Report DIP

Students ID:

Khaled Saleh : 2232890

Mohanad Assaf : 2233114

Doctor:

Zaher Ibrahim Saleh Salah.

Date:

5/17/2024

Assignment 1

```
[1]: from PIL import Image
     from IPython.display import display
```

We imported two libraries: the first one for opening, creating, pasting, cropping, and saving images, and the second one for displaying the images directly in Jupyter itself rather than on the computer.

a) Read and display the images.

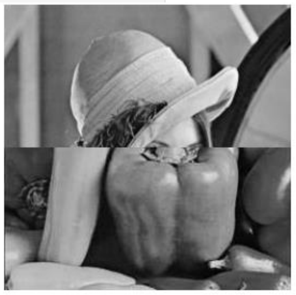
```
[2]: LenaGray = Image.open("LenaGray.jpg")
     PeppersGrey = Image.open("PeppersGrey.jpg")
     display(LenaGray, PeppersGrey)
```

We opened two images using `Image` class in the PIL (Pillow) library, and then displayed them.

b) Define a new 256 × 256 image J as follows: the upper half of J, i.e., the first 128 rows, should be equal to the upper half of the Lena image. The lower half of J, i.e., the 129th row through the 256th row, should be equal to the lower half of the Peppers image.

```
[3]: def combine_pictures(first_img, second_img):
         # .crop(left, top, right, bottom)
         top_part = first_img.crop((0,0,first_img.width, second_img.height//2))
         bottom_part = second_img.crop((0,second_img.height//2, second_img.width, second_img.height))
         J = Image.new('RGB', (first_img.width, first_img.height))
         J.paste(top_part, (0, 0))
         J.paste(bottom_part, (0, first_img.height//2))
         J.save('J.jpg')

     combine_pictures(LenaGray, PeppersGrey)
     J = Image.open('J.jpg')
     display(J)
```

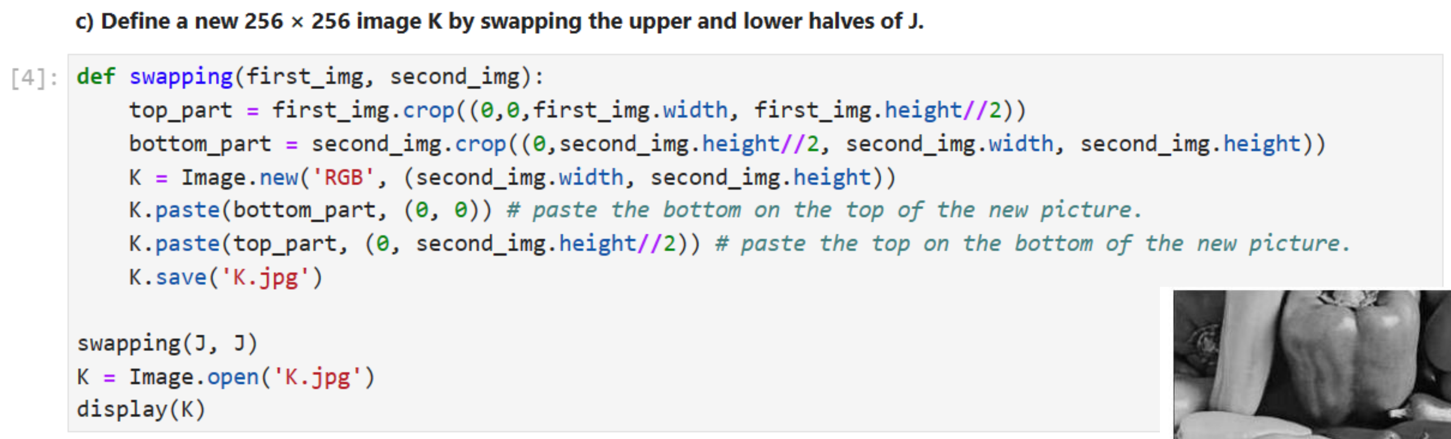


This Python code uses the PIL (Pillow) library to combine two images into one. The function `combine_pictures` is defined, which takes two arguments: `first_img` and `second_img`. Inside the function, the crop method is used to split the two images horizontally into two parts. The `top_part` is the top half of the first image, and the `bottom_part` is the bottom half of the second image. The crop method takes a tuple that defines the left, upper, right, and lower pixel coordinates of the crop rectangle.

A new blank image J is created using the `Image.new` method. The size of the new image is the same as the size of the first image (256x256), and the color mode is set to 'RGB'.

The `paste` method is used to paste the `top_part` and `bottom_part` into the new image. The `top_part` is pasted at the top of the new image, and the `bottom_part` is pasted at the bottom. The new image is saved as 'J.jpg' using the save method.

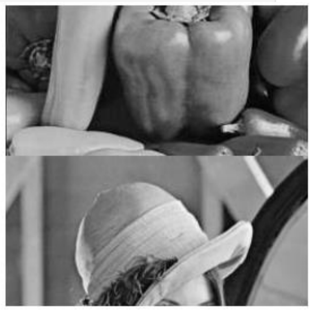
After the function call, the new image (J) is opened and displayed using the `Image.open` function and the display function from the `IPython` library.



c) Define a new 256 × 256 image K by swapping the upper and lower halves of J.

```
[4]: def swapping(first_img, second_img):
         top_part = first_img.crop((0,0,first_img.width, first_img.height//2))
         bottom_part = second_img.crop((0,second_img.height//2, second_img.width, second_img.height))
         K = Image.new('RGB', (second_img.width, second_img.height))
         K.paste(bottom_part, (0, 0)) # paste the bottom on the top of the new picture.
         K.paste(top_part, (0, second_img.height//2)) # paste the top on the bottom of the new picture.
         K.save('K.jpg')

     swapping(J, J)
     K = Image.open('K.jpg')
     display(K)
```



Here, we did the same thing as above, but the function name is swapping. In the paste part, we just pasted the `bottom_part` for the top of the new picture (K) and the `top_part` for the lower part of the new picture (K).

- **Assignment 2**

```
[1]: from PIL import Image, ImageFilter, ImageChops
     from IPython.display import display
     import numpy as np
```

From PIL import `Image, ImageFilter, ImageChops`: ImageFilter for applying filters, and ImageChops, Numpy for arithmetic operations on images.

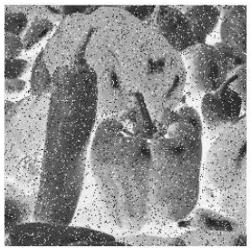
```
[3]: LenaGrayNoisy = Image.open('LenaGrayNoisy.jpg')
     PeppersGrayNoisy = Image.open('PeppersGreyNoisy.jpg')
     display(LenaGrayNoisy) ; display(PeppersGrayNoisy)
```

We opened two images using `Image` class in the PIL (Pillow) library, and then displayed them.

Image Negative

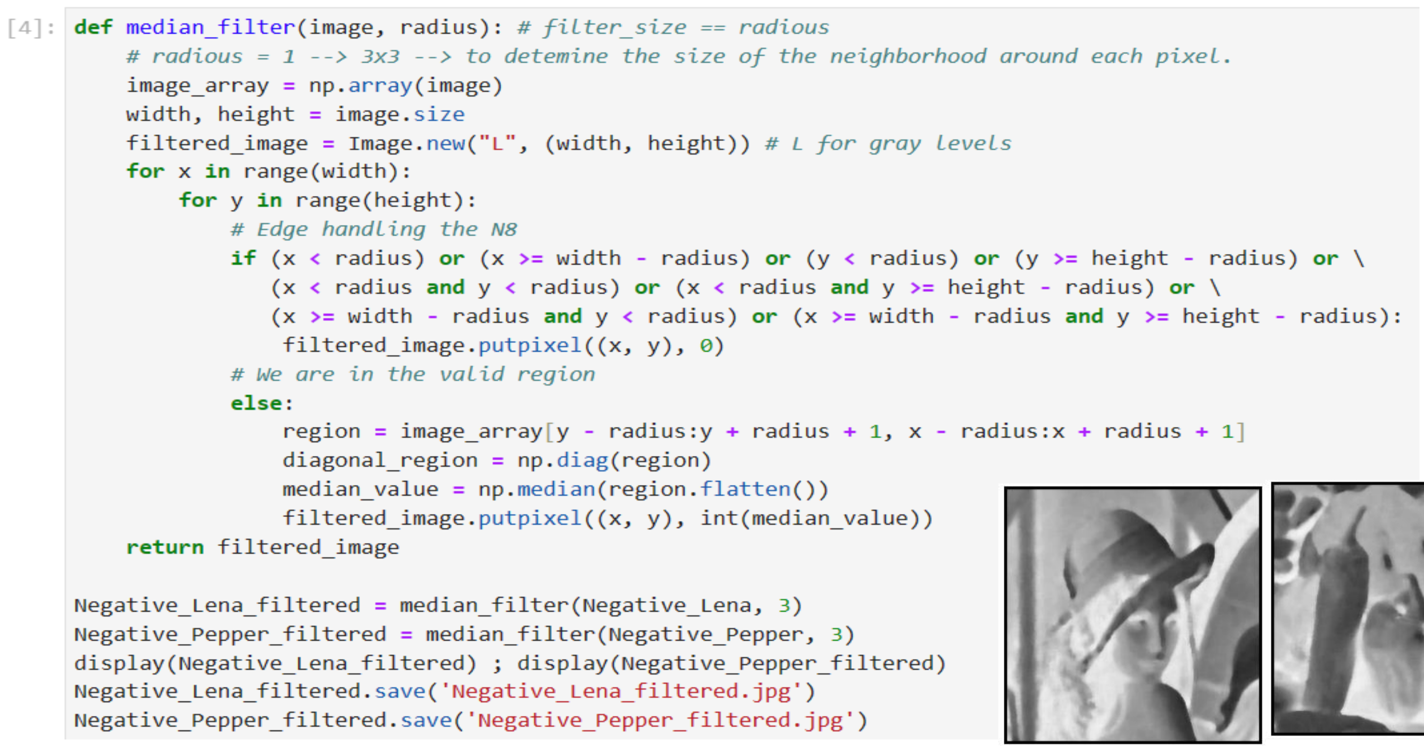
```
[3]: def image_negative(image):
         width, height = image.size
         negative_image = Image.new("L", (width, height))  # Use "L" for grayscale image
         for x in range(width):
             for y in range(height):
                 pixel_value = image.getpixel((x, y))
                 negative_pixel_value = 255 - pixel_value
                 negative_image.putpixel((x, y), negative_pixel_value)
         return negative_image

     Negative_Lena = image_negative(LenaGrayNoisy)
     Negative_Pepper = image_negative(PeppersGrayNoisy)
     display(Negative_Lena) ; display(Negative_Pepper)
     Negative_Lena.save('Negative_Lena.jpg')
     Negative_Pepper.save('Negative_Pepper.jpg')
```



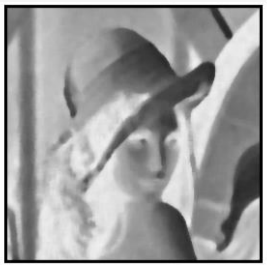
uses the PIL (Pillow) library to create the negative of an image. The function `image_negative` is defined, which takes one argument:

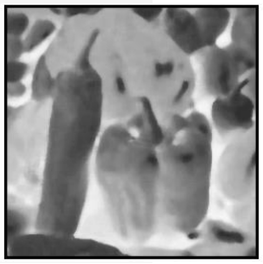
` image` . Inside the function, the size of the image is retrieved using the ` size` attribute, and a new blank grayscale image ` negative_image` of the same size is created using the ` Image.new` method. The function then loops over each pixel in the original image. For each pixel, it retrieves the pixel value using the ` getpixel` method, calculates the negative of the pixel value by subtracting it from 255, and then sets the pixel value in the negative image to this value using the ` putpixel` method. The negative image is then returned by the function.



```
[4]: def median_filter(image, radius): # filter_size == radious
         # radious = 1 --> 3x3 --> to detemine the size of the neighborhood around each pixel.
         image_array = np.array(image)
         width, height = image.size
         filtered_image = Image.new("L", (width, height)) # L for gray levels
         for x in range(width):
             for y in range(height):
                 # Edge handling the N8
                 if (x < radius) or (x >= width - radius) or (y < radius) or (y >= height - radius) or \
                    (x < radius and y < radius) or (x < radius and y >= height - radius) or \
                    (x >= width - radius and y < radius) or (x >= width - radius and y >= height - radius):
                     filtered_image.putpixel((x, y), 0)
                 # We are in the valid region
                 else:
                     region = image_array[y - radius:y + radius + 1, x - radius:x + radius + 1]
                     diagonal_region = np.diag(region)
                     median_value = np.median(region.flatten())
                     filtered_image.putpixel((x, y), int(median_value))
         return filtered_image

     Negative_Lena_filtered = median_filter(Negative_Lena, 3)
     Negative_Pepper_filtered = median_filter(Negative_Pepper, 3)
     display(Negative_Lena_filtered) ; display(Negative_Pepper_filtered)
     Negative_Lena_filtered.save('Negative_Lena_filtered.jpg')
     Negative_Pepper_filtered.save('Negative_Pepper_filtered.jpg')
```





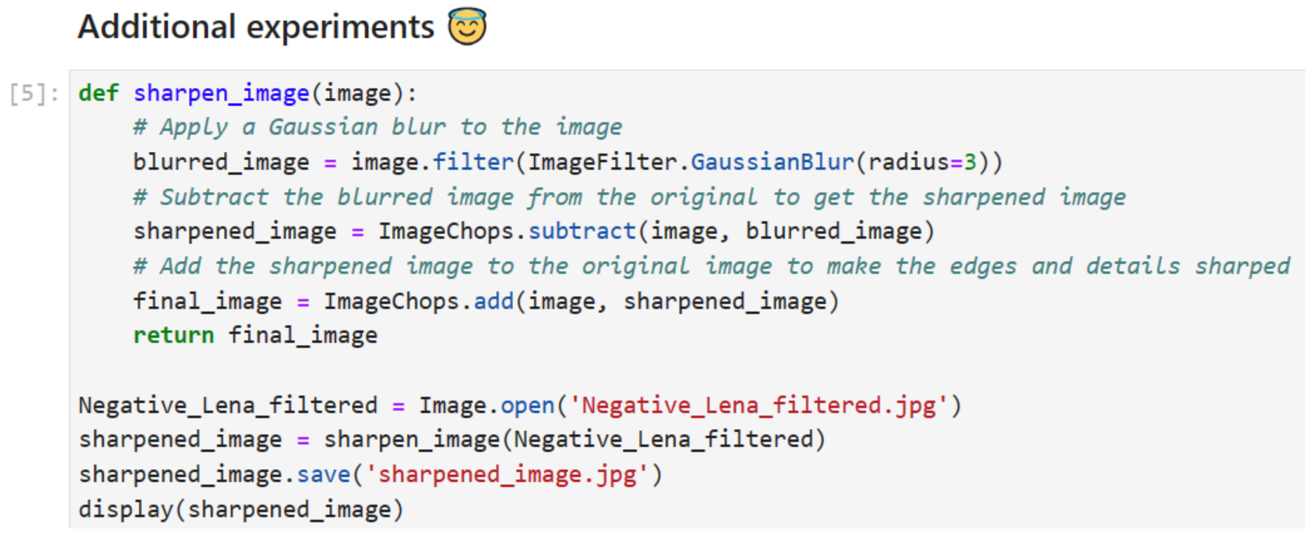
uses the PIL (Pillow) library and Numpy to apply a median filter to an image. The function ` median_filter` is defined, which takes two arguments: ` image` and ` radius` . The ` image` argument is the image itself, and ` radius` determines the size of the neighborhood around each pixel.

Inside the function, the image is converted to a Numpy array, and the size of the image is retrieved. A new blank grayscale image `filtered_image` of the same size is created using the `Image.new` method.

The function then loops over each pixel in the original image. For each pixel, it checks if the pixel is near the edge of the image. If it is, the pixel value in the filtered image is set to 0. If the pixel is not near the edge, a region around the pixel is extracted from the image array. The median value of the region is calculated using Numpy's `median` function, and the pixel value in the filtered image is set to this value.

The filtered image is then returned by the function.

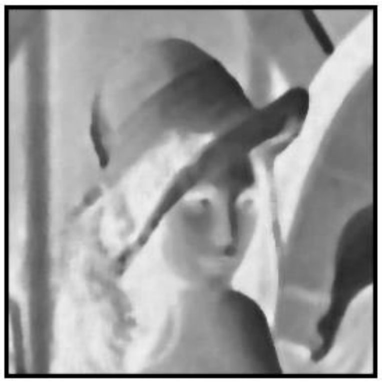
The `median_filter` function is called with two images: `Negative_Lena` and `Negative_Pepper`, and a radius of 3.

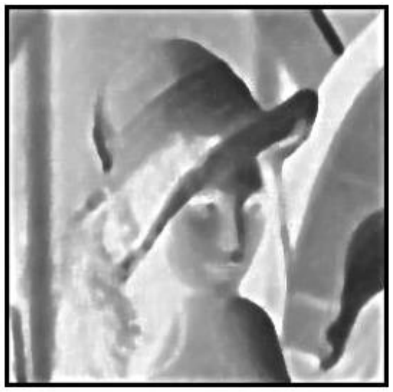


Additional experiments 😇

```
[5]: def sharpen_image(image):
         # Apply a Gaussian blur to the image
         blurred_image = image.filter(ImageFilter.GaussianBlur(radius=3))
         # Subtract the blurred image from the original to get the sharpened image
         sharpened_image = ImageChops.subtract(image, blurred_image)
         # Add the sharpened image to the original image to make the edges and details sharped
         final_image = ImageChops.add(image, sharpened_image)
         return final_image

     Negative_Lena_filtered = Image.open('Negative_Lena_filtered.jpg')
     sharpened_image = sharpen_image(Negative_Lena_filtered)
     sharpened_image.save('sharpened_image.jpg')
     display(sharpened_image)
```





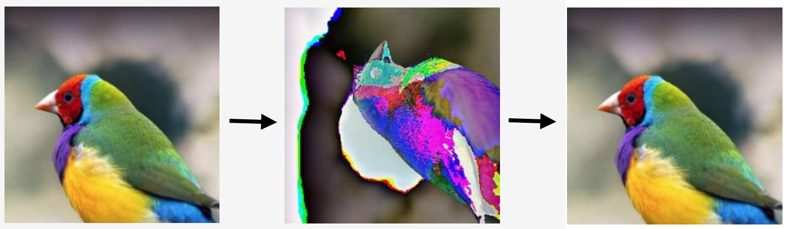
uses the PIL (Pillow) library to sharpen an image. The function `sharpen_image` is defined, which takes one argument: `image`. Inside the function, a Gaussian blur is applied to the image using the `filter` method with the `ImageFilter.GaussianBlur` filter. The radius of the blur is set to 3. A larger radius will result in a wider distribution and therefore a more pronounced blur.

The blurred image is subtracted from the original image using the `ImageChops.subtract` method to create a sharpened image. This process enhances the edges in the image.

The sharpened image is then added to the original image using the `ImageChops.add` method to further enhance the details and edges in the image. The final sharpened image is then returned by the function.

```
[6]: def encrypt_image(image):
         image_array = np.array(image)
         encrypted_image_array = (image_array + 159) % 256 # because it is 8 bit image
         return Image.fromarray(np.transpose(encrypted_image_array, (1,0,2)).astype(np.uint8))
     def decrypt_image(encrypted_image):
         encrypted_image_array = np.array(encrypted_image)
         decrypted_image_array = (np.transpose(encrypted_image_array,(1,0,2)) - 159) % 256
         return Image.fromarray(decrypted_image_array.astype(np.uint8))

     image = Image.open("Picture_to_Encrypt.jpg")
     # display the original image.
     display(image)
     # encrypt the image.
     encrypt = encrypt_image(image)
     encrypt.save("Encrypted_Image.png")
     # display the encrypted image.
     display(encrypt)
     # decrypt the encrypted image.
     encrypted = Image.open('Encrypted_Image.png')
     decrypt = decrypt_image(encrypted)
     decrypt.save("Decrypted_Image.png")
     # display the decrypted image.
     display(decrypt)
```



The code is divided into two main functions: `encrypt_image` and `decrypt_image`. `encrypt_image` Function. This function takes an image as input and performs the following steps:

1. Converts the image into a Numpy array.

2. Adds a constant value (159 in this case) to each pixel value in the array. This is done modulo 256 to ensure that the resulting values still fit within the 8-bit range (0-255) of pixel values.

3. Transposes the array, swapping the first two dimensions (height and width) while leaving the color channels. This is achieved by passing the tuple `(1,0,2)` to the `np.transpose` function.

4. Converts the resulting array back into an image and returns it.

The result is an image that has been both color-shifted and transposed.

`decrypt_image` Function. This function takes an encrypted image as input and reverses the operations performed by the `encrypt_image` function:

1. Converts the encrypted image into a Numpy array.

2. Transposes the array back to its original orientation by swapping the first two dimensions again.

3. Subtracts the constant value (159) from each pixel value in the array, again using modulo 256 to ensure the values stay within the 8-bit range.

4. Converts the resulting array back into an image and returns it.