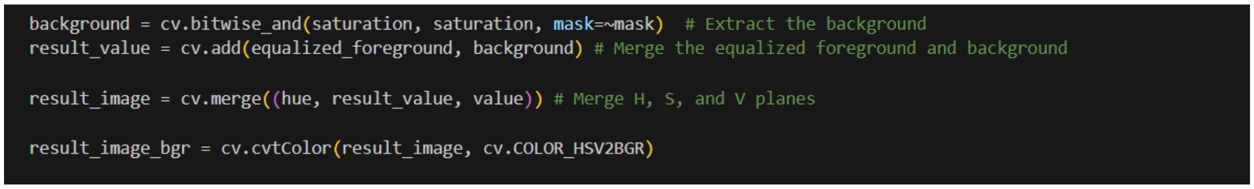
```
threshold_value = 15  # Threshold to obtain the foreground mask
_, mask = cv.threshold(saturation, threshold_value, 255, cv.THRESH_BINARY)

foreground = cv.bitwise_and(saturation, saturation, mask=mask) # Get the foreground
histogram = cv.calcHist([foreground], [0], mask, [256], [0, 256])

cumulative_histogram = np.cumsum(histogram)
```

After that, histogram equalization is performed on the foreground.



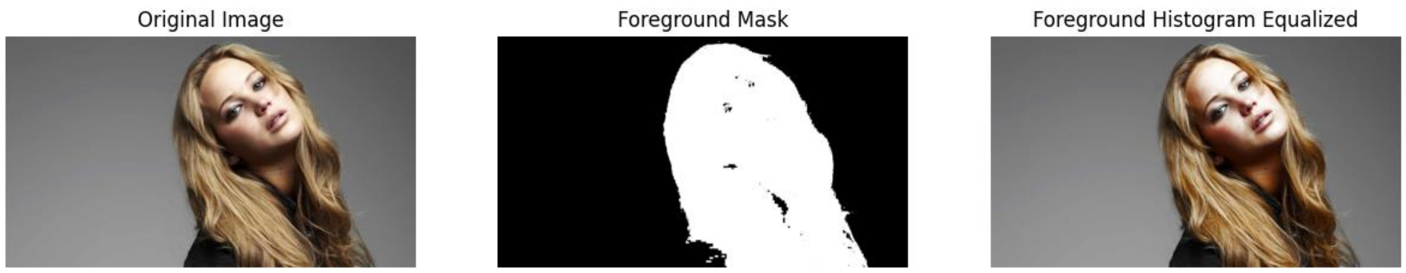
```
# Histogram equalization
equalized_value = (cumulative_histogram[saturation.flatten()] / cumulative_histogram[-1]) * 255
equalized_value = equalized_value.astype(np.uint8)

equalized_foreground = equalized_value.reshape(saturation.shape)
```

Then background is also extracted, and it is merged with histogram equalized foreground. Since the image is in HSV form, H, S and V planes are merged again to form the image. Image is then converted to RGB format to display it using matplot.



```
background = cv.bitwise_and(saturation, saturation, mask=~mask)  # Extract the background
result_value = cv.add(equalized_foreground, background) # Merge the equalized foreground and background

result_image = cv.merge((hue, result_value, value)) # Merge H, S, and V planes

result_image_bgr = cv.cvtColor(result_image, cv.COLOR_HSV2BGR)
```

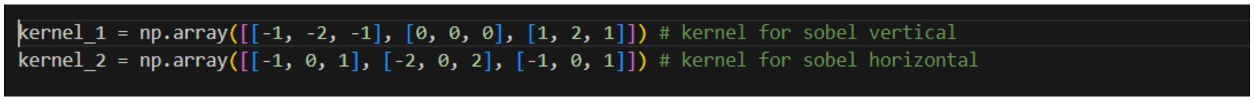
The results are as follows.



**07. Sobel filter.**

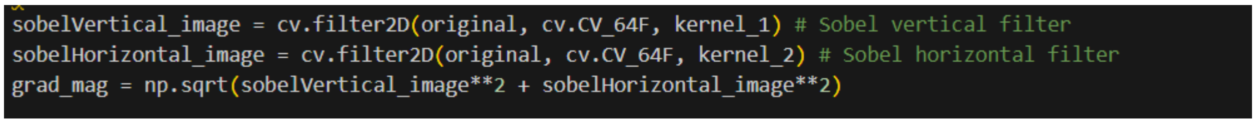
a. First an image is imported (in this case, I have imported in grayscale) and then performed Sobel vertical filter and Sobel horizontal filter and then computed the gradient magnitude. Sobel filter s used for edge detection.

The kernels corresponding to Sobel vertical and Sobel horizontal are defined at the beginning.



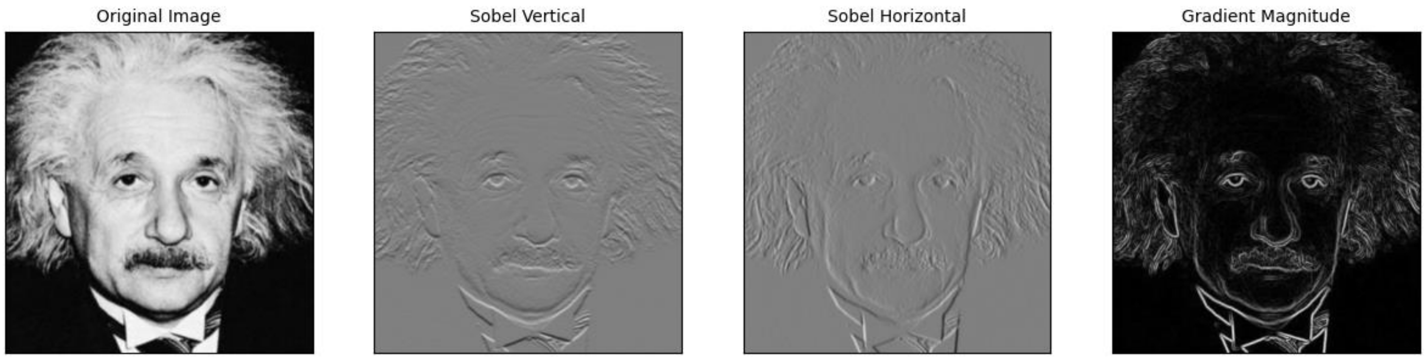
```
kernel_1 = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]]) # kernel for sobel vertical
kernel_2 = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]) # kernel for sobel horizontal
```

Then the filtering process was done by using the function, filter2D in OpenCV and then gradient magnitude was also computed.

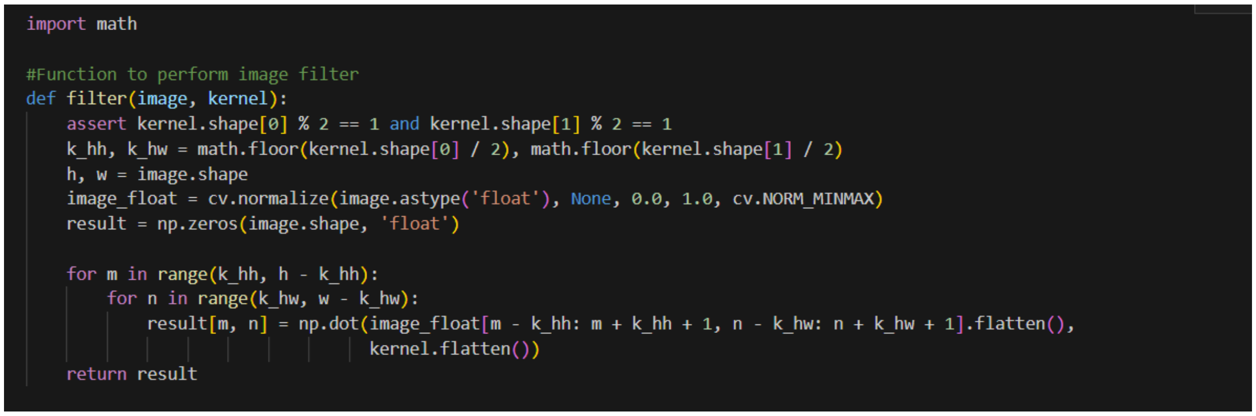


```
sobelVertical_image = cv.filter2D(original, cv.CV_64F, kernel_1) # Sobel vertical filter
sobelHorizontal_image = cv.filter2D(original, cv.CV_64F, kernel_2) # Sobel horizontal filter
grad_mag = np.sqrt(sobelVertical_image**2 + sobelHorizontal_image**2)
```

The final outputs of the code are as follows.

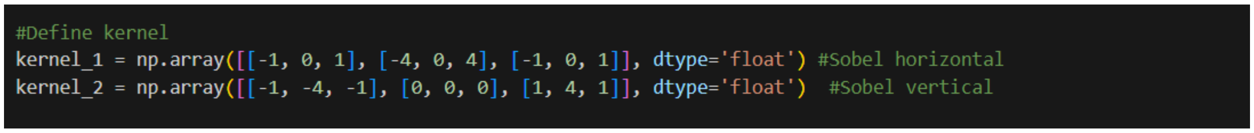


b. The function “filter” is defined to perform image filtering using a custom kernel. It takes two arguments, original image and the kernel of interest and returns the filtered image.



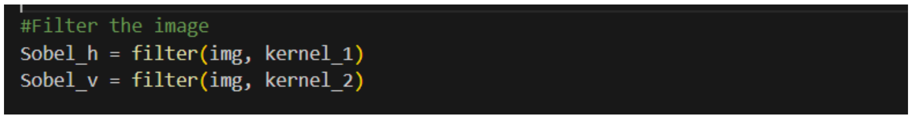
```
import math

#Function to perform image filter
def filter(image, kernel):
    assert kernel.shape[0] % 2 == 1 and kernel.shape[1] % 2 == 1
    k_hh, k_hw = math.floor(kernel.shape[0] / 2), math.floor(kernel.shape[1] / 2)
    h, w = image.shape
    image_float = cv.normalize(image.astype('float'), None, 0.0, 1.0, cv.NORM_MINMAX)
    result = np.zeros(image.shape, '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_float[m - k_hh: m + k_hh + 1, n - k_hw: n + k_hw + 1].flatten(),
                                  kernel.flatten())
    return result
```

Then the kernels for Sobel vertical filter and Sobel horizontal filter are defined.



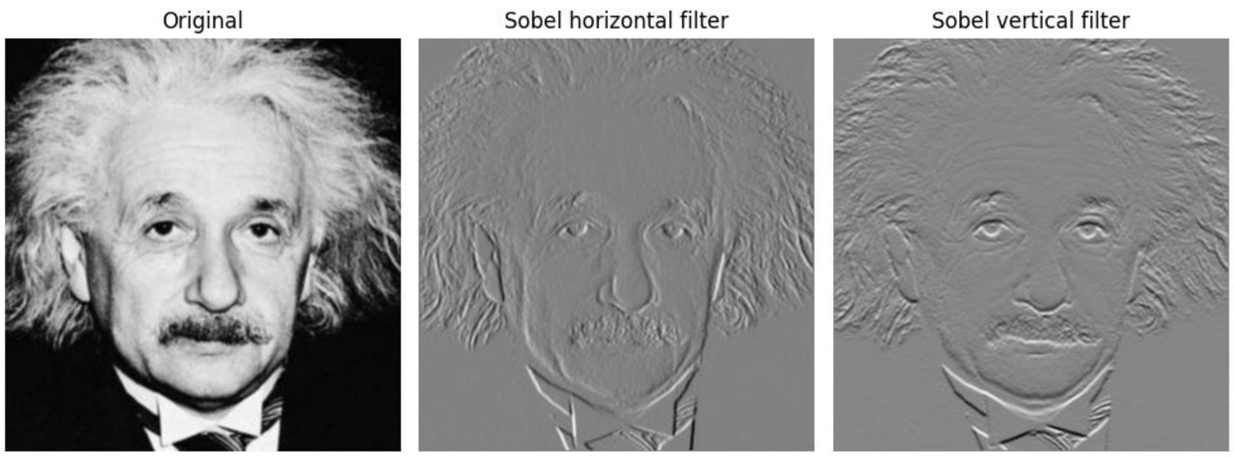
```
#Define kernel
kernel_1 = np.array([[-1, 0, 1], [-4, 0, 4], [-1, 0, 1]], dtype='float') #Sobel horizontal
kernel_2 = np.array([[-1, -4, -1], [0, 0, 0], [1, 4, 1]], dtype='float')  #Sobel vertical
```

Finally, image is filtered by calling the above defined function and the results are displayed along with the original image.



```
#Filter the image
Sobel_h = filter(img, kernel_1)
Sobel_v = filter(img, kernel_2)
```

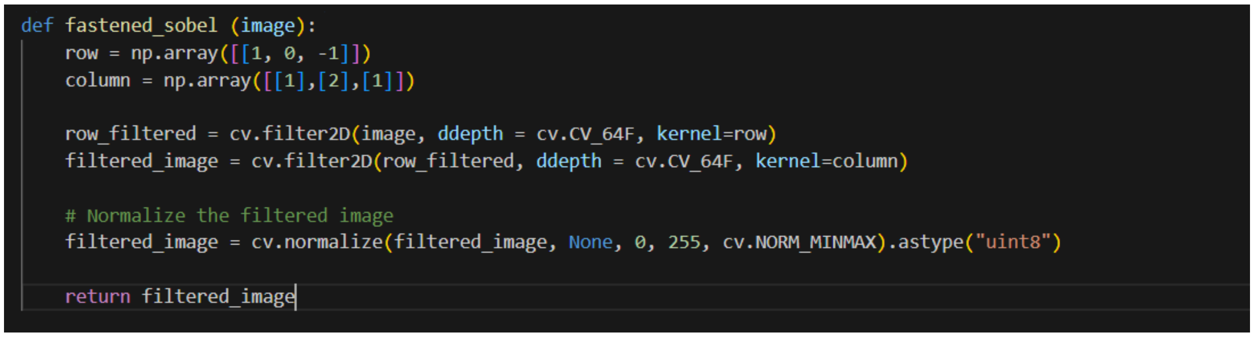
The output of the above code is,



c. Here, an attempt was taken to minimize the complexity of calculations in Sobel filter by using the property that,

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

First a function is defined as "fastened_sobel" to break the 3x3 Sobel kernel into two row and column vectors and then do filtration in two steps.

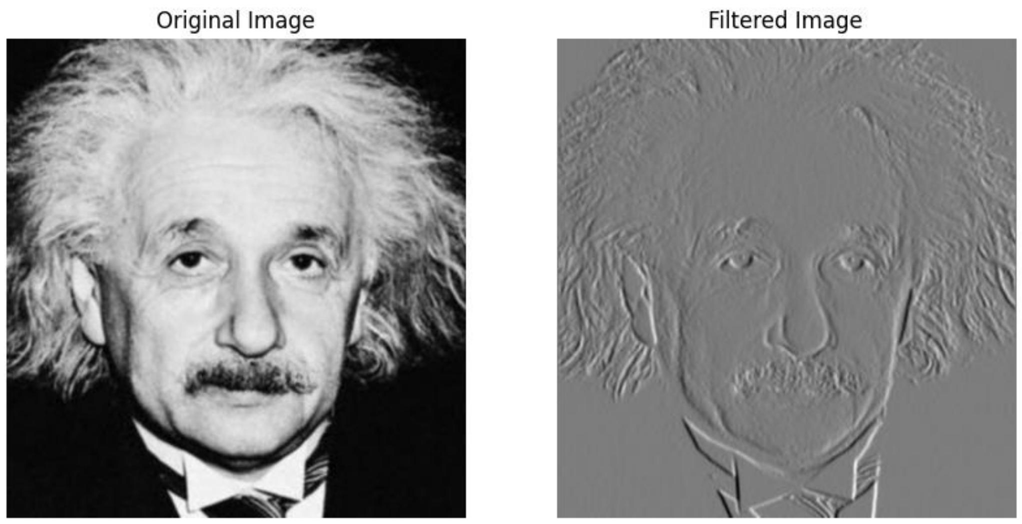


```
def fastened_sobel (image):
    row = np.array([[1, 0, -1]])
    column = np.array([[1],[2],[1]])

    row_filtered = cv.filter2D(image, ddepth = cv.CV_64F, kernel=row)
    filtered_image = cv.filter2D(row_filtered, ddepth = cv.CV_64F, kernel=column)

    # Normalize the filtered image
    filtered_image = cv.normalize(filtered_image, None, 0, 255, cv.NORM_MINMAX).astype("uint8")

    return filtered_image
```

Then filtering is done by calling the function defined above.

```
# Apply Sobel filtering using decomposition
filtered_image = fastened_sobel(original)
original = cv.cvtColor(original, cv.COLOR_BGR2RGB)
filtered_image = cv.cvtColor(filtered_image, cv.COLOR_BGR2RGB)
```

The result is as follows.

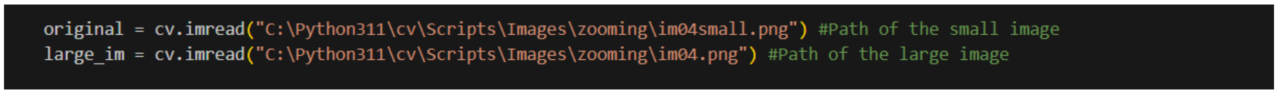


**08. Zoom images by a given scaling factor.**

First function is defined to zoom an input image by a given scaling factor. Zooming is done only for scaling factor, s ∈ (0,10]. If s is not in the range of interest, then it will print a message indicating invalid scaling factor and returns the input image as it is.

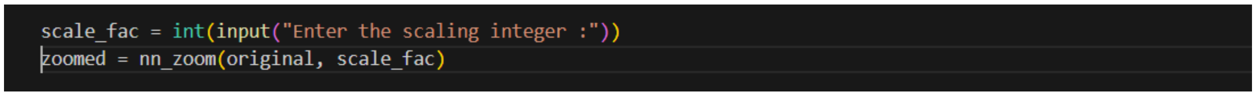
```
#Zoom function
def nn_zoom(image, s):
    if (s>0 and s<=10):
        row = image.shape[0] * s
        column = image.shape[1] * s
        zoomed_Img = np.zeros((row, column, 3), dtype = np.uint8)
        for i in range(row):
            for j in range(column):
                    zoomed_Img[i, j] = image[round(i/s - 0.5), round(j/s - 0.5)]

        return zoomed_Img
    else:
        print("Invalid scaling factor")
        return image
```

The image paths can be adjusted as needed.



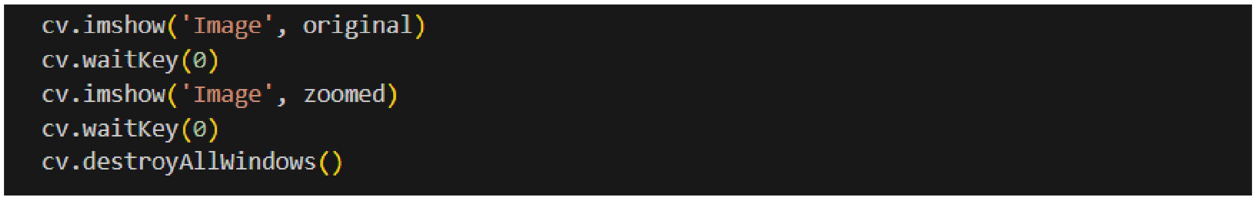
```
original = cv.imread("C:\Python311\cv\Scripts\Images\zooming\im04small.png") #Path of the small image
large_im = cv.imread("C:\Python311\cv\Scripts\Images\zooming\im04.png") #Path of the large image
```

When we run the code, it asks for a user input for the scaling factor. Then it calls function we defined before to zoom the images.



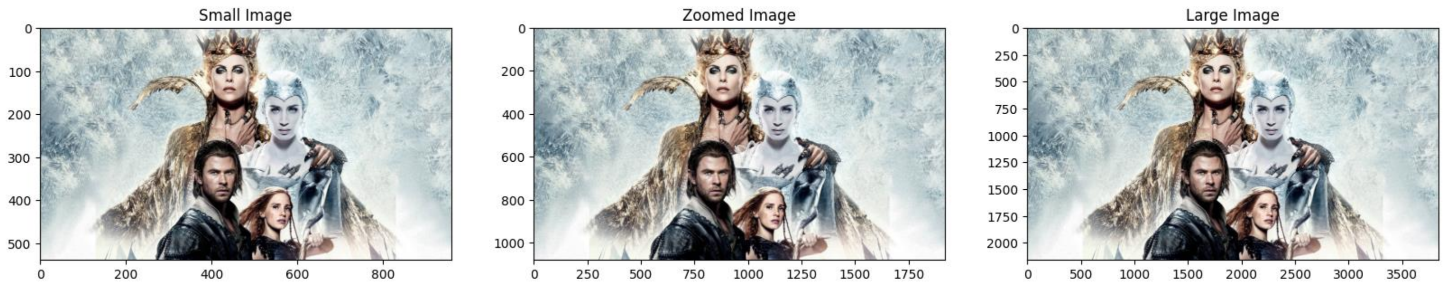
```
scale_fac = int(input("Enter the scaling integer :"))
zoomed = nn_zoom(original, scale_fac)
```

For a good visualization, here I have displayed original image and the zoomed images using OpenCV.



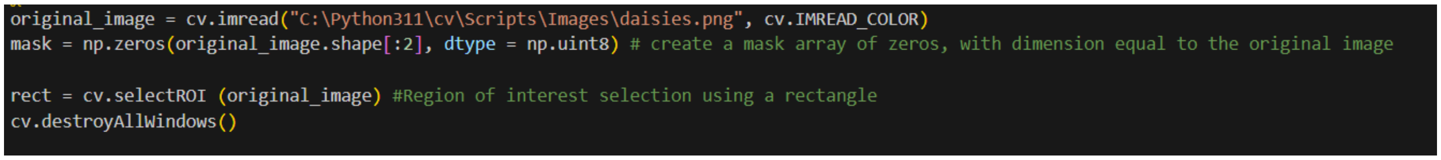
```
cv.imshow('Image', original)
cv.waitKey(0)
cv.imshow('Image', zoomed)
cv.waitKey(0)
cv.destroyAllWindows()
```

Malplotlib also used to display the results. Here is one simulation result for a scaling factor of 2.



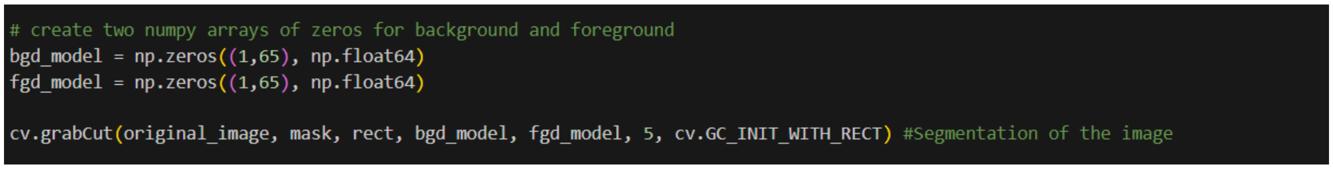
**09. Segmentation of an image and adjusting focus level.**

a. Here we segment a flower image to extract its foreground and the background. To select the foreground region, I have used the function selectROI in OpenCV. When we run the code, the image opens in a new window where we can select the region we need to extract. We can simply draw a rectangle around the flower (since in this particular question we are trying to extract the flower), and it will create a proper mask to extract only the flower.



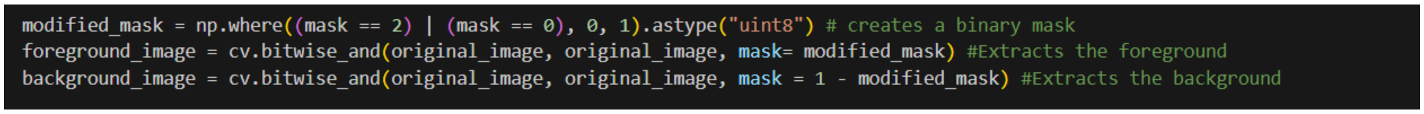
```
original_image = cv.imread("C:\Python311\cv\Scripts\Images\daisies.png", cv.IMREAD_COLOR)
mask = np.zeros(original_image.shape[:2], dtype = np.uint8) # create a mask array of zeros, with dimension equal to the original image

rect = cv.selectROI (original_image) #Region of interest selection using a rectangle
cv.destroyAllWindows()
```

Then the image is segmented using grabcut function in OpenCV. There, a mask is created with pixels related to foreground are in white in color and rest are black in color.



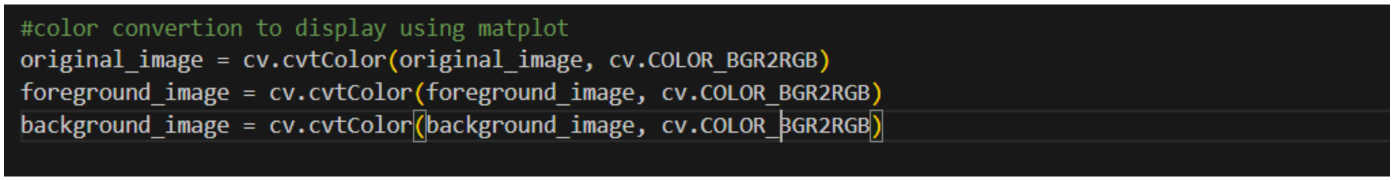
```
# create two numpy arrays of zeros for background and foreground
bgd_model = np.zeros((1,65), np.float64)
fgd_model = np.zeros((1,65), np.float64)

cv.grabCut(original_image, mask, rect, bgd_model, fgd_model, 5, cv.GC_INIT_WITH_RECT) #Segmentation of the image
```

Further modifications are done to create a binary mask and then foreground and the background are extracted.



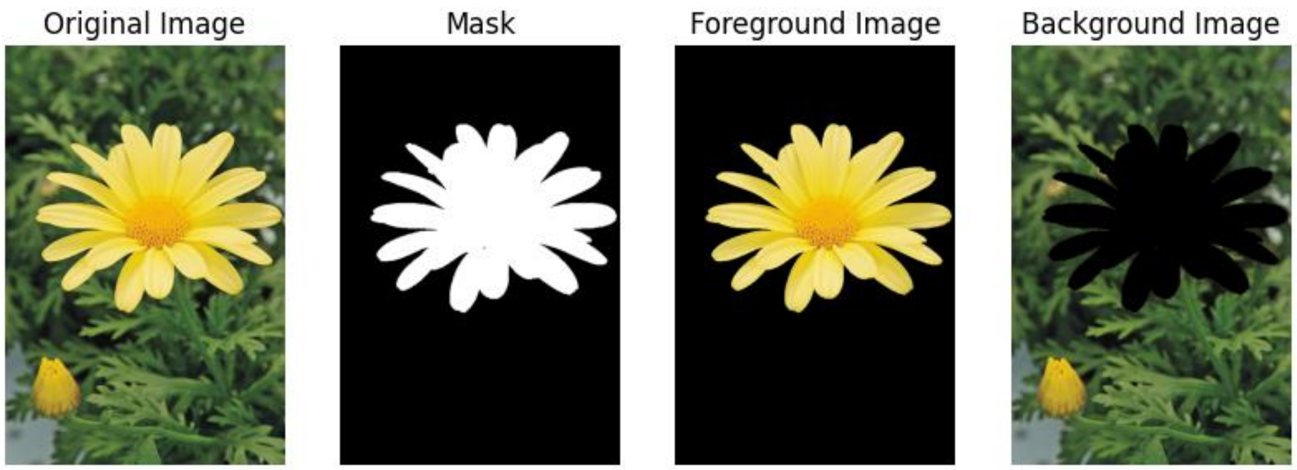
```
modified_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype("uint8") # creates a binary mask
foreground_image = cv.bitwise_and(original_image, original_image, mask= modified_mask) #Extracts the foreground
background_image = cv.bitwise_and(original_image, original_image, mask = 1 - modified_mask) #Extracts the background
```

Finally, a color plane conversion is done to display the images using matplot.



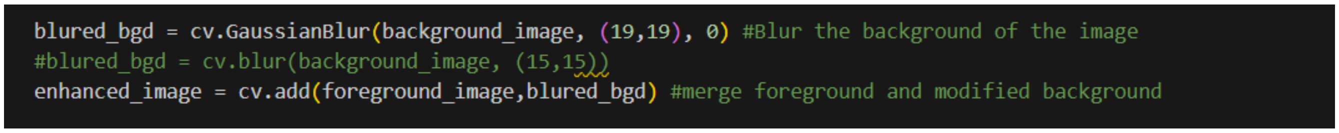
```
#color convertion to display using matplot
original_image = cv.cvtColor(original_image, cv.COLOR_BGR2RGB)
foreground_image = cv.cvtColor(foreground_image, cv.COLOR_BGR2RGB)
background_image = cv.cvtColor(background_image, cv.COLOR_BGR2RGB)
```

The result of above is given below.



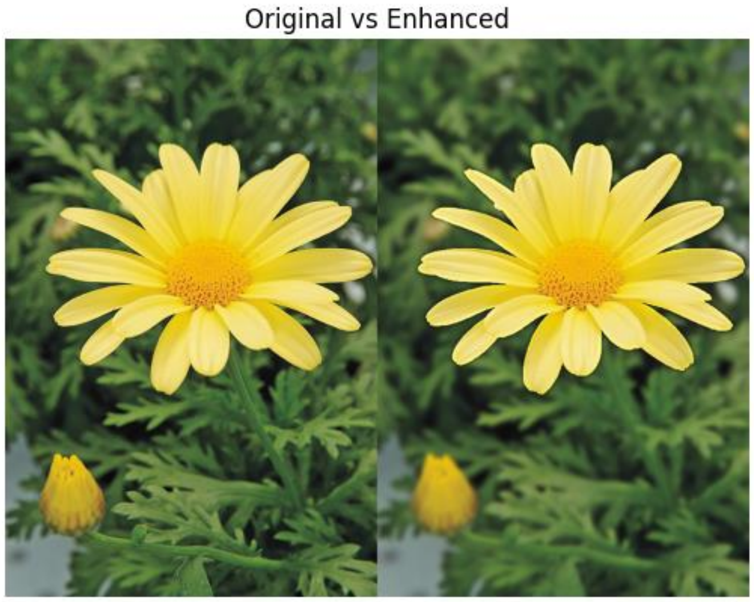
b. Now a blur effect is introduced to the extracted background. This can be done using the blur function or by using GaussianBlur function. Higher the kernel size, stronger the blurring effect.

Then modified background and original foreground are combined together to make a complete image.



```
blured_bgd = cv.GaussianBlur(background_image, (19,19), 0) #Blur the background of the image
#blured_bgd = cv.blur(background_image, (15,15))
enhanced_image = cv.add(foreground_image,blured_bgd) #merge foreground and modified background
```

Result of above processing task is given below.



c. Background just beyond the edge of the flower is quite dark in the enhanced image. Reason:

   In the process of enhancing an image with a blurred background, we are using a kernel filled with ones. This kernel is then normalized by dividing all its values by the total number of elements in the kernel. This kernel performs a weighted average of pixel values within its neighborhood during convolution.

   When you apply this convolution to the image, it effectively averages the pixel values in the background, including some darker pixels near the edges of the flower. This averaging process tends to reduce the intensity of pixels, making them darker. As a result, the background just beyond the edge of the flower appears darker in the enhanced image compared to the original image.