ECE 661 (Fall 2016) - Computer Vision - HW 5

Debasmit Das

October 13, 2016

1 Finding Corresponding points based on SIFT between 2 Images

The whole process of extracting SIFT features and matching is as follows -

- Extract SIFT feature descriptors in first image.
- Extract SIFT feature descriptors in second image.
- Create a table that computes the norm between all the SIFT feature descriptors of 2 images.
- For a particular feature in Image 1 we find that feature in Image 2 with minimum norm provided it passes a threshold given as a ratio of the top 2 minimum norms.
- Repeat the above step for all the feature points.

However, the problem with this matching is that there maybe a lot of false correspondences due to the presence of outliers.

2 RANSAC (Random Sample Consensus)

If we have several corresponding points between 2 images we can estimate the homography between the 2 images. However, there might be false correspondences (outliers) which can affect the incorrect estimation of homography. So, we have to remove these outliers using RANSAC before they can be used to estimate the homography. So the steps are as follows -

- Set Parameters for RANSAC such as N, p, M, n, δ and ϵ before we run the RANSAC. Here is a description of the parameters -

  N : The number of trials.
  p : Probability that at least one of the trials will be free of outliers
  M : Minimum value of the acceptable size of the inlier set.
  n : No. of correspondences that are chosen at each trial.
  ϵ : Probability that a correspondence is an outlier.
  δ : Distance threshold to decide if some corresponding points are inliers. $M = n_{tot}(1-\epsilon)$
  (n_{tot} : the total number of correspondences obtained by SIFT matching between 2 images)
  $N = \frac{ln(1-p)}{ln(1-(1-\epsilon)^n)}$
- Select n correspondences randomly from all correspondences obtained by SIFT matching.

- Calculate homography H between 2 images using linear least squares method based on n correspondences. The homography H is computed using homogeneous linear least square method as follows.

  Let (X,X') be the correspondence between two images such that $X' = HX$. We carry out a cross product such that $X' \times HX = 0$ to get an equation of the form $Ah = 0$ ($A : 2n \times 9$ matrix, $h : 9 \times 1$ vector, containing the elements of the H matrix.). To solve the equation, we use the constraint $||h|| = 1$ to prevent $h = 0$. Then the solution to h is the eigenvector corresponding to the smallest eigenvalue for $A^T A$.

- We compute the distances between the true location (HX) and measured location (X') of all correspondences. If the distance is smaller than the distance threshold δ, that correspondence counts into the inlier set.

- We do the trials N number of times. We keep the homography H which has the most number of inliers.

3 Homography refinement with DogLeg

To implement this method we need to know step size for Gradient Descent and Gaussian Newton. They are defined in the following section. p_k is the vector of homography parameters at the k_{th} iteration. Now we have δ for Gauss Newton and Gradient Descent as follows -

$$\delta_{p,GD} = \frac{||J_f^T \epsilon(p_k)||}{||J_f J_f^T \epsilon(p_k)||} J_f^T \epsilon(p_k), \quad \delta_{p,GN} = (J_f^T J_f + u_k I)^{-1} J_f^T \epsilon(p_k) \tag{1}$$

Here, J_f is the Jacobian matrix of f w.r.t. p, $\epsilon(p_k) = \sum ||X_{phys}^{'} - f(p_k)||^2$, u_k is the damping parameter, $X_{phys}^{'}$ is a point in the physical plane. For applying this method we need to note down what p_k and $f(p_k)$ are.

$$p = [h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}, h_{33}]^T \tag{2}$$

h_{ij} are the parameters of the homography matrix.

$$f(p_k) = \begin{bmatrix} \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \\ \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \end{bmatrix} \tag{3}$$

p_k is updated as follows-

$$p_{k+1} = p_k + \delta_k \tag{4}$$

$$\delta_k = \begin{cases} \delta_{p,GN}, & \text{if } ||\delta_{p,GN}|| < r_k \\ \delta_{p,GD} + \beta(\delta_{p,GN} - \delta_{p,GD}), & \text{if } ||\delta_{p,GD}|| < r_k < ||\delta_{p,GN}|| \\ \frac{r_k}{||\delta_{p,GD}||}\delta_{p,GD}, & \text{otherwise} \end{cases} \tag{5}$$

β should be such that $||\delta_{p,GD} + \beta(\delta_{p,GN} - \delta_{p,GD})||^2 = r_k^2$ and following the condition $||\delta_{p,GD}|| < r_k < ||\delta_{p,GN}||$
r_k is also updated as follows -

$$r_{k+1} = \begin{cases} \frac{r_k}{4}, & \text{if } \rho_{dl} < \frac{1}{4} \\ r_k, & \text{if } \frac{1}{4} < \rho_{dl} \leq \frac{3}{4} \\ 2r_k, & \text{otherwise} \end{cases} \tag{6}$$

where $\rho_{dl} = \frac{\epsilon(p_k)^T \epsilon(p_k) - \epsilon(p_{k+1})^T \epsilon(p_{k+1})}{2\delta_p^T J_f^T \epsilon(p_k) - \delta_k^T J_f^T J_f \delta_k}$ p_k is updated until $\rho_{dl} < 0$ The initial value of p_k is obtained after the SIFT feature correspondence is carried out and RANSAC applied to remove the outliers.

4 Image Mosaicing

For our experiment, we have 5 different images of a scene taken in different angles. Let the left most image be indexed as 1, the right most image be indexed as 5 and the center image indexed as 3. To make a panaroma, we compute homography H between two successive images i.e. (Image 1, Image 2), (Image 2, Image 3), (Image 4, Image 3), (Image 5, Image 4). Let the corresponding homographies be H_{12}, H_{23}, H_{43}, and H_{54}. To combine all images together on center Image 3, we need to know homography, H_{13}, which map Image 1 to Image 3 and the homography H_{53} which maps Image 5 to Image 3. These homographies are as follows - $H_{13} = H_{12}H_{23}$, $H_{53} = H_{54}H_{43}$ Thus all images can be mapped to the center image using Homographies (H_{13}, H_{23}, H_{43}, H_{53}).

5 Experimental Results

The parameters for the experiments are as follows -

No. of Octave Levels (SIFT) = 4
$\sigma = 4$
NCC Local Threshold=0.66
NCC Global Threshold=0.96
No. of features=400
$\epsilon = 0.4$
$\delta = 3$
No. of points per trial (n)=6
$p = 0.999$

The images are as follows -

5.1 Input Images

Figure 1: First Image

Figure 2: Second Image

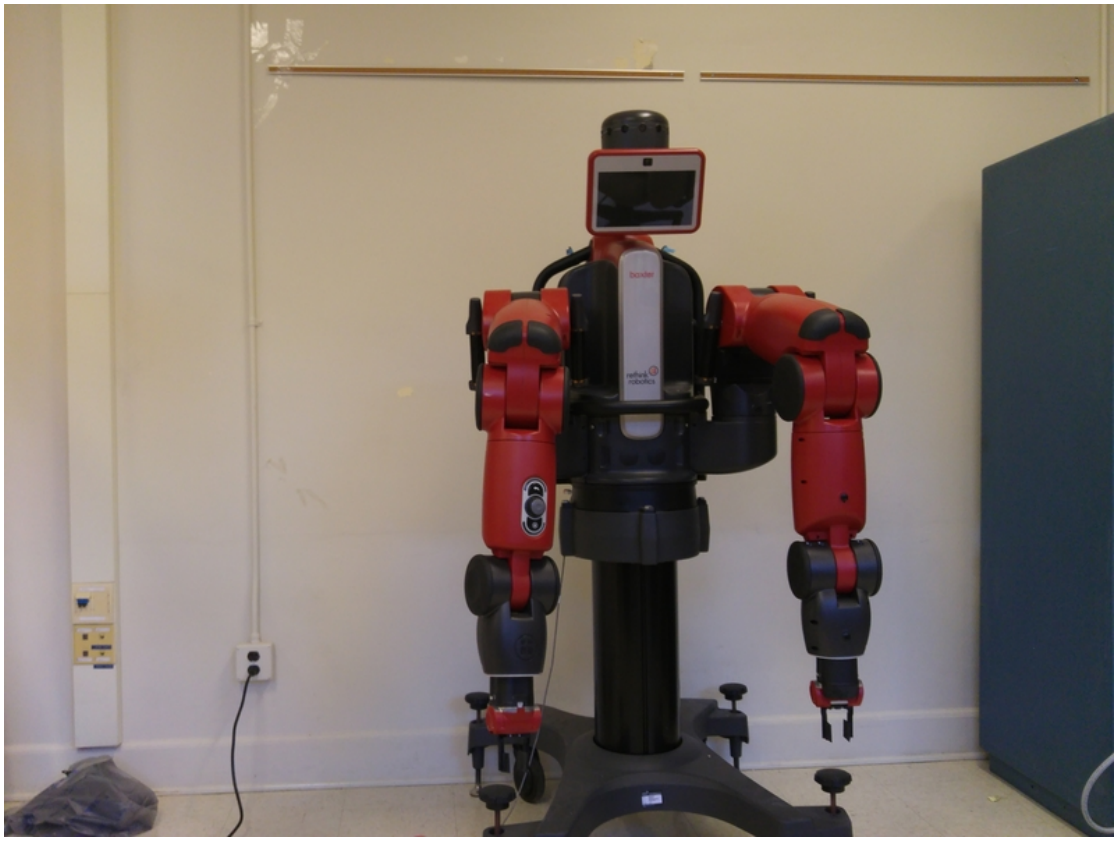

Figure 3: Third Image

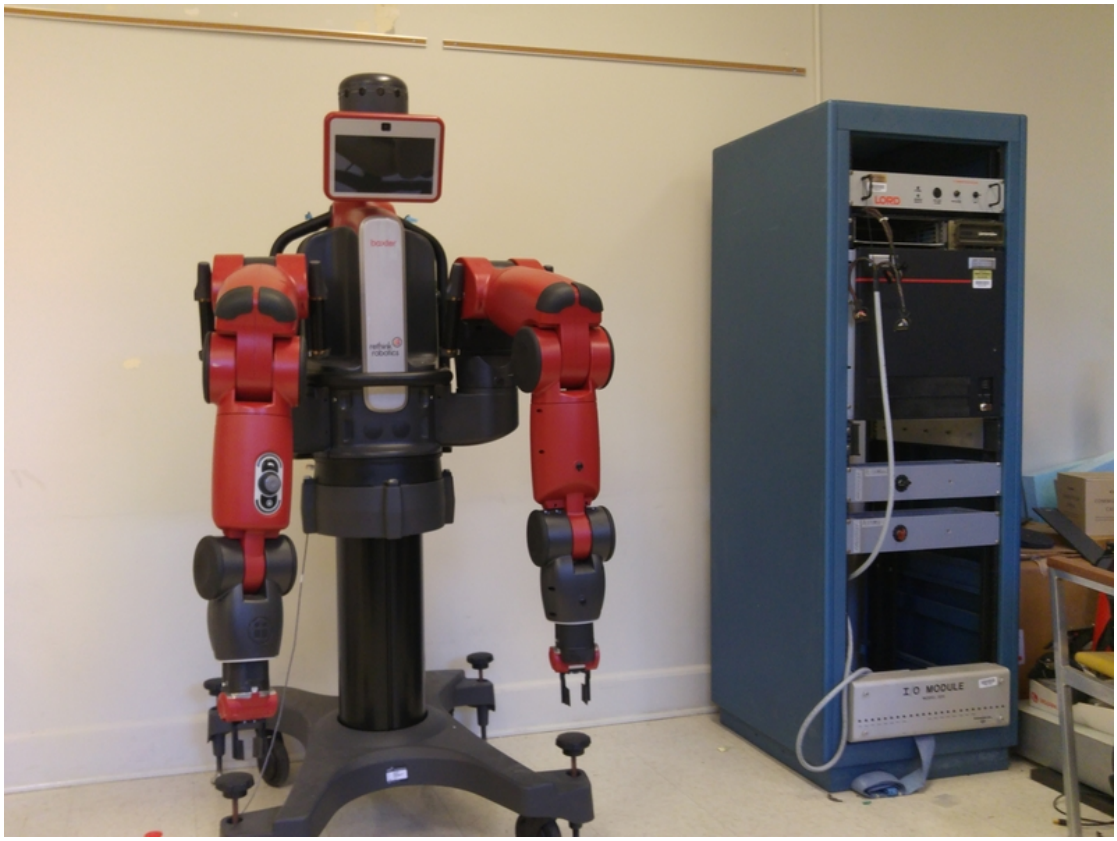

Figure 4: Fourth Image

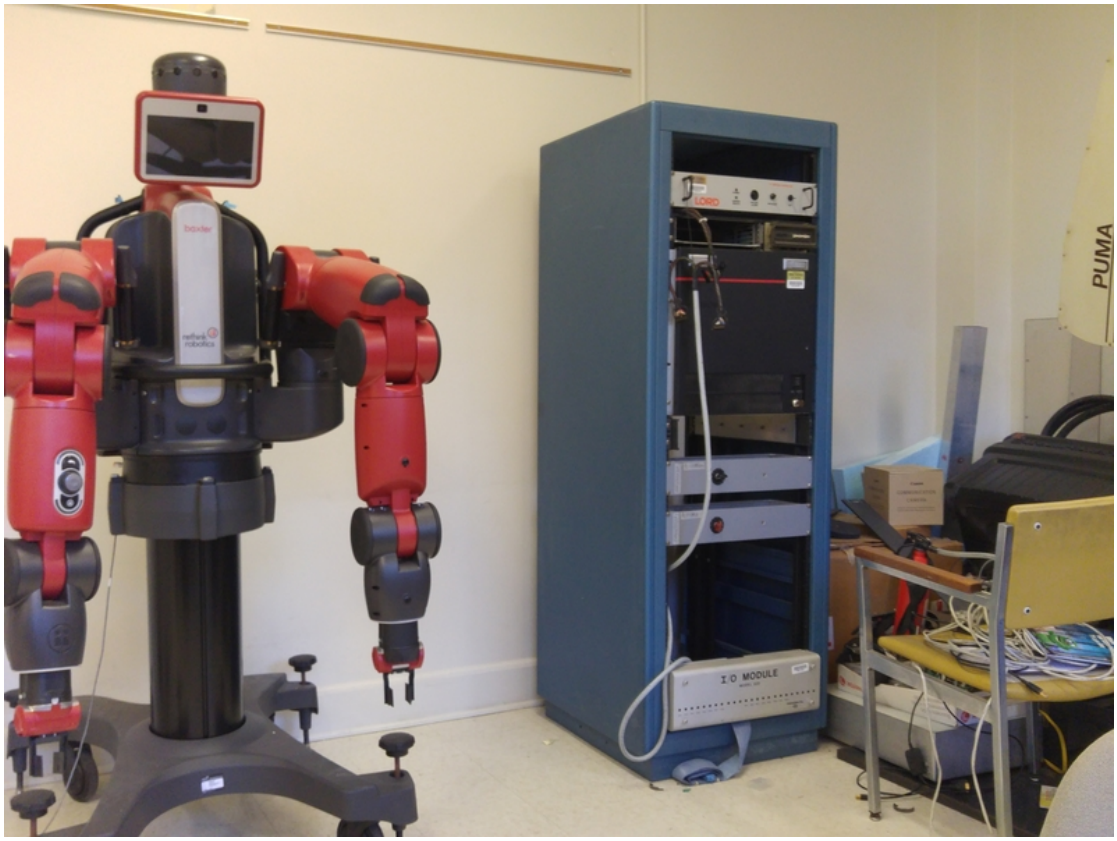

Figure 5: Fifth Image

5.2 Output after RANSAC

Inliers are marked with green and outliers are marked with red

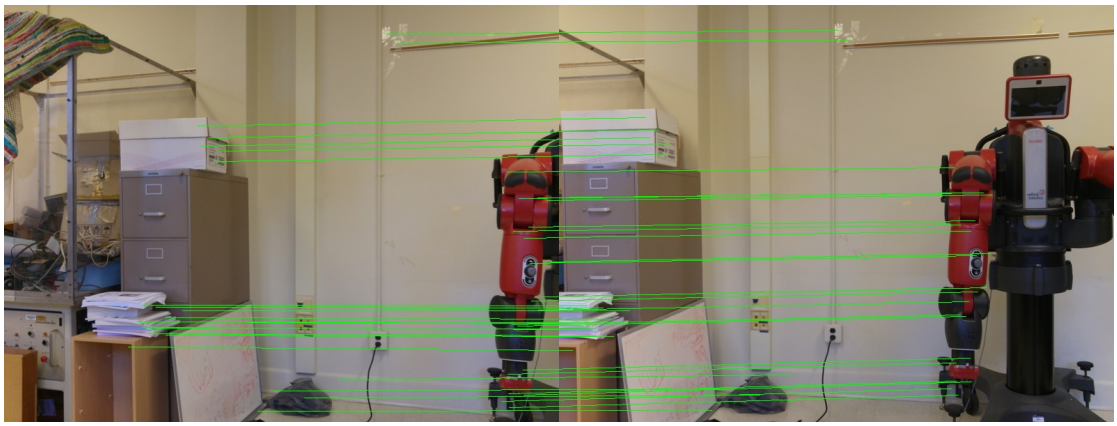

Figure 6: Correspondences between 1st and 2nd

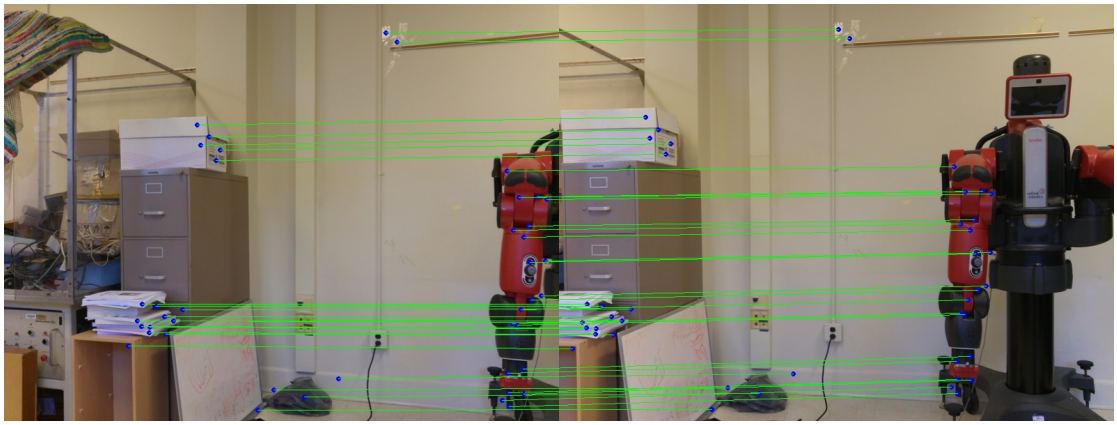

Figure 7: Correspondences (inliers and outliers) between 1st and 2nd

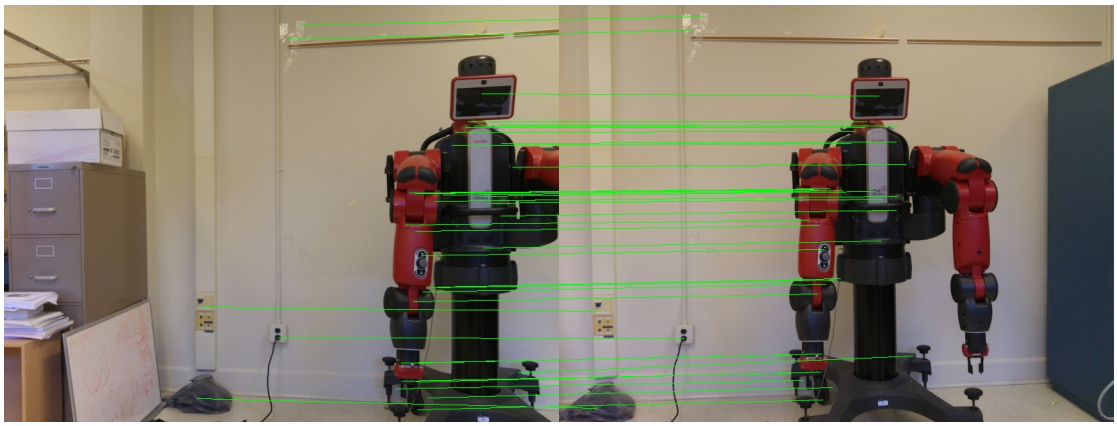

Figure 8: Correspondences between 2nd and 3rd

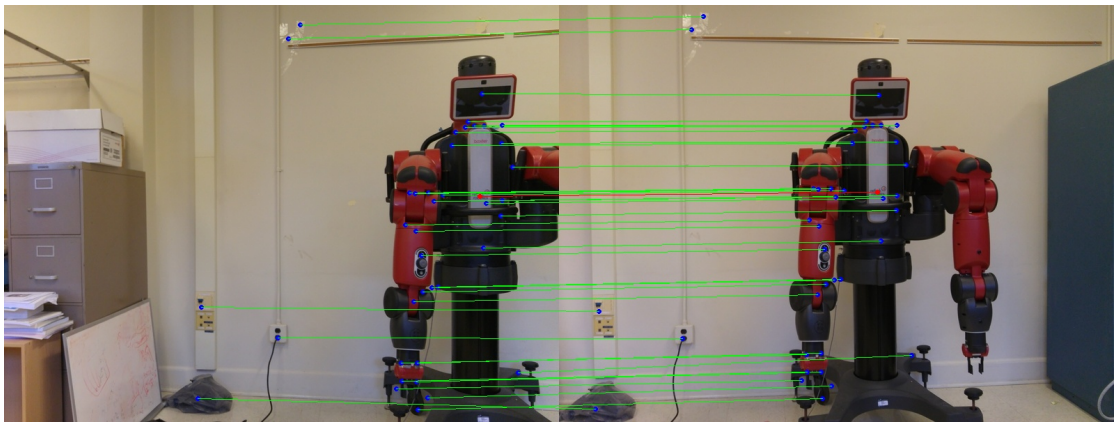

Figure 9: Correspondences (inliers and outliers) between 2nd and 3rd

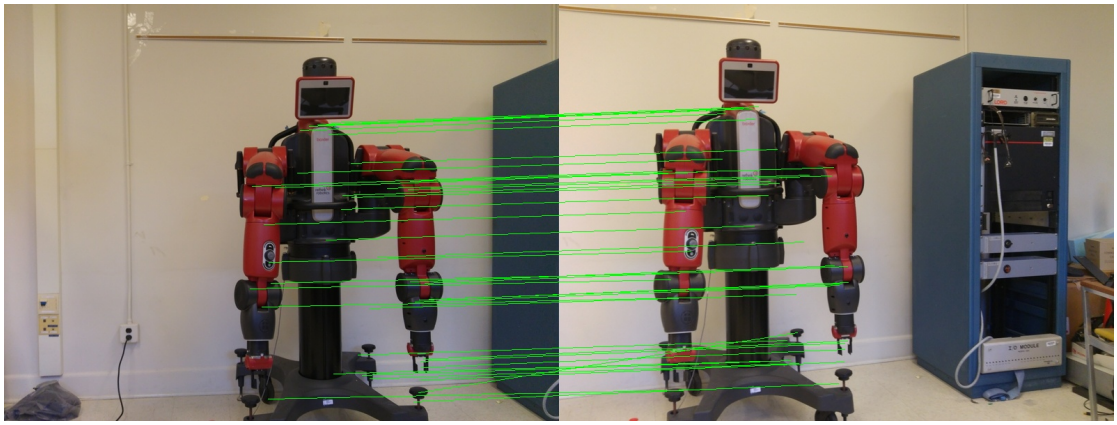

Figure 10: Correspondences between 3rd and 4th

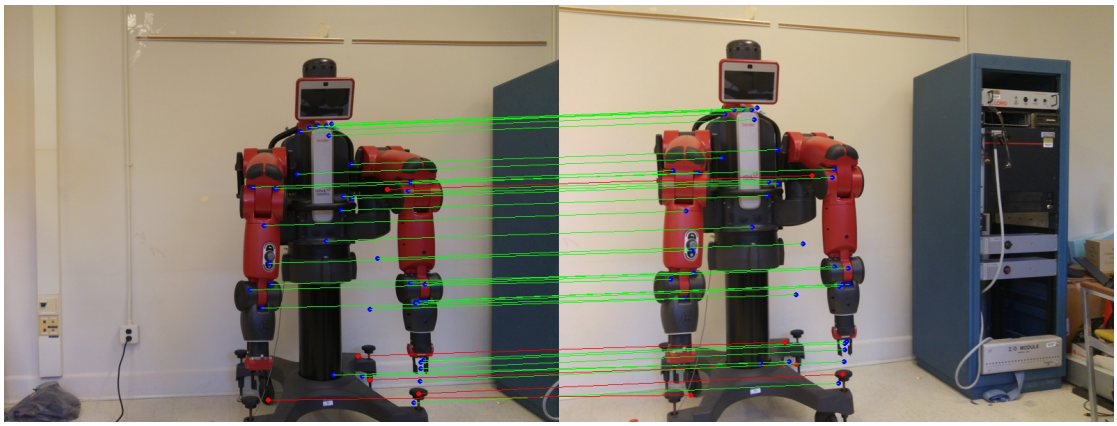

Figure 11: Correspondences (inliers and outliers) between 3rd and 4th

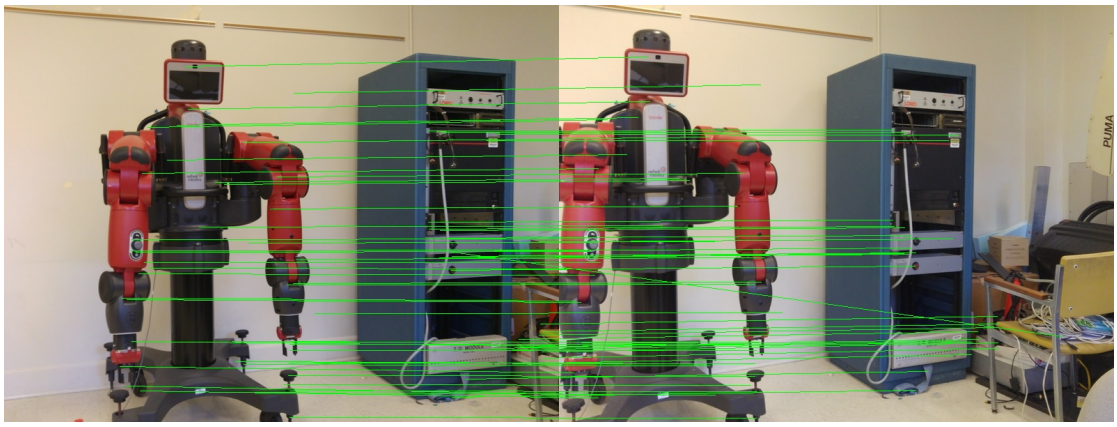

Figure 12: Correspondences between 4th and 5th

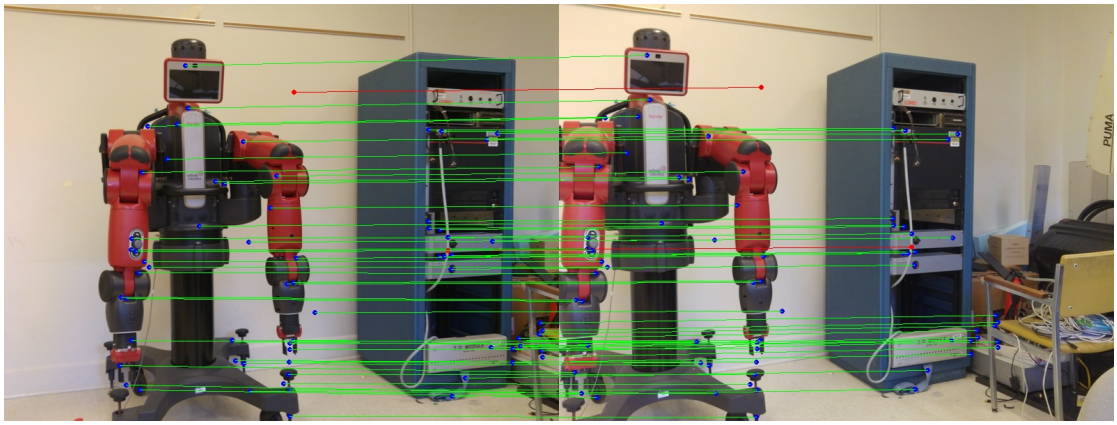

Figure 13: Correspondences (inliers and outliers) between 4th and 5th

5.3 Image Mosaicing

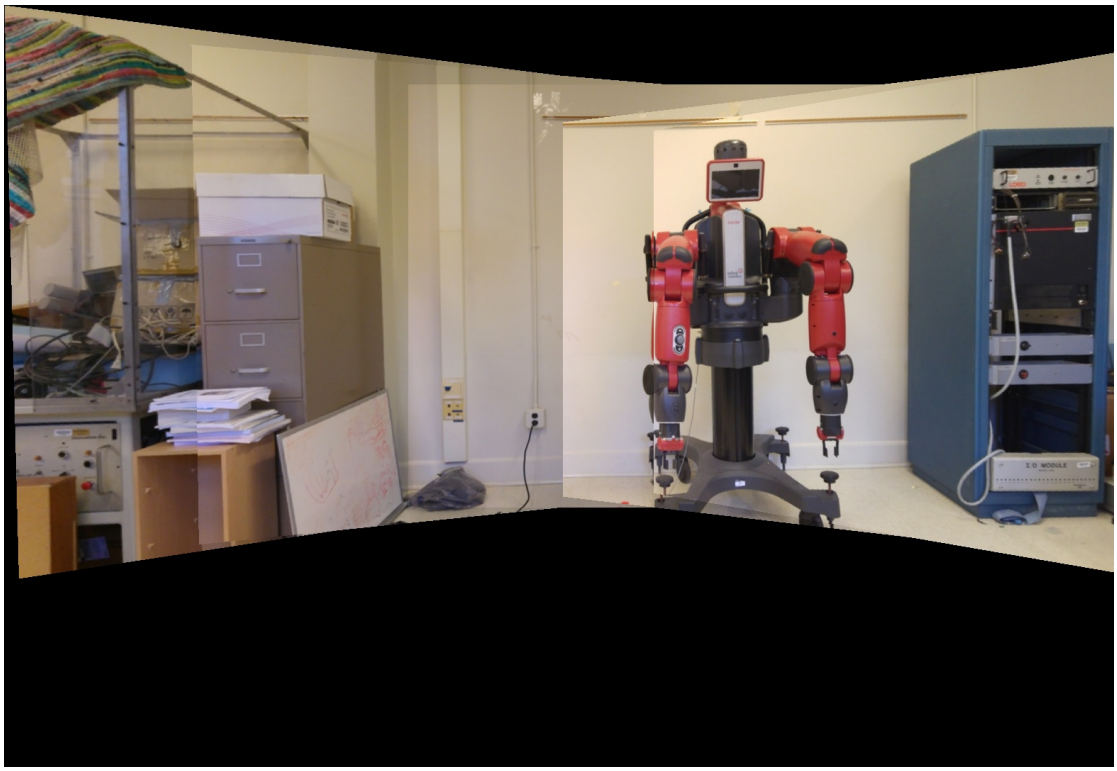

Figure 14: Without DogLeg refinement

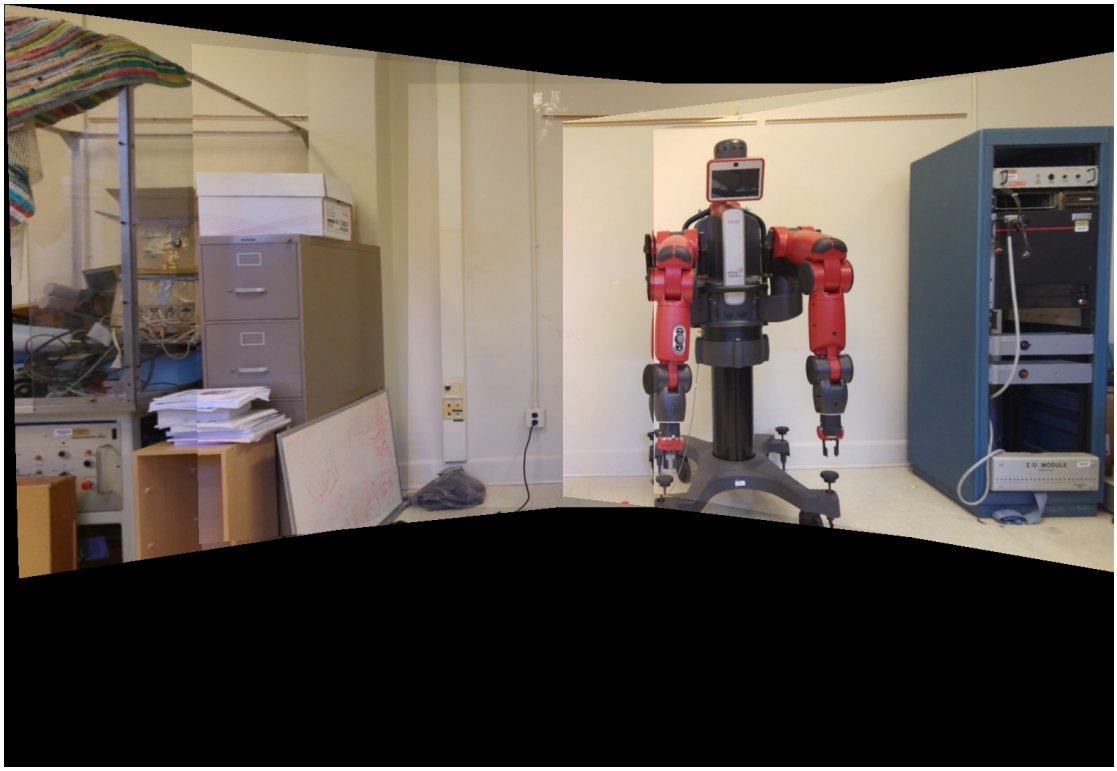

Figure 15: With DogLeg refinement

Comments

We do not see much difference between using DogLeg and without using DogLeg. The reason is that the Homography computed after RANSAC is already very very close to the local minima of the basin from where the DogLeg iterations are started.

Code

The script is as follows and is self-explanatory

```
"""
Author : Debasmit Das
"""

import cv2
import numpy as np
import math

############################################
# SIFT function modules start here
############################################

# Function to detect and compute keypoints using SIFT
def SIFTdetect(img,nfeat,sig,layers):
```

```
imggray=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
SIFT = cv2.xfeatures2d.SIFT_create(nfeatures=nfeat,
    nOctaveLayers=layers,contrastThreshold=0.03,edgeThreshold=10,sigma=sig)
keypts, desc = SIFT.detectAndCompute(imggray,None)
pts=[]
for key in keypts:
pts.append([np.round(key.pt[0],0),np.round(key.pt[1],0)])

return img,np.asarray(pts),np.asarray(desc)


# Function to find the Normalised Cross Corelation for each level starts here
def NCC(kp1,des1,kp2,des2,T_local,T_global):

nccmat=np.zeros((len(kp1),len(kp2)),dtype='float')
for i in range(0,len(kp1)):
for j in range(0,len(kp2)):
src_mean=np.mean(des1[i,:])
dest_mean=np.mean(des2[j,:])
src_norm=np.subtract(des1[i,:],src_mean)
dest_norm=np.subtract(des2[j,:],dest_mean)
num=np.sum(np.multiply(src_norm,dest_norm))
src_sq=np.sum(np.square(src_norm))
dest_sq=np.sum(np.square(dest_norm))
den=np.sqrt(src_sq*dest_sq)
nccmat[i,j]=num/den
pts=[]

# To remove weak correspondences
for i in range(0,len(kp1)):
for j in range(0,len(kp2)):
if nccmat[i,j]==np.max(nccmat[i,:]) and nccmat[i,j]>T_global*np.mean(nccmat):
loc_max=nccmat[i,j]
nccmat[i,j]=np.min(nccmat[i,:])
# Eliminate false correspondences
if np.max(nccmat[i,:])/loc_max < T_local:
        #Removing Many to 1 mappings
nccmat[:,j]=0
nccmat[i,j]=loc_max; print nccmat[i,j]
pts.append([kp1[i,0],kp1[i,1],kp2[j,0],kp2[j,1]])

        return np.asarray(pts)

##########################################
# RANSAC function modules start here
##########################################

#This is the RANSAC method
def Ransac(ptssrc,ptsdest,noise,error,n,pNoOutlier):
    H=np.zeros((3,3),dtype='float')
    pos_sol=[]
    # Spatial resolution is set
```

```
    delta=float(3*noise)
    N=int(math.log(1-pNoOutlier)/math.log(1-(1-error)**n))
    M=int((1-error)*len(ptssrc))

    for loop in range(N):
        random_index=np.random.random_integers(1,len(ptssrc)-1,n)
        pts1=[]
        pts2=[]
        for loop in range(n):
            pts1.append(ptssrc[random_index[loop],:])
            pts2.append(ptsdest[random_index[loop],:])
        ptstmp1=np.asarray(pts1)
        ptstmp2=np.asarray(pts2)

        # While Homography function uses solution to Ah=0
        H=Homography(ptstmp1,ptstmp2)
        InlierCount= Inliers_count(ptssrc,H,ptsdest,delta)
        H_vec=np.reshape(H,9)

        if InlierCount > M:
            pos_sol.append([H_vec,InlierCount])

    # Finally obtaining the Final homography
    solution=np.asarray(pos_sol)

    idx_MaxInlierCount = np.argmax(solution[:,-1])


    H_ransac=np.reshape(solution[idx_MaxInlierCount,0],(3,3))

    return H_ransac


# Finding Homography
def Homography(points_src,points_dest):
    Mat_A=np.zeros((len(points_src)*2,9),dtype='float')

    if points_src.shape[0] != points_dest.shape[0] or points_src.shape[1] != points_dest.shape[1]:
        exit(1)

    for i in range(0,len(points_src)):
        Mat_A[i*2+1]=[points_src[i,0],
        points_src[i,1],1,0,0,0,(
        -1*points_src[i,0]*points_dest[i,0]),
        (-1*points_src[i,1]*points_dest[i,0]),
        -points_dest[i,0]]
        Mat_A[i*2]=[0,0,0,-points_src[i,0],
        -points_src[i,1],-1,
        (1*points_src[i,0]*points_dest[i,1]),
        (1*points_src[i,1]*points_dest[i,1]),
        points_dest[i,1]]
```

```
    U,D,V=np.linalg.svd(Mat_A)
    V_tmp=V.T
    tmp_H=V_tmp[:,-1]
    homography= np.zeros((3,3))
    homography[0]= tmp_H[0:3]/tmp_H[8]
    homography[1]= tmp_H[3:6]/tmp_H[8]
    homography[2]= tmp_H[6:9]/tmp_H[8]
    #print homography
    return homography


# Finding the number of inliers

def Inliers_count(ptssrc,H,ptsdest,delta):
    result=np.zeros((ptssrc.shape),dtype='int')

    for loop in range(len(ptssrc)):
        result_tmp=np.dot(H,([ptssrc[loop,0],ptssrc[loop,1],1]))
        result[loop,0]=result_tmp[0]/result_tmp[2]
        result[loop,1]=result_tmp[1]/result_tmp[2]
    diff=result-ptsdest
    #print diff
    sq_dist=np.square(diff)
    Dist=np.sqrt(sq_dist[:,0]+sq_dist[:,1])
    #print Dist
    Inliers=[1 for val in Dist if (val < delta)]
    Inlier_count= len(Inliers)
    return Inlier_count


# Finding the refined Inliers
def Inliers(ptssrc,ptsdest,H,delta):
    result=np.zeros((ptssrc.shape),dtype='int')

    for loop in range(len(ptssrc)):
        result_tmp=np.dot(H,([ptssrc[loop,0],ptssrc[loop,1],1]))
        result[loop,0]=result_tmp[0]/result_tmp[2]
        result[loop,1]=result_tmp[1]/result_tmp[2]
    diff=result-ptsdest
    #print diff
    sq_dist=np.square(diff)
    Dist=np.sqrt(sq_dist[:,0]+sq_dist[:,1])

    Inlier=[]
    for loop in range(len(Dist)):
        if Dist[loop] < delta:
            Inlier.append([ptssrc[loop,:],ptsdest[loop,:]])

    return np.asarray(Inlier)

##########################################
```

```
# DogLeg function modules start here
########################################

# Minimizing Homography's error using Dogleg

def Dogleg(ptssrcX,ptsdestX1,H):

    r=1
    T=0.6
    # compute Jacobian of homography function
    Jf=Jacobian_fn(ptsdestX1,H)
    # Compute u
    u=T*np.max(np.diag(np.dot(Jf.T,Jf)))

    ptsXlist=[]
    for loop in range(len(ptssrcX)):
        ptsXlist.append(ptssrcX[loop,0])
        ptsXlist.append(ptssrcX[loop,1])
    pts_X=np.asarray(ptsXlist)
    while True:
    # Finding f(p)
        fp=Apply_Homography(ptsdestX1,H)
        # compute episilon pk
        ep=pts_X-fp
        # compute the cost function
        Cp=(np.linalg.norm(ep)**2)
        Jf=Jacobian_fn(ptsdestX1,H)
        # compute delta for gradient descent
        Jt_ep_k= np.dot(Jf.T,ep)

        delta_GD=np.linalg.norm(Jt_ep_k)*Jt_ep_k/np.linalg.norm(np.dot(Jf,Jt_ep_k))
        # compute delta for Gauss-Newton
        delta_GN=np.dot(np.linalg.inv(np.dot(Jf.T,Jf)+u*np.identity(9)),Jt_ep_k)
        delta_p=np.linalg.inv(np.dot(Jf.T,Jf)).dot(np.dot(Jf.T,ep))
        H_old=H

        if np.linalg.norm(delta_GN)<r:
            H=H+np.reshape(delta_GN,(3,3))
        elif np.linalg.norm(delta_GN)>= r and np.linalg.norm(delta_GD)<=r:


            beta_eqn=[np.linalg.norm(delta_GN-delta_GD)**2
            ,2*(delta_GD.T).dot(delta_GN-delta_GD),
            (np.linalg.norm(delta_GD)**2-r**2)]
            beta=np.max(np.roots(beta_eqn))
            H=H+np.reshape(delta_GD,(3,3))+beta*
            (np.reshape(delta_GN,(3,3))-np.reshape(delta_GD,(3,3)))
        else:
            H=H+(r/np.linalg.norm(delta_GD))*np.reshape(delta_GD,(3,3))
        fp1=Apply_Homography(ptsdestX1,H)
        Cp1=(np.linalg.norm(pts_X-fp1)**2)
        delta_DL=(Cp-Cp1)/(2*delta_p.dot(Jt_ep_k)-np.dot(np.dot(Jf,delta_p).T,np.dot(Jf,delta_p)))
```

```
        u=u*max(1/3,(1-(2*delta_DL-1)**3))
        if delta_DL<=0:
            H=H_old
            r=r/2
        elif delta_DL<0.25:
            r=r/4
        elif delta_DL<0.75:
            r=r
            # do nothing
        else:
            r=2*r
        if Cp-Cp1 < 0.001:
            break
        else:
            continue

    return H



# Applying homography to points
def Apply_Homography(ptssrc,H):
    result=np.zeros((ptssrc.shape),dtype='float')
    output=[]
    for loop in range(len(ptssrc)):
        result_tmp=np.dot(H,([ptssrc[loop,0],ptssrc[loop,1],1]))
        result[loop,0]=result_tmp[0]/result_tmp[2]
        result[loop,1]=result_tmp[1]/result_tmp[2]
        output.append(result[loop,0])
        output.append(result[loop,1])
    return np.asarray(output)



# Finding Jacobian of fx
def Jacobian_fn(ptsX1,H):

    Jacobian=np.zeros((2*len(ptsX1),9),dtype='float')
    for loop in range(len(ptsX1)):

        tmp_f=np.dot(H,([ptsX1[loop,0],ptsX1[loop,1],1]))
        Jacobian[loop*2]=[ptsX1[loop,0]/tmp_f[2],
        ptsX1[loop,1]/tmp_f[2],1/tmp_f[2],0,0,0,
        -ptsX1[loop,0]*tmp_f[0]/(tmp_f[2]**2),-ptsX1[loop,1]*tmp_f[0]/(tmp_f[2]**2),
        -1*tmp_f[0]/(tmp_f[2]**2)]
        Jacobian[loop*2+1]=[0,0,0,ptsX1[loop,0]/tmp_f[2],
        ptsX1[loop,1]/tmp_f[2],1/tmp_f[2],
        -ptsX1[loop,0]*tmp_f[1]/(tmp_f[2]**2),-ptsX1[loop,1]*tmp_f[1]/(tmp_f[2]**2),
        -1*tmp_f[1]/(tmp_f[2]**2)]
    return Jacobian
```

```
##########################################
# Modules for Image Mapping starts here
##########################################

# Getting function for the interpolation of points

def getdata(point, img):
    tp_left =img[(math.floor(point[1])),(math.floor(point[0]))]
    tp_right =img[math.floor(point[1]),math.floor(point[0]+1)]
    bt_left =img[math.floor(point[1]+1),math.floor(point[0])]
    bt_right =img[math.floor(point[1]+1),math.floor(point[0]+1)]
    diff_x = point[1] - math.floor(point[1])
    diff_y = point[0] - math.floor(point[0])
    tp_left_weight= pow(pow(diff_x,2)+pow(diff_y,2),-0.5)
    tp_right_weight = pow(pow(diff_x,2)+pow(1-diff_y,2),-0.5)
    bt_left_weight = pow(pow(1-diff_x,2)+pow(diff_y,2),-0.5)
    bt_right_weight = pow(pow(1-diff_x,2)+pow(1-diff_y,2),-0.5)
    resultant_pt = (tp_left*tp_left_weight+tp_right*tp_right_weight+
    bt_left*bt_left_weight+bt_right*bt_right_weight)/(tp_left_weight+
    tp_right_weight+bt_left_weight+bt_right_weight)
    return resultant_pt



# Code for the image mapping starts here

def image_mapping(src_image,dest_image,Homography,offset_xy):

    for i in range(0,src_image.shape[0]):
        for j in range(0,src_image.shape[1]):
                point_tmp = np.array([j+offset_xy[0],i+offset_xy[1], 1])
                trans_coord = np.array(np.dot(Homography,point_tmp))
                trans_coord = trans_coord/trans_coord[2]

                if (trans_coord[1]>0) and (trans_coord[1]<dest_image.shape[0]-1)
                and (trans_coord[0]>0)
                and (trans_coord[0]<dest_image.shape[1]-1):
                    src_image[i][j]=getdata(trans_coord,dest_image)
    return src_image



# Code for finding the boundary of the image starts here

def Boundary(H,src_image):
    # Boundary points of the given image is extracted here
    Points=np.array([[0,0,1],[0,src_image.shape[1],1],
    [src_image.shape[0],0,1],[src_image.shape[0],src_image.shape[1],1]])
    tmp=np.zeros((Points.shape[1],Points.shape[0]))
    # Boundary points of the given image on the new plane is computed here
    tmp=np.array((np.dot(H,Points.T)).T)
```

```
    for i in range(0,Points.shape[0]):
        tmp[i]=tmp[i]/tmp[i,2]
    tmp=tmp.T
    return tmp[0:2,:]


# Image corresponding plotting code starts here

def plotting(Corner_Coord,img1,img2):
# creating a base image which has image 1 and image 2
     img=np.zeros((max(img1.shape[0],img2.shape[0]),img1.shape[1]+img2.shape[1],3))
     img[:img1.shape[0], :img1.shape[1]]=img1
     img[:img2.shape[0], img1.shape[1]:img1.shape[1]+img2.shape[1]]=img2

# plotting the correspondence using lines
for coord in Corner_Coord:
img=cv2.line(img,(int(coord[0]),int(coord[1])),(img1.shape[1]+
        int(coord[2]),int(coord[3])), (0,255,0))
return img




# This code is for plotting the outliers

def outlier_plotting(Corner_Coord,img1,img2,H,sigma):
# creating a concatenated image
img=np.zeros((max(img1.shape[0],img2.shape[0]),img1.shape[1]+img2.shape[1],3))
img[:img1.shape[0], :img1.shape[1]]=img1
img[:img2.shape[0], img1.shape[1]:img1.shape[1]+img2.shape[1]]=img2
pts_src=Corner_Coord[:,0:2]
pts_dest=Corner_Coord[:,2:4]

for loop in range(len(pts_src)):
pts_dest_calc=np.dot(H,[pts_src[loop,0],pts_src[loop,1],1])
pts_dest_calc=pts_dest_calc/pts_dest_calc[2]
pts_diff=np.sqrt(np.sum((pts_dest_calc[0:2]-pts_dest[loop,:])**2))
if  pts_diff < 3*sigma:
# plotting the correspondence using lines
cv2.circle(img,(int(pts_src[loop,0]),int(pts_src[loop,1])),2,(255,0,0),2)
cv2.circle(img,(img1.shape[1]+int(pts_dest_calc[0]),int(pts_dest_calc[1])),2,(255,0,0),2)
cv2.line(img,(int(pts_src[loop,0]),int(pts_src[loop,1])),(img1.shape[1]+
            int(pts_dest_calc[0]),int(pts_dest_calc[1])), (0,255,0))
else :
cv2.circle(img,(int(pts_src[loop,0]),int(pts_src[loop,1])),2,(0,0,255),2)
cv2.circle(img,(img1.shape[1]+int(pts_dest_calc[0]),int(pts_dest_calc[1])),2,(0,0,255),2)
cv2.line(img,(int(pts_src[loop,0]),int(pts_src[loop,1])),(img1.shape[1]+
            int(pts_dest_calc[0]),int(pts_dest_calc[1])), (0,0,255))
return img
```

```
##########################################
# Main method starts here
##########################################

if __name__ == "__main__":


    img_1=cv2.imread('1.jpg')
    img_2=cv2.imread('2.jpg')
    img_3=cv2.imread('3.jpg')
    img_4=cv2.imread('4.jpg')
    img_5=cv2.imread('5.jpg')

    layers=4
    sigma=4
    T_ncc=0.66
    T_max_global=0.96
    f_count=400
    error=0.4
    pts_per_trial=6
    percent_prob=0.999
    G_Noise=1
    """
    ##########################################

    ##########################################
    # Computing SIFT keypoints and Descriptor
    ##########################################
    """

    img1, kp1, desp1 = SIFTdetect(img_1,f_count,sigma,layers)
    img2, kp2, desp2 = SIFTdetect(img_2,f_count,sigma,layers)
    img3, kp3, desp3 = SIFTdetect(img_3,f_count,sigma,layers)
    img4, kp4, desp4 = SIFTdetect(img_4,f_count,sigma,layers)
    img5, kp5, desp5 = SIFTdetect(img_5,f_count,sigma,layers)
    ##########################################
    # Computing correspondence using NCC
    ##########################################
    Coord_NCC12=NCC(kp1,desp1,kp2,desp2,T_ncc,T_max_global)
    img=plotting(Coord_NCC12,img1,img2)
    cv2.imwrite("Pair1.jpg",img)
    Coord_NCC23=NCC(kp2,desp2,kp3,desp3,T_ncc,T_max_global)
    img=plotting(Coord_NCC23,img2,img3)
    cv2.imwrite("Pair2.jpg",img)
    Coord_NCC34=NCC(kp3,desp3,kp4,desp4,T_ncc,T_max_global)
    img=plotting(Coord_NCC34,img3,img4)
    cv2.imwrite("Pair3.jpg",img)
    Coord_NCC45=NCC(kp4,desp4,kp5,desp5,T_ncc,T_max_global)
    img=plotting(Coord_NCC45,img4,img5)
    cv2.imwrite("Pair4.jpg",img)
    ##########################################
```

```
##########################################
# Computing Homography using RANSAC
##########################################
pts_src12=Coord_NCC12[:,0:2]
pts_dest12=Coord_NCC12[:,2:4]
H_coarse12=Ransac(pts_src12,pts_dest12,G_Noise,error,pts_per_trial,percent_prob)
img=outlier_plotting(Coord_NCC12,img1,img2,H_coarse12,G_Noise)
cv2.imwrite("Pair1outliers.jpg",img)

pts_src23=Coord_NCC23[:,0:2]
pts_dest23=Coord_NCC23[:,2:4]
H_coarse23=Ransac(pts_src23,pts_dest23,G_Noise,error,pts_per_trial,percent_prob)
img=outlier_plotting(Coord_NCC23,img2,img3,H_coarse23,G_Noise)
cv2.imwrite("Pair2outliers.jpg",img)

pts_src34=Coord_NCC34[:,0:2]
pts_dest34=Coord_NCC34[:,2:4]
H_coarse34=Ransac(pts_src34,pts_dest34,G_Noise,error,pts_per_trial,percent_prob)
img=outlier_plotting(Coord_NCC34,img3,img4,H_coarse34,G_Noise)
cv2.imwrite("Pair3outliers.jpg",img)

pts_src45=Coord_NCC45[:,0:2]
pts_dest45=Coord_NCC45[:,2:4]
H_coarse45=Ransac(pts_src45,pts_dest45,G_Noise,error,pts_per_trial,percent_prob)
img=outlier_plotting(Coord_NCC45,img4,img5,H_coarse45,G_Noise)
cv2.imwrite("Pair4outliers.jpg",img)

##########################################



##########################################
# Image Mosaicing
##########################################
# considering the image 3 to be the center image

H_coarse13=H_coarse12.dot(H_coarse23)

H_coarse35=H_coarse34.dot(H_coarse45)
H_coarse53=np.linalg.inv(H_coarse35)
H_coarse43=np.linalg.inv(H_coarse34)
H_33=np.array([[1,0,0],[0,1,0],[0,0,1]])
# boundaries of each image in plane of image 3
B1=Boundary(H_coarse13/H_coarse13[2,2],img_1)
B2=Boundary(H_coarse23/H_coarse23[2,2],img_2)
B3=np.array([[0,0],[0,img_3.shape[1]],[img_3.shape[0],0],[img_3.shape[0],img_3.shape[1]]])
B3=B3.T
B4=Boundary(H_coarse43/H_coarse43[2,2],img_4)
B5=Boundary(H_coarse53/H_coarse53[2,2],img_5)
min_xy=np.amin(np.amin([B1,B2,B3,B4,B5],2),0)
max_xy=np.amax(np.amax([B1,B2,B3,B4,B5],2),0)
```

```
offset_dim=max_xy-min_xy
Base_image=np.zeros((offset_dim[1],offset_dim[0],3),dtype='uint')
# Mapping the Homographies into the image on the required plane
output=image_mapping(Base_image,img_1,np.linalg.inv(H_coarse13/H_coarse13[2,2]),min_xy)
output=image_mapping(output,img_2,np.linalg.inv(H_coarse23/H_coarse23[2,2]),min_xy)
output=image_mapping(output,img_3,np.linalg.inv(H_33),min_xy)
output=image_mapping(output,img_4,H_coarse34/H_coarse34[2,2],min_xy)
output=image_mapping(output,img_5,H_coarse35/H_coarse35[2,2],min_xy)
cv2.imwrite("FinalImageWithoutDogleg.jpg",output)
##########################################

Inliersset1=Inliers(pts_src12,pts_dest12,H_coarse12,3*G_Noise)
Inliersset2=Inliers(pts_src23,pts_dest23,H_coarse23,3*G_Noise)
Inliersset3=Inliers(pts_src34,pts_dest34,H_coarse34,3*G_Noise)
Inliersset4=Inliers(pts_src45,pts_dest45,H_coarse45,3*G_Noise)
##########################################
# Fine Tuning Homography using Dogleg
##########################################
H_finetune12=Dogleg(Inliersset1[:,1],Inliersset1[:,0],H_coarse12)
H_finetune23=Dogleg(Inliersset2[:,1],Inliersset2[:,0],H_coarse23)
H_finetune34=Dogleg(Inliersset3[:,1],Inliersset3[:,0],H_coarse34)
H_finetune45=Dogleg(Inliersset4[:,1],Inliersset4[:,0],H_coarse45)

H_finetune13=H_finetune12.dot(H_finetune23)
H_finetune13=H_finetune13/H_finetune13[2,2]
H_finetune35=H_finetune34.dot(H_finetune45)
H_finetune35=H_finetune35/H_finetune35[2,2]
H_finetune43=np.linalg.inv(H_finetune34)
H_finetune53=np.linalg.inv(H_finetune35)
##########################################

# boundaries of each image in plane of image 3 after Dogleg
B1_dg=Boundary(H_finetune13,img_1)
B2_dg=Boundary(H_finetune23,img_2)
B3_dg=np.array([[0,0],[0,img_3.shape[1]],[img_3.shape[0],0],[img_3.shape[0],img_3.shape[1]]])
B3_dg=B3_dg.T
B4_dg=Boundary(H_finetune43,img_4)
B5_dg=Boundary(H_finetune53,img_5)
min_xy_dg=np.amin(np.amin([B1_dg,B2_dg,B3_dg,B4_dg,B5_dg],2),0)
max_xy_dg=np.amax(np.amax([B1_dg,B2_dg,B3_dg,B4_dg,B5_dg],2),0)

offset_dim_dg=max_xy_dg-min_xy_dg
Base_image_dg=np.zeros((offset_dim_dg[1],offset_dim_dg[0],3),dtype='uint')
# Mapping the Homographies and all images into the image on the required plane with dogleg optimis
output1=image_mapping(Base_image_dg,img_1,np.linalg.inv(H_finetune13),min_xy_dg)
output1=image_mapping(output1,img_2,np.linalg.inv(H_finetune23),min_xy_dg)
output1=image_mapping(output1,img_3,np.linalg.inv(H_33),min_xy_dg)
output1=image_mapping(output1,img_4,H_finetune34,min_xy_dg)
output1=image_mapping(output1,img_5,H_finetune35,min_xy_dg)
cv2.imwrite('FinalImageWithDogleg.jpg',output1)
```