CV2024 HW2 Filtering and Frequency

Group10 - 313551073 顏琦恩, 313581027 葉彥谷, 313552001 黃梓濤

I. Introduction

The task of this homework is to produce three types of images: the hybrid image, the image pyramid, and the color image from the digitized Prokudin-Gorskii glass plate images. The three topics will be briefly introduced below.

A hybrid image is the combination of the high-frequency content of one image and the low-frequency content of another image. It is created by separating frequencies and combining images. This technique leverages the human visual system's way of processing different spatial frequencies, allowing a single image to contain layered information.

An image pyramid is a multi-scale representation of an image, where the image is progressively downsampled to form a series of images with varying scales. This is useful in computer vision and image processing because it enables algorithms to process images at different levels of detail, facilitating tasks such as image analysis, image compression, and multi-resolution processing.

The last task, colorizing the Russian Empire refers to the process of adding color to black-and-white photographs from the Russian Empire. To colorize these images, we leverage the glass plate images, which record three exposures of every scene onto a glass plate using red, green, and blue filters. The three color channels are extracted and combined to form a single RGB image, restoring the original scene in full color.

II. Implementation Procedure

Task 1. Hybrid Images

Before applying the Fourier transform, we split the images into RGB three channels for preserving details of each channel. To create hybrid images, the same channel of two images are passed through the Gaussian low-pass filter and high-pass filter respectively. The low-pass (Eq.1) and high-pass (Eq.2) filter is implemented through the following equations.

$$H(u,\ v) = e^{-D^2(u,\ v)/2{D_0}^2} \qquad (1)$$

$$H(u,\ v) = 1 - e^{-D^2(u,\ v)/2{D_0}^2} \qquad (2)$$

After going through the filter, we apply the inverse Fourier transform to restore the image channel and obtain the real part of it. At last, we combine the low-pass image with the high-pass image to get our final result, the hybrid image. The following code block presents the details of our code.

```
def hybrid_image(image1, image2, D0_low=30, D0_high=30):
    # Resize images to the fit the smaller image
    if image1.shape[:2] != image2.shape[:2]:
        if image1.shape[0] * image1.shape[1] > image2.shape[0] *
image2.shape[1]:
            image1 = cv2.resize(image1, (image2.shape[1], image2.shape[0]))
        else:
            image2 = cv2.resize(image2, (image1.shape[1], image1.shape[0]))

    # Applying Gaussian filter on each channel
    hybrid_img = np.zeros_like(image1, dtype=np.float64)
    for channel in range(3):
        low_pass_image = gaussian_filter(image1[:, :, channel], D0_low,
filter_type='low')
        high_pass_image = gaussian_filter(image2[:, :, channel], D0_high,
filter_type='high')

        hybrid_img[:, :, channel] = low_pass_image + high_pass_image
    hybrid_img = np.clip(hybrid_img, 0, 255).astype(np.uint8)

    return hybrid_img

def gaussian_filter(image, D0, filter_type='low'):
    rows, cols = image.shape
    crow, ccol = rows // 2, cols // 2

    # Fourier transform and centered
    fft_image = fftshift(fft2(image))

    u = np.arange(rows) - crow
    v = np.arange(cols) - ccol
    U, V = np.meshgrid(v, u)
    D = np.sqrt(U**2 + V**2)

    # Gaussian filter
    H = np.exp(-(D**2) / (2 * (D0**2))) # low-pass
    if filter_type == 'high':
        H = 1 - H                       # high-pass

    # Inverse Fourier transform
    filtered_fft = fft_image * H
    ifft_image = ifft2(ifftshift(filtered_fft))
    filtered_image = np.real(ifft_image)

    return filtered_image
```

Task 2. Image Pyramid

A Gaussian Pyramid is a multi-scale representation of an image, where each successive layer is a downsampled and smoothed version of the previous one. This technique is widely used in image processing tasks such as compression, feature extraction, and multi-scale analysis. The goal is to progressively reduce the resolution and detail in the image while maintaining key structural information. In this report, we outline the steps involved in constructing a Gaussian Pyramid using Gaussian smoothing and subsampling, supported by code implementation.

The process begins by setting the finest scale of the pyramid as the original image. For each subsequent layer, the image is first smoothed using a Gaussian filter, which reduces high-frequency content, and then downsampled by a factor of 2 to reduce the resolution. This process is repeated until the desired number of layers is achieved, with each layer capturing progressively coarser details. Finally, the pyramid layers can be saved for visualization or further analysis. This method efficiently generates a multi-resolution representation of the image.

```
def gaussian_kernel(size, sigma=1):
    k = size // 2
    x, y = np.mgrid[-k:k+1, -k:k+1]
    g = (1 / (2.0 * np.pi * sigma**2)) * \
        np.exp(-((x**2 + y**2) / (2.0 * sigma**2)))
    return g / g.sum()


def convolve2d(image, kernel):
    kernel_height, kernel_width = kernel.shape
    image_height, image_width = image.shape

    pad_height = kernel_height // 2
    pad_width = kernel_width // 2
    # zero_padding
    padded_image = np.pad(image, ((pad_height, pad_height),
                           (pad_width, pad_width)), mode='constant',
constant_values=0)

    output_image = np.zeros_like(image)

    for i in range(image_height):
        for j in range(image_width):
            region = padded_image[i:i + kernel_height, j:j + kernel_width]
            output_image[i, j] = np.sum(region * kernel)

    return output_image


def gaussian_blur(image, kernel_size, sigma):
    kernel = gaussian_kernel(kernel_size, sigma)
```

```
    channels = cv2.split(image)
    blurred_channels = [convolve2d(ch, kernel) for ch in channels]
    blurred_image = cv2.merge(blurred_channels)

    return blurred_image


def downsample(image, factor):
    if factor <= 1:
        raise ValueError("Downsampling factor must be greater than 1")

    channels = cv2.split(image)
    downsampled_channels = [ch[::factor, ::factor] for ch in channels]
    downsampled_image = cv2.merge(downsampled_channels)

    return downsampled_image


def gaussian_pyramid(image, num_layers):
    pyramid = [image]
    for i in range(num_layers - 1):
        image = gaussian_blur(image, kernel_size=5, sigma=1)
        image = downsample(image, factor=2)
        pyramid.append(image)
    return pyramid
```

Task 3. Colorizing the Russian Empire

This task is about aligning three channel images into one RGB picture. The challenge here is to align each channel properly, because each image has some slight shifting compared to the others. The simplest way is to calculate the difference between pixels in different channels, then shift the channel and repeat the process. However, this method can be time consuming when the image resolution is high, so we use a Gaussian pyramid to progressively align the channels from low resolution to high resolution. Since the low-resolution image can roughly align the geometric features of the image, there is no need to align the entire high resolution image. You only need to use a fixed size shifting area (window) to align the channels, and you can achieve great results.

The key factors that influence the results are window size (shifting boundary) and the cost function. In our experiment, we compared different cost functions and window sizes, and found that a larger window size gave better results. Additionally, normalizing the image before calculating the sum of square error helped optimize the alignment effectively.

```
"""error"""
def square_error(image_a, image_b):

image_a=(image_a-image_a.mean(axis=0))/(image_a.max(axis=0)-image_a.min(axis=0))
```

```
image_b=(image_b-image_b.mean(axis=0))/(image_b.max(axis=0)-image_b.min(axis=0))
    return np.sum((image_a - image_b) ** 2)

"""align"""
def align_channel(fix_ch, align_ch, window_size=15):
    min_error = float('inf')
    align_offset = (0, 0)

    for y_offset in range(-window_size, window_size + 1):
        for x_offset in range(-window_size, window_size + 1):
            al_ch = np.roll(align_ch, (y_offset, x_offset), axis=(0, 1)) # shift
the image channel
            s_error = square_error(fix_ch, al_ch)  # calculate error

            # error
            if s_error < min_error:
                min_error = s_error
                align_offset = (y_offset, x_offset)

    align_ch = np.roll(align_ch, align_offset, axis=(0, 1))
    return align_ch

"""alignment with gaussian pyramid"""
def image_pyramid_align(fix_ch, align_ch, levels=4, window_size=15):
    fix_pyramid = [fix_ch]
    align_pyramid = [align_ch]

    # gaussian pyramid downsample
    for _ in range(levels - 1):
        fix_pyramid.append(cv2.pyrDown(fix_pyramid[-1])) # gaussian pyramid
        align_pyramid.append(cv2.pyrDown(align_pyramid[-1]))  # gaussian pyramid

    # Align from the smallest scale to the largest
    align_ch = align_pyramid[-1]
    for i in range(levels - 1, -1, -1):
        align_ch = align_channel(fix_pyramid[i], align_ch, window_size//(2 **
i)) # decrease window size to reduce computation
        if i > 0:
            align_ch = cv2.pyrUp(align_ch, dstsize=(fix_pyramid[i - 1].shape[1],
fix_pyramid[i - 1].shape[0]))

    return align_ch
```

III. Experimental Results

We conducted the experiment with two different data setups, which are images from TA and images captured ourselves. Note: The description of the output file is provided in Appendix I.

Task 1. Hybrid Images

The following figures present the comparison between high-resolution images and low-resolution images using dataset provided either by TA or by us.

Figure. 1 Comparison results of hybrid images using TA's dataset ($D_0 = 15$).

Figure. 2 Comparison results of hybrid images using our dataset ($D_0 = 15$). Images in the first row went through the low-pass filter, while the second row went through the high-pass filter.

Task 2. Image Pyramid

The following figures present the results of image pyramid using dataset provided either by TA or by us.

Figure. 3 Results of image pyramid using TA's dataset.

Figure. 4 Results of image pyramid using our dataset.

Task 3. Colorizing the Russian Empire

Below are some colorization results from the TA-provided dataset and our captured data. It can be observed that our results are quite good, as normalization stabilizes the alignment. Further comparisons will be discussed in the discussion section.

Figure. 5 Results of Colorizing images using TA's dataset

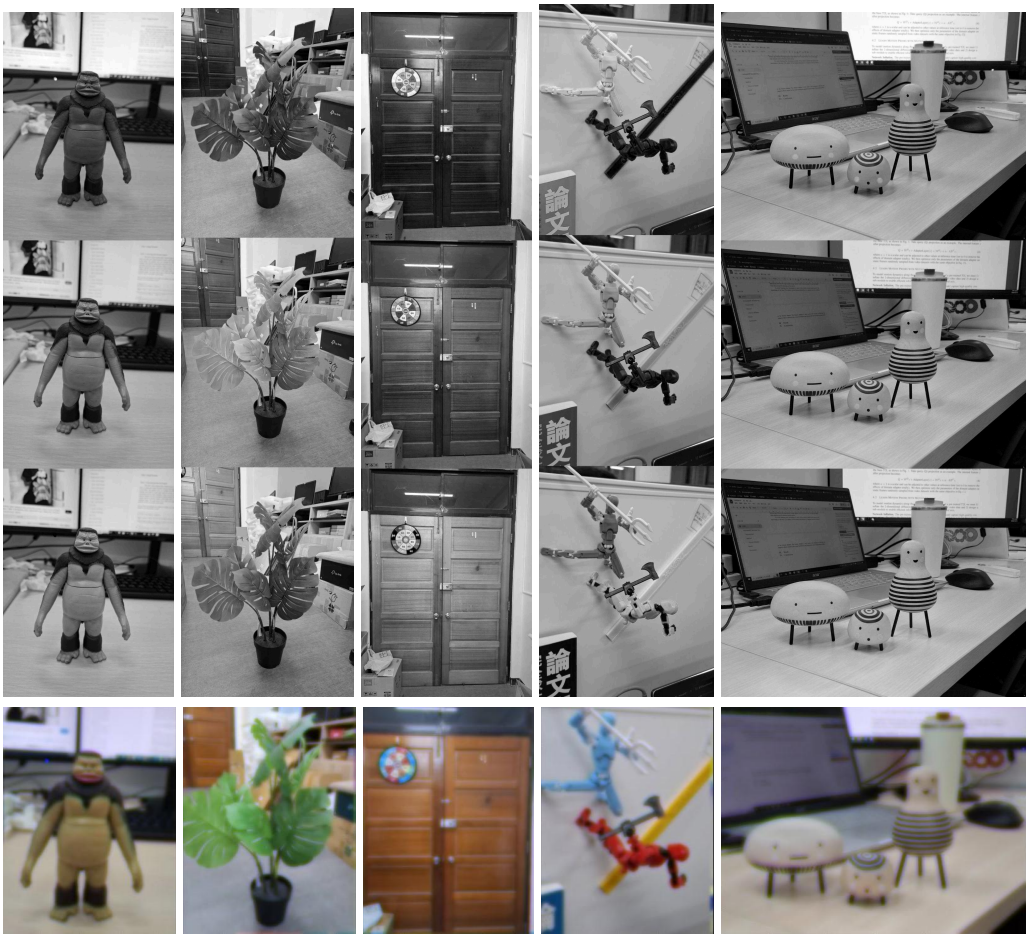

Figure. 6 Results of Colorizing images using our captured data.

IV. Discussion

1. Hybrid Images

Since there is a cutoff frequency in the equation of the Guassian filter, we've tried different frequencies to filter the image frequency. From the result, we can see that the lower cutoff frequency results in a blurrier background (or a clearer edge).

Figure. 7 Comparison results of different cutoff frequencies. (D_0 of left image = 30, D_0 of right image = 15).

2. Different parameters in Gaussian blur

In Gaussian blur, the kernel size determines the area over which the pixel values are averaged, impacting the amount of smoothing or blurring applied to the image. In Fig. 8, there is the comparison of using different kernel sizes. However, it doesn't show a significant difference in the results.

For the sigma parameter, it controls the standard deviation of the Gaussian function, which determines the spread or "width" of the blur. A higher sigma value means a wider Gaussian kernel, which leads to stronger blurring because the averaging is spread over a larger area. In Fig. 9, we can see that the image of sigma=2 is more blurry than that of sigma=1.

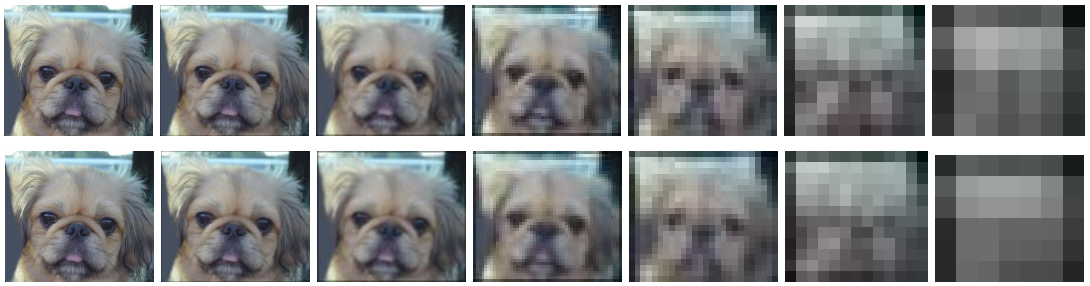

Figure. 8 Comparison results of different kernel sizes of gaussian blur. (top image with kernel =3 , bottom image with kernel=7).

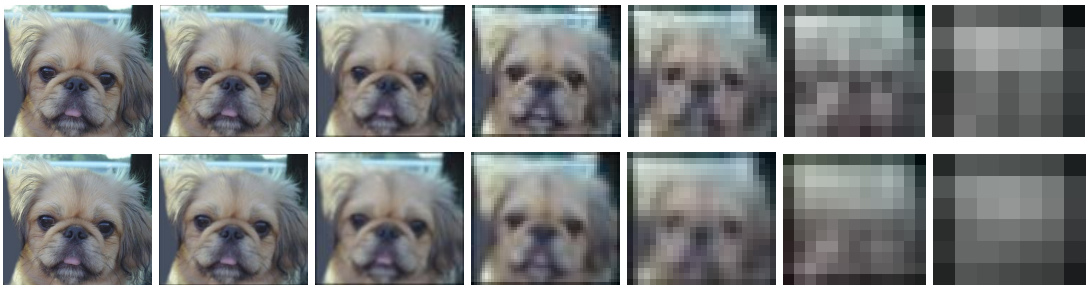

Figure. 9 Comparison results of different sigma of gaussian blur. (top image with sigma=1, bottom image with sigma=2).

3. Colorizing the Russian Empire

Normalization and Cost function

At the beginning of the implementation, we used the sum of squared differences (SSD) to measure the difference between two channel images. However, using only SSD to optimize the alignment led to significant misalignment (Fig. 10). This might be due to differences in the intensity distribution between the two channels, which can result in incorrect alignment. To address this issue, we applied min-max normalization to the images before calculating the SSD, which significantly enhanced the colorization results.

Figure. 10 Comparison: without normalization (top) vs. with normalization (bottom).

Window Size

The window size also affects the quality of results. A larger window can produce more accurate alignment but requires more processing time, making it a trade-off. We compared normalized results with different window sizes (15, 30) and observed that a window size of 30 gave better results.

Figure. 11 Comparison: window size 15 (top) vs. window size 30 (bottom).

V. Conclusion

In conclusion, we've accomplished three tasks in this homework, which is producing (1) the hybrid images, (2) the image pyramid, and (3) colorizing the Russian Empire.

VI. Work Assignment Plan

- ❖ 葉彥谷 : Task3 and report.
- ❖ 顏琦恩 : Task1 and report.
- ❖ 黃梓濤 : Task2 and report.

Appendix I : Filename Description

Path	Description
hybrid.py	Code of task 1
pyramid.py	Code of task 2
colorize.py	Code of task 3, include the main function
align.py	Code of task 3
\my_data\task1amd2_hybrid_pyramid\	Our dataset for task 1 and task 2
\my_data\task3_colorizing\	Our dataset for task 3
\output\task1_hybrid\cus_data\	Output for task 1 using our data
\output\task1_hybrid\TA_data\	Output for task 1 using TA's data
\output\task2_pyramid\cus_data\	Output for task 2 using our data
\output\task2_pyramid\TA_data\	Output for task 2 using TA's data
\output\task3_colorizing\cus_data\	Output for task 3 using our data
\output\task3_colorizing\TA_data\	Output for task 3 using TA's data