

# Computer Vision Homework 2 - Filtering and Frequency

Group 15 - 313553014 廖怡誠、313553013 李品妤、312553007 林芷萱

## 1. Introduction

In Assignment 2, we must implement three filtering and frequency-related image processing methods: hybrid image, image pyramid, and colorizing the Russian Empire. The hybrid image combines two images, allowing viewers to perceive different visuals based on viewing distance. It relies on spatial frequencies, embedding one image in high frequencies (details) and the other in low frequencies (broad shapes). Up close, the high-frequency image dominates, while the low-frequency image appears from a distance. To explore how different cutoff frequencies affected the results, we tested various values in our experiment. The image pyramid is progressively reducing the resolution of an image through Gaussian blurring and down-sampling, creating a series of increasingly smaller images. Additionally, we implemented the Laplacian pyramid, which highlights the edges and finer details at each level by subtracting adjacent Gaussian pyramid levels. This allowed us to analyze the differences in image details across multiple scales, and colorizing the Russian Empire involves using red, green, and blue filters to record three exposures of each scene on a glass plate. The algorithm extracts the three color channel images, overlaps them, and aligns them to form a single RGB color image. By implementing this assignment, we can gain a thorough understanding of how to manipulate images between the pixel domain and the frequency domain, as well as how to use filters to extract the necessary information.

## 2. Implementation Procedure

In this section, we will sequentially introduce hybrid images, image pyramids, and colorizing the Russian Empire, along with our implementation methods.

### 2.1. Hybrid Image

A hybrid image is created by combining the low-pass filtered version of one image with the high-pass filtered version of another image. In this section, we will follow these steps to generate the hybrid image.

#### (1) Create and access file directories, read files, and perform data preprocessing.

In this experiment, the results will be saved in the '1\_result' folder. Therefore, we first check if the folder exists, and if it does not, we create it. Then, we sequentially read the images from the 'task1and2\_hybrid\_pyramid' folder. Since the dimensions of the images

'6\_makeup\_after.jpg' and '6\_makeup\_before.jpg' do not match, we will crop a portion of these images first.

## (2) Multiply the input image by $(-1)^{x+y}$ to center the transform.

In digital image processing, the Fourier transform naturally places low frequencies at the corners of the frequency spectrum. By multiplying the image by  $(-1)^{x+y}$ , these crucial low frequencies are shifted to the center of the frequency domain, making them more accessible for analysis. This centering simplifies filtering operations, enhancing the effectiveness of various image processing techniques. Therefore, we start by normalizing the image, dividing each pixel value by 255. After that, to center the transform, we multiply the entire image by  $(-1)^{x+y}$ , preparing it for the next steps.

```
21 def convolution(img, H):
22     h, w, c = img.shape
23     result = np.zeros((h,w,c))
24
25     for c_tmp in range(0, c):
26         img_tmp = img[:, :, c_tmp].copy() / 255
27         # step 1
28         for i in range(0, h):
29             for j in range(0, w):
30                 img_tmp[i, j] = img_tmp[i, j] * ((-1) ** (i + j))
```

## (3) Perform the Fourier transformation of the input image, resulting in $F(u, v)$ .

The Fourier transform converts an image from the spatial domain into the frequency domain, producing  $F(u, v)$ , which captures the image's frequency components. Here,  $u$  and  $v$  represent the horizontal and vertical frequencies, respectively. This transformation is crucial for filtering operations, as most filters operate by adjusting these frequency components, allowing for more precise image manipulation. In this part, we call the Fourier Transformation function to apply the Fourier transformation (via `np.fft.fft2()`) to the input image.

```
31 # step 2
32 F = Fourier_transformation(img_tmp)

43 def Fourier_transformation(img):
44     return np.fft.fft2(img)
```

## (4) Multiply $F(u, v)$ by the filter function $H(u, v)$ .

In the frequency domain, filtering is done by multiplying the image's Fourier transform,  $F(u, v)$ , with a filter function,  $H(u, v)$ . This filter is carefully designed to either enhance or suppress specific frequencies. For instance, a low-pass filter allows the low frequencies, which represent smoother image features, to pass through while blocking the high frequencies associated with finer details and noise. In the following code screenshot,  $F$  represents the Fourier transform of the image, while  $H$  denotes various filters, including an ideal low-pass filter, ideal high-pass filter, Gaussian low-pass filter, and Gaussian high-pass filter.

```

33                                     # step 3
34                                 tmp = F * H

```

#### (5) Apply the inverse Fourier transformation to the result from step 4.

Once filtering is complete, the next step is to bring the image back to the spatial domain by applying the inverse Fourier transform. This process converts the filtered frequency components back into a spatial image, revealing the modified version of the original image after the filter has been applied. The code screenshot for this section is included with the next step in the following part

#### (6) Extract the real part of the result from step 5.

The inverse Fourier transform can produce complex values due to the nature of the transformation, even if the original image was purely real. In most practical image processing tasks, the imaginary part is typically very small and can be safely discarded. Therefore, we usually focus on the real part to recover the spatial image. In this step, we invoke the Inverse Fourier Transformation function to perform the inverse Fourier transform on the output from step 4 (via `np.fft.ifft2`). After applying the transform, we extract the real component by calling `.real` on the result.

```

35                                     # step 4 & 5
36                                 result[:, :, c tmp] = inverse Fourier transformation(tmp).real

46                                 def inverse_Fourier_transformation(img):
47                                     return np.fft.ifft2(img)

```

#### (7) Multiply the result from step 6 by $(-1)^{x+y}$ .

This final multiplication reverses the centering applied in step 2, shifting the frequency components back to their original positions. This step ensures that the filtered image is correctly aligned in the spatial domain. Therefore, we multiply the result from the previous step by  $(-1)^{x+y}$  to shift the frequency components back. After that, we ensure proper spatial alignment of the filtered image, completing the final step of the process.

```

37                                     # step 6
38                                 for i in range(0, h):
39                                     for j in range(0, w):
40                                         result[i, j, c_tmp] = result[i, j, c_tmp] * ((-1) ** (i + j))

```

### 2.1.1. Ideal Low-Pass Filter

An ideal low-pass filter allows only the low-frequency components of an image or signal to pass through, while it completely blocks the high-frequency ones. In the frequency domain, it zeros out all frequencies beyond a defined cutoff point. This type of filter is commonly used

to reduce noise and smooth images. However, it can also blur the image by stripping away high-frequency details, which are often responsible for fine textures and sharp edges. The following is the formula for an ideal low-pass filter.

$$H(u, v) = \begin{cases} 1 & \text{If } D(u, v) \leq D_0 \\ 0 & \text{If } D(u, v) > D_0 \end{cases}$$

$$D(u, v) = (u^2 + v^2)^{\frac{1}{2}}, D_0: \text{cutoff frequency}$$

### 2.1.2. Ideal High-Pass Filter

An Ideal High-Pass Filter allows only high-frequency components of an image or signal to pass through, while blocking low-frequency components. In the frequency domain, it sets values below a certain cutoff frequency to zero, enhancing edges and fine details but often introducing noise.

$$H(u, v) = \begin{cases} 0 & \text{If } D(u, v) \leq D_0 \\ 1 & \text{If } D(u, v) > D_0 \end{cases}$$

$$D(u, v) = (u^2 + v^2)^{\frac{1}{2}}, D_0: \text{cutoff frequency}$$

Below is the code for the ideal filter. This code creates an ideal filter for image processing in the frequency domain, either a low-pass or high-pass filter, depending on the mode. The filter is defined on a grid with dimensions h and w, representing the image size. The center of this grid (x0 and y0) is calculated, and a matrix H is initialized to represent the filter. For a low-pass filter, the matrix is initially filled with zeros, while for a high-pass filter, it's filled with ones.

The code then determines a circular region around the center with a radius equal to the cut-off frequency (cf). Using the equation of a circle, it calculates the boundaries of this circular area for each column of pixels. Within this region, the matrix values are set to 1 for a low-pass filter or 0 for a high-pass filter. Additionally, the max() and min() functions are used to ensure that the calculations of y\_min and y\_max remain within valid bounds. Finally, the matrix H is returned, representing the frequency filter that can be applied to an image via Fourier Transform multiplication to either allow or block certain frequencies based on the filter type.

```

49 def ideal_filter(h, w, cf, mode):
50     x0, y0 = w // 2, h // 2
51     tmp, H = (1, np.zeros((h, w))) if mode == 'low-pass' else (0, np.ones((h, w)))
52
53     for x in range(x0 - cf, x0 + cf):
54         y_min = max(int(y0 - math.sqrt(cf ** 2 - (x - x0) ** 2)), 0)
55         y_max = min(int(y0 + math.sqrt(cf ** 2 - (x - x0) ** 2)), h)
56         for y in range(y_min, y_max):
57             H[y, x] = tmp
58
59     return H

```

### 2.1.3. Gaussian Low-Pass Filter

A Gaussian low-pass filter is a type of filter used in image processing and signal processing to reduce high-frequency noise or details. It works by applying a Gaussian function, which gives more weight to central pixels (or data points) and gradually less to distant ones. This results in smoothing the image or signal, allowing only the low-frequency components to pass through while attenuating the high-frequency components, leading to a blurred or softened output. In addition, Gaussian filters are preferred due to their smooth, non-sharp transitions and minimal distortion.

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

$$D(u, v) = (u^2 + v^2)^{\frac{1}{2}}, D_0: \text{cutoff frequency}$$

### 2.1.4. Gaussian Low-Pass Filter

A Gaussian high-pass filter is used in image processing to remove low-frequency components (like smooth variations or background noise) while retaining high-frequency details (such as edges or fine textures). The filter is based on the Gaussian function, where the center of the frequency domain is attenuated (low frequencies), and values increase as moving further from the center (high frequencies). This filter creates a smooth transition between the filtered and unfiltered regions, reducing artifacts compared to an ideal high-pass filter, which has a sharp cutoff. It's often applied in the frequency domain using the Fourier Transform.

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

$$D(u, v) = (u^2 + v^2)^{\frac{1}{2}}, D_0: \text{cutoff frequency}$$

Below is the code for the Gaussian filter. This code creates a Gaussian filter for image processing in the frequency domain, either a low-pass or high-pass filter, depending on the mode. The filter is defined on a 2D grid of dimensions h (height) and w (width), representing the image size. The center of this grid is calculated as (x0, y0), and a matrix H is initialized to store the filter values. For a low-pass filter, the matrix is filled with values starting from 0, while for a high-pass filter, it starts with values of 1.

For a low-pass filter, the Gaussian values are added to the matrix H, generating a smooth transition that allows low frequencies to pass. For a high-pass filter, the Gaussian values are subtracted, blocking low frequencies and emphasizing high frequencies. Finally, the matrix H is returned as the filter, which can be applied to an image in the frequency domain (using Fourier Transform techniques) to selectively allow or block certain frequency components.

```

61 def Gaussian_filter(h, w, cf, mode):
62     x0, y0 = w // 2, h // 2
63     tmp, H = (1, np.zeros((h, w))) if mode == 'low-pass' else (-1, np.ones((h, w)))
64
65     for x in range(w):
66         for y in range(h):
67             H[y, x] += (math.exp(-1 * ((x - x0) ** 2 + (y - y0) ** 2) / (2 * cf ** 2))) * tmp
68
69     return H

```

## 2.2. Solving for the Intrinsic Parameters

In this section, we first apply a 5x5 Gaussian filter to the image before down-sampling. The down-sampling is done by taking every second pixel in both dimensions. Additionally, we generate the magnitude spectrum at each level of the pyramid. We also implement the Laplacian pyramid, which is constructed using the Gaussian pyramid levels  $G(i)$  and  $G(i+1)$ .

### 2.2.1. Gaussian Kernel Generation

We first generate a set of evenly spaced points along one axis centered around zero. The size of this grid is defined by the window size  $N$ , and the points range from  $-\frac{N-1}{2}$  to  $\frac{N-1}{2}$ , ensuring symmetry around the center.

$$ax = \left[ -\frac{N-1}{2}, \dots, \frac{N-1}{2} \right]$$

For each point in the 1D grid, we compute the Gaussian value using the formula:

$$G(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

To create a 2D kernel from the 1D Gaussian distribution, we compute the outer product of the 1D Gaussian array with itself. This outer product generates a matrix where each element is the product of two 1D Gaussian values, resulting in a 2D Gaussian distribution.

$$K(i, j) = G(x_i) \cdot G(x_j)$$

Finally, we normalize the kernel by dividing it by the sum of all its elements.

$$K_{norm}(i, j) = \frac{K(i, j)}{\sum_{i,j} K(i, j)}$$

```

31 def gaussian_kernel(self):
32
33     ax = np.linspace(-(self.window - 1) / 2., (self.window - 1) / 2., self.window)
34     gauss = np.exp(-0.5 * np.square(ax) / np.square(self.sigma))
35     kernel = np.outer(gauss, gauss)
36     return kernel / np.sum(kernel)

```

### 2.2.2. Image Reduction

To reduce the size of an image, we apply a Gaussian blur followed by down-sampling without padding. The first step is to convolve the image with a Gaussian kernel. The Gaussian kernel is generated using the method 2.2.1. After the image has been convolved with the Gaussian kernel, we reduce its size by down-sampling. Down-sampling involves selecting every second pixel in both dimensions.

$$g_l(i, j) = \sum_{m=-2}^2 \sum_{n=-2}^2 w(m, n) \cdot g_{l-1}(2i + m, 2j + n)$$

```
39     def reduce(self, gray_img):
40
41         offset = self.window // 2
42         gwindow = self.gaussian_kernel()
43         height, width = gray_img.shape
44         convolved_img = np.zeros((height, width))
45
46         for i in range(offset, height - offset):
47             for j in range(offset, width - offset):
48                 patch = gray_img[i - offset:i + offset + 1, j - offset:j + offset + 1]
49                 convolved_img[i, j] = np.sum(patch * gwindow)
50
51         return convolved_img[offset:height - offset:2, offset:width - offset:2]
```

### 2.2.3. Magnitude Spectrum Generation

To analyze the frequency content of an image, we compute its 2D Fast Fourier Transform (FFT). After computing the FFT, the zero-frequency component is located at the corners of the spectrum. To enhance visualization, we use a shift operation to move the low frequencies to the center of the frequency domain. The magnitude spectrum  $S$  is calculated as the logarithm of the absolute value of the Fourier Transform, scaled to enhance visibility.

```
55     def magnitude_spectrum(self, img):
56         f = np.fft.fft2(img)
57         fshift = np.fft.fftshift(f)
58         spectrum = 20 * np.log(np.abs(fshift) + 1)
59         return spectrum
```

### 2.2.4. Laplacian Pyramid

The Laplacian pyramid is built using the Gaussian pyramid, which is generated by progressively down-sampling the input image. Starting with the original image, we iteratively apply a reduction operation to create the next image in the pyramid. The Laplacian pyramid is created by subtracting the up-sampled version of the next level from the current level.

$$L_l = G_l - \text{Expand}(G_{l+1})$$

```

79 class LaplacianPyramid(GaussianPyramid):
80
81     def run(self):
82         gaussian_pyramid = [self.image]
83         current_image = self.image
84
85         for _ in range(1, self.level):
86             next_image = self.reduce(current_image)
87             gaussian_pyramid.append(next_image)
88             current_image = next_image
89
90         laplacian_pyramid = []
91         for i in range(self.level - 1):
92             next_image_upsampled = self.expand(gaussian_pyramid[i + 1], gaussian_pyramid[i].shape)
93             laplacian = gaussian_pyramid[i] - next_image_upsampled
94             laplacian_pyramid.append(laplacian)

```

## 2.3. Colorizing the Russian Empire

In section 2.3, we have a grayscale image containing three identical contents, each corresponding to the B, G, and R channels. We need to split them apart, align them correctly, and stack them to generate a color image. The implementation details and steps are described as follows:

### 2.3.1. Splitting the Image into individual RGB Channel

First, the original grayscale image must be split into three BGR channel components (red, green, and blue). Basic image manipulation techniques are used to split the image into three regions based on height/3, as shown in **Figure 1**.



Fig 1. BGR channel image after splitting.

```

8 def split_image_to_BGR(img):
9     """ Split the original image into three images each of RGB"""
10    h = img.shape[0]
11    h_unit = h // 3
12    # B, G, R
13    return [img[: h_unit], img[h_unit : 2 * h_unit], img[2 * h_unit : 3 * h_unit]]

```

### 2.3.2. Cropping images

As seen in the three BGR channel images in **Figure 1** and the original grayscale image, there are irregular black borders and often off-center, which may seriously impede alignment



efficiency. To address this, we remove a certain percentage (default 10%) of pixels from each side of each BGR image, and we can obtain clear images, as shown in **Figure 2**.



**Fig 2. BGR channel image after removing block border.**

```
15 def crop_img(org_img, divided_img, crop_ratio= 0.1):
16     """ Crop the image to reduce noise or some unwanted black borders"""
17     h, w = org_img.shape[:2]
18     h_unit, w_unit = h // 3, w
19     crop_h_unit, crop_w_unit = int(h_unit * crop_ratio), int(w_unit * crop_ratio)
20
21     cropped_img = []
22     for img in divided_img:
23         cropped_img.append(
24             img[crop_h_unit : h_unit - crop_h_unit, crop_w_unit : w_unit - crop_w_unit]
25         )
26     return cropped_img
```

### 2.3.3. Aligning Images

This is the most critical and complex part, where we need to align the BGR images in **Figure 2** obtained earlier precisely. Specifically, we want to ensure that the pixel positions of the three-channel images are as close to each other as possible. To achieve this, we choose the Channel B image as the basement and move the Channel G and Channel R image's pixel positions accordingly. We then use Normalized Cross-Correlation (NCC) and Sum of Squared Differences (SSD) to calculate the loss score. The images are translated sequentially within a predefined offset range of  $[-\text{max\_shift}, \text{max\_shift}]$ . Finally, we adjust the image positions using the offset that yields the best loss score to align with the Channel B image. Since the Channel G and Channel R images are aligned based on the Channel B image, we ultimately obtain a well-aligned image, as shown in **Figure 3**.



**Fig 3. Well-aligned BGR image**

The first method we use to evaluate whether two-channel images are aligned is called SSD (Sum of Squared Differences), and the formula is

$$SSD = \sum (img1(u,v) - img2(u,v))^2.$$

where  $img1(u,v)$  is the pixel of image 1, and  $img2(u,v)$  is the pixel of **image 2**. The formula shows that the Sum of Squared Differences (SSD) calculates the difference between each corresponding pixel and then sums the squares of these differences. A smaller SSD indicates that the two images are better aligned, while a larger SSD signifies that the images are less aligned. In implementation, we must change the image type from integer to floating point. Otherwise, there will be insufficient accuracy in calculating the square part.

The second method is called NCC (Normalized Cross-Correlation), and the formula is

$$NCC = \frac{\sum (img1(u,v) - \overline{img1})(img2(u,v) - \overline{img2})}{\sqrt{\sum (img1(u,v) - \overline{img1})^2 \sum (img2(u,v) - \overline{img2})^2}}$$

where  $img1(u,v)$  is the pixel of image 1,  $img2(u,v)$  is the pixel of image 2,  $\overline{img1}$  is the average of all pixels in image 1, and  $\overline{img2}$  is the average of all pixels in image 2. The formula shows that the NCC measures the similarity between two images by comparing the intensity relationships of corresponding pixels, with normalization applied during the comparison process. The advantage of this normalization is that it eliminates overall differences in brightness and contrast between the two images, focusing only on whether their structure or texture is similar.

```

52 > def align_channels(base_channel, offset_channel, loss= "ncc", max_shift= 25):
53     """ Align B, G, R channel """
54     best_shift = (0, 0)
55     best_score = None
56     # run all shift
57 >     for x_shift in range(-max_shift, max_shift + 1):
58 >         for y_shift in range(-max_shift, max_shift + 1):
59             # shift x and y
60             shifted_channel = np.roll(offset_channel, x_shift, axis= 1)
61             shifted_channel = np.roll(shifted_channel, y_shift, axis= 0)
62             # calculate ncc
63 >             if loss == "ncc":
64                 loss_score = ncc(base_channel, shifted_channel)
65             # calculate ssd
66 >             elif loss == "ssd":
67                 loss_score = ssd(base_channel, shifted_channel)
68             # record best
69 >             if best_score is None or loss_score > best_score:
70                 best_score = loss_score
71                 best_shift = (x_shift, y_shift)
72             # print(max_ncc, best_shift)
73     return best_shift

42 def ssd(img1, img2):
43     """ Sum of Squared Differences (SSD) """
44
45     # difference of two images
46     difference = np.array(img1, dtype=np.float64) - np.array(img2, dtype=np.float64)
47     squared_difference = difference ** 2
48     ssd_value = np.sum(squared_difference)
49
50     return -ssd_value

```

```

28 def ncc(img1, img2):
29     """ normalize cross correlation """
30
31     mean1 = np.mean(img1)
32     mean2 = np.mean(img2)
33
34     numerator = np.sum((img1 - mean1) * (img2 - mean2))
35     denominator = np.sqrt(np.sum((img1 - mean1) ** 2) * np.sum((img2 - mean2) ** 2))
36
37     if denominator == 0:
38         return 0
39
40     return numerator / denominator

```

In summary, this is our implementation of colorizing the Russian Empire with the code below. It demonstrates the complete process, including the previously mentioned steps of splitting, removing black borders, and aligning and merging the three BGR images. Additionally, we would like to highlight one implementation detail: when reading the TIF image files, the data type corresponds to float64, resulting in large pixel values. Therefore, it's necessary to normalize the pixel values by dividing them by 65,535 to enable proper processing.

```

75 def colorizing(args, img):
76
77     # pre-processing image
78     divided_img = split_image_to_BGR(img)
79     cropped_B, cropped_G, cropped_R = crop_img(img, divided_img, args.crop_ratio)
80     cropped_h, cropped_w = cropped_B.shape[:2]
81
82     # downsample img to reduce computation cost, then find best shift in downsampled img.
83     down_scale = args.down_scale
84     down_B = cv2.resize(cropped_B, (cropped_h // down_scale, cropped_w // down_scale))
85     down_G = cv2.resize(cropped_G, (cropped_h // down_scale, cropped_w // down_scale))
86     down_R = cv2.resize(cropped_R, (cropped_h // down_scale, cropped_w // down_scale))
87     green_shift = align_channels(down_B, down_G, "ssd", args.shift)
88     red_shift = align_channels(down_B, down_R, "ssd", args.shift)
89
90     print(f"green shift(x, y) {green_shift[0] * down_scale}, {green_shift[1] * down_scale}")
91     print(f"red shift(x, y) {red_shift[0] * down_scale}, {red_shift[1] * down_scale}")
92     # align G, R image with down scale multiply shift of green and red.
93     aligned_G = np.roll(cropped_G, green_shift[0] * down_scale, axis=1)
94     aligned_G = np.roll(aligned_G, green_shift[1] * down_scale, axis=0)
95     aligned_R = np.roll(cropped_R, red_shift[0] * down_scale, axis=1)
96     aligned_R = np.roll(aligned_R, red_shift[1] * down_scale, axis=0)
97
98     # concat RGB channel to RGB image.
99     return np.dstack([aligned_R, aligned_G, cropped_B])

```

```

122         # because type of tif file is u64
123         if splitted_file_name[1] == ".tif":
124             img = img.astype(np.float64)
125             img /= 65535.0

```

### 3. Experimental Result

In this section, we will show our experimental results about hybrid images, image pyramids, and colorizing the Russian Empire.

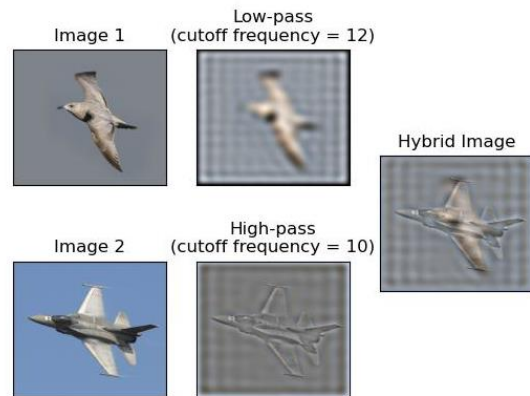
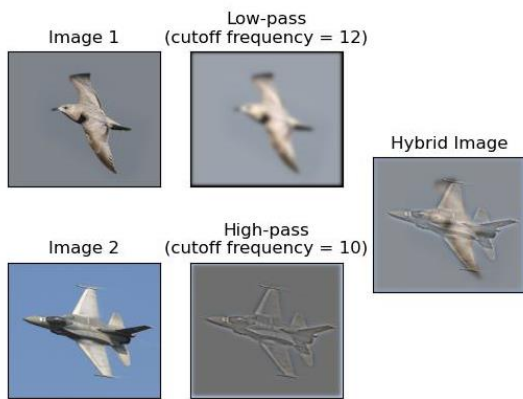
#### 3.1. Hybrid Image

A hybrid image can be composed of the low-pass and high-pass filtered versions with different cutoff frequencies. Therefore, in the Hybrid Image experiment, we attempted to set various cutoff values and observed how these values influenced the results. Below are the

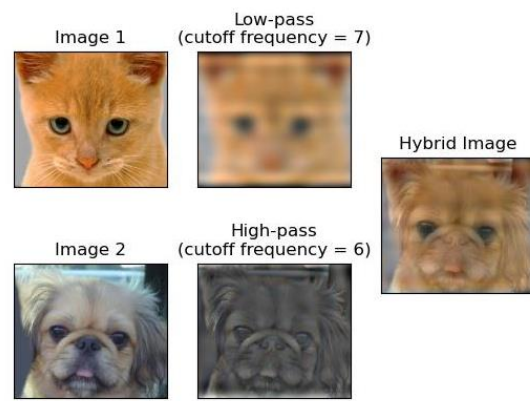
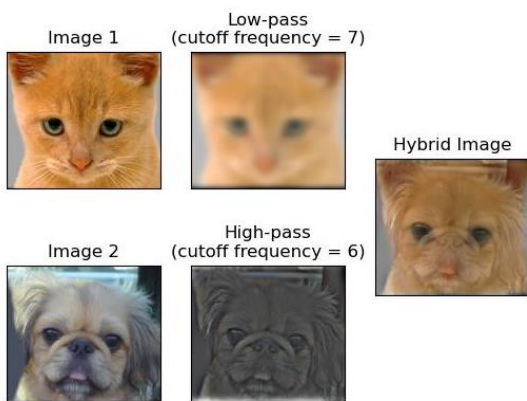
optimal experimental results for each image, with the effects of different cutoff values discussed in the discussion section. In these results, the two images on the left are the original images, the two in the middle are the low-pass and high-pass filtered versions, respectively, and the image on the right is the hybrid image. According to these experimental results, we can easily observe that upon close inspection, one image dominates, typically featuring high spatial frequencies with fine details and sharp edges. As the viewing distance increases, the perception shifts to another image, characterized by low spatial frequencies, representing broader shapes and patterns. Additionally, we found that the Gaussian low-pass and high-pass filtered versions are noticeably smoother than those produced by the ideal filter.

TA's Data	
Gaussian Filter	Ideal Filter
0_Afghan_girl_after.jpg & 0_Afghan_girl_before.jpg	
<div> <div>Image 1</div> </div> <div> <div>Low-pass (cutoff frequency = 9)</div> </div> <div> <div>Image 2</div> </div> <div> <div>High-pass (cutoff frequency = 7)</div> </div> <div> <div>Hybrid Image</div> </div>	<div> <div>Image 1</div> </div> <div> <div>Low-pass (cutoff frequency = 9)</div> </div> <div> <div>Image 2</div> </div> <div> <div>High-pass (cutoff frequency = 7)</div> </div> <div> <div>Hybrid Image</div> </div>
1_bicycle.bmp & 1_motorcycle.bmp	
<div> <div>Image 1</div> </div> <div> <div>Low-pass (cutoff frequency = 6)</div> </div> <div> <div>Image 2</div> </div> <div> <div>High-pass (cutoff frequency = 8)</div> </div> <div> <div>Hybrid Image</div> </div>	<div> <div>Image 1</div> </div> <div> <div>Low-pass (cutoff frequency = 6)</div> </div> <div> <div>Image 2</div> </div> <div> <div>High-pass (cutoff frequency = 8)</div> </div> <div> <div>Hybrid Image</div> </div>

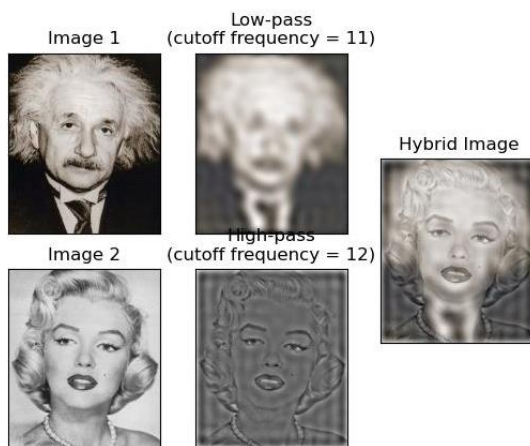
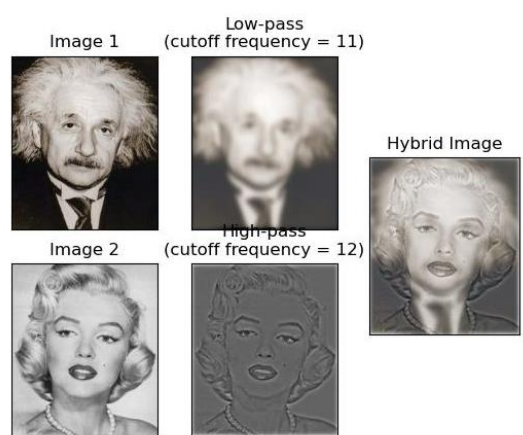
## 2\_bird.bmp & 2\_plane.bmp













## 3\_cat.bmp & 3\_dog.bmp



















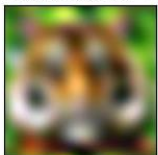

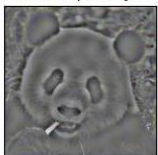

## 4\_einstein.bmp & 4\_marilyn.bmp

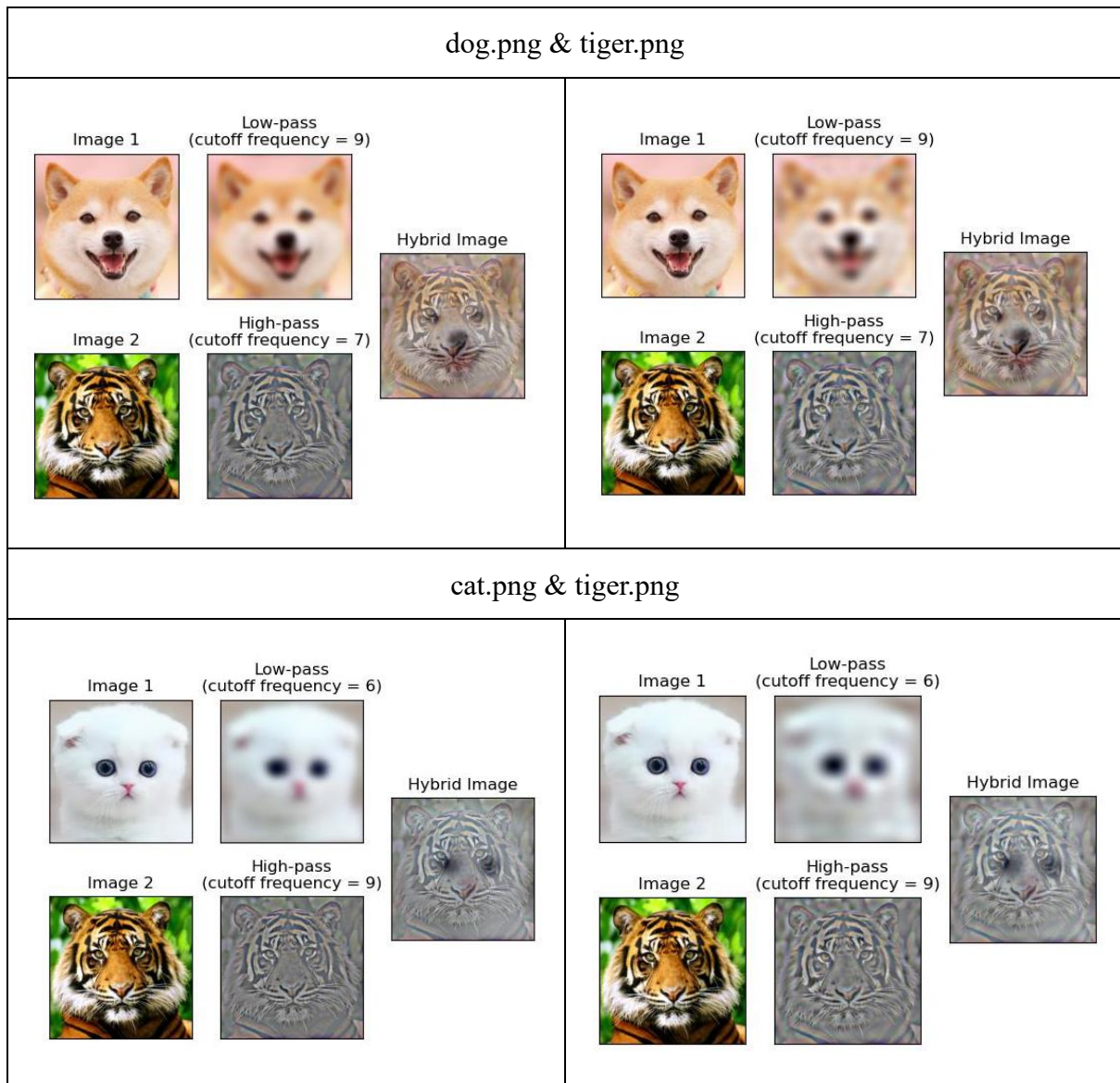




5_fish.bmp & 5_submarine.bmp		
<p>Image 1</p>  <p>Low-pass (cutoff frequency = 9)</p>  <p>Image 2</p>  <p>High-pass (cutoff frequency = 6)</p>  <p>Hybrid Image</p> 	<p>Image 1</p>  <p>Low-pass (cutoff frequency = 9)</p>  <p>Image 2</p>  <p>High-pass (cutoff frequency = 6)</p>  <p>Hybrid Image</p> 	


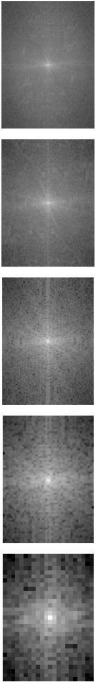

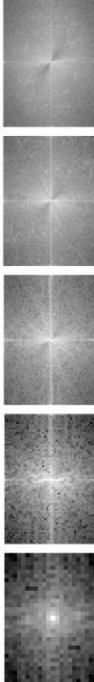

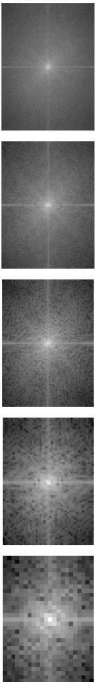

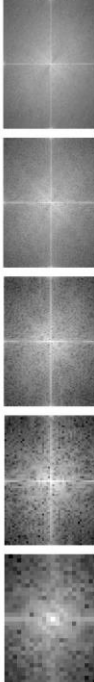
6_makeup_after.jpg & 6_makeup_before.jpg		
<p>Image 1</p>  <p>Low-pass (cutoff frequency = 9)</p>  <p>Image 2</p>  <p>High-pass (cutoff frequency = 6)</p>  <p>Hybrid Image</p> 	<p>Image 1</p>  <p>Low-pass (cutoff frequency = 9)</p>  <p>Image 2</p>  <p>High-pass (cutoff frequency = 6)</p>  <p>Hybrid Image</p> 	

Our Data		
Gaussian Filter	Ideal Filter	
tiger.png & panda.png		
<div><p>Image 1</p></div> <div><p>Low-pass (cutoff frequency = 6)</p></div> <div><p>Image 2</p></div> <div><p>High-pass (cutoff frequency = 8)</p></div> <div><p>Hybrid Image</p></div>	<div><p>Image 1</p></div> <div><p>Low-pass (cutoff frequency = 6)</p></div> <div><p>Image 2</p></div> <div><p>High-pass (cutoff frequency = 8)</p></div> <div><p>Hybrid Image</p></div>	

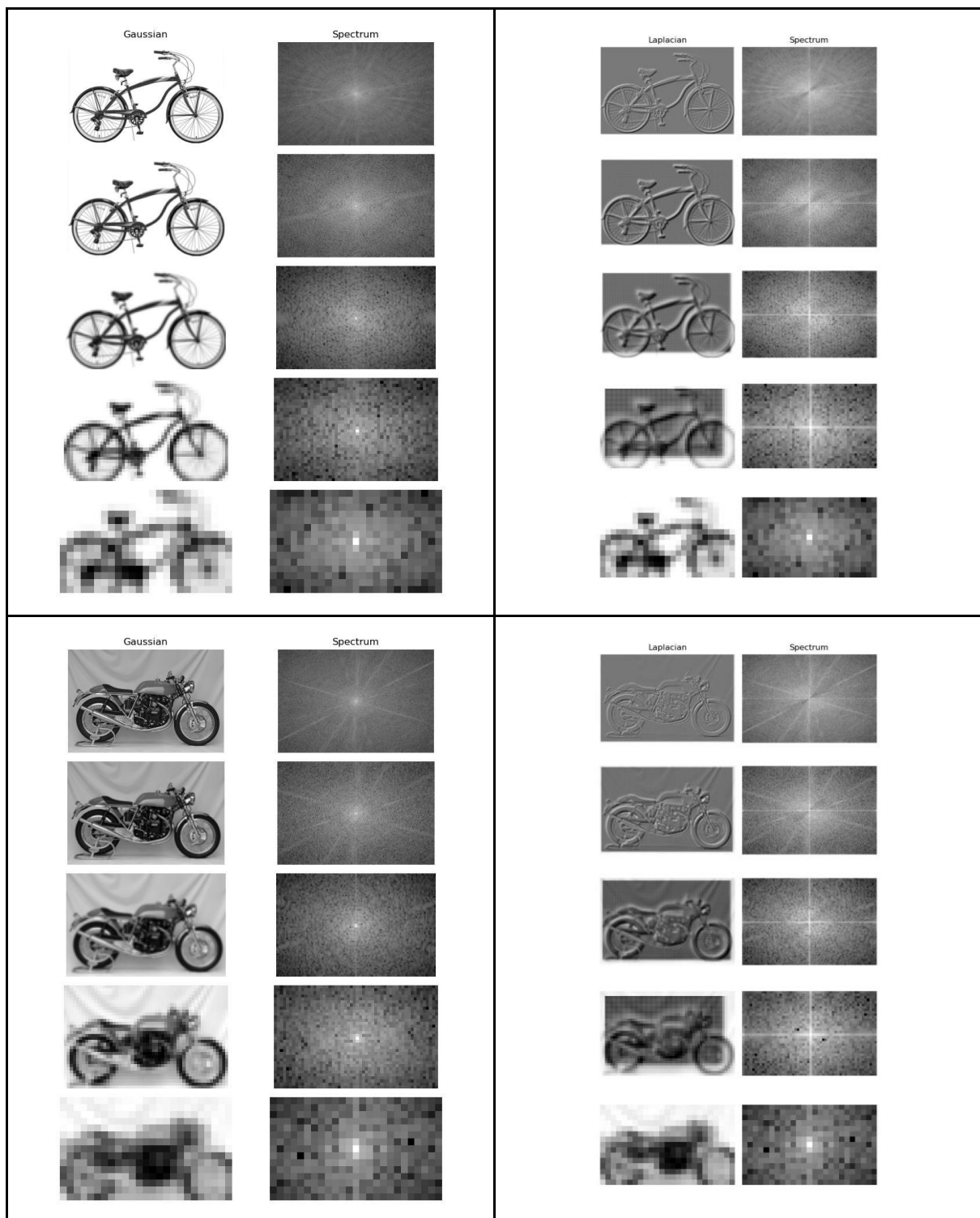


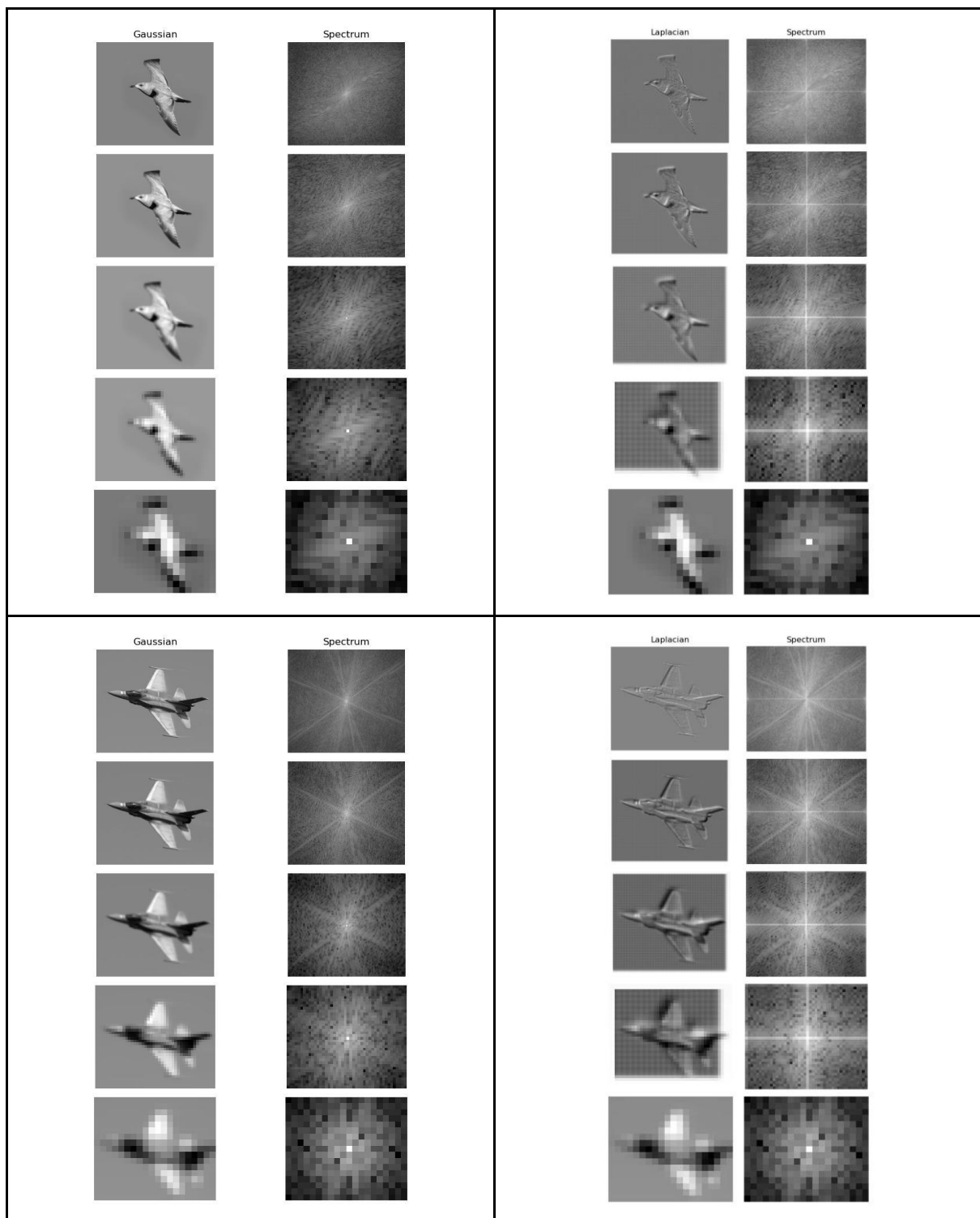
### 3.2. Image Pyramid

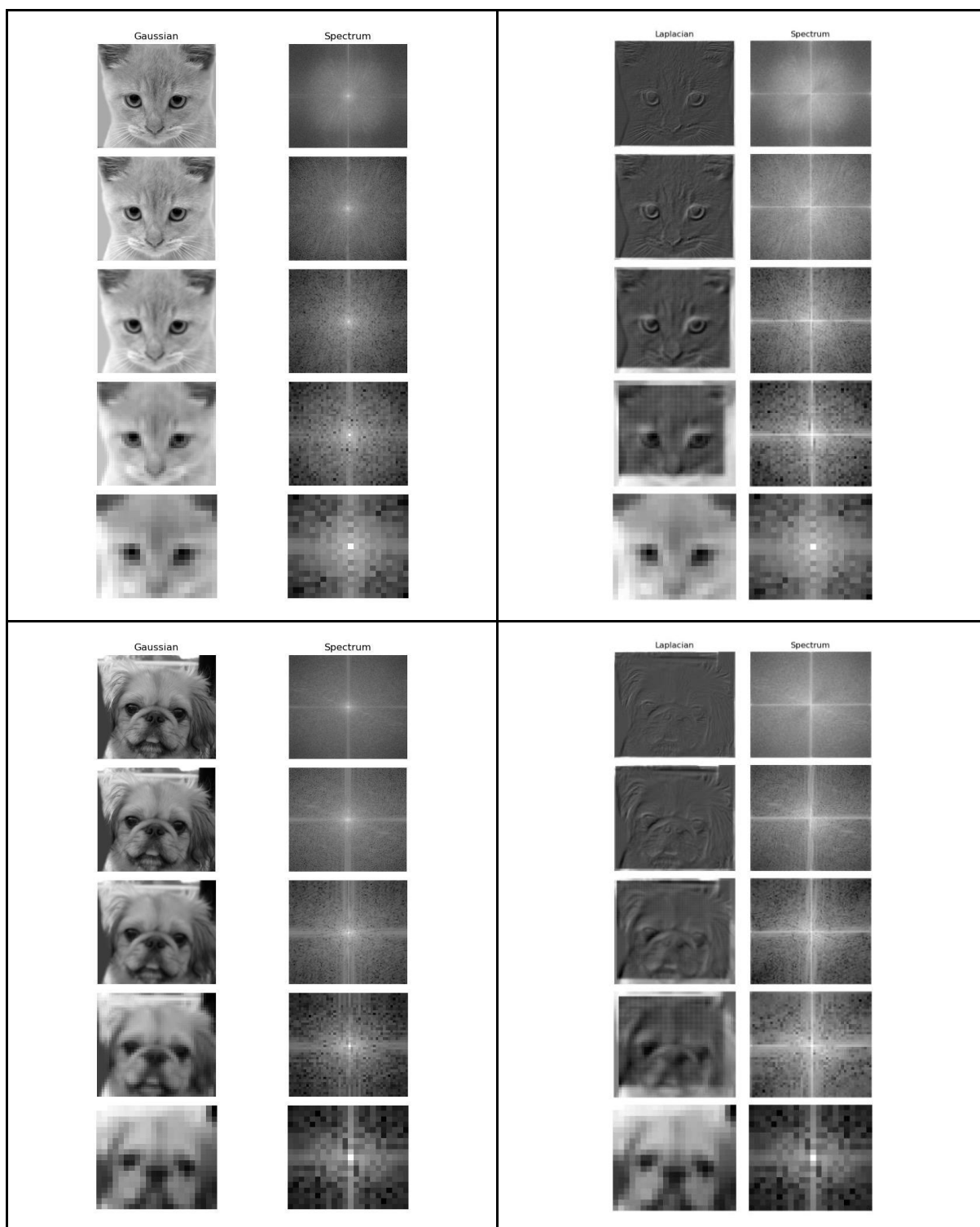
In this experiment, we aim to investigate the differences between Gaussian and Laplacian pyramids by decomposing an image at various levels and analyzing the spatial and frequency domains. The Gaussian Pyramid reduces the image resolution at each level, while the Laplacian Pyramid captures the high-frequency details by subtracting successive levels of the Gaussian Pyramid. Based on the experimental results from the Image Pyramid experiment, we can observe distinct transformations in the visual and frequency domains of the image as it undergoes Gaussian and Laplacian pyramid decompositions. The Gaussian Pyramid progressively smooths the image by reducing high frequencies, leading to blurrier images at each level, as confirmed by the decreasing high-frequency content in the spectra. In contrast, the Laplacian Pyramid captures fine details by isolating high-frequency components, such as edges and textures, with sharper results at each level, which the corresponding spectra also reflect.

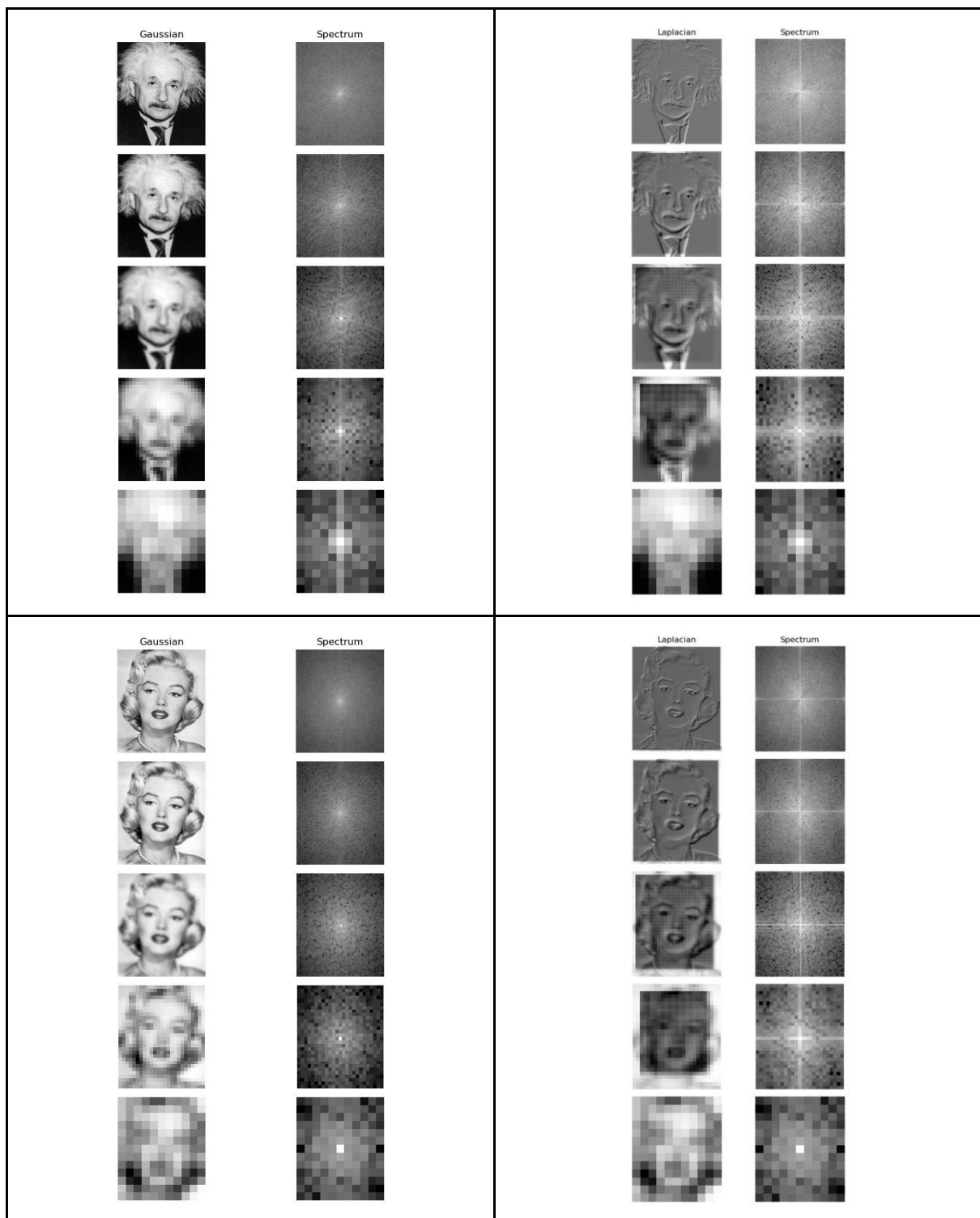
TA's Data			
Gaussian Pyramid		Laplacian Pyramid	
<div><div>Gaussian</div><div>Spectrum</div></div>		<div><div>Laplacian</div><div>Spectrum</div></div>	
<div><div>Gaussian</div><div>Spectrum</div></div>		<div><div>Laplacian</div><div>Spectrum</div></div>	

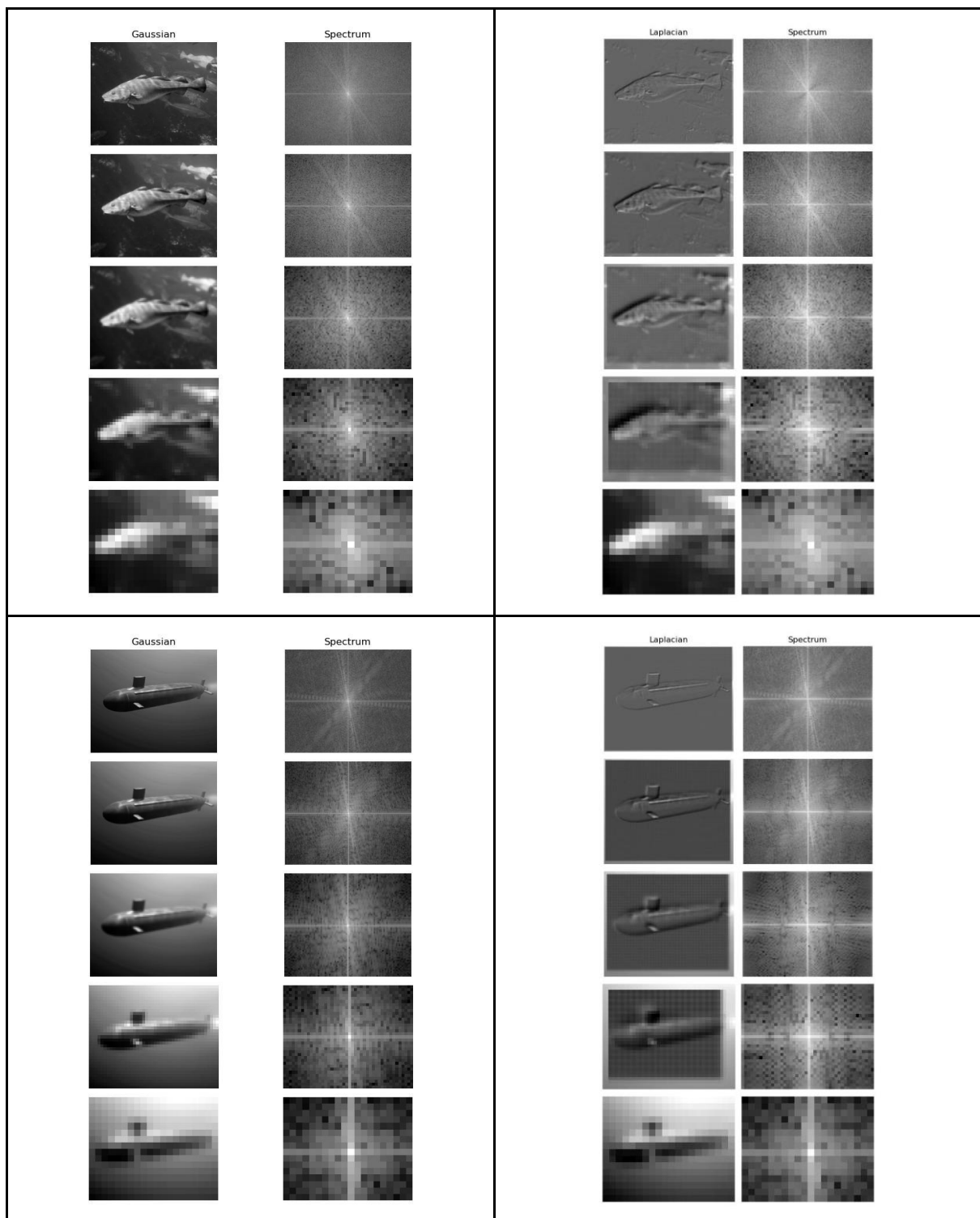




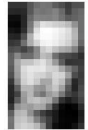




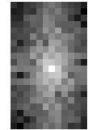
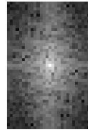
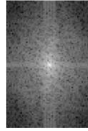
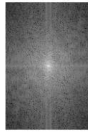
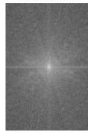




Gaussian



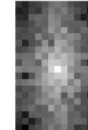
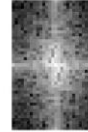
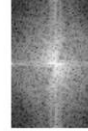
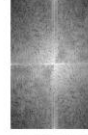
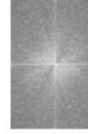
Spectrum



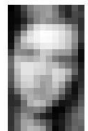
Laplacian



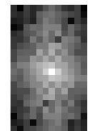
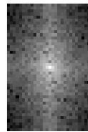
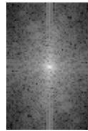
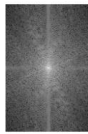
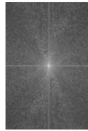
Spectrum



Gaussian



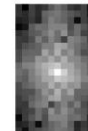
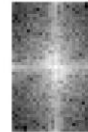
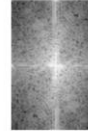
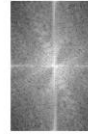
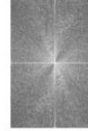
Spectrum


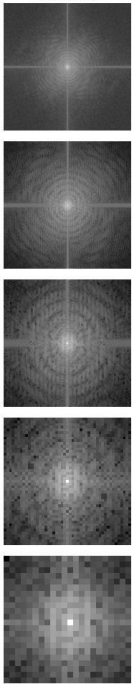

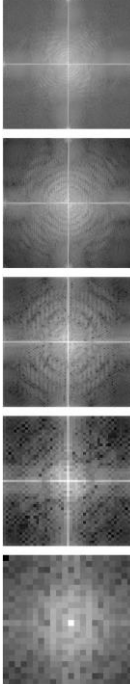

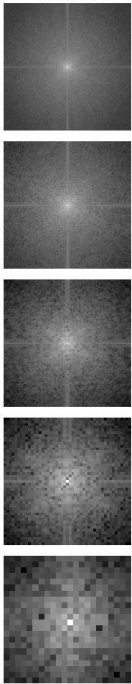

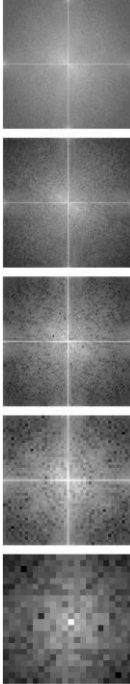


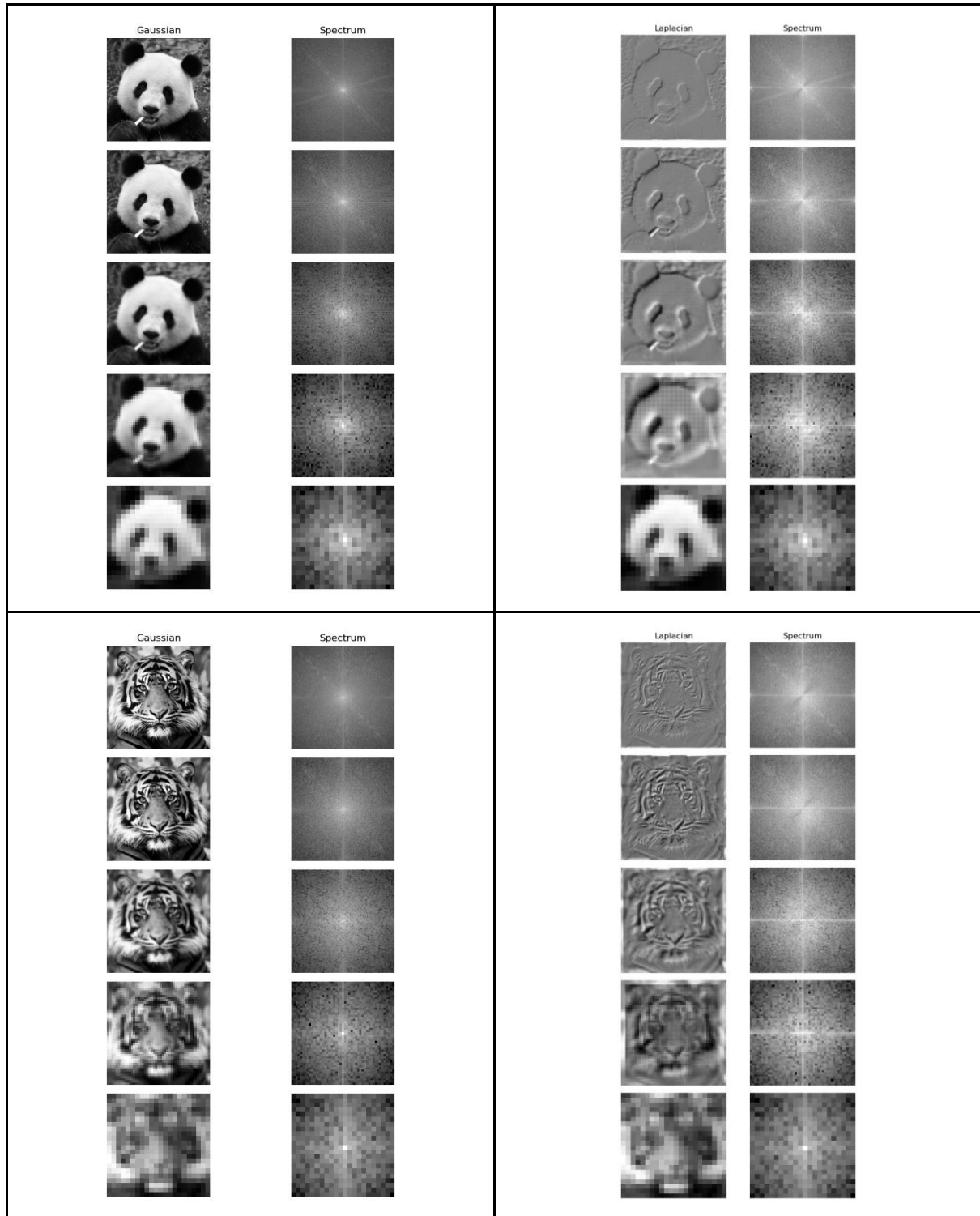
Laplacian



Spectrum



Our Data			
Gaussian Pyramid		Laplacian Pyramid	
<div><div>Gaussian</div><div>Spectrum</div></div>		<div><div>Laplacian</div><div>Spectrum</div></div>	
<div><div>Gaussian</div><div>Spectrum</div></div>		<div><div>Laplacian</div><div>Spectrum</div></div>	


























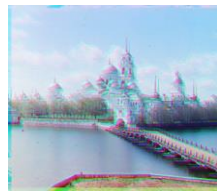



















### 3.3. Coloring the Russian Empire









We used two methods to estimate whether the images aligned correctly and visualized the results to evaluate their effectiveness. The methods we compared are NCC and SSD, analyzing the changes in the images produced under different shift ranges. I used shift ranges of  $[-25, 25]$  and  $[-35, 35]$ , which I referred to as NCC25, NCC35, SSD25, and SSD35, where the names correspond to the method and the shift range used shown in **Table 1**. Overall, the visual results



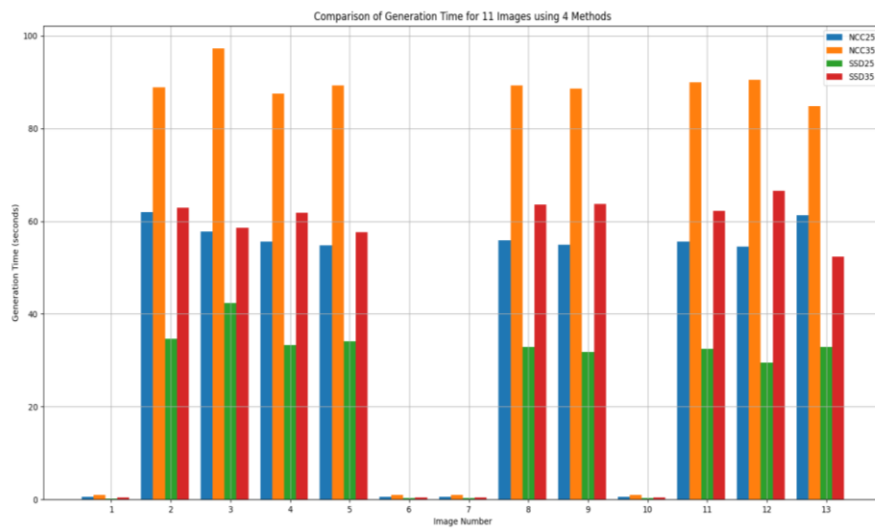
of all the images are quite good, and by observing the shifts in the green and red channel images, we can see that the differences between the four methods are minimal. It is worth noting that in the second row, the emir image produces expected results when the shift range is set to 25 but shows some deviation when it is set to 35. On the other hand, in the fifth row, the melons image shows that the result with a shift range of 25 is slightly more misaligned compared to the result with a shift range of 35. These two cases illustrate that increasing the shift range does not constantly improve the results and can sometimes only increase the image processing time. Alternatively, there may be other influencing factors, which will be discussed in the discussion section. In addition to comparing the quality of the images, we also compared the processing times of the four strategies, as shown in **Figures 4** and **Figures 5**, which display the processing times for all images and small-sized images, respectively. It can be observed that the NCC method is consistently slower than the SSD method, likely because the calculations for NCC are relatively more complex compared to those for SSD. Additionally, a known result is that a broader shift range requires more processing time.

NCC25	NCC35	SSD25	SSD35
			
G (0, 0), R (0,10)	G (0, 0), R (0,10)	G (0, 0), R (0,10)	G (0, 0), R (0,10)
			
G (20, 55), R (50,120)	G (20, 55), R (-175, 115)	G (20, 55), R (50, 120)	G (20, 55), R (-175, 115)
			
G (15, 45), R (20, 105)	G (15, 45), R (20, 105)	G (15, 45), R (20, 105)	G (15, 45), R (20, 105)
			
G (5, 60), R (10, 125)	G (5, 60), R (10, 130)	G (5, 60), R (10, 125)	G (5, 60), R (10, 130)

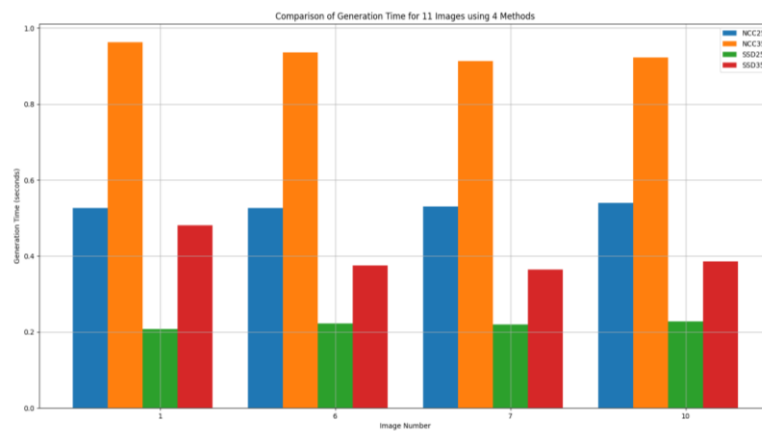
			
G (10, 95), R (10, 125)	G (10, 95), R (10, 175)	G (10, 95), R (10, 125)	G (10, 95), R (10, 175)
			
G (0, -5), R (5, 5)	G (0, -5), R (5, 5)	G (0, -5), R (5, 5)	G (0, -5), R (5, 5)
			
G (0, 0), R (0, 5)	G (0, 0), R (0, 5)	G (0, 0), R (0, 5)	G (0, 0), R (0, 5)
			
G (20, 60), R (30, 125)	G (20, 60), R (30, 125)	G (20, 60), R (30, 125)	G (20, 60), R (30, 125)
			
G (10, 60), R (10, 125)	G (10, 60), R (10, 130)	G (10, 60), R (10, 125)	G (10, 60), R (10, 130)
			
G (5, 5), R (0, 5)	G (5, 5), R (0, 5)	G (5, 5), R (0, 5)	G (5, 5), R (0, 5)
			
G (5, 50), R (25, 100)	G (5, 50), R (25, 100)	G (5, 50), R (25, 100)	G (5, 50), R (25, 100)

			
G (10, 75), R (10, 125)	G (10, 75), R (20, 160)	G (10, 75), R (10, 125)	G (10, 75), R (20, 160)
			
G (0, 60), R (-10, 120)	G (0, 60), R (-10, 120)	G (0, 60), R (-10, 120)	G (0, 60), R (-10, 120)

**Table 1.** The visual result with four methods. G is green shift and R is red shift based on blue channel image.



**Fig 4.** Processing time of all data.

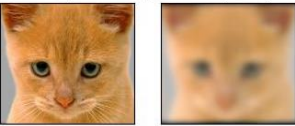











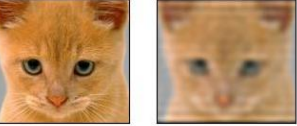
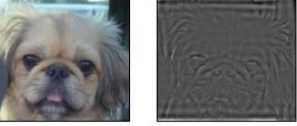



**Fig 5.** Processing time of small-size data.

## 4. Discussion

### 4.1. Hybrid Image

In this section, we experimented with different cutoff values to examine their impact on the results. We tested values ranging from 6 to 13 and compared the resulting differences. Some of our experimental results are shown below, demonstrating and supporting our observations. Here are the results where the low-pass and high-pass filters were set to the same cutoff values, making the comparison easier. The results are presented sequentially for cutoff values of 6, 9, and 13.






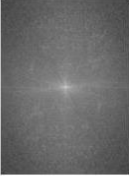
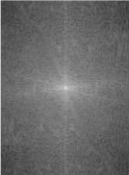
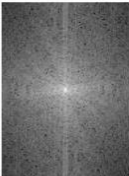
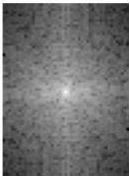
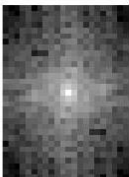





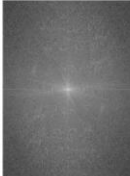
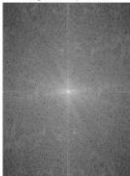
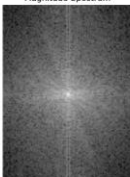
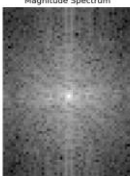
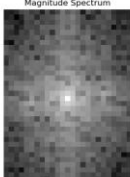
Gaussian Filter	Ideal Filter
<p data-bbox="300 792 584 824">Image 1      Low-pass (cutoff frequency = 6)</p>  <p data-bbox="603 898 711 929">Hybrid Image</p> <p data-bbox="300 987 584 1019">Image 2      High-pass (cutoff frequency = 6)</p>  	<p data-bbox="906 792 1190 824">Image 1      Low-pass (cutoff frequency = 6)</p>  <p data-bbox="1209 898 1318 929">Hybrid Image</p> <p data-bbox="906 987 1190 1019">Image 2      High-pass (cutoff frequency = 6)</p>  
<p data-bbox="300 1218 584 1249">Image 1      Low-pass (cutoff frequency = 9)</p>  <p data-bbox="603 1314 711 1346">Hybrid Image</p> <p data-bbox="300 1404 584 1435">Image 2      High-pass (cutoff frequency = 9)</p>  	<p data-bbox="906 1218 1190 1249">Image 1      Low-pass (cutoff frequency = 9)</p>  <p data-bbox="1209 1314 1318 1346">Hybrid Image</p> <p data-bbox="906 1404 1190 1435">Image 2      High-pass (cutoff frequency = 9)</p>  
<p data-bbox="300 1621 584 1653">Image 1      Low-pass (cutoff frequency = 13)</p>  <p data-bbox="603 1727 711 1758">Hybrid Image</p> <p data-bbox="300 1816 584 1848">Image 2      High-pass (cutoff frequency = 13)</p>  	<p data-bbox="906 1621 1190 1653">Image 1      Low-pass (cutoff frequency = 13)</p>  <p data-bbox="1209 1727 1318 1758">Hybrid Image</p> <p data-bbox="906 1816 1190 1848">Image 2      High-pass (cutoff frequency = 13)</p>  

According to the results shown above, we found that when the cutoff frequency is higher, the low-pass filtered version becomes clearer, while the high-pass filtered version becomes less pronounced. The reason this situation occurs is that, for a low-pass filter, a higher cutoff frequency allows more frequencies to pass through, resulting in a smoother version of the original image while still preserving finer details. For a high-pass filter, a higher cutoff frequency blocks only very low frequencies, so most finer details and textures in the image are preserved. This results in a sharpened image with clear edges and more prominent high-frequency features.

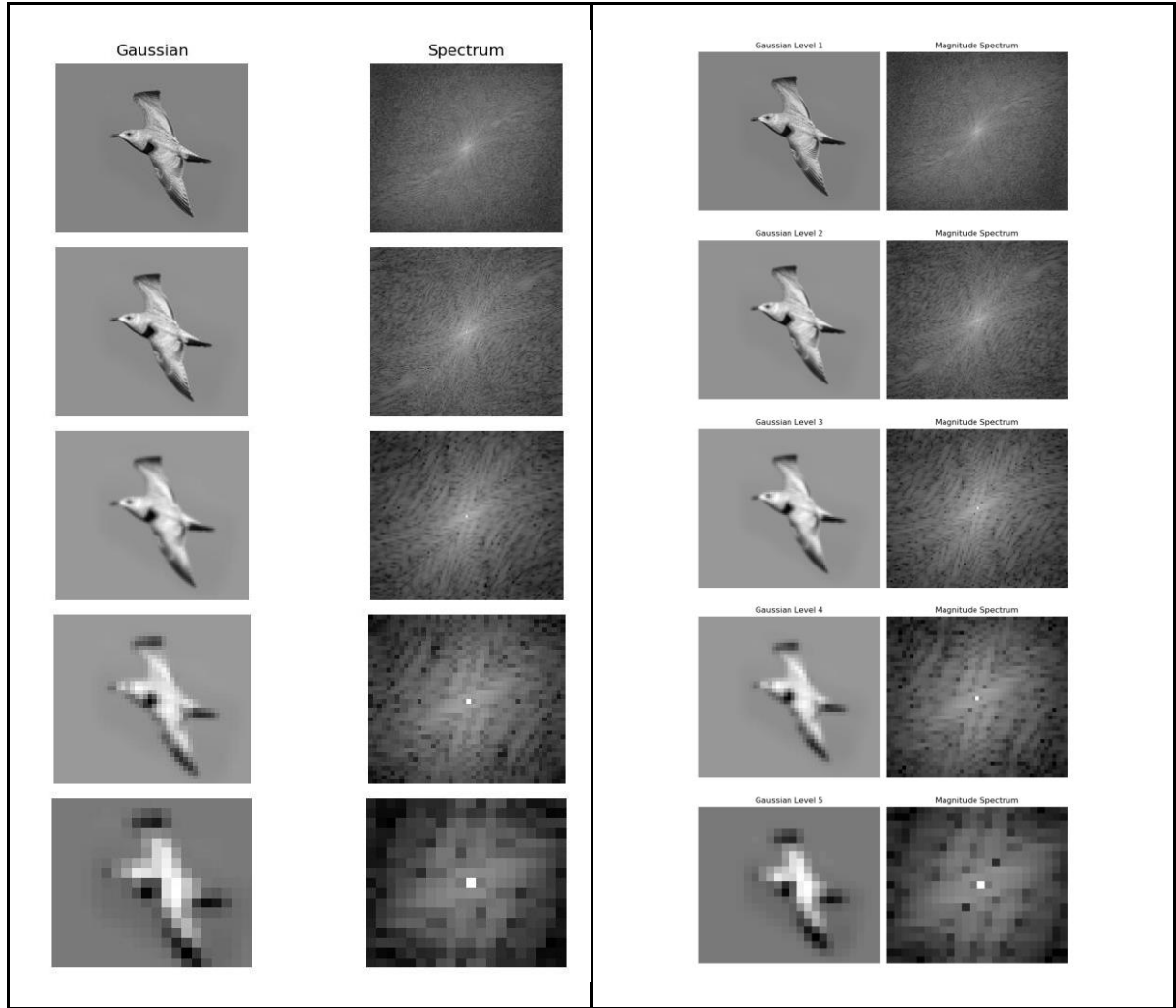
On the contrary, for a low-pass filter, a lower cutoff frequency means that the filter will allow only very low frequencies to pass through, resulting in a highly blurred image. Most details and edges will be removed, leaving only smooth areas. For a high-pass filter, a lower cutoff frequency will block most frequencies, so the image will only display the high-frequency components. This will emphasize sharp edges and fine details, creating a stark, high-contrast appearance. However, most of the image content will be blocked, resulting in a very sparse or skeletal appearance of the edges. For these reasons, the experimental results match what we observed earlier.

## **4.2. Image Pyramid**

We compared the OpenCV (cv2) implementation with our own. The OpenCV-based implementation produced smooth results with minimal artifacts, particularly in the magnitude spectra, which remained consistent across pyramid levels. This suggests that the cv2 implementation efficiently preserves frequency content during down-sampling. In contrast, our implementation exhibited slight misalignments, especially at the lower pyramid levels, with more noticeable changes in the magnitude spectra. This suggests a loss of frequency information, likely due to differences in handling border conditions or filters, leading to more pronounced pixel-level changes and increased blurring at deeper levels. While both methods effectively reduce image resolution, the cv2 implementation better preserves visual quality and frequency consistency across scales.

Gaussian Pyramid			
Our Implementation		CV2	
<p>Gaussian</p>     	<p>Spectrum</p>     	<p>Gaussian Level 1</p>  <p>Gaussian Level 2</p>  <p>Gaussian Level 3</p>  <p>Gaussian Level 4</p>  <p>Gaussian Level 5</p> 	<p>Magnitude Spectrum</p>  <p>Magnitude Spectrum</p>  <p>Magnitude Spectrum</p>  <p>Magnitude Spectrum</p>  <p>Magnitude Spectrum</p> 





### 4.3. Colorizing the Russian Empire

In Section 3.3, we mentioned that some images still exhibit slight misalignments, which is also noted in [1]. The authors observed that since NCC and SSD rely on comparing pixel colors for their calculations, they do not consider the low-level differences between channels, such as brightness, exposure, or blurriness. Therefore, they propose converting the images to edge-only representations using edge detection filters before applying NCC and SSD. This approach can help avoid low-level differences between pixels in the images.

The experimental results confirm this hypothesis. After applying the Sobel filter, we obtained the edge maps for the B, G, and R channels, and it is clear that without the influence of low-level information, the three images appear more apparent, as shown in **Figure 6**. **Table 2** displays the visual comparison results of NCC35, SSD35, SOBEL\_NCC35, and SOBEL\_SSD35, showing a significant improvement in the alignment, effectively reducing the misalignment issues.




B Channel	G Channel	R Channel
		

Fig 6. Edge maps for the B, G, and R channels.





NCC35	SSD35	SOBEL_NCC35	SOBEL_SSD35
			
G (20, 55), R (-175, 115)	G (20, 55), R (-175, 115)	G (20, 55), R (35, 120)	G (20, 55), R (35, 120)

Table 2. The visual result with the Sobel method.

## 5. Conclusion

In Assignment 2, we implemented three filtering and frequency-related image processing methods: hybrid images, image pyramids, and colorizing. These methods performed excellently on both the TA and our dataset provided, proving that our implementation was correct. In our discussion, we also experimented with different hyperparameters for the hybrid image and image pyramid methods, allowing us to understand better the effects that other parameters can bring. In the colorizing method, we solved the problem caused by low-level information. In the future, we hope to improve these methods further and apply them to our project.

## 6. Work Assignment Plan

廖怡誠	<ol style="list-style-type: none"> <li>1. implement code</li> <li>2. write the report - the <b>Colorizing the Russian Empire</b> part</li> </ol>
李品好	<ol style="list-style-type: none"> <li>1. implement code</li> <li>2. write the report - the <b>image pyramid</b> part</li> </ol>
林芷萱	<ol style="list-style-type: none"> <li>1. implement code</li> <li>2. write the report - the <b>hybrid image</b> part.</li> </ol>



## 7. References

- [1] [Discuss the impact of low-level information on colorizing.](#)

## 8. Command

These are our command for three methods. If you need to know more detailed input parameters, you can see the corresponding code.

Hybrid image
<code>python hybrid_image.py</code>
Image Pyramid (our method)
<code>python imagepyramid.py -d ./data/task1and2_hybrid_pyramid</code>
Image Pyramid (CV2 method)
<code>python imagepyramid_cv2.py -d ./data/task1and2_hybrid_pyramid</code>
Colorizing w/o Sobel filter
<code>python colorizing.py -d ./data/task3_colorizing/</code>
Colorizing w/ Sobel filter
<code>python colorizing_additional.py -d ./data/task3_colorizing/</code>