Clab-1 Report

ENGN6528

Name: Fengdan Cui

UID Master: U6589143

05/04/2020

CONTENTS

Task-1: Python Warm-up 2
Task-2: Basic Coding Practice 3
2.1 Load a grayscale image and map to its negative image 3
2.2 Flip the image vertically 4
2.3 Load a colour image and swap the red and blue colour channels 4
2.4 Average the image with its vertically flipped image 5
2.5 Add a random value between [0, 255] to every pixel in the grayscale image and clip the new image between 0 and 255 5
Task-3: Basic Image I/O 6
3.1 Read and resize image 6
3.2 Convert the colour image into three grayscale channels 6
3.3 Compute the histograms for each of the grayscale images 7
3.4 Apply histogram equalisation to the resized image and three grayscale channels 7
Task-4: Image Denoising via a Gaussian Filter 8
4.1 Read and crop square image region 8
4.2 Add Gaussian noise to the new 256x256 image 9
4.3 Display the two histograms 9
4.4 Implement python function that performs a 5x5 Gaussian filtering 10
4.5 Apply the Gaussian filter to the noisy image 10
4.6 Compare the result with that by inbuilt 5x5 Gaussian filter 11
Task -5: Implement 3x3 Sobel filter in Python 12
Task-6: Image Rotation 13
6.1 Implement function my_rotation() for image rotation 13
6.2 Compare forward and backward mapping and analyse their difference 13
6.3 Compare different interpolation methods and analyze their difference 15
Reference 17
Appendices 18
my_Gauss_filter(noisy_image, size, sigma) 18
sobel_filter(image, direction = 'both') 18
my_rotation(image, angle, mapping = 'f') 19
nearest_inter(image_in, image_out, angle) 21
linear_inter(image_in, image_out, angle) 21

Task-1: Python Warm-up

(1) a = np.array([[2,4,5],[5,2,200]])
Function: create a 2 x 3 array ***a***.
Result: [[2   4   5]
[5   2   200]]

(2) b = a[0,:]
Function: take all values of the array ***a*** in (1) with index 0 in dimension 1
Result: [2   4   5]

(3) f = np.random.randn(500, 1)
Function: create a 500 x 1 array ***f*** of the given shape and populate it with random samples from standard normal distribution.
Result: a 500 x 1 array with random numbers

(4) g = f[f < 0]
Function: select values from f that are less than 0.
Result: an array ***g*** where all the elements are selected from array ***f*** in (3) which are less than 0.

(5) x = np.zeros((1, 100)) + 0.35
Function: create a new 1 x 100 array, filled with zeros, and then add 0.35 to each element in the array. Therefore, the new array ***x*** is a 1 x 100 array where all elements are 0.35.
Result: [[0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35
0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35
0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35
0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35
0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35
0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35
0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35 0.35
0.35 0.35]]

(6) y = 0.6 * np.ones([1, len(x.transpose())])
Function: create a new 1 x len(x.transpose()) array, filled with ones, and then multiply each element in the array by 0.6. The parameter ***x.transpose()*** means permuting the dimensions of array ***x***. The shape of array ***x*** is 1 x 100, so that the shape of ***x.transpose()*** is 100 x 1 and the length of it is 100. Therefore, the new array ***y*** is a 1 x 100 array where all elements are 0.6.
Result: [[0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6]]

(7) z = x – y
Function: Each element in the array ***x*** is subtracted from each element in the array ***y*** to form a new array ***z***. The shapes of x, y, and z are the same value.

Result: [[-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25
-0.25 -0.25 -0.25 -0.25]]

(8) a = np.linspace(1, 50)

Function: create a new array ***a*** that is filled with evenly spaced numbers from 1 to 50.

Result: [1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18.
19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36.
37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50.]

(9) b = a[: :-1]

Function: reverse array a in (8), meaning that starts from 50 which is the end, towards 1 which is the first, and assign the inverted array to the new array ***b***.

Result: [50. 49. 48. 47. 46. 45. 44. 43. 42. 41. 40. 39. 38. 37. 36. 35. 34. 33.
32. 31. 30. 29. 28. 27. 26. 25. 24. 23. 22. 21. 20. 19. 18. 17. 16. 15.
14. 13. 12. 11. 10. 9. 8. 7. 6. 5. 4. 3. 2. 1.]

(10) b[b <=50]=0

Function: all elements in array ***b*** less than or equal to 50 are assigned a value of 0. The elements in ***b*** are all between 1 and 50 so that all elements in ***b*** become 0.

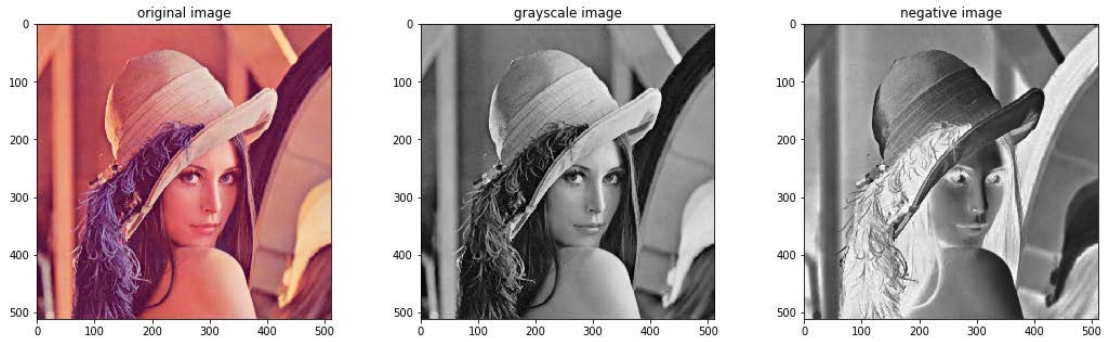
Result: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0.]

Task-2: Basic Coding Practice

2.1 Load a grayscale image and map to its negative image

Using the inbuilt function ***imageio.imread()*** to load the image by its path and assign it to the variable ***img***, which can be seen as a 512x512x3 array. In order to convert the image into grayscale, matrix multiplication between image and the transformation matrix is used by calculating ***np.dot(img[:, :, :3], [0.299, 0.587, 0.114])*** to change the pixels in the image. Then ***img_neg = 255 - img_gray***, which is to use 255 to minus each element in the image pixel to obtain the negative image. The final result can be seen in the following graph.

Graph 1. Image changes and contrast

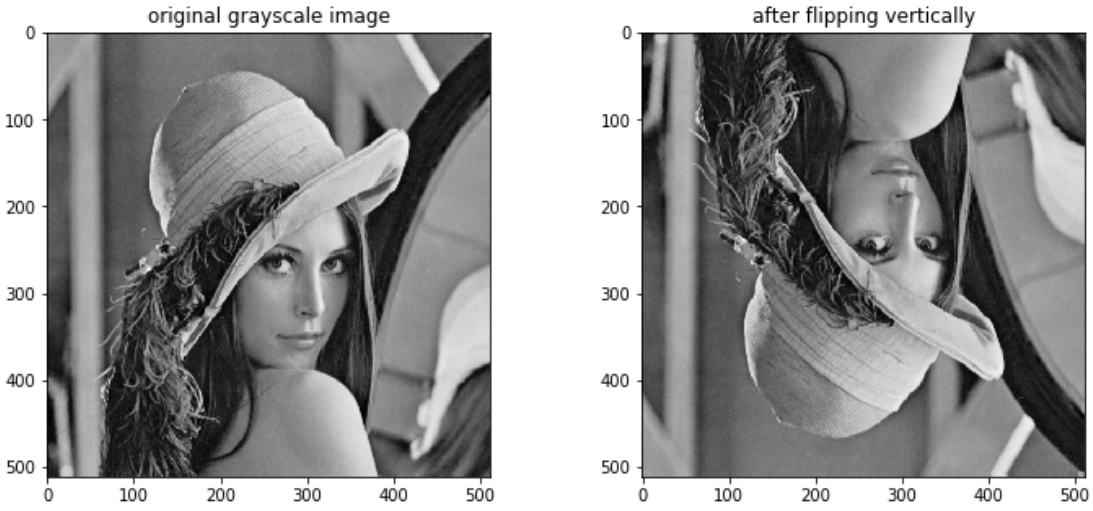


In addition, there is also another direct method that is to call the function in opencv ***cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)*** to transform the image into grayscale and then do the same operation for inverting image.

2.2 Flip the image vertically

Flipping an image upside down can be understood as flipping the array that stores the pixels of the image. Therefore, it can be done by calling ***np.flipud(img)***. The final result can be seen in the following graph.

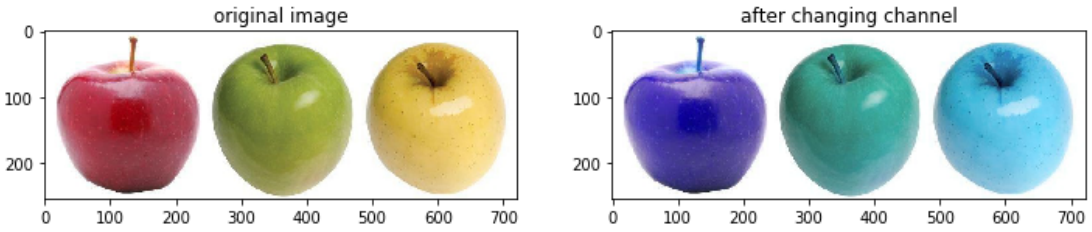
Graph 2. Flip the image vertically



2.3 Load a colour image and swap the red and blue colour channels

Images are stored in RGB format with pixel values so that by changing the location of pixels in image array, red channel and blue channel can be swapped. Using ***img_col_change = img_col[:, :, (2, 1, 0)]*** that means to reorder the channels from (0, 1, 2) to (2, 1, 0). The final result is shown as the following graph.

Graph 3. Swap the red and blue colour channels

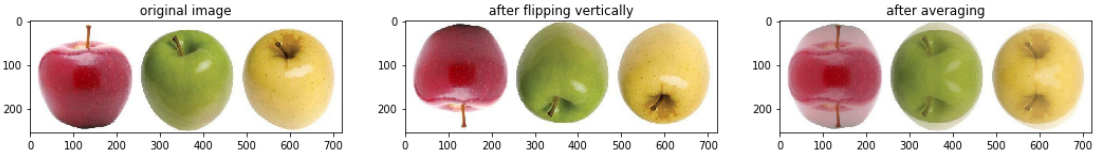


Furthermore, it can also be done by calling the in-built function ***cv2.cvtColor(img_col, cv2.COLOR_RGB2BGR)***

2.4 Average the image with its vertically flipped image

First of all, flipping the image vertically using ***np.flipud(img_col)***. Then calculating the average pixels of the two image arrays. The result may be not integer and can't be identified by ***imshow*** so that it is necessary to explicitly cast the image to uint8 before displaying it. The complete operation is ***np.uint8(np.average([img_col, img_col_ver], axis = 0))*** and the final result is shown below.

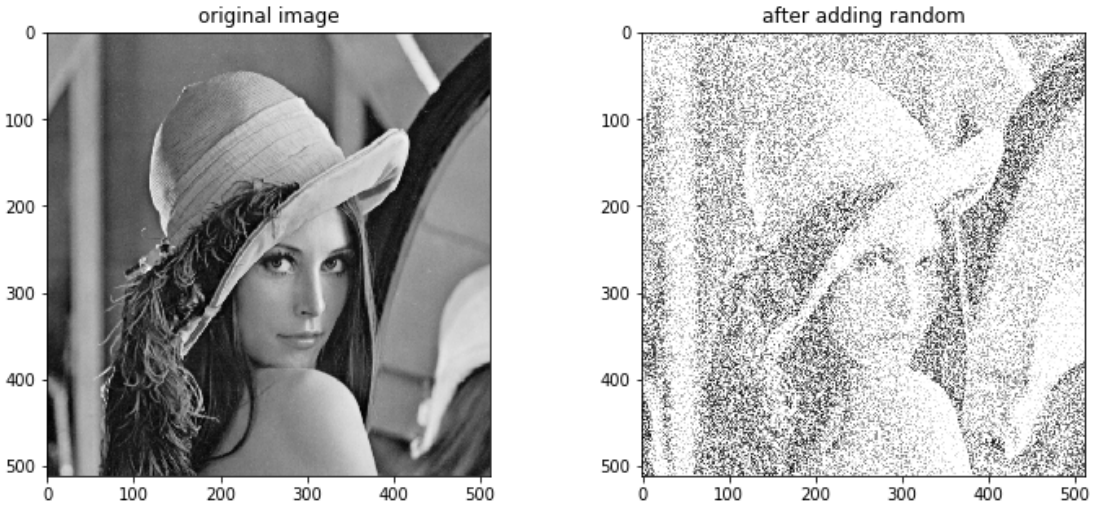
Graph 4. Average the image with its vertically flipped image



2.5 Add a random value between [0, 255] to every pixel in the grayscale image and clip the new image between 0 and 255

In order to operate each pixel of the grayscale image, iterating through each element of the array along the rows and columns of the array. For each pixel, using ***rand = random.randint(0, 255)*** to generate a random value between 0 and 255 and then add the result ***rand*** to that pixel by ***img_gray_thr[r,c] += rand***. After the nested looping, a new array is generated with some values are larger than 255. Therefore, calling ***np.clip(img_gray_thr, 0, 255)*** to limit the values in a specific range with a minimum value of 0 and a maximum value of 255. The final result can be seen in the below graph.

Graph 5. Adding random value to pixels

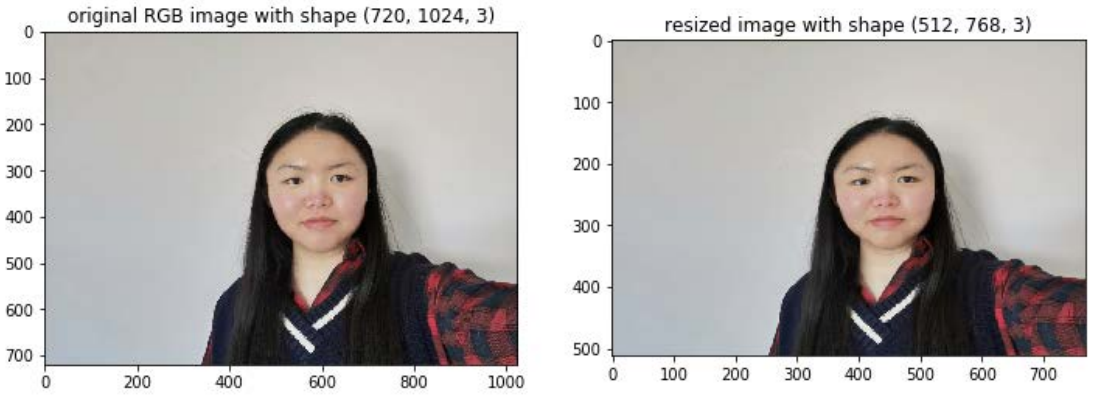


Task-3: Basic Image I/O

3.1 Read and resize image

As for reading and resizing the image, the inbuilt function is used, which is to call ***cv2.imread('../face_01_u6589143.jpg')*** to load the image and ***cv2.resize(img_rgb, (768, 512))*** to change the size to 768x512. Since opencv reads the image in BGR format, it is necessary to use ***cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)*** to convert the BGR format image to RGB format and display it. The final graphs are shown below.

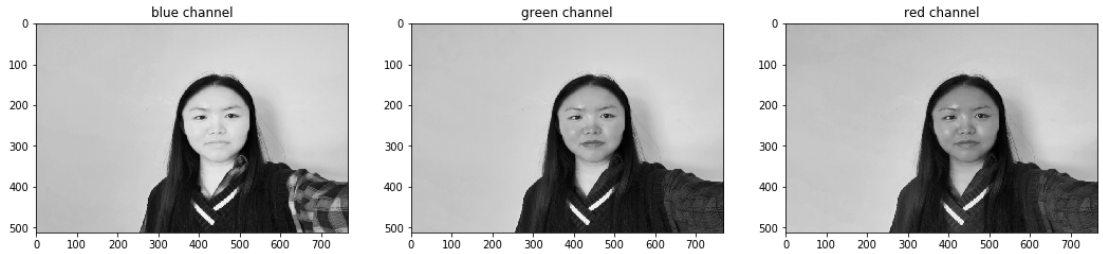
Graph 6. Read and resize face image



3.2 Convert the colour image into three grayscale channels

cv2.split(img_rsz) is used to split the channels of the image ***img_rsz*** into blue, green, and red respectively. The returned values are three 2-D arrays that represent the pixel values of grayscale images of different planes. Using ***imshow*** to plot such channel grayscale images and show them as below.

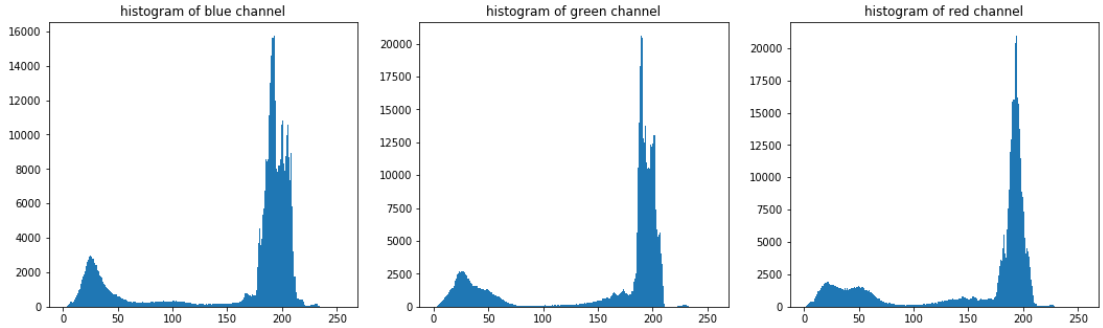
Graph 7. Three channel grayscale images



3.3 Compute the histograms for each of the grayscale images

The histogram is to show the number of pixels for each pixel value, the pixels are in the range 0 to 256 so these values should be used for display along the x-axis. Using ***matplotlib.pyplot.hist()*** to find and plot the histogram, which can be seen in the following graph.

Graph 8. Histograms of different grayscale images

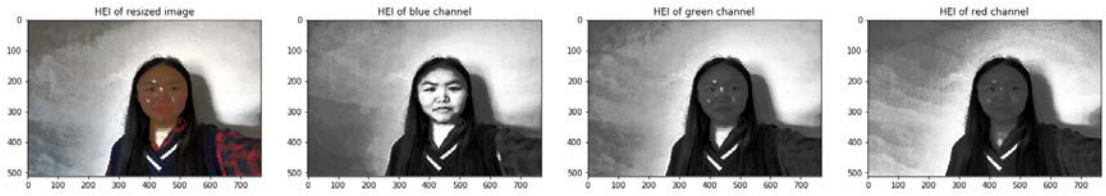


It can be seen in the histograms of the three grayscale images, the number of pixel values around 200 is relatively large, indicating the image is bright overall. In addition, there are also a part of the pixel values gathered between 0 and 50, which shows the image has a small amount of shadow such as my hair in the image.

3.4 Apply histogram equalisation to the resized image and three grayscale channels

Histogram equalization is a way to change the grayscale of each pixel in the image by changing the histogram of the image, transform the histogram of the original image into a uniformly distributed form, so as to achieve the effect of enhancing the overall image contrast. It is easy to apply histogram equalization to the three grayscale channels by using ***cv2.equalizeHist()***. However, histogram equalization ***cv2.equalizeHist()*** only works on 1 channel images so it is necessary to convert the image from RGB space to another colour space such like YUV colour space using ***cv2.cvtColor(img_rsz, cv2.COLOR_BGR2YCrCb)***. Then applying HE to the ***Y*** channel, merging the three channels and switching back to RGB format. The final result is shown below.

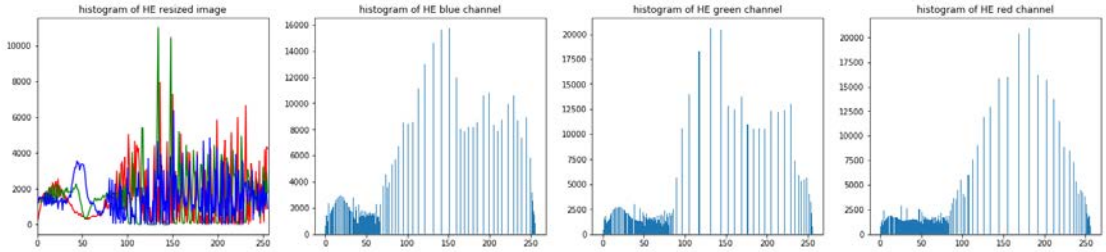
Graph 9. Apply histogram equalisation to images



As you can see from the graph, the contrast of the pictures is enhanced but the overall image is darkened significantly. In order to understand the changes in grayscale, I once again draw histograms for the four images which are shown below. It is obvious that the grayscale range of the image becomes much wider.

Histogram equalization is a processing method that acts globally, meaning that it does not selectively process data. If there are peaks in the histogram of some images such like the image I used above, the contrast is excessively enhanced after equalization, resulting in an unnatural image.

Graph 10. Histograms after applying HE

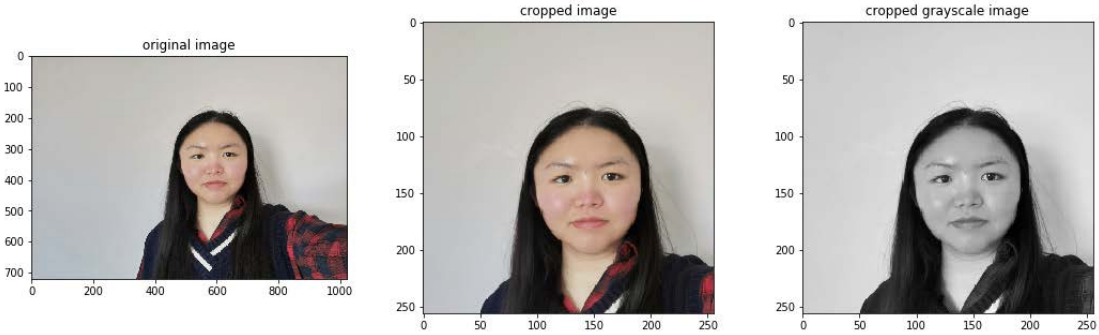


Task-4: Image Denoising via a Gaussian Filter

4.1 Read and crop square image region

Using ***cv2.imread()*** and ***cv2.cvtColor()*** to load the image and convert from BGR format to RGB format. It is obvious that the central facial part of the image is located the rows between 0 to 600 and the columns between 250 to 850 so that using ***cv2.resize(img[0: 600, 250: 850], (256, 256))*** to crop this part of the image and resize to 256x256. Then convert the image to grayscale by ***cv2.cvtColor(img_crop, cv2.COLOR_BGR2GRAY)***. Finally, calling ***cv2.imwrite()*** to save the image locally, which can be found under the same level with the code files named ***"face_01_crop.jpg"*** The final graphs are shown as following.

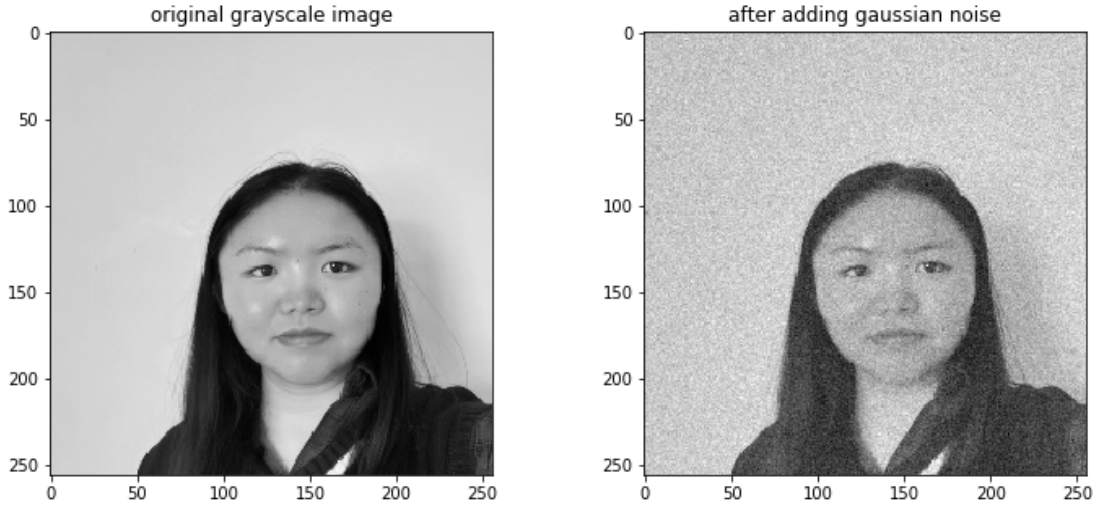
Graph 11. Read, crop, and convert image



4.2 Add Gaussian noise to the new 256x256 image

The shape of the new image is 256x256 and the datatype is uint8, so that the shape of generated Gaussian noise should be kept the same shape, where the elements in the array can be added to each pixel of the image directly. According to the given mean and standard deviation, the expression for generating is ***gaussian=np.random.normal(0, 15, img_crop_gray.shape)***. And then using ***img_crop_gray + gaussian*** to change the values of pixels in ***img_crop_grap***. The final result can be seen in the following graph. After adding Gaussian random values, almost every pixel on the image has noise with random depth, which can be used to simulate random signal interference received by the image during acquisition or transmission.

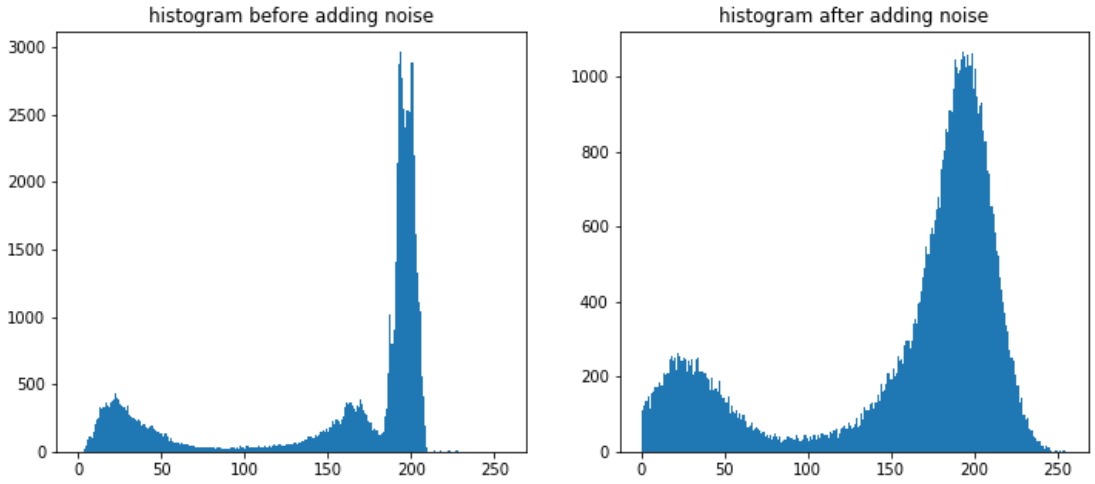
Graph 12. Add Gaussian noise to the image



4.3 Display the two histograms

Such like what I have done in Task 3.3, using ***ax1.hist(img_crop_gray.ravel(), 256, [0,256])*** to plot the histograms of the images. The histograms can be found in the following graph. According to the given graph, the histogram of the image that is added by Gaussian noise looks like a Gaussian distribution. Compared with the original histogram, the peak of the pixel value is reduced because the value of some pixels of the original image moved left or right under the influence of Gaussian noise.

Graph 13. Changes in histogram caused by adding Gaussian noise



4.4 Implement python function that performs a 5x5 Gaussian filtering

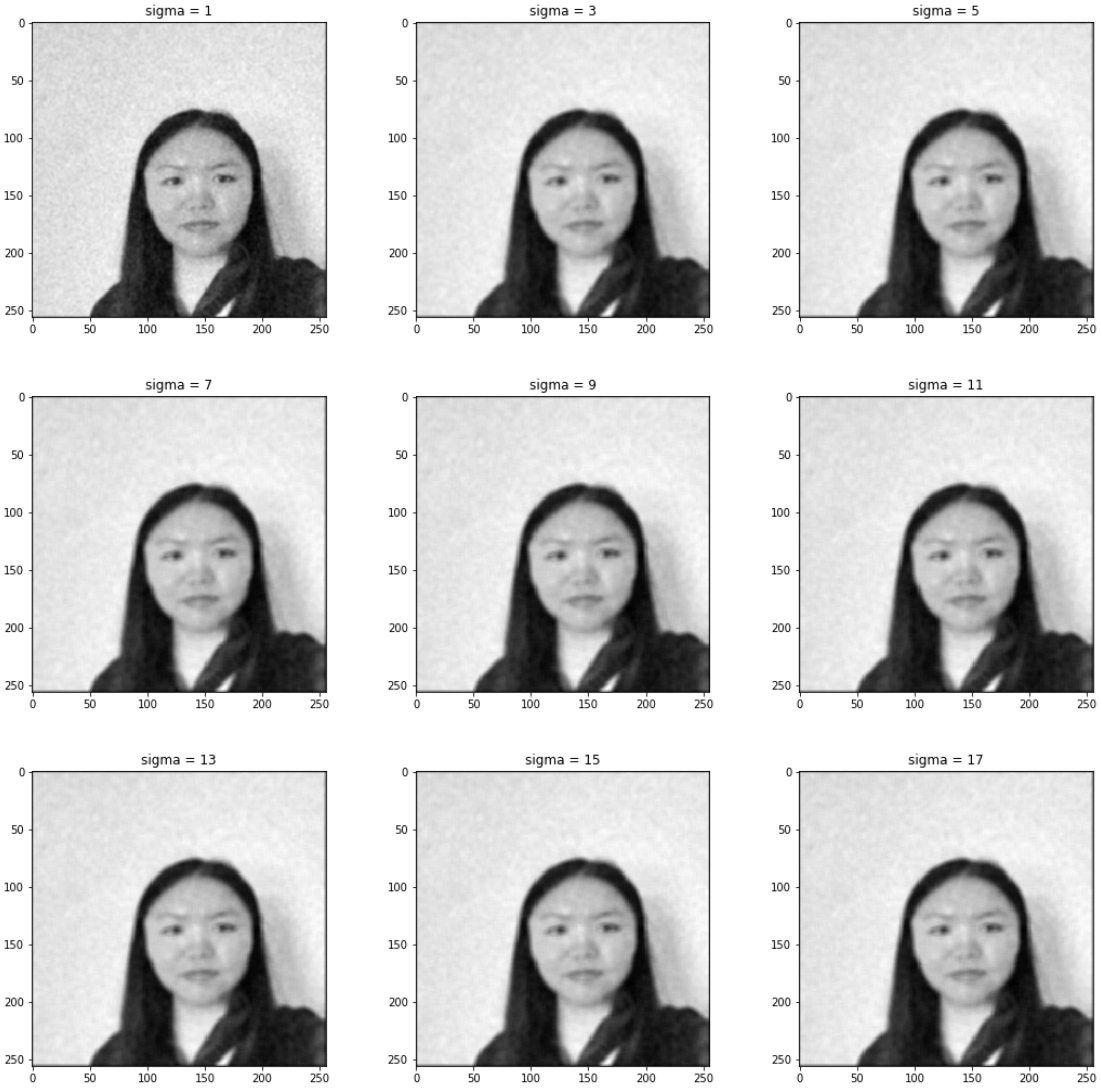
There is a file named ***"Global_lib.py"*** that includes all functions called by the tasks. The function ***my_Gauss_filter()*** is also been implemented in that file and the codes are also added to the appendices at the end of report. The overall algorithm is to convolve the image array and mask and there is a function named ***caculate_conv()*** which is implemented by myself for calculating convolution. To be specific:

1. ***cv2.getGaussianKernel()*** to generate the mask according to the size of mask and sigma along different directions.
2. ***out = np.zeros(inputs.shape)*** to create a new array using the shape of image.
3. ***pad_len = math.floor(size / 2)*** to calculate the padding length by rounding down half of the mask size.
4. ***inputs_pad = np.pad(inputs, ((pad_len, pad_len), (pad_len, pad_len)),'constant', constant_values=(0, 0))*** to add paddings to the image array, meaning to change the shape of array from ***nxm*** to ***(n+2*pad_len)x(m+2*pad_len)***.
5. Iterate the ***inputs_pad***, calculate convolution and assign the result to the corresponding position of ***out***.
6. Return ***out***, which is the Gaussian filtered results.

4.5 Apply the Gaussian filter to the noisy image

After applying my Gaussian filter with different standard deviations to the noise image, the smoothed images are plotted and shown below. According to the given graph, with the increase of stand deviation, the image is smoother and then keep almost same. When ***sigma=1***, denoising effect is not ideal and the denoised image is almost the same as the original noisy image. When sigma is up to 5 and greater, the image becomes smoother and the noise-removal effect is significantly improved. It doesn't make sense to continue to add the value because it can't be seen any visible change. Therefore, it is important to choose an appropriate standard deviation in order to achieve better effect.

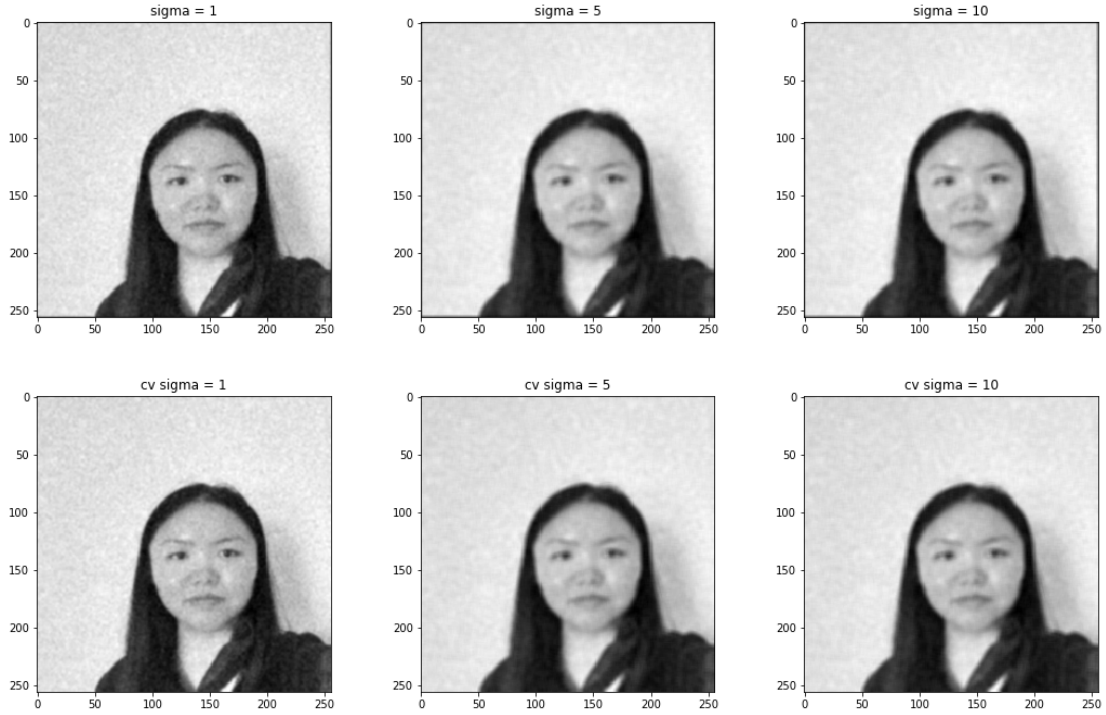
Graph 14. Gaussian filter with different sigma



4.6 Compare the result with that by inbuilt 5x5 Gaussian filter

By calling the inbuilt function ***cv2.GaussianBlur()*** with some same standard deviations 1, 5, and 7, another three denoised images are plotted and shown below with the images that filtered by my own function. It can be seen that the results after filtering using my own function is almost the same as using the inbuilt function.

Graph 15. Comparison with built-in function



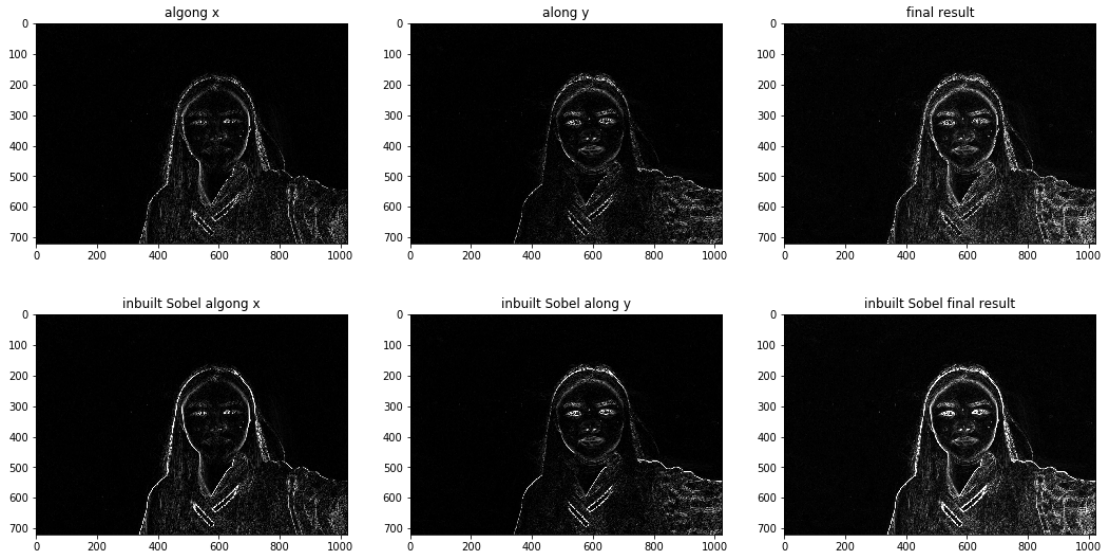
Task -5: Implement 3x3 Sobel filter in Python

The implemented function ***sobel_filter()*** can be found in ***"Global_lib.py"*** and the codes are also copied to the appendices, which is at the end of report. Gradient images can be obtained by convolving filters and images in different directions. In order to compare with inbuilt Sobel edge detection function, using ***cv2.Sobel()*** and ***cv2.addWeighted()*** to generate three images for edge detection along horizonal, vertical and both directions. It is obvious in the following graph that the majority of edges can be detected by my own method compared with the inbuilt function.

In general, Sobel filter is mainly used for edge detection. If there is an edge, there will be a certain change in the grayscale of the image. For example, if there is a boundary to divide the image into two part, one is white and the other is black. When analysing the change of grayscale from white to black, the gradient will change significantly at the edge but the gradient of other non-edge parts will keep zero. Therefore, the edges can be detected by analysing the variation of gradient values in different directions.

However, there isn't a specific function for calculating gradients of an image, so that difference operator can be used for approximate derivative of images. The 3x3 vertical convolution kernel is [-1 -2 -1;0 0 0;1 2 1], and the transpose of such matrix is for horizonal direction. Using such kernels to do convolution with the image to work out the derivative of the image along two directions so that the edge can be detected. For more convenient operation, adding the derivatives of two directions for the whole image.

Graph 16. Edge detection and comparison with inbuilt function

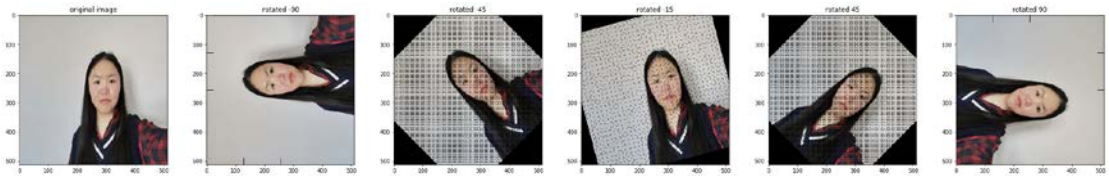


Task-6: Image Rotation

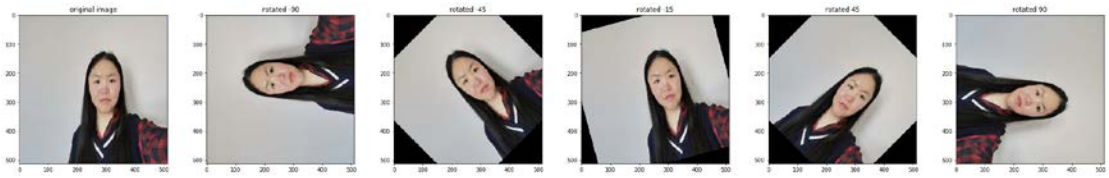
6.1 Implement function my_rotation() for image rotation

The function is implemented in ***"Global_lib.py"*** and can also be found in appendices that is at the end of report. The rotation result can be seen in the following two graphs.

Graph 17. Rotation by forward mapping



Graph 18 Rotation by backward mapping



6.2 Compare forward and backward mapping and analyse their difference

The differences between forward and backward mapping can be discussed in principles and visualized results, where the advantages and disadvantages are also explained in these two parts.

First of all, the algorithm of forward and backward mapping is:

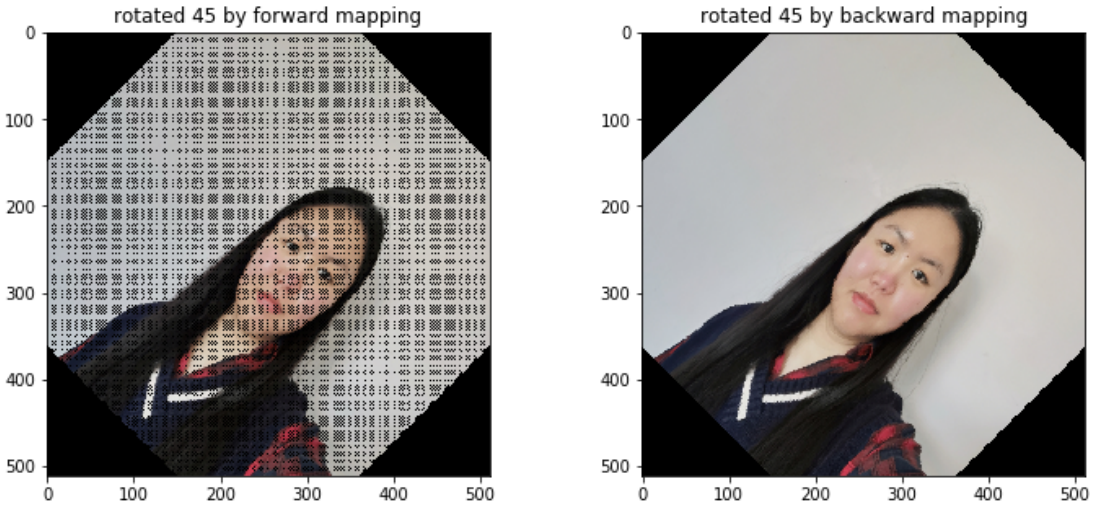
1. ***out = np.zeros(image.shape)*** and ***out_b = np.zeros(image.shape)*** to create new arrays with the same shape of the original image and filled with 0s, which are used to store the transformed pixels for different mapping.
2. Generate the rotation matrix ***r_matrix*** according to the given angle. Then, for different mapping method, the operation for how to deal with pixels is different.
3. If it is forward mapping, iterate the input image array and for a pixel,
   xy = np.dot(np.array([i - cx, j - cy]), r_matrix)
   x = int(xy[0] + cx)
   y = int(xy[1] + cy)
   where ***cx*** and ***cy*** is the rotation center and ***x, y*** is the new coordinates after rotation. If the new coordinates is in the bound of the output image ***out***, then ***out[x, y, :] = image[i, j, :]*** to copy the values to correct location.
4. If it is backward mapping, iterate the output image array and for a pixel,
   xy_b = np.dot(np.array([i - cx, j - cy]), np.linalg.inv(r_matrix))
   x_b = int(xy_b[0] + cx)
   y_b = int(xy_b[1] + cy)
   using the transpose of rotation matrix to determine the corresponding point of the original image. If the position is in the bound of input image, then ***out_b[i, j, :] = image[x_b, y_b, :]*** to copy the value from the input to the output.
5. Return ***out*** and ***out_b***, where ***out*** is for forward mapping and ***out_b*** is for backward mapping.

It is obvious in the algorithm procedure that different algorithms operate in different directions. The forward method aims to iterate source array and copy the pixels from source to target, so that there will be some locations in the output image without pixel sources and some points may locate out of bound after rotation that can’t be seen in the rotated image. In this case, the pixel of a certain point of the output image cannot be obtained directly. It is necessary to traverse all pixel values of the input image, perform coordinate transformation on it, and assign pixel values to integer positions to obtain the pixel values of each pixel of the output image, which is a disadvantage of the forward method. As for backward which is to iterate target array, calculate the corresponding source pixel and copy it from source to target. In this case, some pixels of input image may be oversampled. But it is more intuitive, the position of input image before transformation can be calculated according to the position of any integer point on the output image.

In addition, by comparing the rotation results, we can more intuitively understand the difference between the two methods. As we can seen in the following comparison images, there are many holes in the rotated image using forward mapping because pixels are not mapped one-to-one from the original image to the output. However, there is no hole in the rotated image mapped by backward mapping because each pixel value of the target image comes from the original image. So that the quality of rotated result

of backward is better.

Graph 19. Rotation results by different methods



6.3 Compare different interpolation methods and analyze their difference

In order to discuss the differences of interpolation methods, nearest-neighbour interpolation and bilinear interpolation are implemented by myself and applied to the rotated images. The code can be found both in ***"Global_lib.py"*** and the appendices at the end of report. In addition, inbuilt function ***cv2.resize()*** with different interpolation methods is applied to crop and enlarge an area of the image. After the experiment, it can be discussed from two aspects, principles and visualized results, the advantages and disadvantages of different methods are also explained in these aspects.

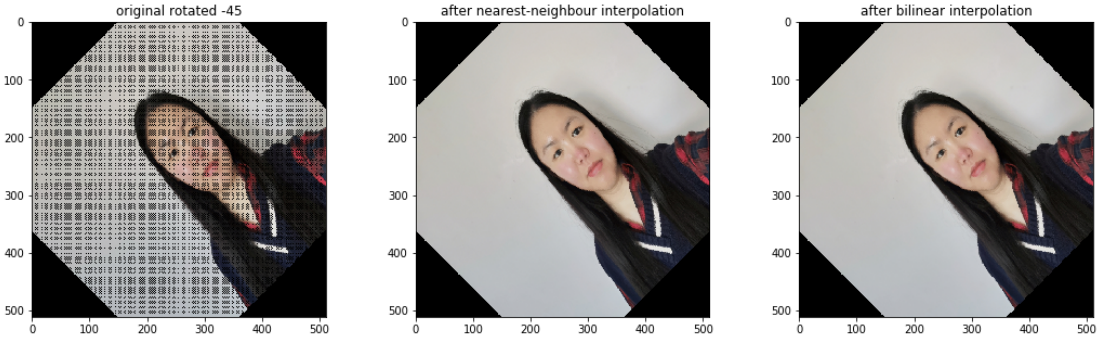
First of all, the nearest neighbour interpolation algorithm is easy to implement and the computational complexity is low. Its principle is to select the pixel from one of the 4 adjacent pixels around the point to be interpolated with the shortest Euclidean distance and assign the point to be interpolated with that gray value. This method only considers the pixel with the shortest distance from the point to be interpolated, without considering the influence of other surrounding points, so that the gray value of the image obtained after interpolation may be discontinuous, causing the image to be blurred.

The core idea of bilinear interpolation is to perform linear interpolation in two directions. The principle is to determine the corresponding weight according to the distance between the point to be sampled and the four neighbouring points, so as to calculate the pixel value of the point to be sampled. Assuming the location of the point to be sampled is (x, y) and the four surrounding integer point is (x_0, y_0), (x_1, y_0), (x_0, y_1), and (x_1, y_1), where $x_0 = \lfloor x \rfloor$, $y_0 = \lfloor y \rfloor$, $x_1 = x_0 + 1$, and $y_1 = y_0 + 1$. Then using the

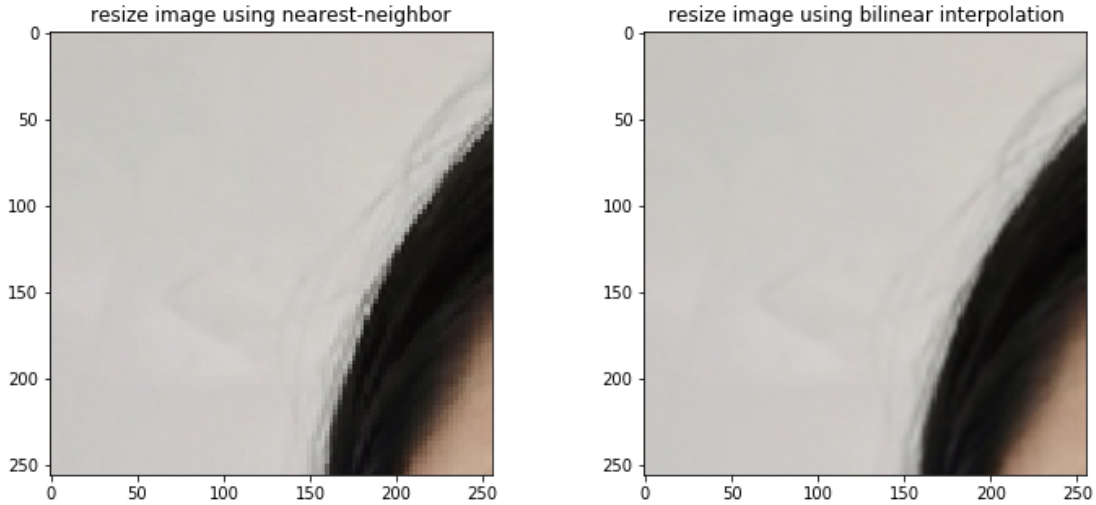
interpolation function to calculate the final values. Therefore, the computational complexity is higher than nearest neighbour interpolation. The gray value of the target point is obtained by averaging the surrounding four pixels according to their weight values so that it is continuous which can play an anti-aliasing effect to a certain extent.

In order to be able to compare the two methods visually, applying two implemented function to the rotated image by forward mapping which is shown in the following graph. It can be seen that both methods can fill the “holes” but there is almost no difference between the output image. Therefore, for further comparison, applying the inbuilt function ***cv2.resize()*** to select a region of the image and enlarge it with different interpolation methods to obtain two new images in Graph 21. There is obvious jaggedness where the gray level changes in the lower right corner of the first picture which is resized by nearest-neighbour, causing block effect due to its discontinuous. In contrast, another image is smoother which is applied of bilinear interpolation. However, despite it is continuity, its first derivative is discontinuous because it does not consider the effect of the rate of the gray value change between adjacent points. In this case, some details of the image will be lost after the enlargement, and the image will become blurred.

Graph 20. Rotated images using different interpolation methods



Graph 21. Resized images using different interpolation methods



Reference

https://docs.opencv.org/master/d1/db7/tutorial_py_histogram_begins.html
https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html
https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm
http://www.adeveloperdiary.com/data-science/computer-vision/how-to-implement-sobel-edge-detection-using-python-from-scratch/
https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic2.htm

Appendices

my_Gauss_filter(noisy_image, size, sigma)

```
# gaussian filter implement for removing noise by caculating convolution

def my_Gauss_filter(noisy_image, size, sigma):

    kernel_x = cv2.getGaussianKernel(size, sigma).reshape(size, 1)
    kernel_y = cv2.getGaussianKernel(size, sigma).reshape(1, size)
    kernel =    kernel_x * kernel_y
    out = caculate_conv(noisy_image, kernel)
    return out

# caculating convolution by dot and sum
# add padding for input in order to ensure the size of output is the same as input

def caculate_conv(inputs, mask):

    out = np.zeros(inputs.shape)
    size = mask.shape[0]
    pad_len = math.floor(size / 2)

    # (rows, cols) -> (rows + 2, cols + 2)
    inputs_pad = np.pad(inputs, ((pad_len, pad_len), (pad_len, pad_len)),'constant',
                        constant_values=(0, 0))
    h, w = inputs_pad.shape

    for i in range(h - size + 1):
        for j in range(w - size + 1):
            out[i, j] = np.sum((inputs_pad[i: i + size, j: j + size]) * mask)

    return out
```

sobel_filter(image, direction = 'both')

```
# sobel filter implement for edge detection in different direction by caculating convolution

def sobel_filter(image, direction = 'both'):
    print("processing sobel filter along", direction)
```

```
    mask_v = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    mask_h = mask_v.T
    filter_h = np.abs(caculate_conv(image, mask_h)).astype(np.uint8)
    filter_v = np.abs(caculate_conv(image, mask_v)).astype(np.uint8)
    filter_all = (filter_h + filter_v).astype(np.uint8)

    if direction == 'h':
        return filter_h
    elif direction == 'v':
        return filter_v
    return filter_all

# caculating convolution by dot and sum
# add padding for input in order to ensure the size of output is the same as input

def caculate_conv(inputs, mask):

    out = np.zeros(inputs.shape)
    size = mask.shape[0]
    pad_len = math.floor(size / 2)

    # (rows, cols) -> (rows + 2, cols + 2)
    inputs_pad = np.pad(inputs, ((pad_len, pad_len), (pad_len, pad_len)),'constant',
                        constant_values=(0, 0))
    h, w = inputs_pad.shape

    for i in range(h - size + 1):
        for j in range(w - size + 1):
            out[i, j] = np.sum((inputs_pad[i: i + size, j: j + size]) * mask)

    return out
```

my_rotation(image, angle, mapping = 'f')

```
# rotate image according to give angle using different methods
# out: rotetion output of forward mapping
# out_b: rotetion output of backward mapping

def my_rotation(image, angle, mapping = 'f'):
    print("processing rotating for", angle)

    h, w, c = image.shape
    cx = h / 2
```

```
    cy = w / 2
    out = np.zeros(image.shape)
    out_b = np.zeros(image.shape)
    r_matrix = rotation_matrix(angle)

    for i in range(h):
        for j in range(w):

            # rotation center is the center of the image

            xy = np.dot(np.array([i - cx, j - cy]), r_matrix)
            x = int(xy[0] + cx)
            y = int(xy[1] + cy)

            # the piexls of the rotated which are out the bound of image can't be
              shown

            if x > -1 and x < h and y > -1 and y < w:
                out[x, y, :] = image[i, j, :]

            xy_b = np.dot(np.array([i - cx, j - cy]), np.linalg.inv(r_matrix))
            x_b = int(xy_b[0] + cx)
            y_b = int(xy_b[1] + cy)

            if x_b > -1 and x_b < h and y_b > -1 and y_b < w:
                out_b[i, j, :] = image[x_b, y_b, :]


    if mapping == 'b':
        return np.uint8(out_b)

    return np.uint8(out)

# 2d rotation matrix

def rotation_matrix(angle):

    angle_pi = angle * math.pi/180.0
    cos = math.cos(angle_pi)
    sin = math.sin(angle_pi)

    return np.array([[cos, -sin],
                     [sin, cos]])
```

nearest_inter(image_in, image_out, angle)

```
# nearest-neighbor interpolation implement
# rounds real-valued coordinates calculated by a geometric transformation to their nearest integers

def nearest_inter(image_in, image_out, angle):
    print("processing nearest-neighbor interpolation for", angle)

    rot_matrix = rotation_matrix(angle)
    h, w, c = image_out.shape
    cx = h / 2
    cy = w / 2
    out = image_out.copy()

    for i in range(h):
        for j in range(w):
            if image_out[i, j].all() == 0:
                xy = np.dot(np.array([i - cx, j - cy]), np.linalg.inv(rot_matrix))

                # found out the nearest point

                x = math.ceil(xy[0] - 0.5 + cx)
                y = math.ceil(xy[1] - 0.5 + cy)

                if x > -1 and x < h and y > -1 and y < w:
                    out[i, j, :] = image_in[x, y, :]

    return np.uint8(out)
```

linear_inter(image_in, image_out, angle)

```
# bilinear interpolation implement
# computed as a hyperbolic distance-weighted function of the four pixels in integer positions (x0,y0),
# (x1,y0), (x0,y1), and (x1,y1), surrounding the calculated real-valued position (x,y).

def linear_inter(image_in, image_out, angle):
    print("processing bilinear interpolation for", angle)

    rot_matrix = rotation_matrix(angle)
    h, w, c = image_out.shape
    cx = h / 2
```

```
cy = w / 2
out = image_out.copy()

for k in range(c):
        for i in range(h):
                for j in range(w):
                        if image_out[i, j].all() == 0:
                                xy = np.dot(np.array([i - cx, j - cy]), np.linalg.inv(rot_matrix))
                                x = xy[0] + cx
                                y = xy[1] + cy
                                xf = math.floor(x)
                                yf = math.floor(y)

                                if xf > -1 and xf < h - 1 and yf > -1 and yf < w - 1:

                                        # four pixels in integer positions (x0,y0), (x1,y0), (x0,y1),
                                           and (x1,y1)
                                        point1 = (xf, yf)
                                        point2 = (xf, yf + 1)
                                        point3 = (xf + 1, yf)
                                        point4 = (xf + 1, yf + 1)

                                        # horizontal interpolation

                                        fr1 = (point2[1] - y) * image_in[point1[0], point1[1], k]
                                              + (y - point1[1]) * image_in[point2[0], point2[1], k]
                                        fr2 = (point4[1] - y) * image_in[point3[0], point3[1], k]
                                              + (y - point1[1]) * image_in[point4[0], point4[1], k]

                                        # vertical interpolation

                                        out[i, j, k] = (point3[0] - x) * fr1 + (x - point1[0]) * fr2
return np.uint8(out)
```