Clab-1 Report

ENGN6528

Kai Hirota
u7233149

24/03/2021

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 the 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). [You only need to complete one set of questions either in matlab or Python.] (0.2 marks each)

Python (Delete this part if you use Matlab)

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

Create an n-dimensional array of shape (2, 3).

(2) b = a[: , 1]

Assign the first column of ndarray in a to b.

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

Create an array of shape (400, 1) consisting of random samples from standard normal distribution X ~ N(0, 1), then add 3 to the resulting array.

(4) g = f[f > 0]*3

Select values greater than 0, multiply by 3, and assign the resulting 1-dimensional array to g.

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

Create an array of zeros of shape (100,), and add 0.45. In other words, an array of a hundred 0.45s.

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

Create an array of 1s of shape (1, 100), and add 0.5.

(7) z = x + y

Element-wise addition between arrays x and y. The resulting array is assigned to z.

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

Create an array of length 250 using integers from the interval [1, 499], with equal spacing between the elements.

(9) b = a[: :-2]

Reverse array a, skip every other element (drop elements at index 1, 3, 5...), and assign to b.

(10) b[b >50]=0

Set elements in b greater than 50 to 0.

Task-2: Basic Coding Practice (1 marks)

Write functions to process an input grayscale image with following requirements, where you need to write a script to load the given image in the Lab package, apply each transformation to the input, and display the results in a figure. For Matlab, you can use subplot() function; for Python, we suggest to use matplotlib.pyplot.subplot(). Please notice that each subplot needs to be labelled with an appropriate title. (0.2 marks each)

1. Load a grayscale image, 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.

```
x = np.arange(256)
reverse_x = x[::-1]
negative_map = {x[i]: reverse_x[i] for i in range(256)}
```

```
original_shape = img.shape
img = img.reshape(original_shape[0] * original_shape[1])
negative_img = np.array(list(map(lambda x: negative_map[x], img)))

assert all([img[i] + negative_img[i] == 255 for i in range(len(img))])

img = img.reshape(original_shape)
negative_img = negative_img.reshape(original_shape)
```

```
compare_two_images(
    label_img={
        'Original': img,
        'Negative': negative_img
    }
)
```

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

```
flipped = copy.deepcopy(img)
flipped = np.array([np.flip(row) for row in img])
```

```
compare_two_images(
    label_img={
        'Original': img,
        'Horizontal Flip': flipped
    }
)
```

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

```
images = []
filenames = []

# read color images
for f in os.listdir('images'):
    if f.startswith('image') and f.endswith('.jpg'):
        images += plt.imread(f'images/{f}', format='jpeg'),
        filenames += f,
```

```
ImageData = np.ndarray

def swap_red_green(arr: ImageData) -> ImageData:
    new_arr = copy.deepcopy(arr)

    for i in range(len(arr)):
        for j in range(len(arr[0])):
            r, g, b = arr[i][j]
            new_arr[i][j] = [g, r, b]

    return new_arr
```

```
plt.figure(figsize=(20, 10))

# plot original
for i in range(len(images)):
    plt.subplot(2,3,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(images[i])
    plt.xlabel(filenames[i])

modified_images = [swap_red_green(img) for img in images]

# plot swapped
for i in range(len(modified_images)):
    plt.subplot(2,3,i+1+len(modified_images))
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(modified_images[i])
    plt.xlabel(f'{filenames[i]} - RG channels swapped')

plt.tight_layout()
```


image1.jpg


image2.jpg


image3.jpg


image1.jpg - RG channels swapped


image2.jpg - RG channels swapped


image3.jpg - RG channels swapped

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

```
avg_img = copy.deepcopy(img)
original_shape = avg_img.shape

avg_img = avg_img.flatten()
flipped = flipped.flatten()

avg_img = (avg_img + flipped) / 2
avg_img = avg_img.reshape(original_shape)
flipped = flipped.reshape(original_shape)

assert avg_img.shape == original_shape, f'{avg_img.shape}, {original_shape}'
```

```
images = [img, flipped, avg_img]
filenames = ['Original', 'Flipped', 'Average']

plt.figure(figsize=(15, 5))
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(images[i], cmap=plt.cm.gray)
    plt.xlabel(filenames[i])
```

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. (Note: the intensity values of the original grayscale image range from 0 to 255.)

```
offset = np.random.randint(0, 128, original_shape)
clip = img + offset
clip[clip > 255] = 255

compare_two_images(
    label_img={
        'Original': img,
        'Clipped': clip
    }
)
```

Task-3: Basic Image I/O (2 marks)

Note: You need to download the image1.jpg, image2.jpg, image3.jpg from wattle.

In this task, you are asked to:

1. Using image1.jpg, develop short computer code that does the following tasks:

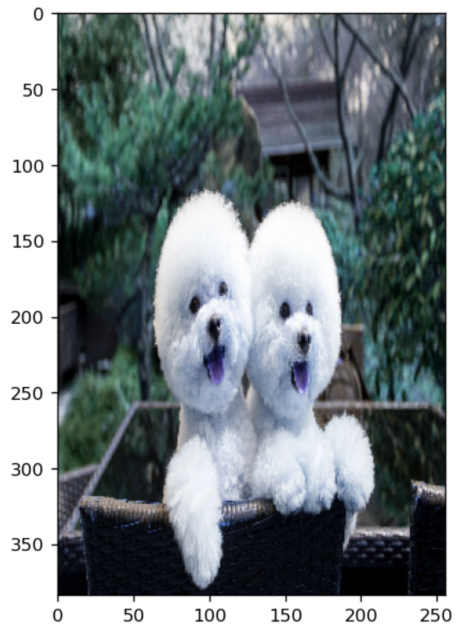
a. Read this image from its JPG file, and resize the image to 384 x 256 in columns x rows (0.2 marks).

```
img = cv2.imread(f'images/image1.jpg')
```

a. Resize

```
resized = cv2.resize(img, (256, 384))
plt.figure(figsize=(10, 6))
plt.imshow(resized)
```

```
<matplotlib.image.AxesImage at 0x7ffb8c870a10>
```

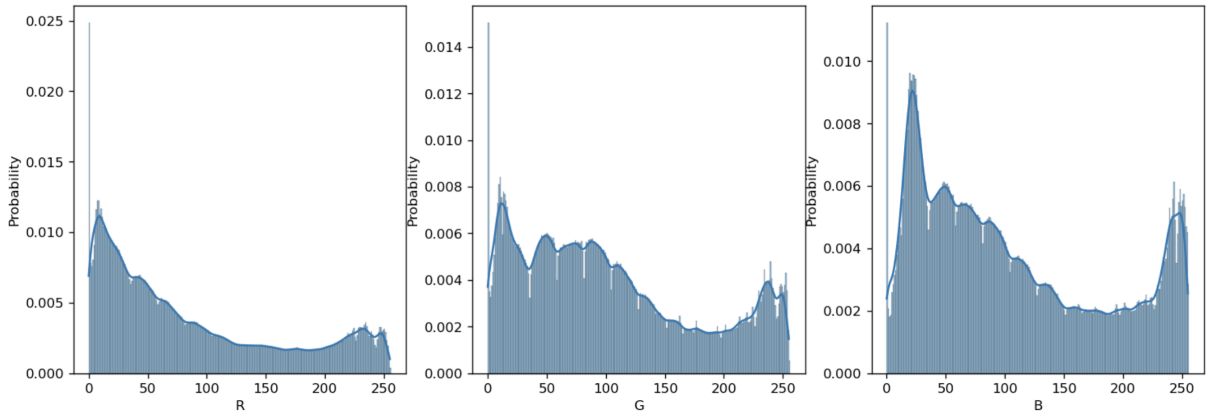


b. Convert the colour image into three grayscale channels, i.e., R,G,B images, and display each of the three channel grayscale images separately (0.2 marks for each channel, 0.6 marks in total).

```
split_images = []

filenames = ['R', 'G', 'B']
plt.figure(figsize=(15, 5))
for k in range(3):
    plt.subplot(1, 3, k+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

    plt.imshow(img[:,:,k], cmap=plt.cm.gray)
    plt.xlabel(filenames[k])
    split_images += img[:,:,k],
```

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).

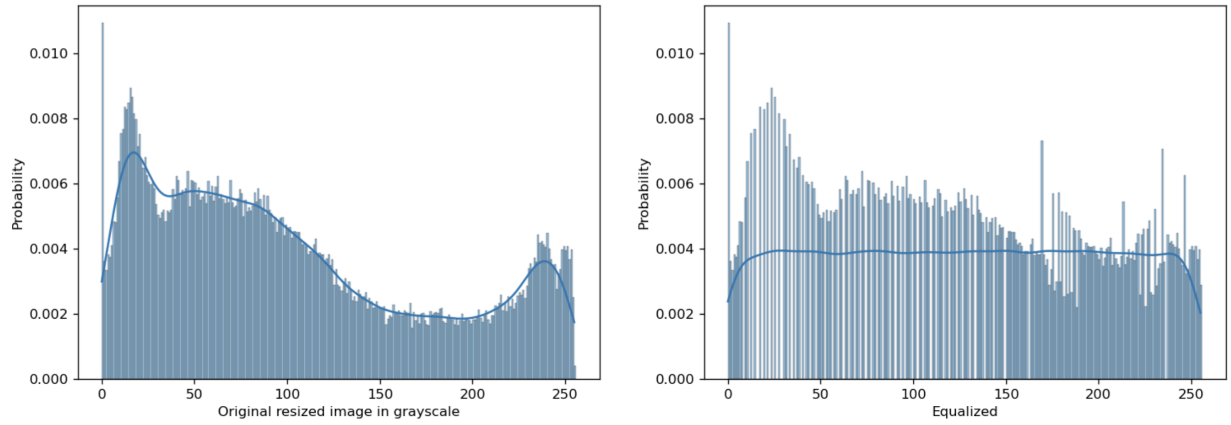
```
plt.figure(figsize=(15, 5))
i = 0
for title, im in zip(filenames, split_images):
    i += 1
    plt.subplot(1, 3, i)
    sns.histplot(
        im.flatten(),
        binrange=(0, 256),
        binwidth=1,
        stat='probability',
        kde=True
    )
    plt.xlabel(title)
```



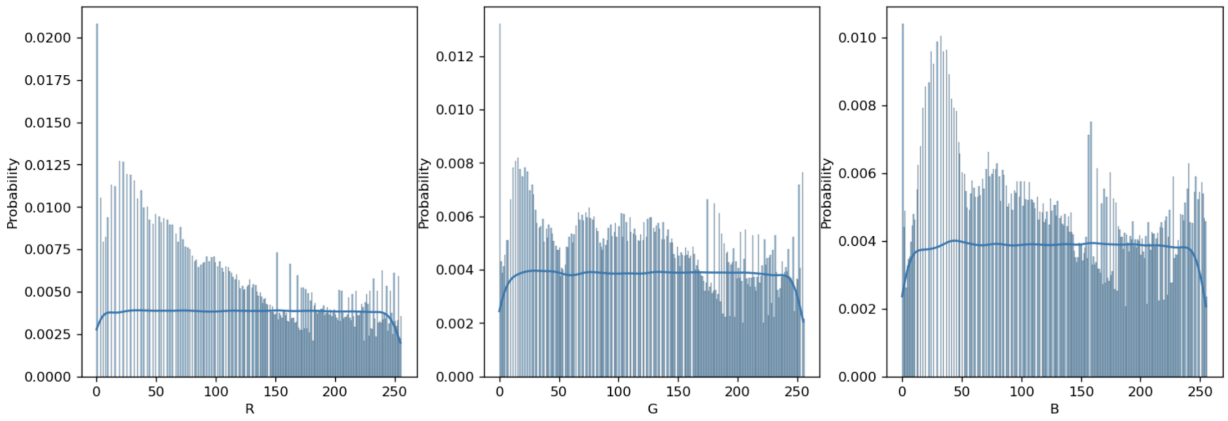
d. Apply histogram equalisation to the resized image and its three grayscale channels, and then display the 4 histogram equalization image (0.15 marks for each histogram, 0.6 marks in total). (Hint: you can use inbuilt functions for implementing histogram equalisation. e.g. histeq() in Matlab or cv2.equalizeHist() in Python).

```
plt.figure(figsize=(15, 5))

original_shape = resized.shape
grayimg = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)

plt.subplot(1, 2, 1)
sns.histplot(
    grayimg.flatten(),
    binrange=(0, 256),
    binwidth=1,
    stat='probability',
    kde=True
)
plt.xlabel('Original resized image in grayscale')

grayimg_equalized = cv2.equalizeHist(grayimg.flatten())
grayimg_equalized = grayimg_equalized.reshape(original_shape[:-1])

plt.subplot(1, 2, 2)
sns.histplot(
    grayimg_equalized.flatten(),
    binrange=(0, 256),
    binwidth=1,
    stat='probability',
    kde=True
)
plt.xlabel('Equalized')
```

```
Text(0.5, 0, 'Equalized')
```

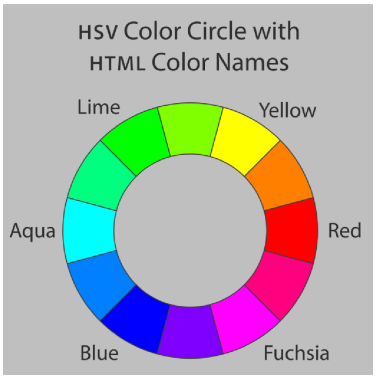


```
plt.figure(figsize=(15, 5))
filenames = ['R', 'G', 'B']

for i, title in enumerate(filenames):

    # extract color channel
    extracted = resized[:,:,i]

    equalized = cv2.equalizeHist(extracted)

    plt.subplot(1, 3, i+1)
    sns.histplot(
        equalized.flatten(),
        binrange=(0, 256),
        binwidth=1,
        stat='probability',
        kde=True
    )
    plt.xlabel(title)
```

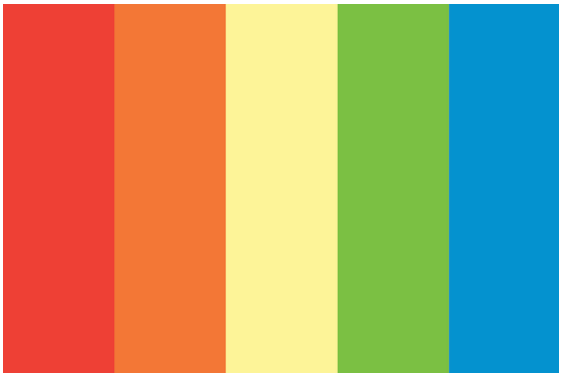


Task-4: Colour space conversion (3 marks)

Use the two images in Fig.2 to study colour space conversion from RGB to YUV (you can download them from the Wattle site):



Fig, 2 (a) Colour Wheel



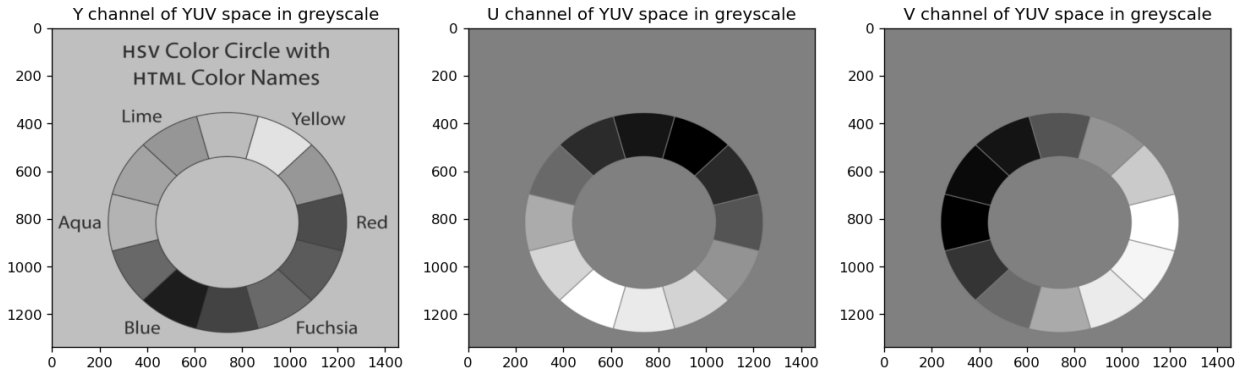
(b) Palettes

1. Based on the formulation of RGB-to-YUV conversion, write your own function cvRGB2YUV()that converts the RGB image to YUV colour space (0.7 marks). Read in Fig.2(a) and convert it with your function, and then display the Y, U, V channels in your report (0.1 marks for each channel, 0.3 marks in total).

```
def cvRGB2YUV(img: ImageData) -> ImageData:
    new_img = copy.deepcopy(img)

    for i in range(len(img)):
        for j in range(len(img[0])):
            rgb = img[i][j]
            r, g, b = rgb
            y = rgb @ [0.299, 0.587, 0.114]
            u = b - y
            v = r - y #C_r
            new_img[i][j] = np.array([y, u, v])
    return new_img
```

```
img = cv2.cvtColor(img, cv2.COLOR_RGBA2RGB)
img = img / 255
new_img = cvRGB2YUV(img)
```

```
plt.figure(figsize=(15, 5))
channels = ['Y', 'U', 'V']

for i, title in enumerate(channels):
    plt.subplot(1, 3, i+1)
    plt.imshow(new_img[:,:,i], cmap=plt.cm.gray)
    plt.title(f'{channels[i]} channel of YUV space in greyscale')
```



2. Compute the average Y values of five colour regions in Fig. 2(b) with your function and the Matlab's inbuilt function rgb2yuv(). Print both of them under the corresponding regions (0.1 marks for each value, 0.5 marks in total). You also need to explain how to distinguish and divide the five regions, and how to calculate the average Y value (1.5 marks, higher marks only for a smarter solution). (Note: The elements of both colormaps are in the range 0 to 1.).

Method used for distinguishing the image into five regions:

1. Use KMeans classifier with k = 5 to assign every pixel in the image into one of five classes by trying to minimize within-cluster sum of squared errors with respect to the centroid.
2. Map the predicted class labels to some value between [0, 1] in order to plot the classification result.

```
original_shape = img.shape
img = img.reshape((img.shape[0] * img.shape[1], img.shape[2]))
```

```
# while we know k should be 5 from looking at the image, we can also use empirical methods to determine the optimum k
kmeans = KMeans(n_clusters=5)
pred = kmeans.fit_predict(img)
pred = pred.reshape(original_shape[:-1])
```

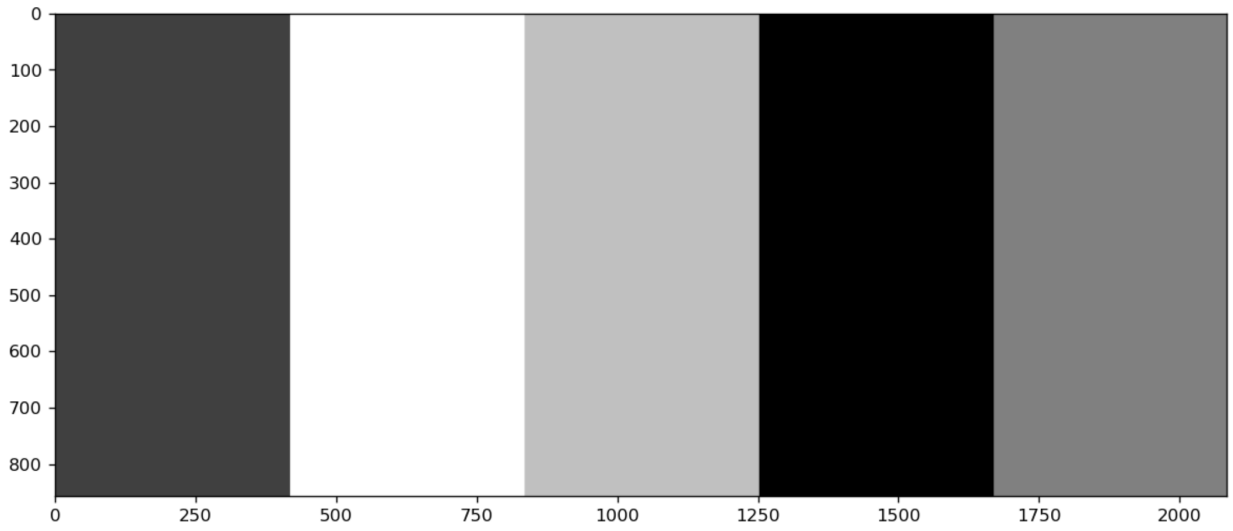
```
# map predicted class label to some greyscale color to visualize the partition

color_mapping = {idx: val for idx, val in enumerate(np.linspace(0, 1, 5))}
tmp = np.zeros(pred.shape, dtype=np.float32)

for i in range(len(pred)):
    for j in range(len(pred[0])):
        tmp[i][j] = color_mapping[pred[i][j]]
```

```
print('class: greyscale intensity')
for i in range(5):
    print(f"{i:^5}: {color_mapping[i]:^3.2f}")
```

```
class: greyscale intensity
  0  : 0.00
  1  : 0.25
  2  : 0.50
  3  : 0.75
  4  : 1.00
```

```
plt.imshow(tmp, cmap=plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x7ffb6b87e750>
```



3. The above image tells us that the partitioning worked very nicely, with seemingly clear and defined borders between the color regions.
4. Convert to YUV space.
5. Iterate over the image and add up Y value of each class / region, and count the occurrences of each class as well.
6. Finally, calculate the average Y value for each region and plot the result.

```
img = img.reshape(original_shape)
# img = img / 255
new_img = cvRGB2YUV(img)
```

```
cumsum = defaultdict(float)

for i in range(len(img)):
    for j in range(len(img[0])):
        group_idx = pred[i][j]
        cumsum[group_idx] += new_img[i][j][0]
```

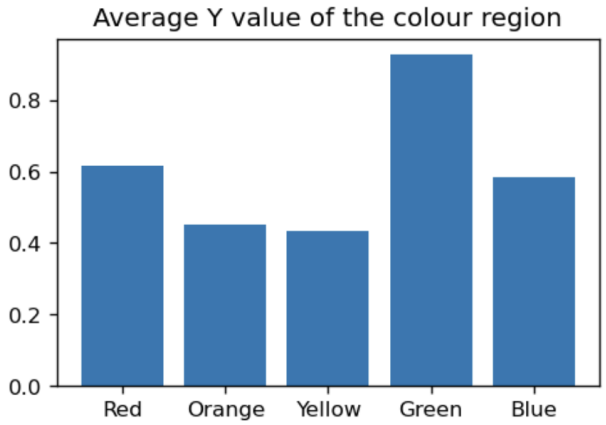
```
classes_count = Counter(pred.flatten())
classes_count

scores = [cumsum[i] / classes_count[i] for i in range(5)]
```

```
scores
```

```
[0.614641043079435,
 0.4501589623203606,
 0.43352913030065193,
 0.9257567050957872,
 0.5830193593725173]
```

```
plt.figure(figsize=(10, 3))

plt.subplot(1, 2, 1)
plt.xticks([])
plt.yticks([])
plt.grid(False)
plt.imshow(img)

plt.subplot(1, 2, 2)
plt.bar(['Red', 'Orange', 'Yellow', 'Green', 'Blue'], scores)
plt.title('Average Y value of the colour region')

print('Average Y value:')
for col, score in zip(['Red', 'Orange', 'Yellow', 'Green', 'Blue'], scores):
    print(f"- {col}: {score:3.2f}")
```

```
Average Y value:
- Red: 0.61
- Orange: 0.45
- Yellow: 0.43
- Green: 0.93
- Blue: 0.58
```



Task-5: Image Denoising via a Gaussian and Bilateral Filter (9 marks)

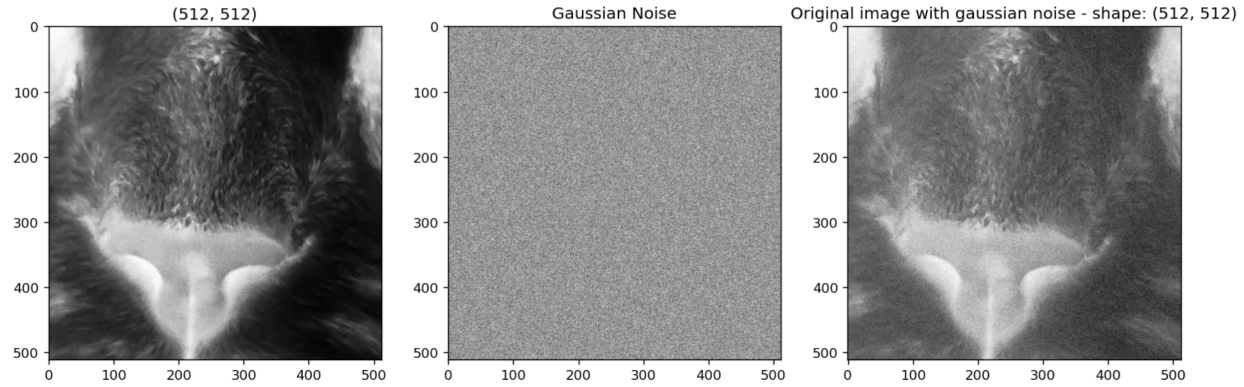
1. Read in image2.jpg. Crop a square image region corresponding to the central part of the image, resize it to 512×512, and save this square region to a new grayscale image. Please display the two images. Make sure the pixel value range of this new image is within [0, 255]. Add Gaussian noise to this new 512x512 image (Review how you generate random number in Task-1). Use Gaussian noise with zero mean, and standard deviation of 15.

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.

Display the two histograms side by side, one before adding the noise and one after adding the noise (0.5 marks).

```
plt.figure(figsize=(15, 5))

target_res = (512, 512)
n_rows, n_cols = img.shape
x = (n_rows // 2) - (target_res[0] // 2)
y = (n_cols // 2) - (target_res[1] // 2)
cropped = img[x:x+target_res[0],y:y+target_res[1]]
plt.subplot(1, 3, 1)
plt.imshow(cropped, cmap=plt.cm.Greys)
plt.title(cropped.shape)

noise = np.random.normal(0, 15, size=target_res)
plt.subplot(1, 3, 2)
plt.imshow(noise, cmap=plt.cm.Greys)
plt.title('Gaussian Noise')

cropped_n = cropped + noise
plt.subplot(1, 3, 3)
plt.title('Original image with gaussian noise - shape: ' + str(cropped_n.shape))
plt.imshow(cropped_n, cmap=plt.cm.Greys)
```

```
<matplotlib.image.AxesImage at 0x7ffb6caebb90>
```



np.clip(cropped_n, 0, 255) is used (not in the screenshot) to ensure that the range is valid.

2. 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

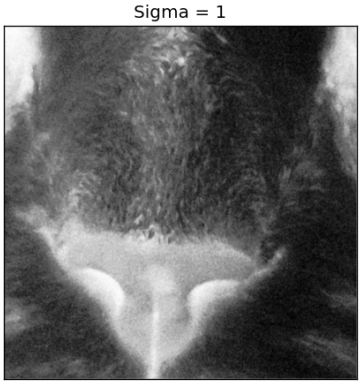
```
sigma = 15
kernel_size = (11, 11)
```

```
def my_Gauss_filter(img, kernel):
    assert kernel.shape[0] == kernel.shape[1]

    padding = kernel.shape[0] // 2

    padded_img = np.pad(img, padding)
    new_img = np.zeros(padded_img.shape)

    for i in range(padding, len(padded_img) - padding):
        for j in range(padding, len(padded_img[0]) - padding):
            new_img[i][j] = np.sum(np.multiply(
                padded_img[i-padding:i+padding+1, j-padding:j+padding+1],
                kernel
            ))
    return new_img[padding:padding+img.shape[0], padding:padding+img.shape[1]]

def create_filter(sigma, kernel_size):
    arr = np.zeros(kernel_size)
    center = (kernel_size[0] // 2, kernel_size[1] // 2)

    for u in range(kernel_size[0]):
        for v in range(kernel_size[1]):
            x, y = center[0] - u, center[1] - v
            exp = -(x ** 2 + y ** 2) / (sigma ** 2)
            arr[u][v] = 0.5 * np.pi * (sigma ** 2) * (np.e ** exp)

    arr /= arr.sum()
    return arr

kernel = create_filter(sigma, kernel_size)
blurred_img = my_Gauss_filter(cropped_n, kernel)
assert cropped_n.shape == blurred_img.shape
```

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 (0.5 marks).

Note: In doing this task, and the bilateral filter below 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.

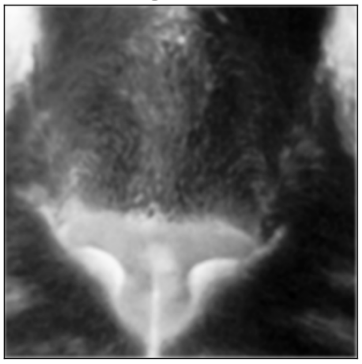
Sigma = 1



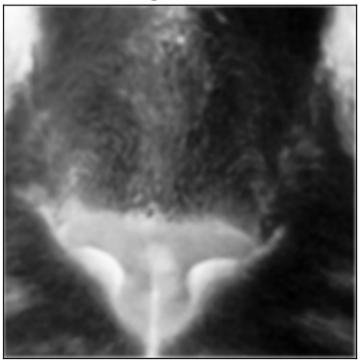
Sigma = 2

Sigma = 3

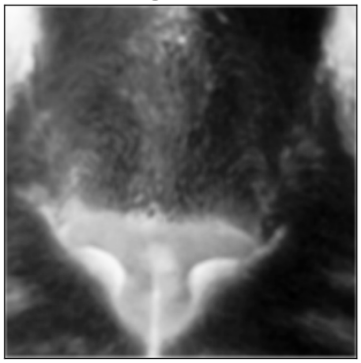
Sigma = 4



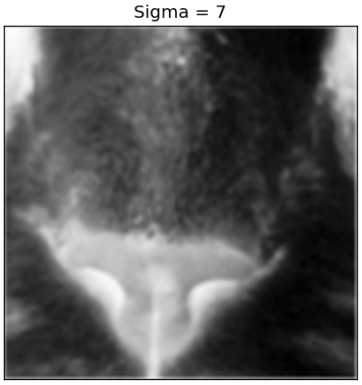
Sigma = 5



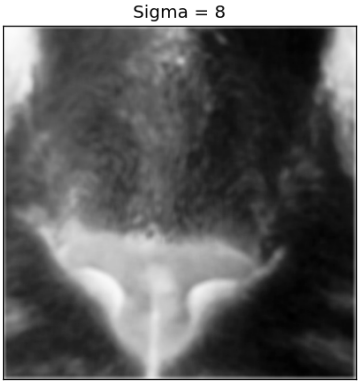
Sigma = 6



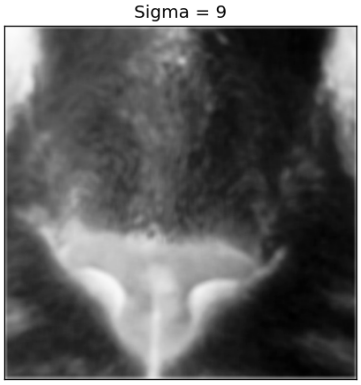
Sigma = 7



Sigma = 8



Sigma = 9



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

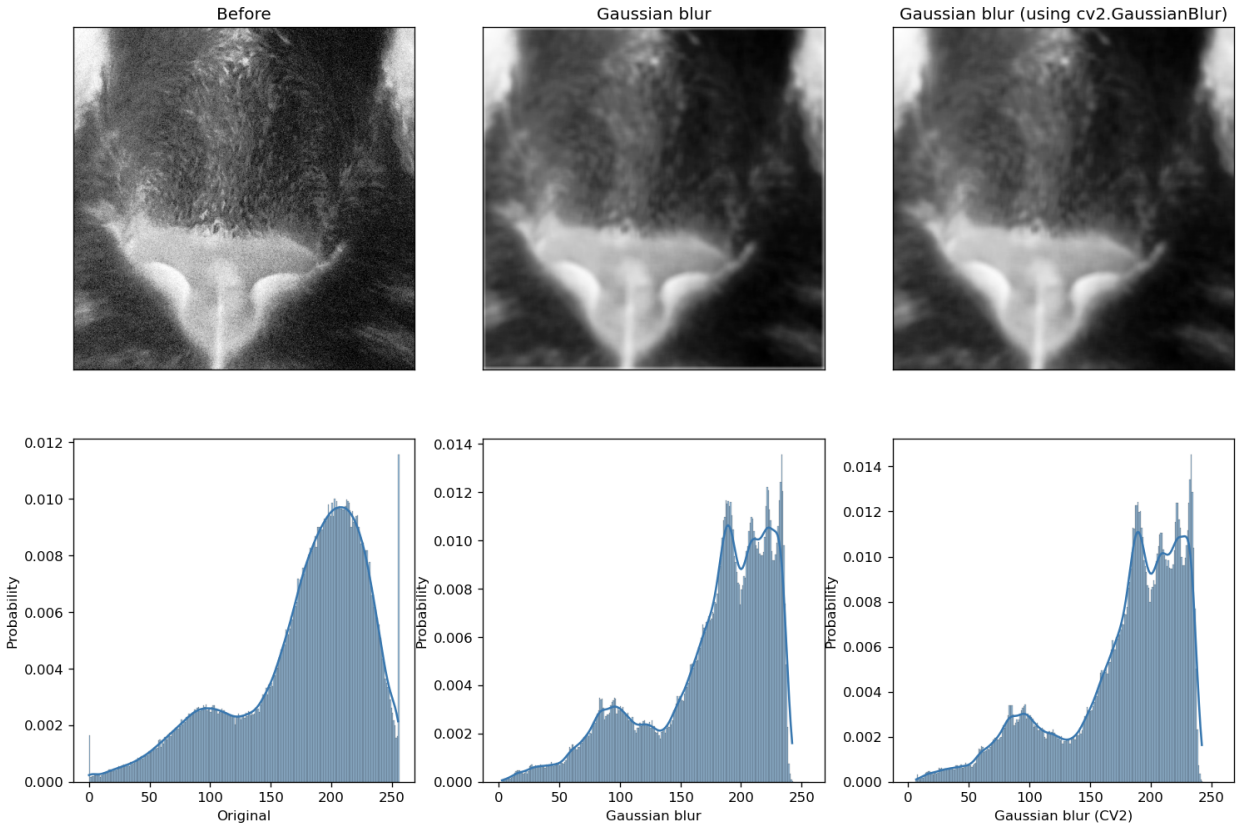


Image Denoising via a Bilateral (Challenge task: this task will be difficult for most students to complete.) (5 marks)

1. Using your Gaussian filter as a base, implement your own Matlab/Python function that performs a 5x5 Bilateral filtering to gray-scale image. (1.5 marks)

Your function interface must be:

my_Bilateral_filter()

input: noisy_image, my_5x5_gausskernel, colour_sigma

output: output_image

where colour_sigma is the sigma applied to the intensity/gray scale part of the filter.

```
def my_Bilateral_filter(noisy_image, my_5×5_gausskernel, colour_sigma):
    return bilateral_filter_grey(noisy_image, my_5×5_gausskernel, colour_sigma)

def bilateral_filter_grey(img, kernel_s, sigma_r):
    assert img.max() ⩽ 1

    padding = kernel_s.shape[0] // 2
    padded_img = np.pad(img, padding)
    new_img = np.zeros(padded_img.shape)

    for i in range(padding, len(padded_img) - padding):
        for j in range(padding, len(padded_img[0]) - padding):
            # create intensity kernel_r
            intensity_diff = np.abs(padded_img[i-padding:i+padding+1, j-padding:j+padding+1] - padded_img[i][j])
            kernel_r = stats.norm.pdf(intensity_diff, loc=0, scale=sigma_r)
#             kernel_r /= kernel_r.sum()

            # multiply the filters with corresponding region in image, then divide by sum of product of kernels to normalize
            new_img[i][j] = np.sum(
                padded_img[i-padding:i+padding+1, j-padding:j+padding+1] * kernel_s * kernel_r
            ) / np.sum(kernel_s * kernel_r)

    return new_img[padding:padding+img.shape[0], padding:padding+img.shape[1]]

def create_filter(sigma, kernel_size):
    arr = np.zeros(kernel_size)
    center = (kernel_size[0] // 2, kernel_size[1] // 2)

    for u in range(kernel_size[0]):
        for v in range(kernel_size[1]):
            x, y = center[0] - u, center[1] - v
            exp = -(x ** 2 + y ** 2) / (sigma ** 2)
            arr[u][v] = 0.5 * np.pi * (sigma ** 2) * (np.e ** exp)

#     arr /= arr.sum()
    return arr
```

```
sigma_s = 15 # space
sigma_r = 0.4 # intensity
kernel_size = (5, 5)
```

```
kernel_s = create_filter(sigma_s, kernel_size)
denoised = my_Bilateral_filter(cropped_n, my_5×5_gausskernel=kernel_s, colour_sigma=sigma_r)
```

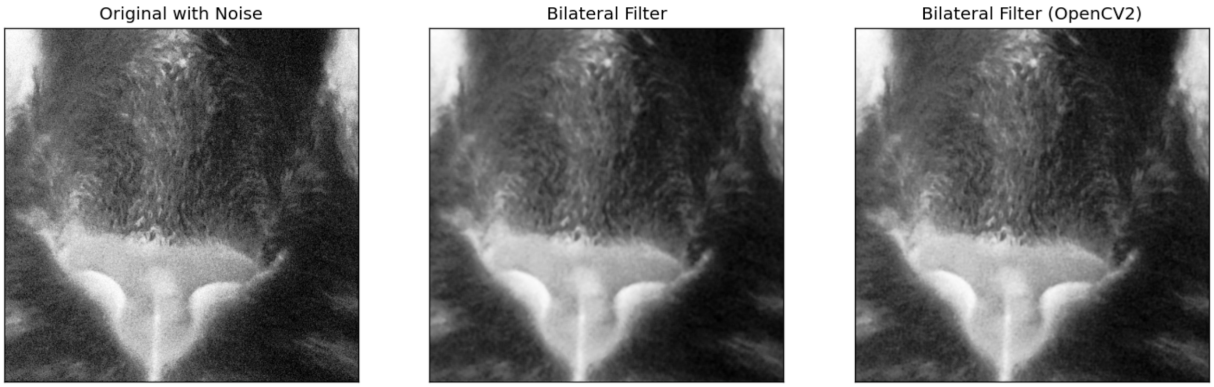
2. Apply your Bilateral filter to the above noisy image (greyscale version) from the last task, and display the smoothed images and visually check their noise-removal and bilateral edge preserving effects (0.5 marks in total).

```
sigma_s = 15 # space
sigma_r = 0.4 # intensity
kernel_size = (5, 5)
```

```
kernel_s = create_filter(sigma_s, kernel_size)
denoised = my_Bilateral_filter(cropped_n, my_5×5_gausskernel=kernel_s, colour_sigma=sigma_r)
```

```
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.xticks([])
plt.yticks([])
plt.imshow(cropped_n, cmap=plt.cm.Greys)
plt.title('Original with Noise')

plt.subplot(1, 3, 2)
plt.xticks([])
plt.yticks([])
plt.imshow(denoised, cmap=plt.cm.Greys)
plt.title('Bilateral Filter')

plt.subplot(1, 3, 3)
denoised_cv = cv2.bilateralFilter(cropped_n.astype(np.float32), kernel_size[0], sigma_r, sigma_s)
plt.xticks([])
plt.yticks([])
plt.imshow(denoised_cv, cmap=plt.cm.Greys)
plt.title('Bilateral Filter (OpenCV2)')
```

```
Text(0.5, 1.0, 'Bilateral Filter (OpenCV2)')
```



In addition to the Gaussian filter, the range filter also has a standard deviation. You may need to test and compare different standard deviations for range (1.0 marks). Note: Do not use in-built functions, the same as for task 4.

See screenshot above

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). (2.0 marks) (Note this task is more difficult again).

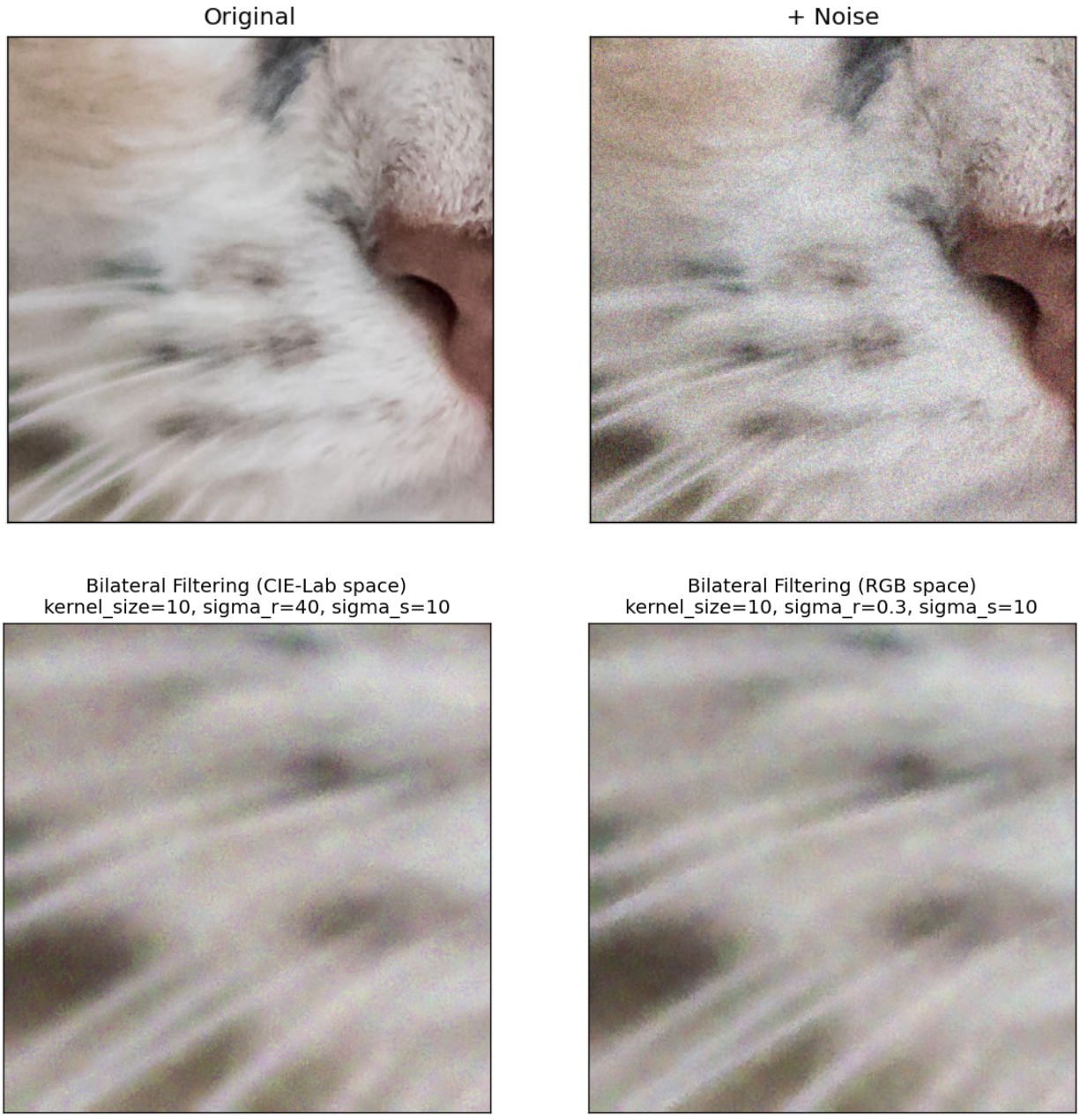
Incomplete - Could not figure out the correct equation…

```python
def bilateral_filter(img, kernel_s, sigma_r):
    padding = kernel_s.shape[0] // 2
    padded_img = np.pad(img, ((padding, padding), (padding, padding), (0, 0)))
    new_img = np.zeros(padded_img.shape)

    for i in range(padding, len(padded_img) - padding):
        for j in range(padding, len(padded_img[0]) - padding):
            # create intensity kernel_r
            color_similarities = np.sqrt(
                np.sum(
                    (padded_img[i-padding:i+padding+1, j-padding:j+padding+1] - padded_img[i,j]) ** 2,
                    axis=2
                )
            )

            kernel_r = stats.norm.pdf(color_similarities, loc=0, scale=sigma_r)

            # multiply the filters with corresponding region in image, then divide by sum of product of kernels to normalize
            new_img[i][j] = np.sum(
                padded_img[i-padding:i+padding+1, j-padding:j+padding+1, k] * kernel_s * kernel_r
            ) / np.sum(kernel_s * kernel_r)

    return new_img[padding:padding+img.shape[0], padding:padding+img.shape[1]]
```

```python
# sigma_s = 15 # space
# sigma_r = 0.4 # intensity
# kernel_size = (5, 5,)
# kernel_s = create_filter(sigma_s, kernel_size)
# denoised_color = bilateral_filter(noisy_img_cie, kernel_s, sigma_r)
```

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. (1 marks)

Bilateral filtering improves on Gaussian filter by taking color similarity into account, in addition to pixel proximity. With this approach, pixels that are closer to the center of the kernel will be given higher weights; however, if the color is very different from the center pixel, the pixel might be given a lower weight (or not be blurred). In order to measure the color similarity, euclidian distance (l2 distance) is used. Since RGB space is highly correlated, colors that are perceptually different may be in close proximity to each other in this space. In theory, this makes RGB space unfavorable for bilateral filtering, and CIE-Lab color space is typically used instead. In CIE-Lab color space, changes in the values that make up a color also reflect the perceptual difference, making distance measured between colors more perceptually meaningful. In the particular image and region selected for comparison, bilateral filtering seemed to achieve identical results despite using RGB space and CIE-Lab space.



Task-6: Image Translation (3 marks)

Choose an image among (image1.jpg, image2.jpg, image3.jpg) and resize it to 512 x 512.

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) (0.20 for each image, 1.0 mark in total). Note: positive for away from upwards and right from the bottom left hand corner.

```
offset = np.random.randint(-100, 100, size=(2,))
offset
```

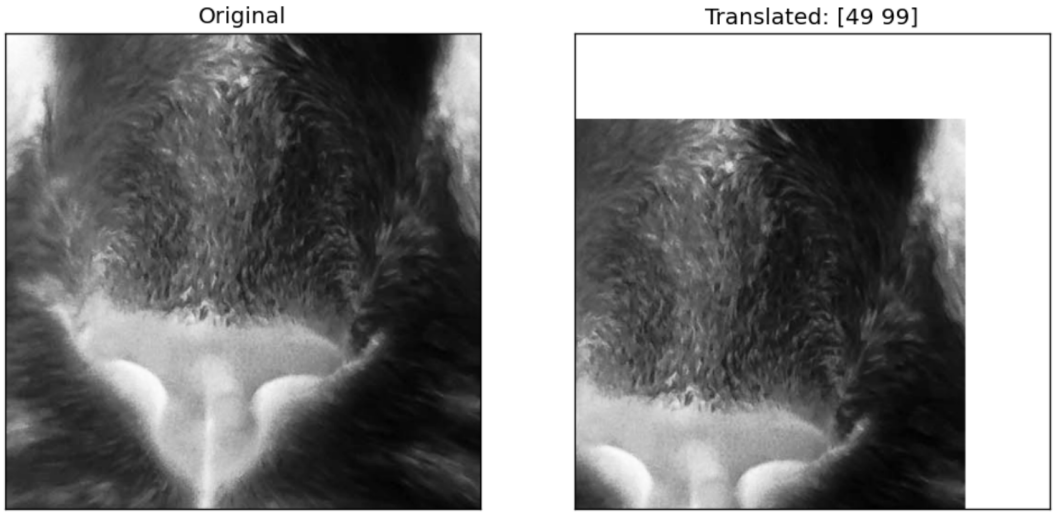
```
array([49, 99])
```

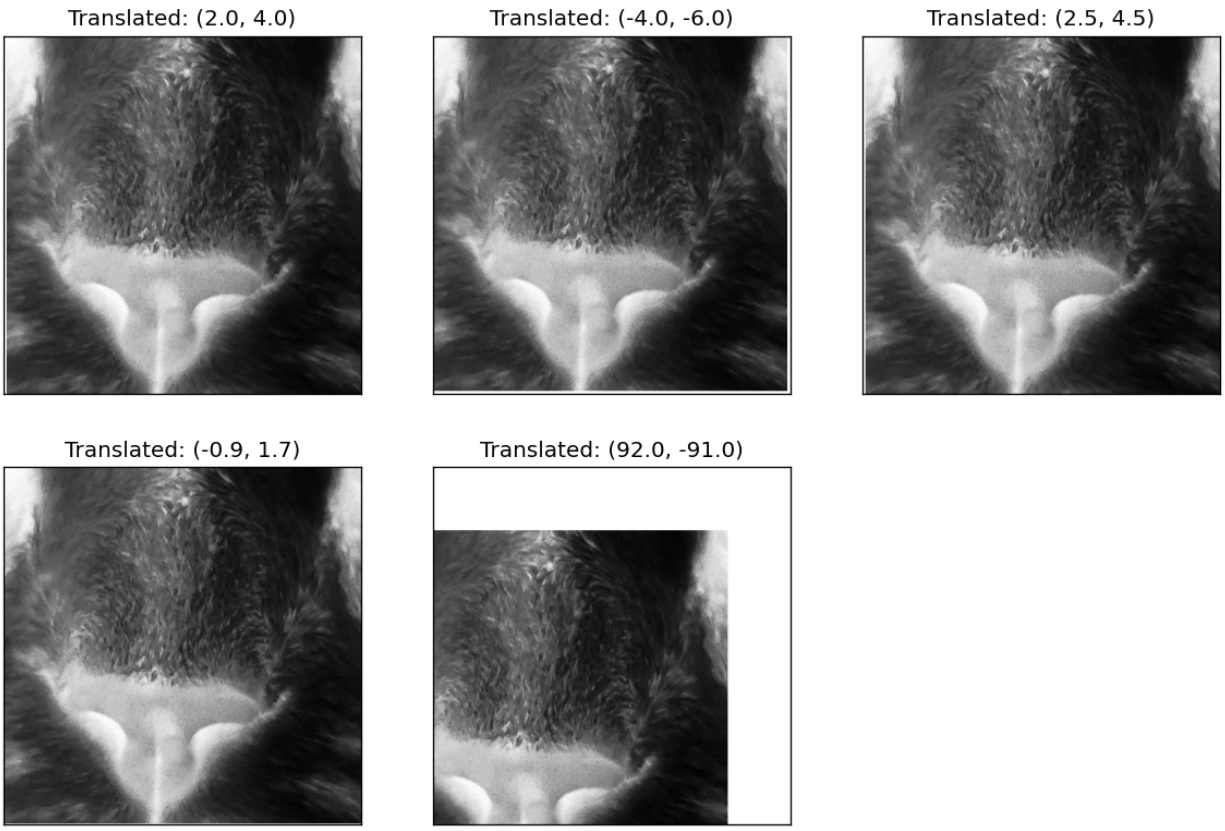
```
def my_translation(img, offset):
    new_img = np.zeros(img.shape)

    for i in range(len(img)):
        for j in range(len(img[0])):
            x, y = int(i + offset[0]), int(j + offset[1])
            if 0 <= x < len(img) and 0 <= y < len(img[0]):
                new_img[x][y] = img[i][j]
    return new_img
```

```
translated = my_translation(cropped, offset)
```

```
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.xticks([])
plt.yticks([])
plt.imshow(cropped, cmap=plt.cm.Greys)
plt.title('Original')

plt.subplot(1, 2, 2)
plt.xticks([])
plt.yticks([])
plt.imshow(new_img, cmap=plt.cm.Greys)
plt.title(f'Translated: {offset}')
```

```
Text(0.5, 1.0, 'Translated: [49 99]')
```





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

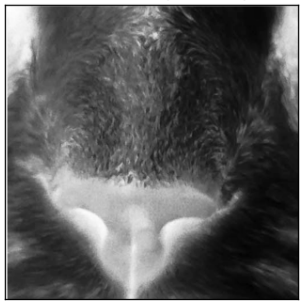
Hint: When analyzing the difference, you can focus on: (1) visual results; (2) the principles in terms of formulation or others relevant; (3) advantages and drawbacks; (4) the computational complexity. You can also think about it from other aspects.

```python
def my_translation_inverse(img, offset):
    tolerance = 0.2

    new_img = np.zeros(img.shape)

    for i in range(len(new_img)):
        for j in range(len(new_img[0])):

            x, y = i - offset[0], j - offset[1]
            int_x, int_y = int(x), int(y)

            if 0 <= x < len(new_img) and 0 <= y < len(new_img[0]):
                if x % 1 > tolerance and y % 1 > tolerance and (int_x < len(img) - 1) and (int_y < len(img[0]) - 1):
                    new_img[i][j] = img[int_x:int_x+2, int_y:int_y+2].mean()
                elif x % 1 > tolerance or y % 1 > tolerance:
                    if x % 1 > tolerance and int_x < len(img) - 1:
                        new_img[i][j] = img[int_x:int_x+2,int_y].mean()
                    elif y % 1 > tolerance and int_y < len(img[0]) - 1:
                        new_img[i][j] = img[int_x,int_y:int_y+2].mean()
                else:
                    new_img[i][j] = img[int_x][int_y]

    return new_img
```

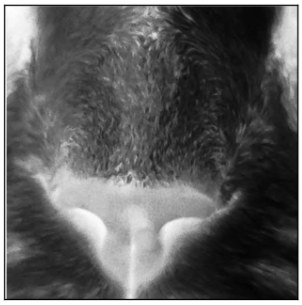
```python
translated_inv = my_translation_inverse(cropped, offset)
```

```python
plt.figure(figsize=(12, 20))
i = 0

for off in [(2.0,4.0), (-4.0,-6.0), (2.5,4.5), (-0.9,1.7), (92.0,-91.0)]:
    i += 1
    new_img = my_translation(cropped, off)
    plt.subplot(5, 3, i)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(new_img, cmap=plt.cm.Greys)
    plt.title(f'Translated: {off}')

    i += 1
    translated_inv = my_translation_inverse(cropped, off)
    plt.subplot(5, 3, i)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(translated_inv, cmap=plt.cm.Greys)
    plt.title('Inverse translation')

    # highlight differences
    i += 1
    if (new_img != translated_inv).any():
        plt.subplot(5, 3, i)
        plt.xticks([])
        plt.yticks([])

        plt.imshow((new_img - translated_inv), cmap=plt.cm.Greys)
        plt.title('Diff')
```

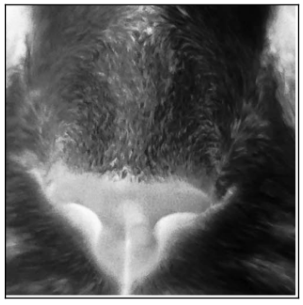
Translated: (2.0, 4.0)



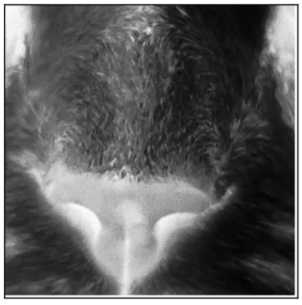
Inverse translation



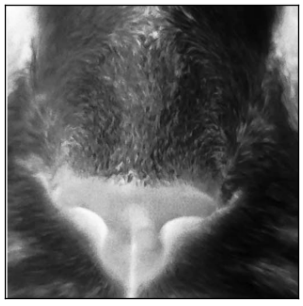
Translated: (-4.0, -6.0)



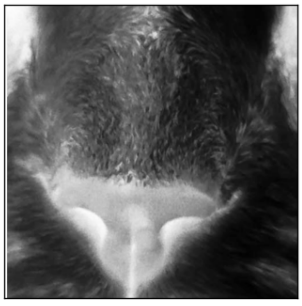
Inverse translation



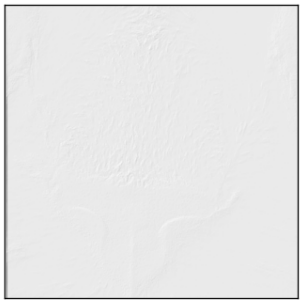
Translated: (2.5, 4.5)



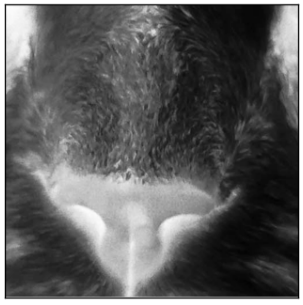
Inverse translation



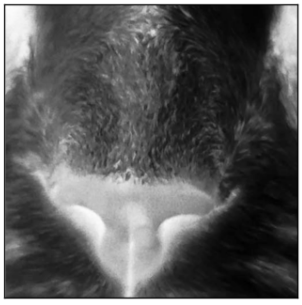
Diff



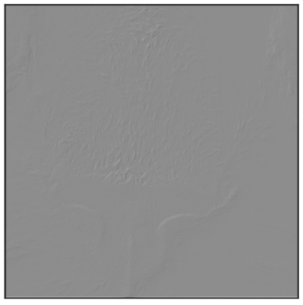
Translated: (-0.9, 1.7)



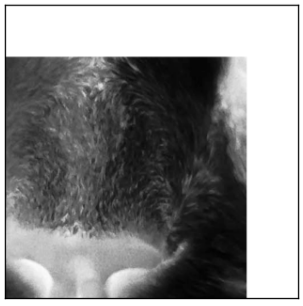
Inverse translation



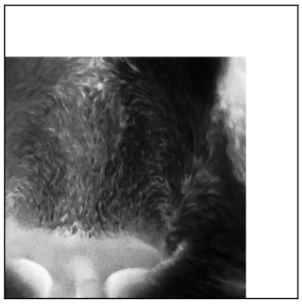
Diff



Translated: (92.0, -91.0)



Inverse translation



1. Visual results

Looking at it visually, there are no visible differences in the results of the two operations.

2. Principles in terms of formulation or others relevant

Forward warping iterates over the source image and calculates the mapped location in the destination image, which is why the new image can have gaps depending on the types of image transformations used. If multiple pixels map to the same location in the destination, older value may be overwritten. Inverse or backward warping starts with the destination image, and calculates the corresponding coordinates in the source image. If the calculated source coordinate falls in between pixels, interpolation will be needed.

3. Advantages and drawbacks

Forward warping is easier to implement, but it is hard to predict the outcome of nonlinear transformations as that might cause multiple pixels in source to be mapped to the same location in the destination image. Inverse warping has the advantage of being able to fill holes, but requires choosing an interpolation method, and only works if the transformation is invertible.

4. The computational complexity

This depends on what transformations are applied, but for simplicity we can assume linear translation in x, y coordinates, which is a constant time operation.

If the image is of dimension (m, n), the worst case time complexity would be O(mn).

3. Compare different interpolation methods and analyze their difference (1.0 mark).

Hint: When analyzing the difference, you can focus on: (1) visual results; (2) the principles in terms of formulation or others relevant; (3) advantages and drawbacks; (4) the computational complexity. You can also think about it from other aspects.

Your description (and/or images, if needed) is here