ECE 661: Homework #5

Due on Thursday, 11 October 2018

Dr. Avinash Kak

Naveen Madapana

1. Homography estimation

1.1 Definition

A homography is a linear transformation that maps physical points from domain plane to a range plane (image plane). A general homography (H), represented by $f : \mathbb{R}^2 \to \mathbb{R}^2$ is a 3×3 nonsingular matrix that transforms 2D physical points from one plane to the other. Also, note that homographies map straight lines to straight lines.

Let $x_1 = \begin{bmatrix} u_1 & v_1 & w_1 \end{bmatrix}^T$ and $x_2 = \begin{bmatrix} u_2 & v_2 & w_2 \end{bmatrix}^T$ be the homogeneous coordinates of two points p_1 and p_2. Suppose p_1 and p_2 represent a particular point in the real world captured by two different cameras. Then, we could estimate a homography H such that:

$$x_2 = Hx_1 \quad \longrightarrow \quad \begin{bmatrix} u_2 \\ v_2 \\ w_2 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ w_1 \end{bmatrix}$$

x_1 and x_2 are homogeneous coordinates. Hence, we can rewrite u_2 and v_2 as u_2/w_2 and v_2/w_2 respectively.

There are three ways in which we can estimate homography H in order to eliminate both the projective distortion and affine distortion. Note that projective distortion preservers straight lines and affine distortion keeps parallel lines parallel. In this homework, linear least squres method via point to point correspondences has been used to estimate the homographies.

1.2 Point to point correspondence

In this approach, it is assumed that we have a one to one correspondence between pixels in the domain plane to the pixels in the range plane. Overall, we need atleast four correspondences (pairs of points) in order to compute the homography in this method.
Without the loss of generality, it can be assumed that $w_1 = 1$ as we are working with homogeneous coordinates. By simplifying the equation further and rearranging the equation in the form of $Ah = 0$, where h is a 9×1 vector consisting of elements in the homography H, we will get,

$$u_2(H_{31}u_1 + H_{32}v_1 + H_{33}) = H_{11}u_1 + H_{12}v_1 + H_{13}w_1$$
$$v_2(H_{31}u_1 + H_{32}v_1 + H_{33}) = H_{21}u_1 + H_{22}v_1 + H_{23}w_1$$

Hence, each pair of matching point will yield two equations (for x and y). where h is a 9×1 vector consisting of elements in the homography H and A is a $2m \times 9$ matrix (m being the number matching pairs of points selected to estimate h). Let a_u and a_v be the coefficients corresponding to x and y dimensions.

$$Ah = 0$$

Where:

$$\begin{bmatrix} -u_1 & -v_1 & -1 & 0 & 0 & 0 & u_1u_2 & v_1u_2 & u_2 \\ 0 & 0 & 0 & -u_1 & -v_1 & -1 & u_1v_2 & v_1v_2 & v_2 \end{bmatrix} h = 0$$

$$h = \begin{bmatrix} H_{11} & H_{12} & H_{13} & H_{21} & H_{22} & H_{23} & H_{31} & H_{32} & H_{33} \end{bmatrix}^T$$

Now if we choose m pairs of points we will get the system $Ah = 0$:

$$\begin{bmatrix} -u_{11} & -v_{11} & -1 & 0 & 0 & 0 & u_{11}u_{12} & v_{11}u_{12} & u_{12} \\ 0 & 0 & 0 & -u_{11} & -v_{11} & -1 & u_{11}v_{12} & v_{11}v_{12} & v_{12} \\ -u_{21} & -v_{21} & -1 & 0 & 0 & 0 & u_{21}u_{22} & v_{21}u_{22} & u_{22} \\ 0 & 0 & 0 & -u_{21} & -v_{21} & -1 & u_{21}v_{22} & v_{21}v_{22} & v_{22} \\ & & & & \vdots & & & & \\ -u_{n1} & -v_{n1} & -1 & 0 & 0 & 0 & u_{n1}u_{n2} & v_{n1}u_{n2} & u_{n2} \\ 0 & 0 & 0 & -u_{n1} & -v_{n1} & -1 & u_{n1}v_{n2} & v_{n1}v_{n2} & v_{n2} \end{bmatrix} \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \\ H_{33} \end{bmatrix} = 0$$

1.3 Linear Least Squares Estimation

Linear-Least Squares method lets us estimate a homography H s.t. $X' = HX$, where (X, X') is a matched pair of points between two images. We can rewrite H as:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \rightarrow H = \begin{bmatrix} {h^1}^T \\ {h^2}^T \\ {h^3}^T \end{bmatrix}, \quad \text{where} \;\; h^1 = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \end{bmatrix}, \; h^2 = \begin{bmatrix} h_{21} \\ h_{22} \\ h_{23} \end{bmatrix}, \; h^3 = \begin{bmatrix} h_{21} \\ h_{22} \\ h_{23} \end{bmatrix}$$

This equation can rewritten as the following cross product:

$$X' \times \begin{bmatrix} {h^1}^T X \\ {h^2}^T X \\ {h^3}^T X \end{bmatrix} = 0$$

If we have a number of n of pairs $X_n = \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix}, X'_n = \begin{bmatrix} x'_n \\ y'_n \\ z'_n \end{bmatrix}$ and $2n > 8$, we will get an overdetermined system that has no solution. In this case, we want to find the optimal h that minimizes Ah The elaborated expression for $Ah = 0$ is given in the previous section.

The vector h can be estimated using singular value decomposition. We want to estimate h such that:

$$\underset{h}{\operatorname{argmin}} \frac{|Ah|^2}{|h|^2}$$

Let $Ah = \lambda h$, where λ is an eigenvalue and h is corresponding eigenvector.

$$\underset{h}{\operatorname{argmin}} \frac{|\lambda h|^2}{|h|^2} \Rightarrow \underset{h}{\operatorname{argmin}} \frac{\lambda^2 |h|^2}{|h|^2} \Rightarrow \underset{h}{\operatorname{argmin}} \lambda^2$$

Hence, h is the eigenvector of A with lowest eigenvalue. Therefore, we compute SVD of A i.e. $A = USV^T$ and choose the last column of V^T. Now, we reshape h in order to get a $H_{3\times3}$.

2. Feature Matching

The features between two images are automatically obtained using OpenCV implementation of the SIFT and SURF. Both SIFT and SURF perform equally well. Hence, the results are not compared between SIFT and SURF. The approximate NCC threshold for SIFT is 0.97 and SURF is 0.99.

2.1 Establishing Point Correspondences

Let u and v be the vectors of same length (k). Let μ_u and μ_v be the mean of u and v respectively.

2.1.1 Sum Squared Differences (SSD)

$$SSD(u, v) = \sum_i (u_i - v_i)^2 \tag{1}$$

2.1.2 Normalized Cross Correlation (NCC)

$$NCC(u, v) = \frac{\sum_i (u_i - \mu_u)(v_i - \mu_v)}{\sqrt{\sum_i (u_i - \mu_u)^2 \sum_i (v_i - \mu_v)^2}} \tag{2}$$

The results obtained by using NCC are better than SSD as the former is mean normalized and is not sensitive the uniform increase/decrease in the intensities. In experiments, the value of threshold NCC ranges any where from 0.97 to 0.999.

3. RANSAC Implementation

It is crucial to remove outliers when we work with automatic homography estimation. RANSAC (Ramndom Sampling and Consensus) was implemented in this homework to eliminate the outliers.

First, a small number of n random matching pairs are selected and the homography is estimated using Linear least squres. Next, the no. of inliers and outliers were computed using a descision threshold (δ) which is close to 4 to 10 pixels. The pseudo code of RANSAC algorithm is given in Algorithm 1. Next, the homography with the largest inlier set was chosen. Next, the final homogrpahy is obtained using all of the inlier point correspondences.

The parameters of the RANSAC approach are depicted in the Table 1.

Algorithm 1 RANSAC

$P_1 = SIFT\ matched\ points\ for\ first\ image$
$P_2 = SIFT\ matched\ points\ for\ second\ image$
for i in $1 : N$ **do**
  Set of $inliers \leftarrow []$
  Set of $outliers \leftarrow []$
  $RP_1 \leftarrow n$ random matched points from P_1
  $RP_2 \leftarrow$ corresponding n random matched points from P_2
  $H \leftarrow estimateHomography(RP_1, RP_2)$ # Using linear least squres method
  for j in $1 : size(RP_1)$ **do**
    $p'_j \leftarrow$ real matched point
    $transformed\_p_j \leftarrow H \times p_j$
    $distance \leftarrow computeL2Norm(p'_j, transformed\_p_j)$
    if $distance < \delta$ **then**:
      $inliers.add(j)$
    end if
    if $distance < dist_threshold$ **then**:
      $outliers.add(j)$
    end if
  end for
  if Choose the random sample that produced largest no. of inliers and # of inliers > M **then**:
    Compute the final homography with all the inliers. This is the best homography estimated by RANSAC.
  end if
end for

Parameter Name	Value / Formula
δ	1-10
p	0.99
e	0.1 - 0.25
n	6 - 8
N	$\frac{\log(1-p)}{\log(1-(1-e)^2)} \approx 8$ (for n = 8)
M	$(1-e)\times$(number of matched points)

Table 1: Parameters of RANSAC Algorithm

4. Levenberg-Marquardt (LM): Non Linear Least Squres Approach

Let $\overrightarrow{f}(\overrightarrow{p}_k)$ be the function that estimates the corresponding pixel coordinates of domain image in the target image. Let the matching point to point correspondences in the domain image be organized as $X = [x_1, y_1, x_2, y_2 \ldots]$. The function f depnds on the 9 variables given as.

$\overrightarrow{p}_k = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{21} & h_{22} & h_{23} & h_{31} & h_{32} & h_{33} \end{bmatrix}^T$

Let the Jacobian of f be defined as.

$J_{\overrightarrow{f}}$ be the Jacobian of $\overrightarrow{f}(\overrightarrow{p}_k)$.

The error that is minimized using LM approach is given as:

$$C(p) = ||X - \overrightarrow{f}(\overrightarrow{p})||^2$$

$$\epsilon(\overrightarrow{p}) = X - \overrightarrow{f}(\overrightarrow{p}$$

$$C(p) = \epsilon(\overrightarrow{p})^T \epsilon(\overrightarrow{p})$$

From Gauss Newton approach, we know that,

$$\overrightarrow{\delta_p} = (J_{\overrightarrow{f}}^T J_{\overrightarrow{f}})^{-1} J_{\overrightarrow{f}}^T \epsilon(\overrightarrow{p})$$

This equation is modified in the LM as the following. The value of μ controls the extent to which we are doing Gradient descent and the Gauss Newton.

The value of μ is changed depending on how the cost function changes when we go from $\overrightarrow{p_k}$ to $\overrightarrow{p_{k+1}}$. If the cost increases, then we will reset the value of parameters, change the value of μ and start all over again.

$$\overrightarrow{\delta_p} = (J_{\overrightarrow{f}}^T J_{\overrightarrow{f}} + \mu I)^{-1} J_{\overrightarrow{f}}^T \epsilon(\overrightarrow{p})$$

5. Results

Two sets of images were considered for image stitching. Each set consisted of seven images.

Set 1

Original Images

(a) Image 1

(b) Image 2

(c) Image 3

(d) Image 4

(e) Image 5

(f) Image 6

Stiching Image 1 - Image 2

Stiching Image 6 and Image 7

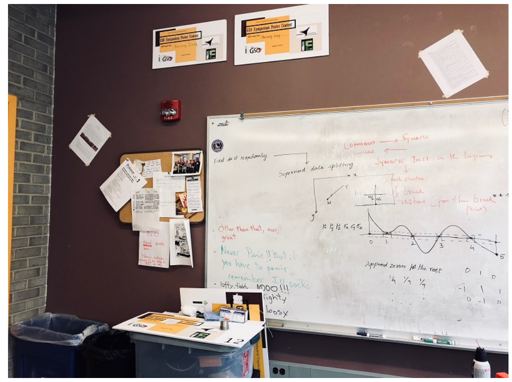
Stiching Image 5, 6 and 7

Overall Stiching

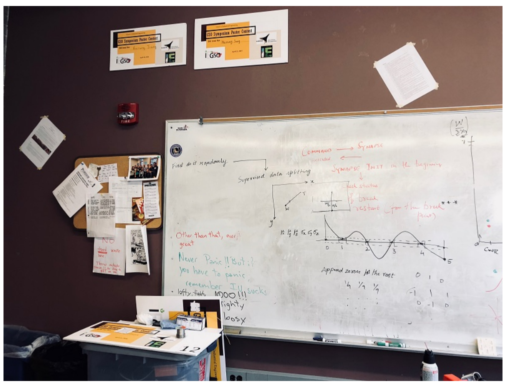
Set 2

Original Images

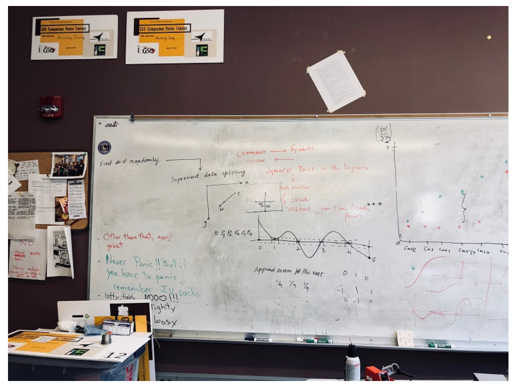
(a) Image 1



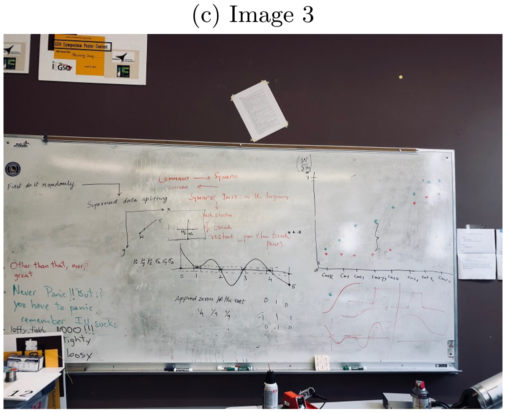
(b) Image 2



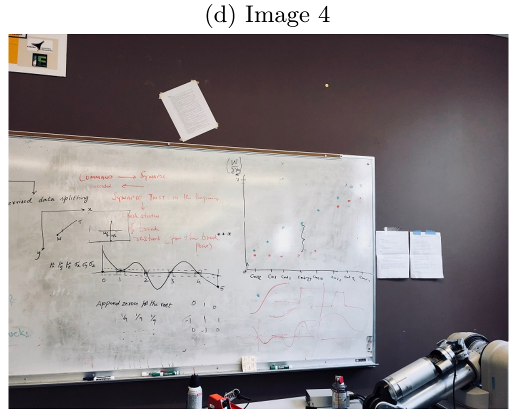
(c) Image 3



(d) Image 4

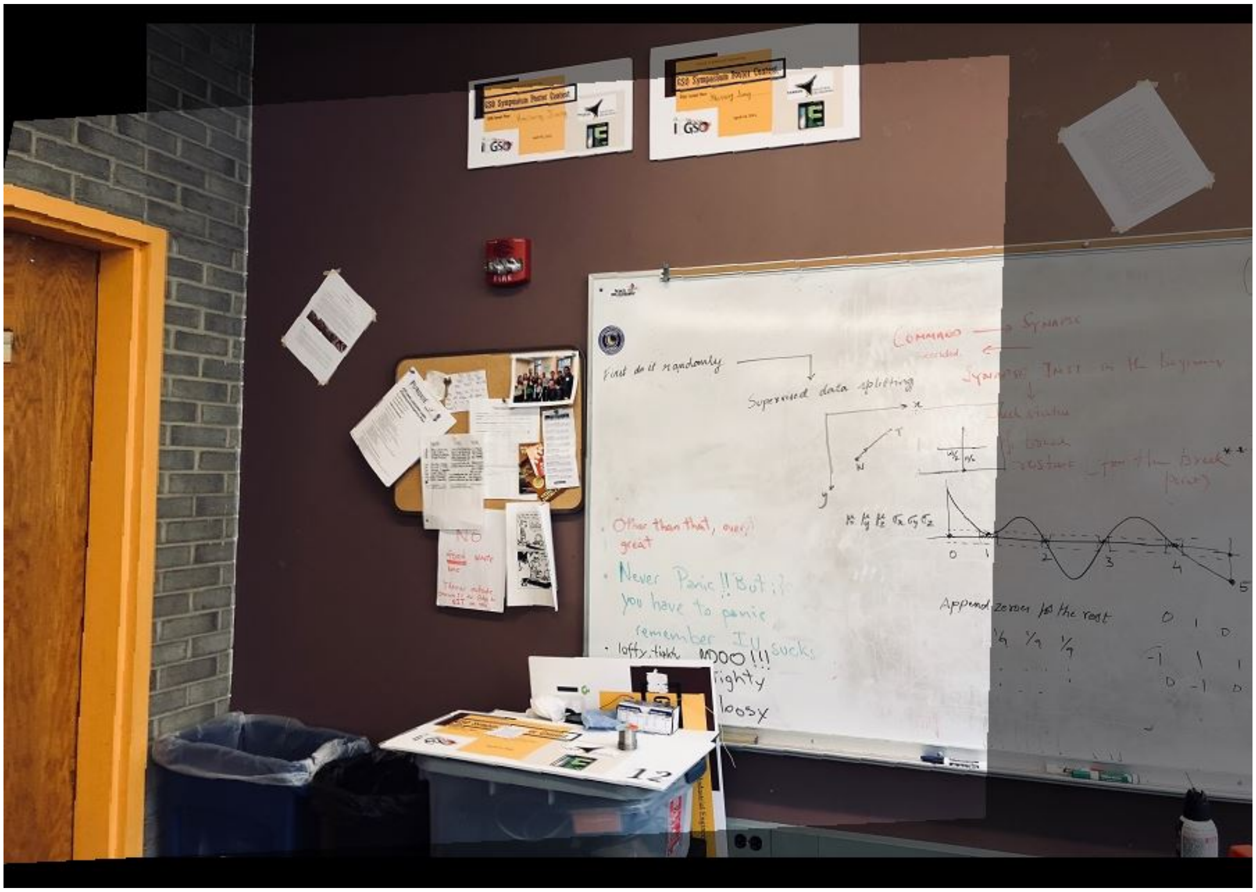


(e) Image 5

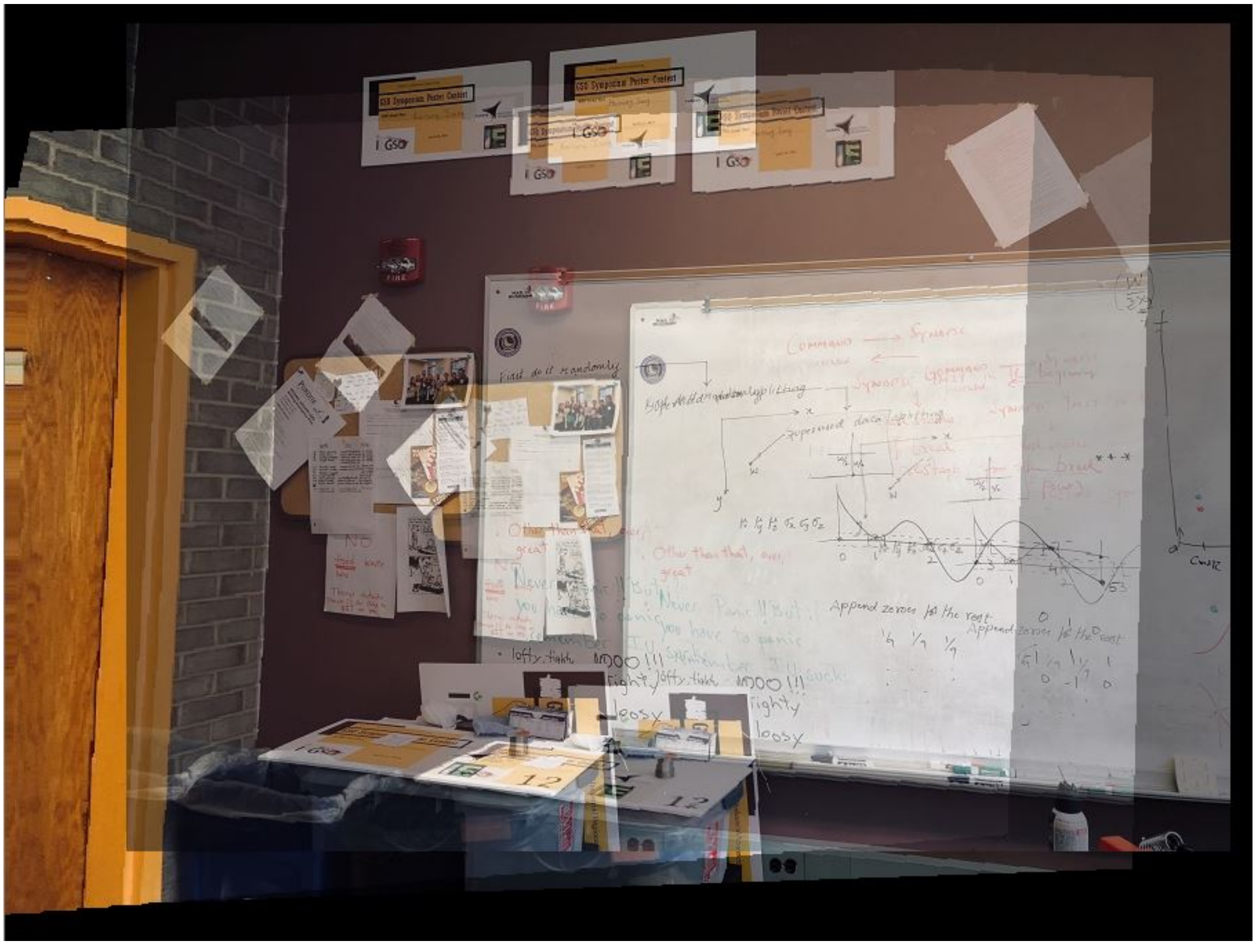


(f) Image 6

Stiching Image 1 - Image 2



Stiching Image 1, 2 and 3



Discssion

Though the image stitching works very well for two to three images, it does not work well when the number of images is more than five. Hence appropriate scaling need to be used.

Further, it was observed that the using non linear least squares after the linear least squres gives better results. Rigorous tuning is needed for the hyperparameters for good stitch.

Dr. Kak's Non linear least squares module was used to to perform LM Optimizaiton.

Code

```
########### HELPERS ##############
import cv2
import numpy as np
```

```python
import os, time, sys

from NonlinearLeastSquares import NonlinearLeastSquares as NLS

hvars = ['h11', 'h12', 'h13', 'h21', 'h22', 'h23', 'h31', 'h32']
Nx = '(h11*{0}+h12*{1}+h13)'
Ny = '(h21*{0}+h22*{1}+h23)'
D = '(h31*{0}+h32*{1}+1)'

def mosaic_two_images(img1, img2, H):
    if(isinstance(img1, str)):
        img1 = cv2.imread(img1)
    if(isinstance(img2, str)):
        img2 = cv2.imread(img2)

    ## H: 2 --> 1
    ## Hinv: 1 --> 2

    ## img1 is assumed to be on the left and img2 on the right.
    Hinv = hinv(H)

    img2_cpts, t_img2_cpts = get_pts(img2.shape, H = H, corners = True)
    img1_cpts, _ = get_pts(img1.shape, H = H, corners = True)
    xmin, ymin = np.min(np.append(t_img2_cpts, img1_cpts, axis = 0), axis = 0)
    xmax, ymax = np.max(np.append(t_img2_cpts, img1_cpts, axis = 0), axis = 0)

    if(xmin < 0):
        t_img2_cpts[0] -= xmin
        W_img = xmax - xmin
    else:
        W_img = xmax
        xmin = 0
    if(ymin < 0):
        t_img2_cpts[1] -= ymin
        H_img = ymax - ymin
    else:
        H_img = ymax
        ymin = 0

    # print xmin, ymin
    # print xmax, ymax

    final_img = np.zeros((H_img+1, W_img+1, 3), dtype = np.uint8)

    _cpts = real_to_homo(t_img2_cpts) # homo. representation
    _cent_cpts = np.mean(_cpts, axis = 0)# centroid of four points

    # Find the four lines of quadrilateral
    lines = [np.cross(_cpts[0], _cpts[1]), np.cross(_cpts[1], _cpts[2]), np.cross(_cpts[2], _cpts[3]

    ## Finding points in the base that are present in the quadrilateral
    base_bool = np.zeros(final_img.shape[:-1]).flatten() == 0 # True -> inside the quadrilateral
    base_all_pts, _ = get_pts(final_img.shape) # get all pts
```

```
    for line in lines:
        line = line / line[-1]
        sn = int(np.sign(np.dot(_cent_cpts, line)))
        nsn = np.int8(np.sign(np.dot(real_to_homo(base_all_pts), line)))
        base_bool = np.logical_and(base_bool, nsn==sn)
    base_bool = base_bool
    base_bool = np.reshape(base_bool, (final_img.shape[0], final_img.shape[1]))
    row_ids, col_ids = np.nonzero(base_bool)
    des_base_pts = np.array([col_ids, row_ids])
    des_base_pts = des_base_pts.transpose() + [xmin, ymin]
    des_base_pts = des_base_pts.transpose()

    # Find corresponding points in the template image
    trans_des_base_pts = homo_to_real(np.dot(hinv(H), real_to_homo(des_base_pts.T).T).T).astype(int)

    # Clip the points
    trans_des_base_pts[0] = np.clip(trans_des_base_pts[0], 0, img2.shape[1]-1)
    trans_des_base_pts[1] = np.clip(trans_des_base_pts[1], 0, img2.shape[0]-1)

    des_base_pts = des_base_pts.transpose() - [xmin, ymin]
    des_base_pts = des_base_pts.transpose()

    # print np.max(trans_des_base_pts, axis = 1)
    # print img2.shape

    final_img[des_base_pts[1], des_base_pts[0], :] = img2[trans_des_base_pts[1], trans_des_base_pts[


    # img2_pts, t_img2_pts = get_pts(img2.shape, H = H)
    # t_img2_pts -= [xmin, ymin]
    # final_img[t_img2_pts[:,1], t_img2_pts[:, 0]] = img2[img2_pts[:, 1], img2_pts[:, 0]]

    # cv2.imshow('Partial image', final_img)
    # cv2.waitKey(0)

    img1_pts, _ = get_pts(img1.shape, targ_shap = final_img.shape)
    m_img1_pts = img1_pts - [xmin, ymin]

    # print np.max(m_img1_pts, axis = 0)
    # print final_img.shape

    final_img[m_img1_pts[:, 1], m_img1_pts[:, 0]] = np.uint8(0.5*np.float32(final_img[m_img1_pts[:,

    cv2.imshow('Mosaic image', final_img)
    cv2.waitKey(0)

    return final_img

def senc(value): return '('+str(value)+')'

def fvec_row(x, y, axis = 'x'):
    if(axis == 'x'):
        fvec = Nx + '/' + D
```

```
    else:
        fvec = Ny + '/' + D
    return fvec.format(senc(x), senc(y))

def jac_row(x, y, axis = 'x'):
    d = [0]*8
    if(axis == 'x'):
        d[0] = '{0}'+'/'+D
        d[1] = '{1}'+'/'+D
        d[2] = '1'+'/'+D
        d[3] = '0'
        d[4] = '0'
        d[5] = '0'
        d[6] = '(-'+Nx+'*'+'{0})/' + '(' + D + '**2)'
        d[7] = '(-'+Nx+'*'+'{1})/' + '(' + D + '**2)'
        # d[8] = '(-'+Nx+'*'+'1)/' + '(' + D + '**2)'
    else:
        d[0] = '0'
        d[1] = '0'
        d[2] = '0'
        d[3] = '{0}'+'/'+D
        d[4] = '{1}'+'/'+D
        d[5] = '1'+'/'+D
        d[6] = '(-'+Ny+'*'+'{0})/' + '(' + D + '**2)'
        d[7] = '(-'+Ny+'*'+'{1})/' + '(' + D + '**2)'
        # d[8] = '(-'+Ny+'*'+'1)/' + '(' + D + '**2)'

    for idx, _ in enumerate(d):
        d[idx] = d[idx].format(senc(x), senc(y))

    return d

def LM_Minimizer(point_corresps, H_init, max_iter = 200, \
                 delta_for_jacobian = 0.000001, \
                 delta_for_step_size = 0.0001, debug = False):

    '''
        H: 2 --> 1
    '''

    nls =  NLS(max_iterations = max_iter, \
               delta_for_jacobian = delta_for_jacobian, \
               delta_for_step_size = delta_for_step_size, debug = debug)

    pts1 = point_corresps[:,:2]
    pts2 = point_corresps[:,2:]

    X = pts1.flatten().reshape(-1, 1) # [x1, y1, x2, y2, ...]

    Jac = []
    Fvec = []
    for x, y in pts2:
        fx = fvec_row(x, y, 'x')
```

```
        fy = fvec_row(x, y, 'y')
        dfx = jac_row(x, y, 'x')
        dfy = jac_row(x, y, 'y')
        Jac.append(dfx)
        Jac.append(dfy)
        Fvec.append(fx)
        Fvec.append(fy)

    Fvec = np.array(Fvec).reshape(-1, 1)
    Jac = np.array(Jac)

    nls.set_Fvec(Fvec)
    nls.set_X(X)
    nls.set_jacobian_functionals_array(Jac)
    nls.set_params_ordered_list(hvars)
    nls.set_initial_params(dict(zip(hvars, H_init.flatten().tolist())))

    # print Jac
    # print ''
    # print Fvec

    return nls.leven_marq()

def count_inliers(point_corresps, H, delta = 40):
    #####################
    # Input:
    #
    #   point_corresps: np.ndarray of shape _ x 4
    #       Column 0 and 1 correspond to [x_coordinate, y_coordinate] of img1
    #       Column 2 and 3 correspond to [x_coordinate, y_coordinate] of img2
    #       It is point correspondences between two images.
    #
    #   H: Homography (np.ndarray) of shape 3 x 3
    #
    #   delta: decision threshold. Either threshold on SSD or NCC to
    #       determine if a corresp. is an inlier or outlier.
    #       Default value is 40 pixels.
    #
    # Return:
    #
    #   inlier_sz: size of the inlier set. No. of points that are inliers.
    #
    #   inlier_ids: a np array containing indices of points in inlier set
    #
    #####################

    pts1 = point_corresps[:,:2]
    pts2 = point_corresps[:,2:]

    homo_pts2 = real_to_homo(pts2)
    trans_homo_pts2 = np.dot(H, homo_pts2.transpose())
    trans_pts2 = homo_to_real(trans_homo_pts2.transpose())

```

```
    err = np.linalg.norm(pts1 - trans_pts2, axis = 1)

    inlier_sz = int(np.sum(err < delta))

    return inlier_sz, np.nonzero(err < delta)[0]

def ransac(point_corresps, param_p = 0.99, eps = 0.1, param_n = 8, delta = 40):
    #####################
    # Input:
    #
    #   point_corresps: np.ndarray of shape _ x 4
    #       Column 0 and 1 correspond to [x_coordinate, y_coordinate] of img1
    #       Column 2 and 3 correspond to [x_coordinate, y_coordinate] of img2
    #       It is point correspondences between two images.
    #
    #   p: prob. that at least one of N trials will be free of outliers.
    #       Default value is 0.99/
    #
    #   eps: prob. that a pt. corresp. is an outlier
    #
    #   n: min. no. of point correspondences needed to estimate the homography
    #
    #   delta: decision threshold. Either threshold on SSD or NCC to
    #       determine if a corresp. is an inlier or outlier.
    #       Default value is 40 pixels.
    #
    # Return:
    #   H: Homography from 2 --> 1. np.ndarray of shape 3 x 3.
    #
    #   new_matches: Point correspondences of points in inlier set.
    #       Datatype is similar to point_corresps but with only inliers.
    #####################

    # Determine num_trials (N). No. of trials or times we need to run RANSAC
    #   so that at least one trial will contain all inliers
    N = np.int16(np.log(1 - param_p) / np.log(1 - (1 - eps)**param_n))

    # thresh_inlier_sz (M)
    #   A minimum size of inlier set that is acceptable.
    n_total = len(point_corresps)
    M = np.int((1 - eps) * n_total)

    print 'Len. of point_corresps: ', len(point_corresps)
    print 'No. of trials (N): ', N
    print 'Min. acceptable size of inlier set (M): ', M

    trial_info = []

    for tr_idx in range(N):
        ## Find 'param_n' point correspondences at random
        tr_match_ids = np.random.randint(0, n_total, param_n)
        # If point correspondences repeat, try until you get unique ids.
        while(len(tr_match_ids) < param_n):
```

```
            tr_match_ids = np.random.randint(0, n_total, param_n)

        ## Find homography with the obtained point correspondences
        tr_matches = point_corresps[tr_match_ids, :]
        tr_H, _ = find_homography_2d(tr_matches[:,:2], tr_matches[:,2:])

        ## Find the size of inlier set
        inlier_sz, _ = count_inliers(point_corresps, tr_H, delta = delta)

        # If size of inlier set exceed M, store the trial information.
        if(inlier_sz >= M):
            trial_info.append((inlier_sz, tr_match_ids))

    if len(trial_info) == 0: return None, None
    # Find the inlier set with maximum inlier size
    inlier_sz_list, inlier_pt_ids = zip(*trial_info)
    best_inlier_tr_idx = np.argmax(inlier_sz_list)
    best_inlier_ids = inlier_pt_ids[int(best_inlier_tr_idx)]

    print '% of inliers:', np.max(inlier_sz_list)/float(n_total)

    ## Estimate the homography with best inlier ids (only param_n corresp.)
    temp_matches = point_corresps[best_inlier_ids, :]
    temp_H, _ = find_homography_2d(temp_matches[:,:2], temp_matches[:,2:])

    ## Find all inlier ids and estimate the homography
    _, all_inlier_ids = count_inliers(point_corresps, temp_H, delta = delta)
    new_matches = point_corresps[all_inlier_ids, :]
    H, _ = find_homography_2d(new_matches[:,:2], new_matches[:,2:])

    return new_matches, H

def find_homography_gh(pts_list):
    ## Find general homography that removes both projective and affine distortion.
    # pts should contain points either in clockwise or anti clockwise order
    # Assertion
    # assert isinstance(pts, np.ndarray), 'pts should be a numpy array'
    # assert pts.shape[1] == 2, 'pts should have two columns'
    # assert pts.shape[0] == 4, 'pts should have four rows'

    ln_pairs = []
    for pts in pts_list:
        pts = real_to_homo(pts)
        ln_pairs.append((nmlz(np.cross(pts[0,:], pts[1,:])), nmlz(np.cross(pts[1,:], pts[2,:]))))
        ln_pairs.append((nmlz(np.cross(pts[1,:], pts[2,:])), nmlz(np.cross(pts[2,:], pts[3,:]))))
        ln_pairs.append((nmlz(np.cross(pts[2,:], pts[3,:])), nmlz(np.cross(pts[3,:], pts[0,:]))))
        ln_pairs.append((nmlz(np.cross(pts[3,:], pts[0,:])), nmlz(np.cross(pts[0,:], pts[1,:]))))
        ln_pairs.append((nmlz(np.cross(pts[0,:], pts[2,:])), nmlz(np.cross(pts[1,:], pts[3,:]))))
    Y = []
    for line1, line2 in ln_pairs:
        r1 = line1[0]*line2[0]
        r2 = line1[0]*line2[1] + line1[1]*line2[0]
        r3 = line1[1]*line2[1]
```

```
        r4 = line1[0]*line2[2] + line1[2]*line2[0]
        r5 = line1[1]*line2[2] + line1[2]*line2[1]
        r6 = line1[2]*line2[2]
        Y.append([r1, r2, r3, r4, r5, r6])
    # print Y
    [_, _, V] = np.linalg.svd(Y, full_matrices = True)
    h = V.T[:,-1]
    h = h / h[-1]
    S = np.array([[h[0], h[1], h[3]],[h[1], h[2], h[4]],[h[3], h[4], h[5]]])
    # print S

    # Find 2 x 2
    [U, D2, V] = np.linalg.svd(S[:-1, :-1], full_matrices = True)

    H = np.eye(3)
    H[:-1,:-1] = np.dot(np.dot(V, np.diag(np.sqrt(D2))), V.T)
    vv = np.dot(np.linalg.inv(H[:-1,:-1]), np.array([h[3], h[4]])).T
    vv = vv / np.linalg.norm(vv)
    H[2,:-1] = vv
    return H

def find_homography_af(pts):
    ## Find homography resulting from vanishing line approach.
    # pts should contain points either in clockwise or anti clockwise order
    # Assertion
    assert isinstance(pts, np.ndarray), 'pts should be a numpy array'
    assert pts.shape[1] == 2, 'pts should have two columns'
    assert pts.shape[0] == 4, 'pts should have four rows'

    pts = real_to_homo(pts)
    ln_pairs = []
    ln_pairs.append((np.cross(pts[0,:], pts[1,:]), np.cross(pts[1,:], pts[2,:])))
    ln_pairs.append((np.cross(pts[1,:], pts[2,:]), np.cross(pts[2,:], pts[3,:])))
    ln_pairs.append((np.cross(pts[2,:], pts[3,:]), np.cross(pts[3,:], pts[0,:])))
    ln_pairs.append((np.cross(pts[0,:], pts[2,:]), np.cross(pts[1,:], pts[3,:])))
    A = []
    for line1, line2 in ln_pairs:
        r1 = line1[0]*line2[0]
        r2 = line1[0]*line2[1] + line1[1]*line2[0]
        r3 = line1[1]*line2[1]
        A.append([r1, r2, r3])
    print A
    [_, _, V] = np.linalg.svd(A, full_matrices = True)
    h = V.T[:,-1]
    h = h / h[-1]
    S = np.array([[h[0], h[1]],[h[1], h[2]]])
    print S
    [_, D2, V] = np.linalg.svd(S, full_matrices = True)
    H = np.eye(3)
    H[0:-1,0:-1] = np.dot(np.dot(V, np.diag(np.sqrt(D2))), V.T)
    return H

def find_homography_vl(pts):
```

```
    ## Find homography resulting from vanishing line approach.
    # pts should contain points either in clockwise or anti clockwise order
    # Assertion
    assert isinstance(pts, np.ndarray), 'pts should be a numpy array'
    assert pts.shape[1] == 2, 'pts should have two columns'
    assert pts.shape[0] == 4, 'pts should have four rows'

    pts = real_to_homo(pts)
    line1 = np.cross(pts[0,:], pts[1,:])
    line2 = np.cross(pts[2,:], pts[3,:])
    point1 = np.cross(line1, line2)

    line3 = np.cross(pts[0,:], pts[3,:])
    line4 = np.cross(pts[1,:], pts[2,:])
    point2 = np.cross(line3, line4)

    van_line = np.cross(point1, point2)
    van_line = van_line / van_line[-1]
    print 'van_line: ', van_line

    H = np.eye(3)
    H[2, 0] = van_line[0]
    H[2, 1] = van_line[1]

    return H

def find_homography_2d(pts1, pts2):
    # H: 2 --> 1

    # Assertion
    assert pts1.shape[1] == 2, 'pts1 should have two columns'
    assert pts2.shape[1] == 2, 'pts2 should have two columns'
    assert pts1.shape[0] == pts2.shape[0], 'pts1 and pts2 should have same number of rows'

    # Forming the matrix A (8 x 9)
    A = []
    for (x1, y1), (x2, y2) in zip(pts1, pts2):
        A.append([x2, y2, 1, 0, 0, 0, -1*x1*x2, -1*x1*y2, -1*x1])
        A.append([0, 0, 0, x2, y2, 1, -1*y1*x2, -1*y1*y2, -1*y1])
    A = np.array(A)

    [U, S, V] = np.linalg.svd(A, full_matrices = True)
    h = V.T[:,-1]
    h = h / h[-1]

    # # Finding the homography. H[3,3] is assumed 1.
    # h = np.dot(np.linalg.pinv(A[:,:-1]), -1*A[:,-1])
    # h = np.append(h, 1)

    H = np.reshape(h, (3, 3))
    return H, hinv(H)

def apply_homography2(img_path, H, num_partitions = 1, suff = ''):
```

```
    img = cv2.imread(img_path)
    img[0,:], img[:,0], img[-1,:], img[:,-1] = 0, 0, 0, 0

    xv, yv = np.meshgrid(range(0, img.shape[1], img.shape[1]-1), range(0, img.shape[0], img.shape[0]-
    img_pts = np.array([xv.flatten(), yv.flatten()]).T
    trans_img_pts = np.dot(H, real_to_homo(img_pts).T)
    ttt = homo_to_real(trans_img_pts.T).T
    _w = np.max(ttt[0, :]) - np.min(ttt[0, :])
    _h = np.max(ttt[1, :]) - np.min(ttt[1, :])
    l1, l2 = img.shape[1] / _w, img.shape[0] / _h
    K = np.diag([l1, l2, 1])
    H = np.dot(K, H)

    xv, yv = np.meshgrid(range(0, img.shape[1], img.shape[1]-1), range(0, img.shape[0], img.shape[0]-
    img_pts = np.array([xv.flatten(), yv.flatten()]).T
    trans_img_pts = np.dot(H, real_to_homo(img_pts).T)
    trans_img_pts = homo_to_real(trans_img_pts.T).astype(int)

    xmin, ymin = np.min(trans_img_pts[:,0]), np.min(trans_img_pts[:,1])
    xmax, ymax = np.max(trans_img_pts[:,0]), np.max(trans_img_pts[:,1])
    W_new = xmax - xmin
    H_new = ymax - ymin

    img_new = np.zeros((H_new+1, W_new+1, 3), dtype = np.uint8)
    print 'Shape of new image: ', img_new.shape

    x_batch_sz = int(W_new/float(num_partitions))
    y_batch_sz = int(H_new/float(num_partitions))
    for x_part_idx in range(num_partitions):
        for y_part_idx in range(num_partitions):
            x_start, x_end = x_part_idx*x_batch_sz, (x_part_idx+1)*x_batch_sz
            y_start, y_end = y_part_idx*y_batch_sz, (y_part_idx+1)*y_batch_sz
            xv, yv = np.meshgrid(range(x_start, x_end), range(y_start, y_end))
            xv, yv = xv + xmin, yv + ymin
            img_new_pts = np.array([xv.flatten(), yv.flatten()]).T
            trans_img_new_pts = np.dot(hinv(H), real_to_homo(img_new_pts).T)
            trans_img_new_pts = homo_to_real(trans_img_new_pts.T).astype(int)
            trans_img_new_pts[:,0] = np.clip(trans_img_new_pts[:,0], 0, img.shape[1]-1)
            trans_img_new_pts[:,1] = np.clip(trans_img_new_pts[:,1], 0, img.shape[0]-1)
            img_new_pts = img_new_pts - [xmin, ymin]
            # This is the bottle nect step. It takes the most time.
            img_new[img_new_pts[:,1].tolist(), img_new_pts[:,0].tolist(), :] = img[trans_img_new_pts

    fname, ext = tuple(os.path.basename(img_path).split('.'))
    write_filepath = os.path.join(os.path.dirname(img_path), fname+suff+'.'+ext)
    print write_filepath
    cv2.imwrite(write_filepath, img_new)


def apply_homography(img_path, H, num_partitions = 1, suff = ''):
    img = cv2.imread(img_path)
    img[0,:], img[:,0], img[-1,:], img[:,-1] = 0, 0, 0, 0

```

```
    xv, yv = np.meshgrid(range(0, img.shape[1], img.shape[1]-1), range(0, img.shape[0], img.shape[0]-
    img_pts = np.array([xv.flatten(), yv.flatten()]).T
    trans_img_pts = np.dot(H, real_to_homo(img_pts).T)
    trans_img_pts = homo_to_real(trans_img_pts.T).astype(int)

    print 'trans_img_pts'
    print trans_img_pts

    xmin, ymin = np.min(trans_img_pts[:,0]), np.min(trans_img_pts[:,1])
    xmax, ymax = np.max(trans_img_pts[:,0]), np.max(trans_img_pts[:,1])
    W_new = xmax - xmin
    H_new = ymax - ymin

    img_new = np.zeros((H_new+1, W_new+1, 3), dtype = np.uint8)
    print 'Shape of new image: ', img_new.shape

    x_batch_sz = int(W_new/float(num_partitions))
    y_batch_sz = int(H_new/float(num_partitions))
    for x_part_idx in range(num_partitions):
        for y_part_idx in range(num_partitions):
            x_start, x_end = x_part_idx*x_batch_sz, (x_part_idx+1)*x_batch_sz
            y_start, y_end = y_part_idx*y_batch_sz, (y_part_idx+1)*y_batch_sz
            xv, yv = np.meshgrid(range(x_start, x_end), range(y_start, y_end))
            xv, yv = xv + xmin, yv + ymin
            img_new_pts = np.array([xv.flatten(), yv.flatten()]).T
            trans_img_new_pts = np.dot(hinv(H), real_to_homo(img_new_pts).T)
            trans_img_new_pts = homo_to_real(trans_img_new_pts.T).astype(int)
            trans_img_new_pts[:,0] = np.clip(trans_img_new_pts[:,0], 0, img.shape[1]-1)
            trans_img_new_pts[:,1] = np.clip(trans_img_new_pts[:,1], 0, img.shape[0]-1)
            img_new_pts = img_new_pts - [xmin, ymin]
            # This is the bottle nect step. It takes the most time.
            img_new[img_new_pts[:,1].tolist(), img_new_pts[:,0].tolist(), :] = img[trans_img_new_pts

    fname, ext = tuple(os.path.basename(img_path).split('.'))
    write_filepath = os.path.join(os.path.dirname(img_path), fname+suff+'.'+ext)
    print write_filepath
    cv2.imwrite(write_filepath, img_new)

def get_pts(shap, corners = False, H=None, targ_shap = None):
    #####
    #
    # Description:
    #
    # Input:
    #   shap: tuple (num_rows, num_cols, optional) - for given image
    #   corners: if True, only corner points, if False, all points
    #   H: 3 x 3 ndarray - given image to target image
    #   targ_shap: tuple (num_rows, num_cols, optional) - for target image
    #
    #   if H is not None, apply the homography
    #   if targ_shap is not None, clip the transformed pts accordingly.
    #
    # Return:
```

```
    #   trasformed points. In (x, y) format. It depends on if H, targ_shap are None
    #
    #####

    M, N = shap[0], shap[1]
    if(corners):
        pts = np.array([[0, 0],[N-1, 0],[N-1, M-1], [0, M-1]])
    else:
        xv, yv = np.meshgrid(range(N), range(M))
        pts = np.array([xv.flatten(), yv.flatten()]).T
    if H is None: return pts, None

    # else
    t_pts = np.dot(H, real_to_homo(pts).T)
    t_pts = homo_to_real(t_pts.T).astype(int)
    if(targ_shap is None): return pts, t_pts

    #else
    t_pts[:,0] = np.clip(t_pts[:,0], 0, targ_shap[1]-1)
    t_pts[:,1] = np.clip(t_pts[:,1], 0, targ_shap[0]-1)
    return pts, t_pts

def apply_trans_patch(base_img_path, template_img_path, H, suff = '_fnew'):
    ## Read images
    base_img = cv2.imread(base_img_path)
    temp_img = cv2.imread(template_img_path)

    ## Find corners in base that correspond to corners in template
    temp_cpts, trans_temp_cpts = get_pts(temp_img.shape, corners=True, H=H, targ_shap = base_img.shap

    _cpts = real_to_homo(trans_temp_cpts) # homo. representation
    _cent_cpts = np.mean(_cpts, axis = 0)# centroid of four points

    # Find the four lines of quadrilateral
    lines = [np.cross(_cpts[0], _cpts[1]), np.cross(_cpts[1], _cpts[2]), np.cross(_cpts[2], _cpts[3])

    ## Finding points in the base that are present in the quadrilateral
    base_bool = np.zeros(base_img.shape[:-1]).flatten() == 0 # True -> inside the quadrilateral
    base_all_pts, _ = get_pts(base_img.shape) # get all pts
    for line in lines:
        line = line / line[-1]
        sn = int(np.sign(np.dot(_cent_cpts, line)))
        nsn = np.int8(np.sign(np.dot(real_to_homo(base_all_pts), line)))
        base_bool = np.logical_and(base_bool, nsn==sn)
    base_bool = base_bool
    base_bool = np.reshape(base_bool, (base_img.shape[0], base_img.shape[1]))
    row_ids, col_ids = np.nonzero(base_bool)
    des_base_pts = np.array([col_ids, row_ids])

    # Find corresponding points in the template image
    trans_des_base_pts = homo_to_real(np.dot(hinv(H), real_to_homo(des_base_pts.T).T).T).astype(int)

    # Clip the points
```

```
    trans_des_base_pts[:, 0] = np.clip(trans_des_base_pts[:, 0], 0, temp_img.shape[1]-1)
    trans_des_base_pts[:, 1] = np.clip(trans_des_base_pts[:, 1], 0, temp_img.shape[0]-1)

    base_img[des_base_pts[1].tolist(), des_base_pts[0].tolist(), :] = temp_img[trans_des_base_pts[1]

    # Write the resulting image to a file
    fname, ext = tuple(os.path.basename(base_img_path).split('.'))
    write_filepath = os.path.join(os.path.dirname(base_img_path), fname+suff+'.'+ext)
    print write_filepath
    cv2.imwrite(write_filepath, base_img)

def real_to_homo(pts):
    # pts is a 2D numpy array of size _ x 2/3
    # This function converts it into _ x 3/4 by appending 1
    if(pts.ndim == 1):
        return np.append(pts, 1)
    else:
        return np.concatenate((pts, np.ones((pts.shape[0], 1))), axis = 1)

def homo_to_real(pts):
    # pts is a 2D numpy array of size _ x 3/4
    # This function converts it into _ x 2/3 by removing last column
    if(pts.ndim == 1):
        pts = pts / pts[-1]
        return pts[:-1]
    else:
        pts = pts.T
        pts = pts / pts[-1,:]
        return pts[:-1,:].T

def save_mps(event, x, y, flags, param):
    fac, mps = param
    if(event == cv2.EVENT_LBUTTONUP):
        mps.append([int(fac*x), int(fac*y)])
        print(int(fac*x), int(fac*y))

def create_matching_points(img_path, suff = ''):
    npz_path = img_path[:-4]+ suff + '.npz'
    flag = os.path.isfile(npz_path)
    if(not flag):
        img = cv2.imread(img_path)
        fac = max(float(int(img.shape[1]/960)), float(int(img.shape[0]/540)))
        if(fac < 1.0): fac = 1.0
        resz_img = cv2.resize(img, None, fx=1.0/fac, fy=1.0/fac, interpolation = cv2.INTER_CUBIC)
        cv2.namedWindow(img_path)
        mps = []
        cv2.setMouseCallback(img_path, save_mps, param=(fac, mps))
        cv2.imshow(img_path, resz_img)
        cv2.waitKey(0)
        np.savez(npz_path, mps = np.array(mps))
        cv2.destroyAllWindows()
    return np.load(npz_path)
```

```
def nmlz(x):
    assert isinstance(x, np.ndarray), 'x should be a numpy array'
    assert x.ndim > 0 and x.ndim < 3, 'dim of x >0 and <3'
    if(x.ndim == 1 and x[-1]!=0): return x/float(x[-1])
    if(x.ndim == 2 and x[-1,-1]!=0): return x/float(x[-1,-1])
    return x

def rem_transl(H):
    assert isinstance(H, np.ndarray), 'H should be a numpy array'
    assert H.ndim == 2, 'H should be a numpy array of two dim'
    assert H.shape[0] == H.shape[1], 'H should be a square matrix'
    H_clone = np.copy(H)
    H_clone[:-1,-1] = 0
    return H_clone

def hinv(H):
    assert isinstance(H, np.ndarray), 'H should be a numpy array'
    assert H.ndim == 2, 'H should be a numpy array of two dim'
    assert H.shape[0] == H.shape[1], 'H should be a square matrix'
    Hinv = np.linalg.inv(H)
    return Hinv / Hinv[-1,-1]

################ MAIN ###################

import cv2
import numpy as np
import time
from os.path import join, basename, splitext, dirname
import sys
from glob import glob

sys.path.insert(0, r'..\utils')
from helpers import *

def extract_kps(image, ftype = 'sift', sigma = 1.414):
    ###############
    # Description:
    #   Find interest points (keypoints) and descriptors
    # Input:
    #   image: RGB image. 3D ndarray (H x W x 3).
    #   ftype: 'sift' or 'surf'
    #   sigma: scale applied to the image
    # Output:
    #   A tuple (keypoints, features)
    #   keypoints: ndarray (Z x 2). each row has (row_idx, col_idx)
    #   features:  ndarray (Z x desc_size**2)
    #   Z is no. of interest points
    ###############

    ## Assertion
    assert image.ndim == 3, 'img is a 3D ndarray (RGB image: H x W x 3)'

    ## convert the image to grayscale
```

```
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    if(ftype.lower() == 'sift'):
        descriptor = cv2.xfeatures2d.SIFT_create()
        # kps (cv2.KeyPoint object) and features (ndarray).
        (kps, features) = descriptor.detectAndCompute(image, None)
        kps = np.float32([kp.pt for kp in kps])
    elif ftype.lower() == 'surf':
        descriptor = cv2.xfeatures2d.SURF_create()
        # kps (cv2.KeyPoint object) and features (ndarray).
        (kps, features) = descriptor.detectAndCompute(image, None)
        kps = np.float32([kp.pt for kp in kps])
    else:
        raise ValueError('Unknown feature type')

    # kps: ndarray (Z x 2); features: ndarray (Z x 128)
    return (kps, features)

def mean_normalize(M, axis = 0):
    ## First, substract mean and next, normalize the rows/columns.
    # Mean normalize matrix M (_ x k) in rows
    if(axis == 1): M = M.transpose() # (k x _)
    M -= np.mean(M, axis = 0)

    norms = np.linalg.norm(M, axis = 0)
    norms[norms == 0.0] = 1e-10

    M /= norms
    if(axis == 1): M = M.transpose()
    return M

def dist_mat_vec(M, vec, method = 'ncc'):
    # Compute distance between each row of 'M' with 'vec'
    # method: 'ncc', 'dot', 'ssd'
    # M : ndarray ( _ x k); vec: (1 x k)
    # Returns a 1D numpy array of distances.
    if(method.lower() == 'ssd'):
        return np.linalg.norm(M - vec, axis = 1)
    elif(method.lower() == 'ncc'):
        # Mean Normalizing rows of M
        M = mean_normalize(M, axis = 1)
        # Mean normalizing vec
        vec = vec - np.mean(vec)
        vect = vec / np.linalg.norm(vec)
        return np.dot(M, vec)
    elif(method.lower() == 'dot'):
        return np.dot(M, vec)

def dist_mat_mat(M1, M2, method = 'ncc'):
    # M1, M2 --> ndarray (y1 x k) and (y2 x k)
    # Returns y1 x y2 ndarray with the distances.
    # If y1 and y2 are huge, it might run into MemoryError
    D = np.zeros((M1.shape[0], M2.shape[0]))
```

```
    if(method.lower() == 'ncc'):
        M1 = mean_normalize(M1, axis = 1)
        M2 = mean_normalize(M2, axis = 1)
        method = 'dot'
    for idx2 in range(M2.shape[0]):
        D[:, idx2] = dist_mat_vec(M1, M2[idx2, :], method = method)
    return D

def filter_kps(kpA, kpB, featuresA, featuresB, method = 'ncc', thresh = 0.97):
    # Filter the keypoints and the descriptors by thresholding.
    # Returns the matches. List of tuples. (a_idx, b_idx). Point correspondences.
    print len(featuresA), len(featuresB)
    if(method.lower() == 'ncc'):
        featuresA = mean_normalize(featuresA, axis = 1)
        featuresB = mean_normalize(featuresB, axis = 1)
        method = 'dot'

    matches = []
    for idxB in range(featuresB.shape[0]):
        temp = dist_mat_vec(featuresA, featuresB[idxB, :], method = method)
        temp[temp<thresh] = 0.0
        ## Append the ones that pass the threshold
        if(np.max(temp) == 0.0): continue
        else: matches.append((np.argmax(temp), idxB))

    return matches

def draw_matches_one_to_one(imageA, imageB, kpsA, kpsB):
    ########
    # kpsA and kpsB should have same no. of rows.
    # There is one to one correspondence between rows of kpsA and kpsB
    ########
    # initialize the output visualization image
    (hA, wA) = imageA.shape[:2]
    (hB, wB) = imageB.shape[:2]
    vis = np.zeros((max(hA, hB), wA + wB, 3), dtype="uint8")
    vis[0:hA, 0:wA] = imageA
    vis[0:hB, wA:] = imageB

    # loop over the matches
    for ptA, ptB in zip(kpsA, kpsB):
        ptA = (int(ptA[0]), int(ptA[1]))
        ptB = (int(ptB[0]) + wA, int(ptB[1]))
        color = tuple(np.random.randint(0, 255, 3).tolist())
        cv2.line(vis, ptA, ptB, color, 2)

    # return the visualization
    return vis

def draw_matches(imageA, imageB, kpsA, kpsB, matches):
    # initialize the output visualization image
    (hA, wA) = imageA.shape[:2]
    (hB, wB) = imageB.shape[:2]
```

```
    vis = np.zeros((max(hA, hB), wA + wB, 3), dtype="uint8")
    vis[0:hA, 0:wA] = imageA
    vis[0:hB, wA:] = imageB

    # loop over the matches
    for queryIdx, trainIdx in matches:
        # only process the match if the keypoint was successfully
        # matched
        # draw the match
        ptA = (int(kpsA[queryIdx][0]), int(kpsA[queryIdx][1]))
        ptB = (int(kpsB[trainIdx][0]) + wA, int(kpsB[trainIdx][1]))
        color = tuple(np.random.randint(127, 255, 3).tolist())
        cv2.line(vis, ptA, ptB, color, 1)

    # return the visualization
    return vis

def run(image1_path, image2_path, ftype = 'sift', method = 'ncc', \
    thresh = 0.97, sigma = 1.414, write_flag = False):
    img1 = cv2.imread(image1_path)
    img2 = cv2.imread(image2_path)

    kps1, features1 = extract_kps(img1, ftype = ftype, sigma = sigma)
    kps2, features2 = extract_kps(img2, ftype = ftype, sigma = sigma)

    # start = time.time()
    matches = filter_kps(kps1, kps2, features1, features2, method = method, thresh = thresh)
    print 'No. of matches: ', len(matches)
    # print 'Filter Kps: %.02f secs'%(time.time()-start)

    vis = draw_matches(img1, img2, kps1, kps2, matches)

    ## Obtain keypoint matches
    matches = np.array(matches)
    kps1, kps2 = np.array(kps1), np.array(kps2)
    ord_kps1 = kps1[matches[:, 0], :]
    ord_kps2 = kps2[matches[:, 1], :]
    # Format of kp_matches: _ x 4 np.ndarray.
    # Columns 0 and 1 for [x1, y1] of image 1
    # Columns 2 and 3 for [x1, y1] of image 2
    kp_matches = np.append(ord_kps1, ord_kps2, axis = 1)

    delta = 0.5
    while True:
        new_kp_matches, H = ransac(kp_matches, delta = 4, eps = 0.20)
        if H is None: delta = delta * 2
        else: break

    new_kps1 = new_kp_matches[:,:2].tolist()
    new_kps2 = new_kp_matches[:,2:].tolist()

    print 'No. matches: ', len(new_kps1)
```

```
    print 'Performing LM: '
    lmres = LM_Minimizer(new_kp_matches, H)

    new_H = np.squeeze(np.asarray(lmres['parameter_values']))
    new_H = np.append(new_H, np.array([1])).reshape(3, 3)
    new_H = nmlz(new_H)

    # mosaic_two_images(img1, img2, new_H)
    # mosaic_two_images(img2, img1, hinv(new_H))

    vis = draw_matches_one_to_one(img1, img2, new_kps1, new_kps2)

    #####
    if(write_flag):
        fname = splitext(basename(image1_path))[0] + '_' + splitext(basename(image2_path))[0]
        fname = fname + '_' + str(ftype) + '_' + str(int(sigma*1000)) + '_' + str(int(thresh*10000))
        fname_path = join(dirname(image1_path), fname)
        print 'Writing to: ', fname_path
        cv2.imwrite(fname_path, vis)

    return vis, new_H

if __name__ == '__main__':
    base_img_dir = 'pair4'
    img_paths = glob(join(base_img_dir, '*.jpg'))
    if(len(img_paths)%2 == 0): img_paths = img_paths[:-1]

    num_images = len(img_paths)
    mid_id = int(num_images/2)

    ftype = 'sift'
    method = 'ncc'
    thresh = 0.995 # SURF 0.9999
    sigma = 2.00

    H = [None] * (num_images-1)
    V = [None] * (num_images-1)
    for idx in range(num_images-1):
        vis, temp_h = run(img_paths[idx], img_paths[idx+1], ftype = ftype, method=method, thresh = th
        V[idx] = vis
        H[idx] = temp_h

    HM =[None]*num_images
    HM[0] = nmlz(np.dot(np.dot(hinv(H[0]), hinv(H[1])), hinv(H[2])))
    HM[1] = nmlz(np.dot(hinv(H[1]), hinv(H[2])))
    HM[2] = nmlz(hinv(H[2]))
    HM[3] = np.eye(3)
    HM[4] = H[3]
    HM[5] = nmlz(np.dot(H[3], H[4]))
    HM[6] = nmlz(np.dot(np.dot(H[3], H[4]), H[5]))

    # img_in_1 = mosaic_two_images(img_paths[1], img_paths[0], hinv(H[0]))
    # img_in_2 = mosaic_two_images(img_paths[2], img_in_1, hinv(H[1]))
```

```
    # img_in_3 = mosaic_two_images(img_paths[3], img_in_2, hinv(H[2]))

    # img_in_5 = mosaic_two_images(img_paths[5], img_paths[6], H[5])
    # img_in_4 = mosaic_two_images(img_paths[4], img_in_5, H[4])
    # nimg_in_3 = mosaic_two_images(img_in_3, img_in_4, H[3])
    # cv2.imshow('new img_in_3', nimg_in_3)

    # omg = mosaic_two_images(img_paths[mid_id], img_paths[0], HM[0])
    # omg = mosaic_two_images(omg, img_paths[1], HM[1])
    # omg = mosaic_two_images(omg, img_paths[2], HM[2])
    # omg = mosaic_two_images(omg, img_paths[3], HM[3])
    # omg = mosaic_two_images(omg, img_paths[4], HM[4])
    # omg = mosaic_two_images(omg, img_paths[5], HM[5])
    # omg = mosaic_two_images(omg, img_paths[6], HM[6])
    # vis = run(img1_path, img2_path, ftype = ftype, method=method, thresh = thresh, sigma = sigma,

    # cv2.imshow('Visualization', vis)
    cv2.waitKey(0)
```