Homework 5

Fangda Li
ECE 661 - Computer Vision

October 15, 2016

1 Methodology

In this section, the algorithms and steps used in obtaining homography between two overlapped images are described in order.

1.1 Feature Extraction and Matching

To extract interest points and their descriptors from an image pair, the OpenCV implementation of SIFT is used. Then, the SURF descriptors in two images are matched using kNN in OpenCV.

1.2 Random Sample Consensus (RANSAC)

In practice, the correspondences found in the previous section will always contain false matchings, which introduce error to the Least Square Method for estimating the homography. As a result, RANSAC is used to reject outliers and prevent the Least Square Method from false matchings. The procedure of RANSAC is as follows:

1. Randomly pick a minimal set of n matchings ($n = 4$ in our case since we need 8 equations to solve H). Calculate the homography H based on this minimal set that transform range image M_1 to the frame of domain image M_2.

2. Apply the homography H to all the of interest points on range image M_1. If the distance between the coordinate of transformed interest point and the coordinate of its corresponding interest point on domain image M_2 suggested by the matching is within a decision threshold δ, then this matching is considered as an inlier. Otherwise, it's considered as an outlier.

3. Repeat the above two steps for N iterations and choose the iteration with the maximum number of inliers to be our winner. Then, use both the 4 pairs of matching from the minimal set and another 4 inlier pairs to form the homogeneous equations in the Least Square Method.

In order to choose δ, we assume the distance between the true coordinate of matching point from SIFT, $\tilde{x'}$, and the measured coordinate after applying the homography, $x'_{measured}$, to be a Gaussian distribution. As a result, we choose $\delta = 3\sigma$.

Since the number of unique minimal sets can be very large, iterating through them exhaustively is not practical. Instead, assume that we know the probability of a matching being false is ϵ by observation, we want to have N trials such that the probability of at least one of the N minimal sets contains only inliers. More specifically, N is calculated by the following equation:

$$N = \frac{ln(1-p)}{ln(1-(1-\epsilon)^n)}. \tag{1}$$

Moreover, we also want to constrain a valid candidate of any of the N trials to have more than M inliers. For choosing M, the idea here is that we want M to be approximately the number of true matchings, as shown in the following equation:

$$M = (1-\epsilon)n_{total}. \tag{2}$$

1.3 Least Square Method

When we use inhomogeneous equations to solve for the 8 unknowns in H, ideally 8 equations (4 pairs of matching) are enough. However, because of the presence of noise and false matching, we want to include more than 8 equations in our linear system to reduce the effect of noise. However, as a consequence, we won't any solution to this over-determined system, $A\overrightarrow{h} = \overrightarrow{b}$, where A is $m \times n$, h is $n \times 1$ vector and b is $m \times 1$. Notably, we have $m > n$, since this is an over-determined system.

Now, we seek to find $\overrightarrow{h}$ that minimizes $||A\overrightarrow{h} - \overrightarrow{b}||$. Interestingly, even $A\overrightarrow{h}$ resides in R^m, it also resides in a n dimension subspace of R^m. The reason is that since each row of A has only n elements, the maximum dimension of the column space of A, $C(A)$, can only be n. Therefore, the h that minimizes $||A\overrightarrow{h} - \overrightarrow{b}||$ is the projection of b onto R^n, as shown in the following equation,

$$A^T(\overrightarrow{b} - A\overrightarrow{h}) = \overrightarrow{0}, \tag{3}$$

which then gives,

$$\overrightarrow{h} = (A^TA)^{-1}A^T\overrightarrow{b}. \tag{4}$$

Now we have obtained h with 8 matchings using the Least Square Method.

1.4 Dog-Leg Algorithm

Dog-Leg is an optimization algorithm that aims to minimize the cost function using a combination of Gradient Descent (GD) and Gauss-Newton (GN). The inefficiency with GD alone is that the step size of GD can get infinitely close to 0 when the cost approaches the minimum. On the other hand, GN fails when the cost is not already sufficiently close to the minimum. Consequently, the key idea of Dog-Leg is to establish a trusted region of radius r_k around

our current point p_k on the hyperplane. Then, we compute both the steps taken by GD and GN, as shown in Equations 5 and 6.

$$\overrightarrow{\delta}_{P,GD} = \frac{||J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k})||}{||J_{\overrightarrow{f}} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k})||} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k}) \tag{5}$$

$$\overrightarrow{\delta}_{P,GN} = \frac{1}{J_{\overrightarrow{f}} J_{\overrightarrow{f}}^T + \mu_k I} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k}) \tag{6}$$

Now we have three cases to consider for the next point p_{k+1}, as shown in Equation 7.

$$p_{k+1} = p_k + \begin{cases} \overrightarrow{\delta}_{P,GN} & if\ ||\overrightarrow{\delta}_{P,GN} < r_k \\ \overrightarrow{\delta}_{P,GD} + \beta(\overrightarrow{\delta}_{P,GN} - \overrightarrow{\delta}_{P,GD}) & if\ ||\overrightarrow{\delta}_{P,GD}|| < r_k < ||\overrightarrow{\delta}_{P,GN}|| \\ \frac{r_k}{||\overrightarrow{\delta}_{P,GD}||} \overrightarrow{\delta}_{P,GD} & otherwise \end{cases} \tag{7}$$

Note that in the second condition, it first takes a step using GD and then take anther step toward the next point calculated by GN, thus the name "Dog-Leg".

Moreover, the equations for obtaining β and updating r_k are as follows:

$$||\overrightarrow{\delta}_{P,GD} + \beta(\overrightarrow{\delta}_{P,GN} - \overrightarrow{\delta}_{P,GD})||^2 = r_k^2, \tag{8}$$

$$r_{k+1} = \begin{cases} \frac{r_k}{4} & if\ \rho^{DL} \leq \frac{1}{4} \\ r_k & if\ \frac{1}{4} < \rho^{DL} \leq \frac{3}{4} \\ 2r_k & otherwise \end{cases}, \tag{9}$$

$$\rho^{DL} = \frac{C(\overrightarrow{p_k}) - C(\overrightarrow{p_{k+1}})}{2\overrightarrow{\delta_P} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k}) - \overrightarrow{\delta_P} J_{\overrightarrow{f}}^T J_{\overrightarrow{f}} \overrightarrow{\delta_P}}. \tag{10}$$

Solving Equation 8 gives the β value that ensures $\overrightarrow{p_{k+1}}$ falls on the circumference of the trusted region, when $||\overrightarrow{\delta}_{P,GD}|| < r_k < ||\overrightarrow{\delta}_{P,GN}||$. The ratio, ρ^{DL}, positively correlates with the effectiveness of the last step and the adjustment of r_k is made accordingly. When ρ^{DL} becomes less than zero, it means that we have hopped over the minimum and the iterations stop.

1.5 Image Mosaicing

The steps for image mosaicing are:

1. Compute the homographies that transform surrounding images to central images. For images that are not immediately adjacent to the central image, the homography is obtained by chaining successive homographies, as shown in Equation 11:

$$H_{L2,C} = H_{L1,C} H_{L2,L1}, \tag{11}$$

where $L2$, $L1$, C are the left most, second left most and central images, respectively, and $H_{L2,C}$ is the homograph that transforms $L2$ onto the frame of C.

2. After all the homographies are obtained, we then compute the bounding box of the final stitched image. Then we transform each surrounding image to the frame of the central image and paint the transformed image onto the canvas.

3. For the overlapped region between two images, we take the average of the two pixel values.

2 Results

The following parameters are used in this experiment:

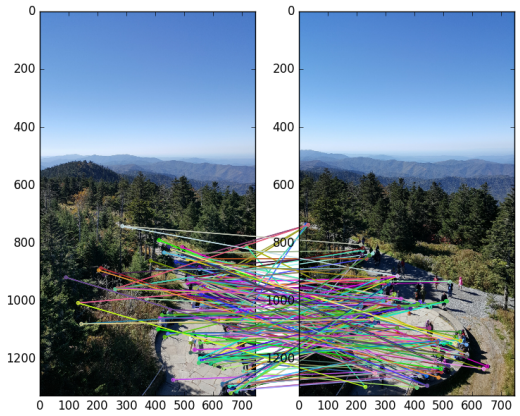
Table 1: Parameters

SIFT	
$nfeatures$	1000
RANSAC	
ϵ	0.9
δ	10

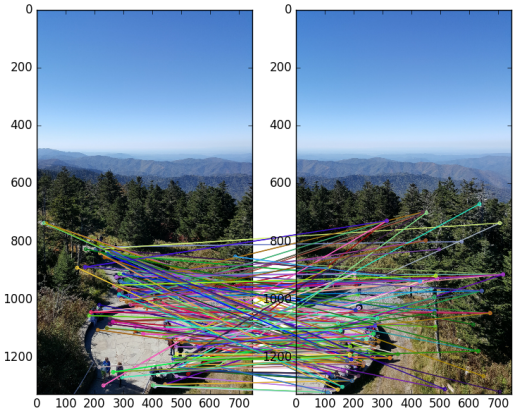
Unfortunately I ran out of time for DogLeg... :(. But the final result still looks decent without it.

(a) L2 (b) L1 (c) C (d) R1 (e) R2

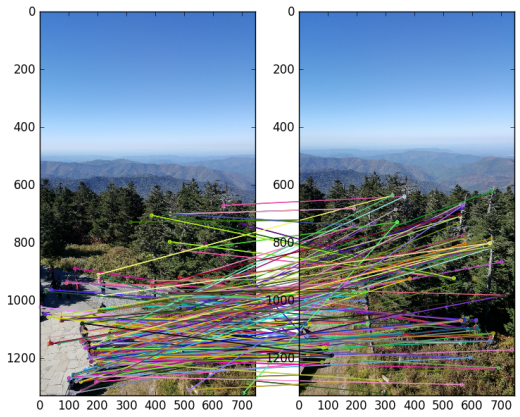
Figure 1: Input images



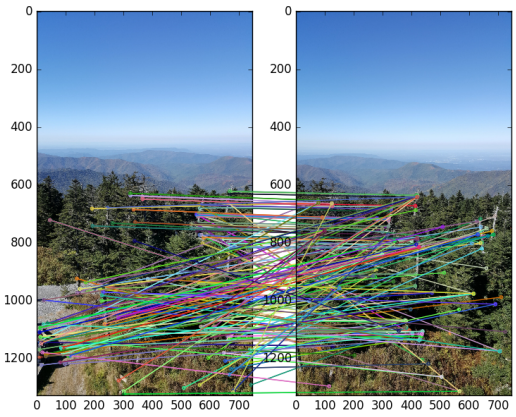
(a) L2 and L1



(b) L1 and C

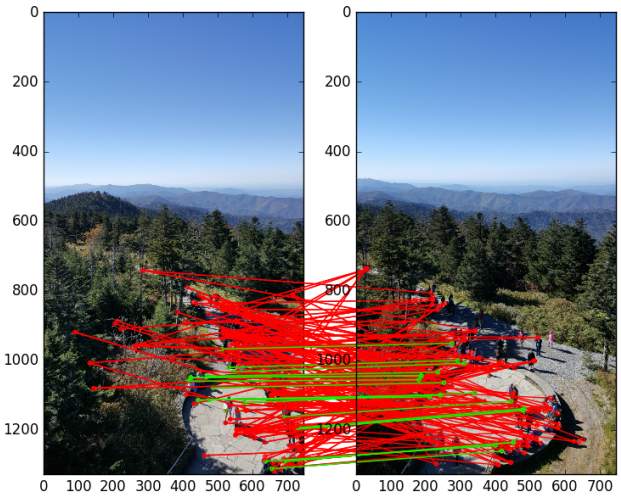


(c) C and R1

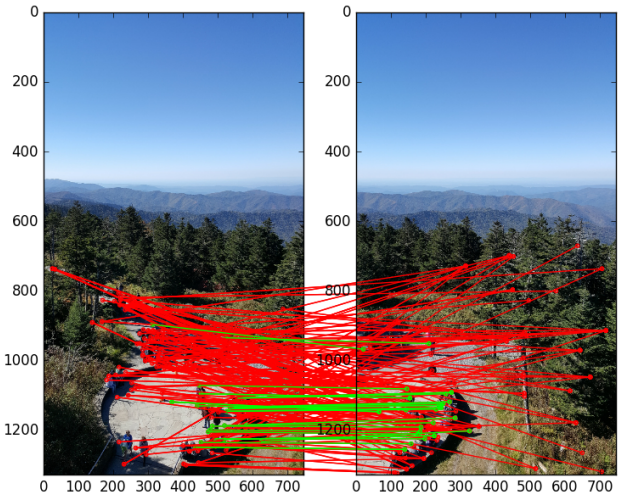


(d) R1 and R2

Figure 2: Matchings between adjacent input images. A significant portion of the raw matchings are false, as shown in Figure 2a. This is because the images contain a lot of similar regions such as trees and people's movement in between shots. Nonetheless, the effectiveness of RANSAC is demonstrated by finding the right homograph with majority of the matchings being false.



(a) L2 and L1



(b) L1 and C

Figure 3: Inliers and outliers. Outliers significantly outnumber inliers. As a result, we chose $\epsilon = 0.9$ for RANSAC to work.

Figure 4: Final output image

3 Source Code

3.1 ransac.py

```
#!/usr/bin/python
import numpy as np
import cv2

def get_N(epsilon, n, p):
        '''
                Calculate the number of trials, N.
                @epsilon: double of probability of outlier
                @n: int of size of minimal set
                @p: double of probability that at least one out of N trials only conta
                @return: int of number of trials
        '''
        return int( np.log(1 - p) / np.log( 1 - (1 - epsilon)**n ) )

def get_M(epsilon, n_total):
        '''
                Calculate minimum number of inliers, M.
                @epsilon: double of probability of outlier
                @n_total: int of total number of samples
                @return: int of minimum number of inliers
        '''
        return int( (1 - epsilon) * n_total )

def get_random_minimal_set(n, pts1, pts2):
        '''
                Randomly pick and copy n matchings.
                @n: int of size of minimal set
                @pts1,pts2: list of points of all the matchings
                @return: tuple of lists of n matchings
        '''
        perm = np.random.permutation(len(pts1))
        s1 = []
        s2 = []
        for i in range(n):
                s1.append( pts1[ perm[i] ] )
                s2.append( pts2[ perm[i] ] )
        return s1, s2

def get_homography(pts1, pts2):
        '''
                Calculate the homography matrix H based on matchings.
```

```
                @matchings: list of matchings
                @return: np.ndarray of H
        '''
        pts1 = np.float32(pts1)
        pts2 = np.float32(pts2)
        return cv2.getPerspectiveTransform(pts1, pts2)

def apply_homography_on_points(points, H):
        '''
                Apply H on all the given points.
                @points: list of points
                @H: np.ndarray of H
                @return: list of points after transformation
        '''
        points = np.array( [np.float32(points)] )
        return np.squeeze( cv2.perspectiveTransform(points, H) )

def get_inliers(trans1, pts1, pts2, delta):
        '''
                Get number of inliers based on Euclidean distance and delta
                @trans1,pts1,pts2: list of points
                @delta: double of threshold
                @return: int of number of inliers
        '''
        count = 0
        in1 = []
        in2 = []
        for i in range(len(trans1)):
                pt1, pt2 = trans1[i], pts2[i]
                dist = np.sqrt( (pt1[0] - pt2[0])**2 + (pt1[1] - pt2[1])**2 )
                if dist < delta:
                        count += 1
                        in1.append( pts1[i] )
                        in2.append( pts2[i] )
        return count, in1, in2

def apply_ransac_on_matchings(pts1, pts2, epsilon, delta):
        '''
                Use RANSAC to find best matchings.
        '''
        N = get_N(epsilon, 4, 0.99)
        M = get_M(epsilon, len(pts1))
        print "M =", M, "; N =", N
        for i in range(N):
                ms1, ms2 = get_random_minimal_set(4, pts1, pts2)
```

```
                H = get_homography(ms1, ms2)
                trans1 = apply_homography_on_points(pts1, H)
                num_inliers, in1, in2 = get_inliers(trans1, pts1, pts2, delta)
                if num_inliers > M:
                        print "Iteration", i
                        print "Number of inliers =", num_inliers
                        return in1, in2
        print "ERROR: RANSAC has failed to find a valid minimal set..."
        return [], []


def main():
        pass

if __name__ == '__main__':
        main()
```

3.2 llsm.py

```
#!/usr/bin/python
import numpy as np
import cv2

def get_inhomo_system(pts1, pts2):
        '''
                Construct A and b of the inhomogeneous equation system.
                @pts1,pts2: list of matching points
                @return: tuple of A and b
        '''
        n = len(pts1)
        A = np.zeros((2*n,8))
        b = np.zeros((2*n,1))
        for i in range(n):
                x,y,w = pts1[i][0], pts1[i][1], 1.
                xp,yp,wp = pts2[i][0], pts2[i][1], 1.
                A[2*i] = [0, 0, 0, -wp*x, -wp*y, -wp*w, yp*x, yp*y]
                A[2*i+1] = [wp*x, wp*y, wp*w, 0, 0, 0, -xp*x, -xp*y]
                b[2*i] = -yp*w
                b[2*i+1] = xp*w
        return A, b

def get_llsm_homograhpy_from_points(pts1, pts2):
        '''
                Obtain the homography using all the inliers from ransac.
                @pts1,pts2: list of matching points
```

```
            @return: np.ndarray of H
        '''
        A, b = get_inhomo_system(pts1, pts2)
        # h is pseudo-inverse multiplied by b
        h = np.dot( np.dot( np.linalg.inv( np.dot(A.transpose(), A) ), A.transpose() ),
        h = np.append(h, 1.)
        h = h.reshape((3,3))
        return h
```

3.3 main.py

```
#!/usr/bin/python
import numpy as np
import cv2
from ransac import apply_ransac_on_matchings
from llsm import get_llsm_homograhpy_from_points
from matplotlib import pyplot as plt
from matplotlib.patches import Circle, ConnectionPatch

def resize_image_by_ratio(image, ratio):
        '''
            Resize an image by a given ratio, used for faster debug.
        '''
        return cv2.resize(image, (int(image.shape[1]*ratio),int(image.shape[0]*ratio)))

def get_bounding_box_after_transformation(image, H):
        '''
            Given an image and the transformation matrix to be applied,
            calculate the bounding box of the image after transformation.
            @image: image to transform
            @H: transformation matrix to be applied
            @return: (h, w, oy, ox)
        '''
        h, w = image.shape[0], image.shape[1]
        corners_1 = [(0,0), (w,0), (0,h), (w,h)]
        corners_2_x = []
        corners_2_y = []
        H = np.matrix(H)
        for corner in corners_1:
            (x,y) = corner
            p_1 = np.array([[x,y,1]])
            (x_2, y_2, z_2) = H * p_1.T
            corners_2_x.append( int(x_2 / z_2) )
            corners_2_y.append( int(y_2 / z_2) )
        return max(corners_2_y)-min(corners_2_y)+1, max(corners_2_x)-min(corners_2_x)+1,
```

```
                        min(corners_2_y), min(corners_2_x)

def get_pixel_by_nearest_neighbor(image, row_f, col_f):
        '''
                Get the pixel value based on float row and column numbers.
                @image: image to be find pixels in
                @row_f, col_f: float row and column numbers
                @return: pixel value from image
        '''
        row = int(round(row_f))
        col = int(round(col_f))
        return image[row][col]

def apply_transformation_on_image(image, H):
        '''
                Given a transformation matrix, apply it to the input image to obtain a
                @image: np.ndarray of input image
                @H: the tranformation matrix to be applied
                @return: np.ndarray of the transformed image
        '''
        # First determine the size of the transformed image
        h, w, oy, ox = get_bounding_box_after_transformation(image, H)
        print "New image size:", (h, w)
        print "Offsets:", oy, ox
        try:
                trans_img = np.ndarray( (h, w, image.shape[2]) )
        except IndexError:
                trans_img = np.ndarray( (h, w, 1) )
        H = np.matrix(H)
        H_inv = H.I
        for y in range(trans_img.shape[0]):
                for x in range(trans_img.shape[1]):
                        p_2 = np.array([[x+ox,y+oy,1]])
                        (x_1, y_1, z_1) = H_inv * p_2.T
                        x_1 = x_1 / z_1
                        y_1 = y_1 / z_1
                        if 0 <= y_1 < image.shape[0]-1 and 0 <= x_1 < image.shape[1]-1:
                                trans_img[y][x] = get_pixel_by_nearest_neighbor(image, y
                        else:
                                trans_img[y][x] = tuple(np.zeros(trans_img.shape[2]))
        return trans_img

def get_canvas(cen_img, sur_imgs, Hs):
        '''
                Allocate storage for the final stitched image.
```

```
        '''
        (h, w, c) = cen_img.shape
        minx, miny, maxx, maxy = 0, 0, w, h
        for i in range(len(sur_imgs)):
                h, w, oy, ox = get_bounding_box_after_transformation(sur_imgs[i], Hs[i])
                minx, miny, maxx, maxy = min(minx, ox), min(miny, oy), \
                                                        max(maxx, ox+w), max(max
        return np.zeros( (maxy-miny, maxx-minx, c) ), minx, miny

def get_sift_kp_des(image, nfeatures=0):
        '''
                Extract SIFT key points and descriptors from image.
                @image: np.ndarray of input image, double type, gray scale
                @return: tuple of key points and corresponding descriptors
        '''
        sift = cv2.xfeatures2d.SIFT_create(nfeatures=nfeatures)
        kps, descs = sift.detectAndCompute(image, None)
        print("# kps: {}, descriptors: {}".format(len(kps), descs.shape))
        return kps, descs

def get_matchings(kp1, des1, kp2, des2):
        '''
                Get matchings using SIFT and return non-openCV points.
        '''
        bf = cv2.BFMatcher()
        matches = bf.match(des1, des2)
        pts1, pts2 = [], []
        for m in matches:
                pt1 = kp1[m.queryIdx].pt
                pt2 = kp2[m.trainIdx].pt
                pts1.append(pt1)
                pts2.append(pt2)
        return pts1, pts2

def paint_image_on_canvas(canvas, ori_img, warped_img, H, canvas_ox, canvas_oy):
        '''
                Copy image onto canvas with blending.
        '''
        h,w,oy,ox = get_bounding_box_after_transformation(ori_img, H)
        for r in range(oy-canvas_oy , oy-canvas_oy+h):
                for c in range(ox-canvas_ox , ox-canvas_ox+w):
                        if np.allclose(canvas[r,c], np.zeros(3)):
                                canvas[r,c] = warped_img[r-(oy-canvas_oy), c-(ox-canvas_
                        elif np.allclose(warped_img[r-(oy-canvas_oy), c-(ox-canvas_ox)],
                                canvas[r,c] = canvas[r,c]
```

```
                        else:
                                canvas[r,c] = (canvas[r,c] + warped_img[r-(oy-canvas_oy)

def smoky_test():
        fpath1 = 'images/1.jpg'
        fpath2 = 'images/2.jpg'
        fpath3 = 'images/3.jpg'
        fpath4 = 'images/4.jpg'
        fpath5 = 'images/5.jpg'
        resize_ratio = 0.5
        nfeatures = 200
        epsilon = 0.9
        delta = 10
        color1 = cv2.imread(fpath1)
        color1 = resize_image_by_ratio(color1, resize_ratio)
        gray1 = cv2.cvtColor(color1, cv2.COLOR_BGR2GRAY)
        color2 = cv2.imread(fpath2)
        color2 = resize_image_by_ratio(color2, resize_ratio)
        gray2 = cv2.cvtColor(color2, cv2.COLOR_BGR2GRAY)
        color3 = cv2.imread(fpath3)
        color3 = resize_image_by_ratio(color3, resize_ratio)
        gray3 = cv2.cvtColor(color3, cv2.COLOR_BGR2GRAY)
        color4 = cv2.imread(fpath4)
        color4 = resize_image_by_ratio(color4, resize_ratio)
        gray4 = cv2.cvtColor(color4, cv2.COLOR_BGR2GRAY)
        color5 = cv2.imread(fpath5)
        color5 = resize_image_by_ratio(color5, resize_ratio)
        gray5 = cv2.cvtColor(color5, cv2.COLOR_BGR2GRAY)
        kp1, des1 = get_sift_kp_des(gray1, nfeatures=nfeatures)
        kp2, des2 = get_sift_kp_des(gray2, nfeatures=nfeatures)
        kp3, des3 = get_sift_kp_des(gray3, nfeatures=nfeatures)
        kp4, des4 = get_sift_kp_des(gray4, nfeatures=nfeatures)
        kp5, des5 = get_sift_kp_des(gray5, nfeatures=nfeatures)
        pts12, pts21 = get_matchings(kp1, des1, kp2, des2)
        pts23, pts32 = get_matchings(kp2, des2, kp3, des3)
        pts34, pts43 = get_matchings(kp3, des3, kp4, des4)
        pts45, pts54 = get_matchings(kp4, des4, kp5, des5)
        in12, in21 = apply_ransac_on_matchings(pts12, pts21, epsilon, delta)
        in23, in32 = apply_ransac_on_matchings(pts23, pts32, epsilon, delta)
        in34, in43 = apply_ransac_on_matchings(pts34, pts43, epsilon, delta)
        in45, in54 = apply_ransac_on_matchings(pts45, pts54, epsilon, delta)
        H12 = get_llsm_homograhpy_from_points(in12, in21)
        H23 = get_llsm_homograhpy_from_points(in23, in32)
        H43 = get_llsm_homograhpy_from_points(in43, in34)
        H54 = get_llsm_homograhpy_from_points(in54, in45)
```

```
H13 = np.dot(H12, H23)
H53 = np.dot(H54, H43)
cen_img = color3
sur_imgs = [color1, color2, color4, color5]
Hs = [H13, H23, H43, H53]
canvas, canvas_ox, canvas_oy = get_canvas(cen_img, sur_imgs, Hs)
print "Canvas:", canvas.shape, canvas_ox, canvas_oy
warped13 = apply_transformation_on_image(color1, H13)
warped53 = apply_transformation_on_image(color5, H53)
warped23 = apply_transformation_on_image(color2, H23)
warped43 = apply_transformation_on_image(color4, H43)
h,w = color3.shape[0], color3.shape[1]
canvas[ 0-canvas_oy : 0-canvas_oy+h , 0-canvas_ox : 0-canvas_ox+w ] = color3
h,w,oy,ox = get_bounding_box_after_transformation(color1, H13)
paint_image_on_canvas(canvas, color1, warped13, H13, canvas_ox, canvas_oy)
h,w,oy,ox = get_bounding_box_after_transformation(color5, H53)
paint_image_on_canvas(canvas, color5, warped53, H53, canvas_ox, canvas_oy)
h,w,oy,ox = get_bounding_box_after_transformation(color2, H23)
paint_image_on_canvas(canvas, color2, warped23, H23, canvas_ox, canvas_oy)
h,w,oy,ox = get_bounding_box_after_transformation(color4, H43)
paint_image_on_canvas(canvas, color4, warped43, H43, canvas_ox, canvas_oy)
plt.imshow(cv2.cvtColor(canvas.astype(np.uint8), cv2.COLOR_BGR2RGB), cmap='jet')

# Plot image and mark the corners
# fig, axes = plt.subplots(1,2)
# axes[0].set_aspect('equal')
# axes[0].imshow(cv2.cvtColor(color2, cv2.COLOR_BGR2RGB), cmap='jet')
# axes[1].set_aspect('equal')
# axes[1].imshow(cv2.cvtColor(color3, cv2.COLOR_BGR2RGB), cmap='jet')
# for i in range(len(pts23)):
#         # color = np.random.rand(3,1)
#         color = (1., 0, 0)
#         pt1 = pts23[i]
#         pt2 = pts32[i]
#         axes[0].add_patch( Circle(pt1, 5, fill=False, color=color, clip_on=F
#         axes[1].add_patch( Circle(pt2, 5, fill=False, color=color, clip_on=F
#         # Draw lines for matching pairs
#         line1 = ConnectionPatch(xyA=pt1, xyB=pt2, coordsA='data', coordsB='d
#         line2 = ConnectionPatch(xyA=pt2, xyB=pt1, coordsA='data', coordsB='d
#         axes[0].add_patch(line1)
#         axes[1].add_patch(line2)
# for i in range(len(in23)):
#         color = (0, 1., 0)
#         pt1 = in23[i]
#         pt2 = in32[i]
```

```
        #         axes[0].add_patch( Circle(pt1, 5, fill=False, color=color, clip_on=F
        #         axes[1].add_patch( Circle(pt2, 5, fill=False, color=color, clip_on=F
        #         # Draw lines for matching pairs
        #         line1 = ConnectionPatch(xyA=pt1, xyB=pt2, coordsA='data', coordsB='d
        #         line2 = ConnectionPatch(xyA=pt2, xyB=pt1, coordsA='data', coordsB='d
        #         axes[0].add_patch(line1)
        #         axes[1].add_patch(line2)
        # plt.show()

def main():
        smoky_test()

if __name__ == '__main__':
        main()
```