Computer Vision - HW2 Hybrid image & Colorizing the Russian Empire

Group 18

312552026 蔡濟謙, 313552049 鄭博涵, 110550035 陳奎元

1. Introduction

In this assignment, there are three tasks: Hybrid Image, Image Pyramid, and Colorizing the Russian Empire. A Hybrid Image is a picture that shows different visual effects depending on the viewing distance. It is created by combining a low-pass filtered image with a high-pass filtered image. From a distance, you will see the low-pass filtered image, while from up close, the high-pass filtered image becomes visible. The Image Pyramid involves downsampling an image multiple times, resulting in a blurred image, which represents a process from high resolution to low resolution. Colorizing the Russian Empire is a technique for coloring black-and-white images by extracting information from the three color channels in glass plates and then combining them to create a colorized image.

2. Implement Procedure

a. Hybrid Image

First, we need to calculate the image's center position which is used for filter positioning.

```
def decide_size(image, div):    # div: dividion ratio
    center_x = image.shape[0] / div + (image.shape[0] % 2)
    center_y = image.shape[1] / div + (image.shape[1] % 2)
    return center_x, center_y
```

Next, generates a filter matrix, which can be either Gaussian or ideal, and determines whether it is a low-pass or high-pass filter based on the provided parameters.

It first calculates the center of the image, then computes the filter value for each pixel based on the selected filter type.

For a Gaussian filter, it uses a Gaussian function to assign weights, whereas for an ideal filter, it determines whether to retain

certain frequencies based on the distance. Finally, it creates a filter matrix suitable for frequency domain filtering.

```python
def create_filter(image, sigma, filter_type='gaussian', highlow=True):
    # sigma: filter's parameter, determine frequency influence scope, highlow: determine high-pass or low-pass filter (True -> low-pass filter)
    center_x, center_y = decide_size(image, 2)

    if filter_type == 'gaussian':
        def filter_func(i, j):
            entry = math.exp(-1.0 * ((i - center_x) ** 2 + (j - center_y) ** 2) / (2 * sigma ** 2))
            return entry if highlow else 1 - entry
    elif filter_type == 'ideal':
        def filter_func(i, j):
            entry = 1 if math.sqrt((i - center_x) ** 2 + (j - center_y) ** 2) <= sigma else 0
            return entry if highlow else 1 - entry
    else:
        raise ValueError("Unknown filter type")

    filter_array = np.array([ [filter_func(i, j) for j in range(image.shape[1])] for i in range(image.shape[0]) ])
    return filter_array
```

Second, we applied a specified filter to each channel of an image in the frequency domain. It first performs a Discrete Fourier Transform on each channel using “fft2” and shifts the zero-frequency component to the center with “fftshift”. Then, it multiplies the transformed image by the filter matrix created by create_filter(). Finally, it performs an inverse Fourier Transform to bring the image back to the spatial domain and collects all processed channels into a single image.

```python
def apply_filter(image, sigma, filter_type='gaussian', highlow=True):
    # DFT: Discrete Fourier Transform
    channels = []
    for i in range(image.shape[2]):     # apply filter to every channels
        shift_DFT = fftshift(fft2(image[:, :, i]))
        filter_DFT = shift_DFT * create_filter(image, sigma, filter_type, highlow)
        channels.append(np.real(ifft2(ifftshift(filter_DFT))))
    return np.stack(channels, axis=-1)
```

Third, generate hybrid image by using apply_filter() function and adjust two images to save size.After that, we need to constraint image pixel value from 0 to 255, and cast datatype to ‘uint8’.

```python
""" generate hybrid image (integrate high-frequency and low-frequency between two images) """
def hybrid_image(image1, image2, high_sigma, low_sigma, filter_type):
    return apply_filter(image1, high_sigma, filter_type, highlow=False) + apply_filter(image2, low_sigma, filter_type, highlow=True)

""" adjust two images to same size """
def resize_images(img1, img2):
    h1, w1 = img1.shape[:2]     # img1's height and width
    h2, w2 = img2.shape[:2]
    h, w = min(h1, h2), min(w1, w2)
    return resize(img1, (h, w)), resize(img2, (h, w))

""" constraint image pixel value to 0~255, and cast datatype to 'uint8' """
def clip_and_cast(array):
    return array.clip(0, 255).astype(np.uint8)
```

Finally, plot and save the result.

```python
def plot_and_save(image, title, save_path):
    plt.imshow(image, cmap='gray')  # cmap='gray': display image in grayscale
    plt.title(title, fontsize=12)
    plt.savefig(save_path, dpi=300)
    plt.clf()    # clean current figure

def process_and_save_hybrid(img1, img2, high_sigma, low_sigma, filter_type, result_name, data_choose):
    filter_image = hybrid_image(img1, img2, high_sigma, low_sigma, filter_type)
    filter_image = filter_image / np.max(filter_image)
    filter_image = np.clip(filter_image, 0.0, 1.0)

    title = f'{filter_type.capitalize()} Filter -> High_sigma: {high_sigma}, Low_sigma: {low_sigma}'
    save_path = os.path.join('output/task1_result', f'{data_choose}_{result_name}_{filter_type}_hybrid.png')
    plot_and_save(filter_image, title, save_path)
```

b. Image Pyramid

First, I defined a class ImagePyramid that constructs both Gaussian and Laplacian pyramids for image processing.

In gaussian_filter(), it constructs filter by initializing a zero matrix of the given size and then filling it with values calculated based on the Gaussian distribution formula. And the computed Gaussian values ensure that the center of the filter has the highest value, with values decreasing symmetrically towards the edges.

Finally, filter is normalized so that all values sum to 1, which helps preserve the overall image intensity during filtering.

```python
class ImagePyramid:
    def __init__(self, filter_size, sigma):
        self.kernel = self.gaussian_filter(filter_size, sigma)

    """ construct gaussian filter """
    def gaussian_filter(self, filter_size, sigma):     # sigma: Gaussian standard
        kernel = np.zeros((filter_size, filter_size))
        center = filter_size // 2

        for i in range(filter_size):
            for j in range(filter_size):
                g = np.exp(
                    -((center - i) ** 2 + (center - j) ** 2) / (2 * (sigma ** 2))
                )   # compute Gaussain value
                g /= 2 * np.pi * (sigma ** 2)
                kernel[i, j] = g    # assign g to correct kernel position
        kernel /= kernel.sum()
        return kernel
```

Second, use Gaussian filter to an image using cv2.filter2D() which helps smoothen the image by reducing noise and softening the details.

Next, generates a Gaussian pyramid of an image by iteratively smoothing and downsampling the image. The iterative time determined by num_layers.

And calculates the magnitude spectrum of an image. It applies a 2D Fast Fourier Transform to convert the image to the frequency domain, followed by a shift (fftshift) to move the zero-frequency component to the center of the image

```
""" use Gaussian filter to smooth image """
def smooth(self, img, kernel):
    return cv2.filter2D(img, -1, kernel)

""" Create Gaussian pyramid """
def gaussian_pyramid(self, img, num_layers, kernel):
    res = [np.array(img)]
    for i in range(num_layers - 1):
        img = self.smooth(img, kernel)
        img = img[::2, ::2]     # downsample
        res.append(np.array(img))
    return res

""" Caculate image's magnitude spectrum """
@ staticmethod
def magnitude_spectrum(img):
    fshift = fftshift(fft2(img))
    return np.log(np.abs(fshift) + 1e-7)
```

Third, generates a Laplacian pyramid by taking the difference between each layer of the Gaussian pyramid and its upsampled version. It starts from the smallest layer of the Gaussian pyramid and iteratively upsamples and subtracts to highlight edges and details. The final result is a list of Laplacian layers, reversed to start from the highest resolution layer.

```
""" Create Laplacian pyramid """
def laplacian_pyramid(self, g_pyramid, num_layers):   # g_pyramid: Gaussian pyramid
    res = [g_pyramid[-1]]   # initialize result list, start at peak of Gaussian pyramid
    for i in range(1, num_layers):
        upsample = cv2.pyrUp(g_pyramid[num_layers - i])
        if g_pyramid[num_layers - i - 1].shape != upsample.shape:
            upsample = cv2.resize(upsample, (g_pyramid[num_layers - i - 1].shape[1], g_pyramid[num_layers - i - 1].shape[0]))

        laplacian = cv2.subtract(g_pyramid[num_layers - i - 1], upsample)   # calculate laplacian layer
        res.append(laplacian)
    return res[::-1]    # reverse it. (start at bottom layer)
```

Finally, visualizes and saves the results of the Gaussian and Laplacian pyramids. It creates a 4-row plot:

- row 1: Gaussian pyramid images.
- row 2: Laplacian pyramid images.
- row 3: Magnitude spectra of the Gaussian pyramid.
- row 4: Magnitude spectra of the Laplacian pyramid.

```python
def plt_result(g_pyramid, l_pyramid, num_layers, filter_size, filter_sigma, img_name):
    plt.figure(figsize=(20, 20))
    plt.suptitle(f"filter_size = {filter_size} x {filter_size}, sigma = {filter_sigma}", fontsize=32)

    text_kwargs = {     # to control appearance text annotations
        'size': 18,
        'ha': 'right',  # horizon alignment to the right
        'va': 'center', # vertical alignment to the center
        'rotation': 'vertical'    # display text by vertical
    }
    for i in range(num_layers):
        plt.subplot(4, num_layers, i + 1)    # 4: total has 4 column subgraph, num_layers: each column has num_layers subgraph, i+1: current subgraph's position
        plt.imshow(g_pyramid[i], cmap='gray'), plt.xticks([], []), plt.yticks([], [])   # xticks([], []): used to hide x axis tick label
        plt.title(f"level {i}", fontsize=20)

        if i == num_layers - 1:     # When arrive final layer, annotate particular text to make it easy to understand
            plt.gca().text(1.3, 0.5, 'Gaussian (final layer)', transform=plt.gca().transAxes, **text_kwargs)    # 1.3, 0.5: x, y coordinate

        plt.subplot(4, num_layers, num_layers + i + 1)
        plt.imshow(l_pyramid[i], cmap='gray'), plt.xticks([], []), plt.yticks([], [])

        if i == num_layers - 1:
            plt.gca().text(1.3, 0.5, 'Laplacian (final layer)', transform=plt.gca().transAxes, **text_kwargs)

        plt.subplot(4, num_layers, num_layers * 2 + i + 1)
        plt.imshow(ImagePyramid.magnitude_spectrum(g_pyramid[i])), plt.xticks([], []), plt.yticks([], [])

        if i == num_layers - 1:
            plt.gca().text(1.3, 0.5, 'Gaussian Sqectrum (final layer)', transform=plt.gca().transAxes, **text_kwargs)

        plt.subplot(4, num_layers, num_layers * 3 + i + 1)
        plt.imshow(ImagePyramid.magnitude_spectrum(l_pyramid[i])), plt.xticks([], []), plt.yticks([], [])

        if i == num_layers - 1:
            plt.gca().text(1.3, 0.5, 'Laplacian Sqectrum (final layer)', transform=plt.gca().transAxes, **text_kwargs)

        plt.tight_layout()  # automatically adjust the spaces between subplots
        plt.savefig(os.path.join('output/task2_result', f'{img_name}_pyd_{num_layers}_size_{filter_size}_sigma_{filter_sigma}.png'), dpi=300)  # dpi: Dots Per Inch
```

c. Colorizing the Russian Empire

First, removes white borders by checking the edge pixels of the image. And removes black borders by cropping the image dimensions to 92% of the original height and width. Therefore, retain only the central part.

```python
""" remove white border from image """
def remove_white_border(img):
    white = 215     # because 215 is close to white value
    while np.all(img[0, :] >= white):   # check the image's first column. If all pixels > 215, remove that.
        img = img[1:, :]    # remove the first column

    while np.all(img[-1, :] >= white):  # check the image's last column.
        img = img[:-1, :]

    while np.all(img[:, 0] >= white):   # check the image's first row.
        img = img[:, 1:]

    while np.all(img[:, -1] >= white):
        img = img[:, :-1]
    return img

def remove_black_border(img):
    h, w = img.shape
    new_h, new_w = int(0.92 * h), int(0.92 * w)
    return img[h - new_h : new_h, w - new_w : new_w]
```

Second, we need to aligns one image channel **(to_align)** with a reference channel **(base)** using translation within a defined **(search_image)**.

It starts from normalized both channels. Then, the function iterates over possible shifts in the x and y directions to find the best offset that minimizes the squared difference between the two

channels. It uses ‘np.roll()’ to shift the ‘to_align’ channel and compares it to the ‘base’ channel.

```python
""" use x, y translation model to align color channels. """
def align_channels(base, to_align, search_range=15):    # base: base channel, to_align: channel to be aligned
    best_offset = (0, 0)
    min_diff = float('inf')
    base = (base - np.mean(base)) / np.std(base)    # standard base channel
    to_align = (to_align - np.mean(to_align)) / np.std(to_align)     # Normalize the pixel value of the channel to the range between 0 and 1

    for x in range(-search_range, search_range + 1):    # search best bias to smallest the difference bewteen two channels.
        for y in range(-search_range, search_range + 1):
            shifted = np.roll(np.roll(to_align, y, axis=0), x, axis=1)     # shift along the y-axis by best_offset[1] pixels, then shift along

            min_height = min(shifted.shape[0], base.shape[0]) # to crop base or shifted so that they can remain the same size
            min_width = min(shifted.shape[1], base.shape[1])
            shifted_cropped = shifted[:min_height, :min_width]
            base_cropped = base[:min_height, :min_width]

            diff = np.sum((base_cropped - shifted_cropped) ** 2)
            if diff < min_diff:
                min_diff = diff
                best_offset = (x, y)
    aligned = np.roll(np.roll(to_align, best_offset[1], axis=0), best_offset[0], axis=1)
    aligned = aligned[:base.shape[0], :base.shape[1]]   # crop to match the size of the base channel
    return aligned
```

Third, Process the glass plate image, split it to R, G, B channels, and align them. Finally merge them into an RGB image.

It starts from read image, and remove white and black border. Next, get Gaussian kernel for smoothing the image. And instance of the ImagePyramid class is created to generate the Gaussian pyramids for the blue, green, and red channels. After that, the green and red channels are aligned to the blue channel.

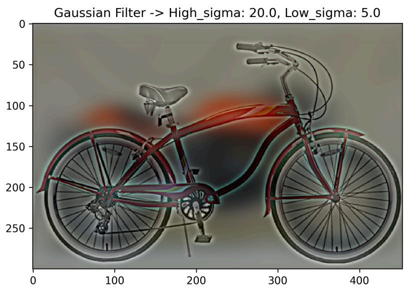
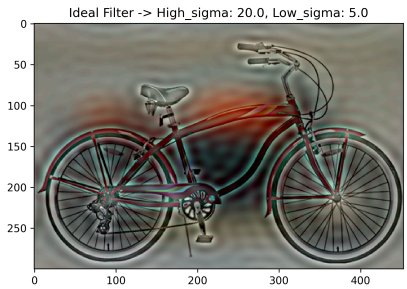
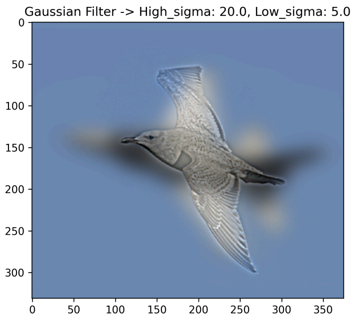
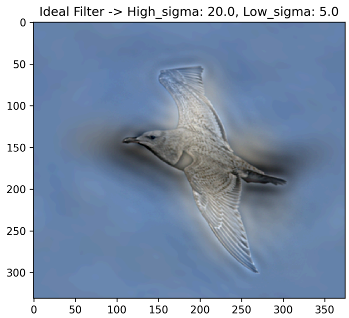
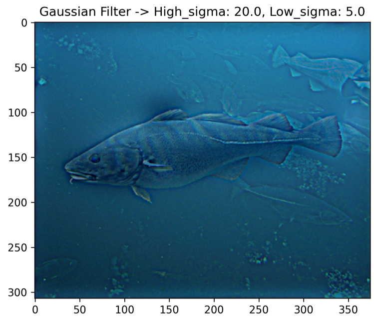
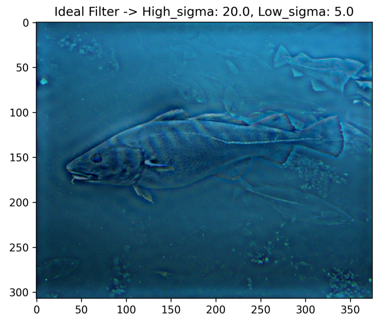
However, all channels are cropped and rescaled to **uint8** before merging. To ensure they have matching dimensions and proper visualization.

```python
""" Process the glass plate image, split it to R, G, B channels, and align them, merge them into an RGB image """
def process_glass_plate(img_path, pyramid_layer=5, filter_size=5, filter_sigma=1.):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img = remove_white_border(img)
    height = img.shape[0] // 3      # img.shape[0]: the height of image, divided by 3 to get each colors height (R, G, B)
    b_channel = remove_black_border(img[:height, :])     # The first 1/3 of the image is used as the blue channel
    g_channel = remove_black_border(img[height:2 * height, :])   # The middle 1/3 of the image is used as the green channel
    r_channel = remove_black_border(img[2 * height:, :])     # , is used to distinguish x, y axis. -> form 2/3 of the image height to the end, selecting all rows.

    kernel = cv2.getGaussianKernel(filter_size, filter_sigma)
    kernel = kernel * kernel.T

    pyramid = ImagePyramid(filter_size, filter_sigma)

    g_pyramid_b = pyramid.gaussian_pyramid(b_channel, pyramid_layer, kernel)[::-1]
    g_pyramid_g = pyramid.gaussian_pyramid(g_channel, pyramid_layer, kernel)[::-1]
    g_pyramid_r = pyramid.gaussian_pyramid(r_channel, pyramid_layer, kernel)[::-1]

    """ align G, R channels to B channels """
    for i in range(1, pyramid_layer):
        g_aligned = align_channels(g_pyramid_b[i], g_pyramid_g[i])
        r_aligned = align_channels(g_pyramid_b[i], g_pyramid_r[i])

    """ Ensure all channels have the same size """
    min_height = min(b_channel.shape[0], g_aligned.shape[0], r_aligned.shape[0])
    min_width = min(b_channel.shape[1], g_aligned.shape[1], g_aligned.shape[1])
    b_channel = b_channel[:min_height, :min_width]
    g_aligned = g_aligned[:min_height, :min_width]
    r_aligned = r_aligned[:min_height, :min_width]

    """ transform channels to uint8 """
    b_channel = b_channel.astype(np.uint8)
    g_aligned = ((g_aligned - np.min(g_aligned)) / (np.max(g_aligned) - np.min(g_aligned)) * 255).astype(np.uint8)   # Rescale these data to the range of 0 to 255
    r_aligned = ((r_aligned - np.min(r_aligned)) / (np.max(r_aligned) - np.min(r_aligned)) * 255).astype(np.uint8)

    color_image = cv2.merge((b_channel, g_aligned, r_aligned))
    return color_image
```

3. Experimental Result

a. Hybride Image

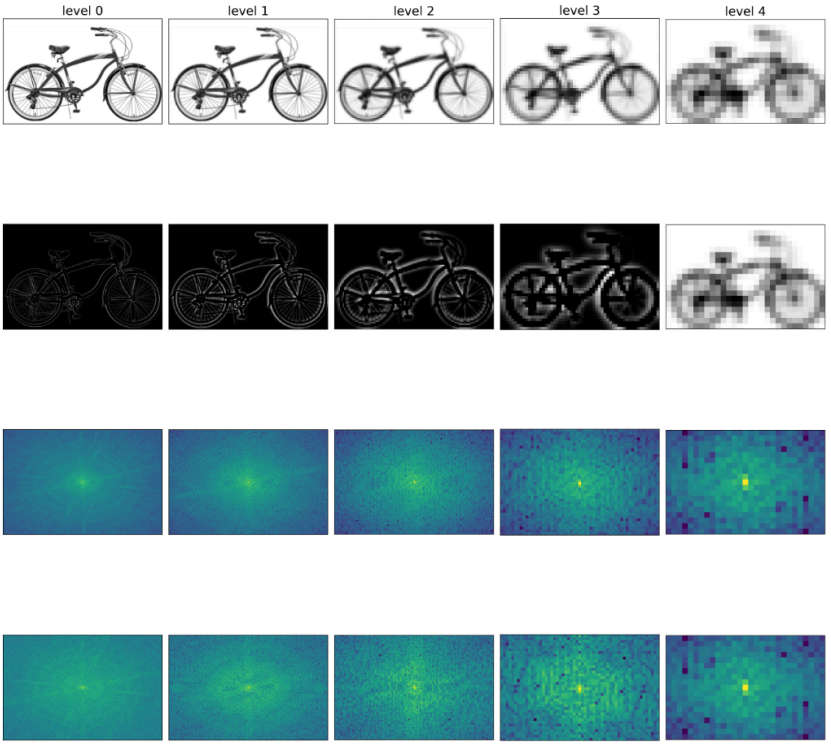
Each row of images corresponds to the gaussia and ideal filter.

<table>
<tr><th>Gaussian</th><th>Ideal</th></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

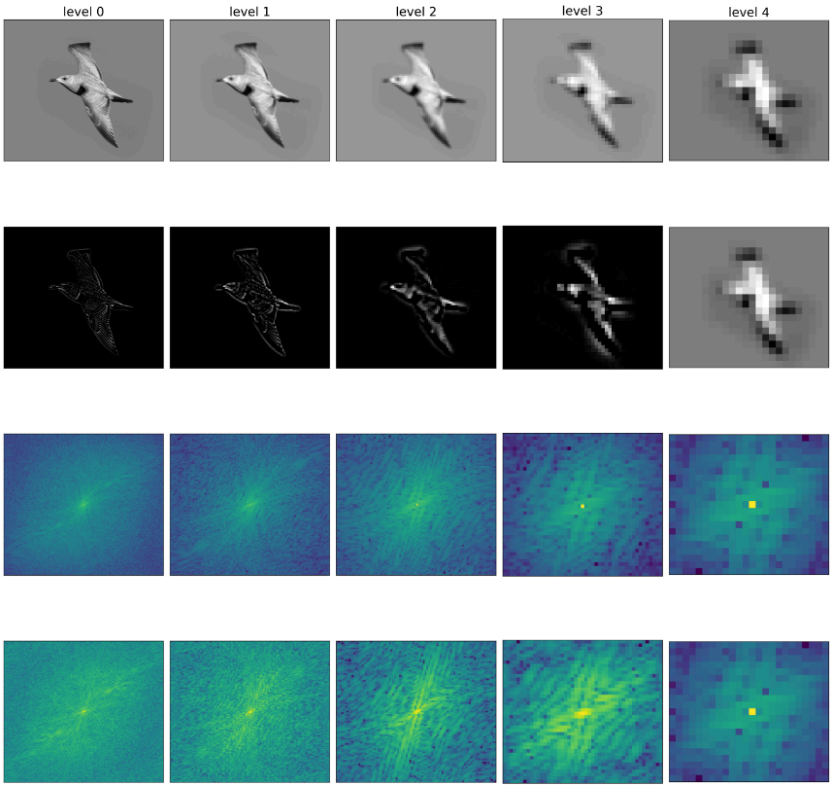
b. Image Pyramid

Each row of images corresponds to the Gaussian pyramid, Laplacian pyramid, magnitude spectrum of the Gaussian pyramid, and Laplacian pyramid.

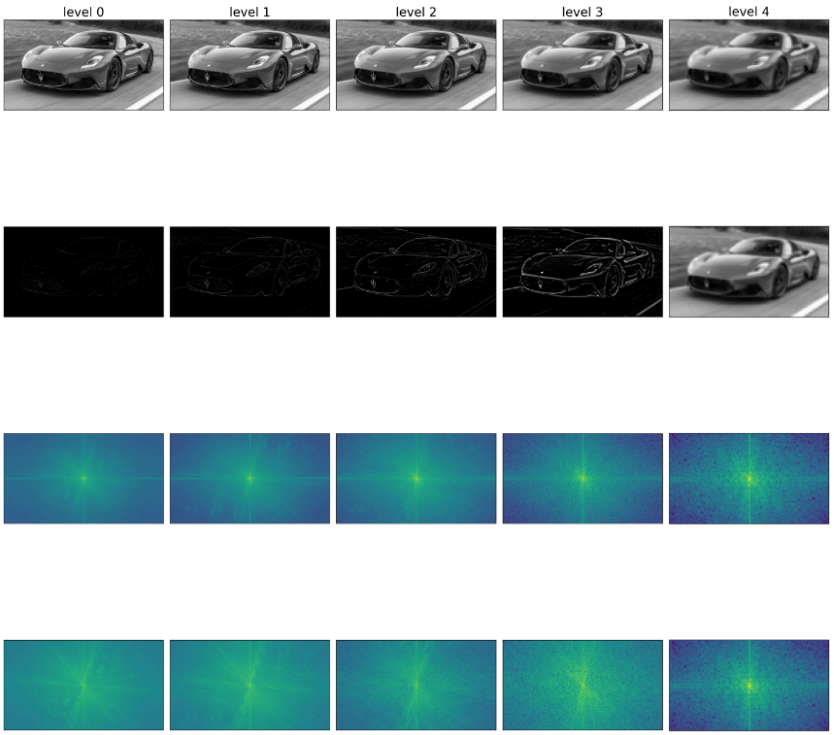
- TA's data: 1_bicycle.bmp



- TA’s data: 2_bird.bmp

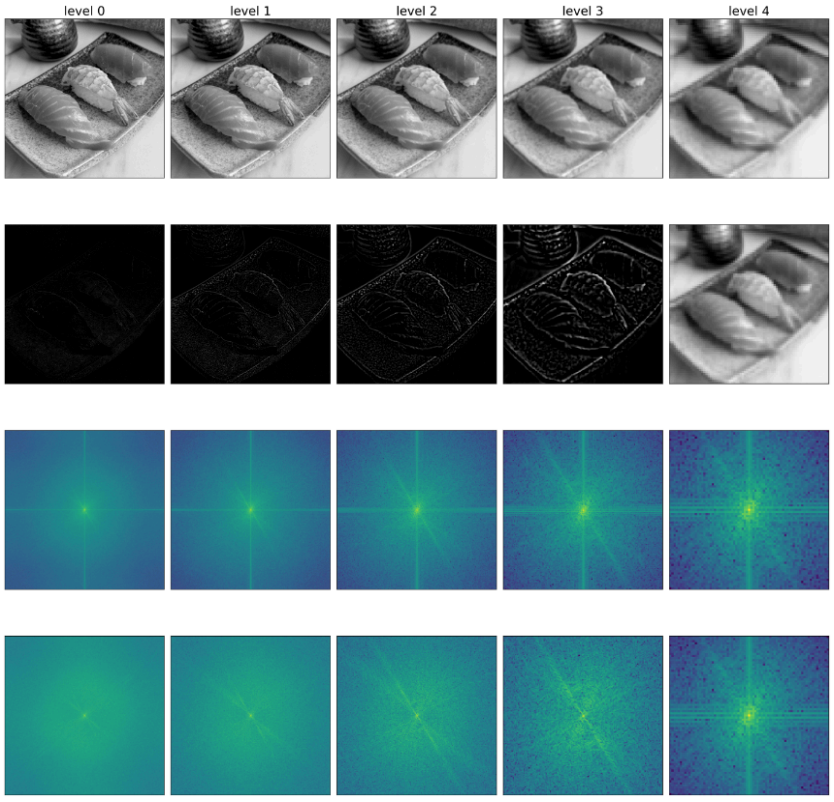


- Our data: car



Reference: https://carwow-uk-wp-3.imgix.net/18015-MC20BluInfinito-scaled-e1707920217641.jpg

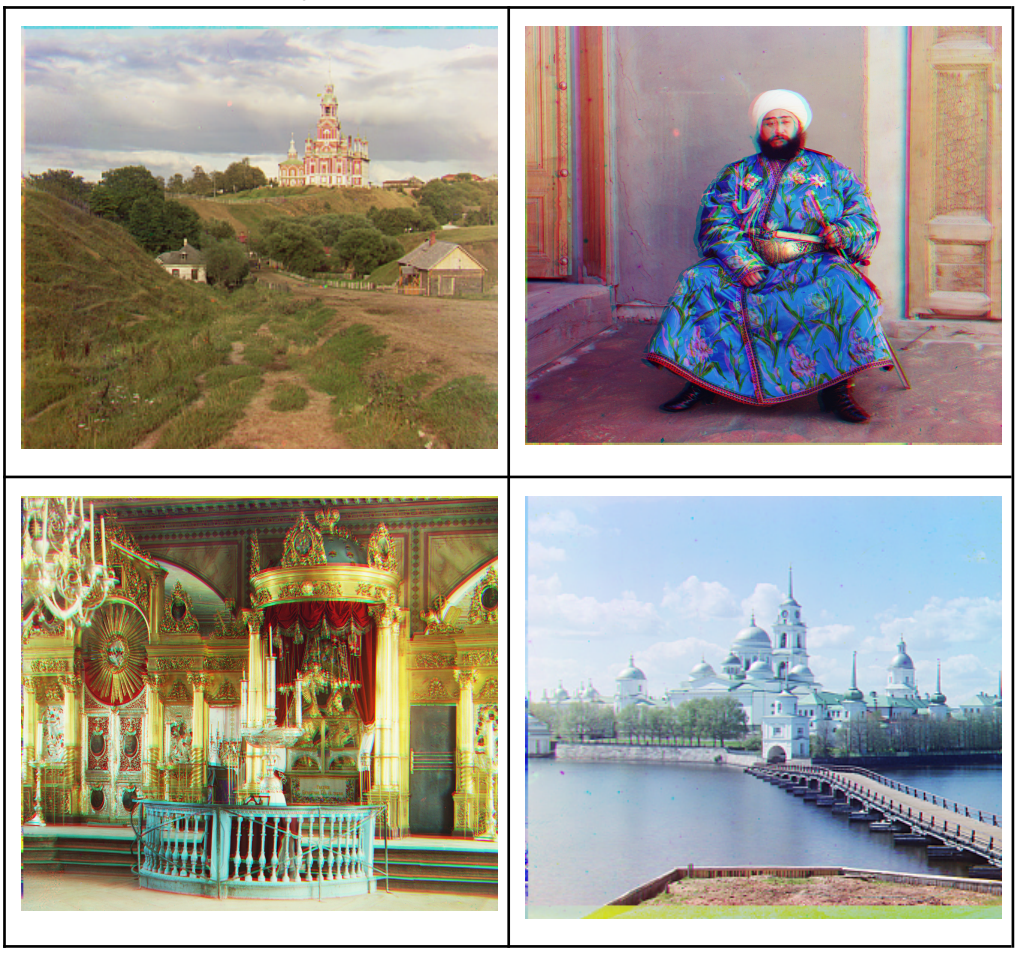
- Our data: sushi



Reference:
https://i0.wp.com/www.craftycookbook.com/wp-content/uploads/2024/04/img_4608-2-1.jpg

c. Colorizing the Russian Empire

The following image was generated using part of the photos provided by the TA.



4. Discussion

a. Hybrid Image

- Using different ratios can result in different visual effects. When the ratio is higher, the low-pass filtered image becomes more blurred. When the ratio is smaller, the high-pass filtered image becomes more blurred. Finding an appropriate ratio to achieve the best visual effect when viewed from different distances is important.

- The size of the images affects the visual effect. If the size difference between the two images is too different, the visual effect may not be optimal.

b. Image Pyramid
   - The Gaussian filter is a low-pass filter that removes high-frequency signals and retains only low-frequency signals. On the other hand, the Laplacian filter is a high-pass filter that removes low-frequency signals and retains only high-frequency signals. The combination of these two filters can blur the image, which is a process that goes from high resolution to low resolution.

c. Colorizing the Russian Empire
   - We used the method from task 2 to generate an image pyramid. The number of layers in the image pyramid is a parameter we can control. We observed that the result deteriorates if the number of layers is too high.
   - When blurring the image using the method from task 2, the color channels do not align. However, the channels align correctly under the same settings when only downsampling the image without blurring.

5. Conclusion

In this experiment, we discovered many interesting results. For the Hybrid Image section, the Gaussian filter produced smoother results compared to the ideal filter. The cutoff frequency is also an important parameter, allowing for better image transitions. In the Image Pyramid section, the use of Gaussian filters for smoothing helped prevent aliasing issues. Both the Gaussian Pyramid and Laplacian Pyramid have their own characteristics; the higher the level of the Gaussian Pyramid, the coarser the image becomes, while the Laplacian Pyramid preserves the residual information of the image. For the Colorizing the Russian Empire section, three images were input as RGB channels, and normalized cross-correlation was applied, resulting in more accurate image colors.

6. Work Assignment Plan between Team Members

a. 蔡濟謙
    - Do experiments
    - Report: Introduction, Experimental Result, Discussion and Conclusion
b. 鄭博涵
    - Code implementation
    - Report: Implementation, Discussion and Conclusion
c. 陳奎元
    - Do experiments
    - Report: Experimental Result, Discussion and Conclusion

7. Command Line

a. Hybrid Image
    - python3 task1.py [OPTION]
        1. --data_choose: pairs of images
        2. --low_sigma
        3. --high_sigma
        4. --filter_type: gaussian or ideal
        5. --result_name: output filename and path
b. Image Pyramid
    - python3 task2.py [OPTION]
        1. --data_choose: images
        2. --num_layers
        3. --filter_size
        4. --filter_sigma
c. Colorizing the Russian Empire
    - python3 task3.py [OPTION]
        1. --data_choose: images
    - Due to e3 file size limit, We can't upload this file
        1. './data/task3_colorizing/emir.tif'
        2. './data/task3_colorizing/icon.tif'
        3. './data/task3_colorizing/lady.tif'
        4. './data/task3_colorizing/melons.tif'
        5. './data/task3_colorizing/onion_church.tif'
        6. './data/task3_colorizing/three_generations.tif'

8. Output Explanation

output/

- task1_result/
  - Hybrid Image results
- task2_result/
  - Image Pyramid results
- task3_result/
  - Colorizing the Russian Empire results