Clab-1 Report

ENGN6528

Yifan Zhu
u7560434

17/03/2023

Task-1: Matlab (Python) Warm-up. (2 marks):

Describe (in words where appropriate) the result/function of each of the following commands of your preferred language in report. Please utilize the inbuilt help() command if you are unfamiliar with these functions.

Note: Different from Matlab, Python users need to import external libraries by themselves. And we assume you already know some common package abbreviations (e.g. numpy = np). (0.2 marks each)

Python

(1) a = np.array([[1, 2, 3], [5, 2, 20]])

Create a 2*3(rows * cols) numpy array, with initialization of the first row with values [1, 2, 3] and the second row with values [5, 2, 20].

(2) b = a[0, :]

Description: Select all columns of the first row of array **a** then assign it to new variable **b**.

(3) f = np.random.randn(200, 1)

Description: Generate an array with 200 rows and 1 column containing random floats from a normal distribution of mean 0 and variance 1, and assign it to variable **f**.

(4) g = f[f > 0]

Description: Choose all elements in **f** which are greater than 0, and then assign them to **g**.

(5) x = np.zeros(20) + 0.5

Description: First generate an array of size 20 and fill it with 0s. Then add 0.5 to each element, and assign it to **x**.

(6) y = 0.5 * np.ones([1, len(x)])

Description: First generate an array of one row and length of **x** columns and fill it with 1s. Then multiply every element with 0.5, and assign it to **y**.

(7) z = x + y

Description: Add **x** and **y** together. Because of the same size of **x** and **y**, just find corresponding elements in both arrays and add them together, then assign the result to **z**. For example, add 1st row and 1st column elements in both **x** and **y**, then assign this result to 1st row and 1st column of **z**.

(8) a = np.linspace(1, 100)

Description: Create an array with evenly spaced numbers from 1 to 100, default generating 50 numbers, and then assign it to **a**.

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

Description: -1 means from the end index of **a** to the start index. This operation means assigning **a** with a reverse order to **b**.

(10) b[b < 25] = 0

Description: Choose all elements in **b** which is less than 25, and then assign these elements to 0.

Hint:

Do the necessary typecasting (uint8 and double) when processing and displaying the image data in the following tasks. For Python, please be aware of the default datatype of different libraries (e.g. Image, matplotlib, cv2). An improper datatype of image will cause many troubles when you want to display the image.

Task-2: Basic Image I/O (2 marks)

In this task, you are asked to:
Given the three images shown in Figure 1(you can find them in the CLAB folder on Wattle):
1. Read those images, and save them to JPG image files named 'image1.jpg', 'image2.jpg' and 'image3.jpg'. (0 marks).
2. Using image1.jpg, develop short computer code that does the following tasks:

a. Read this image from its JPG file, and resize the image to 384 x 256 in columns x rows (0.2 marks).

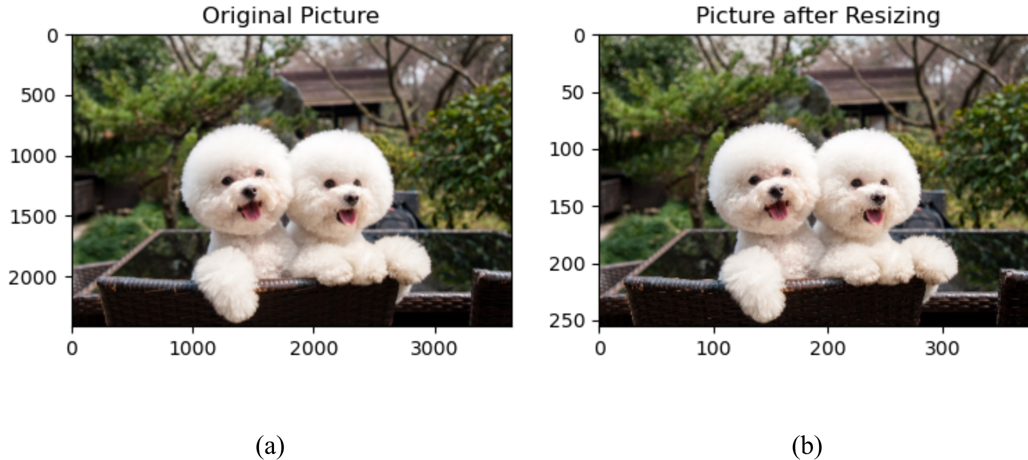
cv2 provides a way to read pictures and resize them. First, use cv2.imread(path) to read an image where the parameter 'path' refers to the path of target picture. Then using cv2.resize(src, disze, …) to resize the pictures. To be noticed that when using cv2.imread(), the image will be read in BGR form. In order to show the image properly, should use cv2.cvtColor() function with parameter 'cv2.COLOR_BGR2RGB' to transform BGR image into RGB one. All images are displayed by matplotlib.pyplot as plt function. Codes are shown in Figure 2.1 and outcomes are shown in Figure 2.2.

```
img = cv2.imread('image1.jpg')  # read in BGR
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title('Original Picture')

# a. Resize img to 384*256
plt.subplot(1, 2, 2)
img2 = cv2.resize(img, (384, 256))
```

```
plt.imshow(img2)
plt.title('Picture after Resizing')
plt.show()
```

Figure 2.1 Code of Reading and Resizing Image

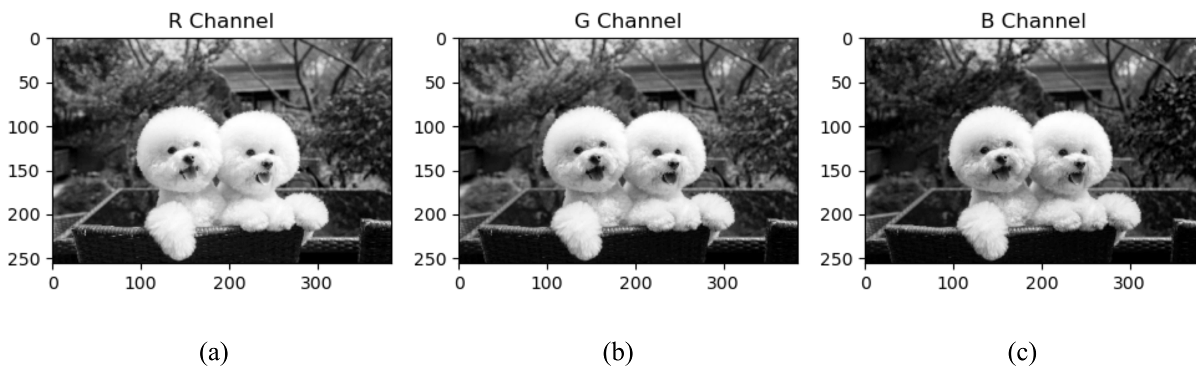


(a) (b)

Figure 2.2 Reading image 'image1.jpg' and resizing it. (a) Original image after reading (b) Resize former image into 384x256. It is hard to distinguish them visually because they look the same size. Therefore, there are two axes besides pictures to indicate their actual size. Alternatively, codes in Figure 2.1 indicate how to resize the images.

b. Convert the colour image into three grayscale channels, i.e., R-channel, G-channel, B-channel images, and display each of the three grayscale images separately (0.2 marks for each channel, 0.6 marks in total).

After reading images with cv2.imread, the image is stored in a 3-dimensional array. The 3rd dimension ranges from 0 to 2 and refers to B channel, G channel and R channel respectively. Because we have used cv2.COLOR_BGR2RGB before, now 3rd array refers to R, G and B channels respectively. Grayscale channel images are shown in Figure 2.3.



(a) (b) (c)

Figure 2.3 Three gray channels of the color image. (a) Grayscale image of R channel (b) Grayscale image of G channel (c) Grayscale image of B channel.

c. Compute the histograms for each of the grayscale images, and display the 3 histograms (0.2 marks for each histogram, 0.6 marks in total).

Matlplotlib.pyplot provide a hist function that we can directly use to draw a histogram of an image. Firstly, because np.hist() only accepts one dimension as the first parameter, we use numpy.ravel() applying to the image, to make the image into a contiguous flattened array[1]. Then we can assign a number to the bin as needed to make the histogram better displayed. Histograms for each grayscale image are shown in Figure 2.4.

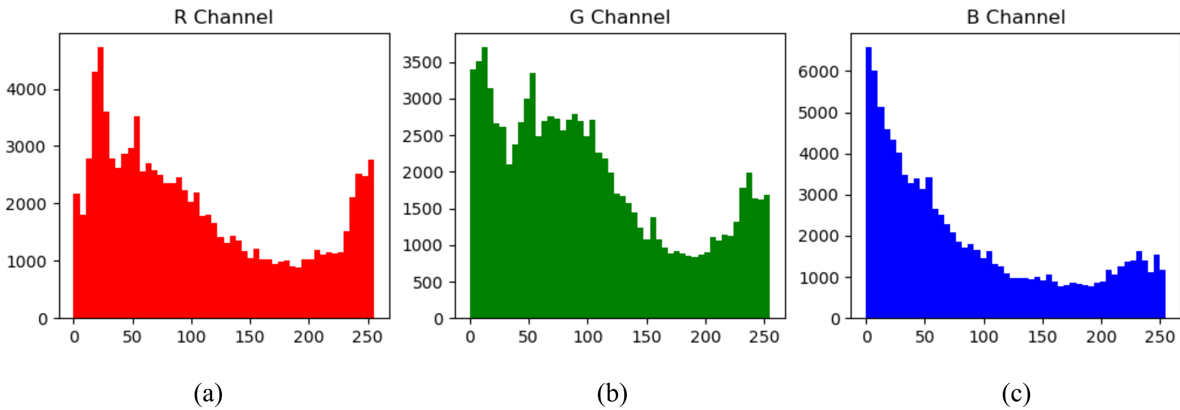
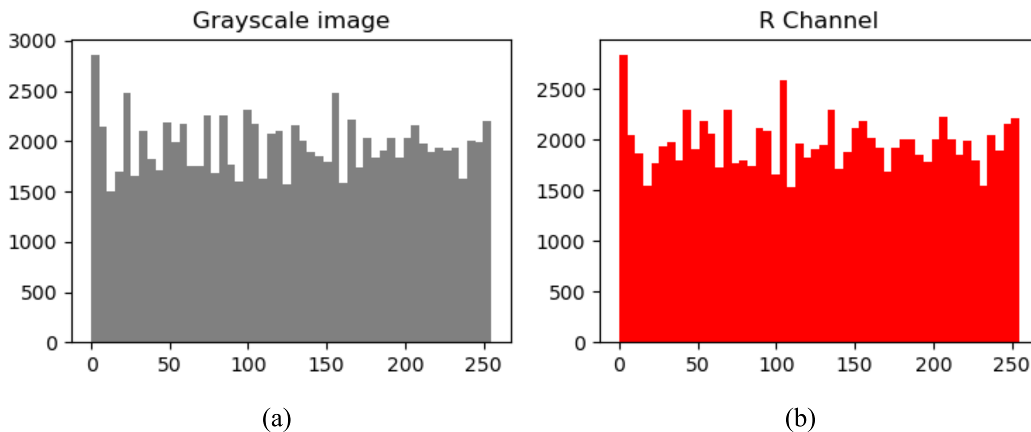
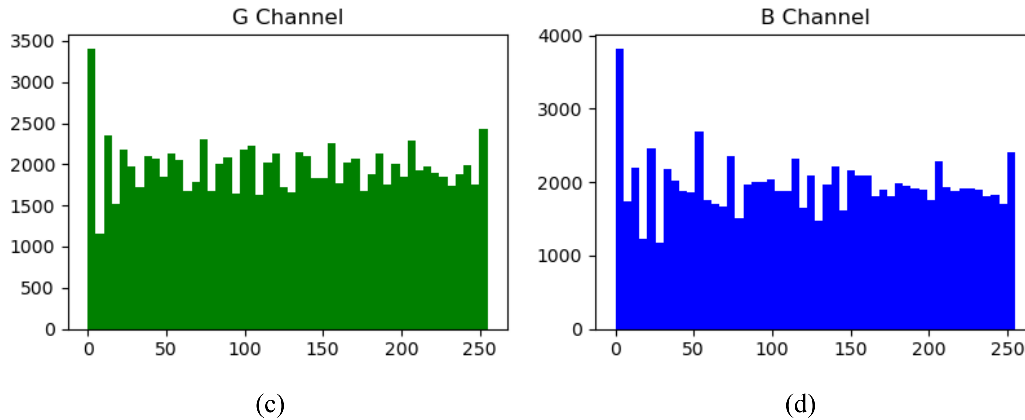


Figure 2.4 Three histograms of grayscale images. (a) Histogram of grayscale R channel image (b) Histogram of grayscale G channel image (c) Histogram of grayscale B channel image.

d. Convert the colour image to a grayscale image and then apply the histogram equalisation to this grayscale image, and the R-channel, G-Channel, and B-Channel images (mentioned in b.), respectively. Then display the 4 images after applying the histogram equalisation process, and their corresponding histograms. (0.15 marks for each (image, histogram pair), 0.6 marks in total). (Hint: you can use inbuilt functions for implementing histogram equalisation. e.g. histeq() in Matlab or cv2.equalizeHist() in Python).

In order to convert a colour image into a grayscale image, we use the function cv2.cvtColor() with the parameter cv2.COLOR_RGB2GRAY. Then we use the inbuild function cv2.equalizeHist() to apply equalisation of all histograms. After equalisation, histograms are shown in Figure 2.5.





(c) (d)

Figure 2.5 Histograms after equalisation. (a) Histogram of grayscale image after equalisation (b) Histogram of R channel image after equalisation (c) Histogram of G channel image after equalisation (d) Histogram of B channel image after equalisation. Different from Figure 2.4, histograms are almost evenly distributed from 0 to 250.

Task-3: Image Denoising via a Gaussian Filter (5 marks)

1. Read in image2.jpg. Crop a square image region corresponding to the central part of the image, resize it to 512×512, and save this square region to a new grayscale image. Please display the two images. Make sure the pixel value range of this new image is within [0, 255] (0.5 marks).

As mentioned in task 2, we use cv2.imread() to read the image and transfer the colour channels. Then we need to discuss how to crop an image from the central part. First, we calculate the length of the cropping square (represented as l_s, where length and width represent the *length* and *width* of the image respectively) with Formula 3.1.

$$l_s = \min(length, width) \tag{3.1}$$

Then we use this length, and Formula 3.2 to get 4 points of the square. x_0 and y_0 refer to the centre coordinate of the image.

$$\begin{cases} p(x_0, y_0) = (x_0 - l_s, y_0 - l_s) \\ p(x_0, y_1) = (x_0 - l_s, y_0 + l_s) \\ p(x_1, y_0) = (x_0 + l_s, y_0 - l_s) \\ p(x_1, y_1) = (x_0 + l_s, y_0 + l_s) \end{cases} \tag{3.2}$$

Apply these four coordinate into the image, we can crop it from central. Then we use cv2.resize() to resize this image to 512x512, with function cv2.cvtColor() with cv2.COLOR_RGB2GRAY to save to a new image. Origin image and cropping image are shown in Figure 3.1.

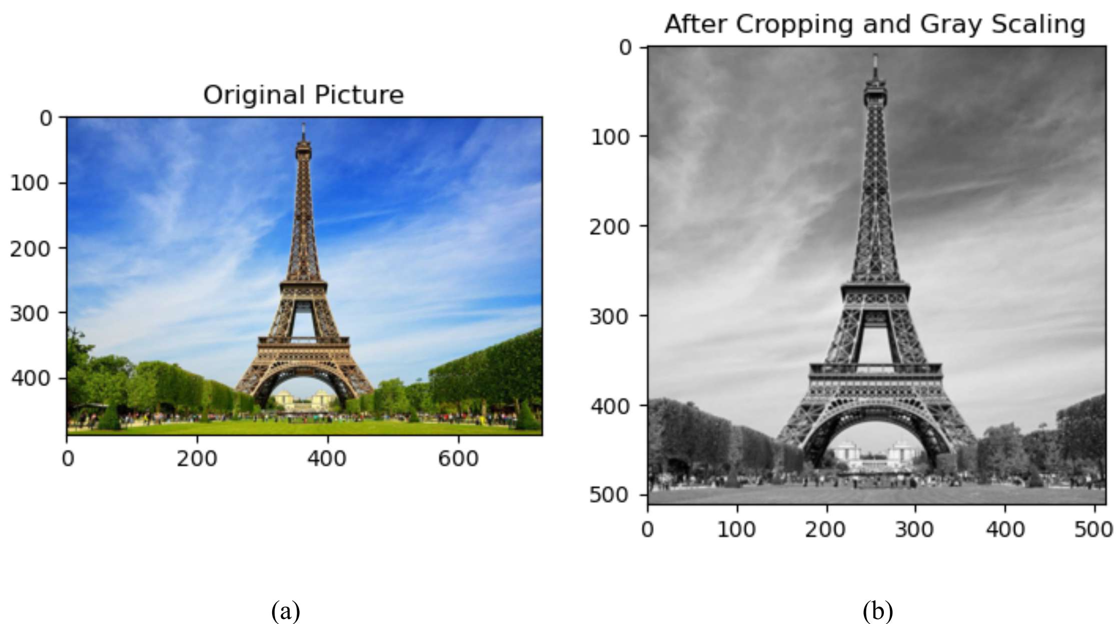


Figure 3.1 Reading a new image, then crop and grayscale it. (a) Original image after reading (b) Cropping the original image into a square image from the central part, then transferring it to a grayscale image.

2. Add Gaussian noise to this new 512x512 image (Review how you generate random number in Task-1). Use Gaussian noise with zero mean, and standard deviation of 15 (0.5 marks). *The intensity values of the Generated image should be within [0, 255].*

Hint: Make sure your input image range is within [0, 255]. Kindly, you may need *np.random.randn()* in Python. While Matlab provides a convenient function *imnoise()*. Please check the default setting of these inbuilt function.

Using np.random.nomal(loc, scale, size), we can generate a group of numbers with Gaussian distribution[2] shown in Formula 3.3.

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.3}$$

In order to generate random numbers with zero mean and standard deviation of 15 with image size, we use np.random.normal(0, 15, img.shape). After generating the noise, directly add it to the image. In order to keep the image range within [0, 255], we use the following code in Figure 3.2 to control.

```
img2New_noise[img2New_noise < 0] = 0
img2New_noise[img2New_noise > 255] = 255
```

Figure 3.2 Make sure values are within [0, 255], by assigning 0 to those less than 0 and 255 to those larger than 255.

Figure 3.3 shows images before and after adding Gaussian noise.

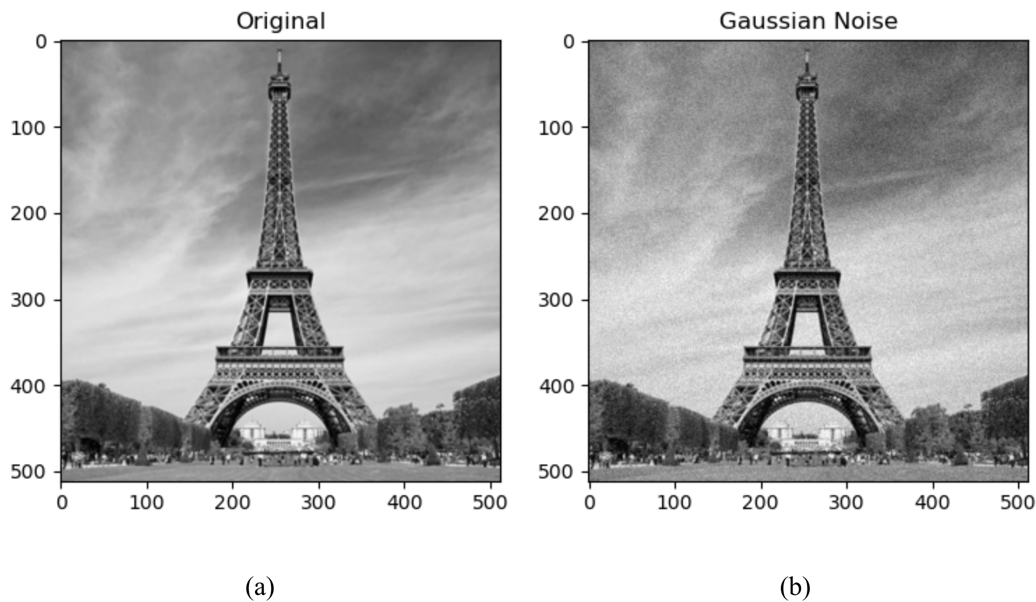


Figure 3.3 Adding Gaussian noise to an image. (a) Original image after cropping and grey scaling (b) Image after adding Gaussian noise. We can observe some noise in this image, especially in the sky area.

3. Display the two histograms side by side, one before adding the noise and one after adding the noise (0.25 marks for each histogram, 0.5 marks in total).

Like task 1, we use plt.hist() to draw histograms and ravel() to make the image into a contiguous flattened array. Histograms a shown in Figure 3.4.

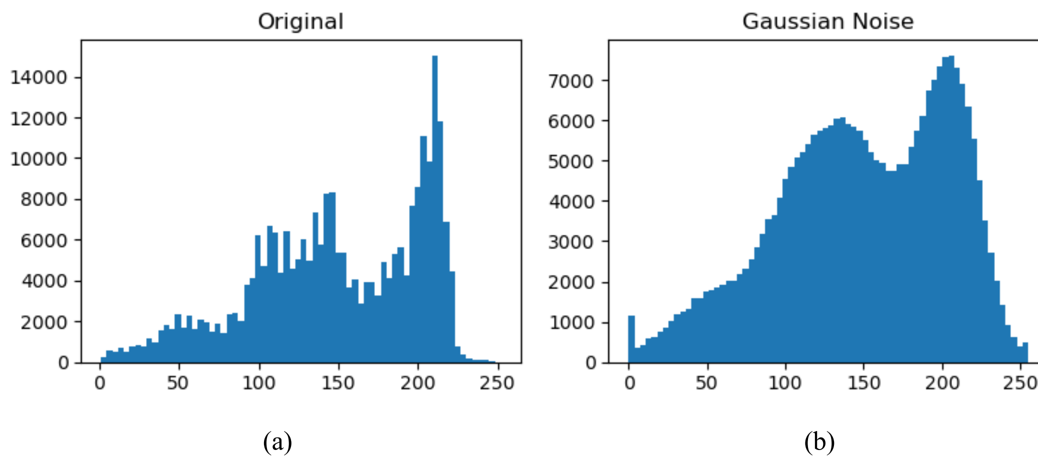


Figure 3.4 Histograms showing the difference between before and after adding Gaussian noise. (a) Before adding Gaussian noise (b) After adding Gaussian noise. It shows that after applying Gaussian noise to the image, the histogram appears less jagged.

4. Implement your own Matlab/Python function that performs a 5x5 Gaussian filtering (1.5 marks). Your function interface is:

my_Gauss_filter()

input: ***noisy_image, my 5x5 gausskernel***

output: ***output_image***

There are two functions below. One is for calculating the sum of the kernel for convenience, the other one is the main function to apply Gaussian filter. The way to apply Gaussian filters is followed by Formula 3.4. Where $T(i, j)$ represent the target pixel with coordinate (i, j), P represents kxk window with the centre of (i, j) of the image, and K represent kxk kernel with the size of k^2.

$$T(i,j) = \sum_{m=0,n=0}^{k} P(m,n) \times K(m,n) \tag{3.4}$$

All codes for both functions are shown in Figure 3.5.

```
# Calculate the sum of kernel
def kernel_sum(kernel):
    s = 0
    for i in range(0, kernel.shape[0]):
        for j in range(0, kernel.shape[1]):
            s += int(kernel[i][j])
    return s


def my_Gauss_filter(noisy_image, kernel):
    # Assume kernel must be square
    if kernel.shape[0] != kernel.shape[1]:
        return 'Kernel is not square!'
    # calculate the coordinate of kernel central
    centre = kernel.shape[0] // 2
    # noise pics length and high
    length = noisy_image.shape[0]
    high = noisy_image.shape[1]
    # kernel length
    k_length = kernel.shape[0]
    # initiate output
    output_image = np.zeros((length, high))
    kernel_s = kernel_sum(kernel)
    # Calculate
    for i in range(centre, length-centre):
        for j in range(centre, high-centre):
            c = 0
            for k in range(0, kernel.shape[0]**2):
                c += noisy_image[i - centre+(k // k_length)][j - centre+(k %
k_length)] * int(kernel[k // k_length][k % k_length])
            output_image[i][j] = c // kernel_s
    return output_image
```

Figure 3.5 Design my Gaussian function

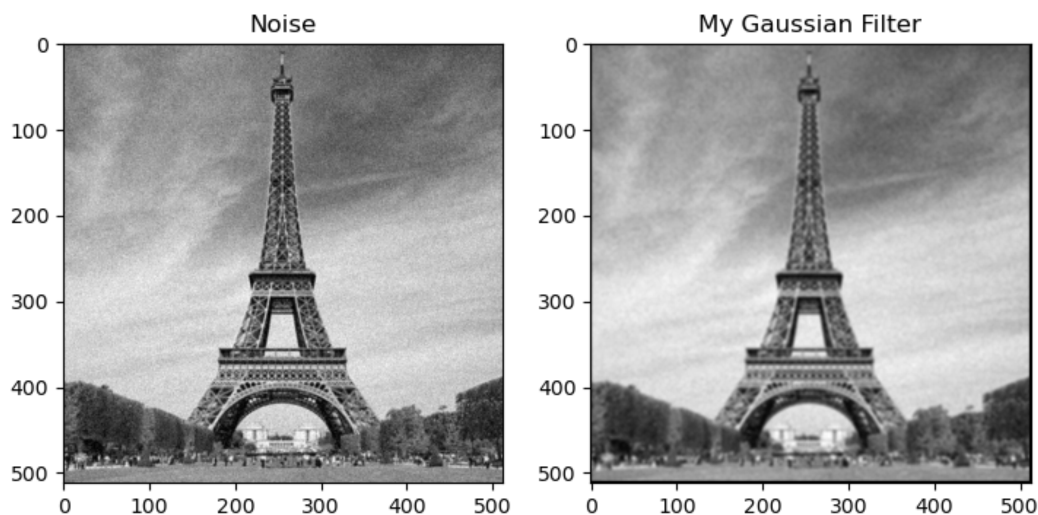
Also, there are two functions to generate different square sizes and different deviations of kernels. If we want to get 5x5 kernel with deviation of 2, we can use build_kernel(5, 2) to build it. All codes to generate a kernel are shown in Figure 3.6.

```python
# gauss function: e^(-x^2+y^2 / 2 sig^2)/(2 pi sig^2)
# x,y: coordinate, sig: standard deviation
# return the value of gaussian function
def gauss_func(x, y, sig):
    #print(x, y)
    return math.exp(-(x**2 + y**2) / (2*(sig**2))) / (2 * math.pi * (sig**2))
# kernel build function
# num: define the size of kernel, sig: standard deviation
# return kernel
def build_kernel(num, sig):
    # the size of kernel must be odd
    if num%2 != 1:
        return 'not odd'
    centre = num // 2
    #print(centre)
    kernel = np.zeros((num, num))
    for i in range(0, num):
        for j in range(0, num):
            # using gauss function to calculate
            kernel[i][j] = gauss_func(-centre+i, -centre+j, sig)
    return kernel / kernel[0][0]
```

Figure 3.6 Functions to generate kernels

5. Apply your Gaussian filter to the above noisy image, and display the smoothed images and visually check their noise-removal effects (0.5 marks in total).

Apply my Gaussian filter by 5x5 kernel with deviation of 2. The noise image and image are shown in Figure 3.7.



(a) (b)

Figure 3.7 Applying my Gaussian filter to the noisy image. (a) noisy image before applying Gaussian filter (b) smoothed image by using my Gaussian filter. We can observe that after applying the filter, the image becomes blurry and has removed some noises.

One of the key parameters to choose for the task of image filtering is the standard deviation of your Gaussian filter. You may need to test and compare different Gaussian kernels with different standard deviations (1.0 marks).

By using build_kernel(num, sig) we can generate different kernels with different standard deviations. We apply sigma with 0.1, 0.5, 1, 5, 10, 50 and 100 and results are shown in Figure 3.8 respectively.

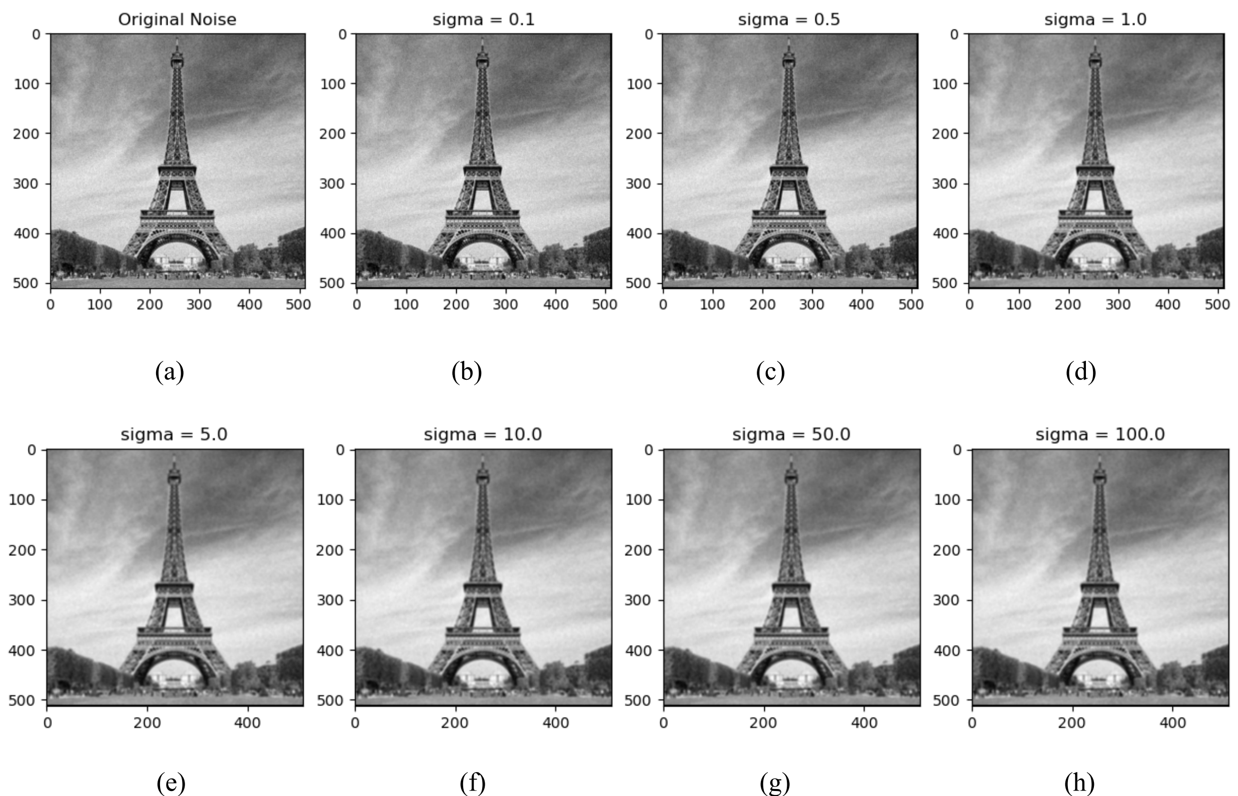


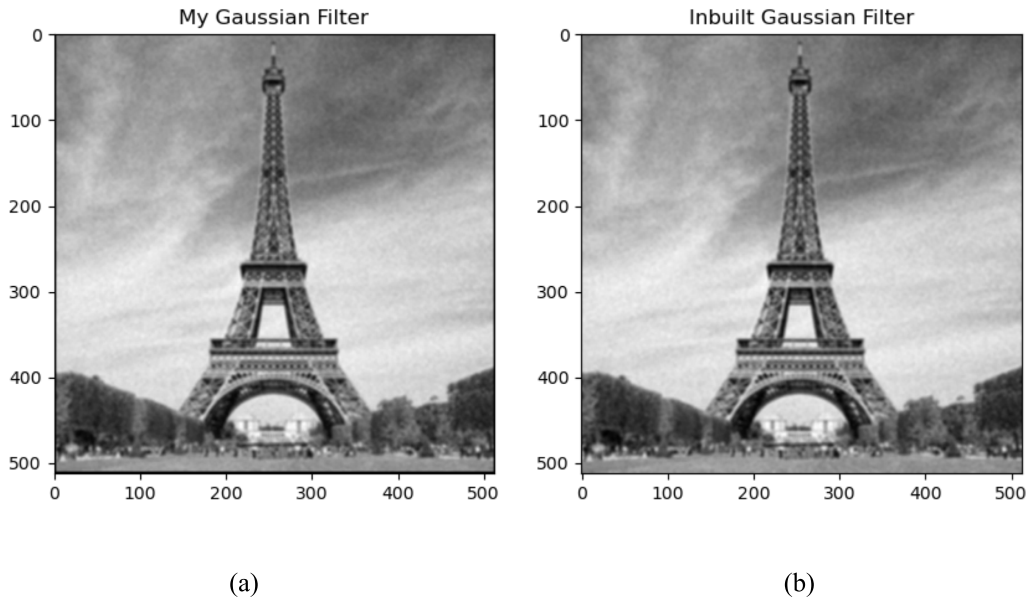
Figure 3.8 The contrast between filtered images after applying Gaussian filter with different sigma. (a) original image (b) sigma = 0.1 (c) sigma = 0.5 (d) sigma = 1 (e) sigma = 5 (f) sigma = 10 (g) sigma = 50 (h) sigma = 100. We can observe that with sigma growing larger, the more blurry the image becomes and the less noise the image has.

Note: In doing this task you MUST NOT use any Matlab's (or Python's) inbuilt image filtering functions (e.g. *imfilter()*, *filter2()* in Matlab, or *cv2.filter2D()* in Python). In other words, you are required to code your own 2D filtering code, based on the original mathematical definition for 2D convolution. However, you are allowed to generate a 5x5 sized Gaussian kernel with inbuilt functions.

6. Compare your result with that by Python's or Matlab's inbuilt 5x5 Gaussian filter (e.g. conveniently cv2.GaussianBlur() in Python or filter2(), imfilter() in Matlab). Please show whether the two results are nearly identical (0.5 marks).

Further reading material: http://setosa.io/ev/image-kernels/

Applying my_Gauss_filter and inbuild filter cv2.GaussianBlur() together with 5x5 kernel with standard deviation of 2. Figure 3.9 shows the result.



(a) (b)

Figure 3.9 Comparison of my_Gauss_filter() and inbuild filter with 5x5 kernel and deviation of 2. (a) Using my_Gauss_filter() (b) Using inbuild filter cv2.GaussianBlur(). We can see that the two images are almost the same, except there is a black edge by using my_Gauss_filter(), for we did not consider the edge situation and let it remain black, i.e. the size of the image becomes slightly smaller.

Task -4: Implement your own 3x3 Sobel filter in Matlab/Python (3 marks)

`Sobel edge detector.`

You need to implement your own 3x3 Sobel filter. Again, you must not use inbuilt functions or any inbuilt edge detection filter. The implementation of the filter should be clearly presented with your code.

It is similar to designing a Gaussian filter, instead of changing kernels. We use Formula 4.1 to detect vertical edges,

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{4.1}$$

while Formula 4.2 detects horizontal edges.

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{4.2}$$

All codes to write my_Sobel_filter are shown in Figure 4.1.

```
def my_Sobel_filter(image, Vertical=True, threshold=100):
    if Vertical == True:
        sobel = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    else:
        sobel = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

    # calculate the coordinate of kernel central
    centre = 1
    # pics length and high
    length = image.shape[0]
    high = image.shape[1]
    # sobel length
    s_length = 3
    # initiate output
    output_image = np.zeros((length, high))
    # Calculate
    for i in range(centre, length - centre):
        for j in range(centre, high - centre):
            c = 0
            for k in range(0, 9):
                c += image[i - centre + (k // s_length)][j - centre + (k %
s_length)] * int(sobel[k // s_length][k % s_length])
            output_image[i][j] = c
    output_image[output_image <= threshold] = 0
    output_image[output_image > threshold] = 255
    return output_image
```

Figure 4.1 Function to realize Sobel filter. My_Sobel_fliter has three parameters. The parameter 'image' needs to input the source image. Vertical means to display whether vertical edge or horizontal edge, default is vertical. The threshold is a control variable to make the image only display the edge with the brightest pixels while the non-edge with the darkest pixels.

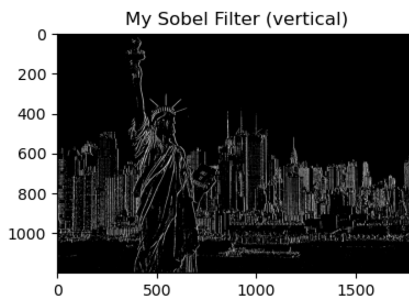
Test it on the image, image3.jpg, and compare your result with inbuilt Sobel edge detection function (0.5 marks),

First, we apply the image to both vertical and horizontal Sobel functions, and use Formula 4.3 to calculate the edge for both directions. In Formula 4.3, S_v refers to using vertical Sobel edge detection while S_h refers to using horizontal edge detection, and S_{vh} is for both directions[3].

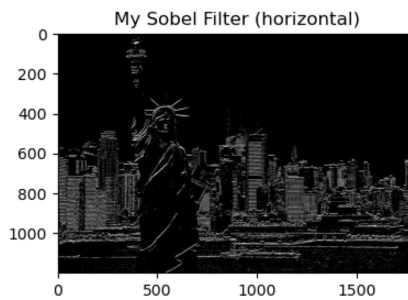
$$S_{vh} = \sqrt{S_v^2 + S_h^2} \tag{4.3}$$

After applying so, results are shown in Figure 4.2.

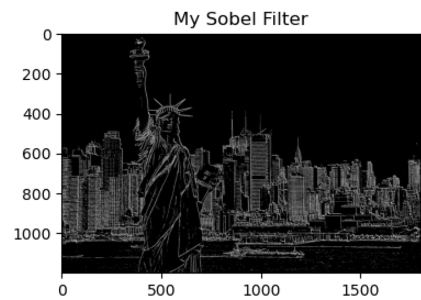
(a)



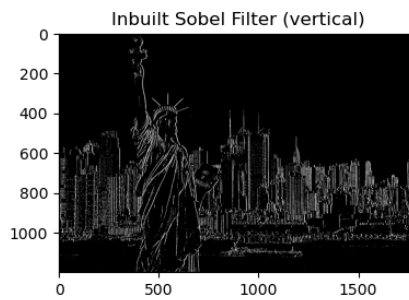
(b)



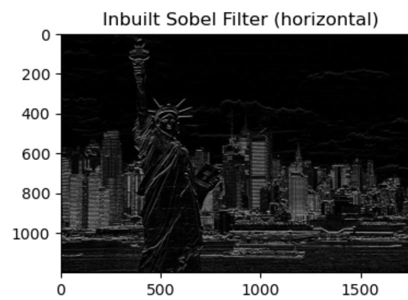
(c)



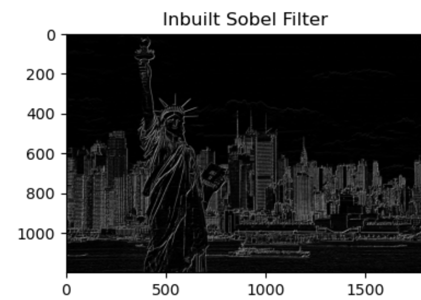
(d)



(e)



(f)



(g)

Figure 4.2 Apply my Sobel filter and inbuilt Sobel filter. (a) Original Grayscale image (b) My Sobel filter on vertical direction (c) My Sobel filter on horizontal direction (d) My Sobel filter in both vertical and horizontal directions (e) Inbuilt Sobel filter in vertical direction (f) Inbuilt Sobel filter in horizontal direction (g) Inbuilt Sobel filter in both vertical and horizontal directions.

Briefly explain the function of Sobel filter and how it can achieve this function (0.5 mark).

The Sobel filter achieves this function by convolving the picture with two kernels, one for the horizontal direction and one for the vertical direction), which are designed to detect changes in intensity along those directions. Sobel filter determines the gradient of the picture intensity along the horizontal and vertical directions, combining these gradient values to create a general indicator of edge strength at each pixel location.

Investigate the accuracy of the Sobel filter in terms of predicting the orientation of the edge. If it is used for edge detection, what might cause any inaccuracy of the edge orientation estimation that you see? Look at the histogram of the gradient orientation and find out the major edge orientations in the image. Please discuss whether the gradient orientation histogram align with respect to the real expected edges for the image or not.

You can use the inbuilt Sobel function to discuss this question. (2.0 mark)

From Figure 4.2, we can see that most edges are displayed properly. It is clearly shown the edge of the Statue of Liberty, the buildings, and the ships. However, there are still some details are missing in Figure 4.3.

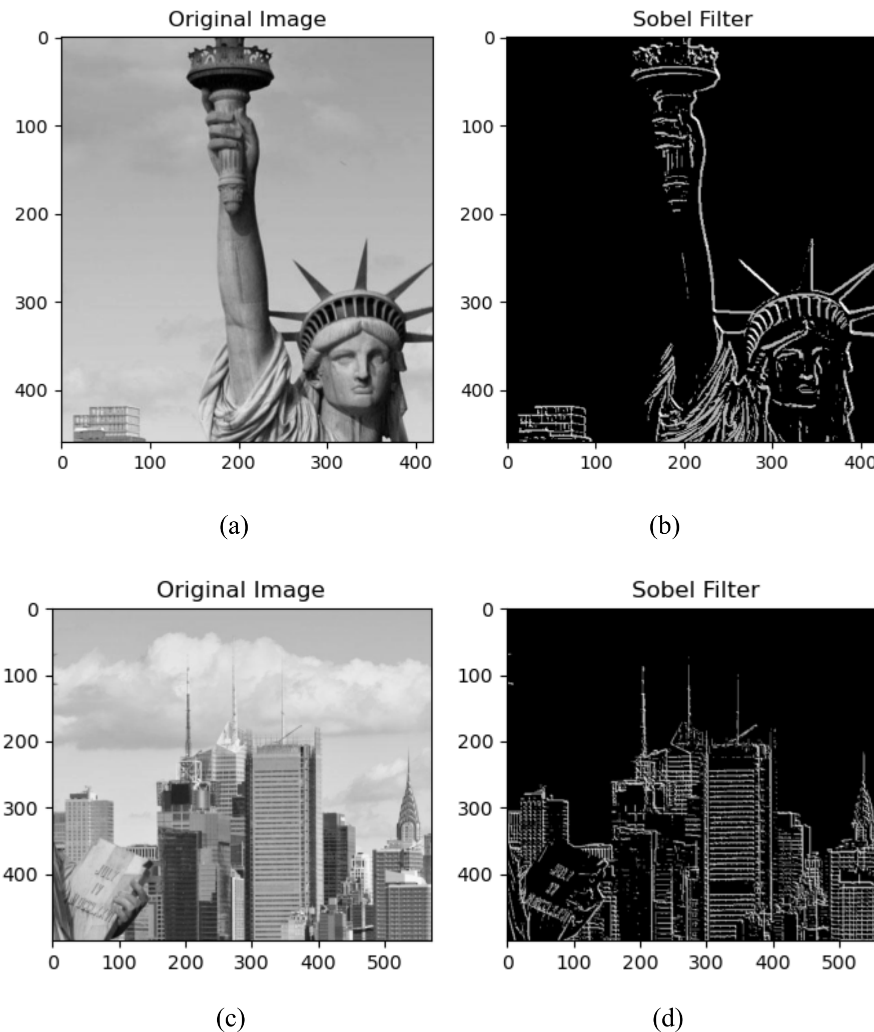


Figure 4.3 Comparison of lost details (a) Original image of the arm of Statue of Liberty (b) After using Sobel filter of the arm of Statue Liberty (c) Original image of the cloud (d) After using Sobel filter of the cloud.

We can see that after using Sobel filter to detect the edge, for the first group of images, the edge of the left side of the arm is missing as well as the right side of the face; for the second group of images, the edge of the cloud is missing. There are several explanations or assumptions according to some studies on Sobel filter[4].

First, we assume that the values of the grey levels could be measured with infinite precision. When the light intensities are digitalized with a finite number of bits of accuracy, errors will be introduced. Besides, edge detector performance will also be impacted by shadows and uneven lighting. Moreover, the object boundaries will typically not be straight. The accepted model is thus simply a rough approximation of the actual situation.

In order to calculate the gradient orientation, we can use cv2.phase() method. It takes four parameters, first for *x* direction, second for *y* direction, third for the output array of angles, and fourth for deciding whether to use angles in degrees or radians. After applying this method to the Sobel filter, we show gradient orientation by using " cmap = 'jet' ", which displays colours from blue to cyan, green, yellow, orange and red with the number increasing. Besides, we draw a histogram of the gradient orientation, and directions are divided into 8 categories. All two images are shown in Figure 4.4.

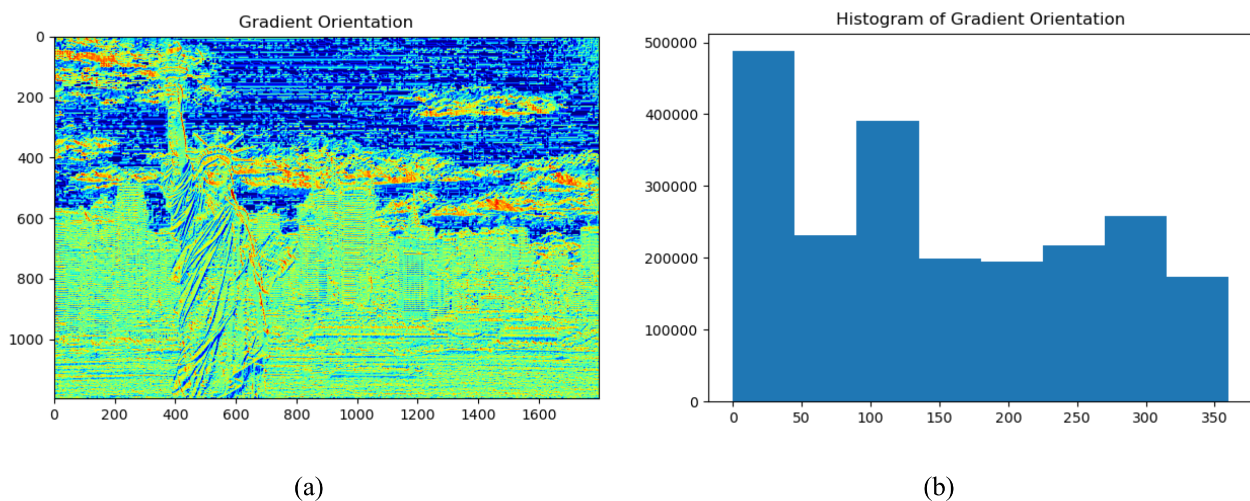


Figure 4.4 Gradient orientation (a) Show gradient orientation in the original image with colour (b) Histogram of gradient orientation

From Figure 4.4(b), we can conclude that the major orientation is horizontal, for the maximum of the histogram is within [0, 45] degrees. It aligns with the real expected edge, for we can distinguish most of the edges from Figure 4.4(a).

Task -5: Image Rotation (3 marks)

Take one frontal face photo of yourself, under normal lighting condition, against a white wall. (i.e., as if a passport photo). The image should be in landscape shape (i.e., the longer side is in the horizontal direction), and resized to 384x 512. (Height x Width)

1. Implement your own function my_rotation() for image rotation by any given angle between [-90°, 90°]. Display images rotated by -90°, -45°, -15°, 45°, and 90° (0.20 for each image, 1.0 mark in total).

Note: positive for clockwise. Negative for anti-clockwise. Eg. -45° means rotate 45° in anti-clockwise direction.

According to rotation rules, we can get a rotation function in Formula 4.1. It is used for forward mapping.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.1}$$

Alternatively, Formula 4.2 shows a way of inverse mapping.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{5.2}$$

In both formulae, *x*' and *y*' mean image after rotation; c_x and c_y are the coordinates of the rotation centre, typically the centre of the image; θ is the rotation degree.

To be easier to code, we transform Formula 5.1 and 5.2 into Formula 5.3 and 5.4.

$$\begin{cases} x' = x\cos\theta - y\sin\theta - c_x\cos\theta + c_y\sin\theta + c_x \\ y' = x\sin\theta + y\cos\theta - c_x\sin\theta - c_y\cos\theta + c_y \end{cases} \tag{5.3}$$

$$\begin{cases} x = x'\cos\theta + y'\sin\theta - c_x\cos\theta - c_y\sin\theta + c_x, \\ y = -x'\sin\theta + y'\cos\theta + c_x\sin\theta - c_y\cos\theta + c_y \end{cases} \tag{5.4}$$

Figure 5.1 shows all codes for the rotating function, including forward or inverse, splatting or not, nearest neighbour or bilinear method. There are 5 parameters. The first is for the image which needs to be rotated. The Second is the rotating angle. The third is whether forward or inverse mapping, default forward. The fourth is whether splatting when forward mapping, with default splatting. The fifth is whether using bilinear or not, default not using.

```
def my_rotation(img, angle, forward = True, splatting = True, bilinear =
False):
    if(angle>-90 and angle<0):
        angle = 360 + angle
    # transfer to radian
    angle = angle * math.pi / 180
    # the length and height of the image
    length = img.shape[1]
    height = img.shape[0]
    # initialize the output
    result = np.zeros(img.shape)
    # forward mapping begins
    if(forward):
        for i in range(0, length):
            for j in range(0, height):
                x = math.cos(angle) * i - math.sin(angle) * j - 0.5 * length
* math.cos(angle) + 0.5 * height * math.sin(angle) + 0.5 * length
                y = math.sin(angle) * i + math.cos(angle) * j - 0.5 * length
* math.sin(angle) - 0.5 * height * math.cos(angle) + 0.5 * height
                # using splatting method. If a point is among pixels, then
splatting it to its 4 neighbours
                if(splatting):
                    # find neighbours
```

```
                    x1 = math.ceil(x)-1
                    x2 = math.floor(x)-1
                    y1 = math.ceil(y)-1
                    y2 = math.floor(y)-1
                    # if pixel is outbound, then discard it
                    if(x1>0 and x1<length and y1>0 and y1<height):
                        result[y1, x1, :] = img[j, i, :]
                        result[y1, x2, :] = img[j, i, :]
                        result[y2, x1, :] = img[j, i, :]
                        result[y2, x2, :] = img[j, i, :]
                else:
                    # not using flatting method. Directly to the nearest
neighbour
                    x1 = int(x)
                    y1 = int(y)
                    if (x1 > 0 and x1 < length and y1 > 0 and y1 < height):
                        result[y1, x1, :] = img[j, i, :]
    # forward mapping end
    # inverse mapping begin
    else:
        for i in range(0, length):
            for j in range(0, height):
                # not using bilinear method. Directly find the nearest
neighbour
                if(not bilinear):
                    x = int(math.cos(angle) * i + math.sin(angle) * j - 0.5 *
length * math.cos(angle) - 0.5 * height * math.sin(angle) + 0.5 * length)
                    y = int(-math.sin(angle) * i + math.cos(angle) * j + 0.5
* length * math.sin(angle) - 0.5 * height * math.cos(angle) + 0.5 * height)
                    if(x-1>0 and x-1<length and y-1>0 and y-1<height):
                        result[j, i, :] = img[y-1, x-1, :]
                else:
                    # using bilinear method
                    x0 = math.cos(angle) * i + math.sin(angle) * j - 0.5 *
length * math.cos(angle) - 0.5 * height * math.sin(angle) + 0.5 * length
                    y0 = -math.sin(angle) * i + math.cos(angle) * j + 0.5 *
length * math.sin(angle) - 0.5 * height * math.cos(angle) + 0.5 * height
                    x1 = math.ceil(x0)
                    x2 = math.floor(x0)
                    y1 = math.ceil(y0)
                    y2 = math.floor(y0)
                    # q11 = (x1, y1), q21 = (x2, y1), q12 = (x1, y2), q22 =
(x2, y2)
                    if(x1>0 and x1<length and x2>0 and x2<length and y1>0 and
y1<height and y2>0 and y2<height):
                        if(x1 != x2 and y1 != y2):
                            f_x_y1 = (x2 - x0)/(x2 - x1) * img[y1, x1, :] +
(x0 - x1)/(x2 - x1) * img[y1, x2, :]
                            f_x_y2 = (x2 - x0)/(x2 - x1) * img[y2, x1, :] +
(x0 - x1)/(x2 - x1) * img[y2, x2, :]
                            f_x_y = (y2 - y0)/(y2 - y1) * f_x_y1 + (y0 -
y1)/(y2 - y1) * f_x_y2
                        elif(x1 == x2 and y1 != y2):
                            f_x_y = (y2 - y0)/(y2 - y1) * img[y1, x1, :] +
(y0 - y1)/(y2 - y1) * img[y2, x1, :]
                        elif(y1 == y2 and x1 != x2):
                            f_x_y = (x2 - x0)/(x2 - x1) * img[y2, x1, :] +
```

```
(x0 - x1)/(x2 - x1) * img[y2, x2, :]
                        else:
                            f_x_y = img[y1, x1, :]
                        result[j, i, :] = f_x_y
    return result
```

Figure 5.1 my_rotation() function

Following shows images rotating -90°, -45°, -15°, 45°, and 90° respectively in Figure 5.2. All use default options discussed before.

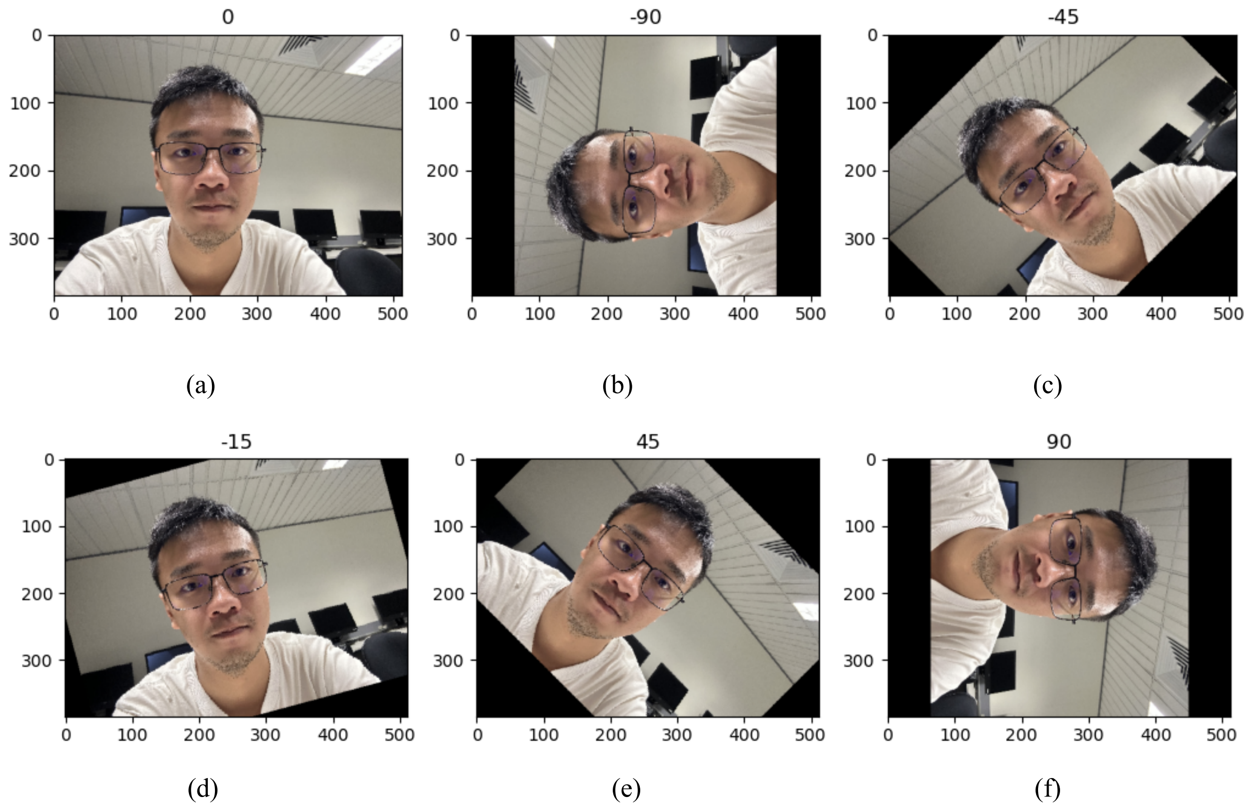
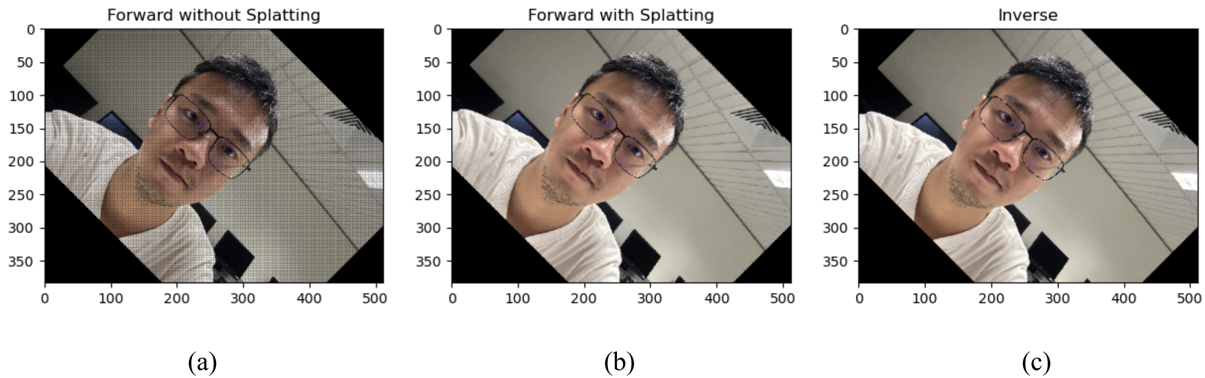


Figure 5.2 Image rotation (a) Original image (b) Rotate image 90 degrees anti-clockwise (c) Rotate image 45 degrees anti-clockwise (d) Rotate 15 degrees clockwise (e) Rotate 45 degrees clockwise (f) Rotate 90 degrees clockwise

2. Compare forward and backward mapping and analyze their difference (1.0 mark).

[Hints: you are required to write my_rotation(), using, forward mapping and backward mapping, respectively. Obtain the images after rotating using two different methods.]

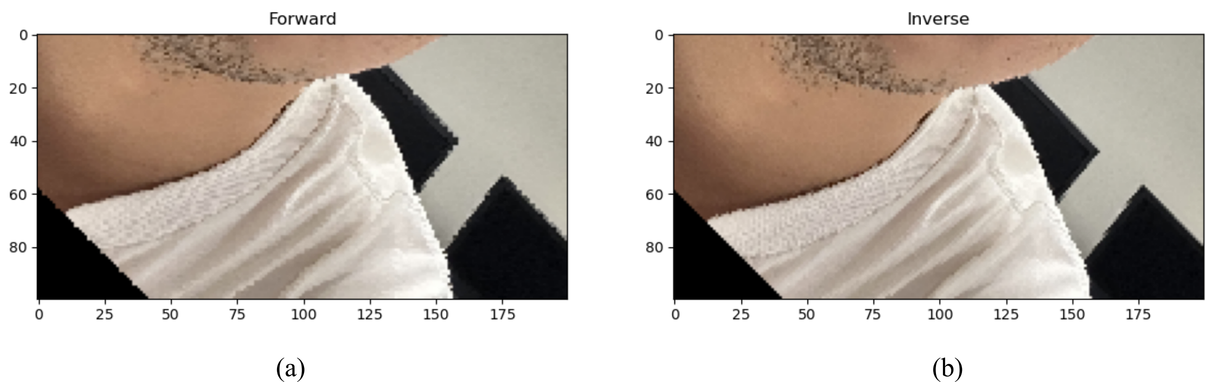
Applying both forward and inverse methods to rotate the image 45 degrees clockwise. First uses forward without flatting, which means when a point is among pixels, just find its nearest neighbour. Second one uses forward with flatting, which means when a point is among pixels, splat this point evenly to all neighbours around it. Last one uses the inverse method, finding the nearest neighbour. All results are shown in Figure 5.3.



(a) (b) (c)

Figure 5.3 Difference between the forward mapping and backward mapping (a) Using forward method but not splatting to neighbours (b) Using forward method with splatting (c) Using inverse method with finding the nearest neighbour

From Figure 5.3, we can discover that when using forward method but without flatting, there are a lot of black holes in the image. After applying splatting, holes are reduced significantly. In order to compare forward and inverse mapping, we magnify some parts of the image, which are shown in Figure 5.4.



(a) (b)

Figure 5.4 Diference between forward and inverse method (a) Forward method (b) Inverse method

From Figure 5.4(a), we can observe the edge of the image. There are some pixels outside the image while 5.4(b) is not. Besides, we can observe the edges of objects in both images, i.e., the edge of the computer monitor, clothes, face and etc. It appears that by using forward method, the edge of the object of the image is much more jagged than inverse method. The quality is better by using inverse method.

3. Compare the methods using bilinear interpolation or using nearest neighbor method in the inverse mapping/warping method and analyze their differences (1.0 mark).

When using inverse mapping, two methods are used if a point is among pixels. First one is the nearest neighbour, which is simply to find the nearest integer of the point. Second method is bilinear method, it is for interpolating functions of two variables using repeated linear interpolation[5]. Figure 5.5 shows the relationship between pixel dots and point we want to interpolate.

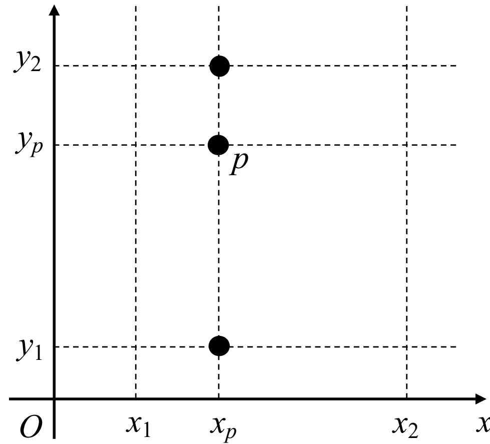


Figure 5.5 Bilinear method. p is the point we need to interpolate

Calculation method followed by Formula 5.5 and 5.6.

$$f(x,y) = a_{00} + a_{10}x + a_{01}y + a_{11}xy \tag{5.5}$$

Where,

$$\begin{cases} a_{00} = f(x_1, y_1), \\ a_{10} = f(x_2, y_1) - f(x_1, y_1), \\ a_{01} = f(x_1, y_2) - f(x_1, y_1), \\ a_{11} = f(x_2, y_2) - f(x_2, y_1) - f(x_1, y_2) + f(x_1, y_1) \end{cases} \tag{5.6}$$

Here are the results for finding the nearest neighbour and using bilinear method for inverse method in Figure 5.6.

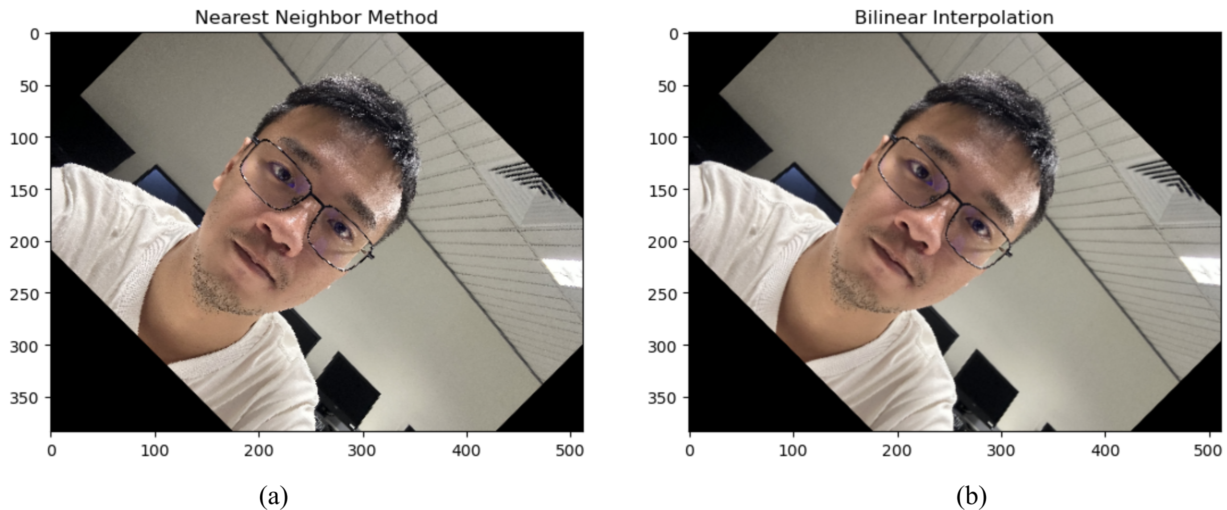


Figure 5.6 Inverse mapping using different methods (a) Using nearest neighbour method (b) Using bilinear interpolation

In order to see the difference clearly, we crop some parts of the image and magnify it to get a better view, in Figure 5.7.

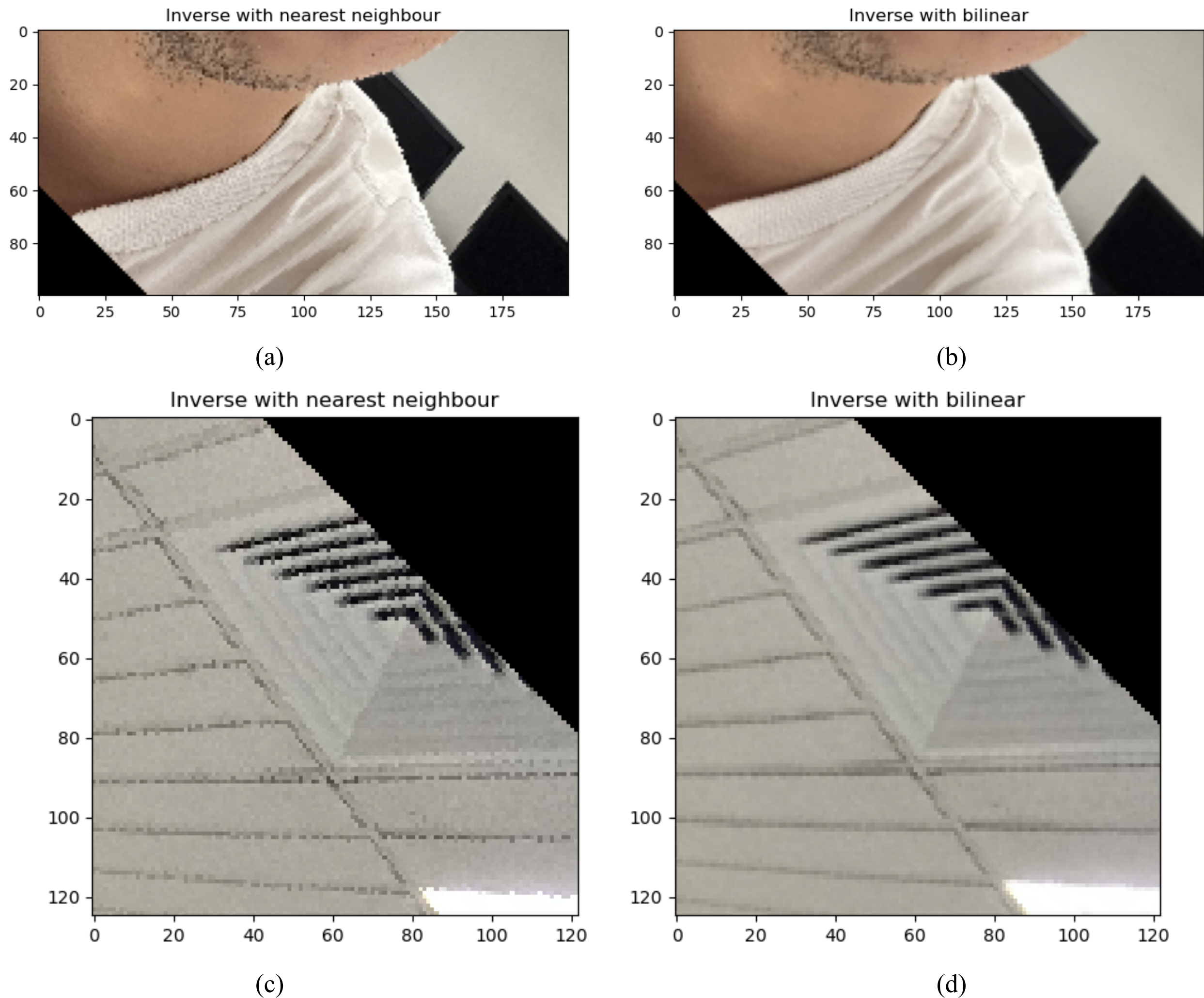


Figure 5.7 Different parts of the image after magnifying (a) Part one of nearest neighbour method (b) Part one of bilinear interpolation (c) Part two of nearest neighbour method (d) Part two of bilinear interpolation

From Figure 5.7(b)(d), compared to 5.7(a)(c), we can see the edge of the monitor, clothes and face is much smoother by using bilinear interpolation method. Besides, we can see details are clearer by bilinear methods, like the beard in 5.7(b), and the ventilator and patterns on the ceil in 5.7(d).

Above all, we can conclude that by using inverse mapping with bilinear interpolation method, we can get a better result.

Reference:

[1] NumPy Developers. *numpy.ravel()*. Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.ravel.html.

[2] NumPy Developers. *numpy.random.normal()*. Retrieved from https://numpy.org/doc/stable/reference/random/generated/numpy.random.normal.html.

[3] Ramesh Jain, Rangachar Kasturi, Brian G. Schunck. (1995). Edge Detection. In Machine Vision. McGraw-Hill, Inc.

[4] Kittler, J. (1983). On the accuracy of the Sobel edge detector. Image and Vision Computing, 1(1), 37-42.

[5] Bilinear Interpolation. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Bilinear_interpolation.