# Homework 5

## Theory Questions

- First, we pick from 4 to 10 correspondences, then using those correspondences we get a homography matrix that maps one set of keypoints to the other. We transform all the available data points using that homography and check whether they are within a $\delta$ geometric distance (euclidean distance) from the available keypoints. If they are, they are considered inliers, if they are not then they are outliers.
- Levenberg-Marquadt algorithm uses a damping term in the approximation to the Hessian matrix ($\mathbf{J}^\top\mathbf{J} + \mu\mathbf{I}$), which allows the algorithm to behave like Gradient Descent when far away from the minima, which is good since that is when gradient descent works best. And when we are close to the minima, it will behave more like Gauss-Newton, since that is where GN method has the upper hand since GD is slow when near the solution.

## Task 1

First of all, I set one ground rule: no unnecessary for loops, no nested for loops. This is due to the fact that the time that HW2 and HW3 took to run was too long. Taking this into account, we again use the binary mask that was used in HW2 and use it to paint the panorama. More on this in the panorama section. For the keypoint detection and matching we used Superpoint and Superglue.

### RANSAC algorithm

For RANSAC, we used a scale of $\sigma = 2$, this got us the best results since using any lower values made it so we didn't find many inliers. The ratio of outliers, $\epsilon$ we are using is 0.1 since it is visually clear that most of the matches that Superpoint+Superglue find are accurate, however we still need to check with RANSAC to see fi they are accurate. The probability $p$ we are using is 0.99. The number of points used is 10. We calculate the number of trials (N) and the number of inliers (M) needed to stop the algorithm using these parameters. We used $\delta = 3\sigma$ to check whether a datapoint is an inlier or not. The process we followed is to pick 10 random datapoints and get a homography using Linear Least Squares and then transform keypoints 1 using it. Then we check whether the geometric distance of the transformed keypoints and the original keypoints is less than $\delta = 3\sigma$, if it is then it is an inlier, if not it is an outlier. Repeat until we get an inlier set with M or more elements.

### Least Squares Homography

For Linear Least Squares homography, we solve an inhomogeneous linear system:

$$\mathbf{Ah} = \mathbf{b}$$

Where in this case, $\mathbf{A}$ is the system built using the datapoints, $\mathbf{h}$ is our homography and $\mathbf{b}$ are our other datapoints. We solve it by solving a linear system similar to the ones from

HW2 and HW3. However, the main difference is that we now use the pseudo-inverse method since this is an overdetermined system (we have 10 datapoints).

$$(\mathbf{A}^\top \mathbf{A})^{-1}\mathbf{A}^\top \mathbf{b} = \mathbf{h}$$

Solving this gives us 8 values of the homography matrix, where the one missing is $h_{3,3}$, which is 1.

### Levenberg-Marquadt Algorithm

For our implementation of the Levenberg-Marquadt algorithm we first calculate the Jacobian matrix using an initial H matrix calculated using the full set of inliers from RANSAC. We then get our initial value of $\mu$ from the maximum value along the diagonal of $(\mathbf{J}^\top \mathbf{J})$ times $\tau = 1.5$. We use this value since it produced the best results. After we start our loop, with 100 iterations maximum. We calculate the error function, which is given by $\mathbf{X} - \mathbf{f}(\mathbf{x})$), $\epsilon_k$ and calculate the new $\delta$ to update our $H_k$ matrix. We get the new matrix $\mathbf{H}_{k+1} = \mathbf{H}_k + \delta_k$. We then calculate the error for our $\mathbf{H}_{k+1}$ matrix. We square both of these errors and using everything we get $\rho_k$. We now check that the cost function, which is the squared of the error function, has decreased from our previous iteration. If it has not, we double the value of $\mu$ for the next iteration, if it has decreased then we calculate our new value of $\mu$ using $\rho$.

### Painting the panorama

As mentioned before, HW2 and HW3 took too long to run, so to solve that we use binary masks. This is so we can pick the exact spots where we need to paint the image in the final panorama, as opposed to having to loop through all of it. These binary masks can be seen in Fig. 2. This made it easy for us to select only the spots we want to paint, reducing the amount of iterations we spend painting the panorama, especially since there are large amount of empty space. We create this panorama by projecting every image into the middle one, image 3. We do this by getting the matches from the following pairs of images: image 1 and 2, 2 and 3, 4 and 3 and 5 and 4. An assumption we make for the code is that there will always only be 5 images. We run RANSAC on the correspondences, procedeed by finding the linear least square homography on the full inlier set per image. We then refine these homographies with LM algorithm. After finding the refined homography matrices, we make use of the fact that we can project images 1 into 3 by multiplying homographies like this to project images 1 into 3 and 5 into 3.

$$\mathbf{x}'_{1\to 3} = \mathbf{H}_{2\to 3}\mathbf{H}_{1\to 2}\mathbf{x}_1$$

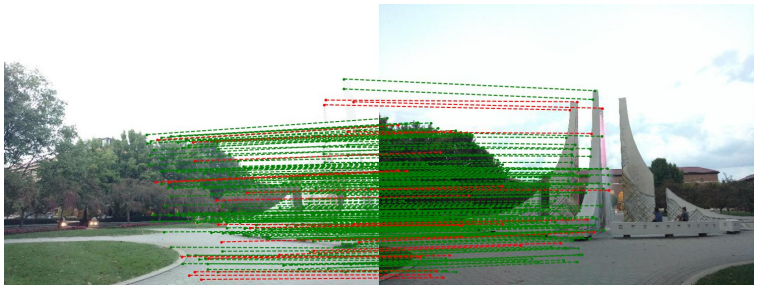
After this we get a binary mask with corners corresponding to the transformed corners per image. Finally, we use only the places where there is a binary mask to paint the images into the final panorama.

## Results

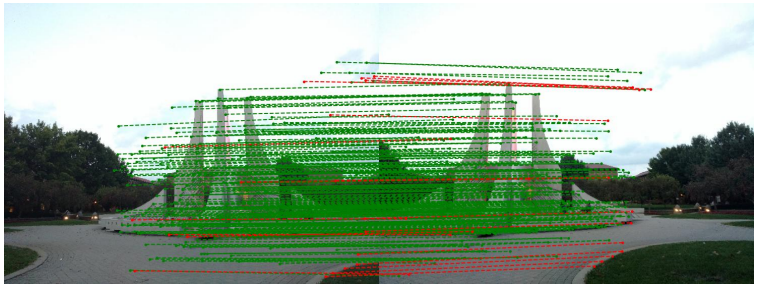
### Task 1: RANSAC

We consider $n = 10$ for number of points considered per iteration. As we can see, Superpoint+Superglue did generate a large amount of inliers, but there are still some outliers, so

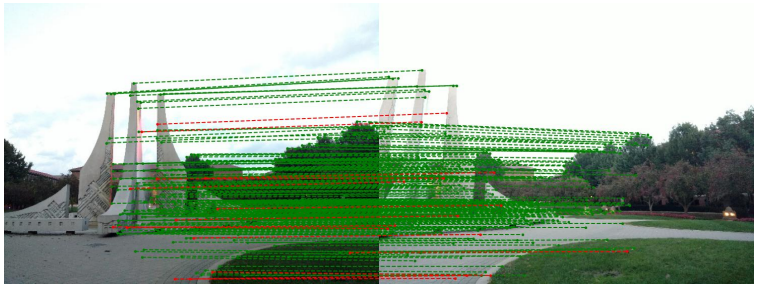
RANSAC was definitely needed, as seen in Fig. 1. These parameters might also, on very rare occasions, not be able to fill the inlier list with enough good correspondences, but this was not usually the case. Another run of the program will more often than not fix that. If trying to be safe, increase $\epsilon$ to 0.2. We also tried to paint the panorama with the homography obtained from the full inlier set with least squares, and it can be seen in Fig. 3. It looks good, even when zooming in.



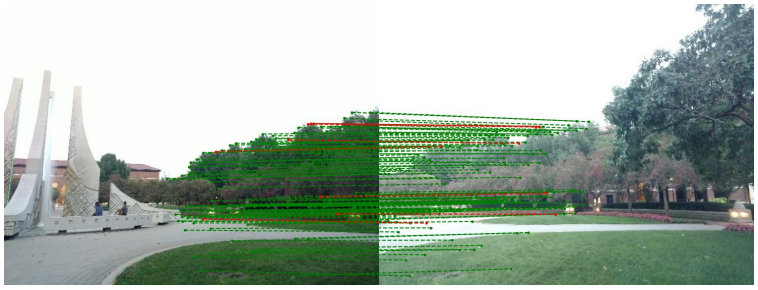
(a) Inliers and outliers in images 1 and 2.



(b) Inliers and outliers in images 2 and 3.

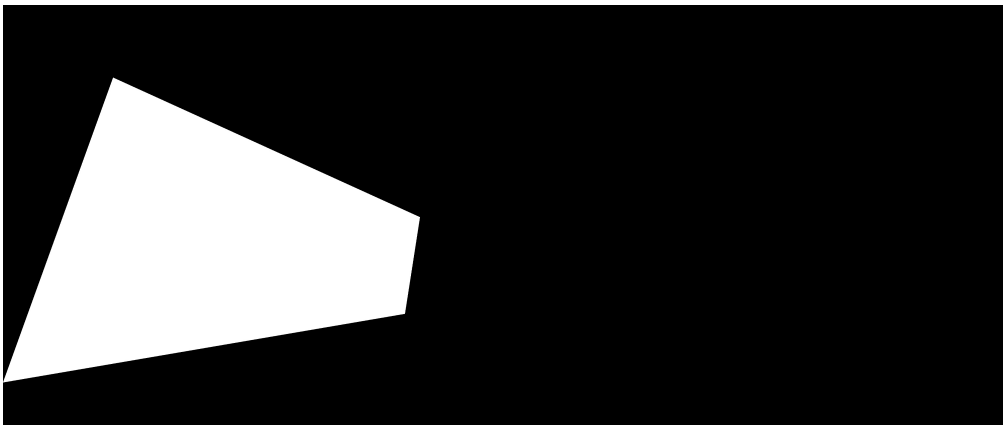


(c) Inliers and outliers in images 3 and 4.

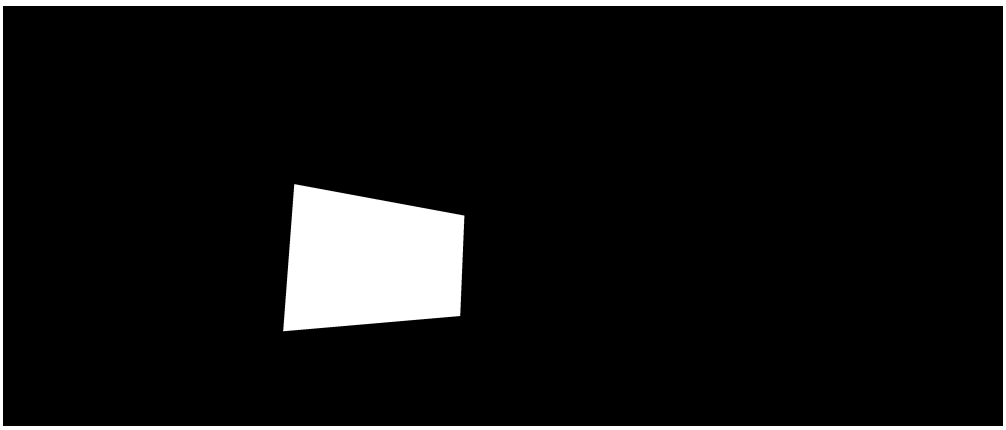


(d) Inliers and outliers in images 4 and 5.

Figure 1: Inlier (green) and outlier (red) pairs.



(a) Binary mask for image 1 in the final panorama.



(b) Binary mask for image 2 in the final panorama.

(c) Binary mask for image 3 in the final panorama.

Figure 2: Some of the binary masks used to paint the final panorama.

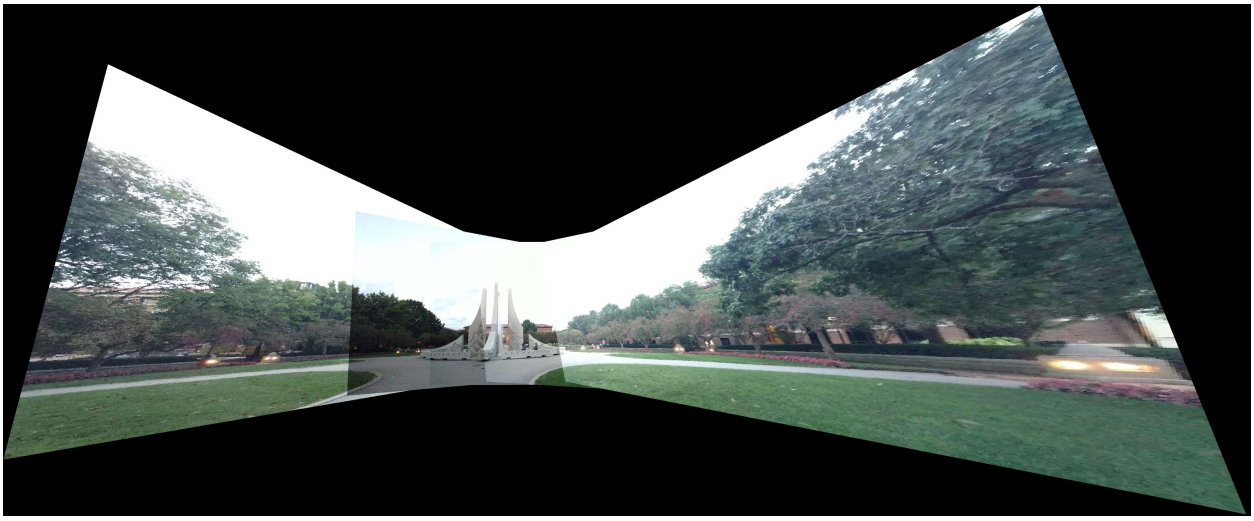


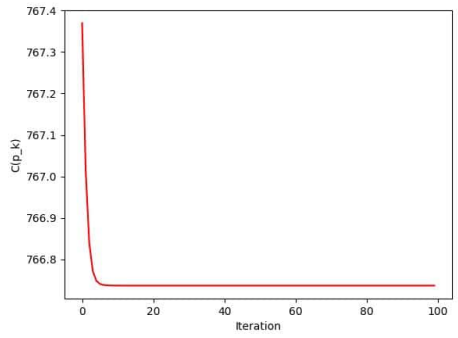
Figure 3: Panorama from the fountain images put together using the homography from Least Squares after RANSAC.

**Task 1: Levenberg-Marquadt**

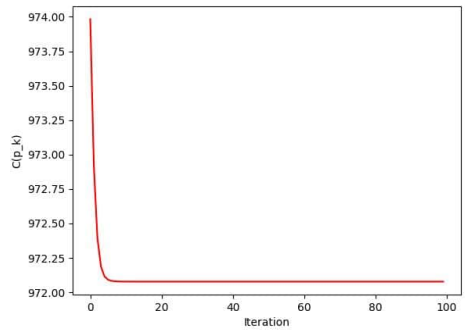
We got the best results from using $\tau = 1.5$, as going lower resulted in the error increasing, and going way higher resulted in less decrease of the error. As seen in Table 1 and Fig. 4, the error did go down for us, but it was very minor for all images. Notice how in Fig. 4c, the line goes up a few times but then down again, this is where the quality check comes into play, it is the reason why it then went down again. I also implemented the scipy version and this one was able to optimize it significantly better. Looking at the final results, however, there are no significant changes that can be easily observed. Compared to the Least Squares homography from Fig. 3, our implementation of LM in Fig. 5 has two main differences. They are in the top half, near the left and right end, these are however very minor. And compared to the scipy implementation in Fig. 6, they are similar in the right side but it is more like the least squares version in the left side. One possible explanation that scipy does it better could be due to the fact that the homography is already very close to the solution and their implementation handles GN in a better way than mine.

| Images | Cost (Least Squares) | Cost (Scipy LM) | Cost (own LM) |
|---|---|---|---|
| 1 & 2 | 766.7365897906659 | 729.9463489563528 | 766.7365897906657 |
| 2 & 3 | 972.0780293746332 | 962.2326251009807 | 972.0780293746319 |
| 3 & 4 | 908.956182042687 | 904.5377961592285 | 908.7804575337003 |
| 4 & 5 | 586.422115479814 | 580.1326831138404 | 586.422115479814 |

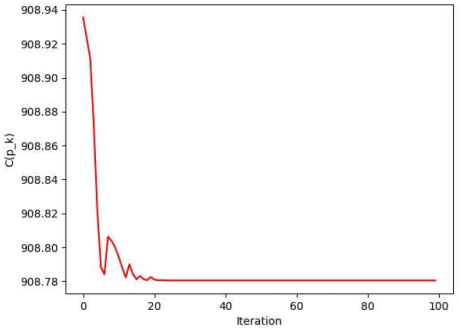
Table 1: End cost function comparison between Least Squares homography, and refined homography with Scipy LM and own implementation of LM.



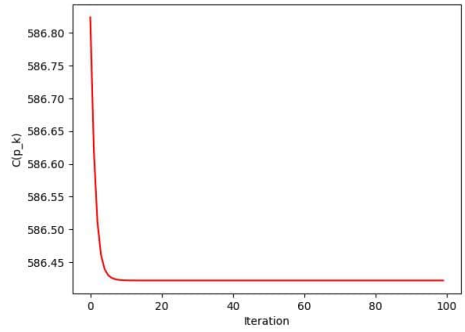

(a) Cost function for images 1 and 2.




(b) Cost function for images 2 and 3.




(c) Cost function for images 3 and 4.




(d) Cost function for images 4 and 5.

Figure 4: Cost function per iteration for our LM for different images.

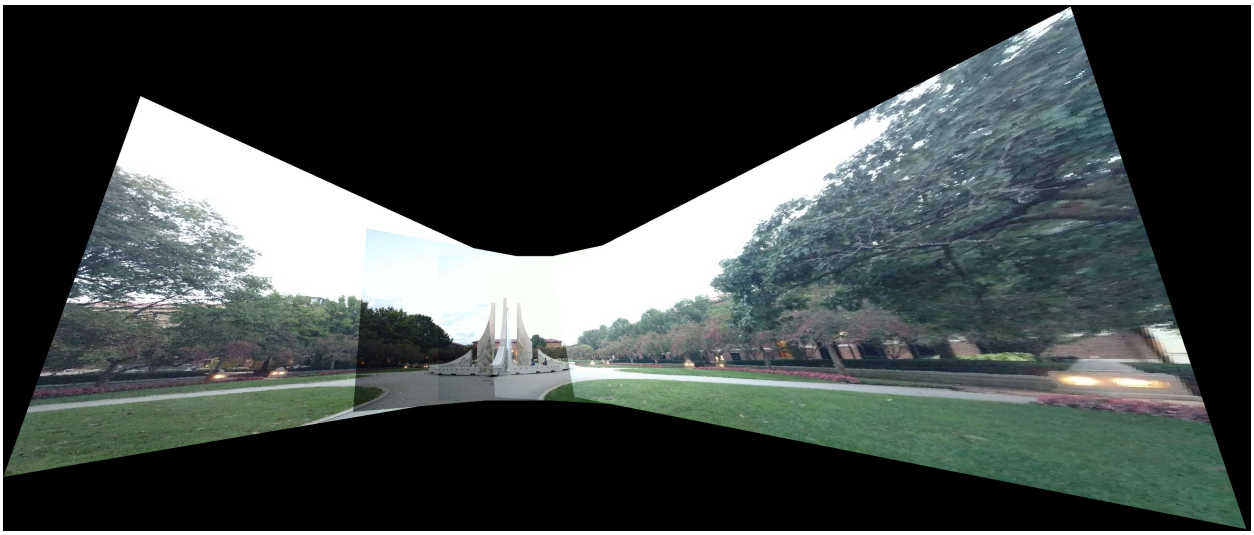


Figure 5: Panorama from the fountain images put together using own implementation of LM algorithm.

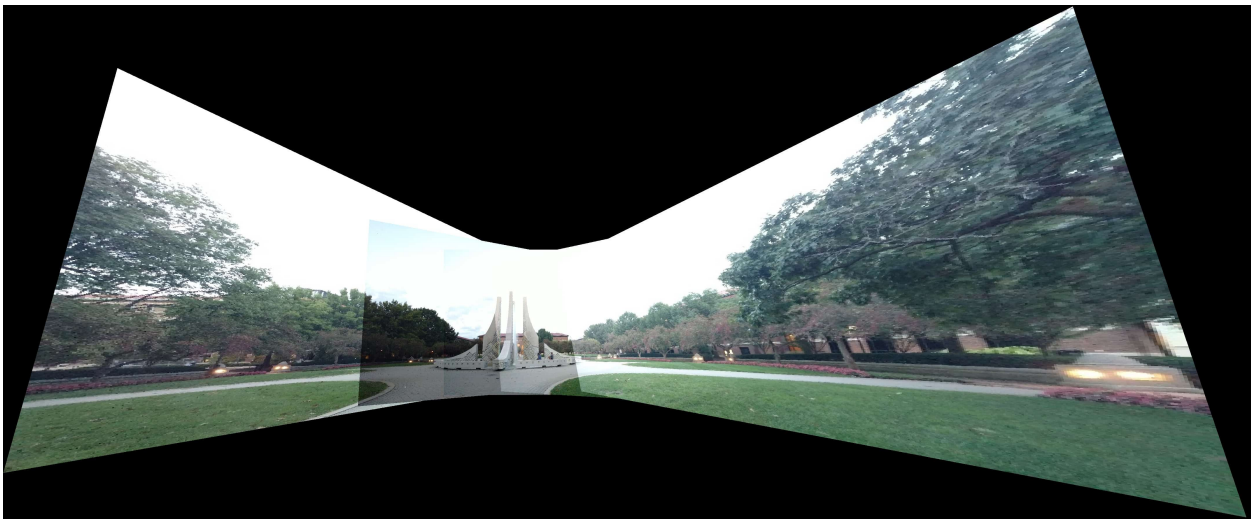


Figure 6: Panorama from the fountain images put together using Scipy LM.

**Task 2**

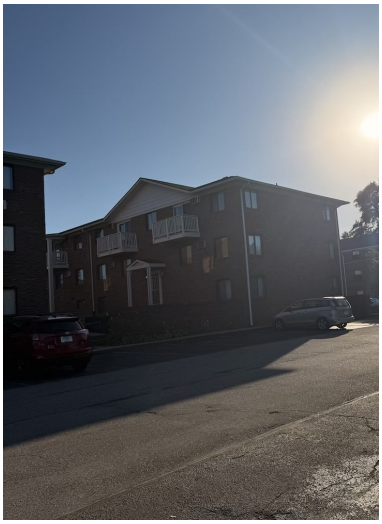
Notice that image 5, in Fig. 7e, is a bit cropped, this is due to the fact that the final panorama was very warped and large due to the other part of image 5, so I just cropped it out.

(a) Image 1

(b) Image 2.

(c) Image 3.

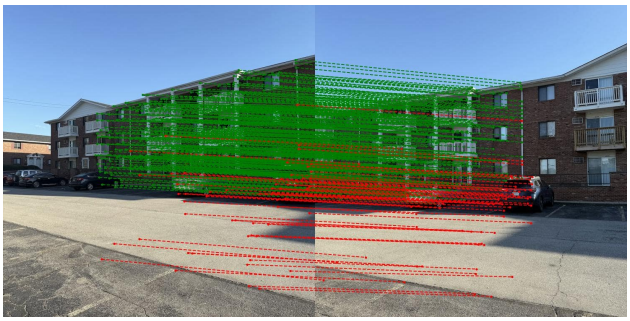
(d) Image 4.



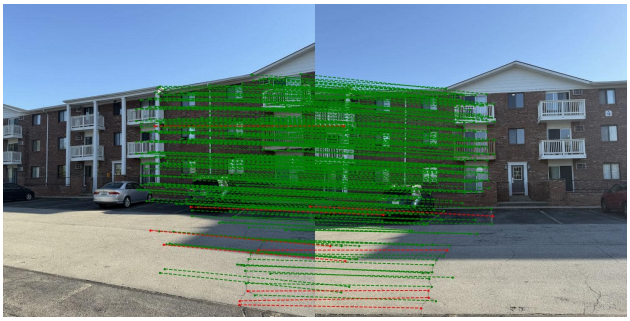
(e) Image 5.

Figure 7: Input images for task 2.

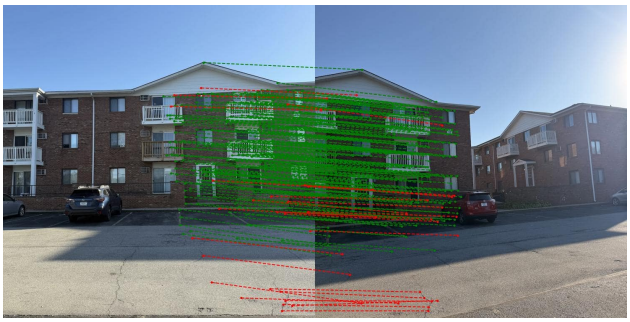
For this part, we considered a different $\sigma$ value, it is 4 in this case, and $\epsilon$ is the now 0.3 for RANSAC. If we did not increase $\sigma$ we were not getting the correct amount of outliers on some of the pairs of images, and changing $\epsilon$ helped with that.



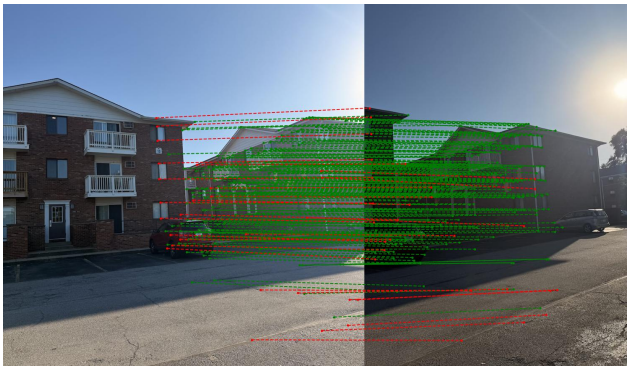
(a) Inliers and outliers in images 1 and 2.



(b) Inliers and outliers in images 2 and 3.



(c) Inliers and outliers in images 3 and 4.



(d) Inliers and outliers in images 4 and 5.

Figure 8: Inlier (green) and outlier (red) pairs.

Overall, the final panorama looks good even with the Least Squares solution. One problem that can be seen is near the middle of the panorama, where one of the railings has been

stitched wrong, there is a sharp discontinuity in there. This happens in all 3, see Fig. 9, 10, 11. Aside from that, we can see that there are some improvements with our LM algorithm, as it looks more like the Scipy LM implementation than the least squares panorama. For our LM algorithm, we now use a different value of $\tau$, it is now 10. This worked for us, since the geometric error did go down, small amounts, similar to task 1.

Figure 9: Panorama from the building images put together using the homography from least squares.

Figure 10: Panorama from the building images put together using the homography from scipy LM.

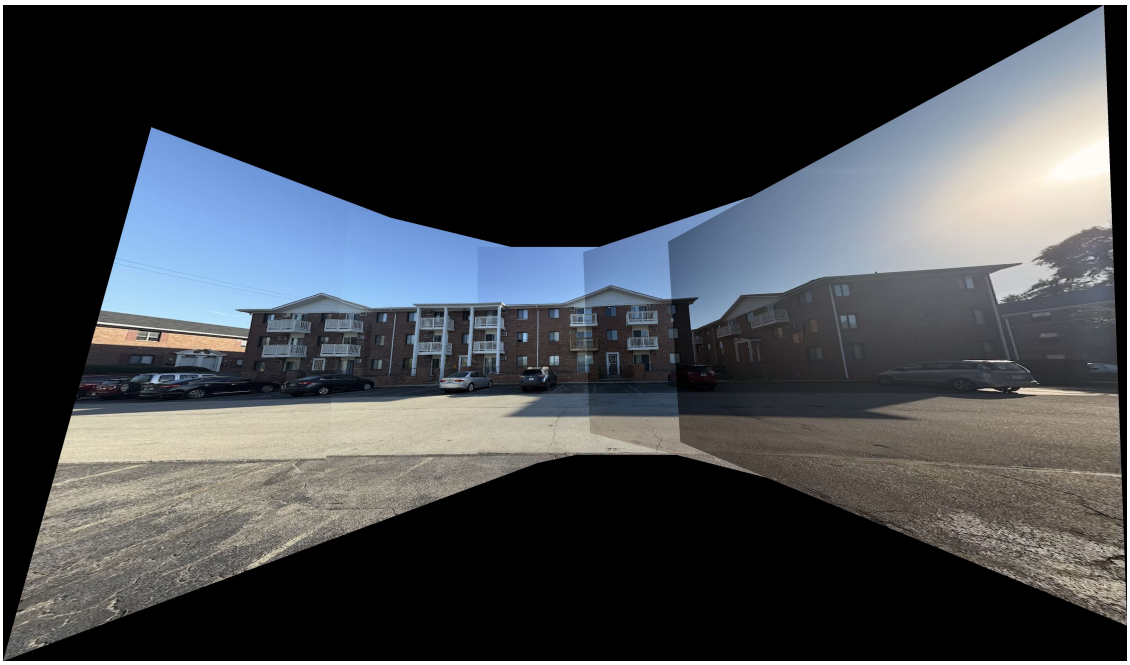


Figure 11: Panorama from the building images put together using the homography from own implementation of LM.

## Source code

```
# we use superpoint so this needs to be inside the superguepretrainednetwork
import cv2
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from superglue_ece661 import *
from models.matching import Matching
from models.utils import process_resize, read_image
from scipy.optimize import least_squares

def point_to_point_system(domain, range_, num_points):
    # we changed this function to accept an arbitrary number of points,
    # this is because most of the systems we will solve have more than 4 data
    mat1 = np.empty((0,8),dtype=float)
    mat2 = np.empty((0,0),dtype=float)
    for i in range(num_points):
        x = domain[i,0]
        x_prime = range_[i,0]
        y = domain[i,1]
        y_prime = range_[i,1]

        mat1 = np.append(mat1,np.array([[x, y, 1, 0, 0, 0, -x*x_prime, -y*x_p
        mat1 = np.append(mat1,np.array([[0, 0, 0, x, y, 1, -x*y_prime, -y*y_p
        mat2 = np.append(mat2,x_prime)
        mat2 = np.append(mat2,y_prime)

    return mat1,mat2.reshape(num_points*2,1)

def get_H_matrix(domain,range_,n_points):
    # we no longer use this, since all the systems we will solve in this home
    # overdetermined, in both ransac and least squares
    mat1, mat2 = point_to_point_system(domain, range_,n_points)
    # we use this function to solve the equation described in the logic,
    # I replaced np.dot with @ as that is what they suggest in the numpy webs
    sol = np.linalg.inv(mat1) @ mat2
    # we append the 1 since this will only have 8 values, the 1 is missing
    sol = np.append(sol,np.array([[1]]),axis=0)
    return sol.reshape((3,3)).astype(float)

def get_H_matrix_overdetermined(domain,range_,n_points):
    # this is the one we use, in both ransac and least squares, it takes in n
    mat1, mat2 = point_to_point_system(domain, range_,n_points)
    # we use this function to solve the equation described in the logic,
    # I replaced np.dot with @ as that is what they suggest in the numpy webs
```

```
    # in this case we use the pseudo inverse (ATA)^-1 AT
    sol = np.linalg.inv(mat1.T @ mat1) @ mat1.T @ mat2
    # we append the 1 since this will only have 8 values, the 1 is missing
    sol = np.append(sol,np.array([[1]]),axis=0)
    return sol.reshape((3,3)).astype(float)

def add_ones(array):
    # this function takes in an array and adds ones
    # this is mainly for applying homographies
    ones_np = np.ones(array.shape[:-1])[...,None]
    # need to expand the dimensions by 1 to be able to concatenate it
    return np.append(array,ones_np,axis=-1)

def apply_homography(positions, H):
    # this gets the homography transformation for all the coordinates in the
    # we do it in a way that exploits broadcasting, so we don't need to use f
    temp_pos = add_ones(positions)
    new_pos = (H @ temp_pos.T).astype(float)
    new_pos /= new_pos[2,:]
    return new_pos[:2,:].T

def check_inliers(kp, kp_compare, tr_kp, delta):
    # we compare using the l2 norm, the distance between the transformed keyp
    dists = np.linalg.norm(tr_kp - kp_compare, axis=1)
    inliers_1 = []
    inliers_2 = []
    outliers_1 = []
    outliers_2 = []
    # loop through it and check if the distance is less than delta,
    # append both image 1 and image 2 keypoints to inliers if it is,
    # append to outliers if it isn't
    inliers_idx = np.argwhere(dists < delta)
    outliers_idx = np.argwhere(dists >= delta)
    # again, we don't need to use for loops here
    inliers_1 = np.take(kp, inliers_idx,axis=0)[:,0,:]
    inliers_2 = np.take(kp_compare, inliers_idx,axis=0)[:,0,:]
    outliers_1 = np.take(kp, outliers_idx,axis=0)[:,0,:]
    outliers_2 = np.take(kp_compare, outliers_idx,axis=0)[:,0,:]
    return inliers_1, inliers_2, outliers_1, outliers_2

def ransac(kp_1, kp_2 , sigma=2, epsilon=0.1, n_points=20):
    # we run ransac here, we take into consideration the scale (sigma), epsil
    # from these values we get the other parameters like delta, N and M
    # our epsilon is 0.1 since superpoint doesn't seem to generate many outli
    num_keypoints = len(kp_1)
```

```
    p = 0.99
    delta = sigma * 3
    N = int(np.log(1-p)/np.log(1- (1 - epsilon)**n_points))
    M = int((1 - epsilon)*num_keypoints)
    random_choose = np.random.randint(0,num_keypoints,(N, n_points))

    for rand in random_choose:
        domain = np.take(kp_1,rand,axis=0)
        range_ = np.take(kp_2,rand,axis=0)
        H = get_H_matrix_overdetermined(domain,range_,n_points)

        h_kp_1 = apply_homography(kp_1,H)

        inliers_1, inliers_2, outliers_1, outliers_2 = check_inliers(kp_1, kp
        if len(inliers_1) >= M:
            return np.array(inliers_1), np.array(inliers_2), np.array(outlier

    return None, None, None, None

def get_new_corners(img,H):
    # we just get the new corners, all mapped to image 3
    # for image 1 and 2 there should be negatives, but that is exactly what w
    # so we can get the actual size of the new image
    h, w, _ = img.shape
    corners = np.array([[0,0],[w,0],[w,h],[0,h]])
    new_corners = apply_homography(corners,H)
    return new_corners

def paint_image(panorama,image, old_positions, new_positions,x_min,y_min):
    # this is to paint an image into the panorama, knowing the coordinates of
    # yes, its old positions even though technically, they are calculated aft
    # old image
    #print(new_positions)
    for idx, n_pos in enumerate(new_positions):
        if 0 <= old_positions[idx,0] < image.shape[1] and 0 <= old_positions[
            panorama[new_positions[idx,1] - y_min,new_positions[idx,0] - x_mi
    return panorama

def get_binary_mask(panorama, corners, x_min, y_min):
    # we want these masks so we only iterate through the parts with the mask,
    # otherwise we need to iterate through the whole final panorama, which we
    canvas = np.zeros((panorama.shape[0],panorama.shape[1])).astype(np.uint8)
    new_corners = corners
    new_corners[:,0] -= x_min
    new_corners[:,1] -= y_min
```

```
    new_corners = new_corners.astype(np.int32)
    canvas = cv2.fillPoly(canvas, pts=[new_corners], color=255)
    return canvas

def error_f(H, kp0,kp1):
    # get the error X - f
    kp_k = apply_homography(kp0, H.reshape(3,3))
    # we use ravel since we want it to be a flat array
    return (kp1 - kp_k).ravel()

def jacobian(H, kp0):
    # our jacobian matrix
    J = np.zeros((2*len(kp0),9))
    for idx in range(len(kp0)):
        num_1 = H[0,0]*kp0[idx,0] + H[0,1]*kp0[idx,1] + H[0,2]
        num_2 = H[1,0]*kp0[idx,0] + H[1,1]*kp0[idx,1] + H[1,2]
        denom = H[2,0]*kp0[idx,0] + H[2,1]*kp0[idx,1] + H[2,2]

        J[idx,0] = kp0[idx,0]/denom
        J[idx,1] = kp0[idx,1]/denom
        J[idx,2] = 1/denom
        J[idx,6] = -kp0[idx,0]*num_1/denom
        J[idx,7] = -kp0[idx,1]*num_1/denom
        J[idx,8] = -num_1/denom
        # now the next row
        J[idx+1,3] = kp0[idx,0]/denom
        J[idx+1,4] = kp0[idx,1]/denom
        J[idx+1,5] = 1/denom
        J[idx+1,6] = -kp0[idx,0]*num_2/denom
        J[idx+1,7] = -kp0[idx,1]*num_2/denom
        J[idx+1,8] = -num_2/denom
    return J

def lm_algorithm(kp_0, kp_1, H_ini, tau=1.5,max_iters=100):
   # our implementation of the LM algorithm
   J_k = jacobian(H_ini, kp_0)
   # initial mu value
   mu_k = tau * np.max(np.diag(J_k.T@J_k))
   # make H be flat instead of a matrix
   H_k = H_ini.ravel()
   C_arr = []
   for iter in range(max_iters):
      # jacobian calculation
      J_f = jacobian(H_k.reshape(3,3), kp_0)
      # error calculation
```

```
        eps_pk = error_f(H_k, kp_0, kp_1)
        # cost calculation
        C_pk = np.linalg.norm(eps_pk)**2
        # get delta_p
        delta_p = np.linalg.inv(J_f.T @ J_f + mu_k*np.eye(9)) @ J_f.T @ eps_pk
        # get new H
        H_new = H_k + delta_p
        # error with new H
        eps_pk1 = error_f(H_new, kp_0, kp_1)
        # new H cost
        C_pk1 = np.linalg.norm(eps_pk1)**2
        C_arr.append([iter,C_pk1])
        # get rho_{k+1}
        rho_n = C_pk - C_pk1
        rho_d = delta_p.T @ (mu_k*np.eye(9) @ delta_p) + delta_p.T @ (J_f.T @ e

        rho_k1 = rho_n/rho_d
        # now the quality check
        if rho_n > 0:
            # if rho_n is positive, that means that the new cost is lower than t
            # which is good, that is what we are looking for
            H_k = H_new
            # pick our new mu_k
            mu_k = mu_k * max(1./3, 1 - (2*rho_k1 - 1)**3)
        else:
            # now, in the case that rho_n is negative, that means our cost has i
            # so we multiply mu_k by 2 and try again
            mu_k = 2*mu_k
            H_k = H_k

    return H_new.reshape(3,3), np.array(C_arr)

def plot_function(costs,figname):
    # just to plot the cost functions
    plt.clf()
    plt.figure()
    plt.plot(costs[:,0],costs[:,1],color='r')
    plt.xlabel("Iteration")
    plt.ylabel("C(p_k)")
    plt.savefig(figname)
    plt.close()

def get_homography(detector, img0, img1, sigma=2,epsilon=0.1,n_points=10, mod
    # this function will get the keypoints for 2 images, run runsac outlier r
    # and then get the homography with least squares, which after it will run
```

```
    mkpts0, mkpts1, _ = detector.match(img0, img1)
    inliers_1, inliers_2, outliers_1, outliers_2 = ransac(mkpts0, mkpts1, sig
    # for least squares we can reuse this function that we used in ransac
    H_leastsquares = get_H_matrix_overdetermined(inliers_1, inliers_2, len(in
    # we also get the lev-mar optimization lm_algorithm(kp_0, kp_1, H_ini, ta
    if mode == "LS": #LS for least squares / LM for Leverberg Marquadt
        #print("Cost for H_leastsquares: ",np.linalg.norm(error_f(H_leastsqua
        return inliers_1, inliers_2, outliers_1, outliers_2, H_leastsquares,
    else:
        # uncomment the following line for own implementation of LM
        H_LM, cost = lm_algorithm(inliers_1, inliers_2, H_leastsquares,tau=ta
        # the following line is the scipy version of LM
        H_LM_scipy = least_squares(error_f, H_leastsquares.ravel(), method='l
        print("Cost for H_leastsquares: ", np.linalg.norm(error_f(H_leastsqua
        print("Cost for H_LM: ",np.linalg.norm(error_f(H_LM,inliers_1,inliers
        print("Cost for H_LM scipy: ",np.linalg.norm(error_f(H_LM_scipy,inlie
        print("Change in cost: ", np.linalg.norm(error_f(H_leastsquares,inlie
        # when using own LM implementation, change the final return value to
        # when using scipy LM change it to None
        return inliers_1, inliers_2, outliers_1, outliers_2, H_LM, cost

def plot_inliers_outliers(image1, image2, inliers_1, inliers_2, outliers_1, o
    # this is a modified function from superglue_ece661.py, plot_keypoints in
    tot_img = np.hstack((image1, image2))
    tot_img = cv2.cvtColor(tot_img, cv2.COLOR_BGR2RGB)
    _, w0, _ = image1.shape
    ms = 0.5
    lw = 0.5
    plt.figure()
    #plt.clf()
    plt.imshow(tot_img);
    # plot inliers in green (match and keypoint)
    for ikp0 in range(len(inliers_1)):
        plt.plot(inliers_1[ikp0,0], inliers_1[ikp0,1], 'g.', markersize=ms)
        plt.plot(inliers_2[ikp0,0]+w0, inliers_2[ikp0,1], 'g.', markersize=ms
        plt.plot((inliers_1[ikp0,0], inliers_2[ikp0,0]+w0), (inliers_1[ikp0,1
    # plot outliers in red
    for ikp0 in range(len(outliers_1)):
        plt.plot(outliers_1[ikp0,0], outliers_1[ikp0,1], 'r.', markersize=ms)
        plt.plot(outliers_2[ikp0,0]+w0, outliers_2[ikp0,1], 'r.', markersize=
        plt.plot((outliers_1[ikp0,0], outliers_2[ikp0,0]+w0), (outliers_1[ikp
    plt.axis('off');
    plt.savefig(figname, bbox_inches='tight', pad_inches=0, dpi=300)
    plt.close()
    return
```

```
def create_panorama(left ,middle, right, detector, sigma=2,epsilon=0.1,n_poin
    # for example, we have 5 images numbered 1 through 5
    # the panorama we want looks like this: 1 2 3 4 5
    # so we know 1 and 2 are to the left, 3 is the middle and 4 and 5 are to
    # so left, and right are arrays of the form: [1, 2] for left and [4,5] fo
    # since we use superglue and superpoint these have to have the names of t
    # then, the main idea is that we will map everything to the middle image
    # for this, we get the following homographies:
    # 1 to 2, 2 to 3, then we get the right side ones: 5 to 4 and 4 to 3
    # we do it in this manner since we can just multiply them to find all the
    # that map everything to 3
    # after getting all the homographies, we can find out the exact shape of
    # to paint all the images to the final canvas, we create binary roi masks
    # then to conduct less matrix multiplications, we find all the positions
    # homography transformation to them
    # after all this, we can paint all the images to our panorama
    # we will make use of dictionaries to help us out

    image_1_info = {}
    image_2_info = {}
    image_3_info = {}
    image_4_info = {}
    image_5_info = {}

    image_1_info["image"] = cv2.imread(left[0])
    image_2_info["image"] = cv2.imread(left[1])
    image_3_info["image"] = cv2.imread(middle)
    image_4_info["image"] = cv2.imread(right[0])
    image_5_info["image"] = cv2.imread(right[1])

    # 1 to 2
    image_1_info["inliers_1"], image_1_info["inliers_2"], image_1_info["outli
    # 2 to 3
    image_2_info["inliers_1"], image_2_info["inliers_2"], image_2_info["outli
    # 4 to 3
    image_4_info["inliers_1"], image_4_info["inliers_2"], image_4_info["outli
    # 5 to 4
    image_5_info["inliers_1"], image_5_info["inliers_2"], image_5_info["outli
    # 1 -> 3 homography
    image_1_info["tot_homography"] = image_2_info["tot_homography"] @ image_1
    # 5 to 3 homography
    image_5_info["tot_homography"] = image_4_info["tot_homography"] @ image_5
    base_name = task
    #PLOT COST FUNCTIONS:
```

```
if mode == "LM":
    if image_1_info["cost"] is not None:
        plot_function(image_1_info["cost"],base_name+"_LM_cost_image_1.jp
    if image_2_info["cost"] is not None:
        plot_function(image_2_info["cost"],base_name+"LM_cost_image_2.jpg
    if image_4_info["cost"] is not None:
        plot_function(image_4_info["cost"],base_name+"LM_cost_image_4.jpg
    if image_5_info["cost"] is not None:
        plot_function(image_5_info["cost"],base_name+"LM_cost_image_5.jpg
# plot inliers and outliers for different pairs:
# image 1 to 2
plot_inliers_outliers(image_1_info["image"], image_2_info["image"], image
# image 2 to 3
plot_inliers_outliers(image_2_info["image"], image_3_info["image"], image
# image 4 to 3, we swap the orders since we mapped 4 to 3 instead of 3 to
plot_inliers_outliers(image_3_info["image"], image_4_info["image"], image
# image 4 to 5, also swap the orders since we mapped 5 to 4 instead of 4
plot_inliers_outliers(image_4_info["image"], image_5_info["image"], image


# we get the corners
image_1_info["corners"] = get_new_corners(image_1_info["image"],image_1_i
image_2_info["corners"] = get_new_corners(image_2_info["image"],image_2_i
image_4_info["corners"] = get_new_corners(image_4_info["image"],image_4_i
image_5_info["corners"] = get_new_corners(image_5_info["image"],image_5_i
# we just use the identity for the 3 -> 3 homography
image_3_info["corners"] = get_new_corners(image_3_info["image"], np.eye(3
# x min and x max are easy to get since its just the very edge images (1
x_min = int(image_1_info["corners"][:,0].min())
x_max = int(image_5_info["corners"][:,0].max())
# y min and max are a bit more involved, since they are all technically a
# the y min and y max of all 5 images
y_min = int(np.array([image_1_info["corners"][:,1].min(),image_2_info["co
y_max = int(np.array([image_1_info["corners"][:,1].max(),image_2_info["co

resulting_images = {}
# from both of these we can get the shape of the resulting panorama
panorama_w = int(x_max - x_min)
panorama_h = int(y_max - y_min)
# create the panorama
panorama = np.zeros((panorama_h,panorama_w,3)).astype(np.uint8)
# now we just need the binary masks
image_1_info["binary"] = get_binary_mask(panorama, image_1_info["corners"
image_2_info["binary"] = get_binary_mask(panorama, image_2_info["corners"
image_3_info["binary"] = get_binary_mask(panorama, image_3_info["corners"
```

```
    image_4_info["binary"] = get_binary_mask(panorama, image_4_info["corners"
    image_5_info["binary"] = get_binary_mask(panorama, image_5_info["corners"
    # get all the coordinates where the binary mask is 255 in a list so we ca
    # we want to do this so we ONLY consider the spots where there is an imag
    # just less loops to go through when painting the final image
    # but when we do argwhere, the x and y axis are swapped since thats how o
    image_1_info["new_coordinates"] = (np.argwhere(image_1_info["binary"] ==
    image_2_info["new_coordinates"] = (np.argwhere(image_2_info["binary"] ==
    image_3_info["new_coordinates"] = (np.argwhere(image_3_info["binary"] ==
    image_4_info["new_coordinates"] = (np.argwhere(image_4_info["binary"] ==
    image_5_info["new_coordinates"] = (np.argwhere(image_5_info["binary"] ==

    # get the converted positions (apply inverse homography, since we are map
    image_1_info["old_coordinates"] = apply_homography(image_1_info["new_coor
    image_2_info["old_coordinates"] = apply_homography(image_2_info["new_coor
    image_3_info["old_coordinates"] = apply_homography(image_3_info["new_coor
    image_4_info["old_coordinates"] = apply_homography(image_4_info["new_coor
    image_5_info["old_coordinates"] = apply_homography(image_5_info["new_coor
    # with all this information we can now paint everything in our panorama
    panorama = paint_image(panorama, image_1_info["image"], image_1_info["old
    panorama = paint_image(panorama, image_2_info["image"], image_2_info["old
    panorama = paint_image(panorama, image_3_info["image"], image_3_info["old
    panorama = paint_image(panorama, image_4_info["image"], image_4_info["old
    panorama = paint_image(panorama, image_5_info["image"], image_5_info["old
    # save all the images and intermediate binary masks to show in report
    resulting_images["panorama"] = panorama
    resulting_images["1_binary"] = image_1_info["binary"]
    resulting_images["2_binary"] = image_2_info["binary"]
    resulting_images["3_binary"] = image_3_info["binary"]
    resulting_images["4_binary"] = image_4_info["binary"]
    resulting_images["5_binary"] = image_5_info["binary"]
    return resulting_images

# load superglue+superpoint weights
detector = SuperGlue.create()

# TASK ONE
left = ["ece661_sample_images/1.jpg","ece661_sample_images/2.jpg"]
middle = "ece661_sample_images/3.jpg"
right = ["ece661_sample_images/4.jpg","ece661_sample_images/5.jpg"]
task = "task1_"
# least squares homography panorama
mode = "LS"
im = create_panorama(left ,middle, right, detector, sigma=2,epsilon=0.1,n_poi
cv2.imwrite(task + "panorama_" + mode + ".jpg",im["panorama"])
```

```
# own implementation of LM homography panorama
mode = "LM"
im = create_panorama(left ,middle, right, detector, sigma=2,epsilon=0.1,n_poi
cv2.imwrite(task + "panorama_" + mode + ".jpg",im["panorama"])

# TASK TWO
left = ["ece661_sample_images/1_b.jpg","ece661_sample_images/2_b.jpg"]
middle = "ece661_sample_images/3_b.jpg"
right = ["ece661_sample_images/4_b.jpg","ece661_sample_images/5_b.jpg"]
task = "task2_"
# least squares homography panorama
mode = "LS"
im = create_panorama(left ,middle, right, detector, sigma=4,epsilon=0.3,n_poi
cv2.imwrite(task + "panorama_" + mode + ".jpg",im["panorama"])
# own implementation of LM panorama
mode = "LM"
im = create_panorama(left ,middle, right, detector, sigma=4,epsilon=0.3,n_poi
cv2.imwrite(task + "panorama_" + mode + ".jpg",im["panorama"])
```