ECE 661 Homework #5

Name: Haoyu Chen
Email: chen1562@purdue.edu
ID Number: 00271-72202

5 Oct 2020

1 Theory Questions

1.1 Conceptually speaking, how do we differentiate between the inliers and the outliers when using RANSAC for solving the homography estimation problem using the interest points extracted from two different photos of the same scene?

In RANSAC algorithm, we take several (N) random experiments. For each of these experiment, we will compute a estimated homography based on only a few points. Using that homography, we will check each point correspondence and see if the computed coordinate lies within acceptable range of the actual coordinate (i.e., distance between computed coordinate and actual coordinate is maller than δ); if so, that point correspondence is considered an inlier. With enough iterations, we shall find a homography estimation with enough inliers and assume that this is a close estimation of the true homography between two images, and inliers are true correspondences between two images. Hence, a refined homography is computed based on all the inlier correspondences.

1.2 As you will see in Lecture 12, the Gradient-Descent (GD) is a reliable method for minimizing a cost function, but it can be excruciatingly slow. At the other extreme, we have the much faster Gauss-Newton (GN) method but it can be numerically unstable. Explain in your own words how the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a method that is reasonably fast and numerically stable at the same time.

Gradient-Descent works best when the estimation is rather further from actual local minimum, but gets much slower near local minimum. Gauss-Newton, however, is rather unstable when further away from local minimum, but could provide fast one-step solution when the estimation is closer to local minimum. Levenberg-Marquardt (LM) combines Gradient-Descent (GD) and Gauss-Newton (GN) by adding a control factor μ. By adjusting μ with iterative trials, one essentially shifts between Gradient-Descent (GD) and Gauss-Newton (GN). When the estimation starts, μ is set to be large so that the step taken is closer to that of Gradient-Descent; then, as the estimation gets closer to actual local minimum, μ decreases and the step size and direction are closer to that of Gauss-Newton, therefore combining the best of two algorithm.

2 Algorithm Implementation

2.1 RANSAC

As briefly stated earlier, In RANSAC algorithm, we take N random experiments hoping to find a close estimation of the actual homography, and then refine the homography by using all inlier correspondence find through the estimated homography. In this homework, I choose the following parameters:

- $\delta = 3$ for the common assumption discussed in lecture, whereas $\sigma = 1$

- $p = 0.99$ for the probability that at least one trial is free of outliers
- For ϵ choice, I tried 0.25, 0.5, and 0.75, and counter-intuitively, 0.5 and 0.75 works better. This could be because the threshold was set too high when establishing correspondences using SIFT+SSD, and percentage of outliers are actually around or higher than 50%, especially in more complex scenes. Do note that even though the percentage of outliers are higher than 50%, they do not form a consistent homography, and the largest set of inliers that agree with an estimated homography would still be the actual inlier set.
- n = 4, for estimating homography from n random point pairs
- $N = \frac{ln(1-p)}{ln(1-(1-\epsilon)^n)}$; N = 70 for $\epsilon = 0.5$, N = 1176 for $\epsilon = 0.75$
- $M = n_{total}(1-\epsilon)$

2.2 Least Square Estimation

Now that we have a group of inlier correspondences, we'll use Linear Least Square to estimate the homography.

As mentioned in hw2, given a homography $\boldsymbol{H}$ that maps $\boldsymbol{x}$ to $\boldsymbol{x'}$, we can write in homogeneous coordinates: $\boldsymbol{x'} = \boldsymbol{H}\boldsymbol{x}$. Let $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$, and $\boldsymbol{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}$ (since homogeneous coordinates is all about ratios, we can set $x_3 = 1$ and $h_33 = 1$). We have $\boldsymbol{x'} = \begin{bmatrix} x_1'' \\ x_2'' \\ x_3'' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$.

Now, if we divide x_1'' and x_2'' by $x_3'' = h_{31}x_1 + h_{32}x_2 + 1$, we now have $\boldsymbol{x'} = \begin{bmatrix} x_1' \\ x_2' \\ 1 \end{bmatrix}$, where

$$x_1' = \frac{h_{11}x_1 + h_{12}x_2 + h_{13}}{h_{31}x_1 + h_{32}x_2 + 1}$$

$$x_2' = \frac{h_{21}x_1 + h_{22}x_2 + h_{13}}{h_{31}x_1 + h_{32}x_2 + 1}$$

i.e., $h_{31}x_1'x_1 + h_{32}x_1'x_2 + x_1' = h_{11}x_1 + h_{12}x_2 + h_{13} \Rightarrow x_1' = h_{11}x_1 + h_{12}x_2 + h_{13} - h_{31}x_1'x_1 - h_{32}x_1'x_2$.

Hence, for each point pair, we could acquire two equations for solving $\boldsymbol{H}$:

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & 1 & 0 & 0 & 0 & -x_1x_1' & -x_2x_1' \\ 0 & 0 & 0 & x_1 & x_2 & 1 & -x_1x_2' & -x_2x_2' \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix}$$

With more than four points, we have essentially an over-determined system, and the linear least-square estimation can be computed using pseudo-inverse

$$\hat{h} = (A^TA)^{-1}A^Tb$$

2.3 Levenberg-Marquardt

Now that we have an estimation for H, we'll use an off-the-shelf implementation of Levenberg-Marquardt algorithm provided by SciPy, to refine the estimation. In this case, the cost function is given by the error between computed coordinates using H and actual coordinates in range space.

2.4 Pixel Mapping

For pixel mapping, I used the value of nearest pixel to computed coordinate. Also, when two images overlap, I choose not to replace the previous layer (i.e., if a pixel already has value assigned to it, the pixel mapping function won't replace it).

3 Images and Results

3.1 Inputs

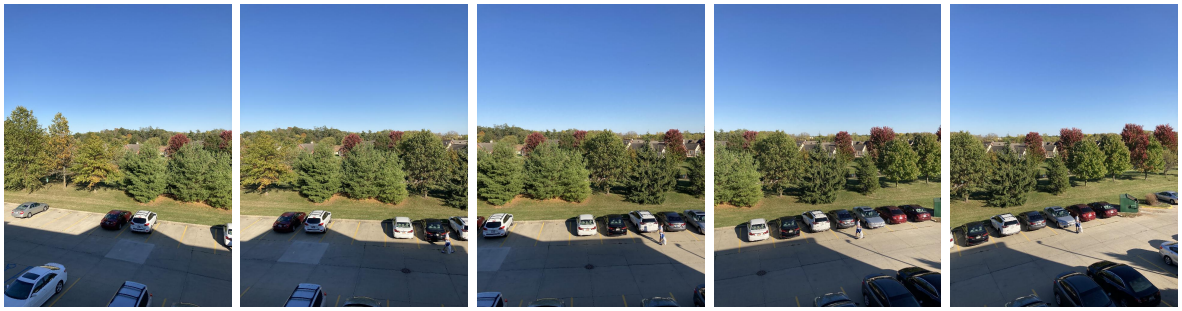

Figure 1: 5 input images

3.2 Correspondences, Inliers, and Outliers

3.2.1 Image 1 and 2

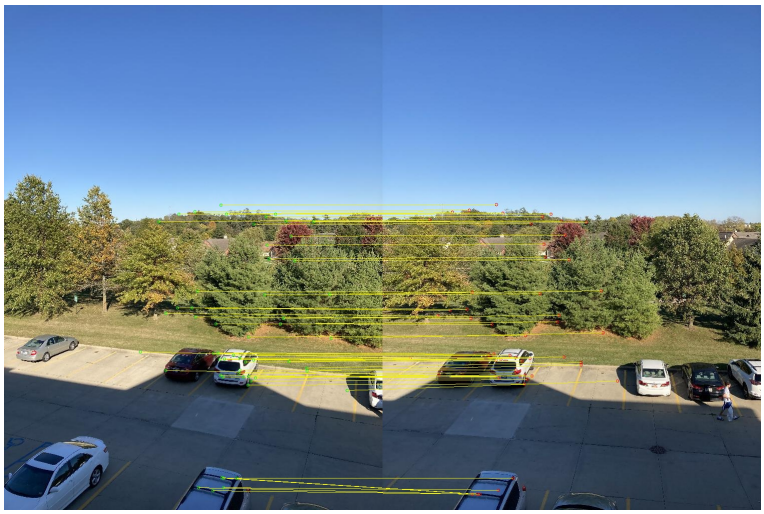

Figure 2: All SIFT correspondences between image 1 and 2

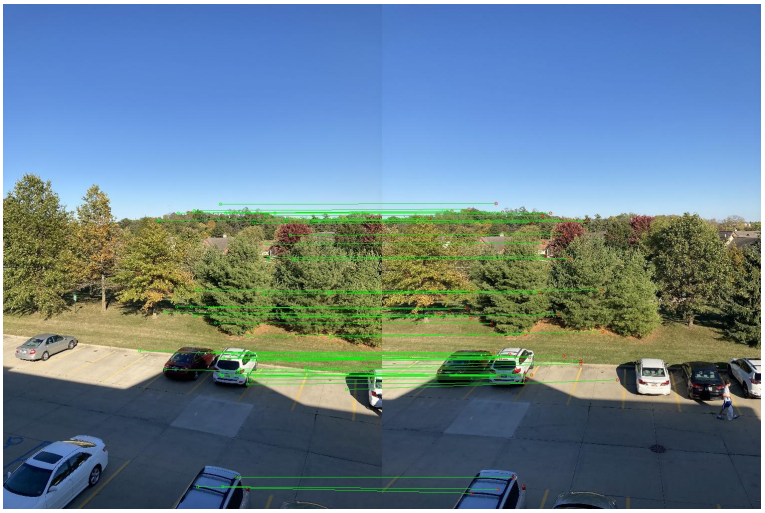

Figure 3: Inliers determined by RANSAC

Figure 4: Outliers rejected by RANSAC

3.2.2 Image 2 and 3

Figure 5: All SIFT correspondences between image 2 and 3

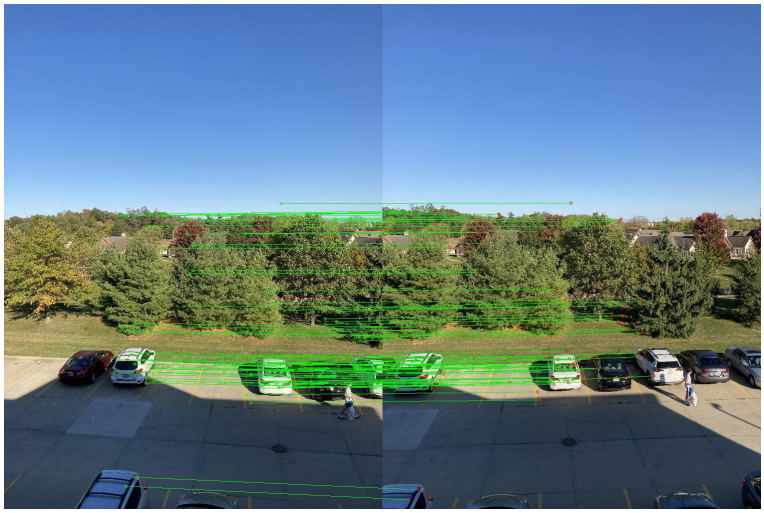

Figure 6: Inliers determined by RANSAC

Figure 7: Outliers rejected by RANSAC

3.2.3 Image 3 and 4

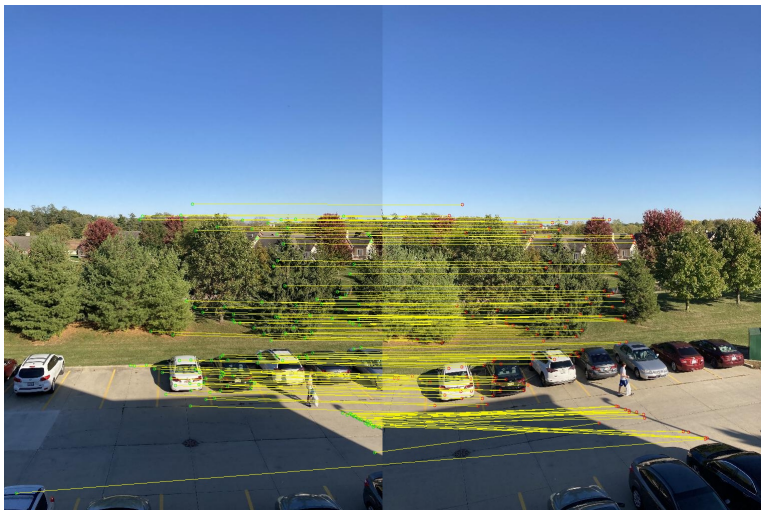

Figure 8: All SIFT correspondences between image 3 and 4

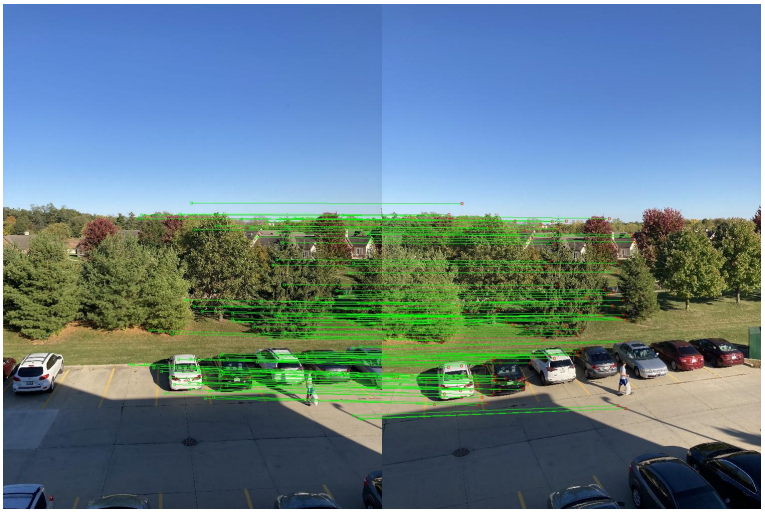

Figure 9: Inliers determined by RANSAC

Figure 10: Outliers rejected by RANSAC

3.2.4 Image 4 and 5

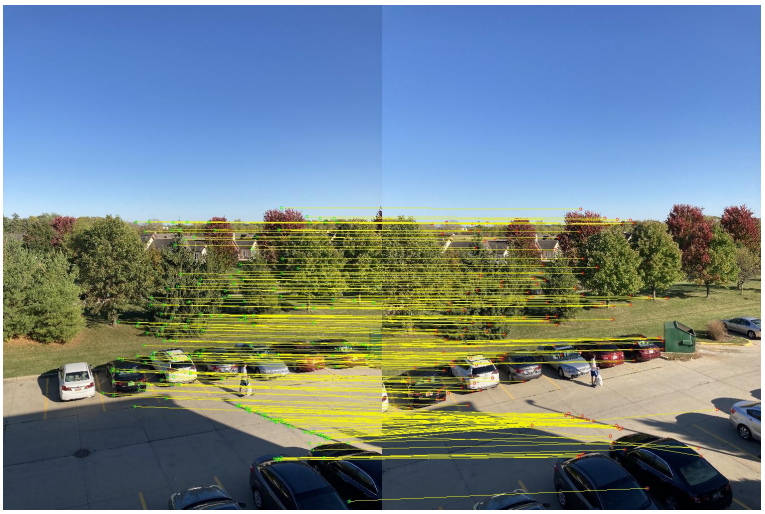

Figure 11: All SIFT correspondences between image 4 and 5

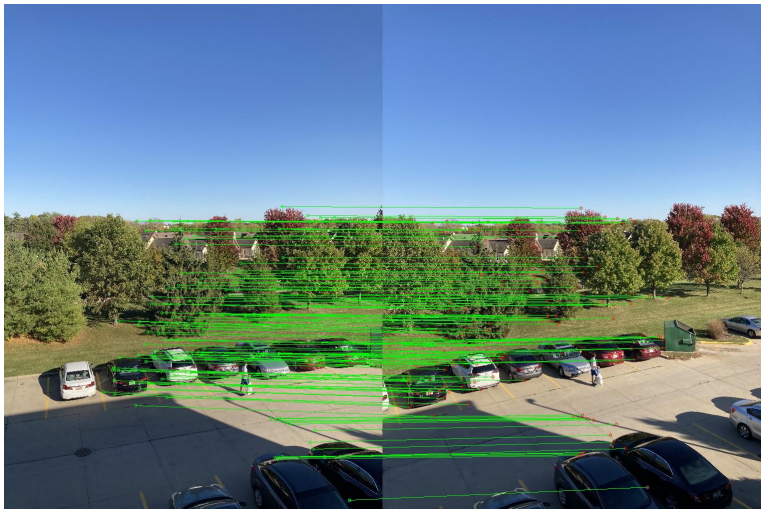

Figure 12: Inliers determined by RANSAC

Figure 13: Outliers rejected by RANSAC

3.3 Output

Figure 14: Output panorama image, without LM refinement

Figure 15: Output panorama image, with LM refinement

4 Remarks

Although the output image from above scene seems pretty good, I did observe a few problems with other scenes, especially indoor scenes.

Alternative scene 1 (Section 5.1) shows example with uneven plane. As the left record and middle one are well-aligned, the right one seems to be a bit uneven, and the result did show a slight discrepancy on the right-most side, with or without LM refinement.

More significantly, in alternative scene 2 (Section 5.2), the two main object (monitor and poster) are on two different plane. Hence, the homography for the two planes across 5 images are different, and the RANSAC algorithm chooses the more dominant plane—monitor. As a result, the panorama image shows a rather consistent monitor plane, but completes fails to connect the poster.

5 Supplemental: alternative scenes and results

5.1 Alternative 1

Figure 16: 5 input images

Figure 17: Sample SIFT correspondences between image 4 and 5

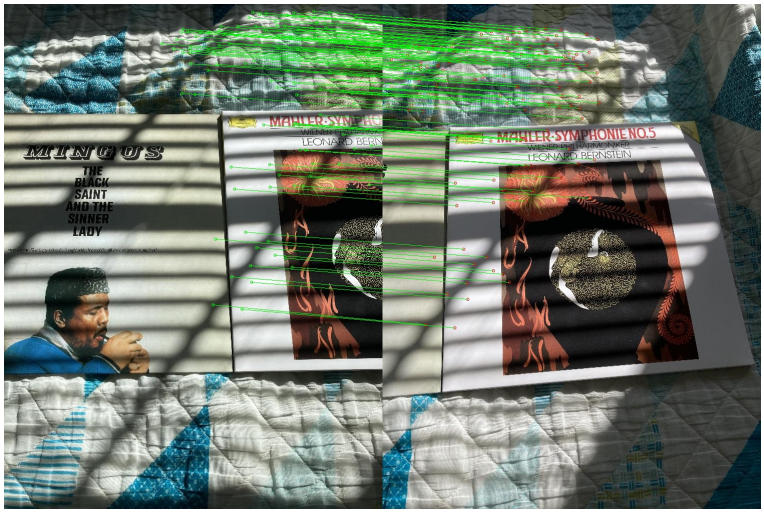

Figure 18: Inliers determined by RANSAC

Figure 19: Outliers rejected by RANSAC

Figure 20: Result panorama image, without LM refinement

Figure 21: Result panorama image, with LM refinement

5.2 Alternative 2

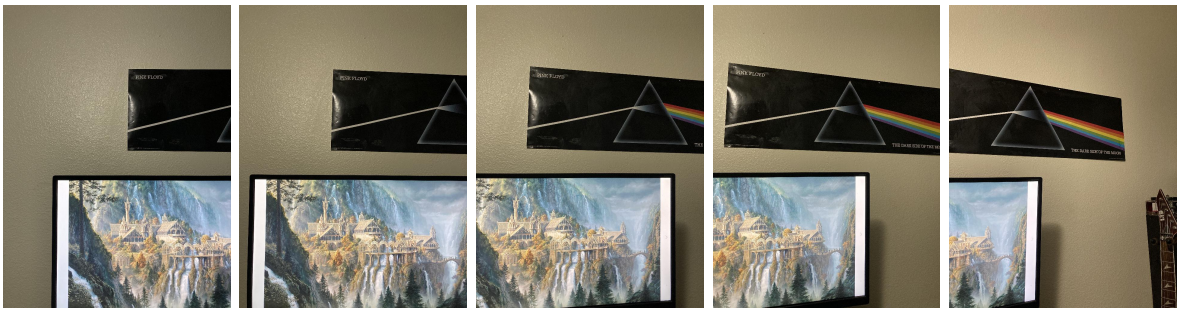

Figure 22: 5 input images

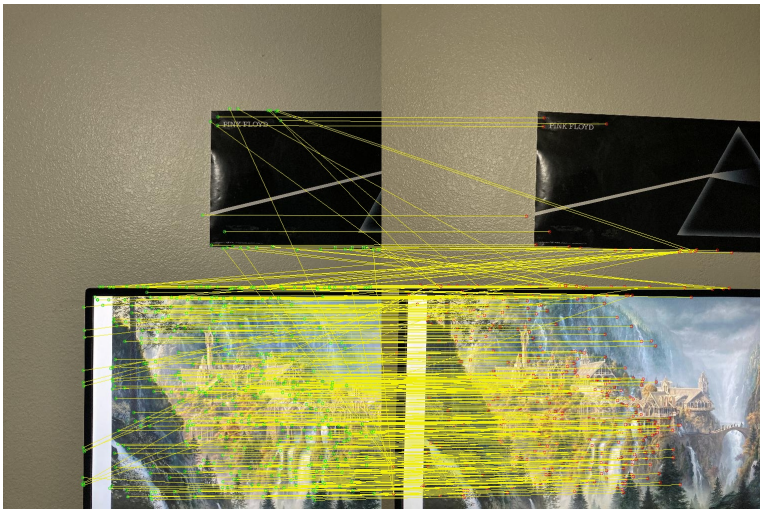

Figure 23: Sample SIFT correspondences between image 1 and 2

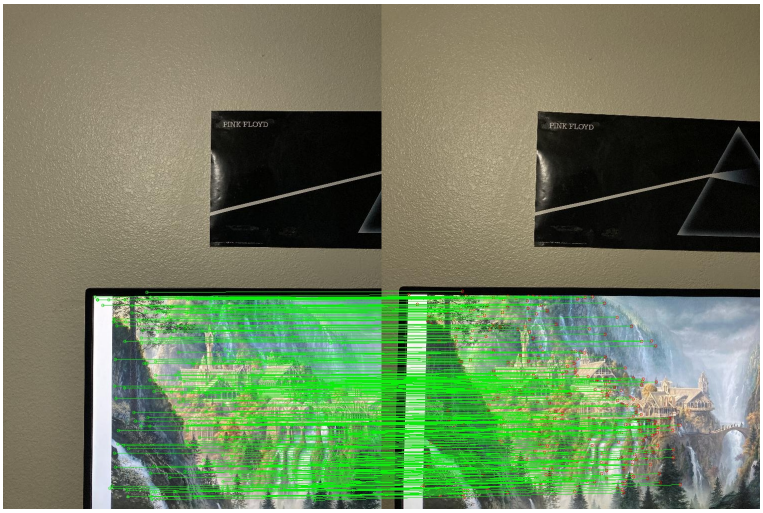

Figure 24: Inliers determined by RANSAC

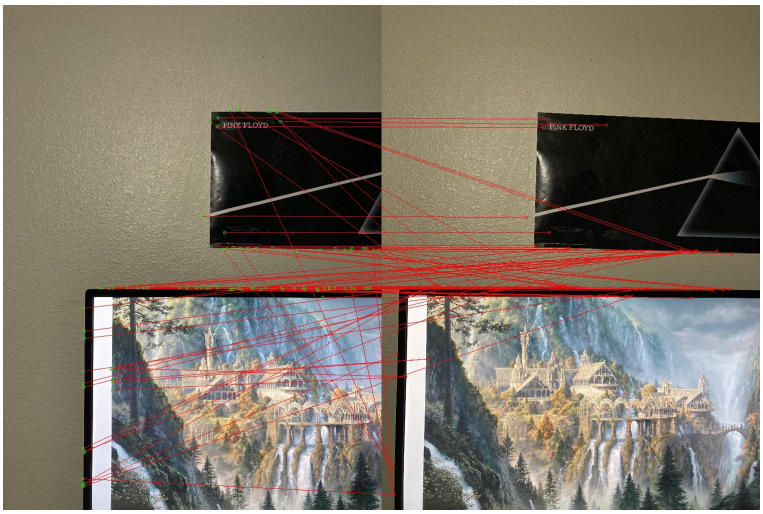

Figure 25: Outliers rejected by RANSAC

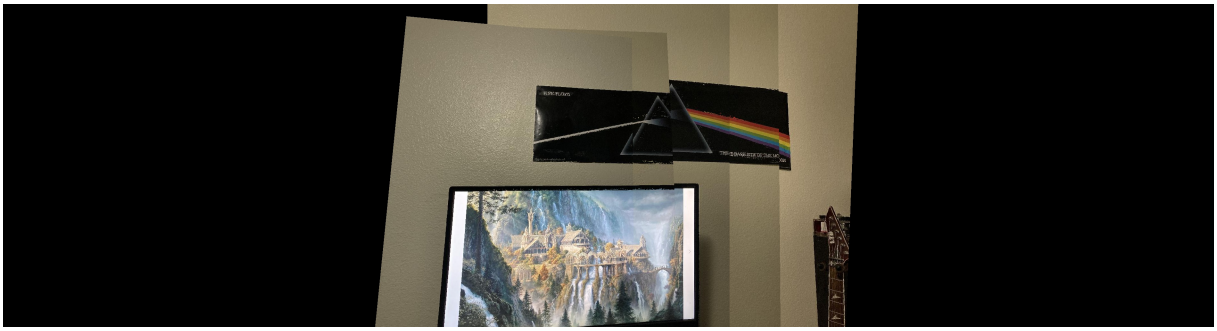

Figure 26: Result panorama image

6 Source Codes

```
1
2  import cv2
3  import numpy as np
4  import math
5  import pdb
6  from scipy.ndimage import map_coordinates
7  from scipy.optimize import least_squares
8
9
10 def compute_distance(F1, F2, mode = 'SSD'):
11   # input: feature from point 1 and 2
12   # same as the one used in HW4
13
14   if mode == 'SSD':
15     diff = F1 - F2
16     distance = np.sum(diff ** 2)
17   elif mode == 'NCC':
18     mu1 = np.mean(F1)
19     mu2 = np.mean(F2)
20     num = np.sum((F1-mu1)*(F2-mu2))
21     denom = np.sqrt(np.sum((F1-mu1)**2) * np.sum((F2-mu2)**2))
22     distance = 1 - (num / denom)
23     # NCC's range is [-1,1], with 1 being the closest match
24     # in order to apply the same "smaller distance = closer match" logic
25     # I use d = 1-NCC, hence smaller = better
26
27   return distance
28
29 def Homography(domain_coord,range_coord):
30   # computing homography using only 4 point pairs
31   # same as the one used in HW2
32   A = np.zeros((8,8))
33   b = range_coord.flatten()
34   for i in range(0,4):
35     A[i*2,0:3] = [domain_coord[i][0],domain_coord[i][1],1]
36     A[i*2,3:6] = [0,0,0]
37     A[i*2,6:8] = [(-1*domain_coord[i][0]*range_coord[i][0]),(-1*domain_coord[i][1]*
       range_coord[i][0])]
38     A[i*2+1,0:3] = [0,0,0]
39     A[i*2+1,3:6] = [domain_coord[i][0],domain_coord[i][1],1]
40     A[i*2+1,6:8] = [(-1*domain_coord[i][0]*range_coord[i][1]),(-1*domain_coord[i
       ][1]*range_coord[i][1])]
41   # print(A)
42   h=np.dot(np.linalg.pinv(A),b)
43   H= np.zeros((3,3))
44   H[0]= h[0:3]
45   H[1]= h[3:6]
```

```
  H[2][0:2]= h[6:8]
  H[2][2]= 1
  # print(H)
  return H


def Homography_LS(domain_coord,range_coord):
  # computing homography for an over-determined system
  # (more than 4 point pairs)
  # using least square estimation
  # same as the one used in HW2
  n = domain_coord.shape[0]
  A = np.zeros((n*2,8))
  b = range_coord.flatten()
  for i in range(0,n):
    A[i*2,0:3] = [domain_coord[i][0],domain_coord[i][1],1]
    A[i*2,3:6] = [0,0,0]
    A[i*2,6:8] = [(-1*domain_coord[i][0]*range_coord[i][0]),(-1*domain_coord[i][1]*
    range_coord[i][0])]
    A[i*2+1,0:3] = [0,0,0]
    A[i*2+1,3:6] = [domain_coord[i][0],domain_coord[i][1],1]
    A[i*2+1,6:8] = [(-1*domain_coord[i][0]*range_coord[i][1]),(-1*domain_coord[i
    ][1]*range_coord[i][1])]
  # print(A)
  tmp = np.linalg.pinv(np.matmul(A.T,A))
  h=np.dot(np.matmul(tmp,A.T),b)
  H= np.zeros((3,3))
  H[0]= h[0:3]
  H[1]= h[3:6]
  H[2][0:2]= h[6:8]
  H[2][2]= 1
  # print(H)
  return H


def sift_pair(img1_raw, img2_raw, name):
  # need to use older version to bypass patent issues
  # opencv-contrib-python == 3.4.2.16
  if len(img1_raw.shape) == 3:
    img1 = cv2.cvtColor(img1_raw, cv2.COLOR_BGR2GRAY)
  else:
    img1 = img1_raw
  if len(img2_raw.shape) == 3:
    img2 = cv2.cvtColor(img2_raw, cv2.COLOR_BGR2GRAY)
  else:
    img2 = img2_raw

  sift = cv2.xfeatures2d.SIFT_create()
  kp1, des1 = sift.detectAndCompute(img1, None)
  kp2, des2 = sift.detectAndCompute(img2, None)

  # OTS brute force matcher by OpenCV
  # bf = cv2.BFMatcher()
  # matches = bf.knnMatch(des1,des2, k=2)
  # good = []
  # for m,n in matches:
  #   if m.distance < 0.75*n.distance:
  #     good.append([m])
  # comb = np.concatenate((img1_raw, img2_raw), axis=1)
  # cv2.drawMatchesKnn(img1_raw,kp1,img2_raw,kp2,good[0:100],comb,flags=2)

  w = img1_raw.shape[1]
  comb = np.concatenate((img1_raw, img2_raw), axis=1)
  domain_coord = np.array([[0,0]])
```

```
  # domain_coord = []
  range_coord = np.array([[0,0]])
  # range_coord = []
  print(len(kp1))
  for i in range(len(kp1)):
    d_tmp = []
    domain_coord_tmp = list(map(int, kp1[i].pt))
    F1 = des1[i]
    # domain_coord.append(list(kp1[i].pt))
    for j in range(len(kp2)):
      F2 = des2[j]
      distance = compute_distance(F1,F2)
      d_tmp.append(distance)
    # pdb.set_trace()
    best_match = list(kp2[np.argsort(d_tmp)[0]].pt)
    best_match = list(map(int, best_match))

    # if i%100 == 0:
    #   print(i)
    # tracking progress

    # enroll the correspondence if the distance is within reasonable range
    if np.min(d_tmp) < 10000:
      domain_coord = np.append(domain_coord,[domain_coord_tmp],axis=0)
      range_coord = np.append(range_coord,[best_match],axis=0)
      # pdb.set_trace()
      pt1 = tuple(domain_coord_tmp)
      pt2 = (best_match[0]+w,best_match[1])

      cv2.circle(comb, pt1, 3, (10,240,10), 1)
      cv2.circle(comb, pt2, 3, (10,10,240), 1)
      cv2.line(comb, pt1, pt2, (10,240,240), 1)
      # visualized

  domain_coord = np.delete(domain_coord, 0, axis=0)
  range_coord = np.delete(range_coord, 0, axis=0)
  # using numpy array rather than list for faster computation
  # to preserve dimensions, i did not initialize them as empty arrays
  # but use a placeholder which would be deleted in the end

  cv2.imwrite(name+'.jpg',comb)
  return domain_coord, range_coord


def ransac_find_H(domain_coord, range_coord, LM):
  # empirical parameters:
  n = 4
  # 4 <= n <= 10
  epsilon = 0.75
  # key parameter, if epsilon = 0.25, N = 12
  # if epsilon = 0.5, N = 70
  # if epsilon = 0.75, N = 1176
  p = 0.99
  delta = 3.0
  N = int(math.log(1 - p) / math.log(1 - (1 - epsilon) ** n))
  # number of trials
  n_total = domain_coord.shape[0]
  # total correspondences
  M = int((1 - epsilon) * n_total)
  # expected number of inliers
  print('N:',N)
  print('M:',M)

  best_inlier_indices = []
```

```
  inlier_pct = 0
  # inlier percentage
  all_indices = list(range(n_total))

  for i in range(N):
    rand_idx = np.random.choice(all_indices,n)
    domain_coord_rand = domain_coord[rand_idx]
    range_coord_rand = range_coord[rand_idx]
    H_tmp = Homography(domain_coord_rand, range_coord_rand)
    inliers_idx = find_inliers(H_tmp, domain_coord, range_coord, delta)
    num_inliers = inliers_idx.shape[0]
    print(num_inliers)
    if num_inliers >= M and num_inliers/n_total > inlier_pct:
    # if num_inliers/n_total > inlier_pct:
      inlier_pct = num_inliers/n_total
      best_inlier_indices = inliers_idx

  H = Homography_LS(domain_coord[best_inlier_indices], range_coord[
    best_inlier_indices])
  if LM == True:
    h = H.flatten()
    res = least_squares(cost_func_H, h, args= (domain_coord[best_inlier_indices],
    range_coord[best_inlier_indices]) , method="lm")
    h_new = res.x
    H = h_new.reshape((3,3))

  return H, best_inlier_indices

def cost_func_H(h, domain_coord, range_coord):
  # h: flattened H for optimization
  H = h.reshape((3,3))

  domain_coord_homo = np.insert(domain_coord,2,1,axis=1)
  range_coord_homo = np.insert(range_coord,2,1,axis=1)
  range_coord_computed = np.matmul(H, domain_coord_homo.T).T
  range_coord_computed = range_coord_computed.T / range_coord_computed.T[2,:]
  range_coord_computed = range_coord_computed.T
  error =  np.abs(range_coord_computed - range_coord_homo)
  return (error**2)


def find_inliers(H, domain_coord, range_coord, delta):

  domain_coord_homo = np.insert(domain_coord,2,1,axis=1)
  range_coord_homo = np.insert(range_coord,2,1,axis=1)
  range_coord_computed = np.matmul(H, domain_coord_homo.T).T
  range_coord_computed = range_coord_computed.T / range_coord_computed.T[2,:]
  range_coord_computed = range_coord_computed.T
  error =  np.abs(range_coord_computed - range_coord_homo)

  d = np.sum(error**2, axis=1)
  # pdb.set_trace()
  ind = np.where(d <= delta ** 2)[0]

  return ind

def visualize_inliers(ind, domain_coord, range_coord, img1_raw, img2_raw, name):
  comb_in = np.concatenate((img1_raw, img2_raw), axis=1)
  comb_out = np.concatenate((img1_raw, img2_raw), axis=1)
  w = img1_raw.shape[1]

  for i in range(domain_coord.shape[0]):
    if i in ind:
      # inlier
```

```
234         pt1 = tuple(domain_coord[i])
235         pt2 = range_coord[i]
236         pt2[0] = pt2[0] + w
237         pt2 = tuple(pt2)
238 
239         cv2.circle(comb_in, pt1, 3, (10,240,10), 1)
240         cv2.circle(comb_in, pt2, 3, (10,10,240), 1)
241         cv2.line(comb_in, pt1, pt2, (10,240,10), 1)
242       else:
243         # outlier
244         pt1 = tuple(domain_coord[i])
245         pt2 = range_coord[i]
246         pt2[0] = pt2[0] + w
247         pt2 = tuple(pt2)
248 
249         cv2.circle(comb_out, pt1, 3, (10,240,10), 1)
250         cv2.circle(comb_out, pt2, 3, (10,10,240), 1)
251         cv2.line(comb_out, pt1, pt2, (10,10,240), 1)
252 
253     cv2.imwrite(name+'_inlier.jpg',comb_in)
254     cv2.imwrite(name+'_outlier.jpg',comb_out)
255 
256 def pixel_mapping(comb, img, H):
257     h = img.shape[0]
258     w = img.shape[1]
259 
260     h_total = comb.shape[0]
261     w_total = comb.shape[1]
262 
263     H = np.linalg.inv(H)
264 
265     X,Y = np.meshgrid(np.arange(0,w_total,1), np.arange(0,h_total,1))
266     pts= np.vstack((X.ravel(),Y.ravel())).T
267     # pts = np.flip(pts,axis=1)
268     pts = np.hstack((pts[:, 0:2], pts[:, 0:1]*0+1))
269 
270     Hpts = np.dot(H, pts.T)
271     Hpts = Hpts / Hpts[2,:]
272     # Hpts = Hpts * (1.0 / np.tile( Hpts[:, 2], (3, 1)) ).T
273     Hpts = Hpts.T[:, 0:2].astype('int')
274     # Hpts = np.flip(Hpts,axis=1)
275     # pdb.set_trace()
276 
277     # b_channel = img[:,:,0]
278     # g_channel = img[:,:,1]
279     # r_channel = img[:,:,2]
280     # # pdb.set_trace()
281     # b_new = map_coordinates(b_channel,Hpts.T)
282     # g_new = map_coordinates(g_channel,Hpts.T)
283     # r_new = map_coordinates(r_channel,Hpts.T)
284 
285     # comb[:,:,0] = np.reshape(b_new, (h_total,w_total))
286     # comb[:,:,1] = np.reshape(g_new, (h_total,w_total))
287     # comb[:,:,2] = np.reshape(r_new, (h_total,w_total))
288     # hw3 stuff, not suitable
289 
290     # only replace part of the (combined) image
291     valid_pts, valid_Hpts = find_valid_pts(pts, Hpts, w-1, h-1)
292     for i in range(valid_pts.shape[0]):
293       # pdb.set_trace()
294       if (comb[valid_pts[i,1],valid_pts[i,0]] != 0).all() == False:
295         comb[valid_pts[i,1],valid_pts[i,0]] = img[valid_Hpts[i,1],valid_Hpts[i,0]]
296 
297 
```

```
  return comb


def find_valid_pts(pts, Hpts, w, h):
  xmin = Hpts[:,0] >= 0
  Hpts = Hpts[xmin,:]
  pts = pts[xmin,:]

  xmax = Hpts[:,0] <= w
  Hpts = Hpts[xmax,:]
  pts = pts[xmax,:]

  ymin = Hpts[:,1] >= 0
  Hpts = Hpts[ymin,:]
  pts = pts[ymin,:]

  ymax = Hpts[:,1] <= h
  Hpts = Hpts[ymax,:]
  pts = pts[ymax,:]

  return pts, Hpts


def panorama(max_num = 5, LM = False):
  # input: max number of pictures as input
  mid_img = max_num // 2

  H_all = []
  for i in range(max_num-1):
    path1 = 'input/' + str(i+1) + '.jpg'
    path2 = 'input/' + str(i+2) + '.jpg'
    img1 = cv2.imread(path1)
    img2 = cv2.imread(path2)
    img1_cp = img1.copy()
    img2_cp = img2.copy()

    domain_coord, range_coord = sift_pair(img1, img2, str(i+1)+'_and_'+str(i+2))
    # compute SIFT features and construct correspondence

    # np.save(str(i+1)+'_and_'+str(i+2)+'_domain',domain_coord)
    # np.save(str(i+1)+'_and_'+str(i+2)+'_range',range_coord)

    # # saving and loading point correspondences
    # # so that I don't need to spend time repeating the process while debugging

    # domain_coord = np.load(str(i+1)+'_and_'+str(i+2)+'_domain.npy')
    # range_coord = np.load(str(i+1)+'_and_'+str(i+2)+'_range.npy')
    # pdb.set_trace()
    H, inliers = ransac_find_H(domain_coord, range_coord, LM)
    np.save(str(i+1)+'_and_'+str(i+2)+'_inlier',inliers)

    visualize_inliers(inliers, domain_coord, range_coord, img1_cp, img2_cp, str(i+1)
    +'_and_'+str(i+2))

    H_all.append(H)
  np.save('H_all',H_all)

  # H_all = list(np.load('H_all.npy'))

  H_to_mid = np.eye(3)
  for i in range(mid_img, len(H_all)):
    H_to_mid = np.matmul(H_to_mid, np.linalg.inv(H_all[i]))
    H_all[i] = H_to_mid

```

```
361   H_to_mid = np.eye(3)
362   for i in range(mid_img - 1, -1, -1):
363     H_to_mid = np.matmul(H_to_mid, H_all[i])
364     H_all[i] = H_to_mid
365   # set H to morph to a mid image
366
367   H_all.insert(mid_img, np.eye(3))
368   tx = 0
369   for i in range(mid_img):
370     path = 'input/' + str(i+1) + '.jpg'
371     img = cv2.imread(path)
372     tx = tx + img.shape[1]
373   H_t = np.array([[1, 0, tx], [0, 1, 0], [0, 0, 1]], dtype=float)
374   # pdb.set_trace()
375   # translational transformation
376   height = 0
377   width = 0
378   for i in range(max_num):
379     path = 'input/' + str(i+1) + '.jpg'
380     img = cv2.imread(path)
381     height = max(height, img.shape[0])
382     width = width + img.shape[1]
383
384   comb = np.zeros((height, width, 3), np.uint8)
385
386   for i in range(max_num):
387     path = 'input/' + str(i+1) + '.jpg'
388     img = cv2.imread(path)
389     H = np.matmul(H_t, H_all[i])
390     comb = pixel_mapping(comb, img, H)
391   cv2.imwrite('panorama.jpg', comb)
392
393 if __name__ == '__main__':
394   panorama(LM=True)
```