Homework 1: Image Enhancement Using Spatial Filters

313553014 廖怡誠

Part I. Implementation (5%):

- **Gaussian Filter**

First, I used **formula (1)** to create a 2D Gaussian map, and then it was necessary to normalize the kernel to ensure that the sum of the values equal 1. The 2D Gaussian map is a distribution where points closer to the center have higher weights, while points closer to the edges have lower weights, as shown in **Figure 1.** A 2D Gaussian map can filter out noise and distortion, allowing each pixel to focus on itself. The sigma and kernel size of the 2D Gaussian map are both hyperparameters that can be adjusted. These will be discussed in more detail in subsequent experiments. Then, I applied the calculated Gaussian map (the Gaussian kernel) to convolve the image. In the convolution function, padding is applied to the input image to extend its edges and prevent a reduction in image size after convolution. The padding size is determined by the kernel size divided by 2. For example, if the kernel size is 5, then 2 pixels are added to each edge of the input image. I used zero padding for the added pixels, resulting in an expanded version of the image. Finally, for each step, I extracted a region from the padded image with the same size as the kernel, multiplied it by the Gaussian kernel, and sum the result as the value of the middle point. Since the values in the Gaussian kernel are small, the resulting pixel values are also reduced, so it's necessary to rescale them back to the 0-255 range and set the type to uint8. This calculation is performed separately for each of the three channels and then concatenated in BGR order.

$$h(x,y) = \frac{1}{\sqrt{2\pi\sigma_x^2}\sqrt{2\pi\sigma_y^2}} e^{-(\frac{x^2}{2\sigma_x^2}+\frac{y^2}{2\sigma_y^2})} \quad (1)$$

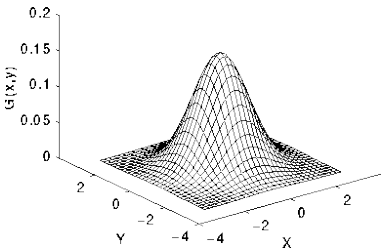


Figure 1. the 2D-gaussian distribution

```python
def gaussian_filter(input_img):

    sigma_x, sigma_y = 51, 51
    # column, row
    kernel_size = (101, 101)
    kernel = np.zeros(kernel_size)
    # y, x offset
    kernel_offset = (kernel_size[0] // 2, kernel_size[1] // 2)
    sqr_sigma_x, sqr_sigma_y = sigma_x**2, sigma_y**2
    for y in range(kernel_size[0]):
        for x in range(kernel_size[1]):
            offset_y = (y - kernel_offset[0])
            offset_x = (x - kernel_offset[1])
            a = np.exp(-1 * ((offset_x**2 / (2 * sqr_sigma_x)) + (offset_y**2 / (2 * sqr_sigma_y))))
            b = np.sqrt(2 * np.pi * sqr_sigma_x) * np.sqrt(2 * np.pi * sqr_sigma_y)
            kernel[y][x] = a / b
    kernel /= np.sum(kernel)

    # 3D surface
    # x = np.arange(kernel_size[1])
    # y = np.arange(kernel_size[0])
    # X, Y = np.meshgrid(x, y)

    # fig = plt.figure()
    # ax = fig.add_subplot(111, projection='3d')
    # ax.plot_surface(X, Y, kernel, cmap='viridis')
    # ax.set_title("3D Surface Plot of Gaussian Kernel")
    # ax.set_xlabel("X")
    # ax.set_ylabel("Y")
    # ax.set_zlabel("Intensity")
    # plt.savefig(f"map_kernel{kernel_size[0]}.png")

    return convolution(input_img, kernel)
def convolution(input_img, kernel):

    kernel_height, kernel_width = kernel.shape
    img_height, img_width = input_img.shape[:2]
    kernel_size = kernel.shape[:2]
    kernel_offset = (kernel_size[0] // 2, kernel_size[1] // 2)
    if input_img.shape[2] == 3:
        BGR_img = []
        for c in range(3):

            padded_img = padding(input_img[:,:,c],  kernel_size)
            output_img = np.zeros_like(input_img[:,:,c], dtype=np.float32)
            for y in range(img_height):
                for x in range(img_width):
                    region = padded_img[y : y + 2 * kernel_offset[0] + 1, x : x + 2 * kernel_offset[1] + 1]
                    output_img[y, x] =  np.sum(region * kernel)

            BGR_img.append(np.clip(output_img, 0, 255).astype(np.uint8))

        # concat B, G, R channel
        output_img = np.dstack(BGR_img)

    return output_img
def padding(input_img, kernel_size):
    # """ zero padding """
    h, w = input_img.shape[:2]
    kernel_offset = (kernel_size[0] // 2, kernel_size[1] // 2)
    output_img = np.zeros((h + 2 * kernel_offset[0], w + 2 * kernel_offset[1]), dtype= input_img.dtype)
    output_img[kernel_offset[0]: kernel_offset[0] + h, kernel_offset[1]: kernel_offset[1] + w] = input_img

    return output_img
```

- **Median Filter**

The median filter is simpler than the Gaussian filter. First, you only need to set the kernel size, and then you can proceed with the convolution. Padding is applied initially to avoid reducing the input image size during convolution, as mentioned with the Gaussian filter. Then, for each step, a region of the input image matching the kernel size is extracted, and the pixel values within this region are sorted. The median value is selected as the new center pixel value. This process is repeated for each BGR channel, then concatenated, ensuring the values remain within the 0-255 range. For example, consider a 3x3 matrix with the following values: [[5, 6, 2], [3, 208, 3], [1, 4, 2]]. After extracting and sorting the pixel values, we get 1, 2, 2, 3, 3, 4, 5, 6, 208. The median is 3, so the center value (208) would be replaced by 3.

```python
def median_filter(input_img):

    kernel_size = (21, 21)
    # y, x offset
    kernel_offset = (kernel_size[0] // 2, kernel_size[1] // 2)

    if input_img.shape[2] == 3:

        BGR_img = []

        for c in range(3):
            padding_img = padding(input_img[:,:,c], kernel_size)
            output_img = np.zeros_like(padding_img)
            for y in range(kernel_offset[0], output_img.shape[0] - kernel_offset[0]):
                for x in range(kernel_offset[1], output_img.shape[1] - kernel_offset[1]):
                    output_img[y][x] = np.median(\
                        padding_img[y - kernel_offset[0]: y + kernel_offset[0] + 1,\
                                    x - kernel_offset[1]: x + kernel_offset[1] + 1].\
                                        reshape((kernel_size[0] * kernel_size[1])))

            BGR_img.append(output_img)

        # concat B, G, R channel
        output_img = np.dstack(BGR_img)

    return output_img
```

- **Laplacian Filter**

The purpose of the Laplacian filter is to detect edges in an image. The mathematical derivation uses the second derivative to identify zero-crossings between pixels. However, in practice, specific masks can achieve similar results. In the masks, the center pixel value is set in relation to the values of the surrounding four pixels, with one positive and one negative value. By applying these masks, we can detect edges in the image.

In the implementation, I predefined the values for the two masks and then performed convolution (the same function as the Gaussian filter). For each step, a region of the padded image that matches the kernel size is extracted and multiplied by the mask; then, the results are summed up. If the center pixel's value significantly differs from that of the surrounding pixels, it indicates the presence of an edge.

```python
def laplacian_sharpening(input_img):

    # filter1
    kernel = np.array([ [0, -1, 0],
                        [-1, 5, -1],
                        [0, -1, 0]])

    # filter2
    kernel = np.array([ [-1, -1, -1],
                        [-1, 9, -1],
                        [-1, -1, -1]])

    # filter3
    # kernel = np.array([ [-1, -1, -1],
    #                     [-1, 11, -1],
    #                     [-1, -1, -1]])

    # return cv2.filter2D(input_img, -1, kernel)

    return convolution(input_img, kernel)
```

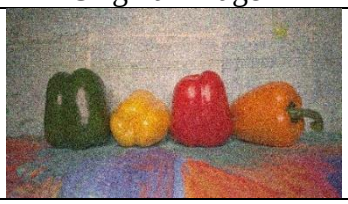
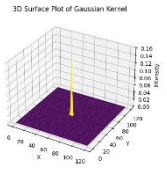
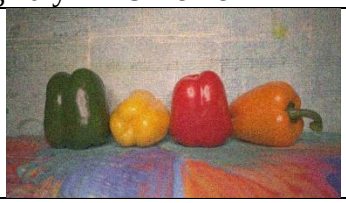
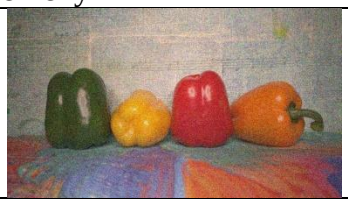
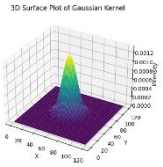
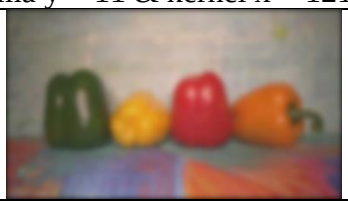
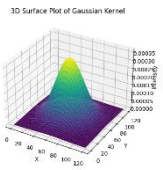
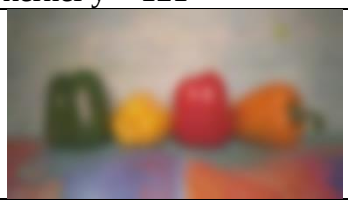
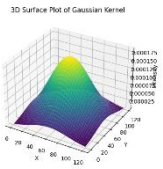
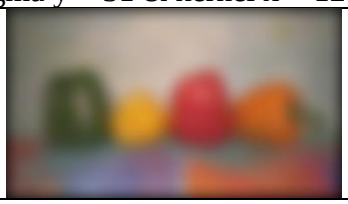
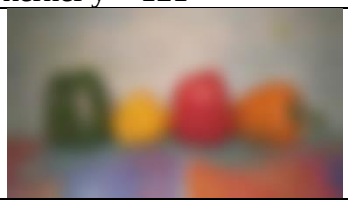
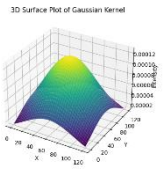
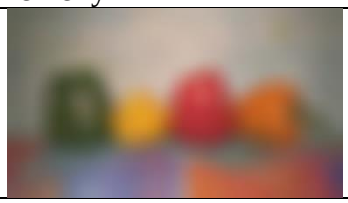
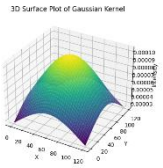
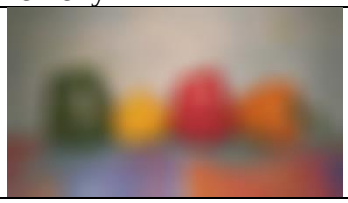
Part II. Results & Analysis (10%):

- **Gaussian filter**

  - **Try three different σ and compare the results**

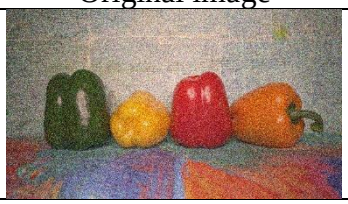
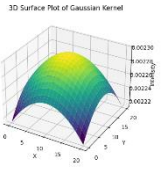
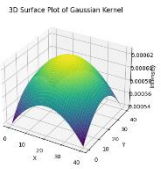
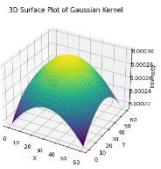
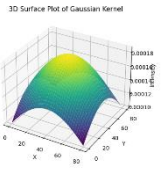
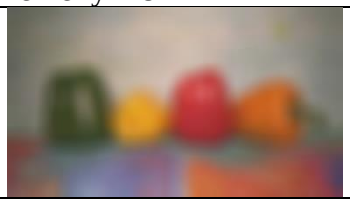
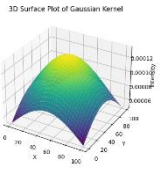
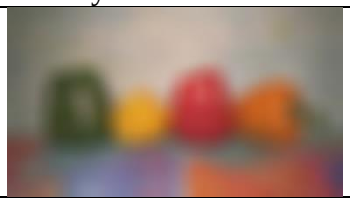
In this section, besides comparing images obtained with different sigma values, I also used **cv2.GaussianBlur()** to generate images to check for any errors in my implementation. Finally, for easier observation, I visualized the corresponding 3D surfaces. I fixed the kernel size at 121 and compared the results for sigma values of 11, 21, 31, 41, and 51. On the 3D surface, as sigma increases, the curve becomes smoother, which results in a smoother (i.e., more blurred) image. With a fixed kernel size, we can observe the effect of different sigma values on the kernel within the same coverage area. By examining the z-axis of the 3D surface, it’s evident

that the values decrease as sigma increases. Therefore, we can also say that a larger sigma, on an x-y plane of the same size, results in reduced variation along the z-axis, giving the kernel a flattened appearance. Compared to the original image, the noise is eliminated, but the image also loses many details.

Original image		
		
3D surface of the kernel	**My implement**	**cv2.GaussainBlur**
sigma x = 1 & sigma y = 1 & kernel x = 121 & kernel y = 121		
		
sigma x = 11 & sigma y = 11 & kernel x = 121 & kernel y = 121		
		
sigma x = 21 & sigma y = 21 & kernel x = 121 & kernel y = 121		
		
sigma x = 31 & sigma y = 31 & kernel x = 121 & kernel y = 121		
		
sigma x = 41 & sigma y = 41 & kernel x = 121 & kernel y = 121		
		
sigma x = 51 & sigma y = 51 & kernel x = 121 & kernel y = 121		
		

○ **Try three different filter sizes and compare the results.**

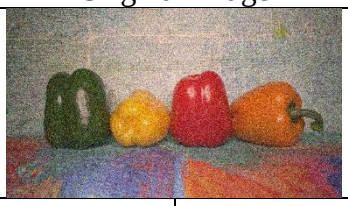
Here, I fixed the sigma at 51 and compared the results for different kernel sizes of 21, 41, 61, 81, 101, and 121. Fixing sigma means that the height along the z-axis remains essentially constant, while different kernel sizes represent variations in the coverage area on the x-y plane. Therefore, we can observe from the 3D surface that as the kernel size increases, the number of squares increases, and overall, the surface becomes more curved and smoother, resulting in a very smooth image.

Original image		
		
3D surface of the kernel	**My implement**	**cv2.GaussainBlur**
sigma x = 51 & sigma y = 51 & kernel x = 21 & kernel y = 21		
		
sigma x = 51 & sigma y = 51 & kernel x = 41 & kernel y = 41		
		
sigma x = 51 & sigma y = 51 & kernel x = 61 & kernel y = 61		
		
sigma x = 51 & sigma y = 51 & kernel x = 81 & kernel y = 81		
		
sigma x = 51 & sigma y = 51 & kernel x = 101 & kernel y = 101		
		
sigma x = 51 & sigma y = 51 & kernel x = 121 & kernel y = 121		

- **Median filter**

○ **Try three different filter sizes and compare the results.**

I compared 21, 41, 61, 81, 101, and 121 kernel sizes in this section. As the kernel size increases, the image becomes more blurred, reducing noise and losing many details. Additionally, since I used zero padding, the black edges at the image's borders also become larger as the kernel size increases.

Original image	
	
My implement	**cv2.GaussainBlur**
kernel x = 21 & kernel y = 21	
	
kernel x = 41 & kernel y = 41	
	
kernel x = 61 & kernel y = 61	
	
kernel x = 81 & kernel y = 81	
	
kernel x = 101 & kernel y = 101	

	
kernel x = 121 & kernel y = 121	
	

- **Smoothing Spatial Filters**

  - **Compare the results of Gaussian filter and median filter.**

    The two tables above show that the images obtained with the median filter visually lose more details. Observing from a kernel size of 21 up to 121, it becomes evident that the median filter blurs the image more quickly, and the black borders at the edges are more pronounced. This is likely because the median filter selects a median value as the new pixel value, unlike the Gaussian filter, which calculates a weighted sum within the kernel. The median filter is more prone to selecting 0, which is why it loses more details.

- **Laplacian filter**

  - **Compare the results of two Laplacian filters.**

    The function of the Laplacian filter is to sharpen the image. We can see that the original image appears relatively blurry, but after applying the Laplacian filter, the resulting image is clearer. Furthermore, filter 2 produces an even sharper result than filter 1; for example, the yellow grass on the ground in the image is visually clearer and more distinguishable.

<table>
<tr><td colspan="3">Original image</td></tr>
<tr><td colspan="3">  </td></tr>
<tr><td>Filter</td><td>My implement</td><td>cv2.filter2D</td></tr>
<tr><td>

0	-1	0
-1	5	-1
0	-1	0

Filter 1
</td><td>  </td><td>  </td></tr>
<tr><td>

-1	-1	-1
-1	9	-1
-1	-1	-1

Filter 2
</td><td>  </td><td>  </td></tr>
</table>

Part III. Answer the questions (15%):

1. Please describe a problem you encountered and how you solved it.

When implementing the Gaussian filter, the pixel values obtained after convolution fall between 0 and 1. If these values are directly output, the result will be a completely black image. Therefore, it's necessary to use **np.clip(output_img, 0, 255).astype(np.uint8)** to map the range back to [0, 255]. Additionally, when calculating the Gaussian kernel, it is essential to normalize it so that the sum of the values within it equals 1. Initially, I averaged the convolved pixel values during the convolution stage, but this was the incorrect approach, resulting in the outcomes shown in **figures 2** and **3**.

Figure 2. Wrong result with Sigma 3.

Figure 3. Wrong result with Sigma 5.

2. What padding method do you use, and does it have any disadvantages? If so, please suggest possible solutions to address them.

I used zero padding, which expands the image edges with zeros. This is the simplest method to implement, and it yields not bad results. The drawback is that as the amount of padding increases, the black areas at the edges of the image also become larger. For example, when using a kernel size of 121 with the Gaussian filter (**Figure 4**), the resulting image shows some black edges, which are more pronounced in the median filter results (**Figure 5**). The quickest solution is to avoid zero padding and instead use methods like replicate or mirror (reflect) padding that do not fill with zeros. While this will still cause some slight blurring at the edges of the image, it prevents the appearance of obvious black edges, as shown in **Figures 6 and 7**.

Figure 4. Zero padding result with kernel 121 (Gaussian filter).

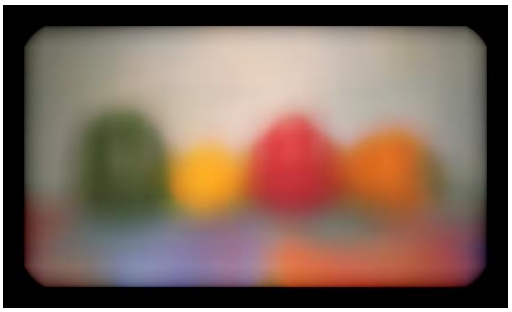


Figure 5. Zero padding result with kernel 121. (Median filter).

Figure 6. Reflect padding result with kernel 121. (Gaussian filter).

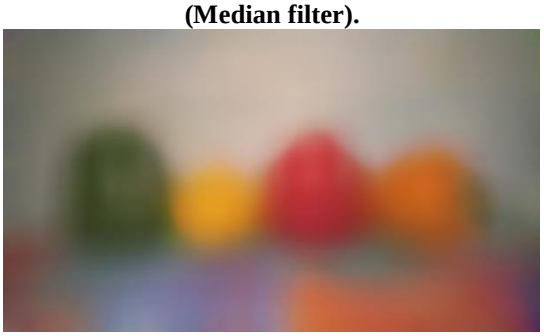


Figure 7. Reflect padding result with kernel 121. (Median filter).

3. What problems do you encounter when using Gaussian filter and median filter to denoise images? Please suggest possible solutions to address them.

Since the Gaussian filter applies uniform averaging across the image, it affects both noisy and non-noisy pixels, causing original details to disappear and edges to blur. A non-local smoothing method could be used here, such as a bilateral filter or joint bilateral filter. These methods combine color and spatial information to preserve details while effectively removing noise. A clear issue with the median filter is that it doesn't use averaging, making it easier to lose details and disrupt the original lines in the image. Additionally, as the kernel size increases, the computational cost also rises. Selecting different kernel sizes for different regions to prevent line disruptions in the original image could reduce line loss. Of course, using non-local smoothing methods would also help retain more details.

4. What problems do you encounter when using Laplacian filters to sharpen images? Please suggest possible solutions to address them.

The Laplacian filter has two main issues. The first is that it amplifies noise, mistakenly treating it as high-frequency information, resulting in a noticeable graininess. The second issue is that as the contrast within the kernel increases, the brightness difference in the resulting image becomes pronounced, causing the image to be overly bright (as seen in Figure 8). To address the first issue, applying a Gaussian blur before using the Laplacian filter can reduce high-frequency noise, making the result smoother, or using edge detection beforehand can help avoid noisy areas. For the second issue, combining the original image with the output from the Laplacian filter can reduce halo intensity. Specifically, this can be achieved by subtracting the low-pass filtered image from the original or using a guided or bilateral filter to enhance sharpening.

Figure 8. Laplacian filter with sharper kernel.