Clab-1 Report

ENGN6528

Han Zhang
u7235649

28/03/2021

Task-1: Python Warm-up

1. a = np.array([[2, 3, 4],[5, 2, 200]])

Initialize a 2-dimension array “a”, the numbers in the first array are 2, 3, 4, and the numbers in the second array are 5, 2, 200.

2. b = a[: , 1]

Let “b” be a list of the second elements of all the arrays in “a”.

3. f = np.random.randn(400,1)+3

Let “f” as an array of shape 400 * 1 with a random number of the standard normal distribution and add 3 to each number.

4. g = f[f > 0]*3

Let “g” as the array of all the elements in “f” greater than 0 that multiplied by 3.

5. x = np.zeros (100) + 0.45

Let “x” be an array of 100 zeros and for each plus 0.45. So “x” is a 1D array of 100 0.45.

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

Let “y” be an array with value 1 whose shape is 1 times the length of “x” multiplied by 0.5. So “y” is an array of shape 1 times the length of “x” with all values 0.5.

7. z = x + y

Let “z” be the array that adds each corresponding value of “x” and “y”. The size of “z” is 1 times the length of “x”.

8. a = np.linspace(1, 499, 250, dtype=int)

Let “a” be the array of 250 evenly spaced numbers, calculated over the interval 1 to 499 (included), the data type of the numbers is integer. The interval is (499 – 1) / (250 – 1) = 2, so “a” is an array of 1, 3, 5, …, 497, 499.

9. b = a[: :-2]

Let “b” be the array of all the first numbers of every 2 numbers in the inverse of “a”. That is, take all the values from the reverse of “a” by index with a step size 2, set them to “b” as a list.

10. b[b >50]=0

Set all values in “b” which is greater than 50 to 0.

Task-2: Basic Coding Practice

1. Load a grayscale image (such as Atowergray.jpg from wattle), and map the image to its negative image, in which the lightest values appear dark and vice versa. Display it side by side with its original version.

Read the image “Atowergray.jpg” using function *imageio.imread()*. Since the image is a grayscale image so it is a 2-D matrix, all the values are in range [0, 255]. To map the image to its negative image, use 255 to minus each value.

The result is like Fig. 1 below. On the left is the original image and on the right is the negative image. The bright pixels in the original image becomes dark pixels in the negative image and vice versa.

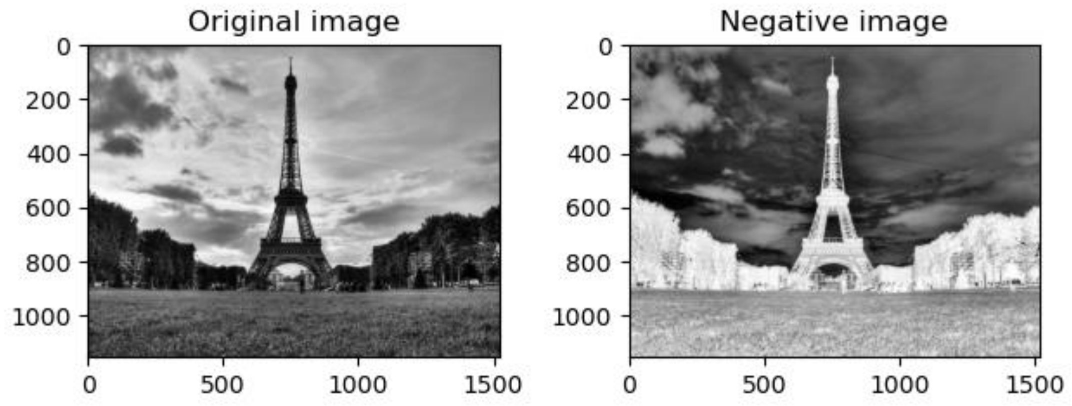


Fig. 1

2. Flip the image horizontally (i.e, map pixels from right to left changed from left to right)

The image is a 2-D matrix, to flip the image horizontally, it is needed to reverse each sub-list but keep the order of the sub-lists. To achieve this by matrix operations with *numpy*, calculate the transpose of the matrix, calculate its reverse and then calculate the transpose again. Or, more conveniently, use function *cv2.flip()* to flip the image directly.

The result is like Fig. 2 below, the image on the left is the original version while the flipped image is on the right.

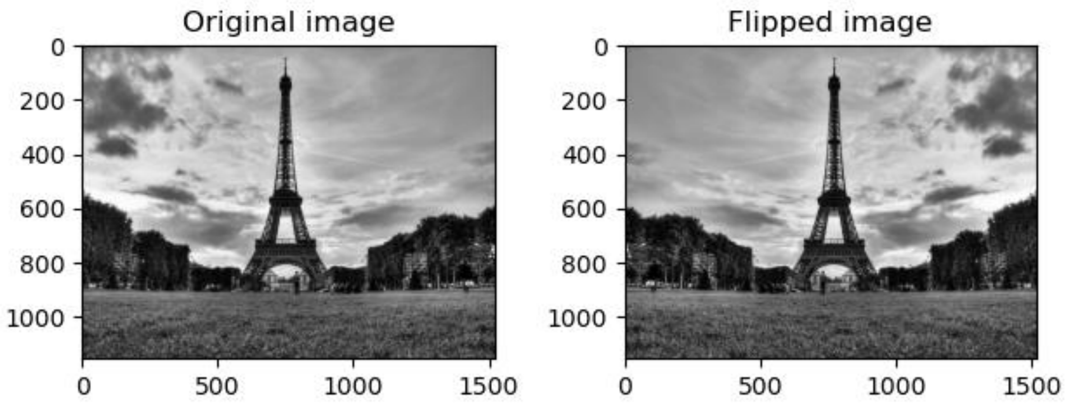


Fig. 2

3. Load a colour image, swap the red and green colour channels of the input.

Load the image “image1.jpg” with function *imageio.imread()*, the order of the channels is “RGB”. For each pixel, the 1st channel and the 2nd channel should be swapped.

The result is like Fig. 3 below, the original image is on the left and the swapped image is on the right. Since the red color channel and the green color channel are swapped, the greenish part in the original image is swapped to become reddish, and the reddish part becomes greenish.

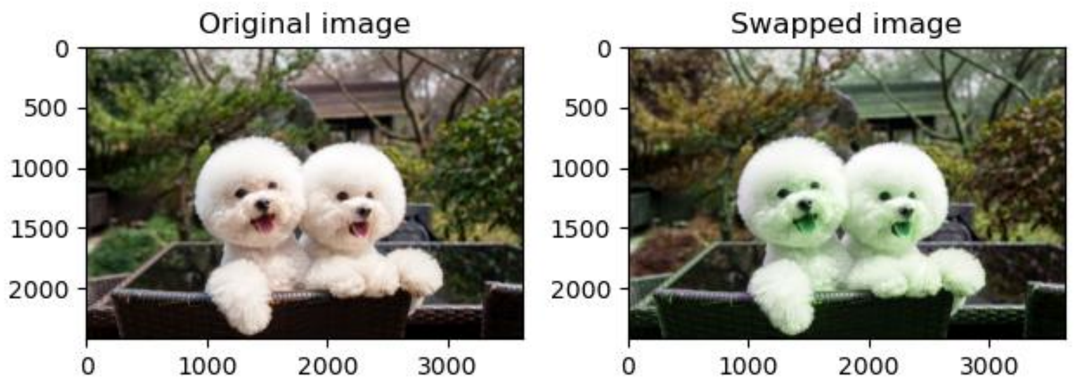


Fig. 3

4. Average the input image with its horizontally flipped image (use typecasting).

The original image and the horizontally flipped image have already there since the previous steps. To get the average of the two images, just add the two images together and then times 0.5 to get the average. When calculating, transfer the type of the images to float, and then after the calculation transfer the type of the resulting image to uint8.

The result is like Fig. 4 below which the original image on the left and the mixed image on the right.

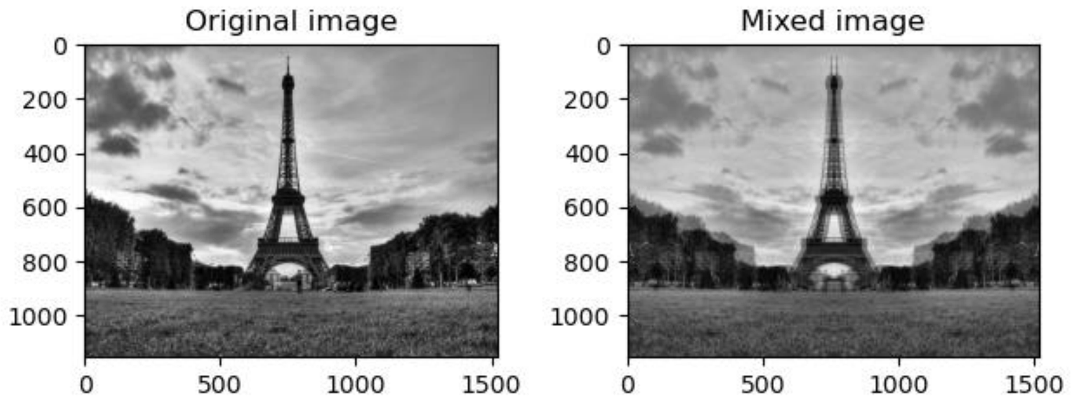
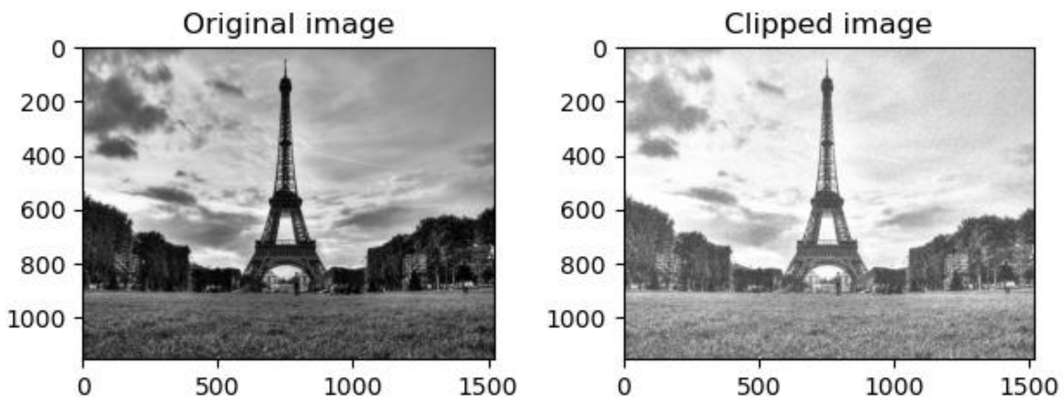


Fig. 4

5. Add a random value between [0, 127] to every pixel in the grayscale image, then clip the new image to have a minimum value of 0 and a maximum value of 255.

Use function *np.random.randint()* to generate the random matrix. Since the range is [0, 127], 0 and 127 are included, the parameter "low" should be set to 0 and "high" should be set to 128. The parameter "size" should be the same as the original image. Add the noise to the original image. Then set all the values greater than 255 to 255, set all the values less than 0 to 0.

The result is like Fig. 5 below with the original image on the left and the clipped image on the right. After adding the random values the image is more noisy. Since all the added values are greater than 0 so the image becomes brighter.



Original image
Clipped image
0
200
400
600
800
1000
0
500
1000
1500

Fig. 5

Task-3: Basic Image I/O

1. 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.

Function *cv2.resize()* can be used to resize the image. The outcome is like Fig. 6 below, the original image is on the left and the resized image is on the right. The coordinates show that the picture has been reduced.

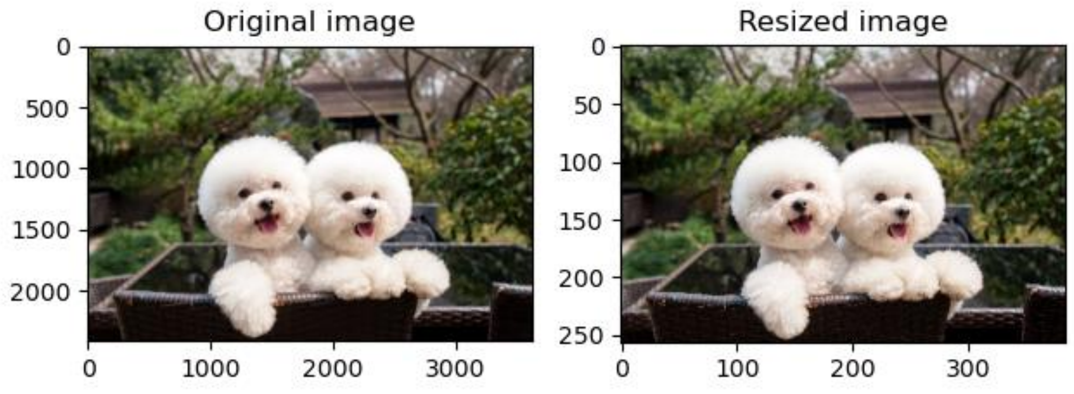


Fig. 6

b. Convert the colour image into three grayscale channels, i.e., R, G, B images, and display each of the three channel grayscale images separately.

The image is loaded by function *imageio.imread()*, the order of channels are “RGB”, so for each pixel, the first value is for red channel, the second for green channel and the third value for blue channel. Or, more conveniently, use function *cv2.split()* to split the image into 3 channels directly.

The outcome is like Fig. 7 below, the red channel, green channel and blue channel are displayed separately.

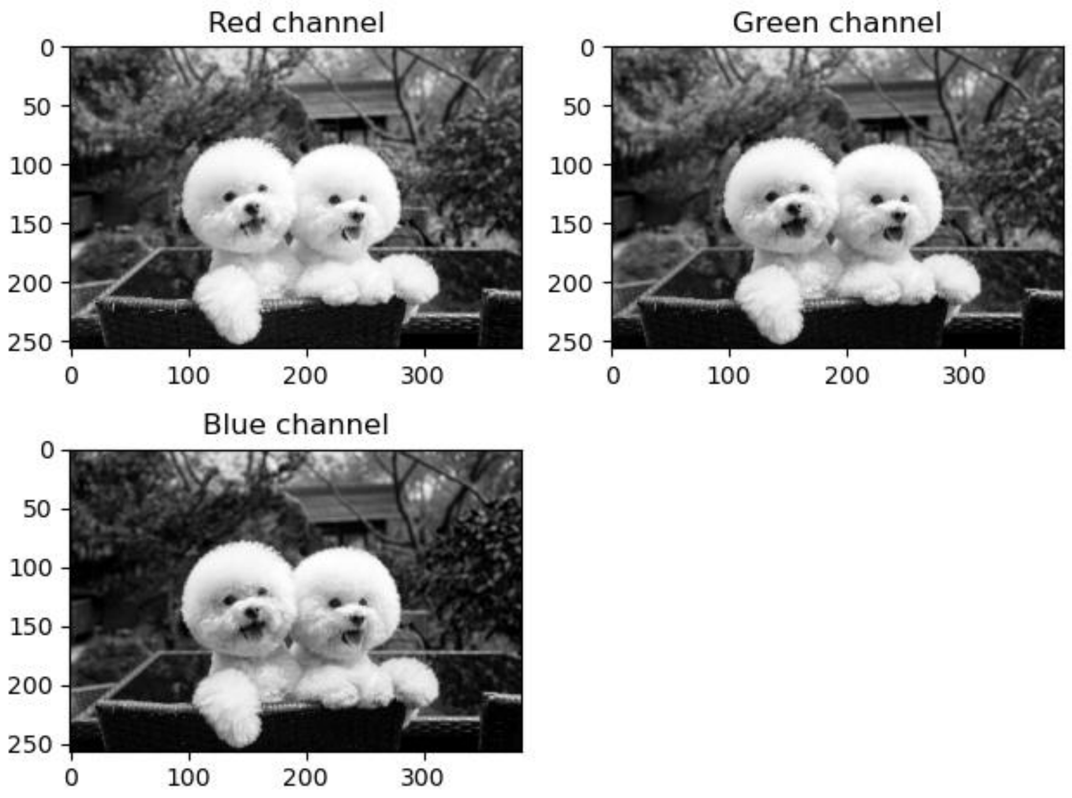


Fig. 7

c. Compute the histograms for each of the grayscale images, and display the 3 histograms.

To calculate the histogram of a channel, initialize a zero array to store the result. The size of the array is 256 because the values of the pixels are from 0 to 255. Calculate the total amount of the pixels of the image by multiplying the height and the width and note it as *n*. Traverse the image array, for every pixel value *p*, find the p^{th} value in the result array and add 1 to that value, then the result represents the total number of pixels with each gray value. Then divide the result array by *n* to get the percentage of the total number of pixels with each gray value, which is the histogram.

The outcome is like Fig. 8 below, the histograms of red channel, green channel and blue channel are displayed separately.

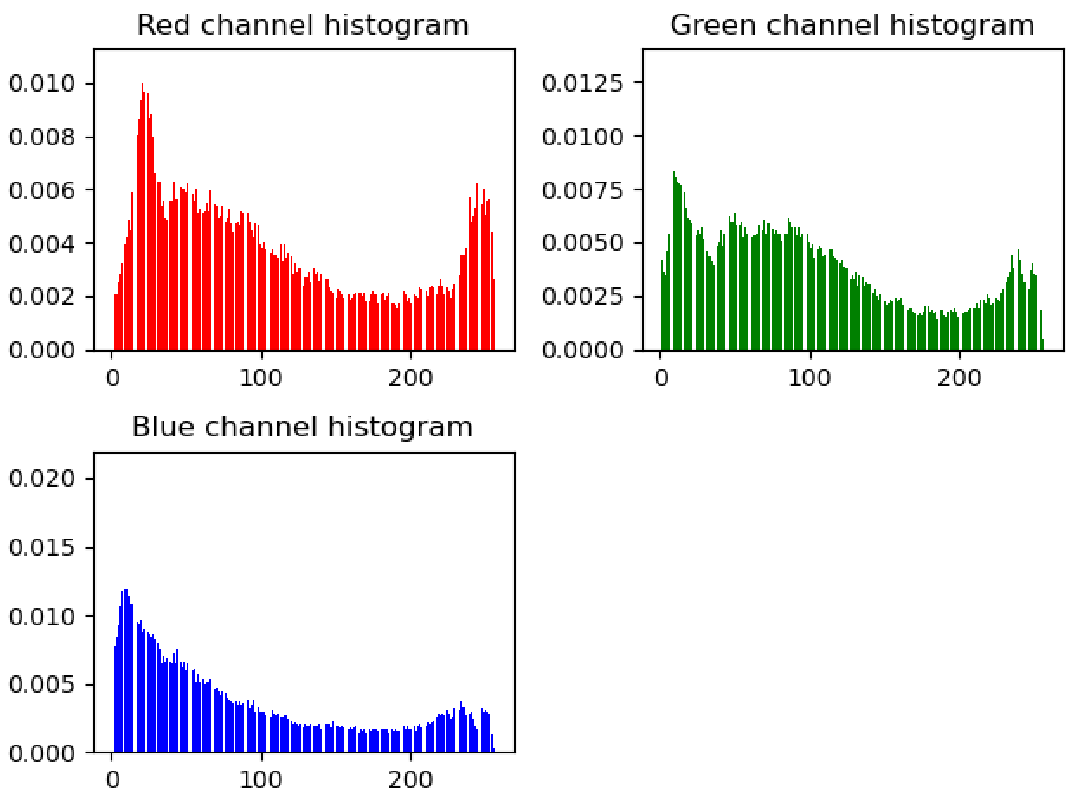


Fig. 8

d. Apply histogram equalisation to the resized image and its three grayscale channels, and then display the 4 histogram equalization images.

Use function *cv2.equalizeHist()* to equalize the histogram. This function can only process one channel, so need to calculate the red, green and blue channels separately and then merge them together to get the equalized color image.

The result is like Fig. 9 below, the first image on the top left is the merged color image, the three left are for each channel. After applying histogram equalization, the contrast of the image is increased.

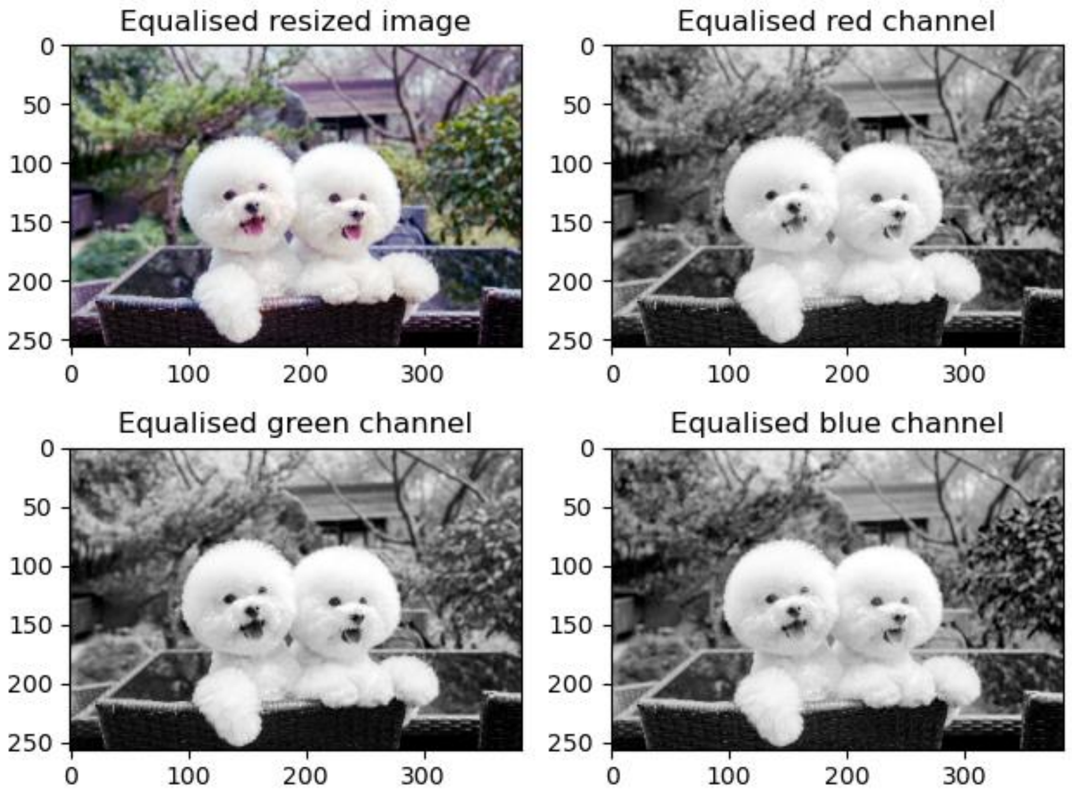

Equalised resized image
Equalised red channel
Equalised green channel
Equalised blue channel
0
50
100
150
200
250
0
100
200
300

Fig. 9

Task-4: Colour space conversion

1. Based on the formulation of RGB-to-YUV conversion, write your own function cvRGB2YUV() that converts the RGB image to YUV colour space. Read in Fig.2(a) and convert it with your function, and then display the Y, U, V channels in your report.

The function receives an RGB image and convert it to YUV color space. According to Szeliski the formulation of YUV color space is

$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

(2021, p.95). After getting each channel, merge them together to get the image in YUV color space.

Load the image "Figure2-a.png" using function *imageio.imread()* and apply the function. The result of the channels after applying the function is like Fig. 10 below.

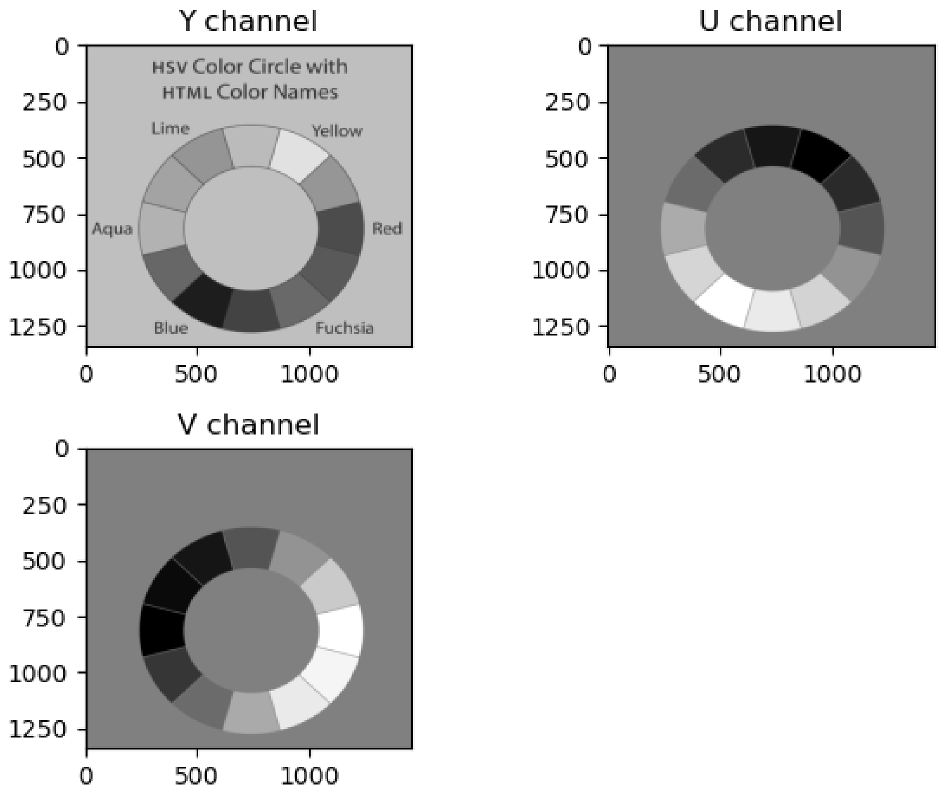


Fig. 10

2. Compute the average Y values of five colour regions in Fig. 2(b) you're your function and the Matlab's inbuilt function rgb2yuv(). Print both of them under the corresponding regions. You also need to explain how to distinguish and divide the five regions, and how to calculate the average Y value.

To distinguish the regions, my thought is firstly set the first pixel of the image as a "standard color" for comparing and store it into a list, then set a threshold. For each standard color in the list, initialize a zero matrix with the same size of the image for storing the pixels. Then loop the image. For each pixel of the image, compare the values in R, G, B channels with the standard color. If one of the differences of the three values is larger than the threshold, this value is in a new region. Store this pixel as a new standard color to the list, initialize a new zero matrix and save this pixel in the matrix. So, after the loop we can get several regions that the pixels for this region is stored and the other pixels are zeros. When looping, calculate the quantity of pixels for each region, and the column index of the first pixel in this region for later displaying. I also set a threshold for the quantity of the pixels in the regions. If the number of pixels in a region is less than this threshold, it means that this might be noises and will not be counted as a region.

To get the average Y value, for each region, apply the convert function to it, get the Y channel and calculate the sum of the Y channel, then divide it by the pixel quantity gotten from the loop, and this is the average. Since the calculation of Y value does not involve the addition of constants so the zeros pixels do not affect the result.

The result is like Fig. 11 below that Y1 is the average Y value calculated with my own function and Y2 is the average Y value calculated with the inbuilt function *cv2.cvtColor(src, cv2.COLOR_RGB2YUV)*. The results are very close.

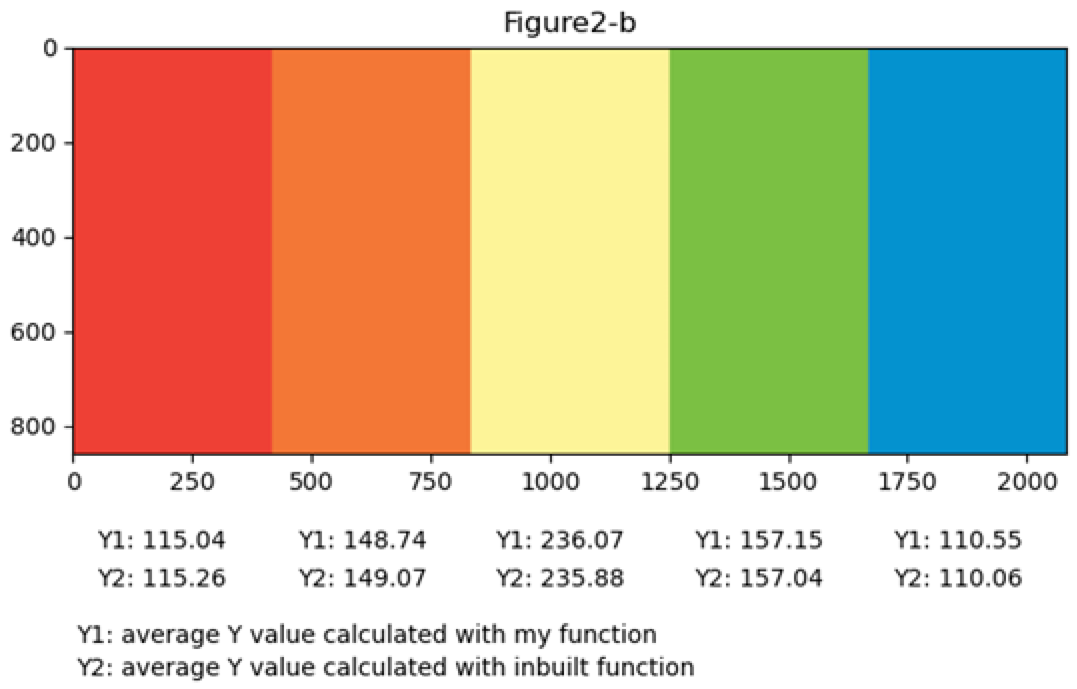


Fig. 11

Task-5: Image Denoising via a Gaussian and Bilateral Filter

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]. Add Gaussian noise to this new 512x512 image. Use Gaussian noise with zero mean, and standard deviation of 15. Display the two histograms side by side, one before adding the noise and one after adding the noise.

Load the image "image2.jpg" using function *imageio.imread()*. Compare the width and the height of the image and calculate the region to crop. Select the region from the original image, resize it to 512 * 512 and save it to a grayscale image.

The outcome is like Fig. 12 below, the original image displays on the left and the cropped square image on the right.

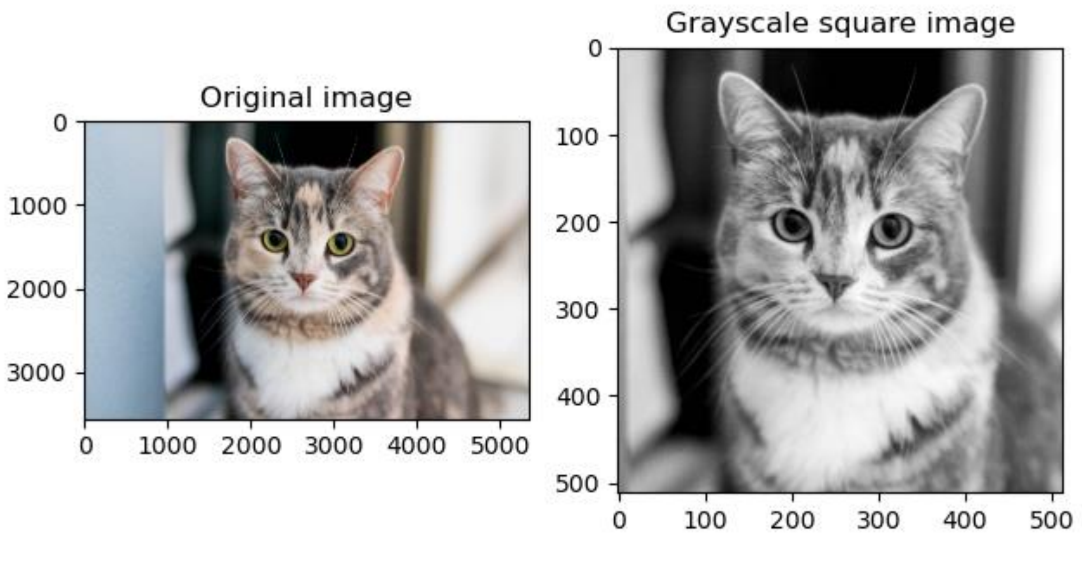


Fig. 12

Use function *np.random.normal()* to initialize an array of Gaussian noise with zero mean, standard deviation of 15 and the same shape with the grayscale resized image.

Add the noise array to the grayscale image to add noise to the image, then calculate the histograms of the original grayscale image and the noisy image and display.

The outcome is like Fig. 13 below, on the left is the histogram of the original grayscale image before adding noise, on the right is the histogram of the noisy image.

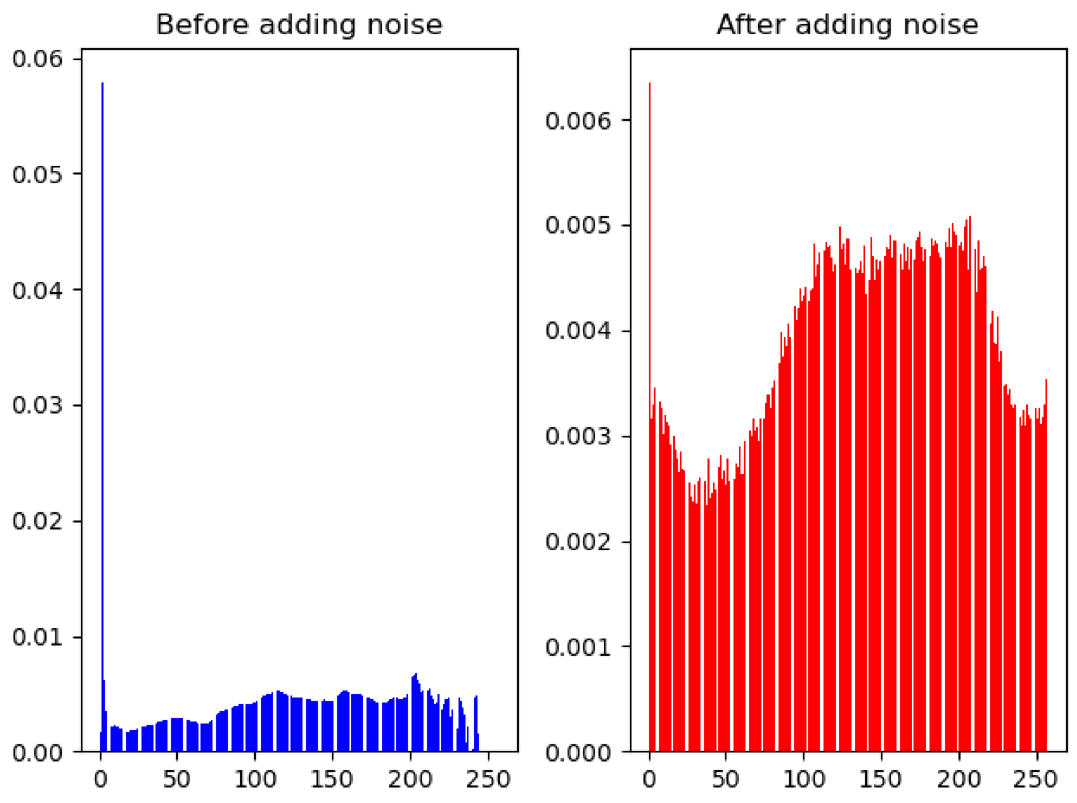


Fig. 13

2. Implement your own Python function that performs a 5x5 Gaussian filtering.

In my function I am using wrap padding to handle the borders and the output size is the same with the input. The function receives two parameters: the noisy image and the kernel, and return the output image. Each pixel in the output image is the product of the kernel and a matrix that is centered on the pixel with the same shape as the kernel.

3. Apply your Gaussian filter to the above noisy image, and display the smoothed images and visually check their noise-removal effects, investigating the effect of modifying the standard deviation of the Gaussian filter. You may need to test and compare different Gaussian kernels with different standard deviations.

I wrote a function to get the kernel. The function receives the kernel size and the deviation as parameters and return a Gaussian kernel. The calculation is to generate an array using function *cv2.getGaussianKernel()* and multiply it with its transpose, then return this matrix output.

I tried five kernels with the standard deviations 1, 1.5, 3, 5 and 8. The outcome is like Fig. 14 below. The first image on the top left is the original noisy image before applying Gaussian filter, and the left three are the filtered images with different standard deviation. When the standard deviation is small the image is sharp. As the standard deviation increase the image gets blurry.

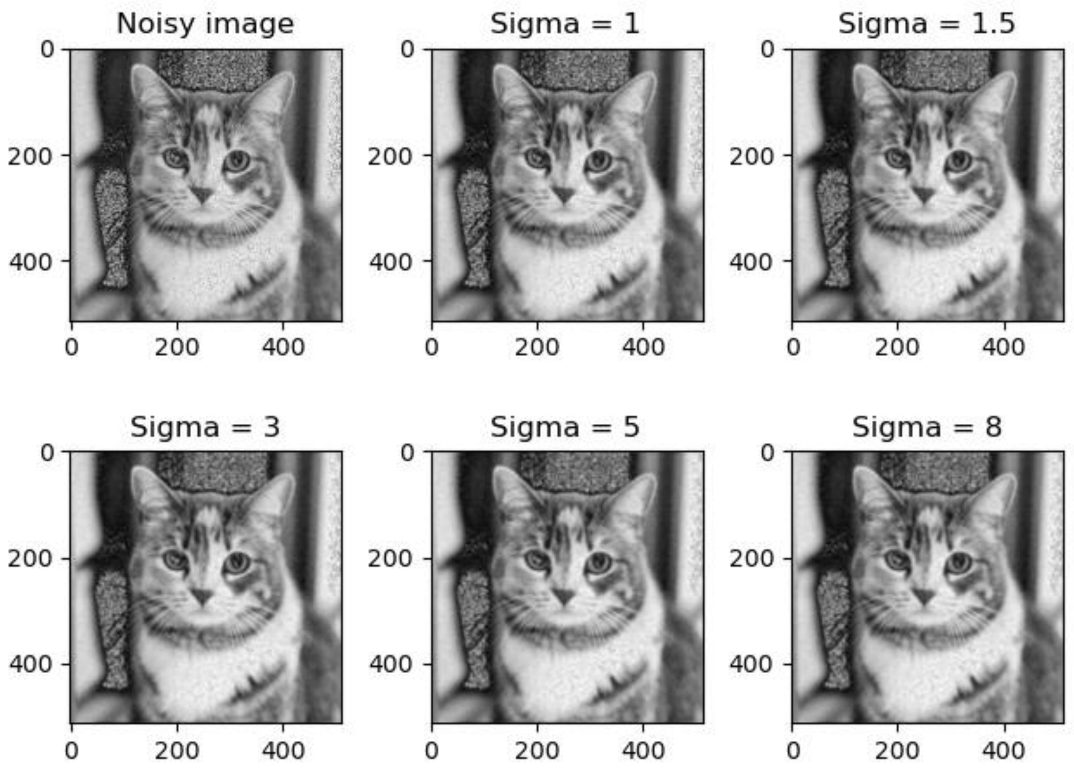


Fig. 14

4. Compare your result with that by Matlab’s inbuilt 5x5 Gaussian filter (e.g. cv2.GaussianBlur() in Python). Please show that the two results are nearly identical.

Apply the inbuilt function *cv2.GaussianBlur()* and my own function to the noisy image with a deviation as 50 and plot the image. The outcome is like Fig. 15 below, the original noisy image is on the left, in the middle is the image filtered by the inbuilt function and the image filtered by my own function is on the right. The two filtered images are similar.

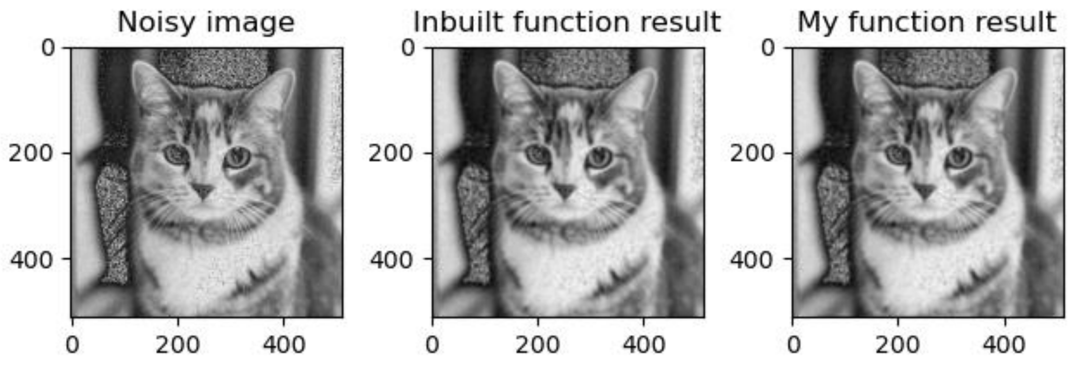


Fig. 15

Part 2:

1. Using your Gaussian filter as a base, implement your own Python function that performs a 5x5 Bilateral filtering to gray-scale image.

In Bilateral filter, in addition to the Gaussian kernel, there is another kernel calculated according to each pixel and the color sigma that represent how close the grayscale of the neighbor pixels to its grayscale. Multiply the Gaussian kernel with it and normalize the product, use this as the filter kernel to filter the image. Loop every pixel in the image, find out its neighbors according to the kernel size and multiply with the kernel to get the filtered value. The larger the color sigma is, the further pixels will be mixed, which means the less protection to the edges.

2. Apply your Bilateral filter to the above noisy image (greyscale version) from the last task, and display the smoothed images and visually check their noise removal and bilateral edge preserving effects.

The space deviation is set to 50 and tried with the color deviations as 0.1, 0.5, 1, 5 and 25. The outcome is like Fig. 16 below. The first image on the top left is the original noisy grayscale image, and the others are filtered images. As the color deviation becomes larger, the edges are more blurry, but the image becomes more smooth.

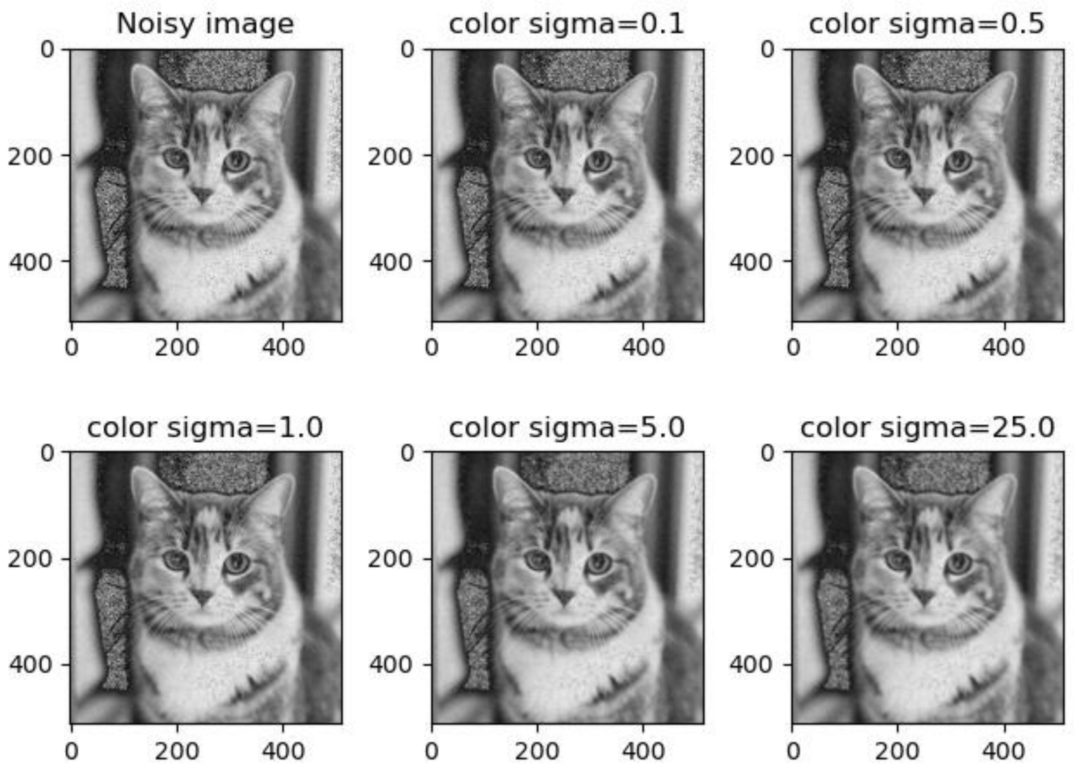


Fig. 16

3. Extend the Bilateral filter to colour images (eg. The color version of the previous grayscale image.). For this you may need to consider the CIE-Lab colour space as described in the paper. You will need to explore this for yourself. Namely, You need to generate the noisy color image and implement the bilateral filter to the color image (CIE-Lab).

The function receives a color image in RGB color space, the Gaussian kernel, and a color sigma, and return the filtered image. For better performance, convert the image to CIE-Lab color space. Apply Bilateral filter function to each channel and convert the image to RGB color space again and return.

I used the image “image2.jpg” and added Gaussian noise with 0 mean and deviation as 10. The kernel is a 5 * 5 Gaussian kernel with sigma as 50 and the color sigma is 1. Apply the Bilateral filter with the function and the outcome is like Fig. 17 below, on the left is the original noisy image, in the middle is the output of my Bilateral filter, and on the right is the output of the inbuilt function *cv2.bilateralFilter()*. The color noise above the cat’s head is different in the two filtered images.

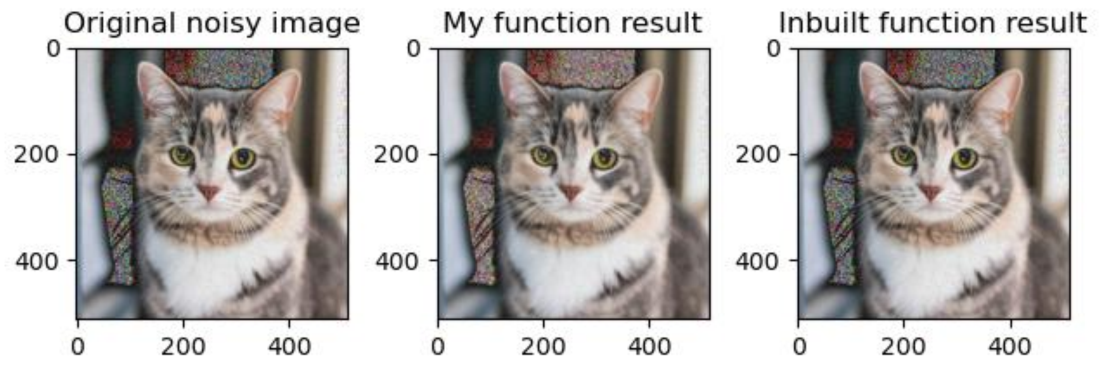


Fig. 17

4. In up to half a page, discuss the impact of smoothing on colour. What are the difficulties of smoothing in RGB colour space? Why is CIE-Lab space a good idea for this smoothing? (Compare images smoothed in CIE-Lab space vs RGB) Does the bilateral filter itself help? You may want to compare Gaussian smoothed colour images and bilateral filtered ones, and investigate filtering in different colour spaces. For this you can use your processed images as examples, possibly cropping and enhancing detail in your report to illustrate your discussion.

The difficulty is for color image, the colors between any two other colors are usually different. This color edge will be emphasized when applying filters in RGB color space and even becomes boarder because the grayscales in the three channels are usually different.

In ICE-Lab color space the channel a and b are already considered the similarity between the colors. When applying the Bilateral filter only perceptually similar colors are averaged together, and only perceptually important edges are preserved. The Bilateral filter itself also help because it can recognize the edge, especially when applying in ICE-Lab color space. Gaussian filter will just simply average the color and ignore the edge as well as the color edge between two other colors.

In Fig. 18 below the first image on the top left is the original noisy image, the second image on the top right is Bilateral filtered in CIE-Lab color space, the third image on the bottom left is Bilateral filtered in RGB color space, the last one on the bottom right is Gaussian filtered in RGB color space. In the second image, on the right side of the border there are less shadow than the other Filtered images and the border details are kept.

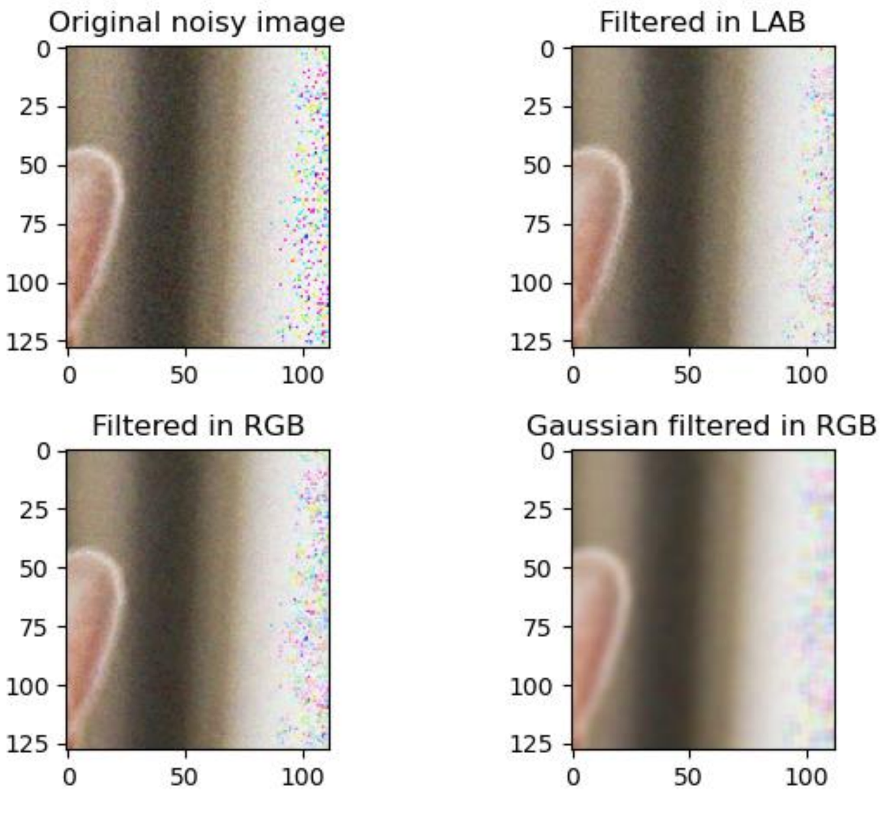


Fig.18

Task-6: Image Translation

1. Implement your own function my_translation() for image translation by any given number of pixels between [-100, +100], in both x and y. Note that this can be a real number (partial pixels). Display images translated by (2.0, 4.0), (4.0, -6.0), (2.5, 4.5), (-0.9, 1.7), (92.0, -91.0).

Since the pixels to transfer can be partial pixels, I used back mapping and bilinear interpolation. Initialize a zero array with the same size with the image as output. For every pixel in the output image, calculate the 4 pixels around it and the distances to these pixels and then calculate the value according to the distances and the values of its 4 neighbor pixels.

The original image I used is “image3.jpg”, after resizing the original image is like Fig. 19 below.

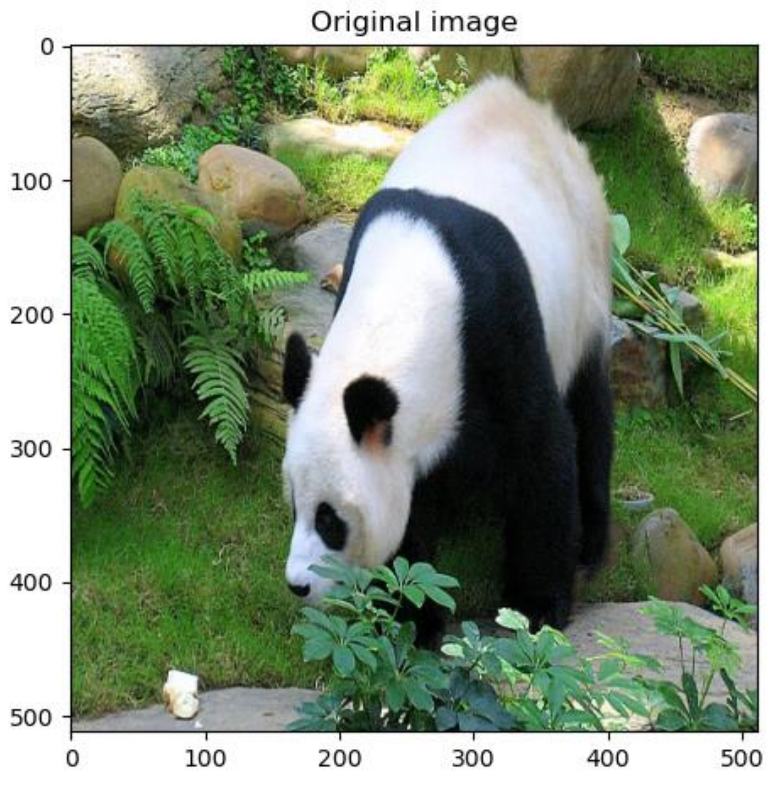


Fig. 19

The image translated by (2.0, 4.0) is like Fig. 20 below. It moved to right for 2 pixels and top for 4 pixels.

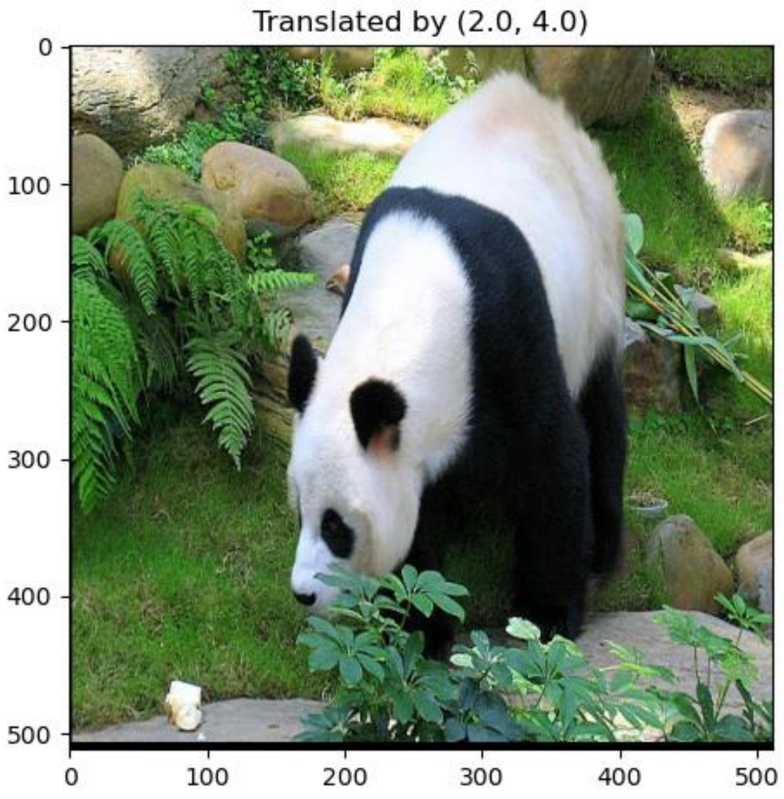


Fig. 20

The image translated by (-4.0, -6.0) is like Fig. 21 below. It moved to left for 4 pixels and bottom for 6 pixels.

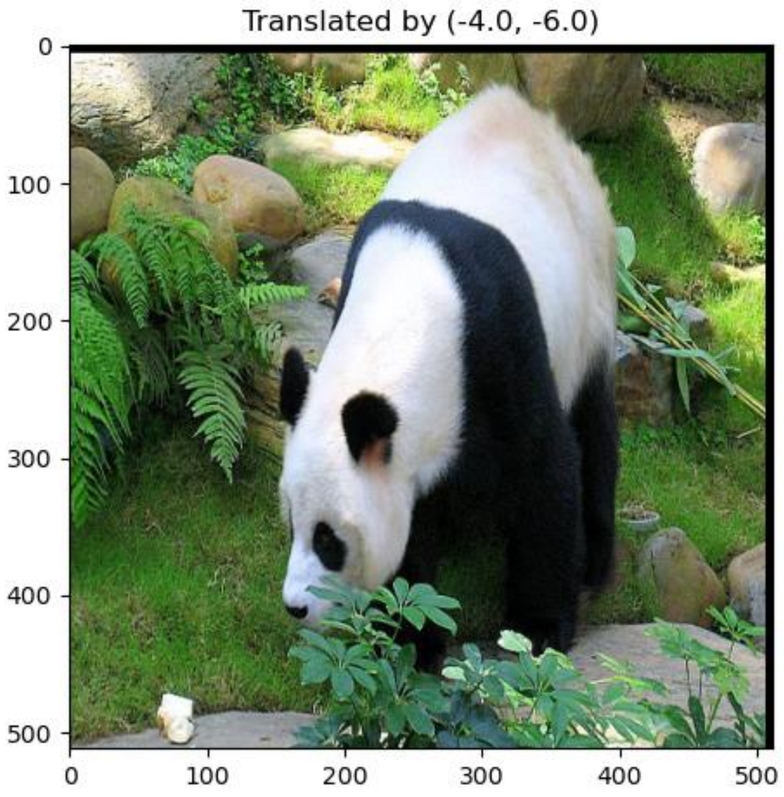


Fig. 21

The image translated by (2,5, 4.5) is like Fig. 22 below. It moved to right for 2.5 pixels and top for 4.5 pixels. The pixels for translation are partial pixels so the borders are blur.

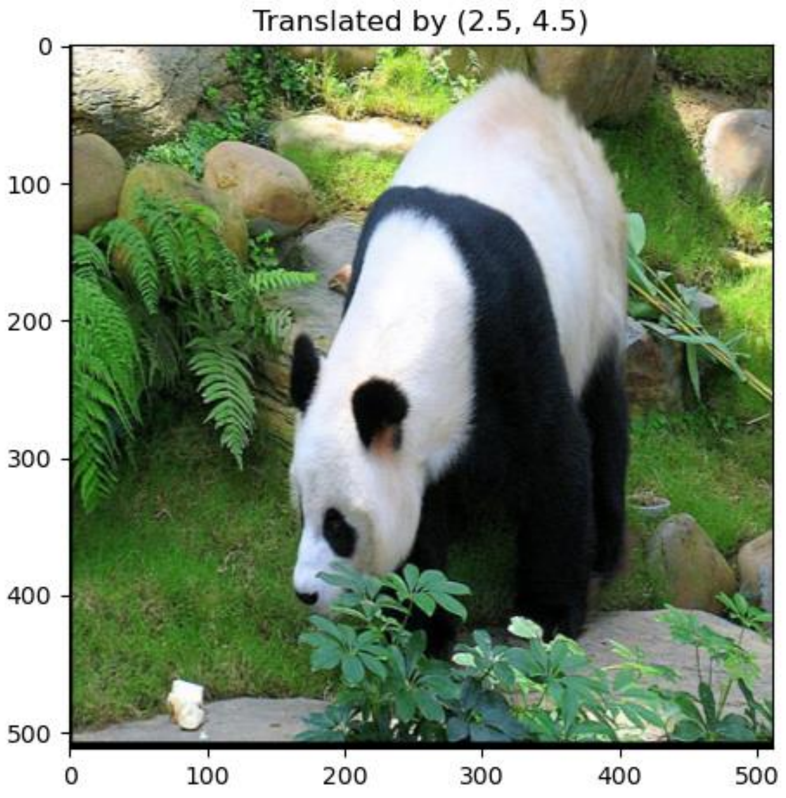


Fig. 22

The image translated by (-0.9, 1.7) is like Fig. 23 below. It moved to left for 0.9 pixels and top for 1.7 pixels. The pixels for translation are partial pixels so the borders are blur.

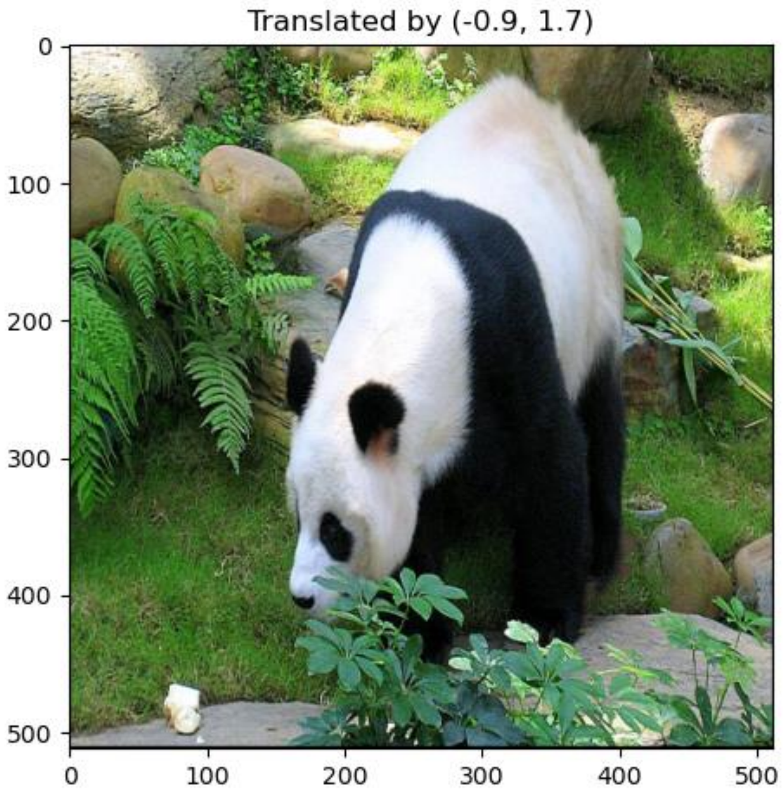


Fig. 23

The image translated by (92.0, -91.0) is like Fig. 24 below. It moved to right for 92 pixels and bottom for 91 pixels.

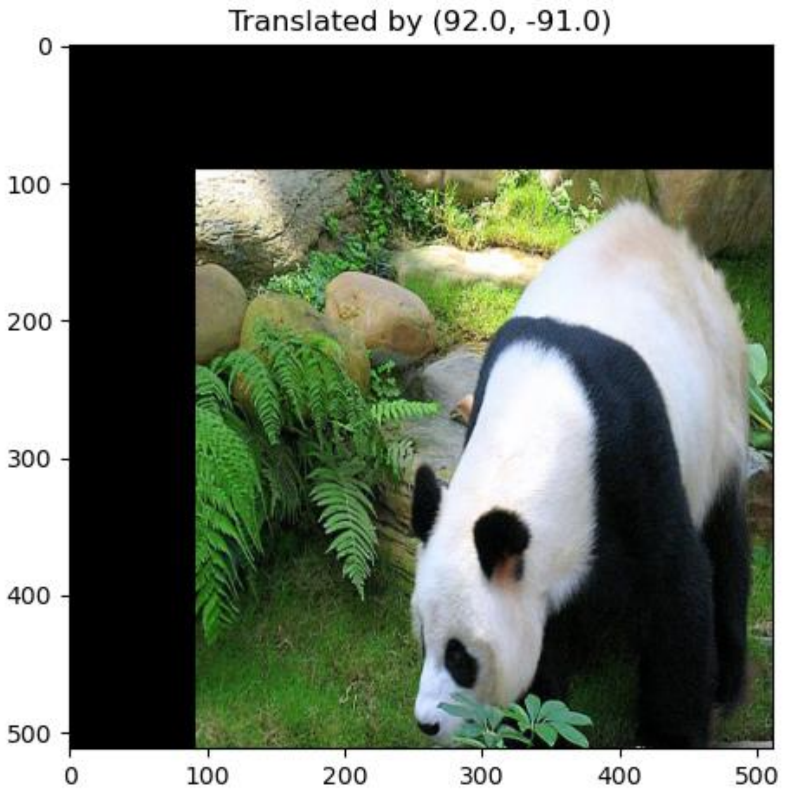


Fig. 24

2. Compare forward and backward mapping and analyze their difference.

Forward mapping iterates over source, calculates the mapped position of the pixels directly. But the calculation may result in partial pixels, so for the output, “splatting” is needed to distribute the pixel among its neighboring pixels complete the mapping of all pixels. The calculation is straightforward and can do non-linear transformations, but it may cause holes on the result when no color distributed to some pixels.

Backward mapping iterates over the output instead of the source and calculate the original pixels, thus there is no holes in the output image. The calculation may also result in partial pixels in the source image, so interpolation is needed to calculate the color according to the neighboring pixels from the original image. For backward mapping there are no holes in the output. The drawback is that it may oversample the source, and because it is needed to calculate each output pixel according to the original pixels so the wrap function must be invertible, which is not always possible.

3. Compare different interpolation methods and analyze their difference.

Nearest-neighbor interpolation is to make the partial output pixel value equal to the value of the nearest pixel in its neighborhood. The calculation is easy and efficient, but the output image may tend to be jagged and not such smooth.

Bilinear interpolation calculates the distance to its four neighboring pixels and calculate the value according to the distance. It considers all the 4 neighboring pixels in the calculation and the nearer pixel affect more on the result. Thus, it requires more calculation but the output is smooth and is less jagged.

Bicubic interpolation considers the neighboring 16 pixels using a third degree polynomial and its derivative. It does not only consider the four adjacent pixels influence but also the influence of the change rate of gray values among adjacent points. This algorithm can make the results more accurate and less jagged, but it is more computationally intensive.