CSC420 Assignment 1

Chen Liang

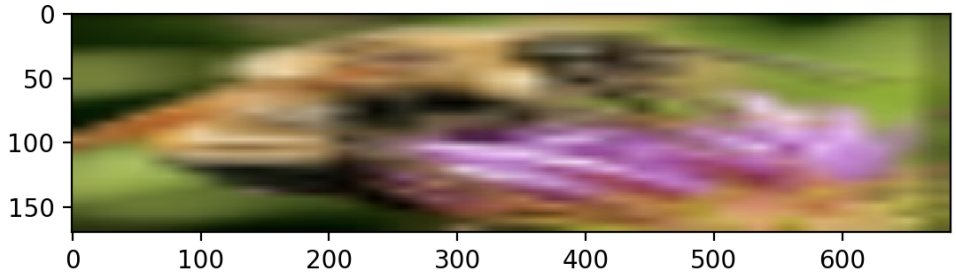
Oct, 2019

Question 1

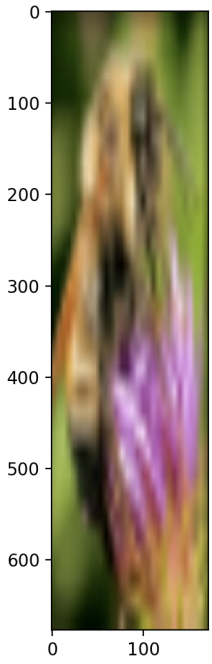
(a) For this question, I quadruple every single row, then quadruple every single column, and the resulting image is 16 times the size of the original image. Below is the code how I implemented 1D upsampling on every single row, and every single column:

```
  def quadruple_image_size_by_column(input_image, total_col_num, total_row_num):
    result_image = np.zeros((total_row_num, total_col_num * 4 - 3))
    for row_num in range(total_row_num):
        whole_row = input_image[row_num, :]
        result_image[row_num, :] = upsample_one_dimension(whole_row, total_col_num)
    return result_image


def quadruple_image_size_by_row(input_image, total_col_num, total_row_num):
    result_image = np.zeros((total_row_num * 4 - 3, total_col_num))
    for col_num in range(total_col_num):
        whole_col = input_image[:, col_num]
        result_image[:, col_num] = upsample_one_dimension(whole_col, total_row_num)
    return result_image


def upsample_one_dimension(pixel_list, num):
    result_pixel_list = [None] * (num * 4 - 3)
    for i in range(num - 1):
        pix1 = pixel_list[i]
        pix2 = pixel_list[i + 1]
        result_pixel_list[i * 4] = pix1
        result_pixel_list[i * 4 + 4] = pix2
        result_pixel_list[i * 4 + 1] = pix1 + (pix2 - pix1) / 4
        result_pixel_list[i * 4 + 2] = pix1 + (pix2 - pix1) / 2
        result_pixel_list[i * 4 + 3] = pix1 + 3 * (pix2 - pix1) / 4
    return result_pixel_list
```

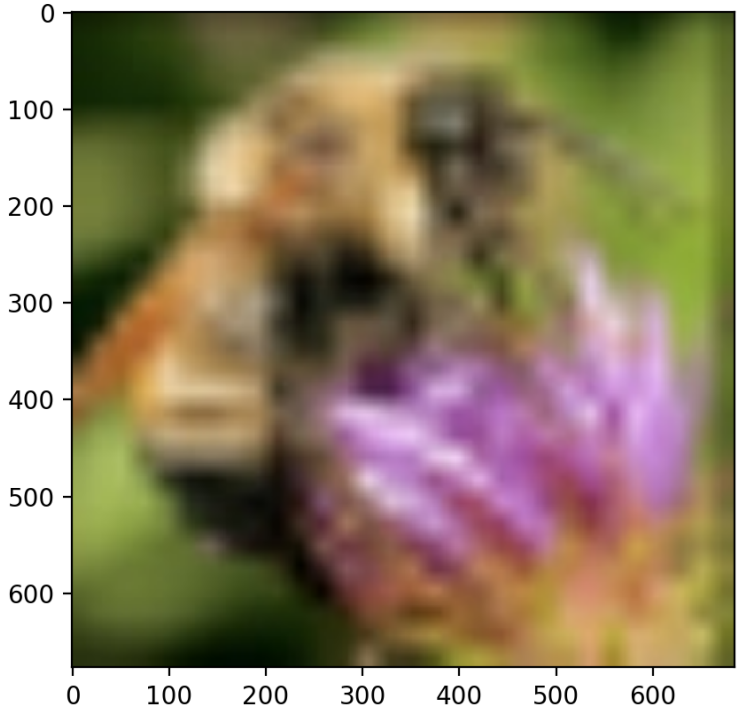
The resulting image for upsampling on every single row looks like:



and the resulting image for upsampling on every single column looks like:



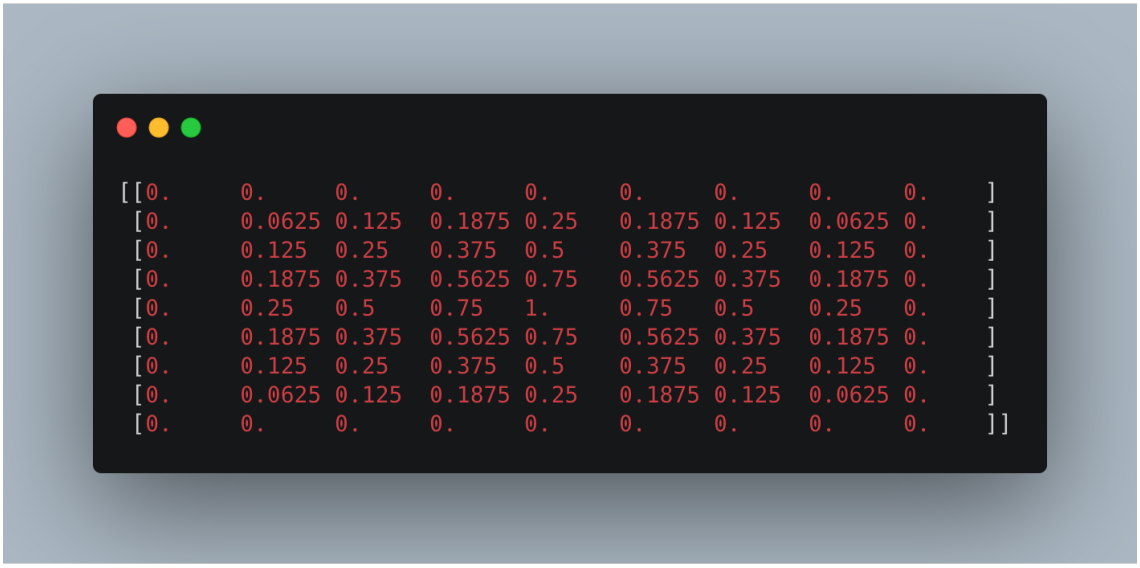
The resulting image for upsampling every single row after upsampling every single column looks like:



Below is the code in main function:

```python
if __name__ == '__main__':
    img = cv2.imread("bee.jpg").astype(np.float32)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img0 = quadruple_image_size_by_column(img[:, :, 0], img.shape[1], img.shape[0])
    img1 = quadruple_image_size_by_column(img[:, :, 1], img.shape[1], img.shape[0])
    img2 = quadruple_image_size_by_column(img[:, :, 2], img.shape[1], img.shape[0])
    result_image_col = np.zeros((img.shape[0], img.shape[1] * 4 - 3, img.shape[2]))
    result_image_col[:, :, 0] = img0
    result_image_col[:, :, 1] = img1
    result_image_col[:, :, 2] = img2
    result_image_col = result_image_col.astype(np.int32)
    plt.imshow(result_image_col)
    plt.show()

    img3 = quadruple_image_size_by_row(img[:, :, 0], img.shape[1], img.shape[0])
    img4 = quadruple_image_size_by_row(img[:, :, 1], img.shape[1], img.shape[0])
    img5 = quadruple_image_size_by_row(img[:, :, 2], img.shape[1], img.shape[0])
    result_image_row = np.zeros((img.shape[0] * 4 - 3, img.shape[1], img.shape[2]))
    result_image_row[:, :, 0] = img3
    result_image_row[:, :, 1] = img4
    result_image_row[:, :, 2] = img5
    result_image_row = result_image_row.astype(np.int32)
    plt.imshow(result_image_row)
    plt.show()

    img5 = quadruple_image_size_by_row(img0, img0.shape[1], img0.shape[0])
    img6 = quadruple_image_size_by_row(img1, img1.shape[1], img1.shape[0])
    img7 = quadruple_image_size_by_row(img2, img2.shape[1], img2.shape[0])
    result_image = np.zeros((img0.shape[0] * 4 - 3, img0.shape[1], 3))
    result_image[:, :, 0] = img5
    result_image[:, :, 1] = img6
    result_image[:, :, 2] = img7
    result_image = result_image.astype(np.int32)
    plt.imshow(result_image)
    plt.show()
```

If we apply 2D filter, the 2D filter looks like below:

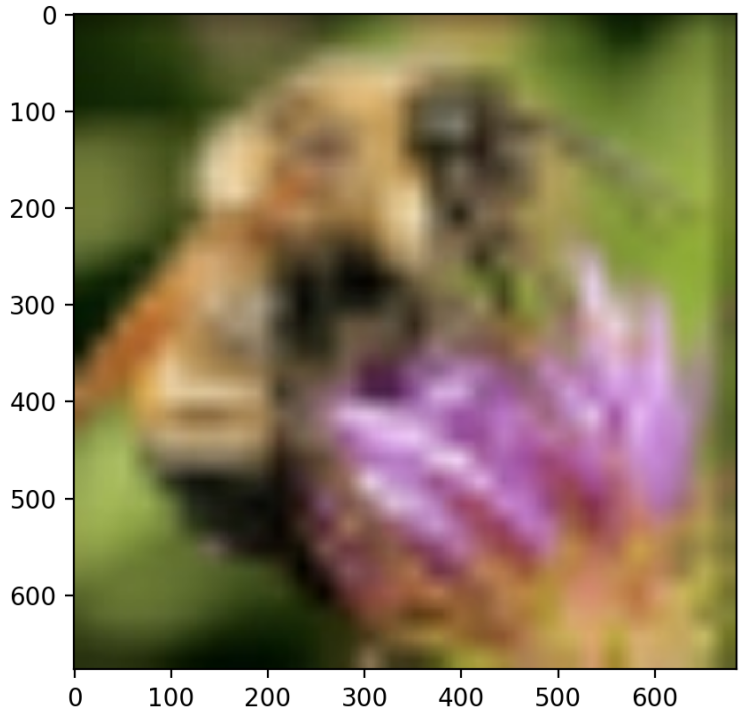


```
[[0.     0.     0.     0.     0.     0.     0.     0.     0.    ]
 [0.     0.0625 0.125  0.1875 0.25   0.1875 0.125  0.0625 0.    ]
 [0.     0.125  0.25   0.375  0.5    0.375  0.25   0.125  0.    ]
 [0.     0.1875 0.375  0.5625 0.75   0.5625 0.375  0.1875 0.    ]
 [0.     0.25   0.5    0.75   1.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

(b) Below is the code for 2-dimensional interpolation:

```
def apply_2D_filter(input_image, total_row_num, total_col_num):
    result_image = np.zeros((total_row_num * 4 - 3, total_col_num * 4 - 3))
    for i in range(total_row_num * 4 - 3):
        for j in range(total_col_num * 4 - 3):
            if i % 4 == 0 and j % 4 == 0:
                result_image[i][j] = input_image[i // 4][j // 4]
    for k in range(total_row_num * 4 -3):
        for l in range(total_col_num * 4 - 3):
            if k % 4 == 0 and l % 4 != 0:
                t = l / 4
                s = int(l/4)
                result_image[k][l] = (s - t + 1) * input_image[k // 4][s] + (t - s) *
input_image[k // 4][s + 1]
    for u in range(total_row_num * 4 - 3):
        for v in range(total_col_num * 4 - 3):
            if u % 4 != 0:
                x = (u // 4) * 4
                result_image[u][v] = (x + 4 - u) * result_image[x][v] / 4 + (u - x) *
result_image[x + 4][v]/4
    return result_image
```

And the resulting image looks like:



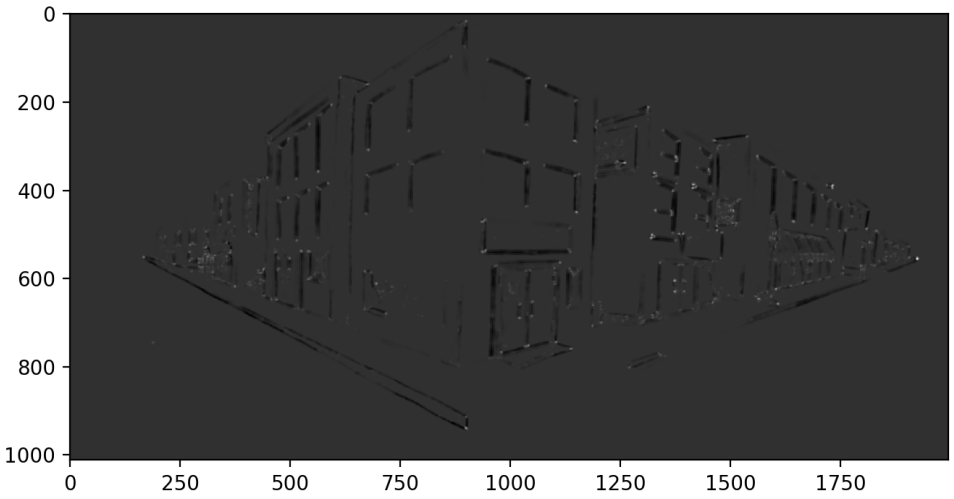
From my observation, the resulting image looks the same as the result of applying 1D interpolation twice.

Question 2

(a) Below is the code for Harris corner detection and Brown corner detection.

```python
def calculate_components(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) * 0.5
    blur = cv2.GaussianBlur(gray, (5, 5), 7)
    Ix = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=5)
    Iy = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=5)
    IxIy = np.multiply(Ix, Iy)
    Ix2 = np.multiply(Ix, Ix)
    Iy2 = np.multiply(Iy, Iy)
    Ix2_blur = cv2.GaussianBlur(Ix2, (7, 7), 10)
    Iy2_blur = cv2.GaussianBlur(Iy2, (7, 7), 10)
    IxIy_blur = cv2.GaussianBlur(IxIy, (7, 7), 10)
    return Ix2_blur, Iy2_blur, IxIy_blur


def get_trace_and_determinant(img):
    Ix2_blur, Iy2_blur, IxIy_blur = calculate_components(img)
    det = np.multiply(Ix2_blur, Iy2_blur) - np.multiply(IxIy_blur, IxIy_blur)
    trace = Ix2_blur + Iy2_blur
    return trace, det


def harris_corner_detector(img, alpha):
    trace, det = get_trace_and_determinant(img)
    return det - alpha * np.multiply(trace, trace)


def brown_corner_detector(img):
    res = np.zeros((img.shap[0], img.shape[0]))
    trace, det = get_trace_and_determinant(img)
    for i in range(trace.shape[0]):
        for j in range(trace.shape[1]):
            if trace[i][j] == 0:
                res[i][j] = 255
            else:
                res[i][j] = det[i][j] / trace[i][j]
    return res
```

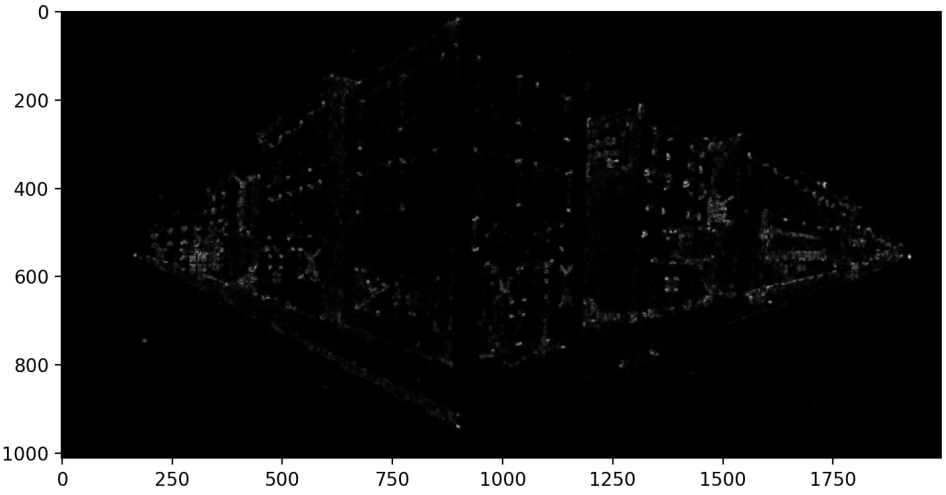
And the result for Harris corner detection looks like below.



For Harris Corner detection, we need to prune the alpha value to 0.04 to 0.06. And In our case, we set the alpha value to 0.05.

```python
if __name__ == '__main__':
    img = cv2.imread("building.jpg")
    img2 = harris_corner_detector(img, alpha=0.05)
    img3 = brown_corner_detector_by_eigenvalue(img)
    plt.imshow(img2, cmap='gray')
    plt.show()
    plt.imshow(img3, cmap='gray')
    plt.show()
    img4 = harris_corner_detector_by_eigenvalue(img, alpha=0.05)
    img5 = brown_corner_detector_by_eigenvalue(img)
    plt.imshow(img4, cmap='gray')
    plt.show()
    plt.imshow(img5, cmap='gray')
    plt.show()
```

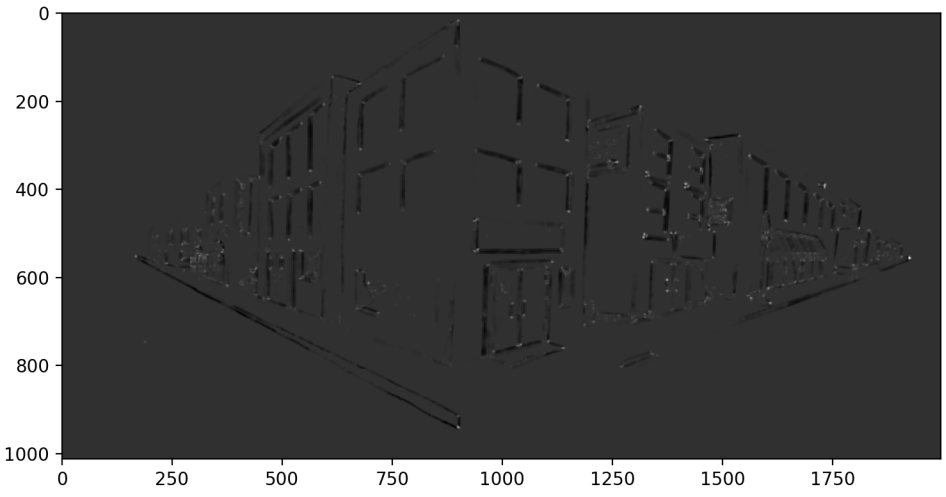
The result for Brown corner detection looks like below.

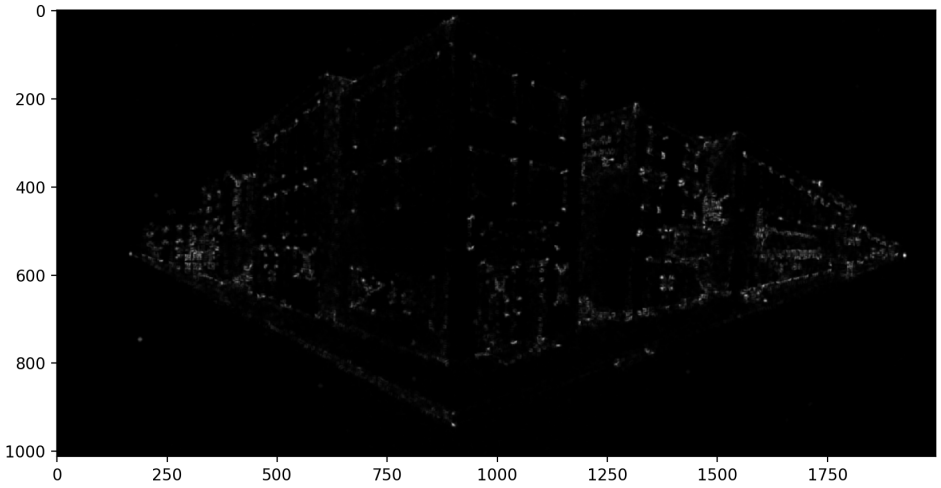


Yes, we can use eigenvalues to detect corners. And below is the code:

```python
def calculate_components(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) * 0.5
    blur = cv2.GaussianBlur(gray, (5, 5), 7)
    Ix = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=5)
    Iy = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=5)
    IxIy = np.multiply(Ix, Iy)
    Ix2 = np.multiply(Ix, Ix)
    Iy2 = np.multiply(Iy, Iy)
    Ix2_blur = cv2.GaussianBlur(Ix2, (7, 7), 10)
    Iy2_blur = cv2.GaussianBlur(Iy2, (7, 7), 10)
    IxIy_blur = cv2.GaussianBlur(IxIy, (7, 7), 10)
    return Ix2_blur, Iy2_blur, IxIy_blur

def harris_corner_detector_by_eigenvalue(img, alpha):
    Ix2_blur, Iy2_blur, IxIy_blur = calculate_components(img)
    result = np.zeros(Ix2_blur.shape)
    for i in range(Ix2_blur.shape[0]):
        for j in range(Ix2_blur.shape[1]):
            M = np.array([[Ix2_blur[i, j], IxIy_blur[i, j]], [IxIy_blur[i, j], Iy2_blur[i,
j]]])
            eigenvalues, _ = np.linalg.eig(M)
            result[i, j] = eigenvalues[0] * eigenvalues[1] - alpha * (eigenvalues[0] +
eigenvalues[1]) ** 2
    return result


def brown_corner_detector_by_eigenvalue(img):
    Ix2_blur, Iy2_blur, IxIy_blur = calculate_components(img)
    result = np.zeros(Ix2_blur.shape)
    for i in range(Ix2_blur.shape[0]):
        for j in range(Ix2_blur.shape[1]):
            M = np.array([[Ix2_blur[i, j], IxIy_blur[i, j]], [IxIy_blur[i, j], Iy2_blur[i,
j]]])
            eigenvalue, _ = np.linalg.eig(M)
            denominator = eigenvalue[0] + eigenvalue[1]
            if denominator == 0:
                result[i][j] = 255
            else:
                result[i][j] = eigenvalue[0] * eigenvalue[1] / denominator
    return result
```

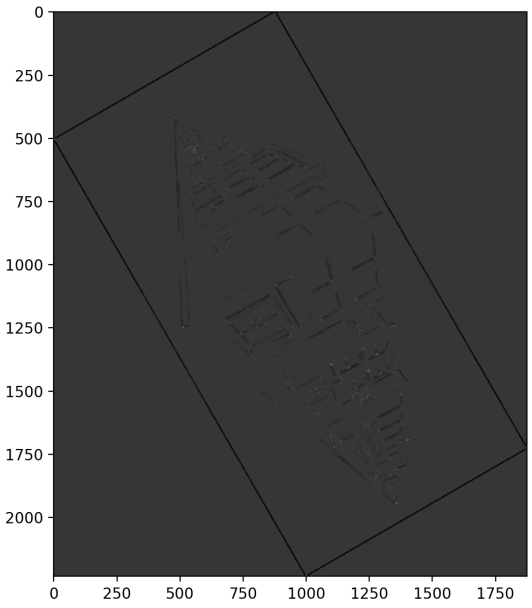
Below are resulting images:





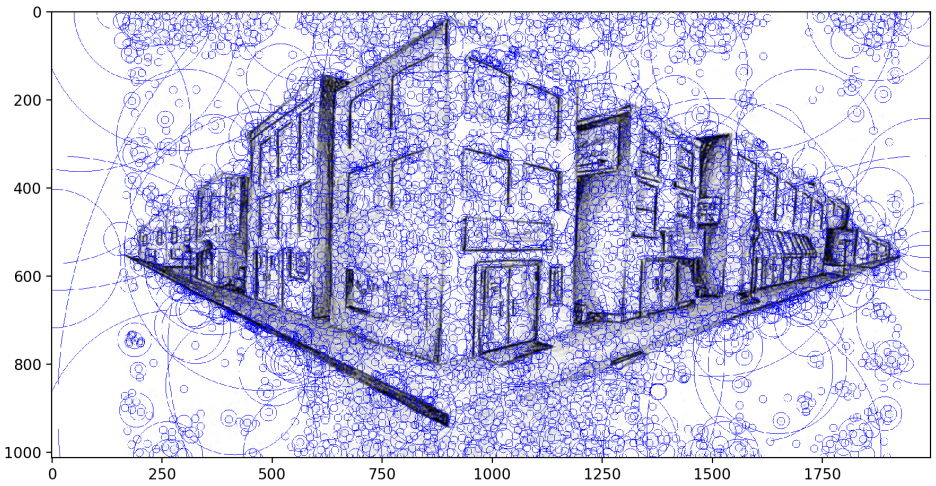
(b) Here's the code that rotate the image, and the result looks like below:

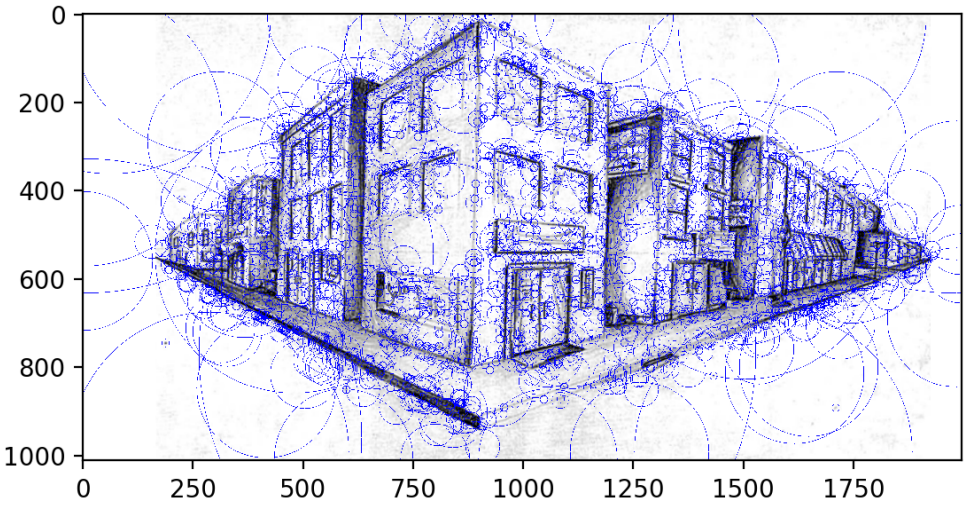
```python
def rotate_image(img):
    rotated = imutils.rotate_bound(img, 60)
    return rotated

def harris_corner_after_rotation(img, alpha):
    rotated_img = rotate_image(img)
    return harris_corner_detector(rotated_img, alpha)
```

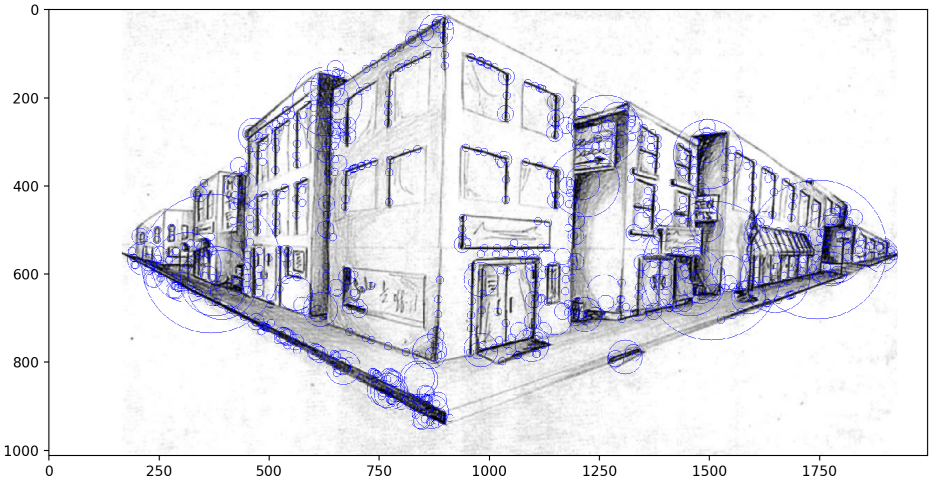


```python
def non_maxima_suppression(img, pyramid, threshold, sigma_list):
    maxima_list = []
    for i in range(1, 4):
        upper = pyramid[i-1]
        current = pyramid[i]
        lower = pyramid[i+1]

        upper_pad = np.zeros((img.shape[0] + 2, img.shape[1] + 2))
        upper_pad[1: 1 + img.shape[0], 1: 1 + img.shape[1]] = upper[0: img.shape[0], 0:
img.shape[1]]

        current_pad = np.zeros((img.shape[0] + 2, img.shape[1] + 2))
        current_pad[1: 1 + img.shape[0], 1: 1 + img.shape[1]] = current[0: img.shape[0], 0:
img.shape[1]]

        lower_pad = np.zeros((img.shape[0] + 2, img.shape[1] + 2))
        lower_pad[1: 1 + img.shape[0], 1: 1 + img.shape[1]] = lower[0: img.shape[0], 0:
img.shape[1]]

        patch_size = 3

        for row in range(img.shape[0]):
            for col in range(img.shape[1]):
                upper_sub = np.array(upper_pad[row: row + patch_size, col: col + patch_size])
                current_sub = np.array(current_pad[row: row + patch_size, col: col +
patch_size])
                lower_sub = np.array(lower_pad[row: row + patch_size, col: col + patch_size])

                cur_max = np.concatenate((upper_sub, current_sub), axis=0)
                cur_max = np.concatenate((cur_max, lower_sub), axis=0)
                max_value = np.max(cur_max)

                if max_value == current_sub[1, 1] and np.percentile(cur_max, threshold) <
current_sub[1, 1]:
                    maxima_list.append(((row, col), sigma_list[i]))
    return maxima_list


def mySift(threshold, img):
    pyramid = [np.zeros(img.shape)]
    sigma_list = [0]
    sigma = 1.6
    scaling_factor = 1

    for octave in range(5):
        gaussian_img = cv2.GaussianBlur(img, ksize=(0, 0),
                                        sigmaX=sigma * scaling_factor, sigmaY=sigma *
scaling_factor)
        laplacian_img = np.abs(cv2.Laplacian(gaussian_img, ddepth=cv2.CV_32F, ksize=5,
scale=1))
        pyramid.append(laplacian_img)
        scaling_factor = scaling_factor * np.sqrt(2)
        sigma_list.append(scaling_factor)

    pyramid.append(np.zeros(img.shape))

    max_list = non_maxima_suppression(img, pyramid, threshold, sigma_list)

    for data in max_list:
        cv2.circle(img, (data[0][1], data[0][0]), input(np.round(data[1])), color=(255, 0, 0))


if __name__ == '__main__':
    img = cv2.imread('bee.jpg')
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).astype(np.float)
    mySift(90, img)
```

Below are resulting images with thresholds at value of 50, 70 and 90 respectively.

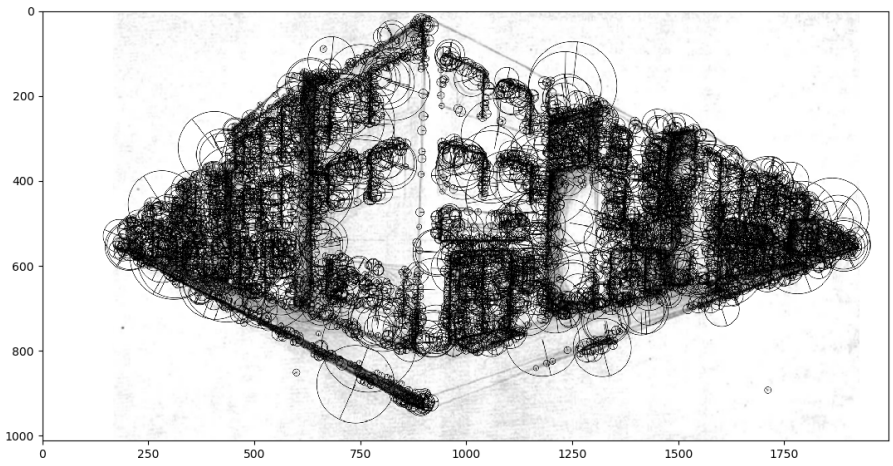



0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750

(d) I choose SURF descriptor. Idea: SURF method uses laplacian of gaussian to detect corner. The laplacian of gaussian is computed by convolution with a box filter with the the help of the integral images: $S(x,y) = \sum_{i=0}^{x} \sum_{i=0}^{y} I(i,j)$ The keypoint is detected with similar method to blob detection, with the help of hessian matrix, where the determinant of the hessian matrix is being evaluated as the measure of the local changes around the point. Non maximum suppression is also used in SURF, where the point with maximum value of hessian determinant in its neighbour is being chosen as keypoint.

```python
def SURF_feature_discriptor(img):
    surf = cv2.xfeatures2d.SURF_create(1000)
    keypoints, _ = surf.detectAndCompute(img, None)
    img2 =cv2.drawKeypoints(img, keypoints, np.array([]), (0,0,0),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    plt.imshow(img2, cmap='gray')
    plt.show()


if __name__ == '__main__':
    img = cv2.imread('building.jpg', 0)
    SURF_feature_discriptor(img)
```

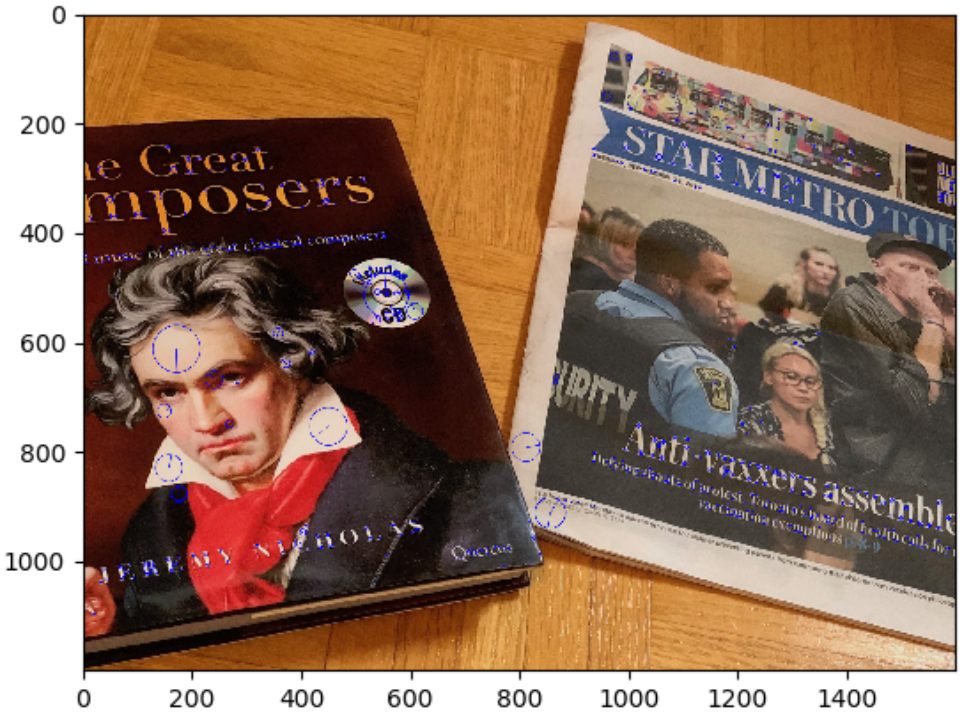


Question 3

(a) Derive closed form expression:
$\nabla^2 g(x,y,\sigma) = \frac{\partial^2 g(x,y,\sigma)}{\partial x^2} + \frac{\partial^2 g(x,y,\sigma)}{\partial y^2} = (\frac{x^2}{2\pi\sigma^6} - \frac{1}{2\pi\sigma^4})e^{-\frac{x^2+y^2}{2\sigma^2}} + (\frac{x^2}{2\pi\sigma^6} - \frac{1}{2\pi\sigma^4})e^{-\frac{x^2+y^2}{2\sigma^2}} = -\frac{1}{\pi\sigma^4}(1 - \frac{x^2+y^2}{2\sigma^2})e^{\frac{x^2+y^2}{2\sigma^2}}$.
It's not separable. Intuitively, we cannot break this expression into the product of two expressions only containing parameter x and y respectively, and more mathematically, by using Question 5 in A1, we could know that this filter is not separable.
(b) For σ_1, σ_2 that are really close, we could use approximation, $\frac{\partial g}{\partial \sigma} = \frac{\partial g(x,y,\sigma)}{\partial \sigma} \approx \frac{g(x,y,\sigma_1 - g(x,y,\sigma_2)}{\sigma_1 - \sigma_2}$. Recall steps on how to build octaves for gaussian pyramid we that for two consecutive layers of images in one octave, the upper layer has an blur width as σ value while the lower layer has value as $k\sigma$, and set into this equation we have $\frac{\partial g}{\partial \sigma} = \sigma\nabla^2 g \approx \frac{g(x,y,k\sigma)-g(x,y,\sigma)}{k\sigma-\sigma}$. In this way, we could conclude that $DoG = g(x,y,k\sigma) - g(x,y,\sigma) \approx \sigma\nabla^2 g \cdot (k\sigma - \sigma) = (k-1)LoG$. In our approximation $\sigma_2 = k\sigma_1$, and if k is not a large integer, then the approximation could be quite accurate since the approximation equation requires σ_1 and σ_2 be two small numbers close to each other. If the k value becomes relatively large, the approximation would not be as accurate as we wanted.

(c) I provide the code and result as below:

```python
def display_image(img, file_name=None):
    flt_img = img.astype(float)
    img_max, img_min = np.max(flt_img), np.min(flt_img)

    norm_img = (((flt_img - img_min) / (img_max - img_min)) * 255).astype(np.uint8)

    if len(img.shape) == 2:
        plt.imshow(norm_img, cmap='gray')
    elif (len(img.shape) == 3):
        plt.imshow(cv2.cvtColor(norm_img, cv2.COLOR_BGR2RGB))
    plt.show()

    if file_name:
        cv2.imwrite(file_name, norm_img)

def get_keypoints_and_features(img):
    sift = cv2.xfeatures2d.SIFT_create(1000)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    img2 = cv2.drawKeypoints(img, keypoints, np.array([]), (0,255,0),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    display_image(img2)
    return keypoints, descriptors


if __name__ == '__main__':
    img1 = cv2.imread('sample1.jpg')
    img2 = cv2.imread('sample2.jpg')
    get_keypoints_and_features(img1)
```


0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400
STAR METRO TOR
Anti-vaxxers assembl


0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400
The Great Composers
STAR METRO TORONTO
Anti-racers assemble

(b) Below is the code for this question

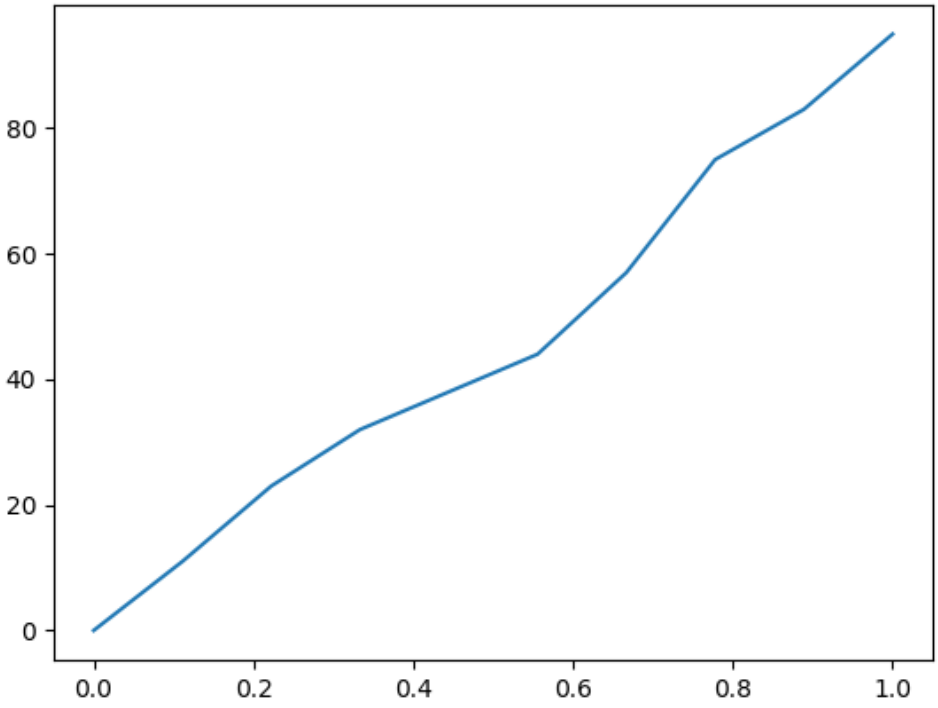
```python
def get_keypoints_and_features(img):
    sift = cv2.xfeatures2d.SIFT_create(1000)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    return keypoints, descriptors


def establish_feature_correspondence(img1, img2, keypoint_1, keypoint_2, descriptor_1,
descriptor_2, threshold):
    distances = spatial.distance.cdist(descriptor_1, descriptor_2, "euclidean")
    sorted_distances = np.argsort(distances, axis=1)
    closest, second_closest = sorted_distances[:, 0], sorted_distances[:, 1]

    ratio_array = np.zeros((closest.shape[0]))
    for i in range(closest.shape[0]):
        ratio = closest[i] / second_closest[i]
        if ratio < threshold:
            ratio_array[i] = ratio
        else:
            ratio_array[i] = 0

    remaining_descriptor_1_index = np.nonzero(ratio_array)[0]
    remaining_descriptor_2_index = closest[remaining_descriptor_1_index]

    pairs = np.stack((remaining_descriptor_1_index, remaining_descriptor_2_index)).transpose()
    pair_distances = distances[pairs[:, 0], pairs[:, 1]]
    sorted_dist_indices = np.argsort(pair_distances)
    sorted_pairs = pairs[sorted_dist_indices]

    location_pairs = []
    for i in range(10):
        temp = []
        ptr_1 = (keypoint_1[sorted_pairs[i, 0]])
        ptr_2 = (keypoint_2[sorted_pairs[i, 1]])
        temp.append(ptr_1)
        temp.append(ptr_2)
        location_pairs.append(temp)

    draw_matches(img1, img2, location_pairs)

    return len(pairs)

def draw_matches(img1, img2, location_pairs):
    result = np.concatenate((img1, img2), axis=1)
    for location_pair in location_pairs:
        x_ptr_0,  y_ptr_0 = location_pair[0].pt
        x_ptr_1, y_ptr_1 = location_pair[1].pt
        x_ptr_1 += img1.shape[1]
        cv2.line(result, (int(x_ptr_0), int(y_ptr_0)), (int(x_ptr_1), int(y_ptr_1)), (255, 0,
0), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    display_image(result)
    return 0

if __name__ == '__main__':
    img1 = cv2.imread('sample1.jpg')
    img2 = cv2.imread('sample2.jpg')
    kp1, dest1 = get_keypoints_and_features(img1)
    kp2, dest2 = get_keypoints_and_features(img2)
    establish_feature_correspondence(img1, img2, kp1, kp2, dest1, dest2, 0.5)
```

Here are the resulting top ten matches for threshold at 0.5 and 0.8

I think for threshold at 0.8 is the best result. Observing the graph we know that for threshold lower than 0.6, the number of matches are pretty small, and for threshold over 0.9, the number of matches increase scientifically fast, meaning that the number of matches mismatches increase. So, for sigma equals to 0.8 would be the best result.

(c) Below is the code:

```python
def get_keypoints_and_features(img):
    sift = cv2.xfeatures2d.SIFT_create(1000)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    return keypoints, descriptors


def establish_feature_correspondence(img1, img2, keypoint_1, keypoint_2, descriptor_1,
descriptor_2, threshold):
    distances = spatial.distance.cdist(descriptor_1, descriptor_2, "euclidean")
    sorted_distances = np.argsort(distances, axis=1)
    closest, second_closest = sorted_distances[:, 0], sorted_distances[:, 1]

    ratio_array = np.zeros((closest.shape[0]))
    for i in range(closest.shape[0]):
        ratio = closest[i] / second_closest[i]
        if ratio < threshold:
            ratio_array[i] = ratio
        else:
            ratio_array[i] = 0

    remaining_descriptor_1_index = np.nonzero(ratio_array)[0]
    remaining_descriptor_2_index = closest[remaining_descriptor_1_index]

    pairs = np.stack((remaining_descriptor_1_index, remaining_descriptor_2_index)).transpose()
    return len(paris)

if __name__ == '__main__':
    img1 = cv2.imread('sample1.jpg')
    img2 = cv2.imread('sample2.jpg')
    kp1, dest1 = get_keypoints_and_features(img1)
    kp2, dest2 = get_keypoints_and_features(img2)

    number_of_matches = []
    threshold_seq = np.linspace(0, 1, num=10)
    for threshold in threshold_seq:
        num_of_match = establish_feature_correspondence(img1, img2, kp1, kp2, dest1, dest2,
threshold)
        number_of_matches.append(num_of_match)

    plt.plot(threshold_seq, number_of_matches, label='number of matches')
    plt.show()
```


80
60
40
20
0
0.0
0.2
0.4
0.6
0.8
1.0

```python
def get_keypoints_and_features(img):
    sift = cv2.xfeatures2d.SIFT_create(200)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    return keypoints, descriptors


def establish_feature_correspondence(img1, img2, keypoint_1, keypoint_2, descriptor_1,
descriptor_2, threshold, mode):
    distances = np.zeros((len(keypoint_1), len(keypoint_2)))
    for i in range(len(keypoint_1)):
        for j in range(len(keypoint_2)):
            if mode == "L1":
                distances[i, j] = calculate_L1_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))
                print(distances[i, j])
            elif mode == "L2":
                distances[i, j] = calculate_L2_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))
            elif mode == "L3":
                distances[i, j] = calculate_L3_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))

    sorted_distances = np.argsort(distances, axis=1)
    closest, second_closest = sorted_distances[:, 0], sorted_distances[:, 1]

    ratio_array = np.zeros((closest.shape[0]))
    for i in range(closest.shape[0]):
        ratio = closest[i] / second_closest[i]
        if ratio < threshold:
            ratio_array[i] = ratio
        else:
            ratio_array[i] = 0

    remaining_descriptor_1_index = np.nonzero(ratio_array)[0]
    remaining_descriptor_2_index = closest[remaining_descriptor_1_index]

    pairs = np.stack((remaining_descriptor_1_index, remaining_descriptor_2_index)).transpose()
    pair_distances = distances[pairs[:, 0], pairs[:, 1]]
    sorted_dist_indices = np.argsort(pair_distances)
    sorted_pairs = pairs[sorted_dist_indices]

    location_pairs = []
    for i in range(10):
        temp = []
        ptr_1 = (keypoint_1[sorted_pairs[i, 0]])
        ptr_2 = (keypoint_2[sorted_pairs[i, 1]])
        temp.append(ptr_1)
        temp.append(ptr_2)
        location_pairs.append(temp)

    draw_matches(img1, img2, location_pairs)

    return len(pairs)

def draw_matches(img1, img2, location_pairs):
    result = np.concatenate((img1, img2), axis=1)
    for location_pair in location_pairs:
        x_ptr_0,  y_ptr_0 = location_pair[0].pt
        x_ptr_1, y_ptr_1 = location_pair[1].pt
        x_ptr_1 += img1.shape[1]
        cv2.line(result, (int(x_ptr_0), int(y_ptr_0)), (int(x_ptr_1), int(y_ptr_1)), (255, 0,
0), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    display_image(result)
    return 0

def calculate_L2_norm(diff_vector):
    sum = 0
    for vec in diff_vector:
        sum += vec ** 2
    return math.sqrt(sum)

def calculate_L1_norm(diff_vector):
    return sum(abs(diff_vector))

def calculate_L3_norm(diff_vector):
    sum = 0
    for vec in diff_vector:
        sum += abs(vec ** 3)
    return sum ** (1/3)

if __name__ == '__main__':
    img1 = cv2.imread('sample1.jpg')
    img2 = cv2.imread('sample2.jpg')
    kp1, dest1 = get_keypoints_and_features(img1)
    kp2, dest2 = get_keypoints_and_features(img2)
    establish_feature_correspondence(img1, img2, kp1, kp2, dest1, dest2, 0.8, "L1")
    establish_feature_correspondence(img1, img2, kp1, kp2, dest1, dest2, 0.8, "L2")
    establish_feature_correspondence(img1, img2, kp1, kp2, dest1, dest2, 0.8, "L3")
```

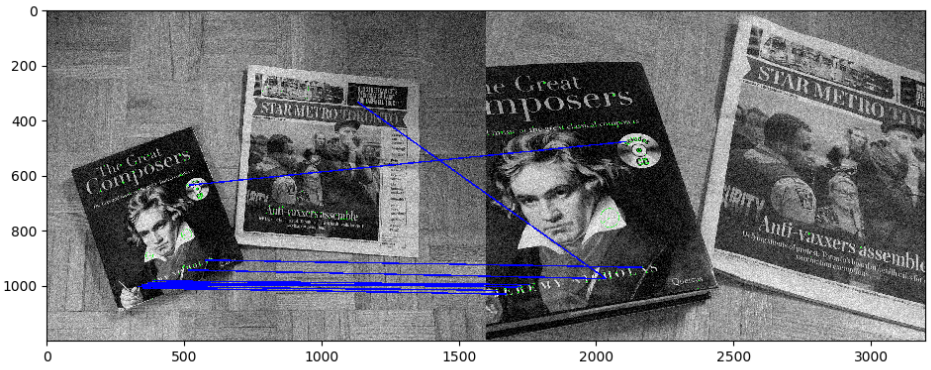
Below are resulting images for L1 norm, L2 norm and L3 norm. I think L2 norm is the best choice since for L1 norm, the top ten matches are all on small letters and cannot capture some important features like the disk on the image. For L3 norm, there are some mismatches. In this way, L2 norm is the best choice to calculate distances.


0
200
400
600
800
1000
0
500
1000
1500
2000
2500
3000
The Great Composers
STAR METRO TORONTO
Anti-vaxxers assemble
STAR METRO TOR
Anti-vaxxers assemble


0
200
400
600
800
1000
0
500
1000
1500
2000
2500
3000
The Great Composers
STAR METRO TORONTO
Anti-vaxxers assemble
STAR METRO TOR
Anti-vaxxers assemble

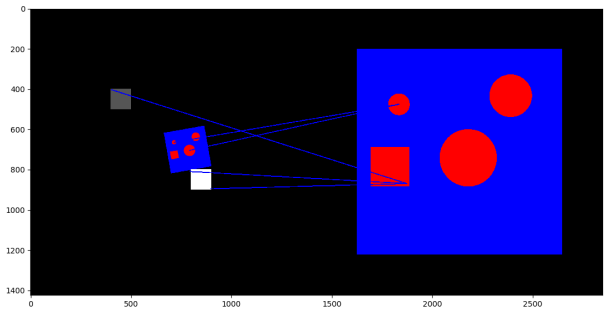
(d) Since we've discussed in question (b) and (c) that 0.8 and L2 norm are optimal choices for threshold and the method to calculate choices, we implement this question with threshold as 0.8 and the way to calculate distance as L2 norm.
After normalize the image to [0, 1], and adding noises, we need to rescale the image back to [0, 255], and the resulting image looks like below.

```python
def get_keypoints_and_features(img):
    sift = cv2.xfeatures2d.SIFT_create(200)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    img2 = cv2.drawKeypoints(img, keypoints, np.array([]), (0,255,0),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    return keypoints, descriptors, img2

def establish_feature_correspondence(img1, img2, keypoint_1, keypoint_2, descriptor_1,
descriptor_2, threshold, mode):
    distances = np.zeros((len(keypoint_1), len(keypoint_2)))
    for i in range(len(keypoint_1)):
        for j in range(len(keypoint_2)):
            if mode == "L1":
                distances[i, j] = calculate_L1_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))
            elif mode == "L2":
                distances[i, j] = calculate_L2_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))
                print(distances[i, j])
            elif mode == "L3":
                distances[i, j] = calculate_L3_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))

    sorted_distances = np.argsort(distances, axis=1)
    closest, second_closest = sorted_distances[:, 0], sorted_distances[:, 1]

    ratio_array = np.zeros((closest.shape[0]))
    for i in range(closest.shape[0]):
        ratio = closest[i] / second_closest[i]
        if ratio < threshold:
            ratio_array[i] = ratio
        else:
            ratio_array[i] = 0

    remaining_descriptor_1_index = np.nonzero(ratio_array)[0]
    remaining_descriptor_2_index = closest[remaining_descriptor_1_index]

    pairs = np.stack((remaining_descriptor_1_index, remaining_descriptor_2_index)).transpose()
    pair_distances = distances[pairs[:, 0], pairs[:, 1]]
    sorted_dist_indices = np.argsort(pair_distances)
    sorted_pairs = pairs[sorted_dist_indices]

    location_pairs = []
    for i in range(10):
        temp = []
        ptr_1 = (keypoint_1[sorted_pairs[i, 0]])
        ptr_2 = (keypoint_2[sorted_pairs[i, 1]])
        temp.append(ptr_1)
        temp.append(ptr_2)
        location_pairs.append(temp)

    draw_matches(img1, img2, location_pairs)

    return len(pairs)


def draw_matches(img1, img2, location_pairs):
    result = np.concatenate((img1, img2), axis = 1)
    for location_pair in location_pairs:
        x_ptr_0,  y_ptr_0 = location_pair[0].pt
        x_ptr_1, y_ptr_1 = location_pair[1].pt
        x_ptr_1 += img1.shape[1]
        cv2.line(result, (int(x_ptr_0), int(y_ptr_0)), (int(x_ptr_1), int(y_ptr_1)), (255, 0,
0), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    display_image(result)
    return 0

def calculate_L2_norm(diff_vector):
    sum = 0
    for vec in diff_vector:
        sum += vec ** 2
    return math.sqrt(sum)

def calculate_L1_norm(diff_vector):
    return sum(abs(diff_vector))

def calculate_L3_norm(diff_vector):
    sum = 0
    for vec in diff_vector:
        sum += abs(vec ** 3)
    return sum ** (1/3)

def addGaussianAndNormalize(img):
    img = img.astype(np.float32)
    img *= (1.0/255)
    img_with_noise = img + np.random.normal(0, 0.08, img.shape)
    img_with_noise[img_with_noise < 0] = 0
    img_with_noise[img_with_noise > 1] = 1
    img_with_noise *= 255
    img_with_noise = img_with_noise.astype(np.uint8)
    return img_with_noise

if __name__ == '__main__':
    img1 = cv2.imread('sample1.jpg', 0)
    img2 = cv2.imread('sample2.jpg', 0)

    img1 = addGaussianAndNormalize(img1)
    img2 = addGaussianAndNormalize(img2)

    kp1, dest1, img3 = get_keypoints_and_features(img1)
    kp2, dest2, img4 = get_keypoints_and_features(img2)

    establish_feature_correspondence(img3, img4, kp1, kp2, dest1, dest2, 0.8, "L2")
```



(e) The way I choose is break the image into three different color channels, then detect matching points for three channels respectively. After getting three results, we check whether these three sets have some common matching points. If they do, then draw the matching of such points on graph.

```
def get_keypoints_and_features(img):
    sift = cv2.xfeatures2d.SIFT_create(1000)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    if (descriptors is None):
        descriptors = np.zeros((img.shape[0], img.shape[1]))
    img2 = cv2.drawKeypoints(img, keypoints, np.array([]), (0,255,0),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    return keypoints, descriptors, img2


def establish_feature_correspondence(keypoint_1, keypoint_2, descriptor_1, descriptor_2,
threshold, mode):
    distances = np.zeros((len(keypoint_1), len(keypoint_2)))
    for i in range(len(keypoint_1)):
        for j in range(len(keypoint_2)):
            if mode == "L1":
                distances[i, j] = calculate_L1_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))
            elif mode == "L2":
                distances[i, j] = calculate_L2_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))
            elif mode == "L3":
                distances[i, j] = calculate_L3_norm(descriptor_1[i].reshape(-1) -
descriptor_2[j].reshape(-1))

    sorted_distances = np.argsort(distances, axis=1)
    closest, second_closest = sorted_distances[:, 0], sorted_distances[:, 1]

    ratio_array = np.zeros((closest.shape[0]))
    for i in range(closest.shape[0]):
        ratio = closest[i] / second_closest[i]
        if ratio < threshold:
            ratio_array[i] = ratio
        else:
            ratio_array[i] = 0

    remaining_descriptor_1_index = np.nonzero(ratio_array)[0]
    remaining_descriptor_2_index = closest[remaining_descriptor_1_index]


    pairs = np.stack((remaining_descriptor_1_index, remaining_descriptor_2_index)).transpose()
    pair_distances = distances[pairs[:, 0], pairs[:, 1]]
    sorted_dist_indices = np.argsort(pair_distances)
    sorted_pairs = pairs[sorted_dist_indices]

    location_pairs = []
    for i in range(10):
        temp = []
        ptr_1 = (keypoint_1[sorted_pairs[i, 0]])
        ptr_2 = (keypoint_2[sorted_pairs[i, 1]])
        temp.append(ptr_1)
        temp.append(ptr_2)
        location_pairs.append(temp)

    return location_pairs

def draw_matches(img1, img2, location_pairs):

    result = np.concatenate((img1, img2), axis = 1)
    for location_pair in location_pairs:
        x_ptr_0,  y_ptr_0 = location_pair[0].pt
        x_ptr_0 += 300
        y_ptr_0 += 300
        x_ptr_1, y_ptr_1 = location_pair[1].pt
        x_ptr_1 += img1.shape[1]
        cv2.line(result, (int(x_ptr_0), int(y_ptr_0)), (int(x_ptr_1), int(y_ptr_1)), (255, 0,
0), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    display_image(result)
    return 0

def calculate_L2_norm(diff_vector):
    sum = 0
    for vec in diff_vector:
        sum += vec ** 2
    return math.sqrt(sum)

def calculate_L1_norm(diff_vector):
    return sum(abs(diff_vector))

def calculate_L3_norm(diff_vector):
    sum = 0
    for vec in diff_vector:
        sum += abs(vec ** 3)
    return sum ** (1/3)

def intersection(ls1, ls2):
    ls3 = [value for value in ls1 if value in ls2]
    return ls3

if __name__ == '__main__':
    img1 = cv2.imread('colourSearch.png')
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
    img2 = cv2.imread('colourTemplate.png')
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)
    img1_1 = img1[:, :, 0]
    img1_2 = img1[:, :, 1]
    img1_3 = img1[:, :, 2]
    img2_1 = img2[:, :, 0]
    img2_2 = img2[:, :, 1]
    img2_3 = img2[:, :, 2]

    kp1_1, dest1_1, img1_1_0 = get_keypoints_and_features(img1_1)
    kp2_1, dest2_1, img2_1_0 = get_keypoints_and_features(img2_1)
    set_A = establish_feature_correspondence(kp1_1, kp2_1, dest1_1, dest2_1, 0.8, "L2")

    kp1_2, dest1_2, img1_2_0 = get_keypoints_and_features(img1_2)
    kp2_2, dest2_2, img2_2_0 = get_keypoints_and_features(img2_2)
    set_B = establish_feature_correspondence(kp1_2, kp2_2, dest1_2, dest2_2, 0.8, "L2")

    kp1_3, dest1_3, img1_3_0 = get_keypoints_and_features(img1_3)
    kp2_3, dest2_3, img2_3_0 = get_keypoints_and_features(img2_3)
    set_C = establish_feature_correspondence(kp1_3, kp2_3, dest1_3, dest2_3, 0.8, "L2")

    set_to_plot = intersection(set_A, set_C)
    set_to_plot = intersection(set_B, set_to_plot)
    draw_matches(img1, img2, set_to_plot)
```


0
200
400
600
800
1000
1200
1400
0
500
1000
1500
2000
2500