CSC420 A2

Xinqi Shen

1. **(a)**

Apply two 1-D linear interpolation

Yes. We can do the same operation with a single 2-dimensional filter. The 2-dimensional filter is just the multiply of two 1-dimensional filter. In this question, the 2-D filter is shown below:

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

(b)

Apply 2-D linear interpolation(bilinear)

I just used the 2-D filter described in (a); the output image is exactly the same.
We can derive the bilinear interpolation:

Suppose we have unknown point (x, y), we know four points Q_{11} = (x1, y1), Q_{12} = (x1, y2), Q_{21} = (x2, y1), and Q_{22} = (x2, y2).
We first do linear interpolation in x-direction:

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

Then, we do linear interpolation in y-direction:

$$f(x,y) \approx \frac{y_2 - y}{y_2 - y_1} f(x,y_1) + \frac{y - y_1}{y_2 - y_1} f(x,y_2)$$

$$= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}$$

We can easily compare the pixel values for both method (two 1-D linear interpolation and one bilinear interpolation) and find the pixel values are the same.

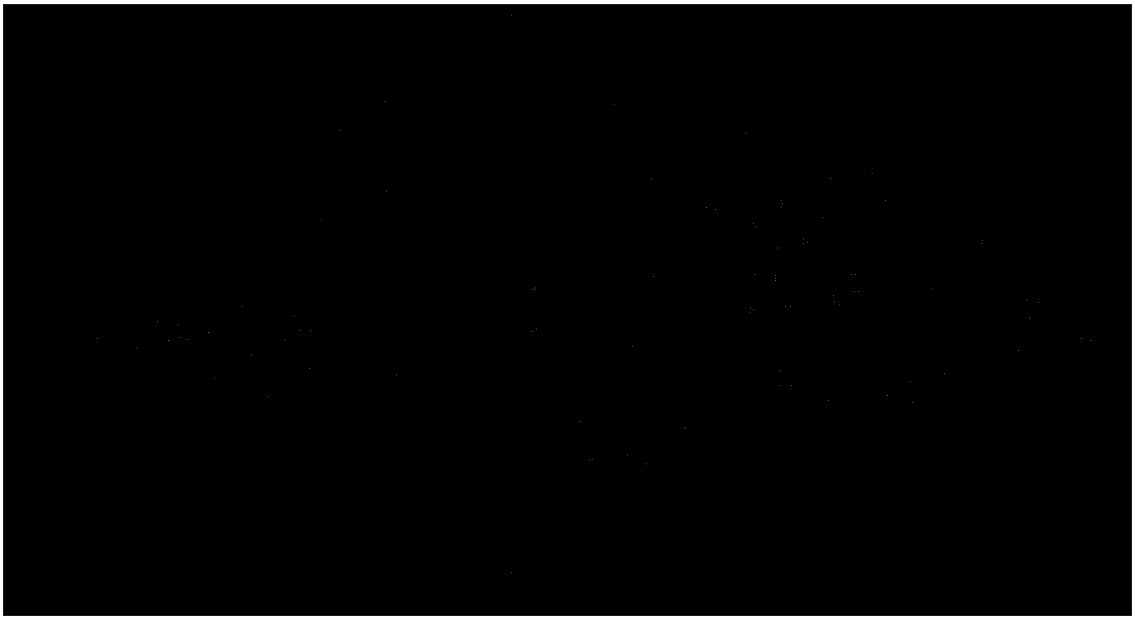
Code:

Noted that I combine question (a) and (b) function into single function with different flag parameter 'dimension'

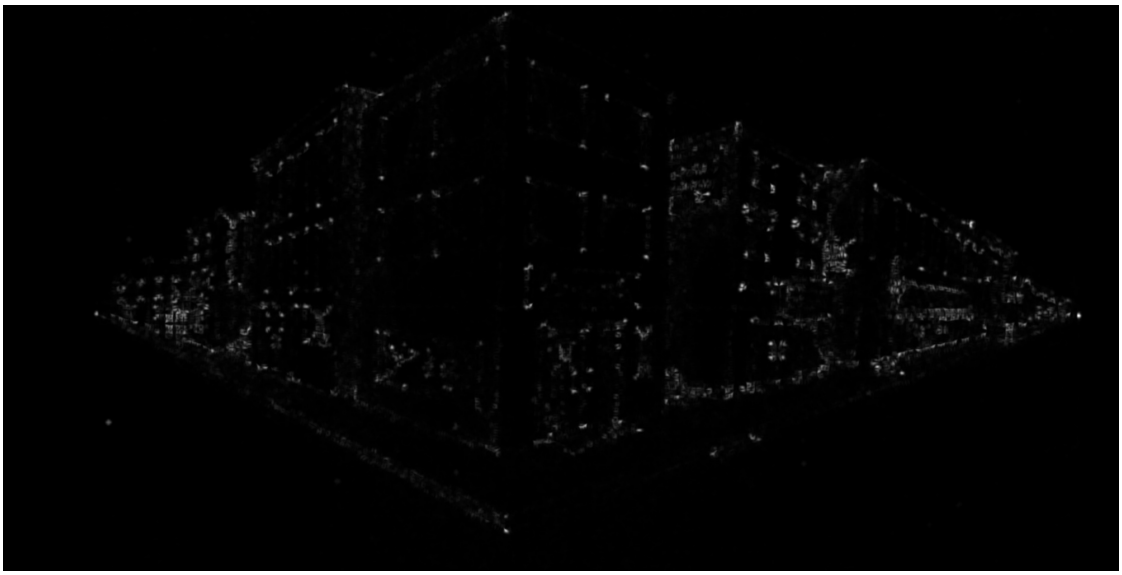
```
#dimension 0: Upsampling in Y dimension
#dimension 1: Upsampling in X demension
#dimension 3: Upsampling in X and Y demension
def quadruple(I,size,dimension,kernel):
  image_h, image_w, image_channel = I.shape[0], I.shape[1], I.shape[2]
  if dimension == 0:
    padding = np.zeros((image_h*size,image_w,image_channel))
    padding[::size] = I
  elif dimension == 1:
    padding = np.zeros((image_h,image_w*size,image_channel))
    padding[::,::size] = I
  elif dimension == 3:
    padding = np.zeros((image_h*size,image_w*size,image_channel))
    padding[::size,::size] = I
  return cv2.filter2D(padding,-1,kernel)

kernel_1 = np.transpose(np.array([[1/4,2/4,3/4,1,3/4,2/4,1/4]]))
res = quadruple(bee,4,0,kernel_1)
kernel_2 = np.array([[1/4,2/4,3/4,1,3/4,2/4,1/4]])
res1 = quadruple(res,4,1,kernel_2)
cv2.imwrite("/content/drive/My Drive/bee_new.jpg",res1)

kernel_3 = kernel_1.dot(kernel_2)
res2 = quadruple(bee,4,3,kernel_3)
cv2.imwrite("/content/drive/My Drive/bee_new1.jpg",res1)
```

2. **(a)**

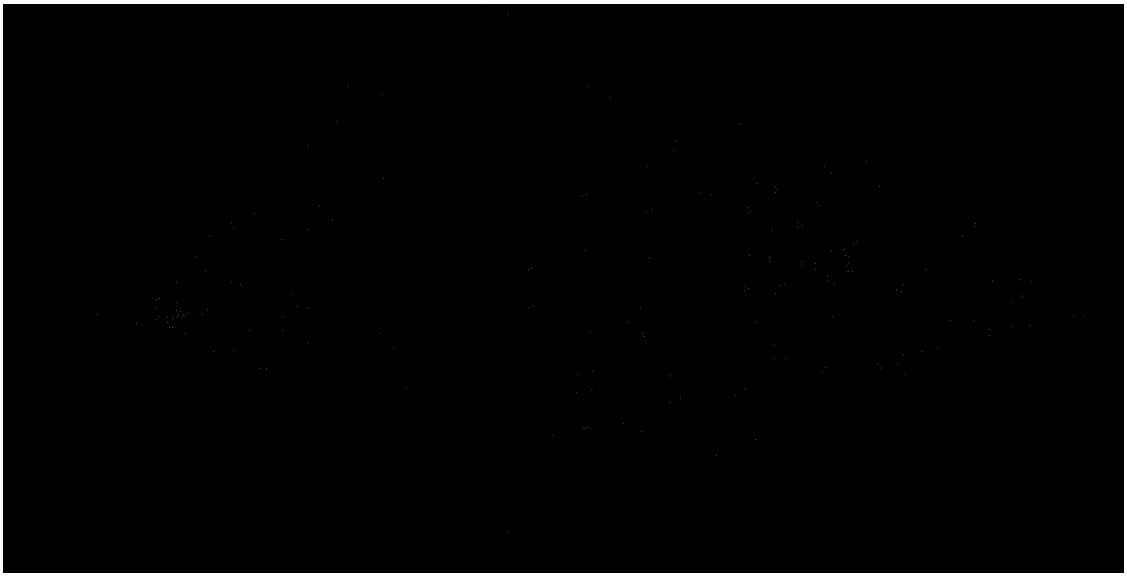
Harris Before Non-Maximum Suppression



Harris After Non-Maximum Suppression (Zoom in to see small points)



Brown Before Non-Maximum Suppression



Brown After Non-Maximum Suppression (Zoom in to see small points)

Yes. We can calculate the eigenvalues and then replace det(M) as $\lambda_0\lambda_1$ and trace(M) as $\lambda_0 + \lambda_1$ for Harris and Brown corner detector. The output image is same as using determinant and trace.

Code:

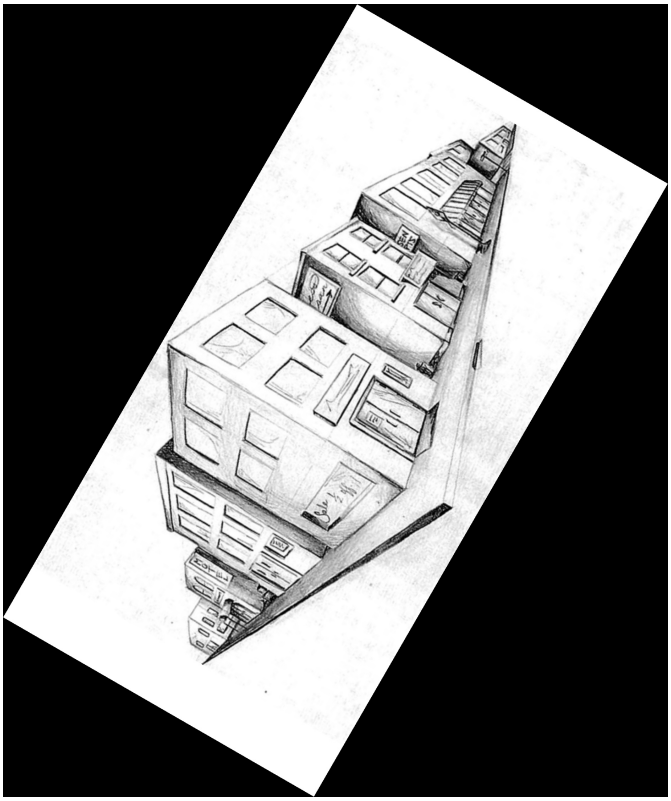
```
def Harris(I,threshold):
  blur = cv2.GaussianBlur(I,(5,5),7)
  Ix = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=5)
  Iy = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=5)
  IxIy = np.multiply(Ix, Iy)
  Ix2 = np.multiply(Ix, Ix)
  Iy2 = np.multiply(Iy, Iy)
  Ix2_blur = cv2.GaussianBlur(Ix2,(7,7),10)
  Iy2_blur = cv2.GaussianBlur(Iy2,(7,7),10)
  IxIy_blur = cv2.GaussianBlur(IxIy,(7,7),10)
  det = np.multiply(Ix2_blur, Iy2_blur) - np.multiply(IxIy_blur,IxIy_blur)
  trace = Ix2_blur + Iy2_blur
  R = det - 0.05 * np.multiply(trace,trace)
  R[R<threshold] = 0
  return R
def Brown(I):
  blur = cv2.GaussianBlur(I,(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)
  det = np.multiply(Ix2_blur, Iy2_blur) - np.multiply(IxIy_blur,IxIy_blur)
  trace = Ix2_blur + Iy2_blur
  mean = np.divide(det,trace+0.01)
  return mean
```

```
def Non_suppression(I):
  out = I.copy()
  height,width = I.shape
  for i in range(1,height-1):
    for j in range(1,width-1):
      patch = I[i-1:i+2, j-1:j+2]
      if np.argmax(patch) == 4 and patch.max()>90:
        out[i][j] = 255
      else:
        out[i][j] = 0
  return out
```

```
def eigenvalue(I,Ix2,Iy2,IxIy):
  height, width = I.shape[0],I.shape[1]
  lamda_0_m = np.zeros((height,width))
  lamda_1_m = np.zeros((height,width))
  for i in range(height):
    for j in range(width):
      pixel_Ix2 = Ix2[i][j]
      pixel_Iy2 = Iy2[i][j]
      Pixel_IxIy = IxIy[i][j]
      M = np.array([[pixel_Ix2,Pixel_IxIy],[Pixel_IxIy,pixel_Iy2]])
      lamda_0 = np.linalg.eigvals(M)[0]
      lamda_1 = np.linalg.eigvals(M)[1]
      lamda_0_m[i][j] = lamda_0
      lamda_1_m[i][j] = lamda_1
  return [lamda_0_m,lamda_1_m]


def Harris_without(I):
  blur = cv2.GaussianBlur(I,(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)
  L0,L1 = eigenvalue(I,Ix2_blur,Iy2_blur,IxIy_blur)
  L0L1 = np.multiply(L0,L1)
  L0_plus_L1 = L0+L1
  R = L0L1- 0.05*np.multiply(L0_plus_L1,L0_plus_L1)
  return R
```

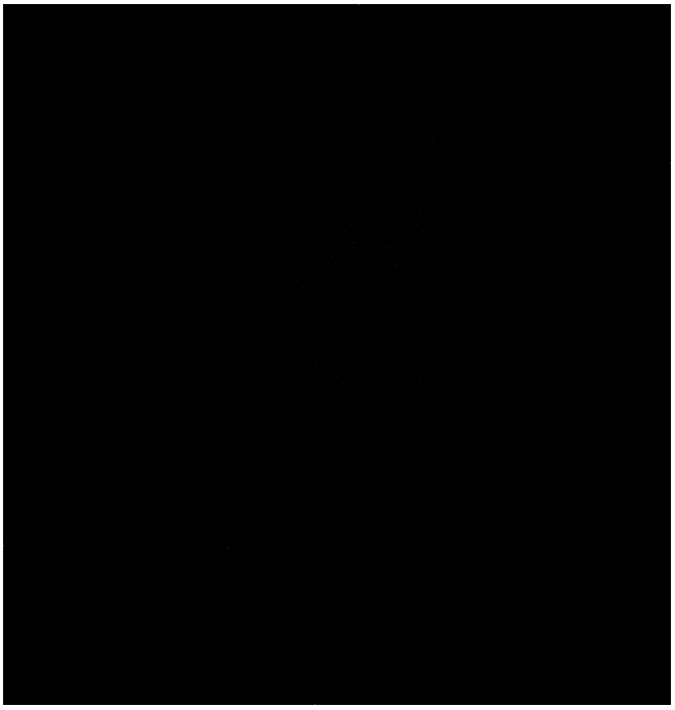
```
def Brown_without(I):
  blur = cv2.GaussianBlur(I,(5,5),7)
  Ix = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=5)
  Iy = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=5)
  IxIy = np.multiply(Ix, Iy)
  Ix2 = np.multiply(Ix, Ix)
  Iy2 = np.multiply(Iy, Iy)
  Ix2_blur = cv2.GaussianBlur(Ix2,(7,7),10)
  Iy2_blur = cv2.GaussianBlur(Iy2,(7,7),10)
  IxIy_blur = cv2.GaussianBlur(IxIy,(7,7),10)
  L0,L1 = eigenvalue(I,Ix2_blur,Iy2_blur,IxIy_blur)
  L0L1 = np.multiply(L0,L1)
  L0_plus_L1 = L0+L1
  mean = np.divide(L0L1,L0_plus_L1+0.01)
  return mean
```

(b)



The original image

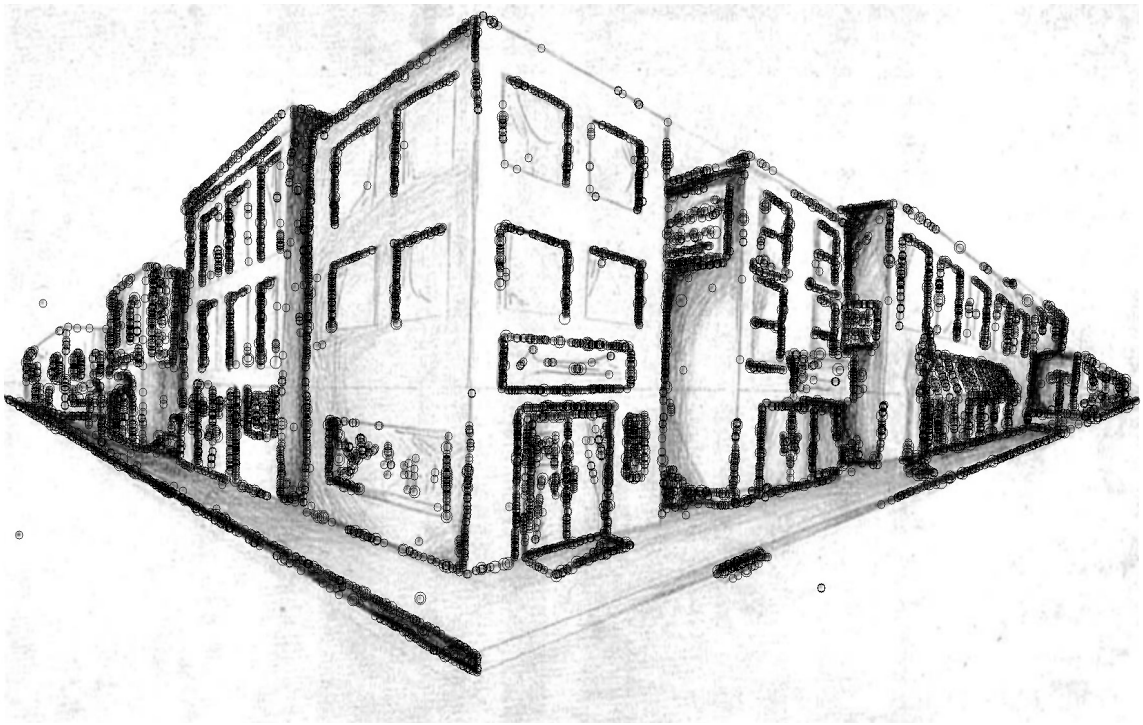
Harris Before Non-Maximum Suppression



Harris After Non-Maximum Suppression (Zoom in to see small points)

Yes. Since the Harris corner detector is rotate invariant. Since we calculate the eigenvalues to get R, then its second moment ellipse rotates but its shape remains the same.

(c)



I draw the keypoints with black circle in this image. (you can zoom in image to see more detail)

Code：

```
def Laplacian(I,ksize,sigma):
  gaussian = cv2.GaussianBlur(I,(ksize,ksize),sigma)
  return cv2.Laplacian(gaussian,-1)

sigma = 2
s = 7
k = 2**(1/s)

sigma_array = [2]
for i in range(0,s-1):
  sigma_array.append(sigma_array[-1]*k)
Log = []

for i in range(0,s):
  Log.append(Laplacian(building_gray,5,sigma_array[i]))

Log_padded = []
height, width = building_gray.shape[0],building_gray.shape[1]

for i in range(0,s):
  padded = np.zeros((height+2,width+2))
  padded[1:height+1,1:width+1] = Log[i]
  Log_padded.append(padded)

Octave = np.concatenate([o[:,:,np.newaxis] for o in Log_padded], axis=2)
keypoints = {}
Octave[:,:,0] = 0
Octave[:,:,-1] = 0
for i in range(1,height+1):
  for j in range(1,width+1):
    for k in range(1,Octave.shape[2]-1):
      patch = Octave[i-1:i+2, j-1:j+2, k-1:k+2]
      if np.argmax(patch) == 13 or np.argmin(patch) == 13:
        if np.abs(patch[1][1][1]) >= 18 and np.abs(patch[1][1][1]) <= 35:
          if (i,j) not in keypoints:
            keypoints[(i,j)] = sigma_array[k]

new_building1 = cv2.cvtColor(building, cv2.COLOR_BGR2GRAY)
for key in keypoints.keys():
  cv2.circle(new_building1,(key[1],key[0]), radius=5*int(keypoints[key]),color=(0,0,255),thickness=1)
```

(d) SURF (Speeded up robust features)

In order to detect interest points, SURF uses an integer approximation of the determinant of Hessian blob detector, which can be computed with 3 integer operations using a precomputed integral image. Its feature descriptor is based on the sum of the Haar wavelet response around the point of interest. These can also be computed with the aid of the integral image.

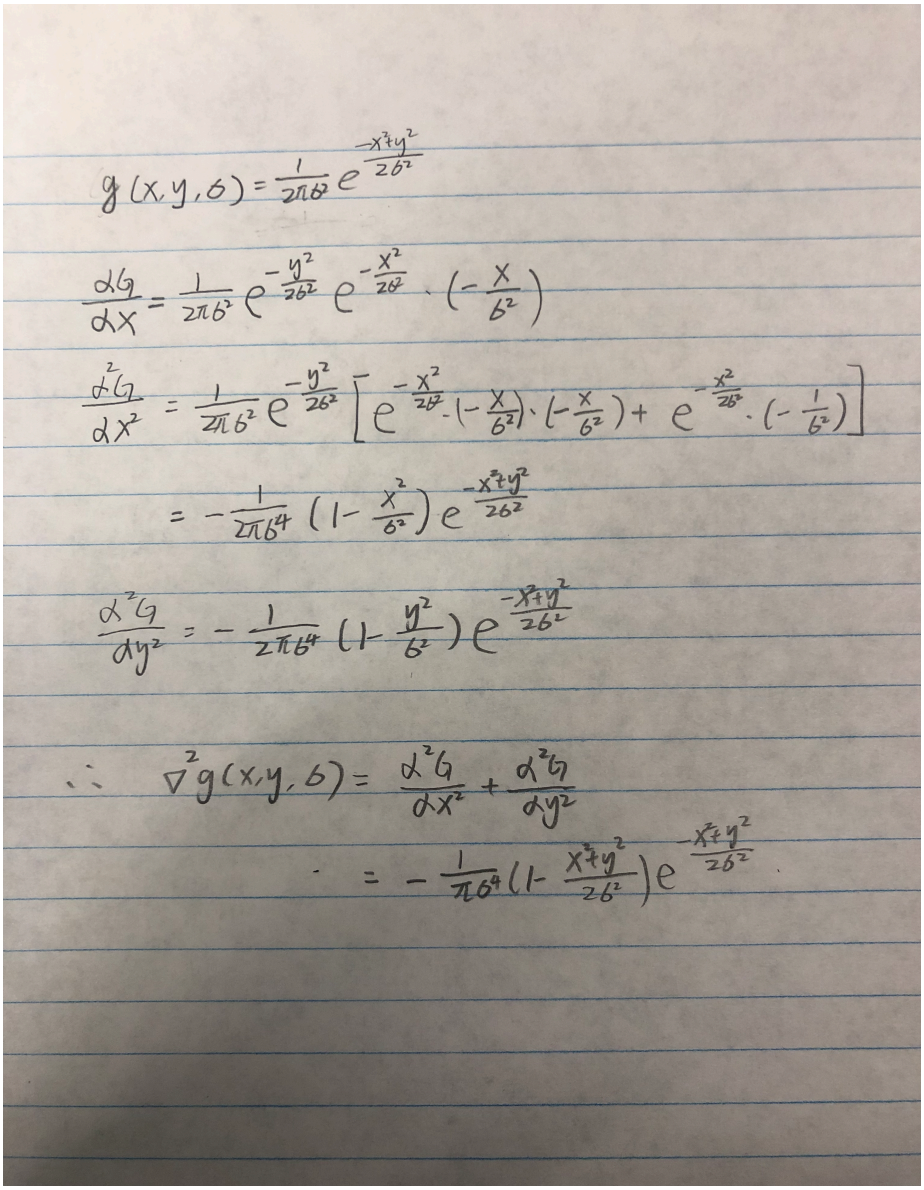
1. **Find key points and features**. SURF approximates LOG with Box Filter. Convolution with box filter can be easily calculated with the help of integral images. And it can be done in parallel for different scales. Also, the SURF relies on determinant of Hessian matrix for both scale and location.

2. **Orientation assignment.** SURF uses Wavelet responses in horizontal and vertical direction for a neighborhood of size 6s. Adequate gaussian weights are also applied to it. The dominant orientation is estimated by calculating the sum of all responses within a sliding orientation window of angle 60 degrees.

3. **Feature description**. SURF uses Wavelet responses in horizontal and vertical direction again. A neighborhood of size 20sX20s is taken around the keypoints where s is the size. It is divided into 4x4 subregions. For each subregion, horizontal and vertical wavelet responses are taken and a vector is formed like this, $\mathrm{v} = (\sum d_x,\ \sum d_y,\ \sum|d_x|,\ \sum|d_y|)$. This when represented as a vector gives SURF feature descriptor with total 64 dimensions.

4. **Matching**. By comparing the descriptors obtained from different images, matching pairs can be found.

3. (a)



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

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

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

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

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

$$\therefore \nabla^2 g(x,y,\sigma) = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2}$$

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

The LOG filter is not separable since the term ($1 - \frac{x^2+y^2}{2\sigma^2}$) cannot be rewritten to a product of two linear filters.

(b)

From heat diffusion equation

$$\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G$$

Then $\nabla^2 G$ can be computed from the finite difference approximation to $\frac{\partial G}{\partial \sigma}$, using the difference of nearby scales at $k\sigma$ and σ.

let $k\sigma = \sigma_1$ and $\sigma = \sigma_2$

$$\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{k\sigma - \sigma} = \frac{G(x,y,\sigma_2) - G(x,y,\sigma_1)}{\sigma_2 - \sigma_1}$$

so $G(x,y,\sigma_1) - G(x,y,\sigma_2) \approx (k-1)\sigma^2 \nabla^2 G = (\sigma_2 - \sigma_1)\sigma \nabla^2 G$

This shows that when the difference of Gaussian function has scales differing by a constant factor it already incorporates the σ^2 scale normalization required for the scale-invariant, thus the choice of σ_1 and σ_2 does not influence extrema location.

4. **(a)**

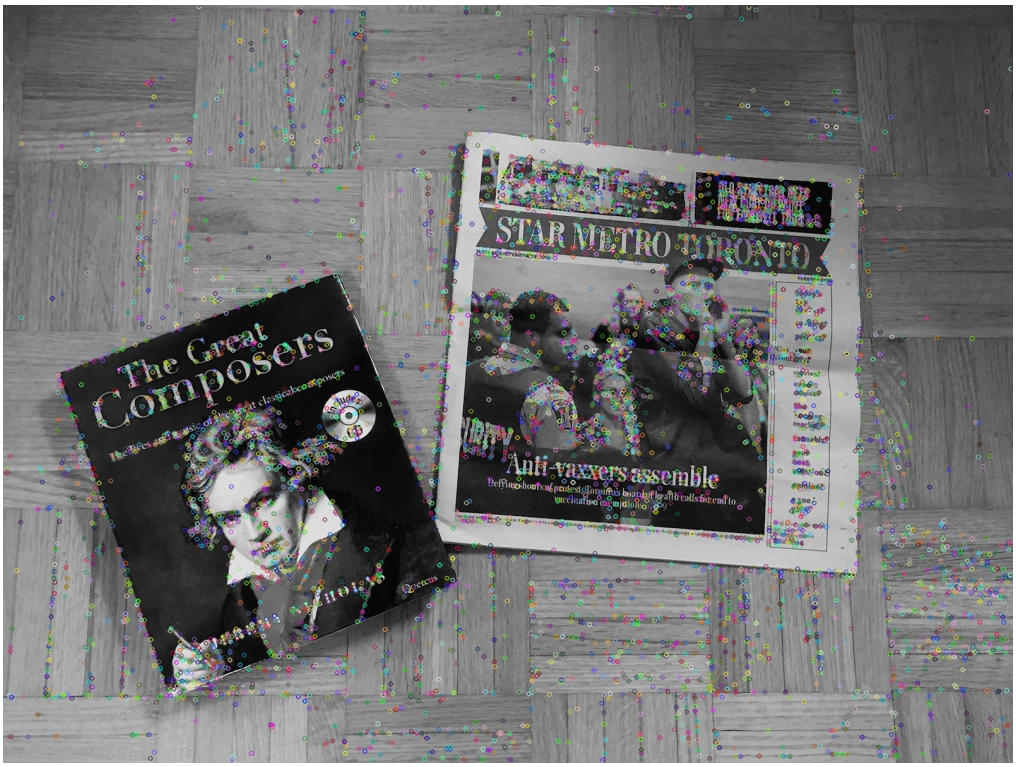


Image1 with keypoints

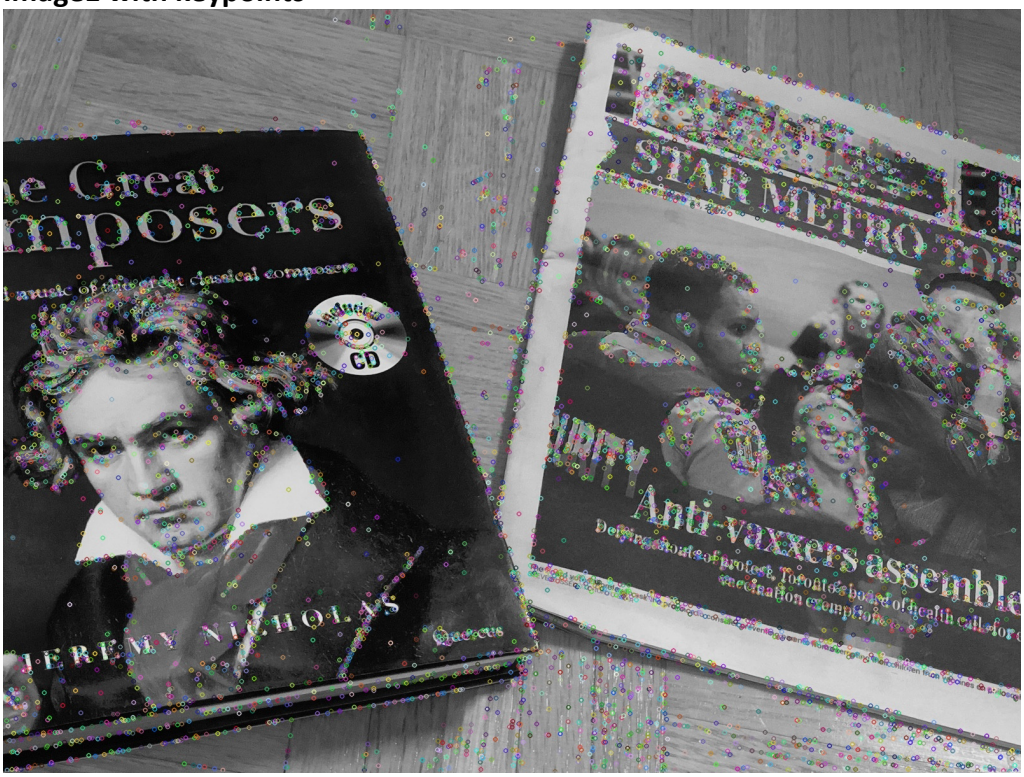
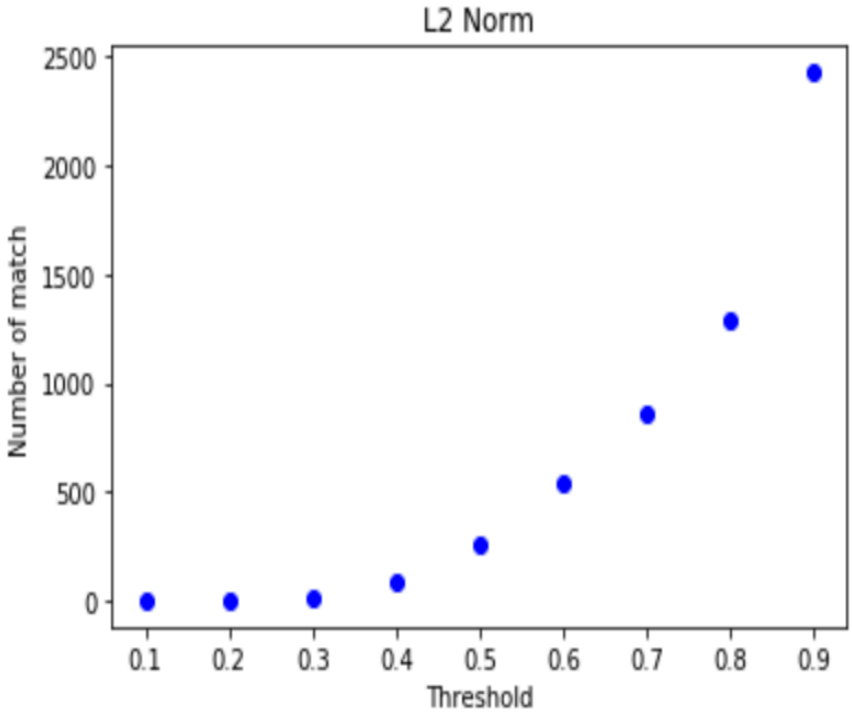


Image2 with keypoints

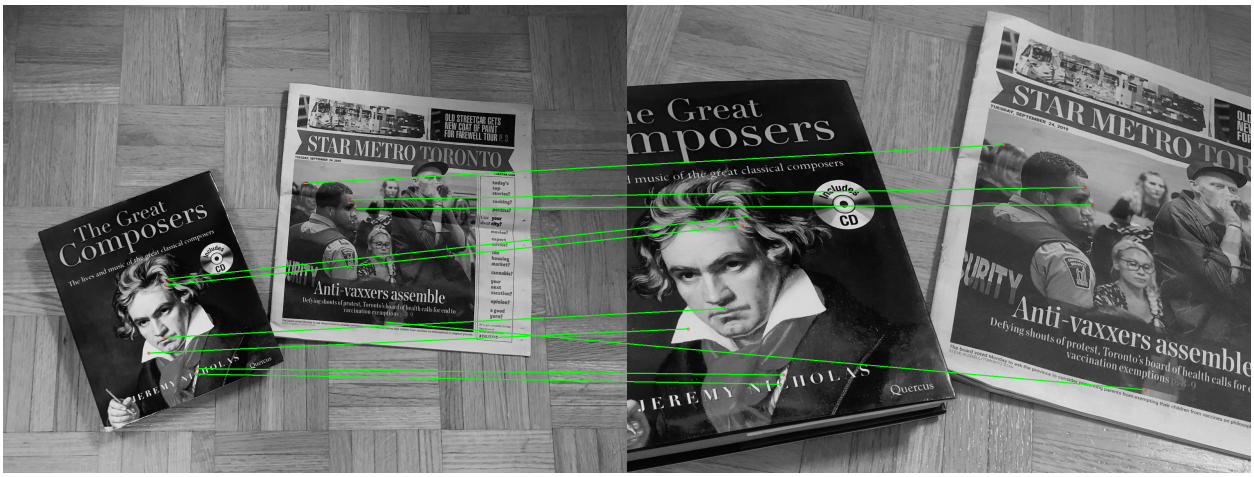
Code:

```
def SIFT(I):
  sift = cv2.xfeatures2d.SIFT_create()
  kp, des = sift.detectAndCompute(I,None)
  img = cv2.drawKeypoints(I,kp,None)
  return img
```

(b)



The best threshold is 0.8, since it gives a reasonably large number of matches among all the threshold. Noted that, even though threshold 0.9 gives the largest number of matches, it suddenly doubles the number of matches than threshold 0.8, it implies the actual matches will contain some more “false positive” matches (incorrect match being returned).



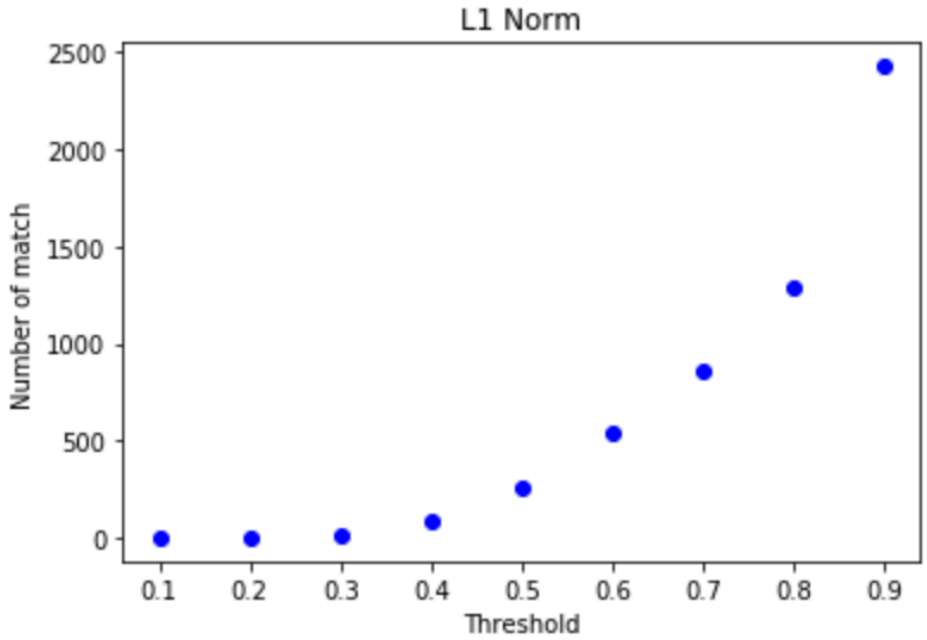
L2-norm

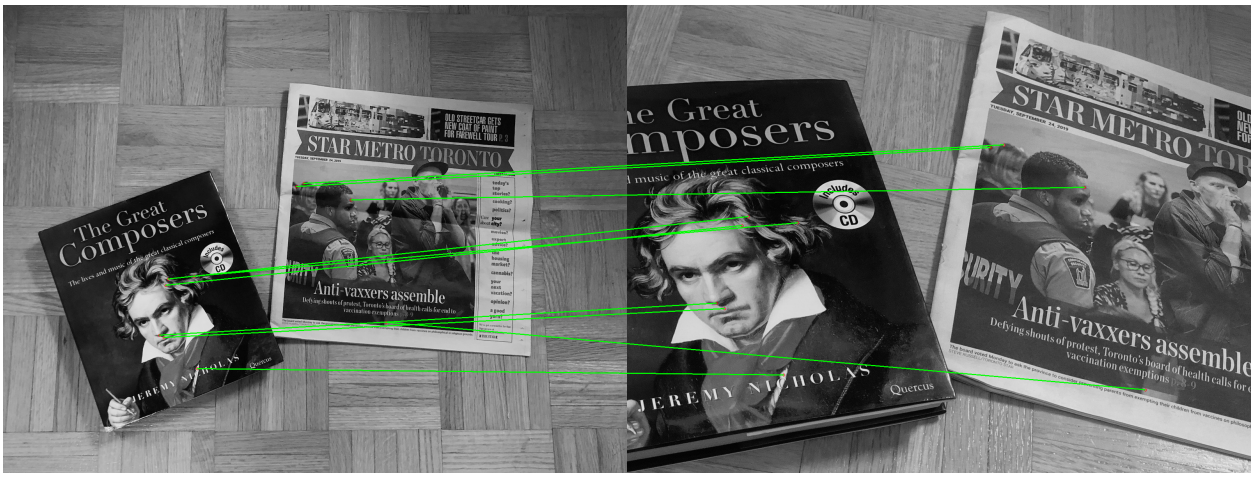
Code:

```
def sift_matching(img1,img2,threshold,norm_type):
  sift = cv2.xfeatures2d.SIFT_create()
  kp1, des1 = sift.detectAndCompute(img1,None)
  kp2, des2 = sift.detectAndCompute(img2,None)
  matched_pair = {}
  for i in range(len(kp1)):
    difference = des1[i,:] - des2
    differnece_abs = np.abs(difference)
    if norm_type == 1: #L1 norm
      res = np.sum(differnece_abs,axis=1)
    elif norm_type == 2: #L2 norm
      res_sum = np.sum(differnece_abs**2,axis=1)
      res = res_sum**(1/2)
    elif norm_type == 3: #L3 norm
      res_sum = np.sum(differnece_abs**3,axis=1)
      res = res_sum**(1/3)
    index_sorted = np.argsort(res)[:2]
    smallest = index_sorted[0]
    second_smallest = index_sorted[1]
    ratio = res[smallest] / res[second_smallest]
    if ratio < threshold:
      matched_pair[ratio] = (kp1[i],kp2[smallest])

  return matched_pair
```

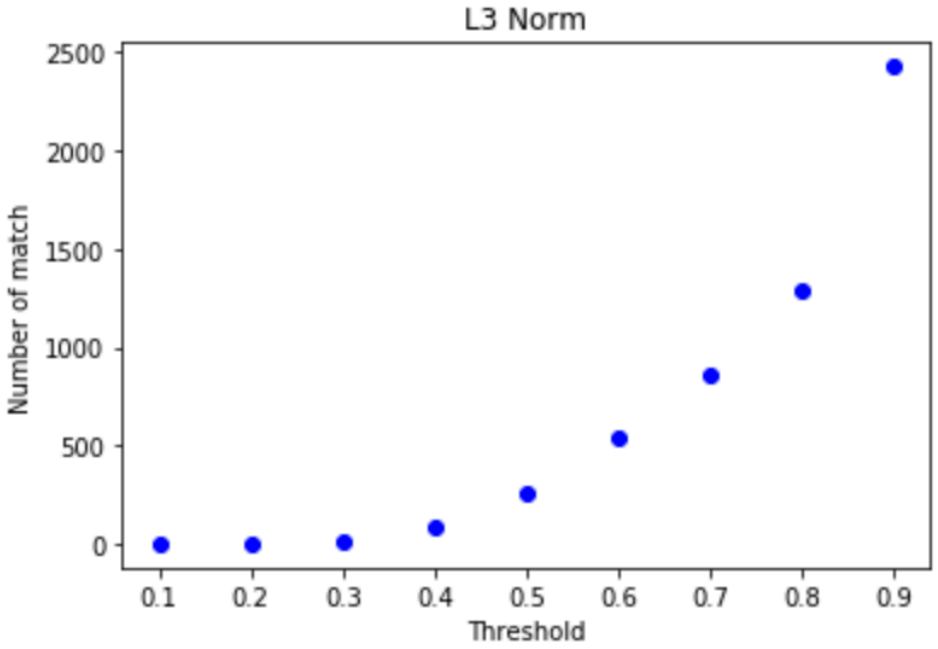
```python
#result from sift_matching function
def find_kp(result):
  points_in_img1 = []
  points_coordinate_img1 = []
  points_in_img2 = []
  points_coordinate_img2 = []
  count = 0
  for key in sorted(result):
    if count == 10:
      break
    img1_pt = (int(result[key][0].pt[0]),int(result[key][0].pt[1]))
    img2_pt = (int(result[key][1].pt[0]),int(result[key][1].pt[1]))
    if img1_pt not in points_coordinate_img1:
      points_in_img1.append(result[key][0])
      points_coordinate_img1.append(img1_pt)
      points_in_img2.append(result[key][1])
      points_coordinate_img2.append(img2_pt)
      count += 1
  return points_in_img1,points_coordinate_img1,points_in_img2,points_coordinate_img2
```

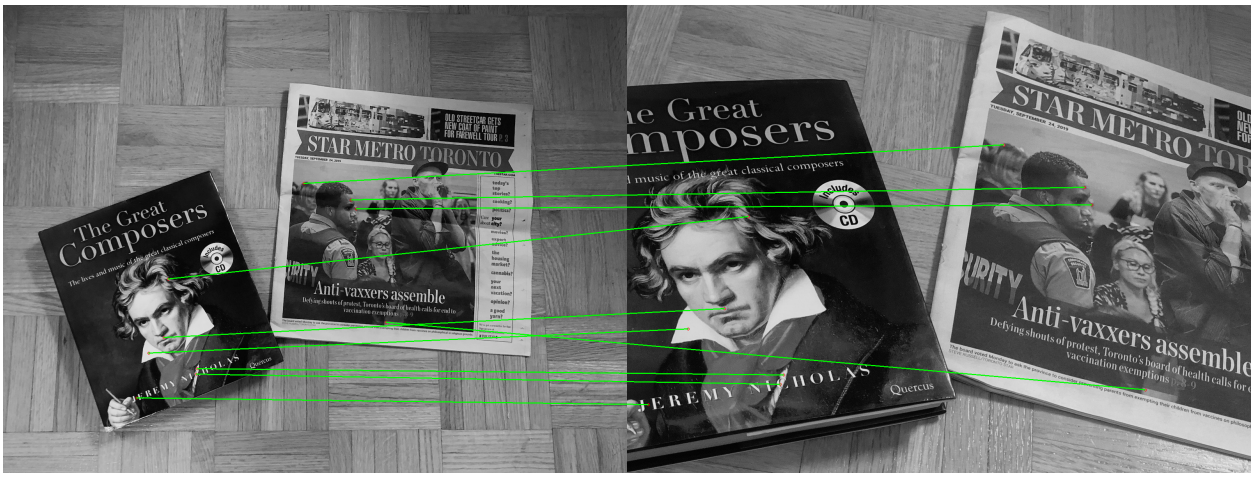
(c)




The Great Composers
STAR METRO TORONTO
Anti-vaxxers assemble
Includes CD
JEREMY NICHOLAS
Quercus

L1-norm


L3 Norm
Number of match
Threshold
2500
2000
1500
1000
500
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9



L3-norm

There is no distinct difference among L1-norm, L2-norm and L3-norm. Firstly, the number of matches plot kept the same with only tiny difference, and we still found that **0.8** is the best threshold value in each distance function. Then, in terms of matches, majority of the matching points are unchanged, the changed key points also appear at interesting position, either corner or edge. In summary, the distance function makes no large difference from my observation. We may need more sample images for further comparison.

(d)

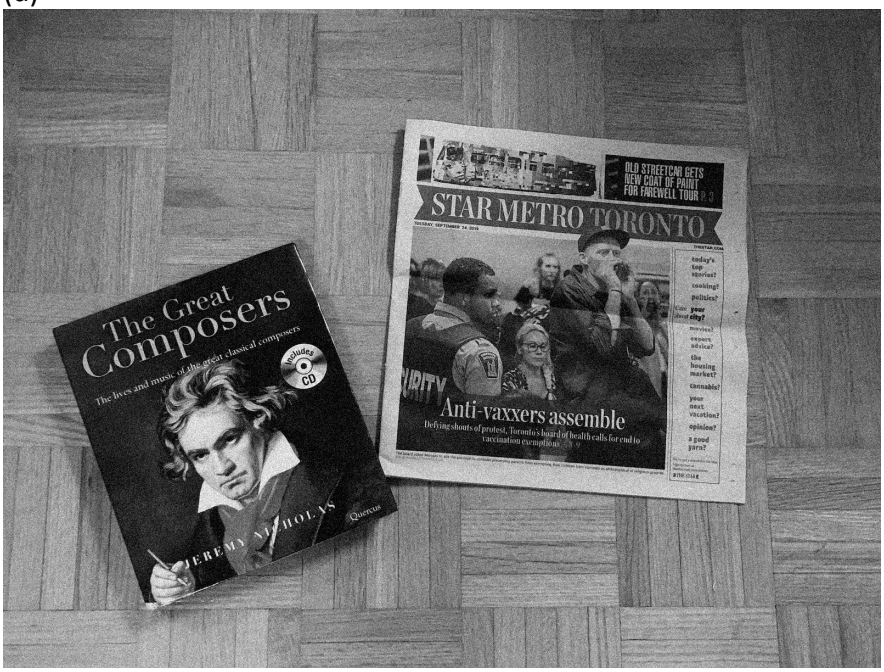


Image1 with Gaussian noise

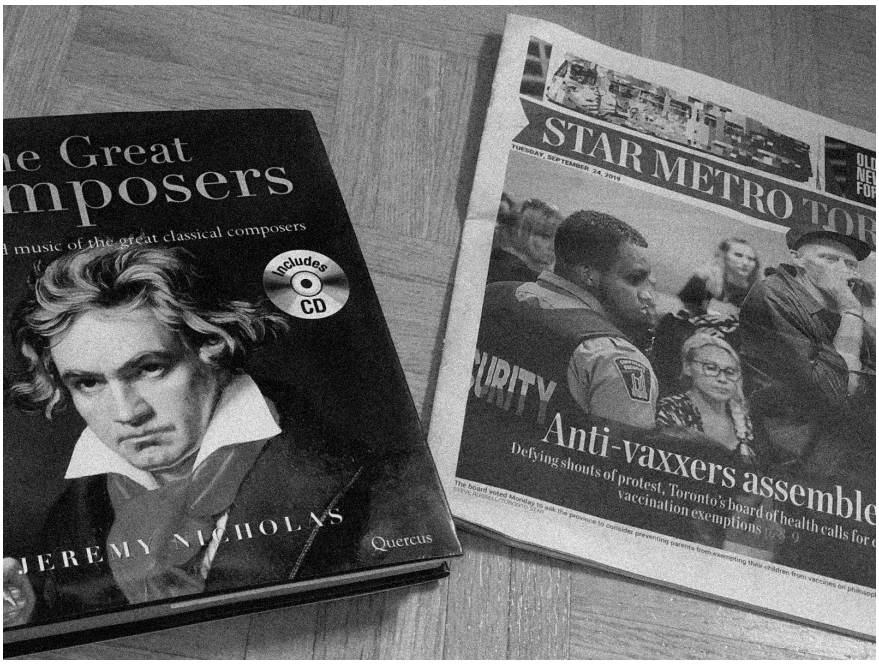
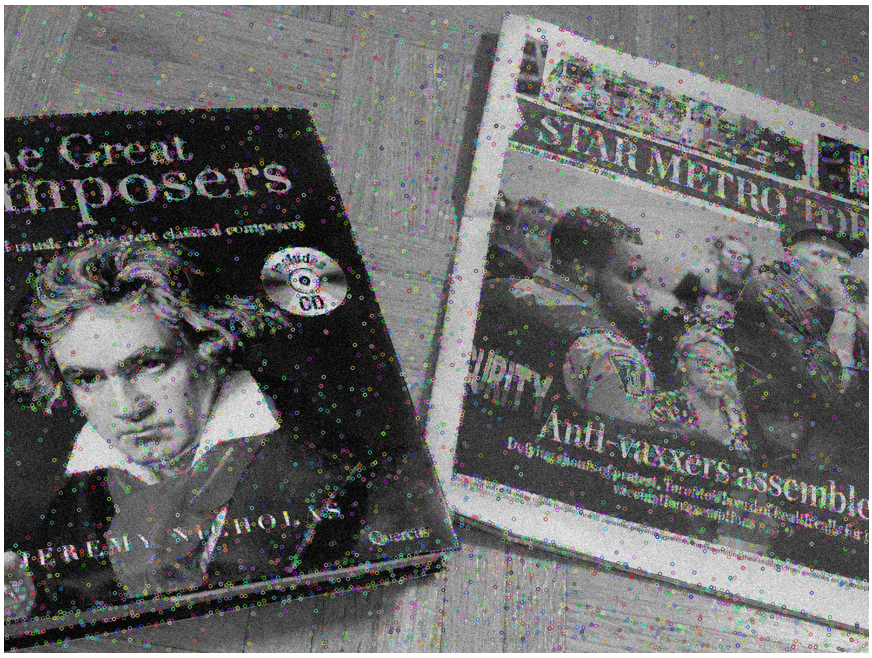


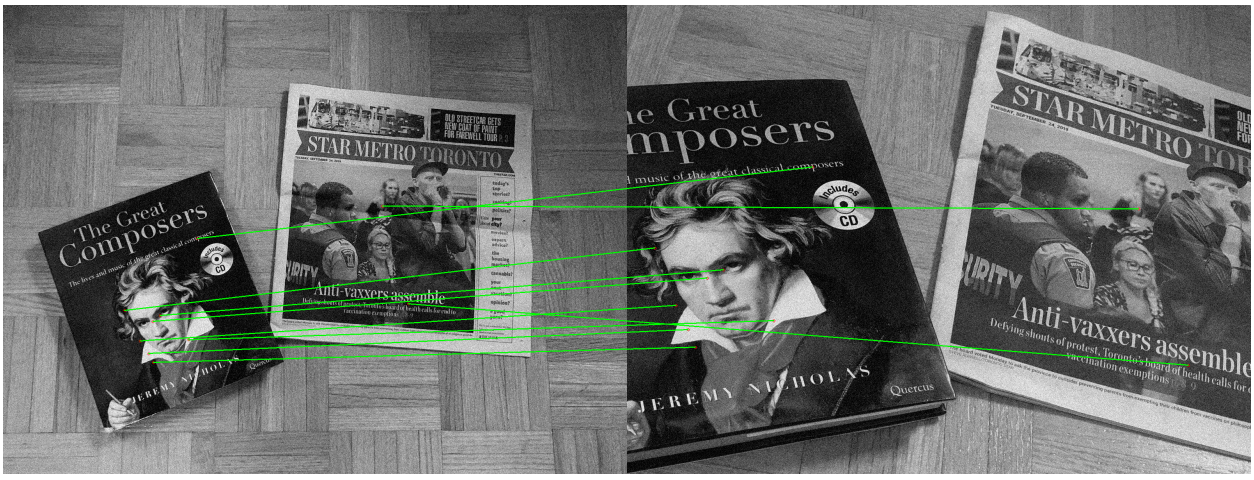
Image2 with Gaussian noise

Gaussian noise image1 with keypoints



Gaussian noise image2 with keypoints

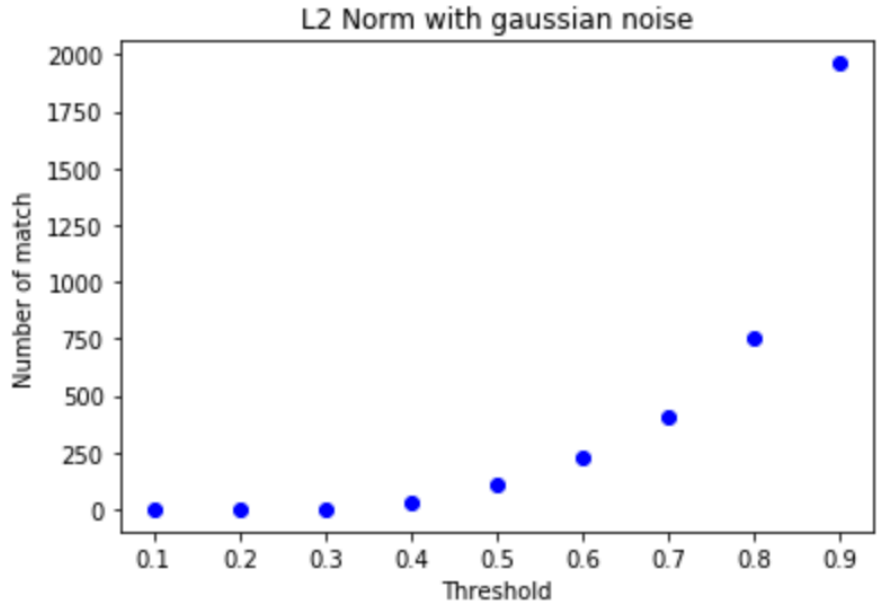
In terms of detection, the distribution of key points was more irregular after applying Gaussian noise. We can look at corners and edges to compare the difference. Firstly, the number of key points obviously decreased along the edges and corners in the image with gaussian noise. Also, we found many key points appear at very strange location, for example, the top left floor and the cover of book, those locations do not look like interesting points. In general, the distribution of key points concentrated around edge and corner in original image, however, more scatter in noisy image.

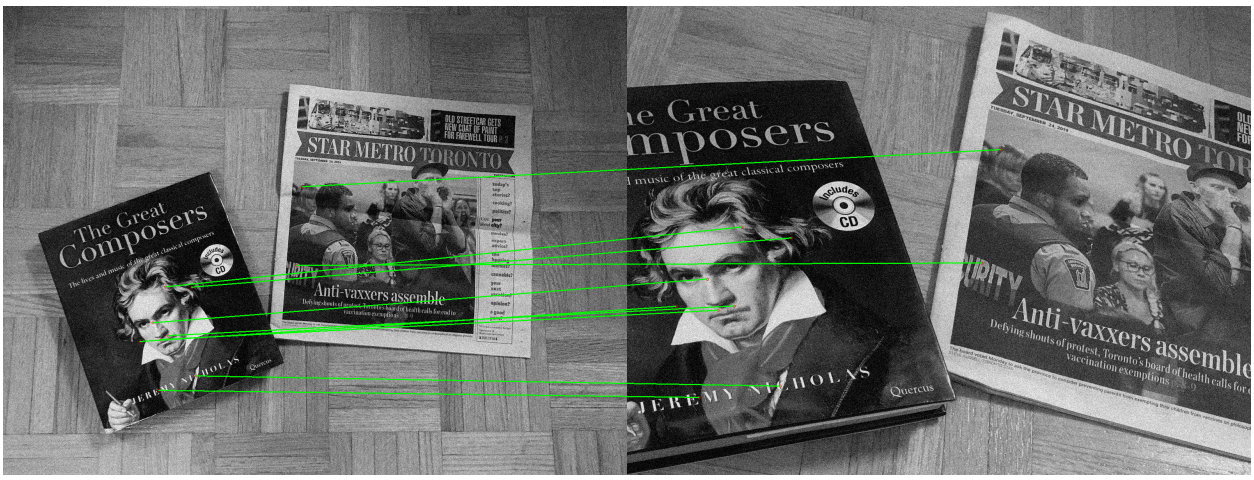


L2-norm with Gaussian noise

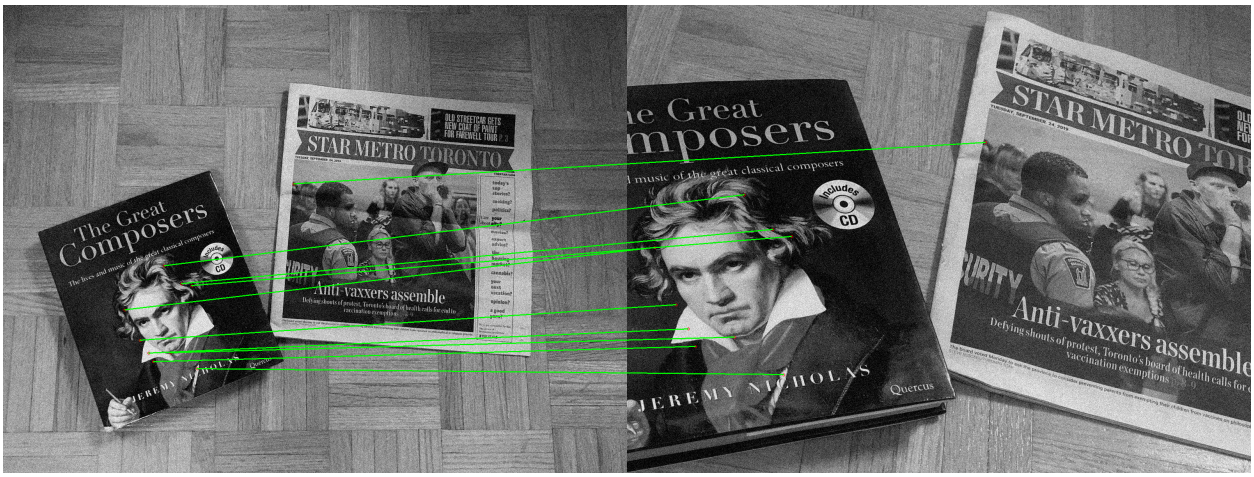
In terms of matching, I found that the noise also affects a lot. The top 10 matches almost changed completely. The keypoints of gaussian noise image become non-representative than

the original image. Also, the number of matches plot shown below demonstrated that the number of matches reduced by half than original image. (except unmeaningful threshold 0.9).



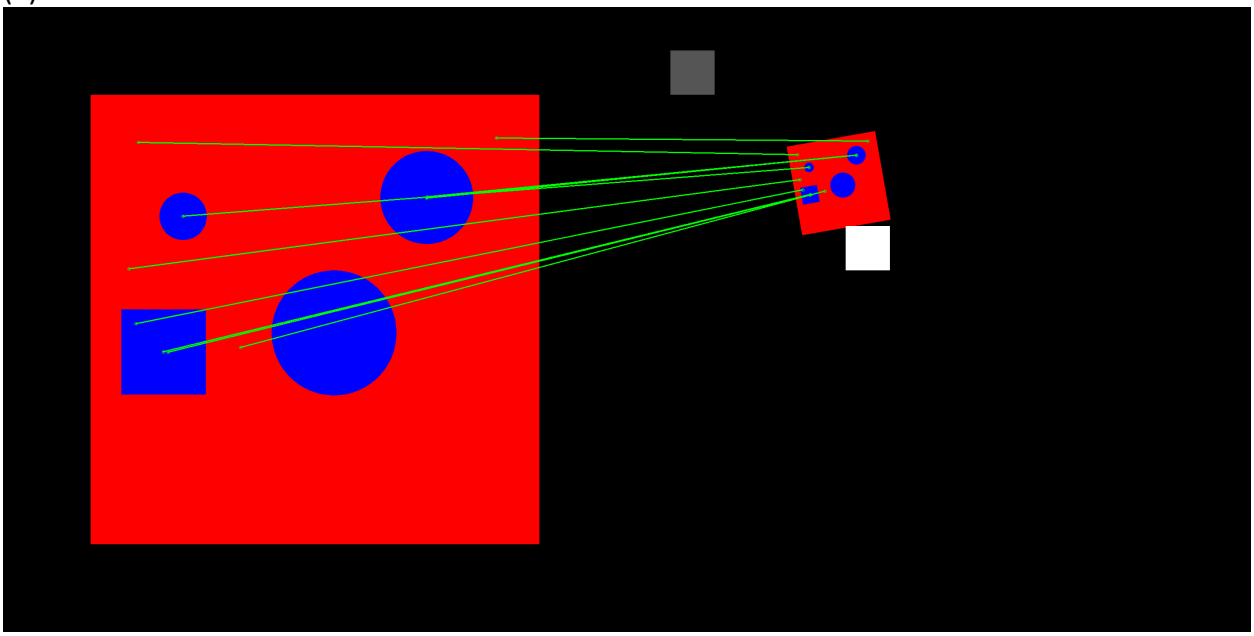


L1-norm with Gaussian noise



L3-norm with Gaussian noise

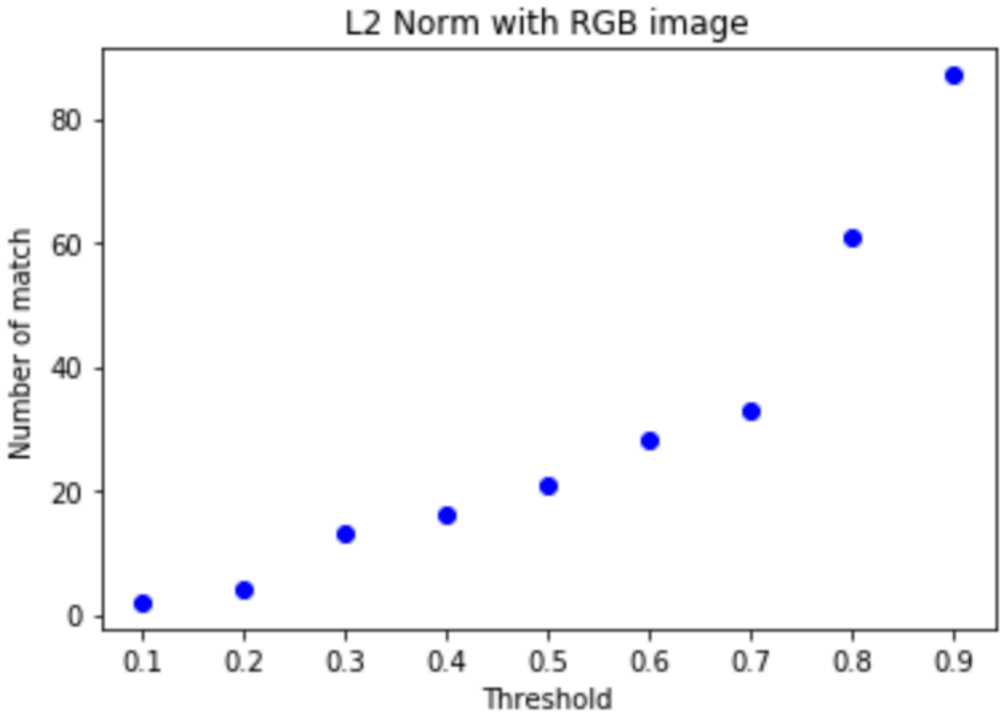
(e)



RGB image keypoints matching

I just separated channels from RGB image. Then, I use the same method for each channel individually and get corresponding keypoints as well as their distance ratio. Finally, I combine the results from three channels and find the top 10 matches among all these three channels.

The rationale is that the original SIFT method will convert image to gray scale first, and then find keypoints and descriptors, however, it may lose some color information. In other words, some key points may only appear at specific color channel, so I treat channels separately.



Code:

```
template = cv2.imread('/content/drive/My Drive/colourTemplate.png')
search = cv2.imread('/content/drive/My Drive/colourSearch.png')

template_blue_channel = template[:,:,0]
template_green_channel = template[:,:,1]
template_red_channel = template[:,:,2]

search_blue_channel = search[:,:,0]
search_green_channel = search[:,:,1]
search_red_channel = search[:,:,2]

result0 = sift_matching(template_blue_channel,search_blue_channel,0.8,2)
result1 = sift_matching(template_green_channel,search_green_channel,0.8,2)
result2 = sift_matching(template_red_channel,search_red_channel,0.8,2)

new_result = {**result0, **result1, **result2}
```