**CSC420 Assignment2**

**Name: Yilin Qu**

**Student number: 1002995734**

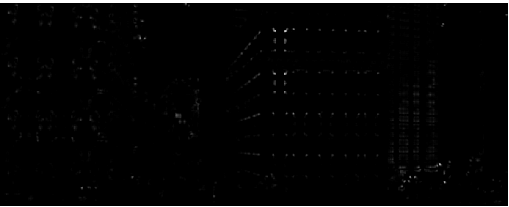
**Utorid: quyilin**

**Question 1:**

**(a)**

**Harris**

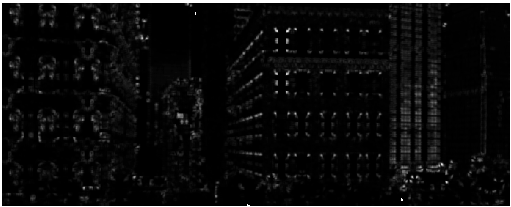
**def Harris**(img):  
 gray = cv.cvtColor(img, cv.COLOR\_BGR2GRAY)  
 blur = cv.GaussianBlur(gray, (5, 5), 7)  
 Ix = cv.Sobel(blur, cv.CV\_64F, 1, 0, ksize=5)  
 Iy = cv.Sobel(blur, cv.CV\_64F, 0, 1, ksize=5)  
 IxIy = np.multiply(Ix, Iy)  
 Ix2 = np.multiply(Ix, Ix)  
 Iy2 = np.multiply(Iy, Iy)  
  
 Ix2\_blur = cv.GaussianBlur(Ix2, (7, 7), 10)  
 Iy2\_blur = cv.GaussianBlur(Iy2, (7, 7), 10)  
 IxIy\_blur = cv.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)  
 # plt.subplot(1, 2, 1), plt.imshow(img), plt.axis('off'), plt.show()  
 # plt.subplot(1, 2, 2), plt.imshow(R, cmap='gray'), plt.axis('off'), plt.show()  
 # adding threshold  
 t = 0.01 \* R.max()  
 **for** i **in** range(R.shape[0]):  
 **for** j **in** range(R.shape[1]):  
 **if** R[i, j] < t:  
 R[i, j] = 0  
 # adding threshold  
  
 **return** R

****

**Brown**

**def Brown**(img):  
 gray = cv.cvtColor(img, cv.COLOR\_BGR2GRAY)  
 blur = cv.GaussianBlur(gray, (5, 5), 7)  
 Ix = cv.Sobel(blur, cv.CV\_64F, 1, 0, ksize=5)  
 Iy = cv.Sobel(blur, cv.CV\_64F, 0, 1, ksize=5)  
 IxIy = np.multiply(Ix, Iy)  
 Ix2 = np.multiply(Ix, Ix)  
 Iy2 = np.multiply(Iy, Iy)  
  
 Ix2\_blur = cv.GaussianBlur(Ix2, (7, 7), 10)  
 Iy2\_blur = cv.GaussianBlur(Iy2, (7, 7), 10)  
 IxIy\_blur = cv.GaussianBlur(IxIy, (7, 7), 10)  
  
 det = np.multiply(Ix2\_blur, Iy2\_blur) - np.multiply(IxIy\_blur, IxIy\_blur)  
 trace = Ix2\_blur + Iy2\_blur  
  
 R = np.divide(det, trace)

**return** R

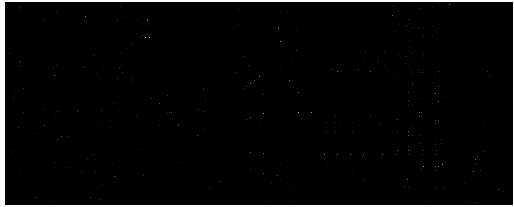
****

It seems that Brown will detect more keypoints than Harris. It is because in Harris we need to choose a threshold, while in Brown we only use the value of det(img) and trace(img), therefore the result won’t be influenced by different thresholds.

**(b)Non-Max-Suppression with circular-like kernel**

**def non\_max\_sup**(img, r):  
  
 img = Brown(img)  
  
 Width, Height = img.shape[0], img.shape[1]  
 padded = np.zeros((Width + 2\*r, Height + 2\*r))  
 padded[r: Width+r, r:Height+r] = img[:, :]  
 **for** i **in** range(Width):  
 **for** j **in** range(Height):  
 **if** img[i, j] != padded[i:i+2\*r+1, j:j+2\*r+1].max():  
 img[i, j] = 0  
  
 **return** img

**R = 1**

****

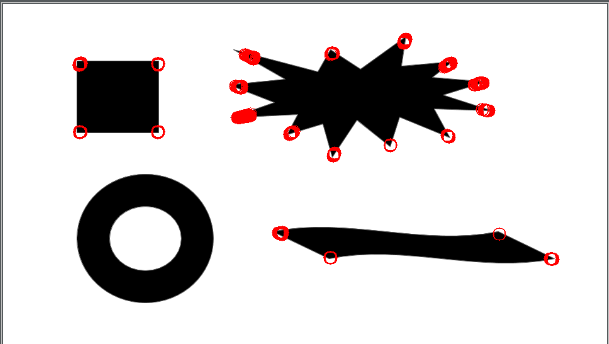
**R = 3**

****

With the increasing of R value, we will detect a more wide space over our image, so the number of points is decreasing, meaning that they are more likely to be corners.

**(c) Blob detection and draw keypoints**

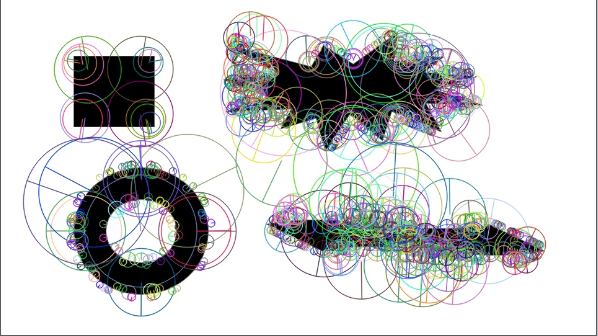
**def blob\_detector**(name): #  
  
 img = cv.imread(name, 0)  
 img = img.astype(np.float32)  
 M, N = img.shape  
 image = np.zeros((M, N))  
 Sigma\_array, LoG\_array = [], []  
 sigma = 2  
 number\_of\_layers = 15  
  
 k = 2 \*\* 0.5  
 Sigma\_array.append(sigma)  
 **for** i **in** range(number\_of\_layers - 1):  
 Sigma\_array.append(Sigma\_array[-1] \* k)  
 **for** i **in** range(number\_of\_layers):  
 blur = cv.GaussianBlur(img, (5, 5), Sigma\_array[i])  
 LoG\_array.append(ndimage.gaussian\_laplace(blur, Sigma\_array[i]))  
  
 r = 1  
  
  
 Width, Height = img.shape[0], img.shape[1]  
 padded = np.zeros((Width + 2\*r, Height + 2\*r))  
 padded[r: Width+r, r:Height+r] = img[:, :]  
 Mm, Nn = padded.shape  
  
 Pr\_array = []  
  
 Pr\_array.append(np.zeros((Mm, Nn)))  
 **for** image **in** LoG\_array:  
 Width, Height = image.shape[0], image.shape[1]  
 padded = np.zeros((Width + 2 \* r, Height + 2 \* r))  
 padded[r: Width + r, r:Height + r] = image[:, :]  
 Pr\_array.append(padded)  
  
 Pr\_array.append(np.zeros((Mm, Nn)))  
 keypoints = {}  
 **for** l **in** range(1, len(Pr\_array) - 1):  
 print(Pr\_array[l].max(), Pr\_array[l].min())  
 **for** i **in** range(M):  
 **for** j **in** range(N):  
 **if** Pr\_array[l][i, j] == max(Pr\_array[l][i:i+2 \* r + 1, j:j+2 \* r + 1].max(), Pr\_array[l-1][i:i+2 \* r + 1, j:j+2 \* r + 1].max(), Pr\_array[l+1][i:i+2 \* r + 1, j:j+2 \* r + 1].max()) **and** \  
 Pr\_array[l][i, j] >= 11 **and** Pr\_array[l][i, j] <= 21:  
 **if**((j, i) **not in** keypoints):  
 keypoints[(j, i)] = Sigma\_array[l-1]  
 **return** keypoints  
  
**def drawkp**(keypoints):  
 image = cv.imread("synthetic.png")  
 **for** item **in** keypoints.keys():  
 cv.circle(image, (item[0], item[1]), radius= 5 \* int(keypoints[item]), color=(0, 0, 255), thickness=1)  
 plt.subplot(1, 2, 2), plt.imshow(image, cmap='gray'), plt.axis('off'), plt.show()  
 cv.imwrite("q1c.png", image)



**(d)**

**Use SURF detection:**

**import** cv2  
  
**def SURF\_building**(img):  
  
 gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 surf = cv2.xfeatures2d.SURF\_create()  
 kp = surf.detect(gray, **None**)  
 img = cv2.drawKeypoints(gray, kp, img, flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)  
 cv2.imwrite('building\_keypoints.jpg', img)  
  
img2 = cv2.imread("building.jpg")  
SURF\_building(img2)  
  
**def SURF\_synthetic**(img):  
  
 gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 surf = cv2.xfeatures2d.SURF\_create()  
 kp = surf.detect(gray, **None**)  
 img = cv2.drawKeypoints(gray, kp, img, flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)  
 cv2.imwrite('synthetic\_keypoints.jpg', img)  
  
  
# img1 = cv2.imread("synthetic.png", 0)  
img3 = cv2.imread("synthetic.png")  
SURF\_synthetic(img3)

****

****

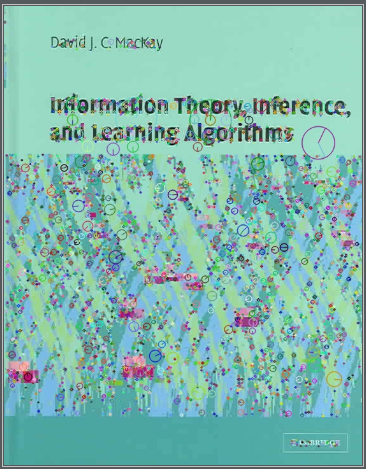
SURF works as follows:

To detect interest points, SURF uses an integer approximation of the Hessian blob detector, which can be computed with 3 integer operations using a precomputed intergral 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.

**Question 2:**

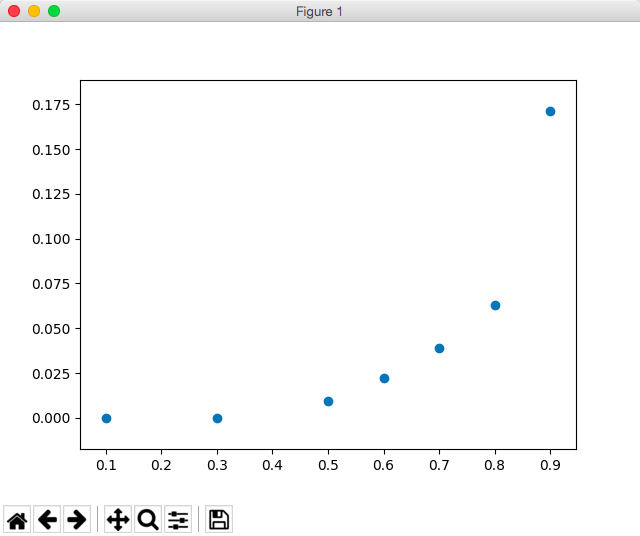
**(a)**

**import** cv2 **as** cv  
  
**def SIFT**(name):  
 img = cv.imread(name)  
 gray = cv.cvtColor(img, cv.COLOR\_BGR2GRAY)  
 sift = cv.xfeatures2d\_SIFT.create()  
 kp = sift.detect(gray, **None**)  
 img = cv.drawKeypoints(img, kp, img, flags=cv.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)  
 cv.imwrite(name + '\_q2\_keypoints.jpg', img)  
  
SIFT("book.jpeg")  
SIFT("findBook.png")

****

**(b)**

**import** numpy **as** np  
**import** matplotlib  
matplotlib.use('TkAgg')  
**import** matplotlib.pyplot **as** plt  
**import** cv2 **as** cv  
  
**def dist**(v1, v2):  
 **return** np.linalg.norm(v1 - v2)  
  
  
**def SIFT\_matching**(name1, name2, threshold):  
 img1 = cv.imread(name1)  
 gray1 = cv.cvtColor(img1, cv.COLOR\_BGR2GRAY)  
 sift1 = cv.xfeatures2d.SIFT\_create()  
  
 img2 = cv.imread(name2)  
 gray2 = cv.cvtColor(img2, cv.COLOR\_BGR2GRAY)  
 sift2 = cv.xfeatures2d.SIFT\_create()  
  
 kp1, des1 = sift1.detectAndCompute(gray1, **None**)  
 kp2, des2 = sift2.detectAndCompute(gray2, **None**)  
 # (\_, axes) = plt.subplots(1, 2)  
 # axes[0].set\_title('threshold {}'.format(threshold))  
 # axes[1].set\_title('matches {}'.format(np.sum(actual\_match > -1)))  
  
 result = np.zeros((len(kp1), 2)) # keep track of all distances  
 pairs = np.zeros((len(kp1), 2))  
  
 matched\_kp1 = {}  
  
 matches = 0  
 **for** i **in** range(len(kp1)):  
 temp = []  
 **for** j **in** range(len(kp2)):  
 # item = (distance, des1, des2, i, j)  
 temp.append((dist(des1[i, :], des2[j, :]), (kp1[i].pt, kp2[j].pt)))  
 temp.sort()  
 **if** temp[0][0] / temp[1][0] < threshold:  
 matches += 1  
 matched\_kp1[temp[0][0] / temp[1][0]] = (temp[0][1][0], temp[0][1][1])  
 **return** matches / len(kp1), matched\_kp1

****

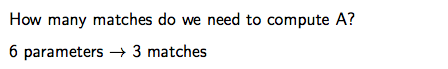
Notice that: it seems the best value is around 0.8. Although the match rate for threshold=0.9 is higher, the actual matches contain some more “false” matches.

**(c) Affine and**

# -----------Q2c----------  
**def affine**(matched\_kp1, k):  
 order = sorted(matched\_kp1.keys())  
 # list of keys in order  
 # print(matched[order[0]], matched[order[1]])  
 # print(matched[order[0]][0][0], matched[order[0]][0][1])  
 # print(matched[order[0]][1][0], matched[order[0]][1][1])  
 P = np.array([[matched\_kp1[order[0]][0][0], matched\_kp1[order[0]][0][1], 0, 0, 1, 0],  
 [0, 0, matched\_kp1[order[0]][0][0], matched\_kp1[order[0]][0][1], 0, 1]])  
 P\_prime = np.array([[matched\_kp1[order[0]][1][0], matched\_kp1[order[0]][1][1]]])  
 # print(P, P\_prime)  
  
 **for** i **in** range(1, k):  
 P = np.concatenate((P, (np.array([[matched\_kp1[order[i]][0][0], matched\_kp1[order[i]][0][1], 0, 0, 1, 0],  
 [0, 0, matched\_kp1[order[i]][0][0], matched\_kp1[order[i]][0][1], 0, 1]]))), axis=0)  
 P\_prime = np.concatenate((P\_prime, np.array([[matched\_kp1[order[i]][1][0], matched\_kp1[order[i]][1][1]]])),  
 axis=1)  
 # print(P.shape, P\_prime.shape)  
 # Use pinv for inverse matrix  
 P\_prime = P\_prime.reshape((2\*k, 1))  
 PP = np.linalg.pinv(P)  
 **return** np.matmul(PP, P\_prime)  
# -----------Q2c----------

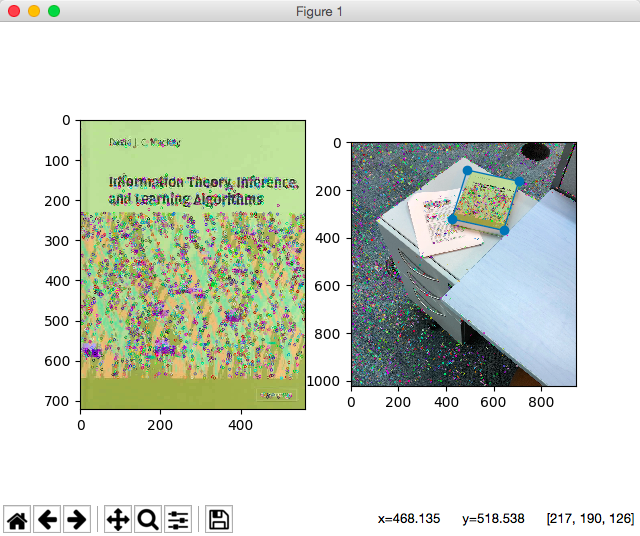
**Minimum k is 3**. Recall the format of Affine transformation, we have (a, b, c, d, e, f) 6 unknowns in total, which means we need at least 6 equations, implies we need to choose **3** different (xi, yi) and (xi’, yi’) to make the matrix invertible.

Also according to slides:

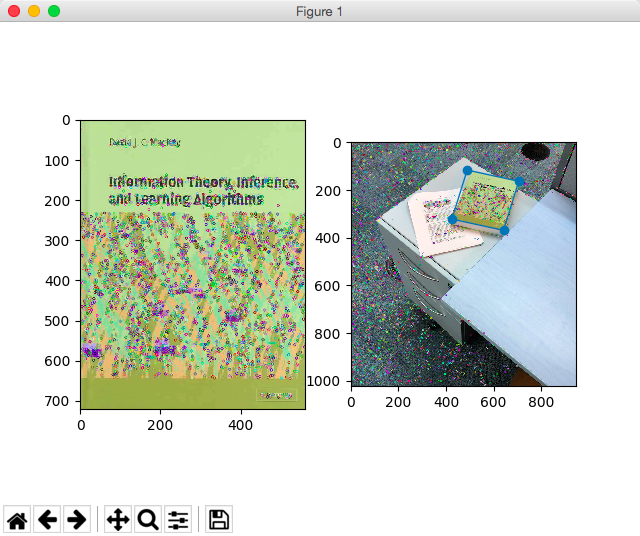


**(d) Visualize affine**

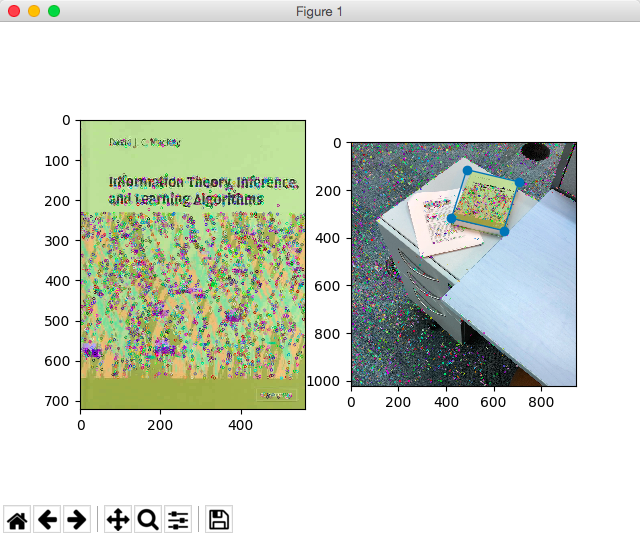
# -----------Q2d----------  
**def visualize\_affine**(name1, name2, mapping):  
 image1 = cv.imread(name1)  
 gray1 = cv.cvtColor(image1, cv.COLOR\_BGR2GRAY)  
 sift1 = cv.xfeatures2d.SIFT\_create()  
  
 image2 = cv.imread(name2)  
 gray2 = cv.cvtColor(image2, cv.COLOR\_BGR2GRAY)  
 sift2 = cv.xfeatures2d.SIFT\_create()  
  
 kp1, des1 = sift1.detectAndCompute(gray1, **None**)  
 kp2, des2 = sift2.detectAndCompute(gray2, **None**)  
  
 M, N = image1.shape[0], image1.shape[1]  
 fig = plt.figure()  
  
 # reshape [a, b, c, d, e, f] to [[a, b, e],  
 # [c, d, f]]  
 # print(mapping)  
 c, d, e = mapping.item(2), mapping.item(3), mapping.item(4)  
 mapping[2] = e  
 mapping[3] = c  
 mapping[4] = d  
 mapping = mapping.reshape(2, 3)  
 # print(mapping)  
 # reshape [a, b, c, d, e, f] to [[a, b, e],  
 # [c, d, f]]  
  
 ax2 = fig.add\_subplot(122)  
 ax1 = fig.add\_subplot(121)  
 ax1.imshow(image1, cmap='Greys\_r')  
 ax2.imshow(image2, cmap='Greys\_r')  
  
 template = cv.drawKeypoints(image1, kp1, **None**)  
 image = cv.drawKeypoints(image2, kp2, **None**)  
 ax1.imshow(template)  
 ax2.imshow(image)  
  
 x = np.array([[0, 0], [0, M - 1], [N - 1, 0], [N - 1, M - 1]])  
  
 x = np.append(x, np.ones((4, 1)), axis=1)  
 # print(x)  
 transformed\_x = np.dot(mapping, x.T)  
 # print(transformed\_x)  
 ax2.scatter(transformed\_x[0, :], transformed\_x[1, :])  
  
 line = matplotlib.lines.Line2D((transformed\_x[0, 0], transformed\_x[0, 1]), (transformed\_x[1, 0], transformed\_x[1, 1]), linewidth=1)  
 ax2.add\_line(line)  
 line = matplotlib.lines.Line2D((transformed\_x[0, 1], transformed\_x[0, 3]), (transformed\_x[1, 1], transformed\_x[1, 3]), linewidth=1)  
 ax2.add\_line(line)  
 line = matplotlib.lines.Line2D((transformed\_x[0, 3], transformed\_x[0, 2]), (transformed\_x[1, 3], transformed\_x[1, 2]), linewidth=1)  
 ax2.add\_line(line)  
 line = matplotlib.lines.Line2D((transformed\_x[0, 0], transformed\_x[0, 2]), (transformed\_x[1, 0], transformed\_x[1, 2]), linewidth=1)  
 ax2.add\_line(line)  
  
 plt.show()  
# -----------Q2d----------

****

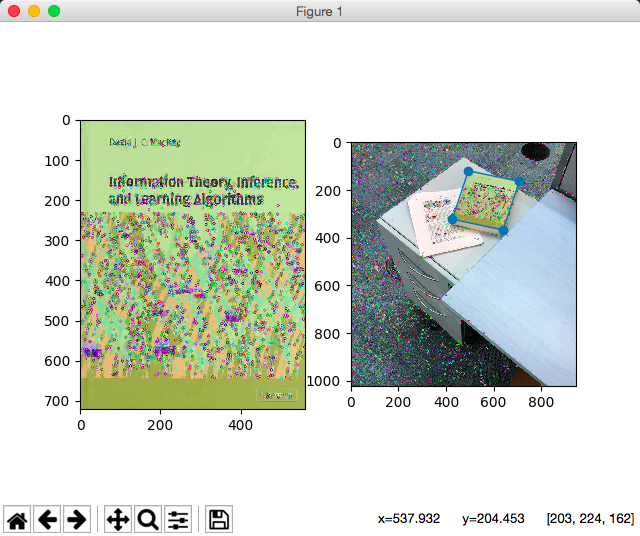
**k = 3**

****

**k = 10**

****

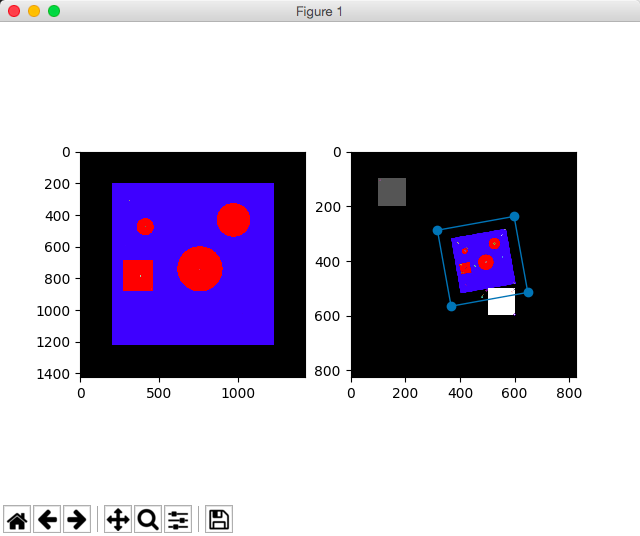
**k = 20**

****

**k = 30**

**(e) Color SIFT**

# -----------Q2e----------  
**def color\_SIFT**(name1, name2):  
 temp = cv.imread(name1)  
 find = cv.imread(name2)  
  
 M1, N1 = temp.shape[0], temp.shape[1]  
 M2, N2 = find.shape[0], temp.shape[1]  
  
 temp\_g = cv.cvtColor(temp, cv.COLOR\_BGR2GRAY)  
 find\_g = cv.cvtColor(find, cv.COLOR\_BGR2GRAY)  
 sift = cv.xfeatures2d.SIFT\_create()  
  
 kp\_temp = sift.detectAndCompute(temp\_g, **None**)[0]  
 kp\_find = sift.detectAndCompute(find\_g, **None**)[0]  
  
 temp\_r, temp\_g, temp\_b = temp[0:, 0:, 2], temp[0:, 0:, 1], temp[0:, 0:, 0]  
 find\_r, find\_g, find\_b = find[0:, 0:, 2], find[0:, 0:, 1], find[0:, 0:, 0]  
  
 tr = sift.compute(temp\_r, kp\_temp)[1]  
 tg = sift.compute(temp\_g, kp\_temp)[1]  
 tb = sift.compute(temp\_b, kp\_temp)[1]  
 fr = sift.compute(find\_r, kp\_temp)[1]  
 fg = sift.compute(find\_g, kp\_temp)[1]  
 fb = sift.compute(find\_b, kp\_temp)[1]  
  
 des1, des2 = [], []  
 # init two descriptors  
 **for** i **in** range(len(kp\_temp)):  
 aa = np.concatenate(tr[i], tg[i], 0)  
 aa = np.concatenate(aa, tb[i], 0)  
 des1.append(aa)  
  
 matched\_kp1 = {}  
 **for** i **in** range(len(kp\_find)):  
 bb = np.concatenate(fr[i], fg[i], 0)  
 bb = np.concatenate(bb, fb[i], 0)  
 des1.append(bb)  
  
 threshold = 0.8  
 matches = 0  
 **for** i **in** range(len(kp\_temp)):  
 temp = []  
 **for** j **in** range(len(kp\_find)):  
 # item = (distance, des1, des2, i, j)  
 temp.append((dist(des1[i, :], des2[j, :]), (kp\_temp[i].pt, kp\_find[j].pt)))  
 temp.sort()  
 **if** temp[0][0] / temp[1][0] < threshold:  
 matches += 1  
 matched\_kp1[temp[0][0] / temp[1][0]] = (temp[0][1][0], temp[0][1][1])  
  
 result = affine(matched\_kp1, k=5)  
 visualize\_affine(name1, name2, result)  
# -----------Q2e----------

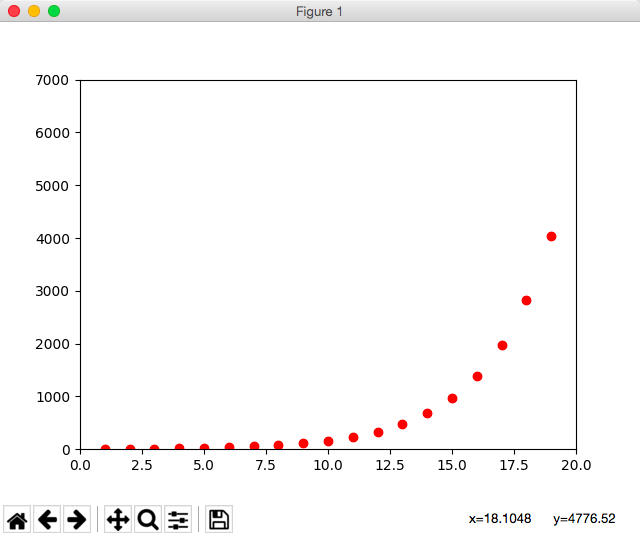
****

**Main idea:** First we find keypoints using SIFT on the grayscale image, then we

isolate 3 colour channels from the template image and make a descriptor for each of them on every keyppint, then concatenate them back together(length = 128 \* 3) and apply SIFT matching and affine transformation on the new-concatenated descriptor.

**Question 3:**

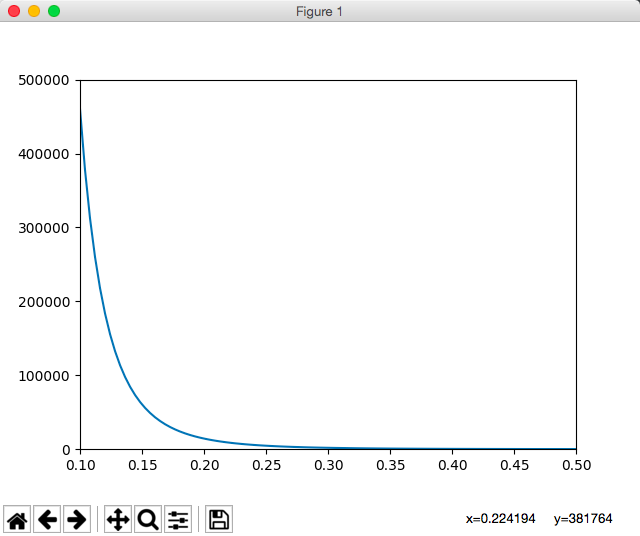
**(a)**

****

**def Q3\_a**(x):  
 **return** (np.log(0.01)) / (np.log(1 - pow(0.7, x)))

**(b)**

**def Q3\_b**(x):  
 **return** (np.log(0.01)) / (np.log(1 - pow(x, 5)))

****

**(c): Number of iterations with k = 5 and p = 0.2 is:**

****

And the number of required iterations won’t change. Since k and p have been set and on one specific iteration, the number of agreed sample will not have actual adjustment on the parameters (i.e., k and p), so on the next iteration it will use the same k and p and continue randomly choosing points for comparison. **Therefore the number of iterations won’t change.**