1. explain your implementation

SIFT:

```
40  def get_SIFT_des(image):
41      SIFT_Detector = cv2.SIFT_create()
42      kp, des = SIFT_Detector.detectAndCompute(image, None)
43      return kp, des
```

直接使用 OpenCV 內建的 SIFT Detector 去偵測並計算特徵點

KNN:

```
45  def l2_distance(p1, p2):
46      p1 = np.array(p1)
47      p2 = np.array(p2)
48      dis = np.linalg.norm(p1-p2)
49      return dis
```

首先我宣告了一個函式用來計算兩個向量之間的 L2 距離

```
51  def knn(target, kps, points, k):
52      neighbor_pts = []
53
54      for i, pt in enumerate(points):
55          distance = l2_distance(target, pt)
56          neighbor_pts.append((pt, distance, kps[i]))
57
58      neighbor_pts.sort(key=lambda tup: tup[1])
59      neighbor_pts = neighbor_pts[0:k]
60
61      distances = [i[1] for i in neighbor_pts]
62      neighbor_pt = [i[0] for i in neighbor_pts]
63      neighbor_kps = [i[2] for i in neighbor_pts]
64      return neighbor_pt, distances, neighbor_kps
```

這是我的 KNN 的實作，傳入的 target 代表要找最近鄰的目標點，points 為尋找的點集合。Kps 是指各個 descriptor 所對應到的影響座標，k 是指要找多少個最近鄰。這邊我是用了暴力法去進行計算，算出所有點與 target 的 L2 距離，並且進行排序，取出前 k 個最近的點

Feature matching:

```python
def find_good_match(kp1, kp2, des1, des2, Lowe_ratio = 0.7):
    matches = []
    points_matches = []
    for i, item in enumerate(des1):
        neighbor_pt, distances, neighbor_kps = knn(item, kp2, des2, 2)
        if distances[0] < distances[1] * Lowe_ratio:
            matches.append((item, neighbor_pt[0]))
            points_matches.append((kp1[i], neighbor_kps[0]))

    return matches, points_matches
```

這邊是使用 Lowe's Ratio test 去尋找好的匹配的過程，我使用了暴力法，分別以第一張影像找到的所有特徵點作為 target，去與第二張影像的特徵點進行 KNN(k=2)的匹配，找到之後去比較距離較短的 match 是否低於較長的 match 的某個倍數之下，如果是的話就加入好的 match 當中。

Homography:

```python
def find_homography(pts1, pts2):
    a = np.zeros((8, 9), np.float)
    for i in range(4):
        a[i*2][0] = -pts1[i][0]
        a[i*2][1] = -pts1[i][1]
        a[i*2][2] = -1

        a[i*2+1][3] = -pts1[i][0]
        a[i*2+1][4] = -pts1[i][1]
        a[i*2+1][5] = -1

        a[i*2][6] = pts1[i][0] * pts2[i][0]
        a[i*2][7] = pts1[i][1] * pts2[i][0]

        a[i*2+1][6] = pts1[i][0] * pts2[i][1]
        a[i*2+1][7] = pts1[i][1] * pts2[i][1]

        a[i*2][8] = pts2[i][0]
        a[i*2+1][8] = pts2[i][1]

    u, s, vh = np.linalg.svd(a, full_matrices=True)
    homography = vh[-1].reshape(3, 3)
    homography /= homography[2][2]
    return homography
```

直接依照課本的資訊去建立線性方程組，這邊我使用了較為不同的方程組建立方式，因為 spec 上頭的方法只能先求出前 8 項參數，h33 還得另外求

解，這邊使用了可以一次求解 9 項參數的方式。接著利用 svd 分解找出最小奇異值對應到的向量，此即為 homography matrix 最後將其除以 h33 使 h33 變為 1。

RANSAC:

```
115     for i in range(iter):
116         l = []
117         while len(l) < 4:
118             temp = np.random.randint(match_num)
119             if temp not in l:
120                 l.append(temp)
121         homography = find_homography(pt1[l], pt2[l])
122         a = homography @ pt1.T
123 
124         divide = np.array(a[2], np.float)
125         a = a / divide
126         a = a[0:2]
127         a = a.T
128 
129         temp = np.linalg.norm(a-pt2, axis=1)
130         temp = temp < 10
131         temp = np.sum(temp)
132 
133         if temp / match_num > 0.5:
134             best_homography = homography
135             break
136         if temp > best:
137             best = temp
138             best_homography = homography
139 
140     return best_homography
```

在 ransac 的每一個 iteration 當中，首先我會去隨機抽取 4 組匹配點去計算他們的 homography matrix (line 117 to 121)。接著去計算圖一的特徵點對應到圖二座標系的位置(line 122 to 127)，以及對應過去之後與匹配點座標的距離(line 129)。接著我在 ransac 中定義為 inliers 的定義為圖一特徵點對應過去圖二的距離小於 10 個像素。接著，如果 inlier 佔了全體的 50%以上我就會 return 該 iteration 所算的 homography。同時為避免 inlier 無法達到 50%以上，我還會同時儲存最佳的 homography 最後來 return。

整體流程:

```
img_gray1 = img_to_gray(img1)
img_gray2 = img_to_gray(img2)

kp0, des0 = get_SIFT_des(img_gray1)
kp1, des1 = get_SIFT_des(img_gray2)
```

讀到影像之後先將其轉為灰階圖像並計算他們的 SIFT 特徵

```
matches, points_matches = find_good_match(kp0, kp1, des0, des1)
homography = ransac(points_matches, 50000)
```

使用 lowe;s ratio test 去尋找好的匹配點，並使用 ransac 去計算好的 homography matrix

```
corner = np.array([[0, 0, 1],[0, size[1], 1],[size[0], 0, 1],[size[0], size[1], 1]])
corner = homography @ corner.T
corner = corner / np.array(corner[2], np.float)
x1 = int(min(np.min(corner[0]), 0))
y1 = int(min(np.min(corner[1]), 0))
print(x1, y1)
size = (WIDTH + abs(x1), HEIGHT + abs(y1))
```

使用 homography matrix 以及影像的 4 個角落點來計算需要多少尺寸的影像

```
A = np.array([[1, 0 , -x1],  [0, 1, -y1], [0, 0, 1]], np.float)
homography = A @ homography
img1 = cv2.warpPerspective(img1, homography, size)

A = np.array([[1, 0 ,-x1], [0, 1, -y1], [0, 0, 1]], np.float)
img2 = cv2.warpPerspective(img2, A, size)
img2 = cv2.copyTo(img1, img1, img2)
```

將影像平移並對 image1 套用 homography，貼至 image2 上頭

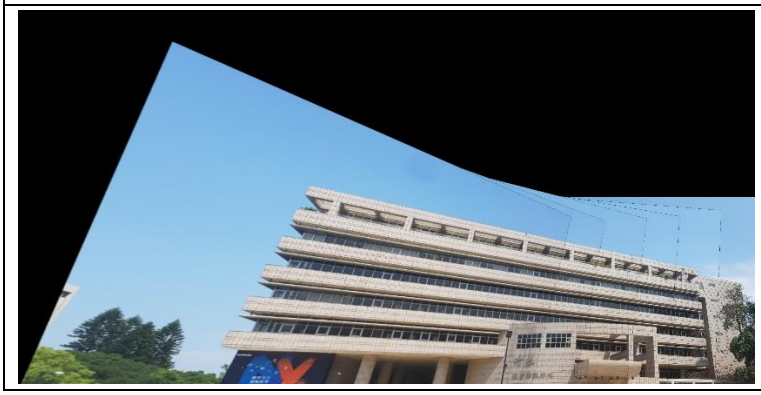
```
creat_im_window("result", img2)
im_show()


cv2.imwrite('result.jpg', img2)
```

最後顯示並寫入檔案之中

2. show the result of stitching 2 image

3. try to stitch more images as you can and compare with them

	2 張

	3 張
	4 張
	5 張
	6 張

	7 張
	8 張
圖片歪到出錯	大於 8 張

從這些圖片的比較中可以發現到愈早出現的圖片到後面的變形程度會愈來愈大，這是因為我始終使用左側的圖片來投射到右側的圖片上頭。並且我沒有使用圓柱投影，而是指使用平面投影的方式，導致其形變愈發增大。

4. stitching at least 4 images clearly