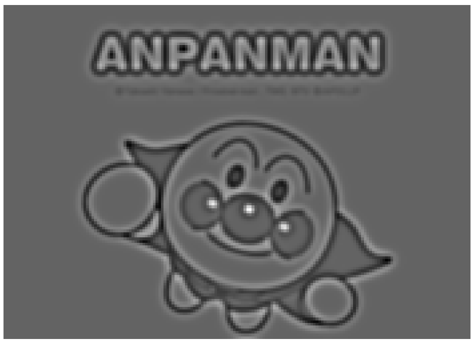
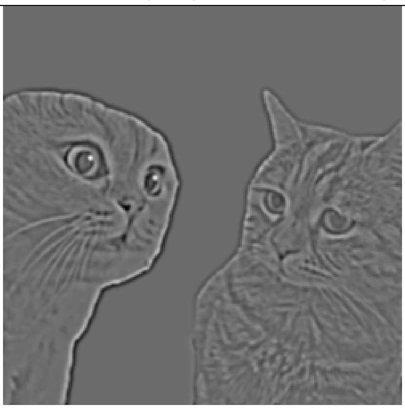
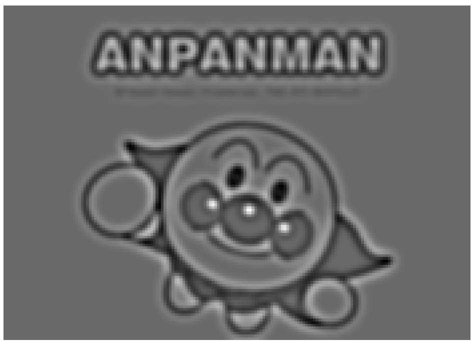
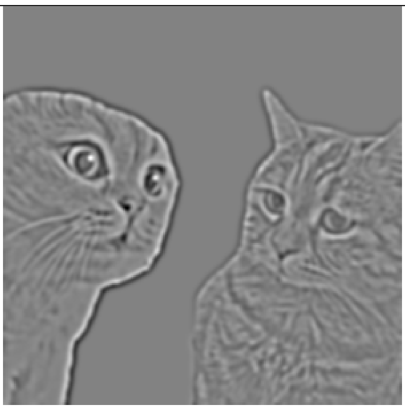
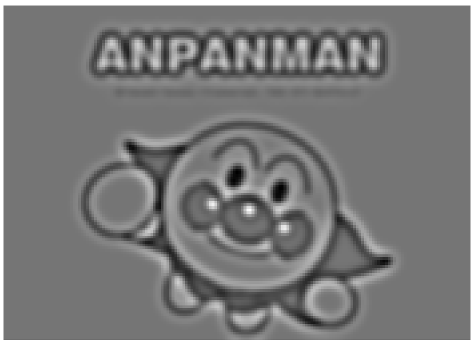
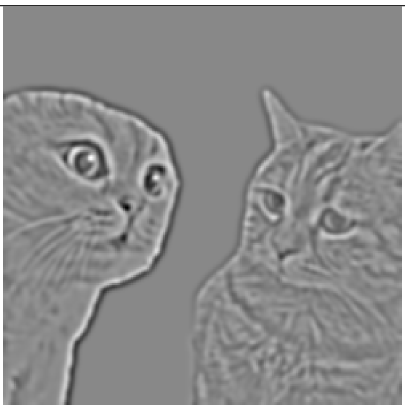
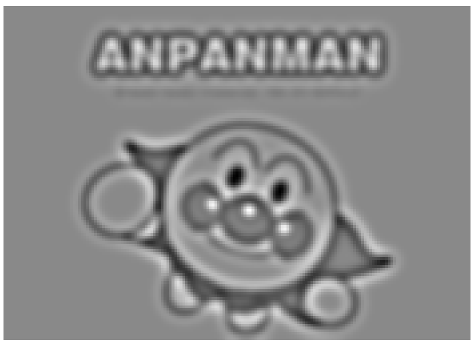
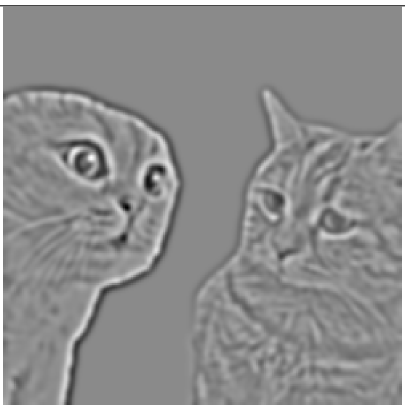
1 Part 1

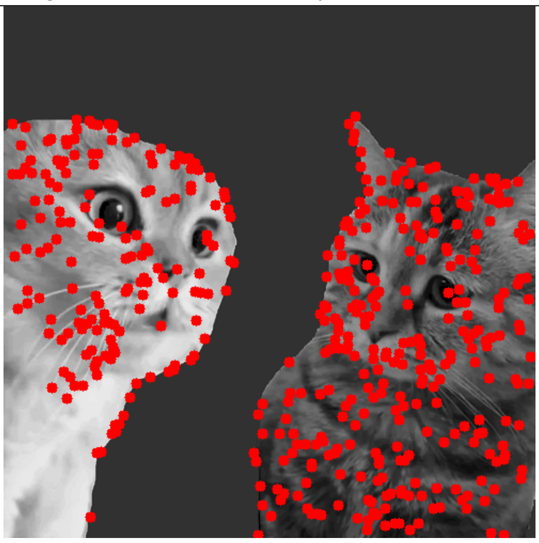
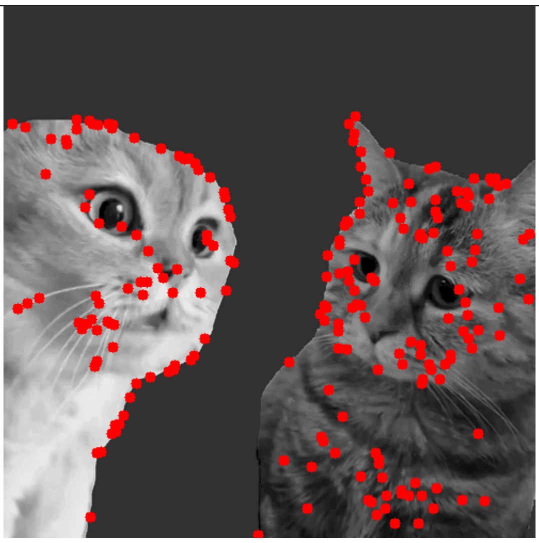
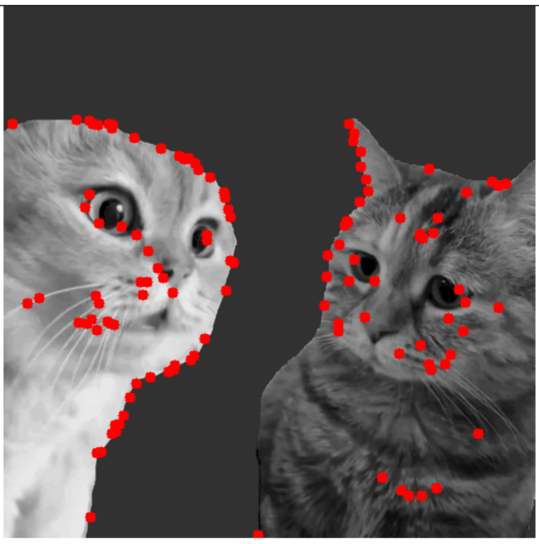
1.1 Visualize the DoG images of 1.png

Table 1: Visualizing DoG Images for 1.png and 2.png

	DoG Image (threshold = 3)		DoG Image (threshold = 3)
DoG1-1.png		DoG2-1.png	
DoG1-2.png		DoG2-2.png	
DoG1-3.png		DoG2-3.png	
DoG1-4.png		DoG2-4.png	

1.2 Use three thresholds (1,2,3) on 2.png and describe the difference

Table 2: Keypoints detected on 2.png with three thresholds

Threshold	Image with detected keypoints on 2.png
1	
2	
3	

(describe the difference)

By plotting the Difference of Gaussians (GoG) features on 2.png using three distinct threshold values (1, 2, 3), it becomes apparent that lower thresholds yield a great number of key points, whereas higher thresholds lead to a reduction in the number of detected key points. The keypoints are predominantly concentrated in the foreground rather than the background regions of the image because the colors in cat's body exhibit richer variation comared to the background color. Additionally, keypoints with higher levels of contrast receive more weighting, while keypoints with lower contrast may prone to being assotiated with noise points. Consequently, in instances where higher thresholds are applied, some keypoints on the cat's bodies may be omitted and filtered out, leaving predominately those points far from the background with more pronounced gradients, and noise points will be remoed from the keypoints list.

2 Part 2

2.1 Report the cost for each filtered image

Table 3: Cost evaluation for 1.png

Gray Scale Setting	Cost (1.png)
cv.2COLOR_BGR2GRAY	1207799
R*0.0+G*0.0+B*1.0	1439568
R*0.0+G*1.0+B*0.0	1305961
R*0.1+G*0.0+B*0.9	1393620
R*0.1+G*0.4+B*0.5	1279697
R*0.8+G*0.2+B*0.0	1127913

Table 4: Cost evaluation for 2.png

Gray Scale Setting	Cost (2.png)
cv2.COLOR_BGR2GRAY	183850
R*0.1+G*0.0+B*0.9	77882
R*0.2+G*0.0+B*0.8	86023
R*0.2+G*0.8+B*0.0	188019
R*0.4+G*0.0+B*0.6	128341
R*1.0+G*0.0+B*0.0	110862

2.2 Show original RGB image / two filtered RGB images and two grayscale images with highest and lowest cost.

(Describe the difference between those two grayscale images)

Upon examining Table 3, it becomes apparent that the combination yielding a higher cost is $R*0.0+G*0.0+B*1.0(1439568)$, while the one with a lower cost is $R*0.8+G*0.2+B*0.0(1127913)$. Since the predominant presence of red and green hues in the image, if the combination heavily favors blue or colors outside the color gamut during the RGB to YUV color space conversion, it may incur higher costs, and the brightness of the grayscale images will be diminished, leading to a suboptimal transformation in the final images. Although noise reduction is achieved, the image quality appears somewhat blurred. Conversely, the lower-cost combination utilizes weights from red and green, which are more prevalent in the image, resulting in a smaller mean shift and brighter overall brightness. This combination demonstrates better performance in the outcome image, ultimately delivering noise-free details on the original image.

Table 5: Comparing origin and filtered images with different costs on 1.png

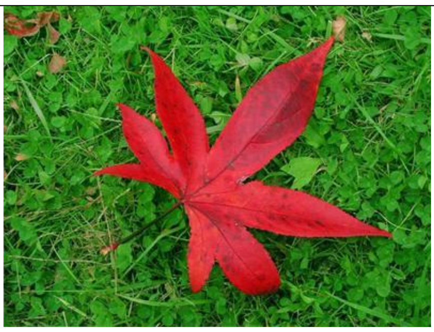
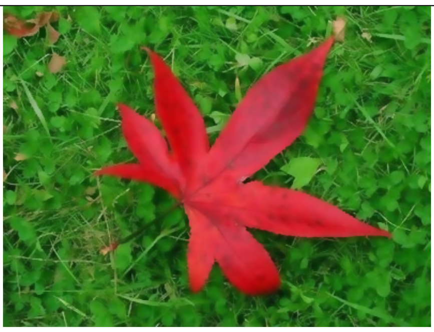
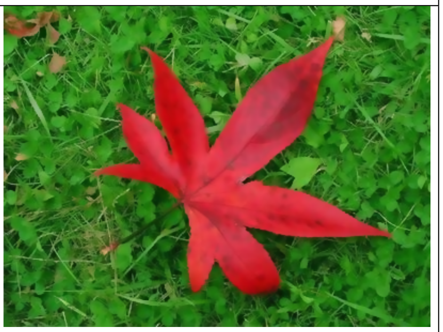
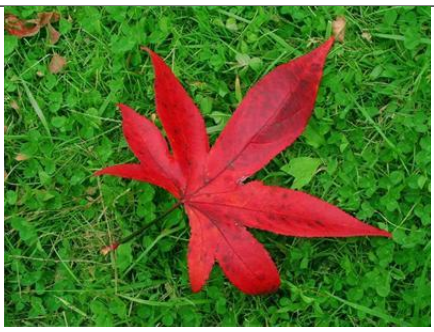
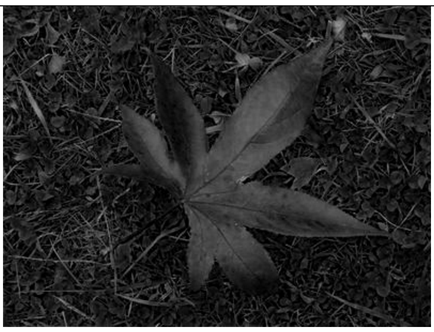
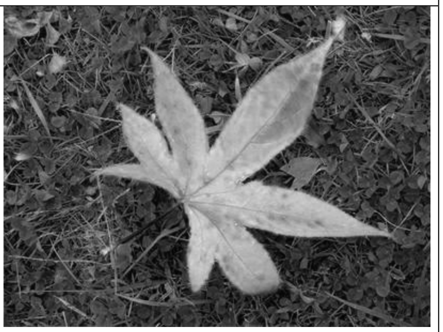
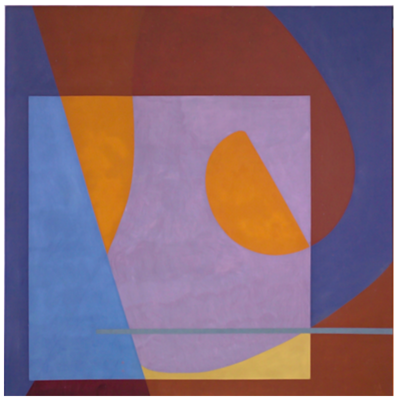
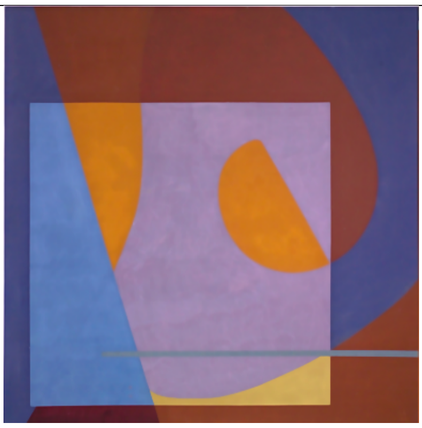
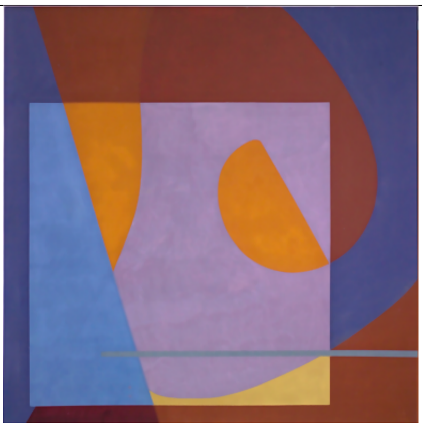
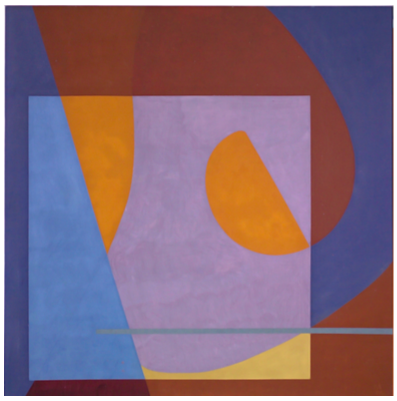
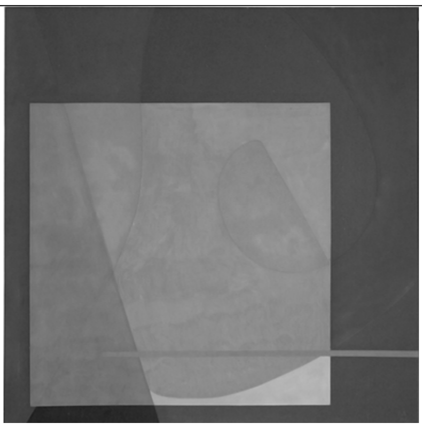
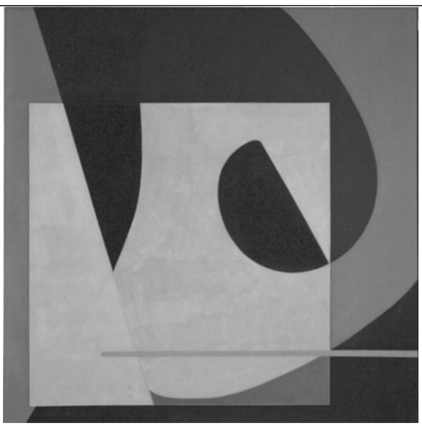
Original RGB image (1.png)	Filtered RGB image and Grayscale image of Highest cost	Filtered RGB image and Grayscale image of Lowest cost
		
		

Table 6: Comparing origin and filtered images with different costs on 2.png

Original RGB image (2.png)	Filtered RGB image and Grayscale image of Highest cost	Filtered RGB image and Grayscale image of Lowest cost
		
		

(Describe the difference between those two grayscale images)

Upon examining Table 4, it becomes apparent that the combination yielding a higher cost is $R*0.2+G*0.8+B*0.0(188019)$, while the one with a lower cost is $R*0.1+G*0.0+B*0.9(77882)$. Due to the variety of image color composition, direct observation via RGB colors poses a challenge to us. Nevertheless, through analyzing the fluctuation in grayscale luminance and subsequent texture alterations in RGB representation, we discern that the high-cost image, despite the noise has been removed, suffers from subdued color contrast across the entirety of the image. Conversely, the low-cost counterpart showcases enhanced contrast, facilitating a more distinct delineation of texture within the image.

2.3 Describe how to speed up the implementation of bilateral filter

From the definition, we have

$$\begin{aligned}
I^{\text{filtered}}(x) &= \frac{1}{W_p}\sum_{x_i\in\Omega} I(x_i) f_r(\|I(x_i)-I(x)\|) g_s(\|x_i-x\|) \\
\Rightarrow g(x,y) &= \frac{1}{W_p}\sum_{i,j\in[-r,r]} \underbrace{h_s(i,j)}_{\text{spatial kernel}}\ \underbrace{h_r(i,j)}_{\text{range kernel}}\ f(x-i,y-j) \\
W_p &= \sum_{x_i\in\Omega} f_r(\|I(x_i)-I(x)\|) g_s(\|x_i-x\|) \\
w(i,j,k,l) &= d(i,j,k,l)\cdot r(i,j,k,l) \\
&= \text{exp}\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}-\frac{\|I(i,j)-I(k,l)\|^2}{2\sigma_r^2}\right) \\
I_D(i,j) &= \frac{\sum_{k,l} I(k,l)w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)} \\
h_s(i,j) &= \text{exp}^{-\frac{i^2+j^2}{2\sigma_s^2}} \\
h_r(i,j) &= \text{exp}^{-\frac{[f(x-i,y-j)-f(x,y)]^2}{2\sigma_r^2}}
\end{aligned}$$

where

I^{filtered} is the filtered image.
I is the original input image.
x are the coordinates of the current pixel to be filtered.
Ω is the window centered in x, so $x \in \Omega$ is another pixel.
f_r is the range kernel for smoothing differences in intensities (Gaussian function).
g_s is the spatial kernel for smoothing differences in coordinates (Gaussian function).
W_p is the normalization term, also called weight.
$(i,j),(k,l)$ are the location of the pixel and the neighboring pixel.
$d(i,j,k,l), r(i,j,k,l)$ are the weight assigned for pixel (i,j) and pixel (k,l).
I_d is the denoised intensity of pixel (i,j)
σ_d, σ_r are smoothing parameters.

Upon analyzing the properties of the equations provided above, the program efficiency can be optimized through three key strategies.

1. The coefficient of spatial distance $g_d(i,j,k,l)$ is independent of pixel values and solely relies on the matrix dimensions. Hence, precomputing and looking up when required offers optimization potential. [1]

2. Optimizing grayscale value computation is imperative. Due to the large number of pixels, calculating each one individually incurs significant computational overhead. This indicates that the sheer volume of pixels necessitates a more efficient approach. Leveraging a lookup table for values within the $0-255$ range streamlines operations, obviating the need for individual computation, and it is good for efficient retrieval.

3. Streamlining the template from a two-dimensional to a one-dimensional format presents an opportunity for reducing algorithm complexity. This can be processed by refining the indexing mechanism, facilitating more streamlined processing.

As a result, the optimization process can be implemented by calculating the look-up table of the range weights and spatial weights, followed by computing the total weight by producing them together and then calculating the weights based on different channels. After computing the weights, the code updates the result and weight matrices by accumulating the weighted pixel values and weights. Finally, the filtered output is obtained by dividing the accumulated result by the accumulated weight.

3 Source code

3.1 Part 1 main.py

```python
import numpy as np
import cv2
import argparse
from DoG import Difference_of_Gaussian


def plot_keypoints(img_gray, keypoints, save_path):
    img = np.repeat(np.expand_dims(img_gray, axis=2), 3, axis=2)
    for y, x in keypoints:
        cv2.circle(img, (x, y), 5, (0, 0, 255), -1)
    cv2.imwrite(save_path, img)


def main():
    parser = argparse.ArgumentParser(
        description='main function of Difference of Gaussian')
    parser.add_argument(
        '--threshold',
        default=3.0,  # default=5.0,
        type=float,
        help='threshold value for feature selection')
    parser.add_argument('--image_path',
                        default='./testdata/1.png',
                        help='path to input image')
    args = parser.parse_args()

    print('Processing %s ...' % args.image_path)
    img = cv2.imread(args.image_path, 0).astype(np.float32)

    ### TODO ###
    DoG = Difference_of_Gaussian(args.threshold)
    keypoints = DoG.get_keypoints(img)
    plot_keypoints(
        img, keypoints,
        f'./result/DoG_{args.image_path[-5:-4]}_{args.threshold}_result.png')


if __name__ == '__main__':
    main()
```

3.2 Part 1 DoG.py

```
import numpy as np
import cv2


class Difference_of_Gaussian(object):

    def __init__(self, threshold):
        self.threshold = threshold
        self.sigma = 2**(1 / 4)
        self.num_octaves = 2
        self.num_DoG_images_per_octave = 4
        self.num_guassian_images_per_octave = self.num_DoG_images_per_octave + 1

    def findkeypoints(self, images: np.ndarray, octave: int) -> set:
        keypoints = set()
        row, column = images[0].shape
        for x in range(1, row - 2):
            for y in range(1, column - 2):
                for DoG in range(1, self.num_DoG_images_per_octave - 1):
                    pixel = images[DoG, x, y]
                    cube = images[DoG - 1:DoG + 2, x - 1:x + 2,
                                  y - 1:y + 2].flatten()
                    if (np.absolute(pixel) >= self.threshold
                            and (pixel >= max(cube) or pixel <= min(cube))):
                        keypoints.add((x * octave, y * octave))
        return keypoints

    def save_dog_images(self, images: np.ndarray) -> None:
        for idx, img in enumerate(images):
            max_val = max(img.flatten())
            min_val = min(img.flatten())
            norm_img = (img - min_val) * 255 / (max_val - min_val)
            cv2.imwrite(f'./result/DoG_1_{idx+1}.png', norm_img)

    def get_keypoints(self, image):
        ### TODO ####
        # Step 1: Filter images with different sigma values (5 images per octave, 2 octave in total)
        # - Function: cv2.GaussianBlur (kernel = (0, 0), sigma = self.sigma**___)
        gaussian_images = []

        # 1st octave
        first_octave = [image]
        first_octave.extend(
            cv2.GaussianBlur(image, ksize=(0, 0), sigmaX=self.sigma**idx)
            for idx in range(1, self.num_guassian_images_per_octave))
        # 2nd octave
        resize_factor = 0.5
        second_octave = [
            cv2.resize(
                first_octave[-1],
                None,  # dsize
                fx=resize_factor,
                fy=resize_factor,
                interpolation=cv2.INTER_NEAREST)  # INTER_LENEAR is default
        ]
        second_octave.extend(
```

```python
            cv2.GaussianBlur(
                second_octave[0], ksize=(0, 0), sigmaX=self.sigma**idx)
            for idx in range(1, self.num_guassian_images_per_octave))

        # gaussian_images.extend(first_octave)
        # gaussian_images.extend(second_octave)

        # Step 2: Subtract 2 neighbor images to get DoG images (4 images per octave, 2 octave in total)
        # - Function: cv2.subtract(second_image, first_image)
        dog_images = []

        dog_images = [0] * (self.num_DoG_images_per_octave * self.num_octaves)
        for idx in range(self.num_DoG_images_per_octave):
            dog_images[idx] = cv2.subtract(first_octave[idx],
                                           first_octave[idx + 1])
            dog_images[idx + self.num_DoG_images_per_octave] = cv2.subtract(
                second_octave[idx], second_octave[idx + 1])

        # save images
        # self.save_dog_images(dog_images[4:])

        # Step 3: Thresholding the value and Find local extremum (local maximun and local minimum)
        #         Keep local extremum as a keypoint

        keypoints = set()
        keypoints.update(
            self.findkeypoints(
                np.array(dog_images[0:self.num_DoG_images_per_octave]), 1))
        keypoints.update(
            self.findkeypoints(
                np.array(dog_images[self.num_DoG_images_per_octave:]), 2))
        keypoints = list(map(list, keypoints))

        # Step 4: Delete duplicate keypoints
        # - Function: np.unique

        keypoints = np.unique(np.array(keypoints), axis=0)

        # sort 2d-point by y, then by x
        keypoints = keypoints[np.lexsort((keypoints[:, 1], keypoints[:, 0]))]
        return keypoints
```

3.3 Part 2 main.py

```python
import numpy as np
import pandas as pd
import cv2
import argparse
import os
from JBF import Joint_bilateral_filter


def read_settings(setting_path: str) -> tuple[list, int, float]:
    with open(setting_path, "r", encoding="UTF-8") as f:
        setting = [line.rstrip('\n').split(',') for line in f.readlines()]
    f.close()
    return setting[1:6], int(setting[6][1]), float(setting[6][3])


def main():
    parser = argparse.ArgumentParser(
        description='main function of joint bilateral filter')
    parser.add_argument('--image_path',
                        default='./testdata/2.png',
                        help='path to input image')
    parser.add_argument('--setting_path',
                        default='./testdata/2_setting.txt',
                        help='path to setting file')
    args = parser.parse_args()

    img = cv2.imread(args.image_path)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    ### TODO ###
    RGB_params, sigma_s, sigma_r = read_settings(args.setting_path)
    JBF = Joint_bilateral_filter(sigma_s, sigma_r)
    img_BF = JBF.joint_bilateral_filter(img_rgb, img_rgb)
    img_JBF = JBF.joint_bilateral_filter(img_rgb, img_gray)

    cost = dict()
    cost['BGR2GRAY'] = np.sum(
        np.abs(img_BF.astype('int32') - img_JBF.astype('int32')))
    # save image
    cv2.imwrite(
        f'./result/{args.image_path[-5:-4]}_image_cv2_COLOR_BGR2GRAY.png',
        img_gray)
    cv2.imwrite(f'./result/{args.image_path[-5:-4]}_BF.png', img_BF)
    cv2.imwrite(f'./result/{args.image_path[-5:-4]}_JBF.png', img_JBF)

    # 6 gray-scale images
    for R, G, B in RGB_params:
        # rgb to gray
        img_param_gray = img_rgb[:, :, 0] * float(
            R) + img_rgb[:, :, 1] * float(G) + img_rgb[:, :, 2] * float(B)
        img_param_JBF = JBF.joint_bilateral_filter(img_rgb, img_param_gray)
        cost[f'RGB_{R}_{G}_{B}'] = np.sum(
            np.abs(img_BF.astype('int32') - img_param_JBF.astype('int32')))
        img_param_JBF = cv2.cvtColor(img_param_JBF, cv2.COLOR_BGR2RGB)
        cv2.imwrite(
            f'./result/{args.image_path[-5:-4]}_RGB_{R}_{G}_{B}_gray.png',
            img_param_gray)
        cv2.imwrite(
            f'./result/{args.image_path[-5:-4]}_RGB_{R}_{G}_{B}_JBF.png',
            img_param_JBF)
    pd.Series(cost, name='Cost(1.pmg)').to_excel(
        f'./result/{args.image_path[-5:-4]}_cost_table.xlsx')


if __name__ == '__main__':
    main()
```

3.4 Part 2 JBF.py

```python
import numpy as np
import cv2
from tqdm import trange


class Joint_bilateral_filter(object):

    def __init__(self, sigma_s, sigma_r):
        self.sigma_r = sigma_r
        self.sigma_s = sigma_s
        self.wndw_size = 6 * sigma_s + 1
        self.pad_w = 3 * sigma_s

    def joint_bilateral_filter(self, img, guidance):
        BORDER_TYPE = cv2.BORDER_REFLECT
        padded_img = cv2.copyMakeBorder(img, self.pad_w, self.pad_w,
                                        self.pad_w, self.pad_w,
                                        BORDER_TYPE).astype(np.int32)
        padded_guidance = cv2.copyMakeBorder(guidance, self.pad_w, self.pad_w,
                                             self.pad_w, self.pad_w,
                                             BORDER_TYPE).astype(np.int32)
        ### TODO ###
        # G_s(p, q) = e ^ (-(((x_p - x_q)^2 + (y_p - y_q)^2) / (2 * sigma_s ^ 2)))
        table_G_s = np.exp(-(np.arange(self.pad_w + 1)**2) /
                           (2 * self.sigma_s**2))

        # T is single-channel:  G_r(T_p, T_q) = e ^ (-((T_p-T_q)^2 / (2 * sigma_s ^ 2)))
        # T is color image:     G_r(T_p, T_q) = e ^ (-(((T_p^r-T_q^r)^2+(T_p^q-T_q^q)^2+(T_p^b-T_q^b)^2) /
        table_G_r = np.exp(-((np.arange(256) / 255)**2 /
                             (2 * self.sigma_r**2)))

        hwc = padded_img.shape
        weight = np.zeros(hwc)
        result = np.zeros(hwc)

        for x in trange(-self.pad_w, self.pad_w + 1):
            for y in range(-self.pad_w, self.pad_w + 1):
                dT = table_G_r[np.abs(
                    np.roll(padded_guidance, [y, x], axis=[0, 1]) -
                    padded_guidance)]
                r_weight = dT if dT.ndim == 2 else np.prod(dT, axis=2)
                s_weight = table_G_s[np.abs(x)] * table_G_s[np.abs(y)]
                t_weight = s_weight * r_weight
                padded_img_roll = np.roll(padded_img, [y, x], axis=[0, 1])
                for channel in range(padded_img.ndim):
                    result[:, :,
                           channel] += padded_img_roll[:, :,
                                                       channel] * t_weight
                    weight[:, :, channel] += t_weight
        output = (result / weight)[self.pad_w:-self.pad_w,
                                   self.pad_w:-self.pad_w, :]

        return np.clip(output, 0, 255).astype(np.uint8)
```

References

[1] K. N. Chaudhury, D. Sage, and M. Unser, “Fast $o(1)$ bilateral filtering using trigonometric range kernels,” *IEEE Transactions on Image Processing*, vol. 20, no. 12, pp. 3376–3382, 2011.