HW 02 – REPORT

소속 : 정보컴퓨터공학부

학번 : 201924624

이름 : Nemekhbayar Nomin

1. Introduction

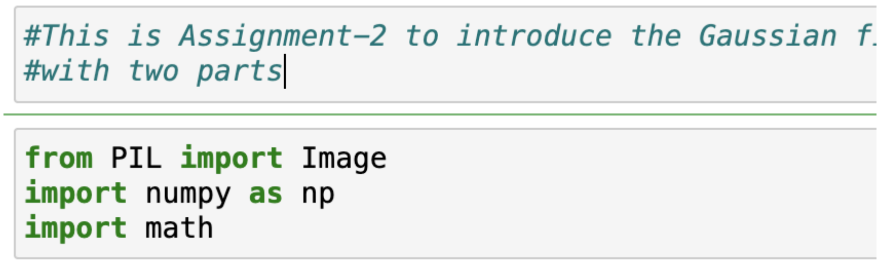
Practice Objectives and Theoretical Background Descriptions (Pages 1-2)

Reviewing and using the image processing and manipulating from the assignment-01, this assignment delve into the application of Gaussian filtering for image processing, specifically focusing on implementing various components of Gaussian filtering such as boxfilter() and gauss1d(), gauss2d() filters without builtin cv2.GaussianBlur() of the opencv library.

In the last part of exercise of Part2 of the assignment, the concept of hybrid image is explored where low and high frequency of two images combined to create a visually composition like two versions of the images are overlaid on top of each other.

Intro of the implementation:

where we import libraries/modules into the environment needed for this assignment.



```
#This is Assignment-2 to introduce the Gaussian f
#with two parts
```

```
from PIL import Image
import numpy as np
import math
```

- Pillow as PIL is a fork of an older library called PIL which is Python Imaging Library known as the main library for opening, manipulating and saving image file formats of python programming language.
- Numpy is a library for the Python programming language to support large, multi-dimensional arrays and matrices.
- math is a built-in module that provides standard mathematical functions and constants.

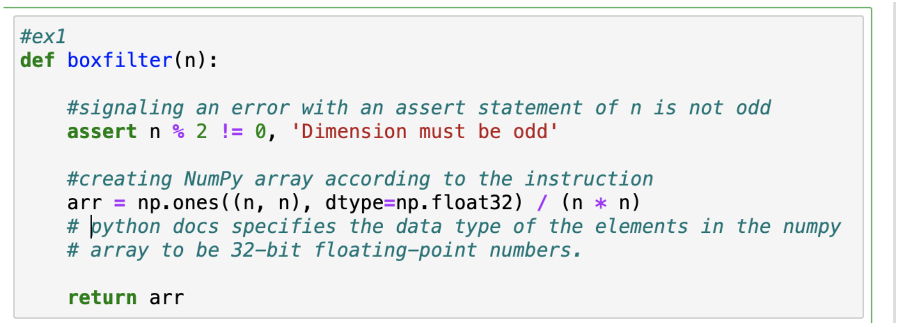
2. Process

Training content and results description (2 pages or more)

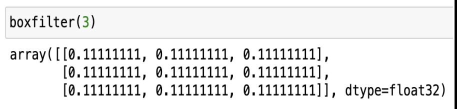
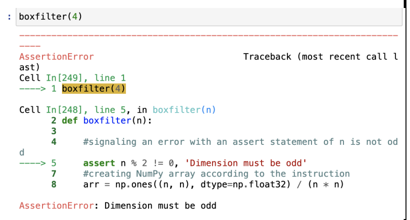
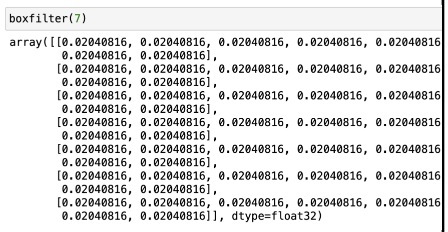
Part 1: Gaussian Filtering

1. Box Filter

As a warm ap exercise, box filter is written, that return a matrix size of (n, n).



```python
#ex1
def boxfilter(n):

    #signaling an error with an assert statement of n is not odd
    assert n % 2 != 0, 'Dimension must be odd'

    #creating NumPy array according to the instruction
    arr = np.ones((n, n), dtype=np.float32) / (n * n)
    # python docs specifies the data type of the elements in the numpy
    # array to be 32-bit floating-point numbers.

    return arr
```

Case n=3:	**Case n=4:**	**Case n=7:**
		

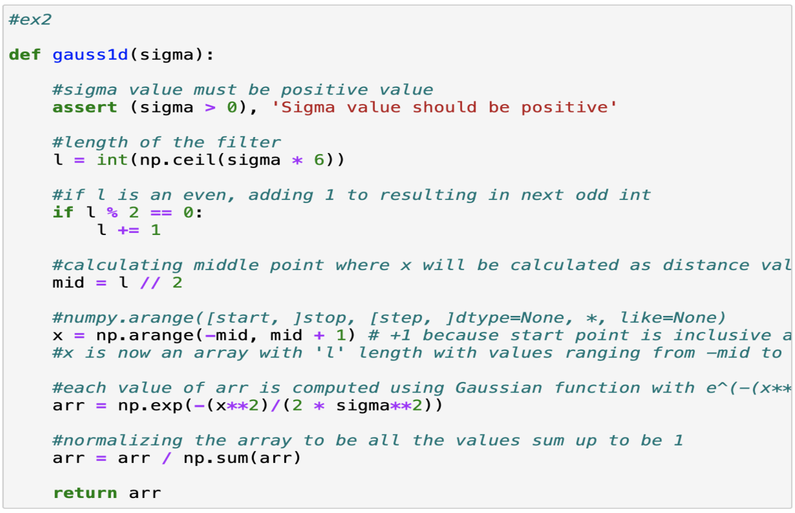
```
boxfilter(3)

array([[0.11111111, 0.11111111, 0.11111111],
       [0.11111111, 0.11111111, 0.11111111],
       [0.11111111, 0.11111111, 0.11111111]], dtype=float32)
```

```
: boxfilter(4)

---------------------------------------------------------------------------
----
AssertionError                            Traceback (most recent call l
ast)
Cell In[249], line 1
----> 1 boxfilter(4)

Cell In[248], line 5, in boxfilter(n)
      2 def boxfilter(n):
      3 
      4     #signaling an error with an assert statement of n is not od
d
----> 5     assert n % 2 != 0, 'Dimension must be odd'
      7     #creating NumPy array according to the instruction
      8     arr = np.ones((n, n), dtype=np.float32) / (n * n)

AssertionError: Dimension must be odd
```

```
boxfilter(7)

array([[0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816
        0.02040816, 0.02040816]], dtype=float32)
```

2. 1D Gaussian Filter

Next, gauss1d(sigma) function is implemented, to generated a 1D Gaussian filter for a given value of sigma.



```
#ex2

def gauss1d(sigma):

    #sigma value must be positive value
    assert (sigma > 0), 'Sigma value should be positive'

    #length of the filter
    l = int(np.ceil(sigma * 6))

    #if l is an even, adding 1 to resulting in next odd int
    if l % 2 == 0:
        l += 1

    #calculating middle point where x will be calculated as distance val
    mid = l // 2

    #numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None)
    x = np.arange(-mid, mid + 1) # +1 because start point is inclusive a
    #x is now an array with 'l' length with values ranging from -mid to

    #each value of arr is computed using Gaussian function with e^(-(x**
    arr = np.exp(-(x**2)/(2 * sigma**2))

    #normalizing the array to be all the values sum up to be 1
    arr = arr / np.sum(arr)

    return arr
```

Case sigma = 0.3:	**Case sigma = 0.5:**

`print(gauss1d(0.3))`
`[0.00383626 0.99232748 0.00383626]`
	`gauss1d(0.5)`
	`array([0.10650698, 0.78698604, 0.10650698])`

Case sigma = 1:	**Case sigma = 2:**

`gauss1d(1)`
`array([0.00443305, 0.05400558, 0.24203623, 0.39905028, 0.24203623,`
`       0.05400558, 0.00443305])`
	`gauss1d(2)`
	`array([0.0022182 , 0.00877313, 0.02702316, 0.06482519, 0.12110939,`
	`       0.17621312, 0.19967563, 0.17621312, 0.12110939, 0.06482519,`
	`       0.02702316, 0.00877313, 0.0022182 ])`

3. 2D Gaussian Filter

We extended our Gaussian filtering to 2D by implementing the function `gauss2d(sigma)`. This function utilizes the 1D Gaussian filter generated by `gauss1d(sigma)` and applies it using `np.outer` to obtain a 2D Gaussian filter. As with the 1D filter, the values are normalized to ensure a sum of 1.

```python
#ex3
def gauss2d(sigma):

    #1d gaussian array filter using gauss1d()
    temp = gauss1d(sigma)

    #numpy.outer(a, b, out=None)[source]
    #using np.outer to add a new axis to the existing array returned by
    arr = np.outer(temp, temp)

    #normalizing the values in the filter so they sum to 1.
    arr = arr / np.sum(arr)
    #print(f'filt shape: ', arr.shape)

    return arr
```

Case sigma = 0.5:	Case sigma = 1:

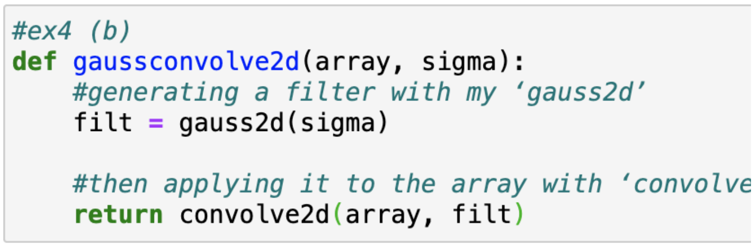
`gauss2d(0.5)`
 	
`array([[0.01134374, 0.08381951, 0.01134374],`
`       [0.08381951, 0.61934703, 0.08381951],`
`       [0.01134374, 0.08381951, 0.01134374]])`
	`gauss2d(1)`
	
	`array([[1.96519161e-05, 2.39409349e-04, 1.07295826e-03, 1.76900911e-03,`
	`         1.07295826e-03, 2.39409349e-04, 1.96519161e-05],`
	`       [2.39409349e-04, 2.91660295e-03, 1.30713076e-02, 2.15509428e-02,`
	`         1.30713076e-02, 2.91660295e-03, 2.39409349e-04],`
	`       [1.07295826e-03, 1.30713076e-02, 5.85815363e-02, 9.65846250e-02,`
	`         5.85815363e-02, 1.30713076e-02, 1.07295826e-03],`
	`       [1.76900911e-03, 2.15509428e-02, 9.65846250e-02, 1.59241126e-01,`
	`         9.65846250e-02, 2.15509428e-02, 1.76900911e-03],`
	`       [1.07295826e-03, 1.30713076e-02, 5.85815363e-02, 9.65846250e-02,`
	`         5.85815363e-02, 1.30713076e-02, 1.07295826e-03],`
	`       [2.39409349e-04, 2.91660295e-03, 1.30713076e-02, 2.15509428e-02,`
	`         1.30713076e-02, 2.91660295e-03, 2.39409349e-04],`
	`       [1.96519161e-05, 2.39409349e-04, 1.07295826e-03, 1.76900911e-03,`
	`         1.07295826e-03, 2.39409349e-04, 1.96519161e-05]])`

4. Convolution

(a) To perform convolution between an image array and a filter, we implemented the `convolve2d(array, filter)` function. This function iterates through each neighborhood of the image and computes the convolution using two nested loops. Zero-padding is applied to handle boundary cases.

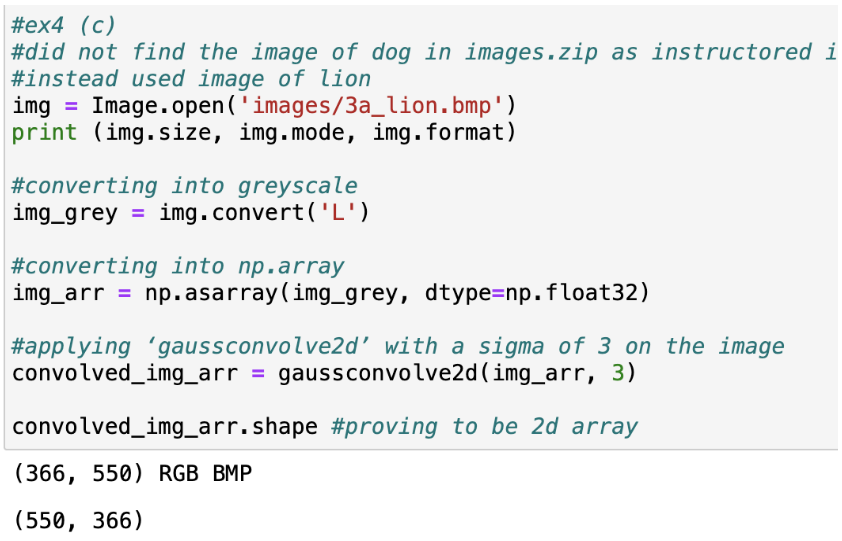
```python
#ex4 (a)

def convolve2d(array, filter):
    m, n = array.shape

    f = filter.shape[0] #since filter is square size

    #pad_m, pad_n = (fm - 1) // 2, (fn - 1) // 2
    pad_size = (f - 1) // 2 #since padding height and width is equal

    #np.pad(array, pad_width, mode='constant', **kwargs)[source]
    #using np.pad to pad the input array image
    base = np.pad(array, ((pad_size, pad_size), (pad_size, pad_size)), mode='constant')
    #below comment is for my own understanding!
    #(pad_m, pad_m) represents the amount of padding to add above and below each values,
    #(pad_n, pad_n) represents the amount of padding to add before and after each column


    #creating np.arrays filled with 0s with same size as array using np.zeros()
    res = np.zeros_like(array)

    # computing the convolution using two loops as instructed
    for i in range(m):
        for j in range(n):
            cut = base[i:i+f, j:j+f] #the neighborhood area where filter covers
            res[i, j] = np.sum(cut * filter) #element-wise multiplication to compute the

    return res.astype(np.float32)
```

(b) Implementing ‘gaussconvolve2d(array, sigma)’ function that applies Gaussian convolution to 2d array with given value of sigma.



```
#ex4 (b)
def gaussconvolve2d(array, sigma):
    #generating a filter with my ‘gauss2d’
    filt = gauss2d(sigma)

    #then applying it to the array with ‘convolve
    return convolve2d(array, filt)
```

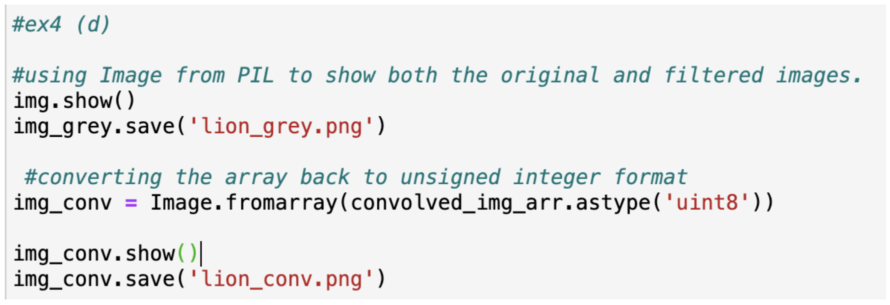
(c) Applying the ‘gaussconvolve2d(array, sigma)’ function to an image.



```
#ex4 (c)
#did not find the image of dog in images.zip as instructored i
#instead used image of lion
img = Image.open('images/3a_lion.bmp')
print (img.size, img.mode, img.format)

#converting into greyscale
img_grey = img.convert('L')

#converting into np.array
img_arr = np.asarray(img_grey, dtype=np.float32)

#applying ‘gaussconvolve2d’ with a sigma of 3 on the image
convolved_img_arr = gaussconvolve2d(img_arr, 3)

convolved_img_arr.shape #proving to be 2d array
```

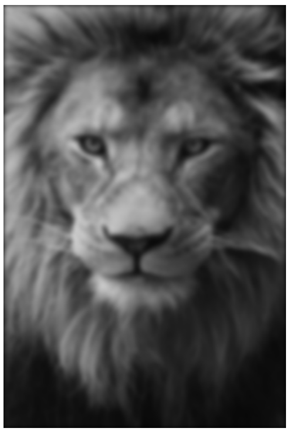
```
(366, 550) RGB BMP

(550, 366)
```

(d) Displaying the original and filtered images



```
#ex4 (d)

#using Image from PIL to show both the original and filtered images.
img.show()
img_grey.save('lion_grey.png')

 #converting the array back to unsigned integer format
img_conv = Image.fromarray(convolved_img_arr.astype('uint8'))

img_conv.show()
img_conv.save('lion_conv.png')
```

Results:

Original image	Filtered image
	

Part 2: Hybrid Images

```
#Part 2: hybrid Images
sigma = 5 #sigma value to be used in the following exercise
```

1. **Low Frequency Version**

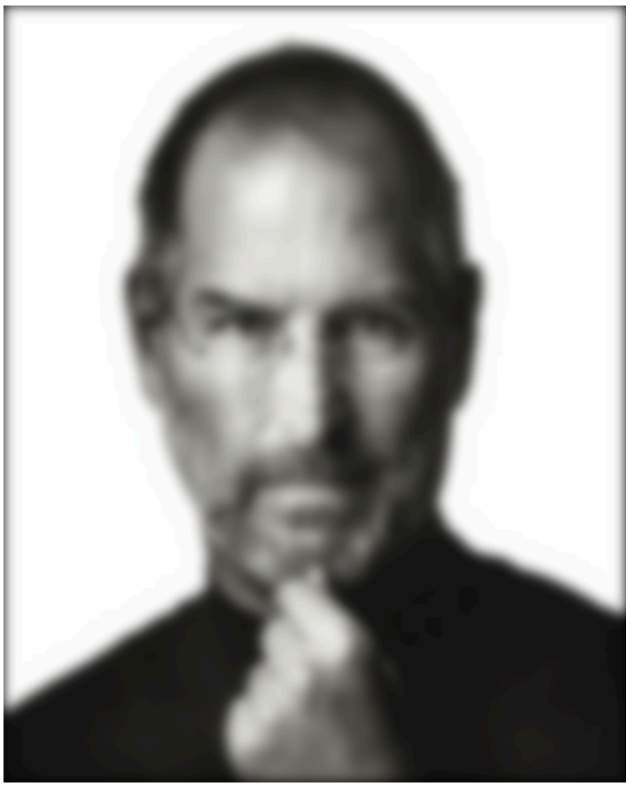
We generated the low-frequency version of an image by applying Gaussian filtering to blur the image. This was done separately for each channel of the image, resulting in three blurred channel arrays, which were then merged to obtain the low-frequency image.

```
#ex1 low frequency
#with 1a image
img_1a = Image.open('images/1a_steve.bmp')

#checking the image size, if both images have same size, no need to re
print(img_1a.size)

#splitting the channels
r, g, b = img_1a.split()

#each channels converted to np.array using np.asarray
r_arr = np.asarray(r, dtype=np.float32)
g_arr = np.asarray(g, dtype=np.float32)
b_arr = np.asarray(b, dtype=np.float32)


#using gaussconvolve2d to filter the channels of the image to blur
low_r_a = gaussconvolve2d(r_arr, sigma)
low_g_a = gaussconvolve2d(g_arr, sigma)
low_b_a = gaussconvolve2d(b_arr, sigma)

#forming in image back from each of the channels with unsigned intege

low_r_img = Image.fromarray(low_r_a.astype('uint8'))

low_g_img = Image.fromarray(low_g_a.astype('uint8'))

low_b_img = Image.fromarray(low_b_a.astype('uint8'))

#merging the three channels to form the low frequency image version
low_img_1a = Image.merge('RGB', (low_r_img, low_g_img, low_b_img))

#displaying and saving the low frequency image version
low_img_1a.show()
low_img_1a.save('steve_low.png')
```

Output:

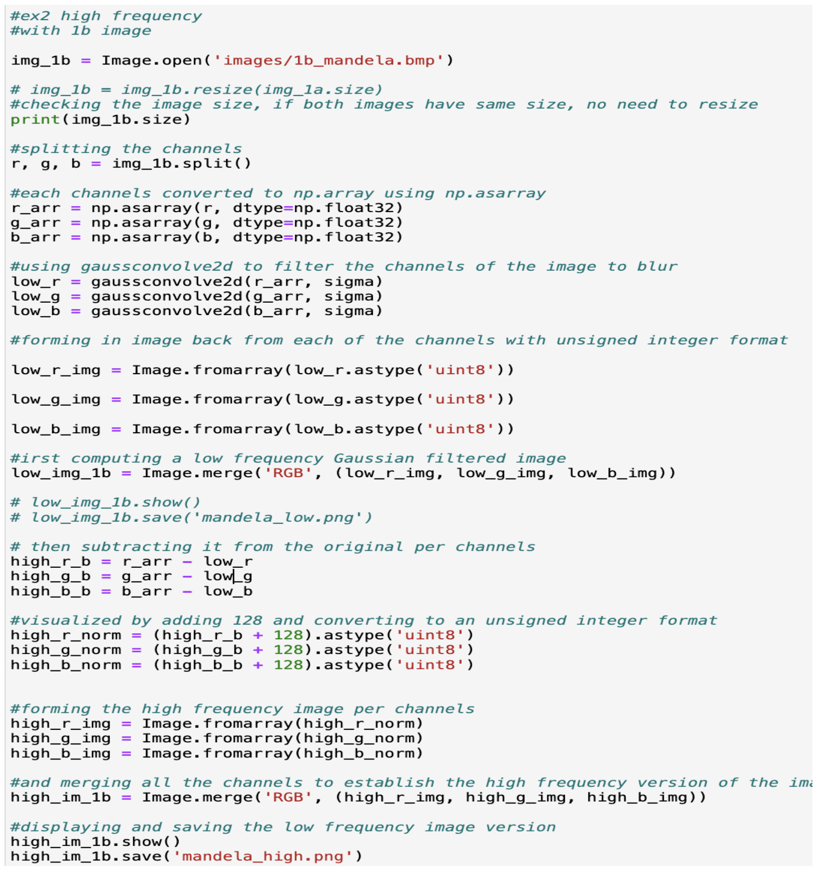
```
(502, 627)
```

Results:



2. High Frequency Version

To create the high-frequency version of another image, we first blurred the image using Gaussian filtering similar to the low-frequency version. Then, we subtracted the blurred image from the original to obtain the high-frequency components. The resulting negative values were adjusted for visualization.

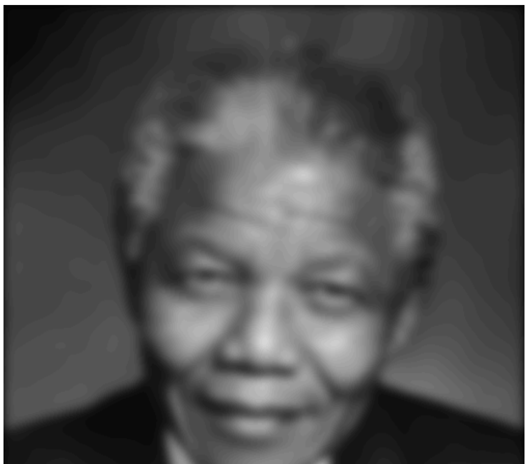


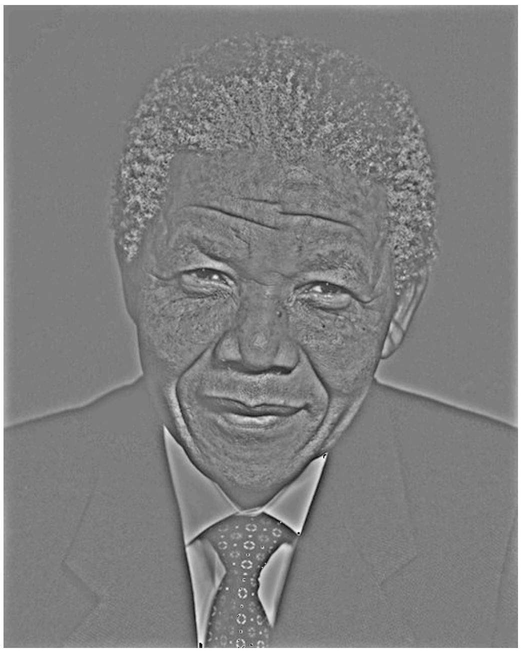
```
#ex2 high frequency
#with 1b image

img_1b = Image.open('images/1b_mandela.bmp')

# img_1b = img_1b.resize(img_1a.size)
#checking the image size, if both images have same size, no need to resize
print(img_1b.size)

#splitting the channels
r, g, b = img_1b.split()

#each channels converted to np.array using np.asarray
r_arr = np.asarray(r, dtype=np.float32)
g_arr = np.asarray(g, dtype=np.float32)
b_arr = np.asarray(b, dtype=np.float32)

#using gaussconvolve2d to filter the channels of the image to blur
low_r = gaussconvolve2d(r_arr, sigma)
low_g = gaussconvolve2d(g_arr, sigma)
low_b = gaussconvolve2d(b_arr, sigma)

#forming in image back from each of the channels with unsigned integer format

low_r_img = Image.fromarray(low_r.astype('uint8'))

low_g_img = Image.fromarray(low_g.astype('uint8'))

low_b_img = Image.fromarray(low_b.astype('uint8'))

#irst computing a low frequency Gaussian filtered image
low_img_1b = Image.merge('RGB', (low_r_img, low_g_img, low_b_img))

# low_img_1b.show()
# low_img_1b.save('mandela_low.png')

# then subtracting it from the original per channels
high_r_b = r_arr - low_r
high_g_b = g_arr - low_g
high_b_b = b_arr - low_b

#visualized by adding 128 and converting to an unsigned integer format
high_r_norm = (high_r_b + 128).astype('uint8')
high_g_norm = (high_g_b + 128).astype('uint8')
high_b_norm = (high_b_b + 128).astype('uint8')


#forming the high frequency image per channels
high_r_img = Image.fromarray(high_r_norm)
high_g_img = Image.fromarray(high_g_norm)
high_b_img = Image.fromarray(high_b_norm)

#and merging all the channels to establish the high frequency version of the ima
high_im_1b = Image.merge('RGB', (high_r_img, high_g_img, high_b_img))

#displaying and saving the low frequency image version
high_im_1b.show()
high_im_1b.save('mandela_high.png')
```

Output:

```
(502, 627)
```

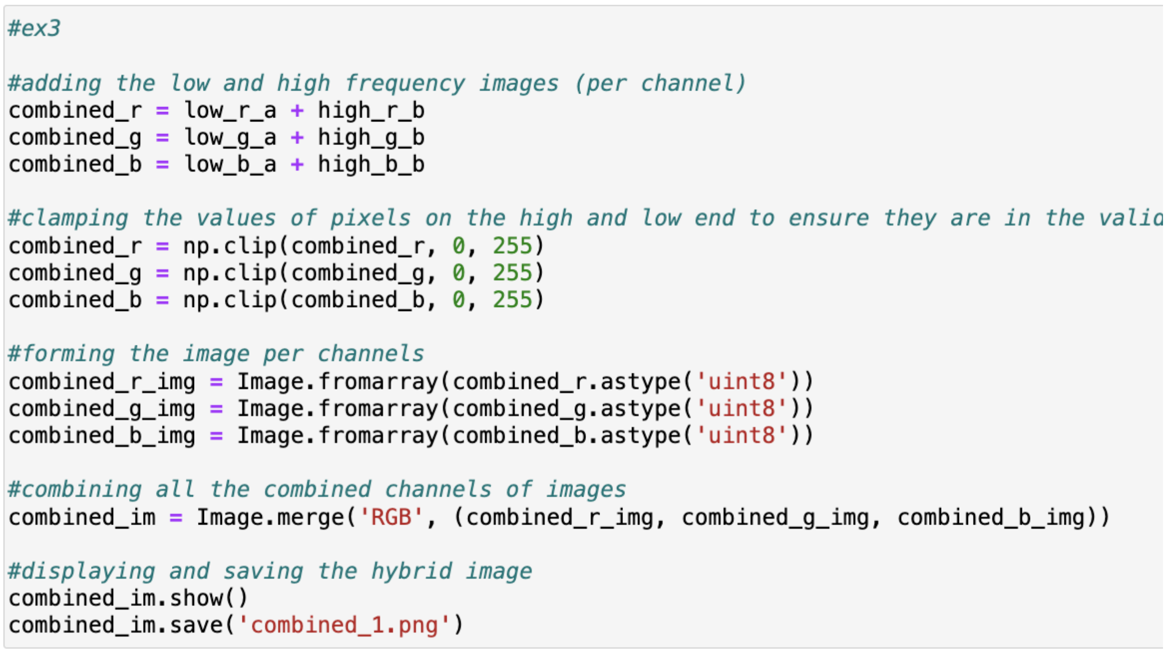
Result:





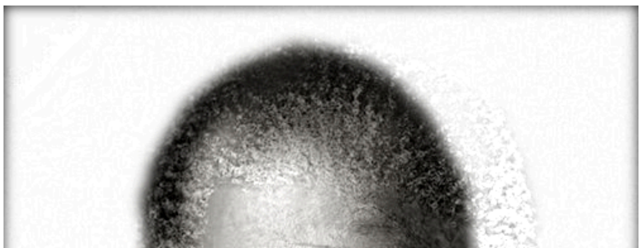
3. Combining Low and High Frequency Images

Finally, we combined the low-frequency version of one image with the high-frequency version of another image. This involved adding the corresponding channel arrays together after adjusting their values to ensure they fall within the valid range.



```
#ex3

#adding the low and high frequency images (per channel)
combined_r = low_r_a + high_r_b
combined_g = low_g_a + high_g_b
combined_b = low_b_a + high_b_b

#clamping the values of pixels on the high and low end to ensure they are in the valid
combined_r = np.clip(combined_r, 0, 255)
combined_g = np.clip(combined_g, 0, 255)
combined_b = np.clip(combined_b, 0, 255)

#forming the image per channels
combined_r_img = Image.fromarray(combined_r.astype('uint8'))
combined_g_img = Image.fromarray(combined_g.astype('uint8'))
combined_b_img = Image.fromarray(combined_b.astype('uint8'))

#combining all the combined channels of images
combined_im = Image.merge('RGB', (combined_r_img, combined_g_img, combined_b_img))

#displaying and saving the hybrid image
combined_im.show()
combined_im.save('combined_1.png')
```

Result:



3. Conclusion

Discussion and Conclusion (Page 1)

In conclusion, this assignment provided hands-on experience with Gaussian filtering and image processing techniques. I have recognized the importance of Gaussian function in image processing and how it is actually computed following the step by step instruction.

We implemented various components of Gaussian filtering, including box filters and 1D/2D Gaussian filters, and applied them to images for blurring and edge detection. And how high frequency version of image which is known as an edge detection is computed by subtracting the low frequency version of the image from the original image. Additionally, I have learned np.outer() function when implementing 2D Gaussian filter array from 1D Gaussian filter array.

Finally, we explored the creation of hybrid images by combining low and high-frequency components, resulting in visually interesting compositions.

Through this exercise, I have gained a deeper understanding of image processing fundamentals and their practical applications which was honestly, fun for me.