CSC420 Assignment 2
Zhihong Wang
wangz154
1002095207

Q1.(a).
Code:

```
def OneDimLinearInterpolation(image, d, direction):

    image_rows = image.shape[0]
    image_cols = image.shape[1]

    temp_image = np.zeros(shape=(image_rows, image_cols*d, 3))

    rows = temp_image.shape[0]
    cols = temp_image.shape[1]

    result = []

    if direction == 0:

        for m in range(rows):
            temp = []
            for n in range(cols):
                i = n/d
                if i.is_integer():
                    output = image[m, int(i)]
                else:
                    x = i
                    x_1 = math.floor(x)
                    x_2 = math.ceil(x)
                    if x_2 == image_cols:
                        break
                    else:
                        r_1 = (x_2 - x) / (x_2 - x_1)
                        r_2 = (x - x_1) / (x_2 - x_1)
                        output = r_1 * image[m, x_1] + r_2 * image[m, x_2]
                temp.append(output)
            result.append(temp)

    output_image = np.array(result)

    if direction == 1:

        rotated_image = np.rot90(image, -1)
        temp_result = OneDimLinearInterpolation(rotated_image, d, 0)
        output_image = np.rot90(temp_result, 1)

    return output_image
```

1D Final Result: 685 × 677

First, I applied 1D interpolation, horizontally, then vertically, the result is above. If you do vertically first, then horizontally, it will also lead to the same result.

2D Final Result: 685 × 677

When d = 4 (quadruple), the filter is:

```
[[ 0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ],
       [ 0.    ,  0.0625,  0.125 ,  0.1875,  0.25  ,  0.1875,  0.125 ,  0.0625,  0.    ],
       [ 0.    ,  0.125 ,  0.25  ,  0.375 ,  0.5   ,  0.375 ,  0.25  ,  0.125 ,  0.    ],
       [ 0.    ,  0.1875,  0.375 ,  0.5625,  0.75  ,  0.5625,  0.375 ,  0.1875,  0.    ],
       [ 0.    ,  0.25  ,  0.5   ,  0.75  ,  1.    ,  0.75  ,  0.5   ,  0.25  ,  0.    ],
       [ 0.    ,  0.1875,  0.375 ,  0.5625,  0.75  ,  0.5625,  0.375 ,  0.1875,  0.    ],
       [ 0.    ,  0.125 ,  0.25  ,  0.375 ,  0.5   ,  0.375 ,  0.25  ,  0.125 ,  0.    ],
       [ 0.    ,  0.0625,  0.125 ,  0.1875,  0.25  ,  0.1875,  0.125 ,  0.0625,  0.    ],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ]]
```

Q1.(b).

Suppose that we want to find the value of the unknown function f at the point (x, y). It is assumed that we know the value of f at the four points $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$, and $Q_{22} = (x_2, y_2)$.

We first do linear interpolation in the x-direction. This yields

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}),$$
$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}).$$

We proceed by interpolating in the y-direction to obtain the desired estimate:

$$\begin{aligned} f(x, y) &\approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \\ &= \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right) \\ &= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \big(f(Q_{11})(x_2 - x)(y_2 - y) + f(Q_{21})(x - x_1)(y_2 - y) + f(Q_{12})(x_2 - x)(y - y_1) + f(Q_{22})(x - x_1)(y - y_1) \big) \\ &= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}. \end{aligned}$$

Note that we will arrive at the same result if the interpolation is done first along the y direction and then along the x direction.[1]

```python
def TwoDimLinearInterpolation(image, d):

    image_rows = image.shape[0]
    image_cols = image.shape[1]

    temp_image = np.zeros(shape=((image_rows - 1) * d + 1, (image_cols - 1) * d + 1, 3))

    temp_image[::d, ::d] = image.copy()

    one_d_filter = []
    for i in range(d):
        one_d_filter.append(i/d)
    for i in range(d, -1, -1):
        one_d_filter.append(i/d)

    one_d_filter = np.array(one_d_filter)

    two_d_filter = np.matmul(np.transpose(np.array([one_d_filter])), np.array([one_d_filter]))

    result = cv2.filter2D(temp_image,-1,two_d_filter)

    return result
```

2D Final Result: 685 × 677

This result is the same as part (a).

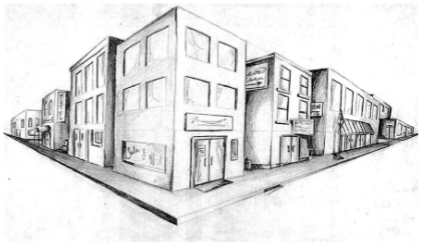
Q2.(a).
Test:

```
667
668     # q2a_harris = harris(img_building, 0.05, 0.01)
669     # q2a_harris2 = harris2(img_building, 0.05, 0.01)
670
671     # q2a_brown = brown(img_building)
672     # q2a_brown2 = brown2(img_building)
673
674     # q2a_harris_suppression = harris(img_building, 0.05, 0.01, 1)
675     # q2a_brown_suppression = brown(img_building, 5)
```

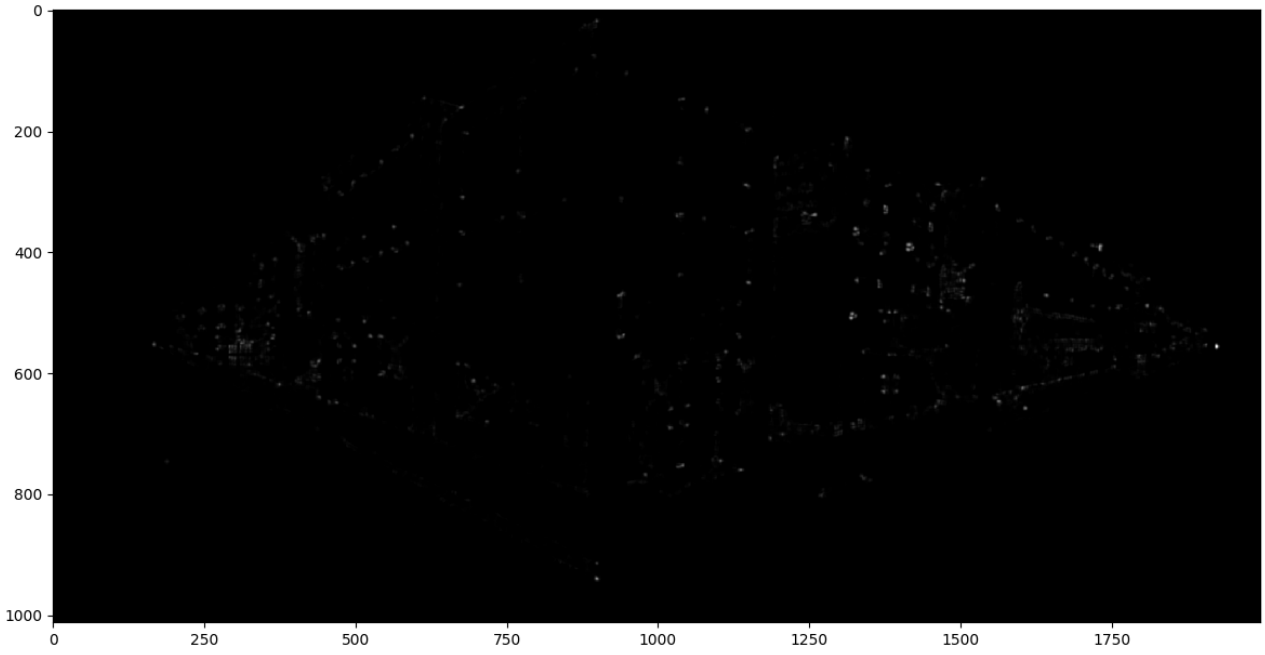
Harris:

```
78  def harris(image, alpha, threshold, radius):
79
80      gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
81      blur = cv2.GaussianBlur(gray, (5, 5), 7)
82      ddepth = cv2.CV_64F
83
84      # STEP1: Compute gradient Ix and Iy
85      Ix = cv2.Sobel(src=blur, ddepth=ddepth, dx=1, dy=0, ksize=5)
86      Iy = cv2.Sobel(src=blur, ddepth=ddepth, dx=0, dy=1, ksize=5)
87
88      # STEP2: Compute Ix^2, Iy^2 and Ix*Iy
89      IxIy = np.multiply(Ix, Iy)
90      Ix2 = np.multiply(Ix, Ix)
91      Iy2 = np.multiply(Iy, Iy)
92
93      # STEP3: Average(Gaussian)
94      Ix2_blur = cv2.GaussianBlur(Ix2, (7, 7), 10)
95      Iy2_blur = cv2.GaussianBlur(Iy2, (7, 7), 10)
96      IxIy_blur = cv2.GaussianBlur(IxIy, (7, 7), 10)
97
98      # STEP4: Compute R
99      det = np.multiply(Ix2_blur, Iy2_blur) - np.multiply(IxIy_blur, IxIy_blur)
100     trace = Ix2_blur + Iy2_blur
101     R = det - alpha * np.multiply(trace, trace)
102
103     # plt.subplot(1, 2, 1), plt.imshow(image), plt.axis('off')
104     # plt.subplot(1, 2, 2), plt.imshow(R, cmap='gray'), plt.axis('off')
105     # plt.imshow(R, cmap="gray")
106     # plt.show()
107
108     #R = R * (R > (threshold * R.max())) * (R > 0)
109     R[R < threshold] = 0
110
111     # plt.subplot(1, 2, 1), plt.imshow(image), plt.axis('off'), plt.show()
112     # plt.subplot(1, 2, 2), plt.imshow(R, cmap='gray'), plt.axis('off'), plt.show()
113     # plt.imshow(R, cmap="gray")
114     # plt.show()
115
116     result = non_max_suppress(R, radius)
117
118     plt.imshow(result, cmap="gray")
119     plt.show()
120
121     return 0
```

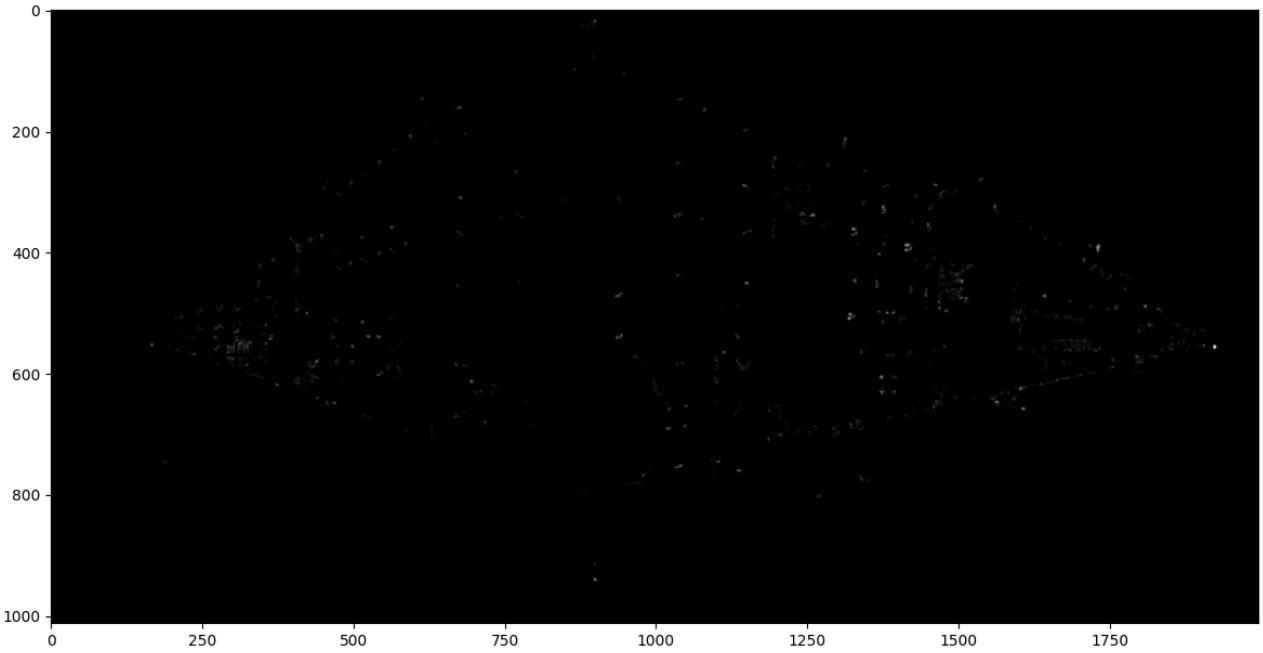
Harris Compare:



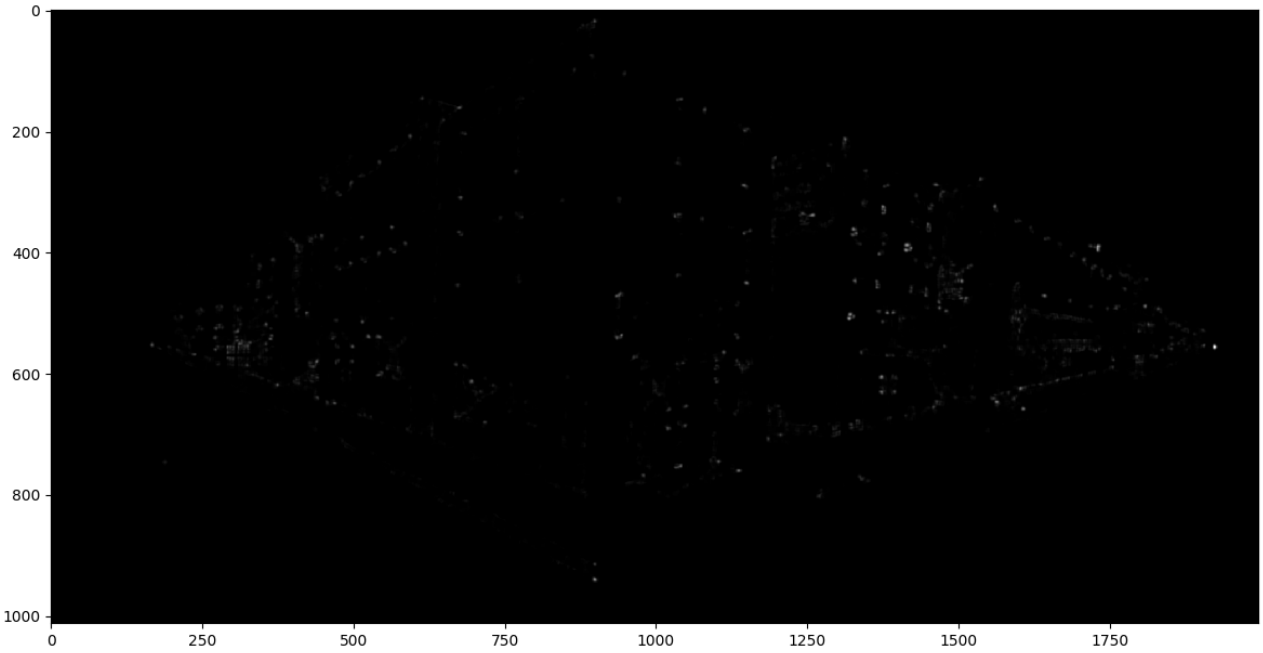
Harris Threshold:
Alpha = 0.01



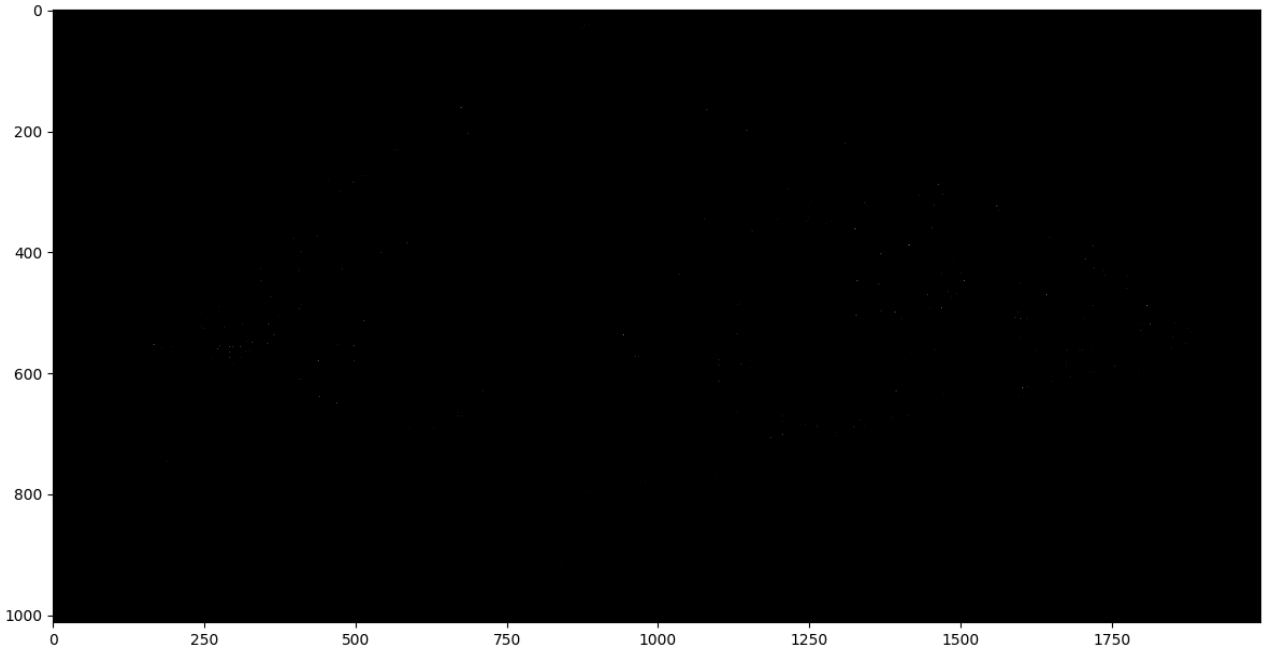
Alpha = 0.05



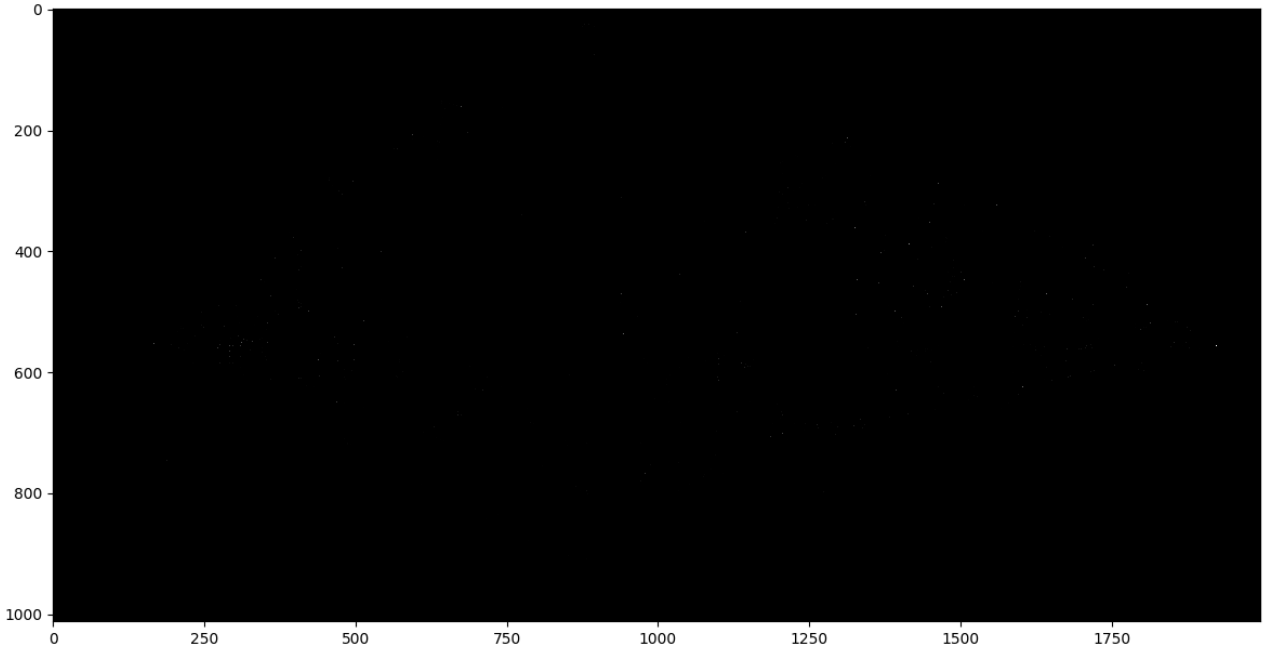
Alpha = 0.1



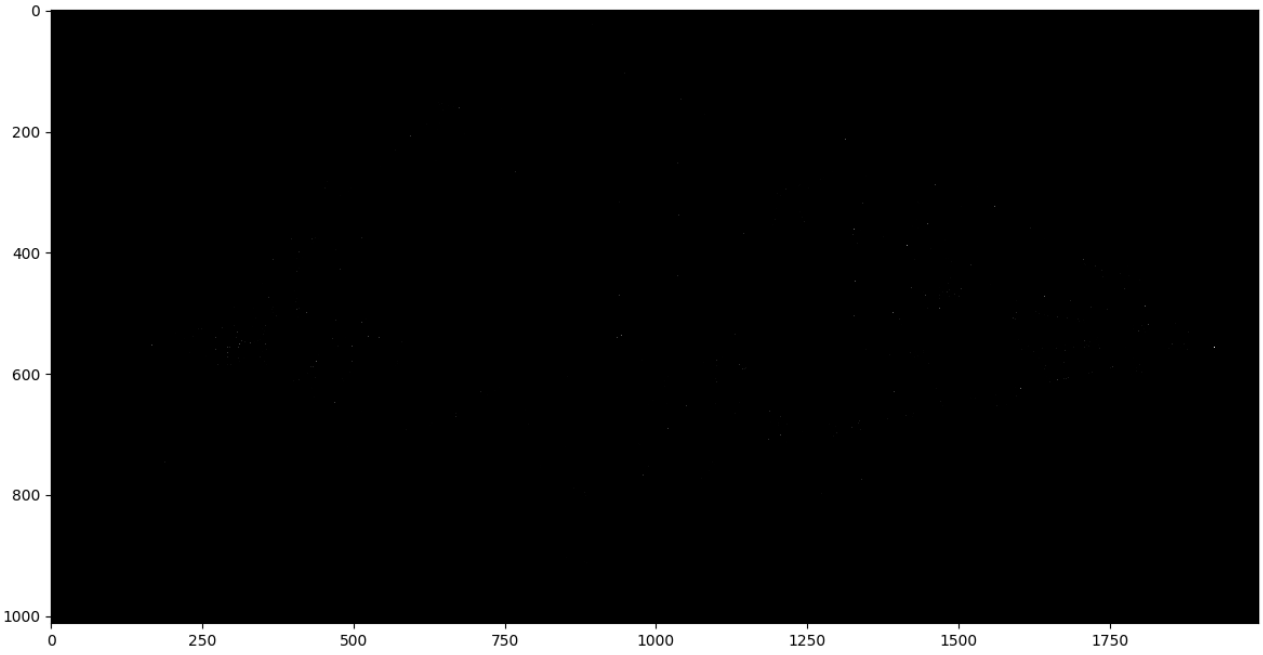
Harris Non-maximal Suppression:
Alpha = 0.01



Alpha = 0.05



Alpha = 0.1

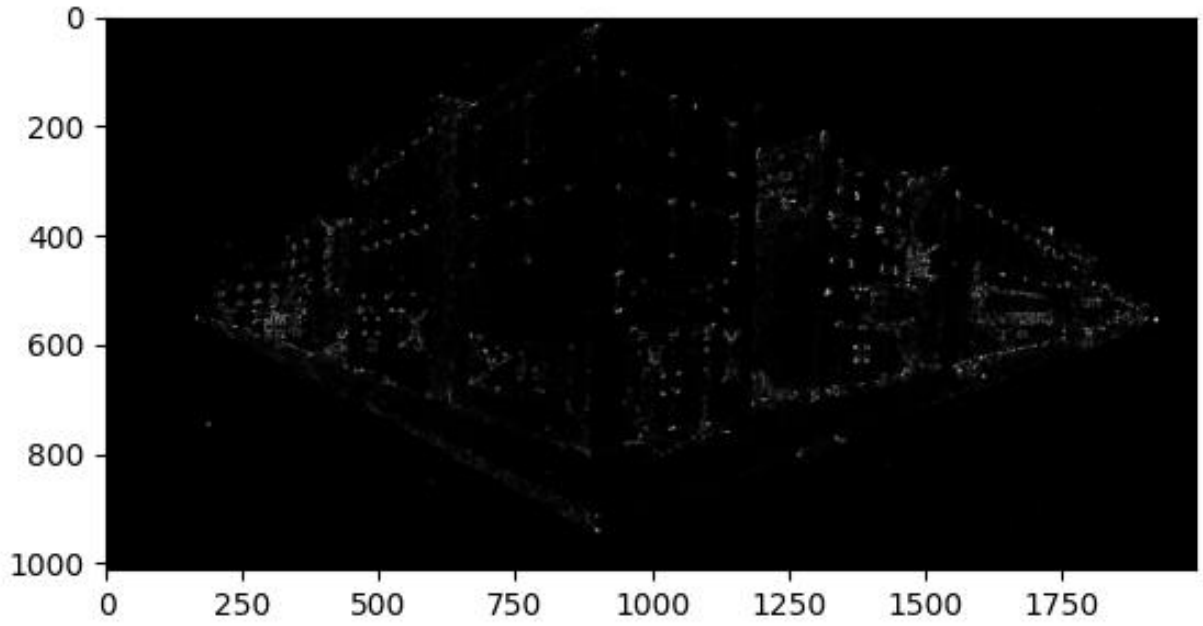


There is no big difference of these when alpha increasing little by little, the corners is slightly clearer.

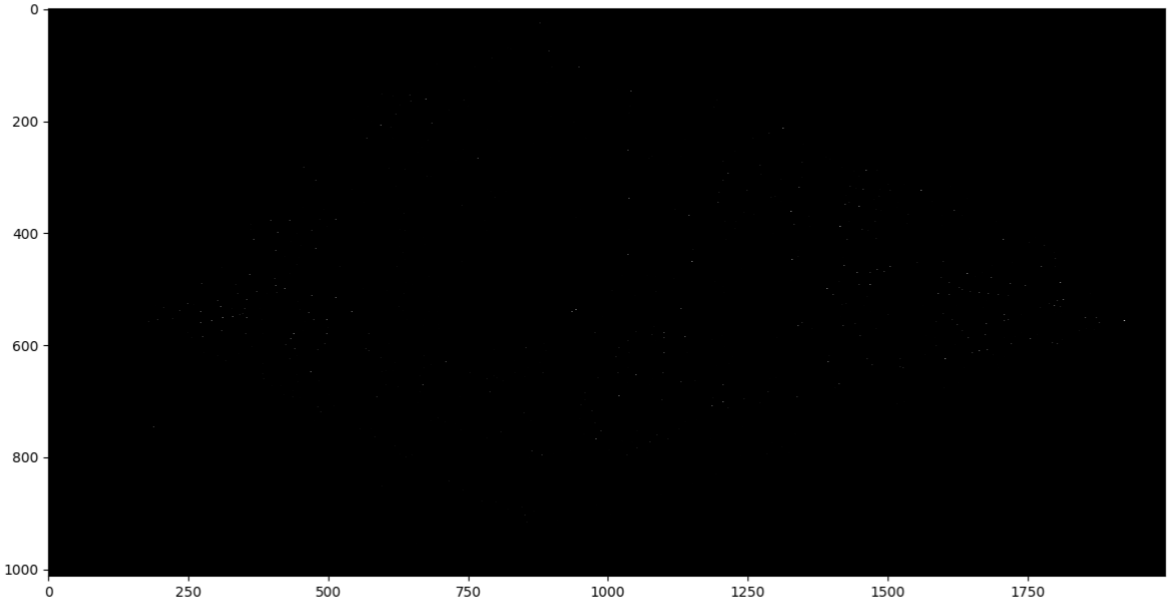
Brown:

```python
def brown(image, radius):

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (5, 5), 7)
    ddepth = cv2.CV_64F

    # STEP1: Compute gradient Ix and Iy
    Ix = cv2.Sobel(src=blur, ddepth=ddepth, dx=1, dy=0, ksize=5)
    Iy = cv2.Sobel(src=blur, ddepth=ddepth, dx=0, dy=1, ksize=5)

    # STEP2: Compute Ix^2, Iy^2 and Ix*Iy
    IxIy = np.multiply(Ix, Iy)
    Ix2 = np.multiply(Ix, Ix)
    Iy2 = np.multiply(Iy, Iy)

    # STEP3: Average(Gaussian)
    Ix2_blur = cv2.GaussianBlur(Ix2, (7, 7), 10)
    Iy2_blur = cv2.GaussianBlur(Iy2, (7, 7), 10)
    IxIy_blur = cv2.GaussianBlur(IxIy, (7, 7), 10)

    # STEP4: Compute R
    det = np.multiply(Ix2_blur, Iy2_blur) - np.multiply(IxIy_blur, IxIy_blur)
    trace = Ix2_blur + Iy2_blur
    R = np.divide(det, trace+0.01)

    # plt.subplot(1, 2, 1), plt.imshow(image), plt.axis('off'), plt.show()
    # plt.subplot(1, 2, 2), plt.imshow(R, cmap='gray'), plt.axis('off'), plt.show()
    # plt.imshow(R, cmap="gray")
    # plt.show()

    result = non_max_suppress(R, radius)

    plt.imshow(result, cmap="gray")
    plt.show()

    return 0
```

Brown Threshold:



Brown non-maximal Suppression:



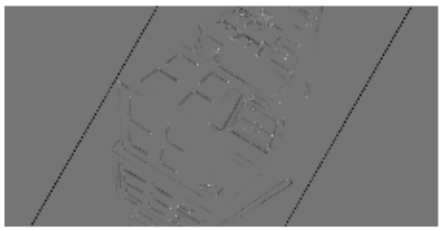
Harris without det and trace:

```python
def harris2(image, alpha, threshold):

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (5, 5), 7)
    ddepth = cv2.CV_64F

    # STEP1: Compute gradient Ix and Iy
    Ix = cv2.Sobel(src=blur, ddepth=ddepth, dx=1, dy=0, ksize=5)
    Iy = cv2.Sobel(src=blur, ddepth=ddepth, dx=0, dy=1, ksize=5)

    # STEP2: Compute Ix^2, Iy^2 and Ix*Iy
    IxIy = np.multiply(Ix, Iy)
    Ix2 = np.multiply(Ix, Ix)
    Iy2 = np.multiply(Iy, Iy)

    # STEP3: Average(Gaussian)
    Ix2_blur = cv2.GaussianBlur(Ix2, (7, 7), 10)
    Iy2_blur = cv2.GaussianBlur(Iy2, (7, 7), 10)
    IxIy_blur = cv2.GaussianBlur(IxIy, (7, 7), 10)

    matrix = np.zeros((Ix2_blur.shape[0], Ix2_blur.shape[1], 2, 2))
    matrix[:, :, 0, 0] = Ix2_blur
    matrix[:, :, 0, 1] = IxIy_blur
    matrix[:, :, 1, 0] = IxIy_blur
    matrix[:, :, 1, 1] = Iy2_blur

    lambda_result = np.linalg.eigvals(matrix)
    lambda_0 = lambda_result[:, :, 0]
    lambda_1 = lambda_result[:, :, 1]

    # STEP4: Compute R
    R = lambda_0 * lambda_1 - alpha * (lambda_0 + lambda_1) ** 2
    R[R < threshold] = 0

    plt.subplot(1, 2, 1), plt.imshow(image), plt.axis('off'), plt.show()
    plt.subplot(1, 2, 2), plt.imshow(R, cmap='gray'), plt.axis('off'), plt.show()

    plt.imshow(R, cmap="gray")
    plt.show()

    return 0
```

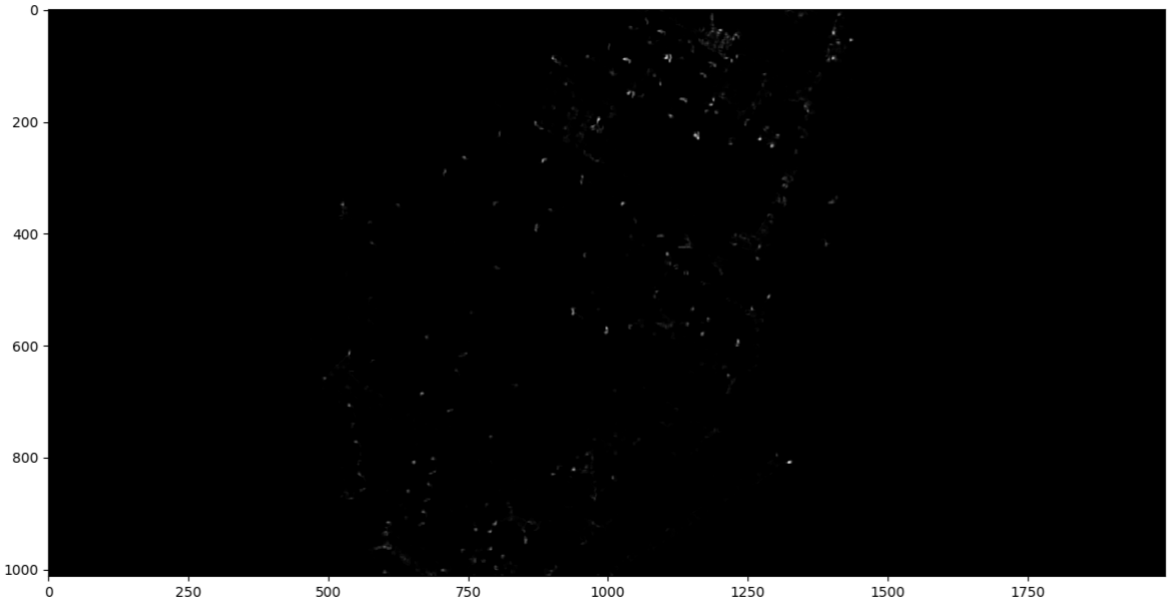
Brown without det and trace:

```
202
203 def brown2(image):
204
205     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
206     blur = cv2.GaussianBlur(gray, (5, 5), 7)
207     ddepth = cv2.CV_64F
208
209     # STEP1: Compute gradient Ix and Iy
210     Ix = cv2.Sobel(src=blur, ddepth=ddepth, dx=1, dy=0, ksize=5)
211     Iy = cv2.Sobel(src=blur, ddepth=ddepth, dx=0, dy=1, ksize=5)
212
213     # STEP2: Compute Ix^2, Iy^2 and Ix*Iy
214     IxIy = np.multiply(Ix, Iy)
215     Ix2 = np.multiply(Ix, Ix)
216     Iy2 = np.multiply(Iy, Iy)
217
218     # STEP3: Average(Gaussian)
219     Ix2_blur = cv2.GaussianBlur(Ix2, (7, 7), 10)
220     Iy2_blur = cv2.GaussianBlur(Iy2, (7, 7), 10)
221     IxIy_blur = cv2.GaussianBlur(IxIy, (7, 7), 10)
222
223     matrix = np.zeros((Ix2_blur.shape[0], Ix2_blur.shape[1], 2, 2))
224     matrix[:, :, 0, 0] = Ix2_blur
225     matrix[:, :, 0, 1] = IxIy_blur
226     matrix[:, :, 1, 0] = IxIy_blur
227     matrix[:, :, 1, 1] = Iy2_blur
228
229     lambda_result = np.linalg.eigvals(matrix)
230     lambda_0 = lambda_result[:, :, 0]
231     lambda_1 = lambda_result[:, :, 1]
232
233     # STEP4: Compute R
234     R = np.divide(lambda_0 * lambda_1 , (lambda_0 + lambda_1)+0.01)
235
236     plt.subplot(1, 2, 1), plt.imshow(image), plt.axis('off'), plt.show()
237     plt.subplot(1, 2, 2), plt.imshow(R, cmap='gray'), plt.axis('off'), plt.show()
238
239     plt.imshow(R, cmap="gray")
240     plt.show()
241
242     return 0
243
```

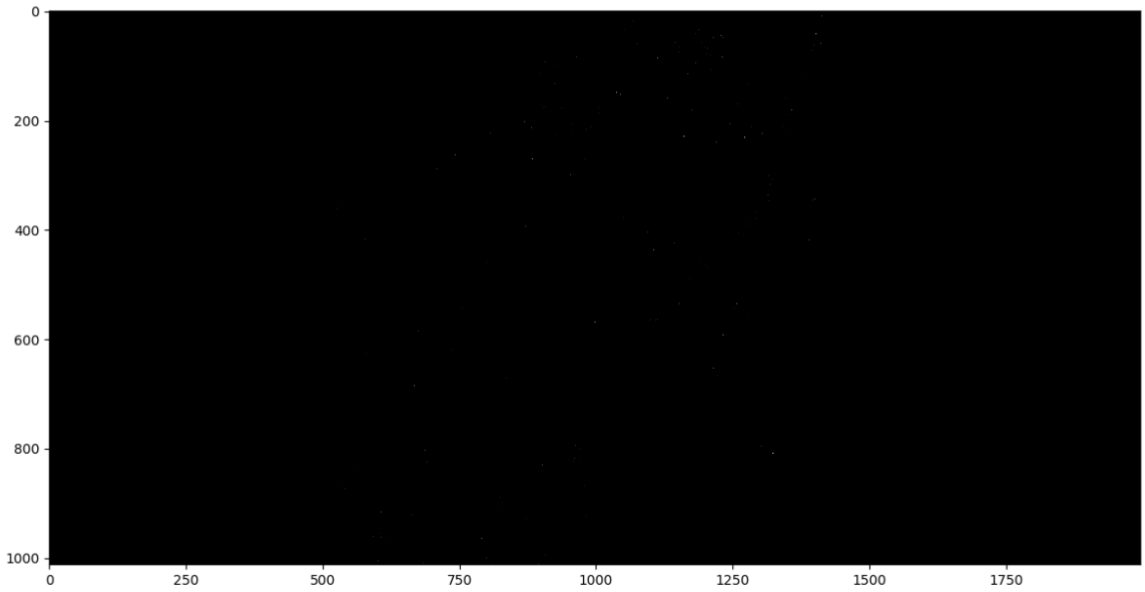
Q2.(b).
Rotated 60 degree:



Rotated Threshold:



Rotated Non-maximal Suppression:

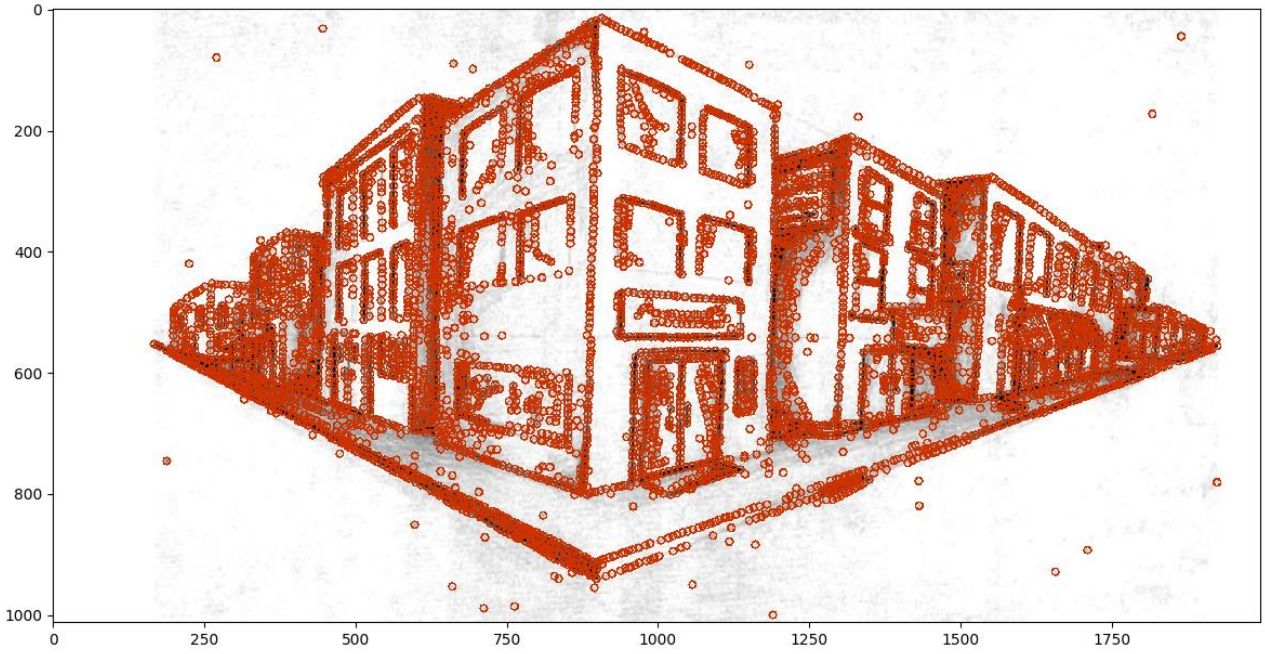


By observation, we can see the same corners can be still detected. Harris is rotation invariant because Second moment ellipse rotates but its shape (i.e. eigenvalues) remains the same.

Q2.(c).

```python
def LOG(image, levels, initial_sigma, scale, threshold):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY).astype(np.float)

    n = image.shape[0]
    m = image.shape[1]

    offset = 2
    # construct pyramid
    sigma = initial_sigma
    sigma_list = [0]
    increment = 1

    pyramid = np.empty((levels + offset, n, m))

    # pyramid = [np.zeros(gray.shape)]
    for level in range(levels):
        new_level = nd.gaussian_laplace(gray, sigma * increment)
        pyramid[level + 1] = new_level
        increment = increment * scale
        sigma_list.append(sigma * increment)
    # pyramid.append(np.zeros(gray.shape))

    # Non Maximum Suppression
    max_list = []
    for level in range(1, levels):
        # Construct upper current and lower matrix
        # Current Level
        cur_level = pyramid[level]
        cur_level_mtx = np.zeros((n + offset, m + offset))
        cur_level_mtx[1: n + 1, 1: m + 1] = cur_level[0: n, 0: m]
        # Upper Level
        upper_level = pyramid[level - 1]
        up_level_mtx = np.zeros((n + offset, m + offset))
        up_level_mtx[1: n + 1, 1: m + 1] = upper_level[0: n, 0: m]
        # Lower Level
        lower_level = pyramid[level + 1]
        low_level_mtx = np.zeros((n + offset, m + offset))
        low_level_mtx[1: n + 1, 1: m + 1] = lower_level[0: n, 0: m]

        for row in range(n):
            for col in range(m):
                # Prepare for filtering
                patch_size = 3
                c_p = (patch_size - 1) // 2
                # Get currently filtering matrices from upper, current, lower levels
                upper_mtx = np.array(up_level_mtx[row: row + patch_size, col: col + patch_size])
                cur_mtx = np.array(cur_level_mtx[row: row + patch_size, col: col + patch_size])
                lower_mtx = np.array(low_level_mtx[row: row + patch_size, col: col + patch_size])
                # Get max value form upper, current, lower levels
                up_cur_low_max = []
                up_cur_low_max.append(np.max(upper_mtx))
                up_cur_low_max.append(np.max(cur_mtx))
                up_cur_low_max.append(np.max(lower_mtx))
                max_value = np.amax(up_cur_low_max)
                # Get current filtering center point value
                c_p_value = cur_mtx[c_p, c_p]
                # Threshold
                if max_value == c_p_value and threshold < c_p_value:
                    max_list.append(((row, col), sigma_list[level]))

    for max in max_list:
        cv2.circle(image, (max[0][1],
                           max[0][0]),
                   int(np.round(max[1])) * 3,
                   color=(123, 255, 222),
                   thickness=2)

    plt.imshow(image)
    plt.show()
```

```
677     # LOG(img_building,levels=5,initial_sigma=1.6,scale=np.sqrt(2),threshold=5)
```



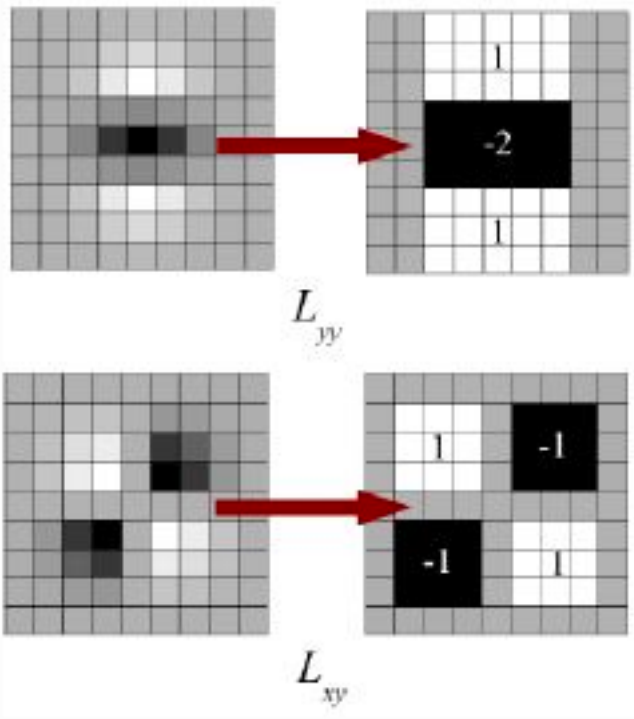
Q2.(d).
SURF detector: SURF stands for Speeded Up Robust Features which is inspired by SIFT and approaches corner detection via Laplacian of Gaussian. SURF uses an integer approximation of the determinant of Hessian blob detector using the precomputed integral image:

$$S(x, y) = \sum^{x} \sum^{y} i{=}0\ j{=}0\ \ I(i, j)$$

$$\text{Hessian matrix: } \mathrm{H}(\mathrm{p}, \sigma) = \begin{pmatrix} L_{xx}(\mathrm{p}, \sigma) & L_{xy}(\mathrm{p}, \sigma) \\ L_{yx}(\mathrm{p}, \sigma) & L_{yy}(\mathrm{p}, \sigma) \end{pmatrix}$$

The determinant of the Hessian matrix is evaluated as the measurement of the local changes around the point. For non-maximum suppression, choose the point with maximum determinant of Hessian in its neighbor as the key-point.

In SIFT, Lowe approximated Laplacian of Gaussian with Difference of Gaussian for finding scale-space. SURF goes a little further and approximates LoG with Box Filter. Below image shows a demonstration of such an approximation. One big advantage of this approximation is that, convolution with box filter can be easily calculated with the help of integral images. And it can be done in parallel for different scales. Also the SURF rely on determinant of Hessian matrix for both scale and location.



The creation of SURF descriptor takes place in two steps. The first step consists of fixing a reproducible orientation based on information from a circular region around the keypoint. Then, we construct a square region aligned to the selected orientation and extract the SURF descriptor from it.

1. Surf first calculate the Haar-wavelet responses in x and y-direction, and this in a circular neighborhood of radius 6s around the keypoint, with s the scale at which the keypoint was detected. Also, the sampling step is scale dependent and chosen to be s, and the wavelet responses are computed at that current scale s. Accordingly, at high scales the size of the wavelets is big. Therefore integral images are used again for fast filtering.

2. Then we calculate the sum of vertical and horizontal wavelet responses in a scanning area, then change the scanning orientation (add π/3), and re-calculate, until we find the orientation with largest sum value, this orientation is the main orientation of feature descriptor.

3. The first step consists of constructing a square region centered around the keypoint and oriented along the orientation we already got above. The size of this window is 20s.

4. Then the region is split up regularly into smaller 4 × 4 square sub-regions. For each sub-region, we compute a few simple features at 5×5 regularly spaced sample points. For reasons of simplicity, we call **dx** the Haar wavelet response in the horizontal direction and **dy** the Haar wavelet response in the vertical direction (filter size 2s). To increase the robustness towards geometric deformations and localization errors, the responses dx and **dy** are first weighted with a Gaussian (σ = 3.3s) centered at the keypoint.

Then, the wavelet responses **dx** and **dy** are summed up over each subregion and form a first set of entries to the feature vector. In order to bring in information about the polarity of the intensity changes, we also extract the sum of the absolute values of the responses, **|dx|** and **|dy|**. Hence, each sub-region has a four-dimensional descriptor vector v for its underlying intensity structure **V .** This results in a descriptor vector for all 4×4 sub-regions of **length 64** (In **Sift**, our descriptor is the **128-D vector**, so this is part of the reason that SURF is faster than Sift).

Q3.(a).

Q3.

(a). $G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$

$$\frac{\partial G(x,y,\sigma)}{\partial x} = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \cdot \left(-\frac{2x}{2\sigma^2}\right)$$

$$= -\frac{x}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$\frac{\partial^2 G}{\partial x^2} = -\frac{1}{2\pi\sigma^4} \cdot \left(e^{-\frac{x^2+y^2}{2\sigma^2}} + e^{-\frac{x^2+y^2}{2\sigma^2}} \cdot -\frac{x^2}{\sigma^2}\right)$$

$$= -\frac{1}{2\pi\sigma^4} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}} \left(1 - \frac{x^2}{\sigma^2}\right)$$

$$\therefore \frac{\partial^2 G}{\partial y^2} = -\frac{1}{2\pi\sigma^4} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}} \left(1 - \frac{y^2}{\sigma^2}\right)$$

$$\therefore \nabla^2 G(x,y,\sigma) = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2} = -\frac{1}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}} \left(2 - \frac{x^2+y^2}{\sigma^2}\right)$$

LoG is not separable.

LoG cannot be rewritten to a product of two linear filters because it has $\left(2 - \frac{x^2+y^2}{\sigma^2}\right)$ term in closed-form expression.

Also, like this example: $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ ← This LoG filter ranked as 2, which is not separable.

Q3.(b).

Q3(b). By SIFT paper from Lec 8, the scale-normalized LoG $\sigma^2\nabla^2 G$ has maxima & minima produce most stable features.

Heat diffusion equation: $\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G$

So we can get: $\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{k\sigma - \sigma}$

($\nabla^2 G$ can be computed from finite difference approximation to $\frac{\partial G}{\partial \sigma}$)

(scale at $k\sigma$ and σ)

Therefore, DoG (left hand side) can approximation to scale-normalized LoG $\sigma^2\nabla^2 G$. $G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2\nabla^2 G$

For DoG, let $\sigma_1 = k\sigma$, $\sigma_2 = \sigma$.

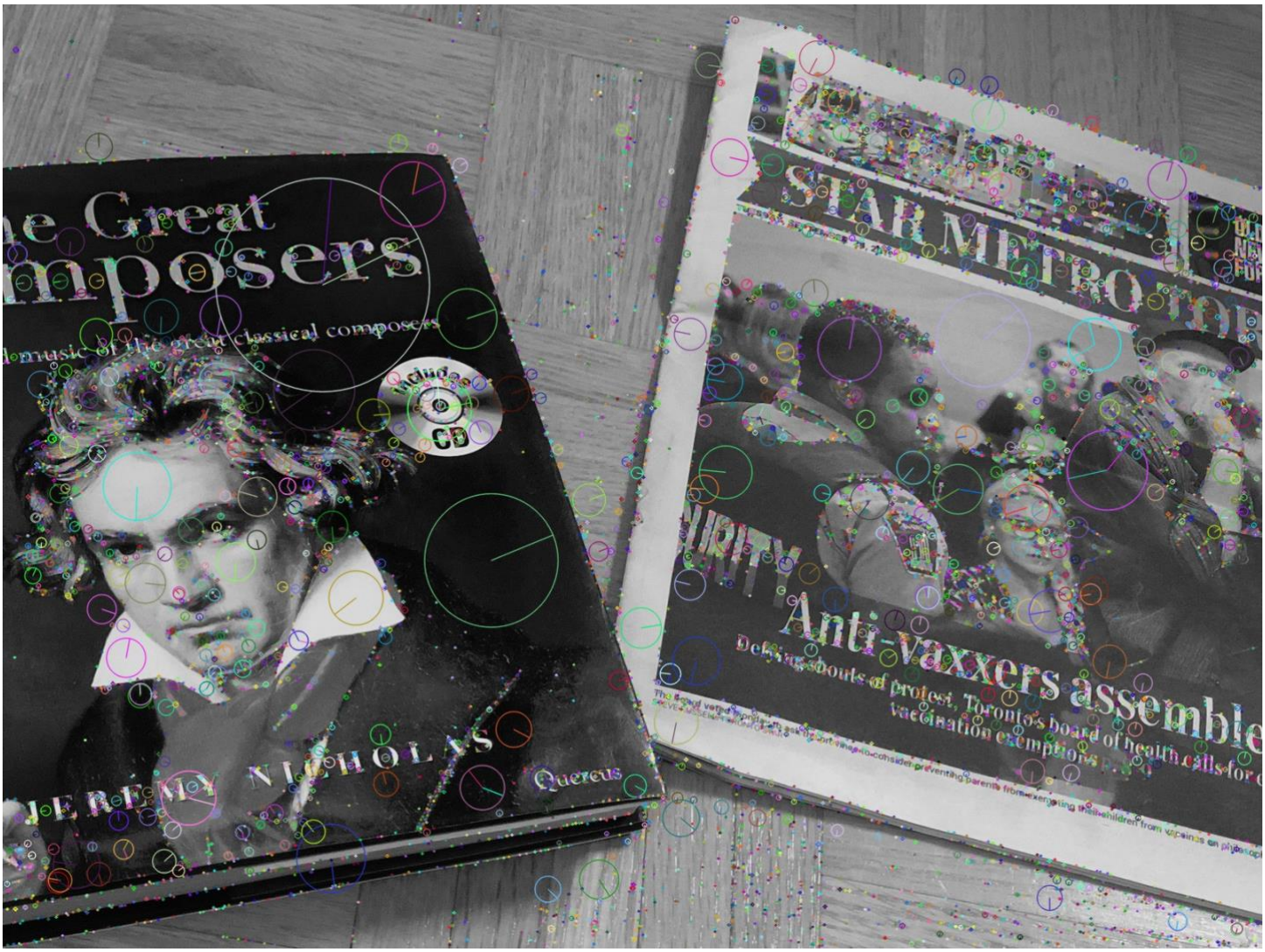
We want $\sigma_1 \approx \sigma_2 \Rightarrow k\sigma \approx \sigma \Rightarrow k \approx 1$

When $k \Rightarrow 1$, approximation error $\Rightarrow 0$.

This maximizes the approximation to LoG $\sigma^2\nabla^2 G$

Q4.(a).

```python
def sift_extract(image):
    img = cv2.imread(image)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Initialized sift
    sift = cv2.xfeatures2d.SIFT_create()
    flag = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS

    # Get keypoints and features
    keypoints, features = sift.detectAndCompute(gray_img, None)
    result = cv2.drawKeypoints(gray_img, keypoints, flags=flag, outImage=None)

    cv2.imwrite('sift_keypoints.jpg', result)
    return keypoints, features
```


e Great
mposers
Anti-vaxxers assemble
vaccination exemptions

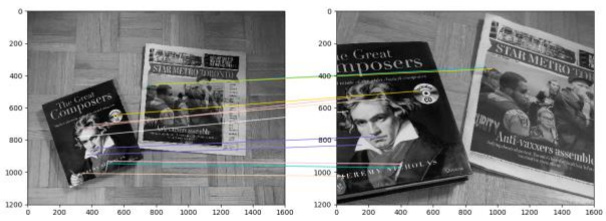
Q4.(b).

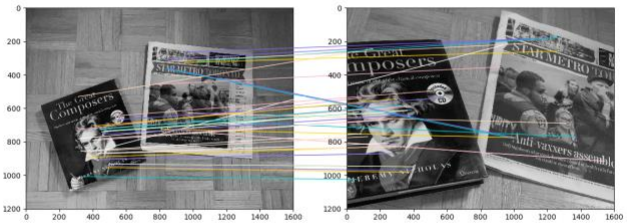
```python
def sift_matching(image1, image2, ratio_threshold, norm_level):
    img1 = cv2.imread(image1)
    img2 = cv2.imread(image2)

    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    kp_1, des_1 = sift_extract(image1)
    kp_2, des_2 = sift_extract(image2)

    candidate_keypoint_location = []
    candidate_distance = []

    num_des_1 = np.shape(des_1)[0]
    num_des_2 = np.shape(des_2)[0]

    num_matching_keypoint = num_des_1

    # Compare all keypoints
    for i in range(num_des_1):
        distance = []
        for j in range(num_des_2):
            euclidean_dis = euclidean_distance(des_1[i], des_2[j], norm_level)
            distance.append(euclidean_dis)
        # find two smallest distance match
        two_smallest_distance = heapq.nsmallest(2, distance)
        candidate_distance.append(two_smallest_distance)
        # find minimum distance index at keypoint 2
        np_distance = np.array(distance)
        min_dis_index = np.unravel_index(np.argmin(np_distance, axis=None), np_distance.shape)[0]
        candidate_keypoint_location.append(min_dis_index)


    keypoint1 = []
    keypoint2 = []
    keypoints_pairs = {}
    count = 0

    for i in range(num_matching_keypoint):
        ratio = np.true_divide(candidate_distance[i][0], candidate_distance[i][1])
        if ratio < ratio_threshold:
            count = count + 1
            kp1 = kp_1[i]
            kp2 = kp_2[candidate_keypoint_location[i]]
            keypoint1.append(kp1)
            keypoint2.append(kp2)
            keypoints_pairs[(kp1,kp2)] = ratio

    sorted_keypointes = sorted(keypoints_pairs.items(), key=lambda kv: kv[1])

    print("number of matching = ", count)

    top_ten = sorted_keypointes[0:9]
    print(top_ten)

    flag = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS

    result_img1 = cv2.drawKeypoints(gray1, keypoint1, outImage=None, flags=flag)
    result_img2 = cv2.drawKeypoints(gray2, keypoint2, outImage=None, flags=flag)
    cv2.imwrite("sample1_keypoints_final.jpg", result_img1)
    cv2.imwrite("sample2_keypoints_final.jpg", result_img2)
    show_matching(gray1, keypoint1, gray2, keypoint2)

    return top_ten
```

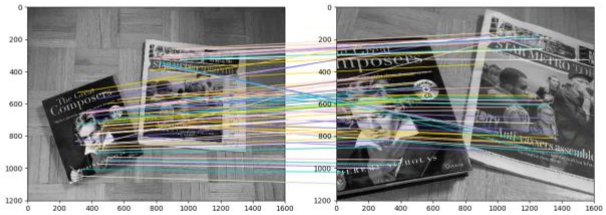
When norm level = 2, it is L2 norm:

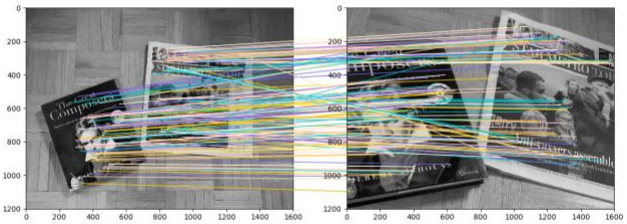
```
458
459     def euclidean_distance(vector1, vector2, norm_level):
460         vector1 = vector1.reshape(-1)
461         vector2 = vector2.reshape(-1)
462         distance = np.linalg.norm(vector1 - vector2, ord=norm_level)
463         return distance
474
475 def show_matching(image1, keypoint1, image2, keypoint2):
476     ax1 = plt.subplot(1, 2, 1)
477     ax2 = plt.subplot(1, 2, 2)
478     img1 = cv2.drawKeypoints(image1, keypoint1, None)
479     img2 = cv2.drawKeypoints(image2, keypoint2, None)
480     ax1.imshow(img1)
481     ax2.imshow(img2)
482
483     colors = ["white", "LightPink", "DarkTurquoise", "PeachPuff", "MediumSlateBlue", "Gold"]
484     color_i = 0
485     for kp1, kp2 in zip(keypoint1, keypoint2):
486         coord1 = kp1.pt
487         coord2 = kp2.pt
488         c = colors[color_i % 6]
489         con = ConnectionPatch(xyA=coord2, xyB=coord1, coordsA="data", coordsB="data",
490                               axesA=ax2, axesB=ax1, arrowstyle="-", color=c)
491         ax2.add_patch(con)
492         color_i += 1
493     plt.show()
494
442 def generate_top_ten(image1, image2, top_ten):
443     img1 = cv2.imread(image1)
444     img2 = cv2.imread(image2)
445
446     gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
447     gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
448
449     keypoint1 = []
450     keypoint2 = []
451
452     for item in top_ten:
453         keypoint1.append(item[0][0])
454         keypoint2.append(item[0][1])
455
456     show_matching(gray1, keypoint1, gray2, keypoint2)
457
```

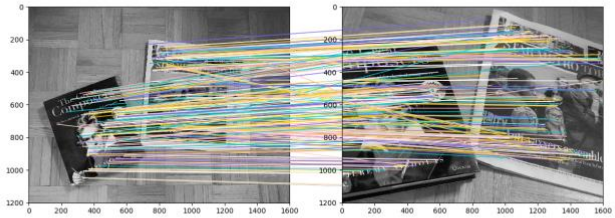
Results when threshold increasing:

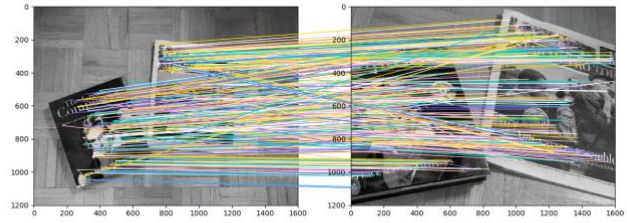






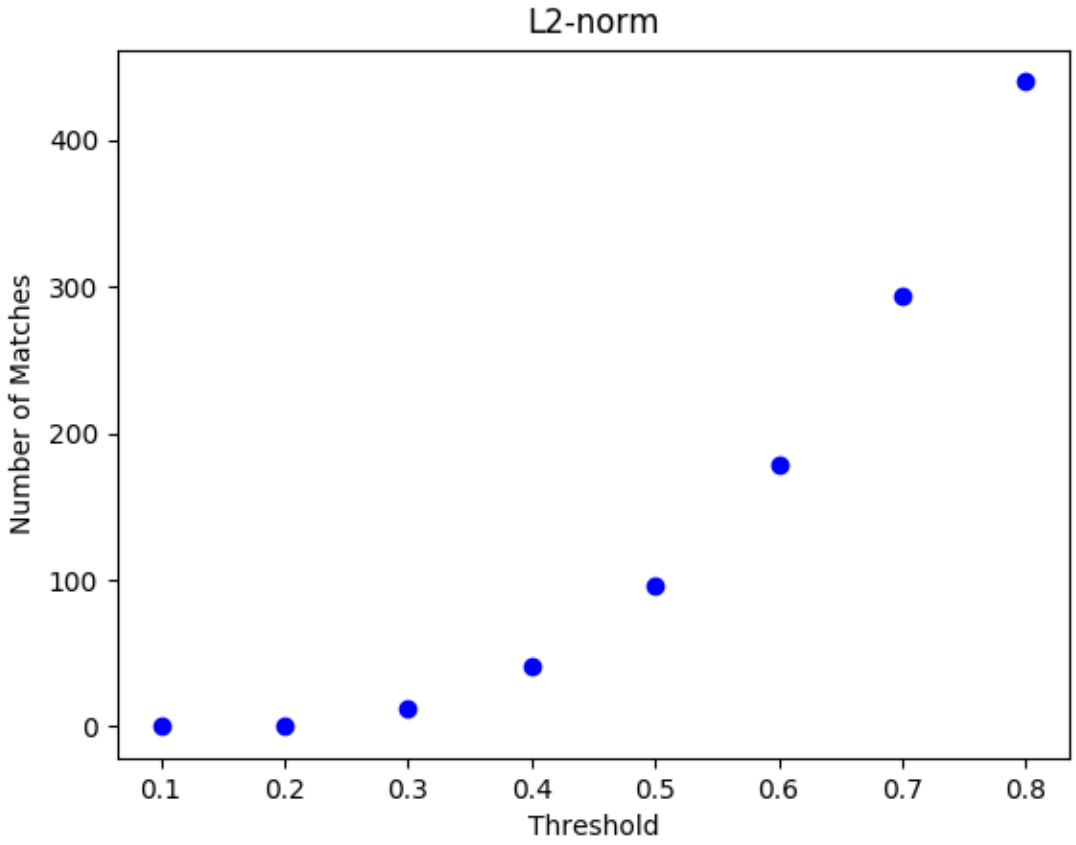




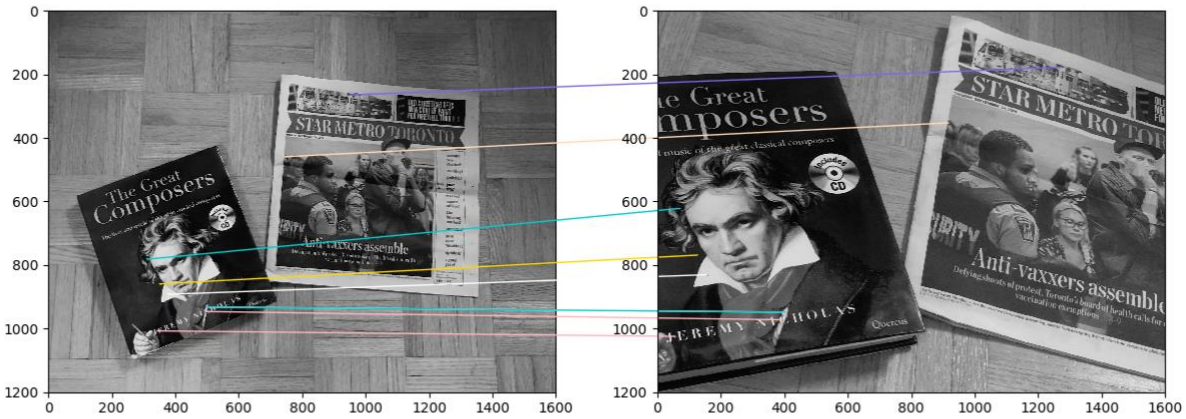


Number of Matches (Please know, since the code execution on my laptop is very long for Q4, to make sure I can finish this assignment on time, I set the limitation as 2000 during sift initialization so that the running can be faster. The results of number of matches and the matching will be slightly less powerful than normal setting. However, the tendency is the same.):

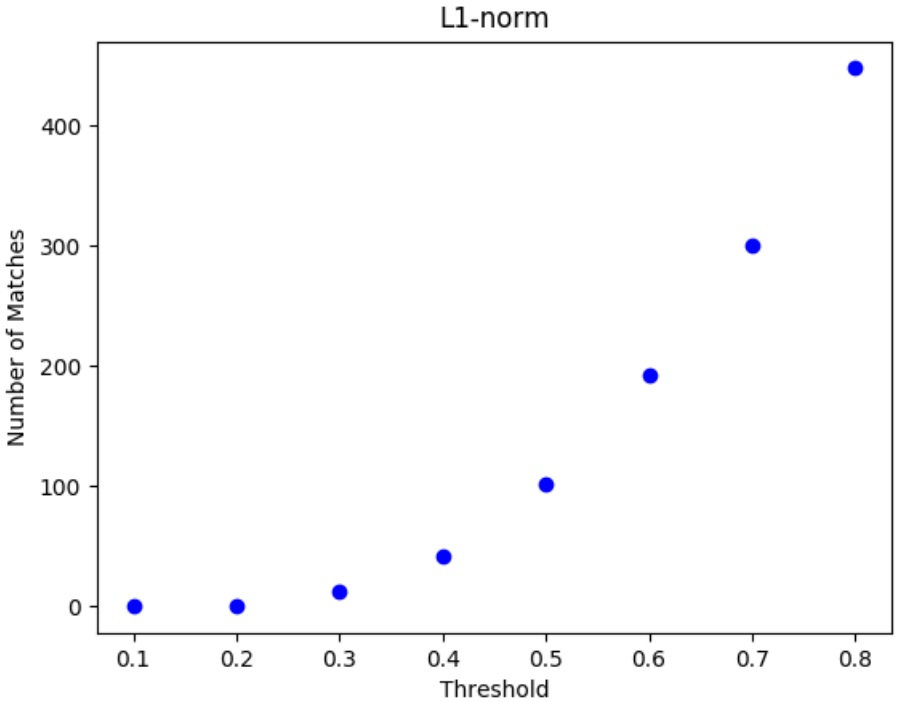
```python
def sift_extract(image):
    img = cv2.imread(image)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Initialized sift
    sift = cv2.xfeatures2d.SIFT_create(2000)
    flag = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS

    # Get keypoints and features
    keypoints, features = sift.detectAndCompute(gray_img, None)
    result = cv2.drawKeypoints(gray_img, keypoints, flags=flag, outImage=None)

    cv2.imwrite('sift_keypoints.jpg', result)
    return keypoints, features
```



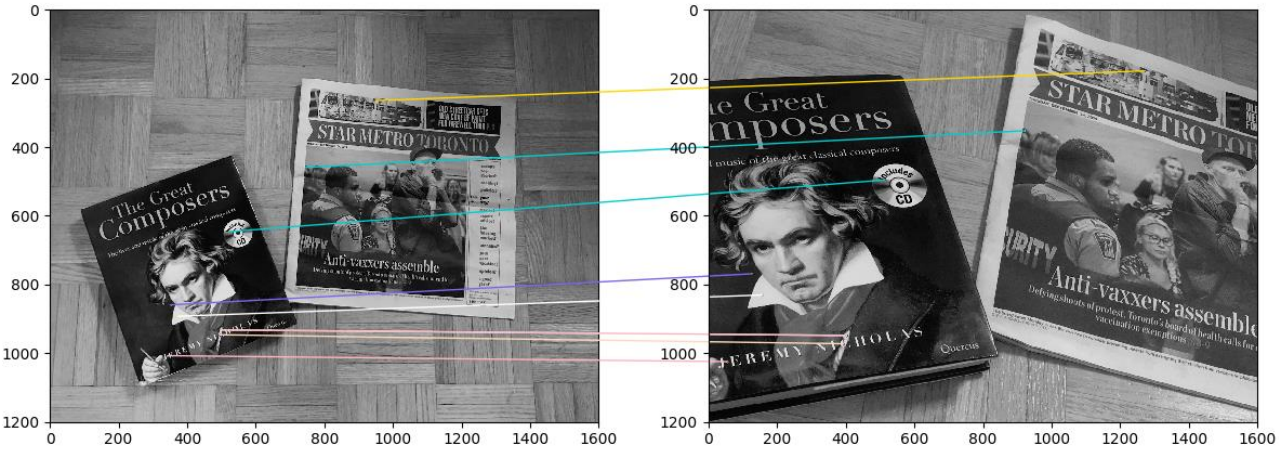
L2: Top 10 matches at best threshold 0.8

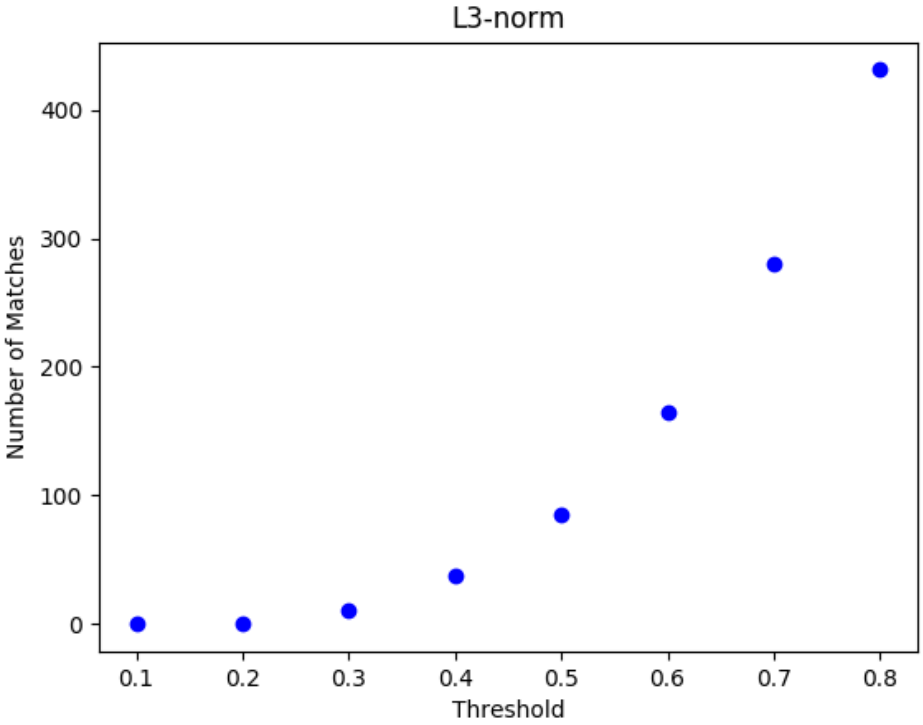


Q4.(c).
Number of Matches (Please know, since the code execution on my laptop is very long for Q4, to make sure I can finish this assignment on time, I set the limitation as 2000 during sift initialization so that the running can be faster. The results of number of matches and the matching will be slightly less powerful than normal setting. However, the tendency is the same.):

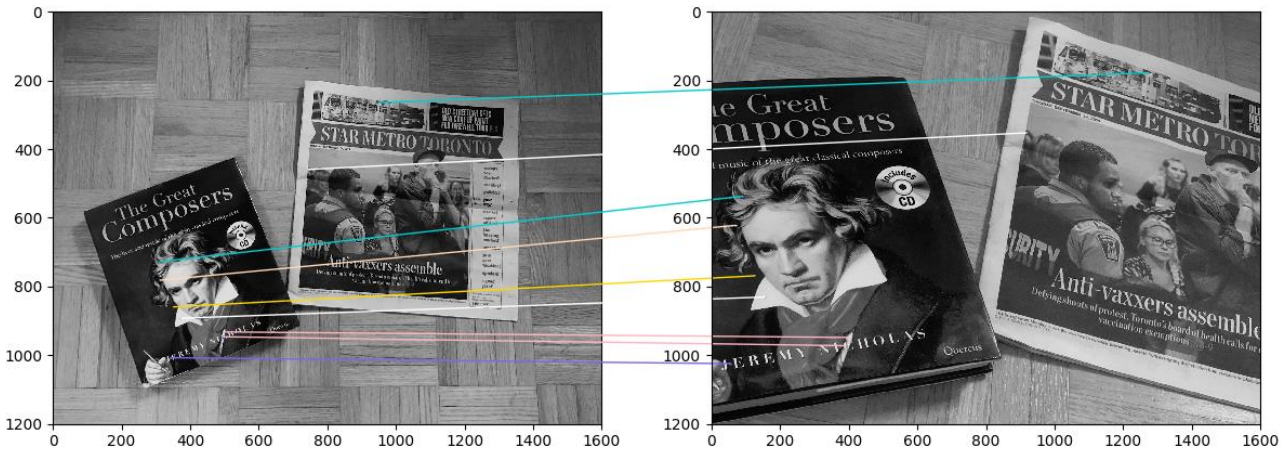


L1: Top 10 matches at best threshold 0.8:





L3: Top 10 matches at best threshold 0.8:

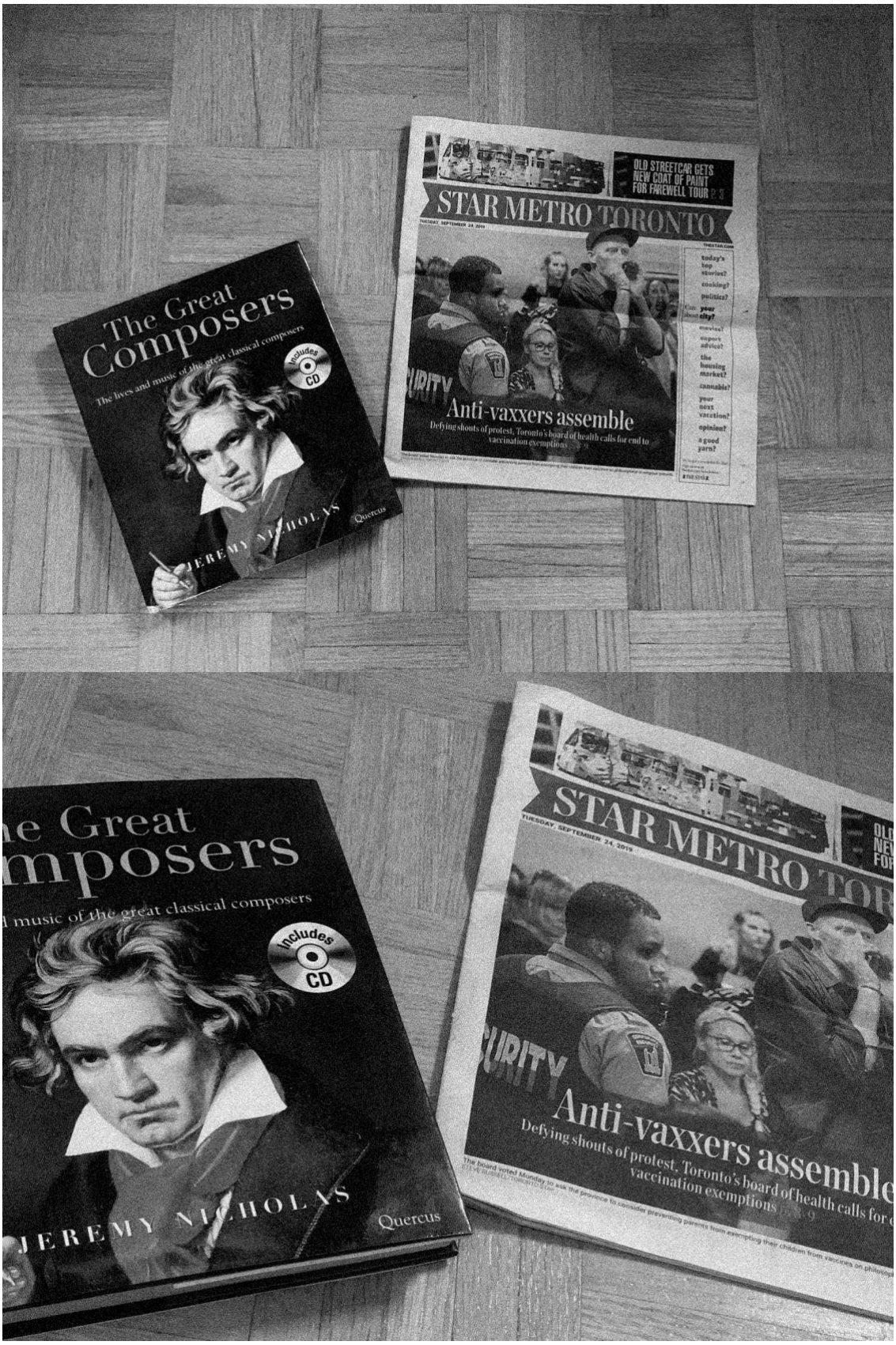


Based on the observation, there is no big difference among L1, L2, L3. We may need more pictures to actually find the differences under varied situations, however, I think the differences won't be large.

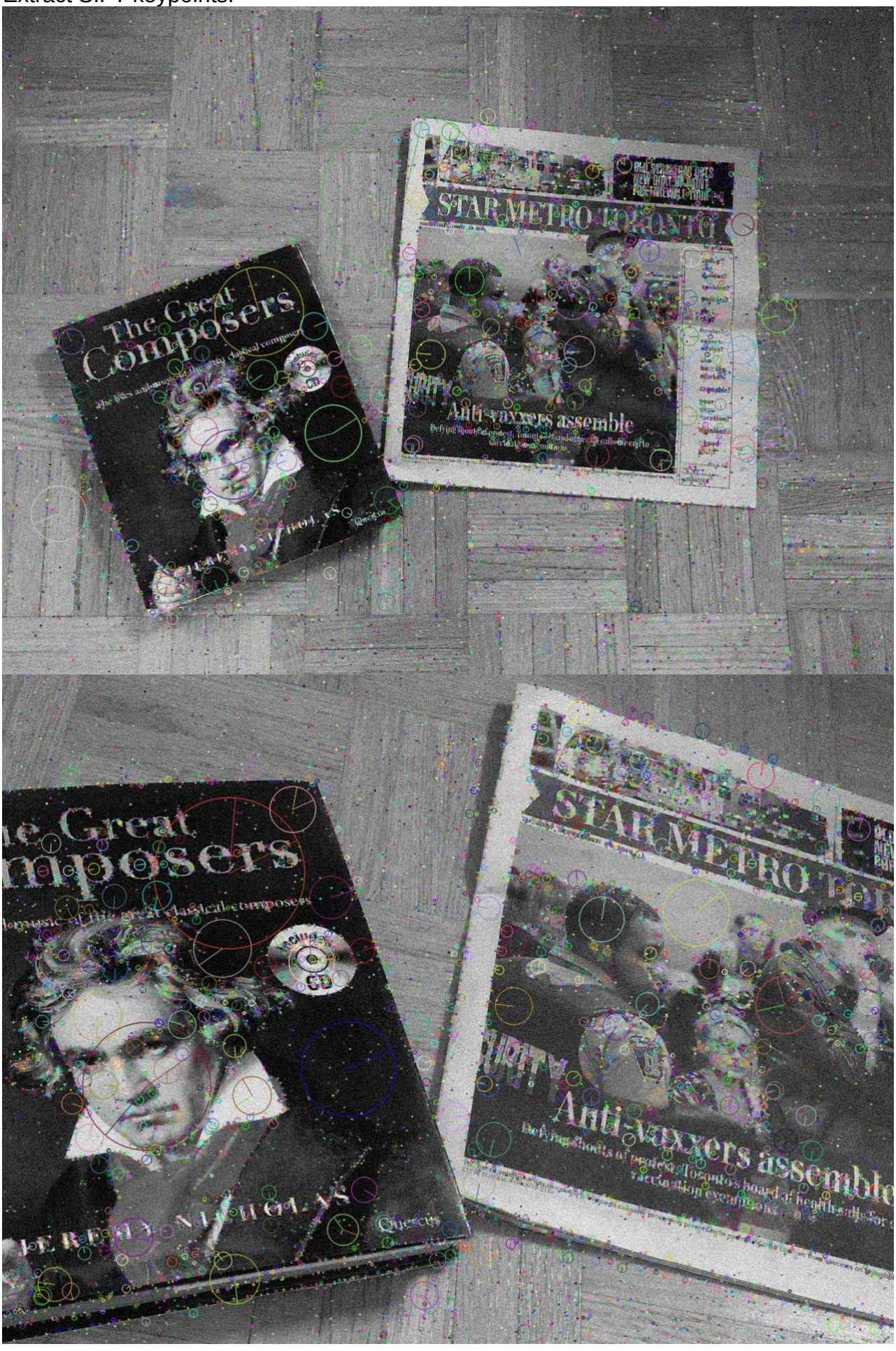
Q4.(d).

```
def AddGaussianNoise(image):

    img = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
    mean = 0
    sigma = 0.08
    temp = img / 255

    gaussian = np.random.normal(mean, sigma, size=temp.shape)  # np.zeros((224, 224), np.float32)

    noisy_image = temp + gaussian

    return noisy_image * 255
```

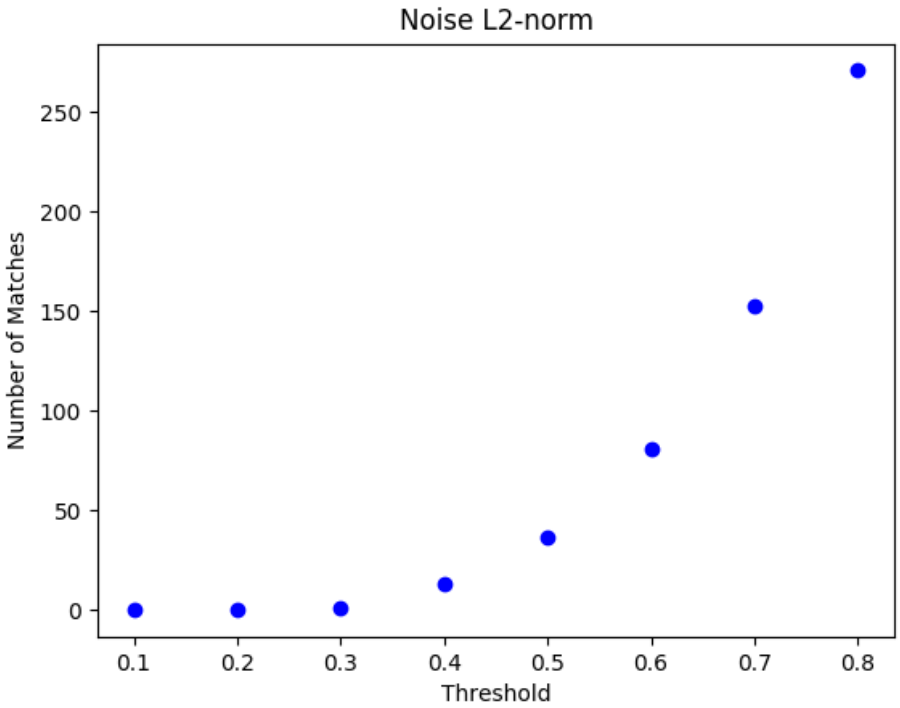
Added noises:



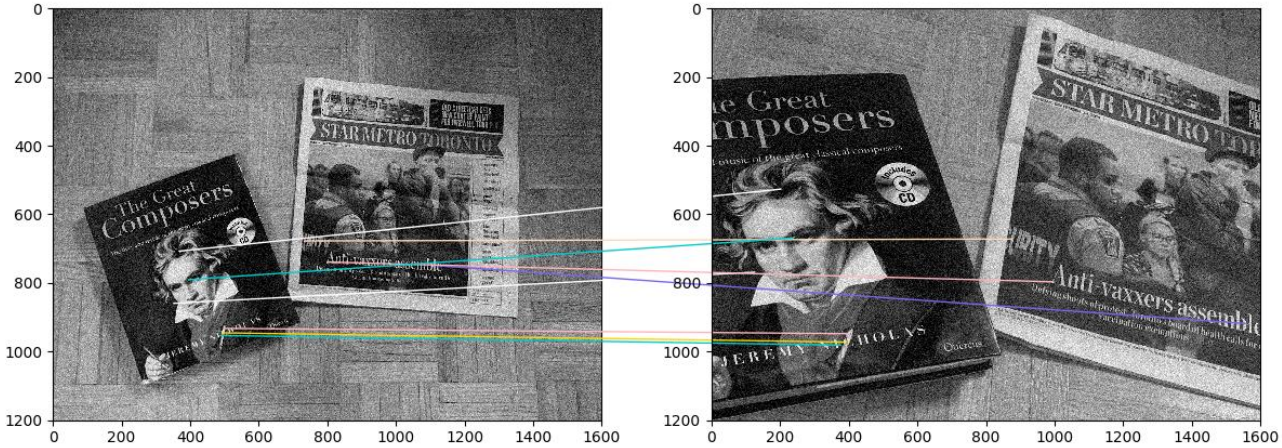
Extract SIFT keypoints:



Number of Matches (Please know, since the code execution on my laptop is very long for Q4, to make sure I can finish this assignment on time, I set the limitation as 2000 during sift initialization so that the running can be faster. The results of number of matches and the matching will be slightly less powerful than normal setting. However, the tendency is the same.):



Top 10 matches at best threshold 0.8:



As you see, the noise decreased the number of matches we found, which is because during the keypoints detection, SIFT mistakenly treat many noises as keypoints, since they are similar to corners. Those wrong keypoints affect the judgement of matching as well. The less number of matches we have, the less matchings we can do.

Q4.(e).

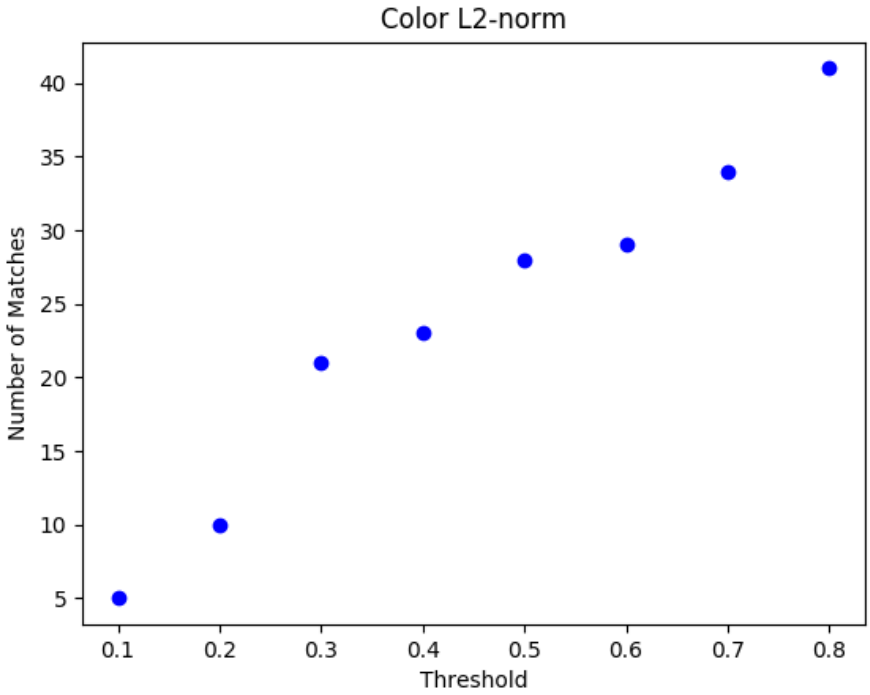
```
509     def sift_matching_color(image1, image2, threshold):
510         img1 = cv2.imread(image1)
511         img2 = cv2.imread(image2)
512 
513         # Only use h of hsv (h include all colour information)
514         img1_b = img1[0:img1.shape[0], 0:img1.shape[1], 0]
515         img2_b = img2[0:img2.shape[0], 0:img2.shape[1], 0]
516 
517         img1_g = img1[0:img1.shape[0], 0:img1.shape[1], 1]
518         img2_g = img2[0:img2.shape[0], 0:img2.shape[1], 1]
519 
520         img1_r = img1[0:img1.shape[0], 0:img1.shape[1], 2]
521         img2_r = img2[0:img2.shape[0], 0:img2.shape[1], 2]
522 
523         kp_1, des_1 = sift_extract(image1)
524         kp_2, des_2 = sift_extract(image2)
525 
526 
527         # Initialize sift
528         sift = cv2.xfeatures2d.SIFT_create(2000)
529 
530         kp_1, des_1b = sift.compute(img1_b, kp_1)
531         kp_2, des_2b = sift.compute(img2_b, kp_2)
532         euclidean_dis_b = euclidean_distance2(des_1b, des_2b)
533 
534         kp_1, des_1g = sift.compute(img1_g, kp_1)
535         kp_2, des_2g = sift.compute(img2_g, kp_2)
536         euclidean_dis_g = euclidean_distance2(des_1g, des_2g)
537 
538         kp_1, des_1r = sift.compute(img1_r, kp_1)
539         kp_2, des_2r = sift.compute(img2_r, kp_2)
540         euclidean_dis_r = euclidean_distance2(des_1r, des_2r)
541 
542 
543         # find minimum distance index in keypoint 2
544         min_keypoints_index_b = np.argmin(euclidean_dis_b, axis=1)
545         min_keypoints_index_g = np.argmin(euclidean_dis_g, axis=1)
546         min_keypoints_index_r = np.argmin(euclidean_dis_r, axis=1)
547 
```

```
548
549     num_matching_keypoint_b = len(min_keypoints_index_b)
550     num_matching_keypoint_g = len(min_keypoints_index_g)
551     num_matching_keypoint_r = len(min_keypoints_index_r)
552
553     # find smallest and 2nd smallest distance
554     smallest_b = np.sort(euclidean_dis_b, axis=1)[:, 0]
555     second_smallest_b = np.sort(euclidean_dis_b, axis=1)[:, 1]
556
557     smallest_g = np.sort(euclidean_dis_g, axis=1)[:, 0]
558     second_smallest_g = np.sort(euclidean_dis_g, axis=1)[:, 1]
559
560     smallest_r = np.sort(euclidean_dis_r, axis=1)[:, 0]
561     second_smallest_r = np.sort(euclidean_dis_r, axis=1)[:, 1]
562
563     # compute ratio
564     ratio_b = np.true_divide(smallest_b, second_smallest_b)
565     ratio_g = np.true_divide(smallest_g, second_smallest_g)
566     ratio_r = np.true_divide(smallest_r, second_smallest_r)
567
568     keypoint1 = []
569     keypoint2 = []
570     matched_ratio = []
571     keypoints_pairs = {}
572     count = 0
573
574     for i in range(num_matching_keypoint_b):
575         print(ratio_b[i])
576         if ratio_b[i] < threshold:
577             count = count + 1
578             kp1 = kp_1[i]
579             kp2 = kp_2[min_keypoints_index_b[i]]
580             keypoint1.append(kp1)
581             keypoint2.append(kp2)
582             matched_ratio.append(ratio_b[i])
583             keypoints_pairs[(kp_1[i], kp2)] = ratio_b[i]
```

```
    for i in range(num_matching_keypoint_g):
        if ratio_g[i] < threshold:
            count = count + 1
            kp1 = kp_1[i]
            kp2 = kp_2[min_keypoints_index_g[i]]
            keypoint1.append(kp1)
            keypoint2.append(kp2)
            matched_ratio.append(ratio_g[i])
            keypoints_pairs[(kp_1[i], kp2)] = ratio_g[i]

    for i in range(num_matching_keypoint_r):
        if ratio_r[i] < threshold:
            count = count + 1
            kp1 = kp_1[i]
            kp2 = kp_2[min_keypoints_index_r[i]]
            keypoint1.append(kp1)
            keypoint2.append(kp2)
            matched_ratio.append(ratio_r[i])
            keypoints_pairs[(kp_1[i], kp2)] = ratio_r[i]

    sorted_keypointes = sorted(keypoints_pairs.items(), key=lambda kv: kv[1])

    print("number of matching = ", count)

    top_ten = sorted_keypointes[0:9]
    print(top_ten)

    flag = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS

    result_img1 = cv2.drawKeypoints(img1, keypoint1, outImage=None, flags=flag)
    result_img2 = cv2.drawKeypoints(img2, keypoint2, outImage=None, flags=flag)
    cv2.imwrite("sample1_keypoints_final_color.jpg", result_img1)
    cv2.imwrite("sample2_keypoints_final_color.jpg", result_img2)
    show_matching(img1, keypoint1, img2, keypoint2)

    return top_ten

def euclidean_distance2(vector1, vector2):
    vector1_square = np.sum(np.square(vector1), axis=1).reshape(-1, 1)
    vector2_square = np.sum(np.square(vector2), axis=1).reshape(-1, 1)
    vector1_vector2 = np.dot(vector1, np.transpose(vector2))

    distance = np.sqrt(vector1_square - 2 * vector1_vector2 +
                       np.transpose(vector2_square))

    return distance
```

The algorithm is similar to Q4b, but with extra R, G, and B channels. Keypoints are extracted from grayscale image, then compute three descriptors separately from R, G, B channels. Then perform sift match on these three descriptors to calculate new matched keypoints.

Number of Matches (Please know, since the code execution on my laptop is very long for Q4, to make sure I can finish this assignment on time, I set the limitation as 2000 during sift initialization so that the running can be faster. The results of number of matches and the matching will be slightly less powerful than normal setting. However, the tendency is the same.):



Top 10 matches at best threshold 0.8: