CS217 : Assignment 2 : Structure From Motion

Please edit the cell below to include your name and student ID #

name: Saad Manzur

SID: 83166813

1. Feature Matching ¶

1.1 SIFT matching library

Install an implementation of SIFT-based feature matching; I recommend using the one integrated into opencv-python. However, getting it installed is a little bit tricky because the algorithm is patented so it isn't included with some distributions. I used the recipe below to install on my system.

conda install opencv

pip install opencv-contrib-python==3.4.2.16

I have provided some example code below exercising the OpenCV SIFT matching functions. Once you are comfortable with this, please implement a function which takes as input the two images and returns the coordinates for the candidate matching points for use in the remainder of the assignment. If you decide not to use OpenCV then you should implement this function to wrap whatever version of SIFT you use.

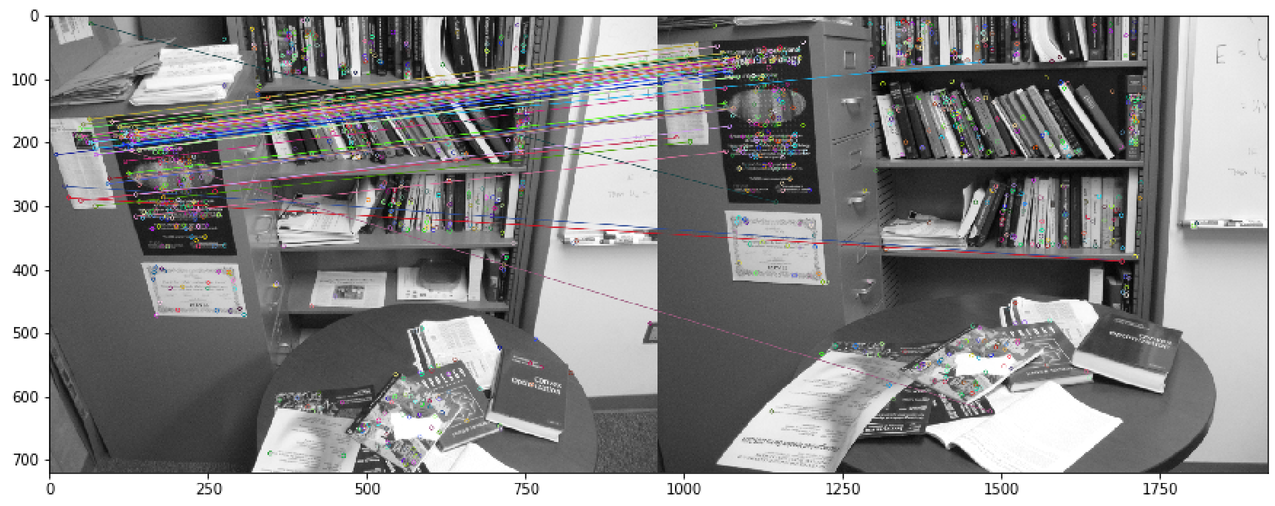
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize
import cv2 as opencv

# you can use matplotlib notebook during development but
# when you print out your notebook as pdf there may be
# problems with figures displaying so you may want to switch
# to inline
%matplotlib inline
```

In [2]:
```
def getMatches(img1,img2):
    """
    Given two grayscale images, return a set of candidate matches using SIFT keypoint mat
ching

    Parameters
    ----------
    img1 : 2D numpy.array
    img2 : 2D numpy.array
            Grayscale images to match

    Returns
    -------
    pts1 : 2D numpy.array (dtype=float)
    pts2 : 2D numpy.array (dtype=float)
        Candidate pairs of matching points from img1 and img2 stored in arrays of shape
 (2,N)


    """
    sift = opencv.xfeatures2d.SIFT_create()
    keypoints1, descriptor1 = sift.detectAndCompute(img1, None)
    keypoints2, descriptor2 = sift.detectAndCompute(img2, None)

    bf = opencv.BFMatcher()
    matches = bf.knnMatch(descriptor1, descriptor2, k=2)

    pts1 = []
    pts2 = []
    kpts1 = []
    kpts2 = []
    good = []
    for m, n in matches:
        if m.distance < 0.75*n.distance:
            pts2.append(keypoints2[m.trainIdx].pt)
            pts1.append(keypoints1[m.queryIdx].pt)
            kpts2.append(keypoints2[m.trainIdx])
            kpts1.append(keypoints1[m.queryIdx])
            m.trainIdx = len(pts2)-1
            m.queryIdx = len(pts1)-1
            good.append([m])

    pts1 = np.array(pts1)
    pts1 = np.float32(pts1[:, :])
    pts1 = pts1.T

    pts2 = np.array(pts2)
    pts2 = np.float32(pts2[:, :])
    pts2 = pts2.T

    return (pts1,pts2,kpts1,kpts2,good)
```

1.2 Visualize SIFT candidate matches

Write code below to exercise your **getMatches** function. Choose a pair of images you want to work with. I've provided a few examples but you are encouraged to take your own photos or use some images off the web. Visualize the resulting set of candidate matches. You can either modify your **getMatches** function to also return the **cv2** keypoint data structures and use **cv2.drawMatchesKnn**, or alternately come up with your own visualization technique to show correpsondences.

In [19]:
```
img1 = opencv.imread('sfm/IMG_1434.JPG')
img1 = opencv.cvtColor(img1, opencv.COLOR_BGR2RGB)
gray1 = opencv.cvtColor(img1, opencv.COLOR_RGB2GRAY)
dim1 = (0.6*np.asarray(gray1.shape)).astype(int)
gray1 = opencv.resize(gray1, (dim1[1], dim1[0]))

img2 = opencv.imread('sfm/IMG_1435.JPG')
img2 = opencv.cvtColor(img2, opencv.COLOR_BGR2RGB)
gray2 = opencv.cvtColor(img2, opencv.COLOR_RGB2GRAY)
dim2 = (0.6*np.asarray(gray2.shape)).astype(int)
gray2 = opencv.resize(gray2, (dim2[1], dim2[0]))

pts1, pts2, kpts1, kpts2, good = getMatches(gray1, gray2)

img3 = opencv.drawMatchesKnn(gray1,kpts1,gray2,kpts2,good[:56],None)

plt.rcParams['figure.figsize'] = [15,15]
plt.imshow(img3)
plt.show()
```



2. Estimation of the Fundamental Matrix

As you can see from the matching results, SIFT finds matches which are not consistent with a single overall camera motion. In order to find a consistent set of matches, implement RANSAC to estimate the fundamental matrix F. For each random sample you should estimate F using the normalized 8 point algorithm (as described in Hartley's paper "In defense of ..." linked from the class website and discussed in section 4.4 of Hartley and Zisserman). You will need to experiment with the appropriate error threshold for determining inliers and outliers

In [39]:
```
# For debugging
def fitFundamentalFromOpenCV(pts2L, pts2R):
    F, mask = opencv.findFundamentalMat(pts2L.T, pts2R.T, opencv.FM_8POINT)
    pts2LH = np.vstack((pts2L, np.ones((1,pts2L.shape[1]))))
    pts2RH = np.vstack((pts2R, np.ones((1,pts2R.shape[1]))))

    if not isinstance(F, np.ndarray):
        return F, 99999

    fiterror = np.average((pts2LH.T @ (F @ pts2RH) )**2)

    return F, fiterror


def getNormalizationMatrix(pts2):
    mean = np.mean(pts2[:2], axis=1)
    scale = np.sqrt(2) / (np.std(pts2[:2]))**2
    transformation = np.array([[scale, 0, -scale*mean[0]],
                               [0, scale, -scale*mean[1]],
                               [0, 0, 1]])
    return transformation

def fitFundamental(pts2L,pts2R):
    """
    Estimate fundamental matrix from point correspondences in two views

    Parameters
    ----------
    pts2L : 2D numpy.array (dtype=float)
    pts2R : 2D numpy.array (dtype=float)
        Coordinates of N points in the left and right view stored in arrays of shape (2,
N)


    Returns
    -------
    F : 2D numpy.array (dtype=float)
        Fundamental matrix of shape (3,3)

    fiterror : float
        The quality of the fit measured as the average of (x^T*F*x')^2 over all correspon
ding points x,x'

    """
    #Fcv, mask = opencv.findFundamentalMat(pts2L.T, pts2R.T, opencv.FM_8POINT)

    if pts2L.shape[1] != pts2R.shape[1]:
        return np.zeros((3,3)), 99999

    numberOfPoints = pts2L.shape[1]

    pts2L = np.vstack((pts2L, np.ones((1, numberOfPoints))))
    pts2R = np.vstack((pts2R, np.ones((1, numberOfPoints))))

    '''
    #Step 1: First do the normalization by doing a translation and isotropic scaling
    meanL = np.mean(pts2L[:2], axis=1)
    scaleL = np.sqrt(2) / (np.std(pts2L[:2]))**2
    transformationL = np.array([[scaleL, 0, -scaleL*meanL[0]],
                               [0, scaleL, -scaleL*meanL[1]],
                               [0, 0, 1]])
    '''
    transformationL = getNormalizationMatrix(pts2L)
    pts2LN = np.dot(transformationL, pts2L)

    '''
    meanR = np.mean(pts2R[:2], axis=1)
    scaleR = 1 / (np.std(pts2R[:2]))**2
    transformationR = np.array([[scaleR, 0, -scaleR*meanR[0]],
                                [0, scaleR, -scaleR*meanR[1]],
                                [0, 0, 1]])
    '''
    transformationR = getNormalizationMatrix(pts2R)
    pts2RN = np.dot(transformationR, pts2R)

    A = np.zeros((numberOfPoints, 9))
    for i in range(numberOfPoints):
        A[i] = [pts2LN[0,i]*pts2RN[0,i], pts2LN[0,i]*pts2RN[1,i], pts2LN[0,i]*pts2RN[2,i
],
                pts2LN[1,i]*pts2RN[0,i], pts2LN[1,i]*pts2RN[1,i], pts2LN[1,i]*pts2RN[2,i
],
                pts2LN[2,i]*pts2RN[0,i], pts2LN[2,i]*pts2RN[1,i], pts2LN[2,i]*pts2RN[2,i
]]

    U, E, Vt = np.linalg.svd(A)
    F = Vt[-1].reshape(3,3)

    U, E, Vt = np.linalg.svd(F)
    E[-1] = 0
    F = U @ np.diag(E) @ Vt
    F = F/F[2,2]

    F = np.dot(transformationR.T, np.dot(F, transformationL))
    F = F/F[2,2]

    fiterror = (pts2L.T @ F @ pts2R)**2
    fiterror = np.average(fiterror)

    return (F,fiterror)
```

In [6]:

```
#
#  I recommend debugging / testing your fitting function with some synthetic example
#  where you know what the "true" F matrix is. You can do this by generating points
#  from two views using the project function from Assignment #1.
#

F, fiterror = fitFundamental(pts1[:,:8], pts2[:,:8])
print(fiterror)
print(F)
```

```
0.1488228568321213
[[-1.08040364e-06  6.64299137e-06 -2.54079476e-03]
 [-5.85430889e-06  1.06302239e-06  3.02392731e-04]
 [ 2.41454434e-03 -3.23087051e-03  1.00000000e+00]]
```

```
In [7]: def fitFundamentalRANSAC(pts2L,pts2R,thresh,niter):
            """
            Robust estimation of fundamental matrix from point correspondences in two views

            Parameters
            ----------
```

```
    pts2L : 2D numpy.array (dtype=float)
    pts2R : 2D numpy.array (dtype=float)
            Coordinates of N points in the left and right view stored in arrays of shape
 (2,N)

    thresh : float
        Error threshold to decide whether a point is an inlier or outlier

    niter : int
        Number of RANSAC iterations to run

    Returns
    -------
    F : 2D numpy.array (dtype=float)
        Fundamental matrix of shape (3,3)

    inliers : numpy.array (dtype=int)
        Indices of the points which were determined to be inliers

    fiterror : float
        The quality of the fit measured as the average of (x^T*F*x')^2 over all correspon
ding points x,x'

    """

    MIN_INLIER_COUNT = 6

    maxInlier = 0
    inliers = []
    chosenF = np.zeros((3,3))
    chosenFiterror = 9999

    for i in range(niter):
        modelIndices = np.random.choice(pts2L.shape[1], 8, replace=False)

        pts2Lmodel = pts2L[:,modelIndices]
        pts2Rmodel = pts2R[:,modelIndices]

        pts2Ltest = np.vstack((pts2L, np.ones(pts2L.shape[1])))
        pts2Rtest = np.vstack((pts2R, np.ones(pts2R.shape[1])))

        F, fiterror = fitFundamental(pts2Lmodel, pts2Rmodel)
        #F, fiterror = fitFundamentalFromOpenCV(pts2Lmodel, pts2Rmodel)

        if not isinstance(F, np.ndarray):
            continue

        currentInliers = []
        for j in range(pts2Ltest.shape[1]):
            if j not in modelIndices:
                error = (pts2Ltest[:,j].T @ F @ pts2Rtest[:,j])**2

                if error < thresh:
                    currentInliers.append(j)

        if len(currentInliers) > MIN_INLIER_COUNT:
            #np.append(np.asarray(currentInliers), modelIndices)

            F, fiterror = fitFundamental(pts2L[:,currentInliers], pts2R[:,currentInliers
])
            #F, fiterror = fitFundamentalFromOpenCV(pts2L[:,currentInliers], pts2R[:,curr
entInliers])
            if not isinstance(F, np.ndarray):
                continue

            if fiterror < chosenFiterror:
                currentInliers = np.append(np.asarray(currentInliers), modelIndices)
                inliers = currentInliers
                maxInlier = len(currentInliers)
                chosenF = F
                chosenFiterror = fiterror
                print(i, ": ", chosenFiterror)

    #print(inliers)

    return (chosenF,inliers,chosenFiterror)
```

In [16]:

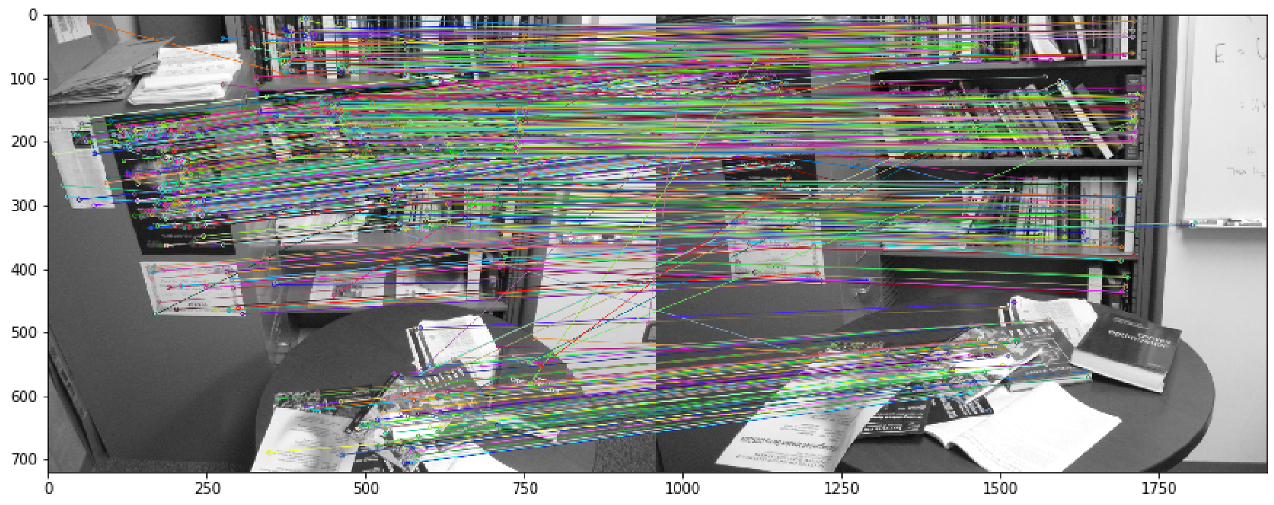
```
#  Again, it is probably good to test your RANSAC function using some toy data.
#  You can add in some random point matches as outliers and see that RANSAC
#  correctly eliminates them.
#

F, inliers, fiterror = fitFundamentalRANSAC(pts1, pts2, 4e-5, 10000)
```

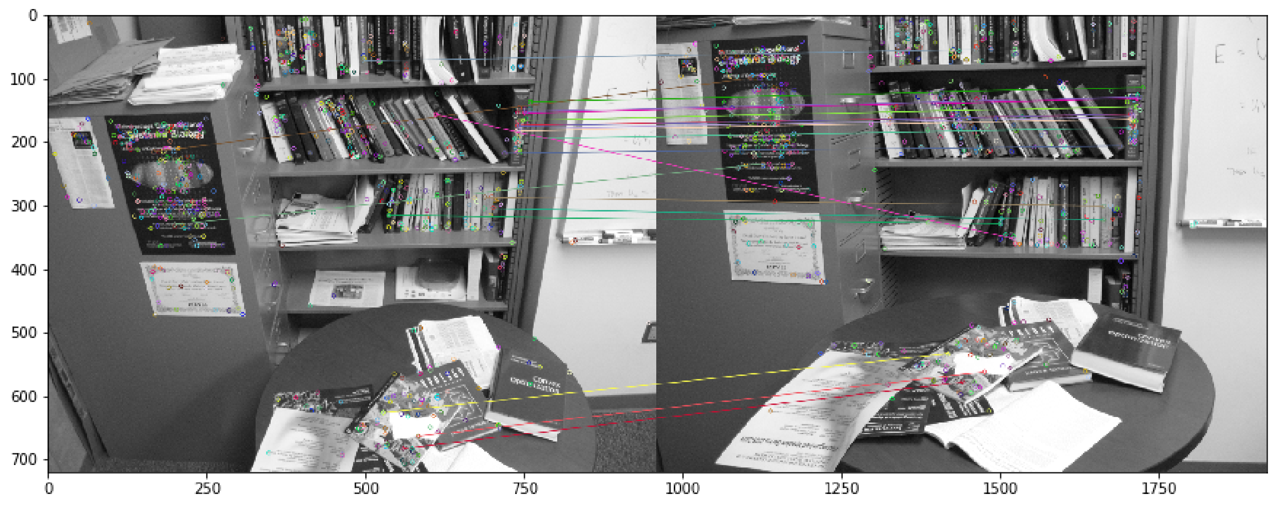
```
32 :  4.433240140035975
43 :  0.040215423435044468
96 :  0.009226011237508103
119 :  0.0029392316944516113
287 :  0.0003557968960264199
432 :  4.304213526256655e-05
558 :  5.192681168975393e-06
1300 :  1.3260799723698784e-06
1751 :  5.256734018756632e-07
3992 :  3.976479231910706e-07
7131 :  1.1294798364241344e-09
```

2.1 Visualize Correspondences

Using the same image pairs as in Problem 1, show the best set of correspondences chosen by RANSAC as inliers. If things are working correctly, the correspondences should be more consistent than in Problem 1 since they all now satisfy the same epipolar constraint. Experiment with the inlier threshold and number of iterations in order to get the best result

In [21]:
```
img1 = opencv.imread('sfm/IMG_1434.JPG')
img1 = opencv.cvtColor(img1, opencv.COLOR_BGR2RGB)
gray1 = opencv.cvtColor(img1, opencv.COLOR_RGB2GRAY)
dim1 = (0.6*np.asarray(gray1.shape)).astype(int)
gray1 = opencv.resize(gray1, (dim1[1], dim1[0]))

img2 = opencv.imread('sfm/IMG_1435.JPG')
img2 = opencv.cvtColor(img2, opencv.COLOR_BGR2RGB)
gray2 = opencv.cvtColor(img2, opencv.COLOR_RGB2GRAY)
dim2 = (0.6*np.asarray(gray2.shape)).astype(int)
gray2 = opencv.resize(gray2, (dim2[1], dim2[0]))

new = []
for i in inliers:
    new.append([good[i][0]])

pts1, pts2, kpts1, kpts2, good = getMatches(gray1, gray2)

img3 = opencv.drawMatchesKnn(gray1,kpts1,gray2,kpts2,new[:],None)

img4 = opencv.drawMatchesKnn(gray1,kpts1,gray2,kpts2,good[:],None)

plt.rcParams['figure.figsize'] = [15,15]
plt.imshow(img4)
plt.show()
plt.imshow(img3)
plt.show()
```



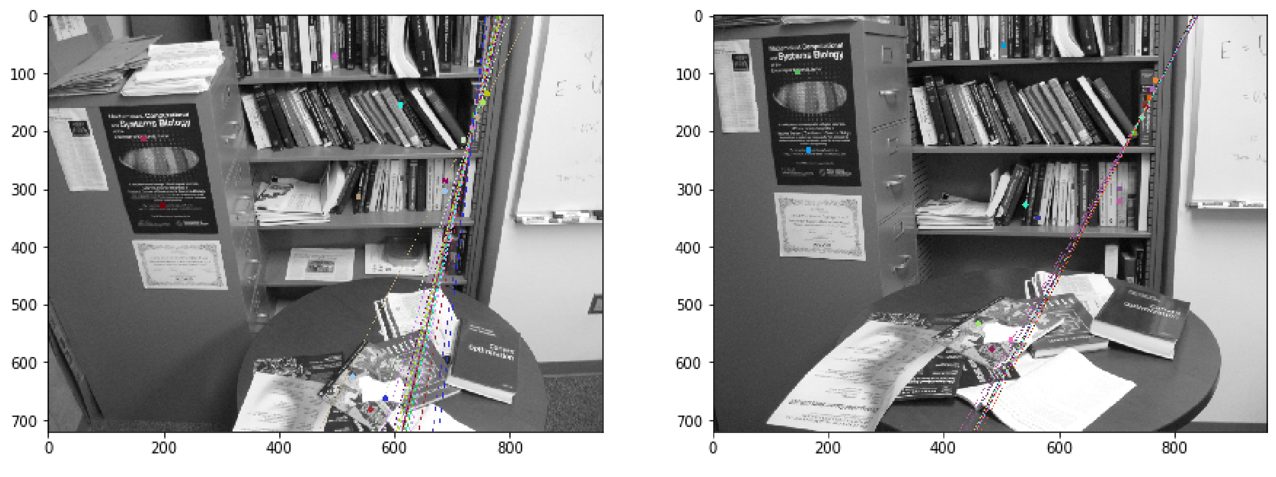


It can be seen the features are more consistent then the previous matches. From observation, decreasing the threshold reduces the chances of picking a noisy inlier set.

2.2 Epipolar Lines

Visualize the epipolar geometry for a subset of 20 feature points in one of your image pairs. For both image you should plot ~10 feature points along with the epipolar lines on which they should lie given your estimate of F and their corresponding point in the other image. What can you say about the relative pose of the two camera views based on the arrangement of epipolar lines? Can you tell which is the "left" and "right" camera from the lines alone.

In [20]:
```python
def drawlines(img1,img2,lines,pts1,pts2):
    ''' img1 - image on which we draw the epilines for the points in img2
        lines - corresponding epilines '''
    r,c = img1.shape
    img1 = opencv.cvtColor(img1,opencv.COLOR_GRAY2BGR)
    img2 = opencv.cvtColor(img2,opencv.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img1 = opencv.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = opencv.circle(img1,tuple(pt1),5,color,-1)
        img2 = opencv.circle(img2,tuple(pt2),5,color,-1)
    return img1,img2

#F, mask  = opencv.findFundamentalMat(pts1.T, pts2.T, opencv.FM_RANSAC, 4e-5, 100000)

inliers1H = np.vstack((pts1[:,inliers], np.ones((1, len(inliers)))))
inliers2H = np.vstack((pts2[:,inliers], np.ones((1, len(inliers)))))

# Find epilines corresponding to points in right image (second image) and
# drawing its lines on left image
lines1 = opencv.computeCorrespondEpilines(pts2[:,inliers].T.reshape(-1,1,2), 2,F)
lines1 = lines1.reshape(-1,3)
img5,img6 = drawlines(gray1,gray2,lines1,pts1[:,inliers].T,pts2[:,inliers].T)
# Find epilines corresponding to points in left image (first image) and
# drawing its lines on right image
lines2 = opencv.computeCorrespondEpilines(pts1[:,inliers].reshape(-1,1,2), 1,F)
lines2 = lines2.reshape(-1,3)
img3,img4 = drawlines(gray2,gray1,lines2,pts2[:,inliers].T,pts1[:,inliers].T)
plt.subplot(121),plt.imshow(img5)
plt.subplot(122),plt.imshow(img3)
plt.show()
```



The code for generating the epipolar lines were taken from https://docs.opencv.org/3.4.4/da/de9/tutorial_py_epipolar_geometry.html (https://docs.opencv.org/3.4.4/da/de9/tutorial_py_epipolar_geometry.html). On the left figure from the epipolar lines, it seems it was viewed from the left and right one from the right.

2.3 Stability

How stable is your estimate of F? If you rerun RANSAC multiple times with random samplings, does it always return the same F matrix? Please write a bit of code below to carry out this experiment and compute some quantitative measure of the variability in estimated F. How could you get a more robust estimate?

In [36]:
```
thresholds = [4e-3, 4e-4, 4e-5, 4e-6]
iterations = [1000, 10000, 100000]

allFs = []
for t in thresholds:
    for n in iterations:
        print(t, ',', n)
        F, inliers, fiterror = fitFundamentalRANSAC(pts1, pts2, t, n)
        allFs.append(F)

print("Means: ", np.mean(allFs, axis=0))
print("Standard deviations: ", np.std(allFs, axis=0))
```

```
0.004 , 1000
3 :  37.03628508330083
5 :  0.4429310396734829
10 :  0.0008088889822654922
101 :  0.0004800575761928026
483 :  4.949095786587681e-05
0.004 , 10000
3 :  3.212145739572145
5 :  0.14839835269751545
16 :  0.0007964699219131458
59 :  0.00017231437896748916
214 :  8.997856516531613e-05
414 :  7.508317000055976e-05
1011 :  6.042509927939239e-05
2093 :  3.694246166662634e-05
2123 :  2.560201178112279e-05
2938 :  1.4974553384849821e-05
4660 :  1.2335440430628229e-06
6443 :  3.6212429749740177e-07
0.004 , 100000
1 :  888.7704416028754
5 :  15.343316506596846
10 :  1.0558265249180179e-05
851 :  9.06732202264032e-06
6203 :  6.106668681022518e-06
14787 :  3.94839231369427e-06
22892 :  1.2884737593092134e-07
0.0004 , 1000
3 :  645.1153685090079
5 :  0.004435499282445631
62 :  8.829339496235385e-05
620 :  1.0736841329915724e-05
0.0004 , 10000
3 :  0.08400672163488156
63 :  0.0010958901369349933
163 :  0.0002507952273915533
191 :  0.00020136551095474545
321 :  6.060246078415647e-05
2602 :  1.8169846824334432e-05
4762 :  1.260603288346379e-05
6808 :  6.435681791339157e-07
0.0004 , 100000
4 :  0.2643007579790092
6 :  0.0015677821969303314
63 :  0.00015196597870173692
410 :  0.0001437852735224905
543 :  9.902189109074671e-05
757 :  3.353747802454026e-05
1505 :  3.602921408191937e-07
41915 :  1.2884737593092134e-07
4e-05 , 1000
28 :  0.00446937175391159
29 :  7.75142971225205e-06
4e-05 , 10000
19 :  10.686918963434739
42 :  1.4762889808976682
46 :  0.999833454979473
58 :  0.26319434707582223
233 :  0.0018614469806042693
273 :  0.00047252109549581454
3153 :  5.740095954489268e-05
4081 :  8.604197490261761e-07
7876 :  5.189771326353358e-07
4e-05 , 100000
52 :  0.3879697177555824
282 :  0.03138519136035469
315 :  0.0001525462695165743
5663 :  8.560636298220281e-06
14403 :  1.961379121879653e-06
23651 :  1.5508592367929797e-06
63263 :  8.952175237052846e-07
4e-06 , 1000
951 :  2.1916360730048834
4e-06 , 10000
698 :  7.105193279629517
2350 :  3.132617160725437
```

```
2372 :  0.33272213613072427
3508 :  0.00810116801898211
4e-06 , 100000
299 :  9.470890896379938
872 :  0.6791393681598972
2249 :  0.07188028591840545
3808 :  0.01621595754591354
17465 :  0.002264290856603659
24967 :  0.0014022668317723527
34691 :  0.0004492598248556017
96765 :  0.00036432137097822874
Means:  [[ 7.36623395e-06  6.32088981e-06 -4.73752073e-03]
 [-1.09132093e-05  4.85371127e-08  3.34429051e-03]
 [-9.93372214e-04 -1.45911542e-03  1.00000000e+00]]
Standard deviations:  [[6.01523885e-06 2.20655642e-05 3.88449464e-03]
 [2.02328503e-05 2.53163984e-06 5.40354427e-03]
 [4.44793186e-03 5.60375344e-03 0.00000000e+00]]
```

In [37]:
```
print("Mean to std ratio: ", np.std(allFs, axis=0)/np.mean(allFs, axis=0))
```

```
Mean to std ratio:  [[ 0.81659623  3.49089525 -0.81994251]
 [-1.85397802 52.15884701  1.61575206]
 [-4.47760849 -3.84051416  0.        ]]
```

From mean to std ratio, it seems the standard deviation is far off from the mean most of the time, so the computation of the fundamental matrix is not stable.

3. Calibrated Structure From Motion

Download the calibration images provided (sfm.zip). Modify the provided camera calibration script **calibrate.py** as needed and use it to estimate the intrinsic parameter matrix K and enter the results below.

In [24]:
```
K = np.array([[1.66278079e+03, 0.00000000e+00, 8.38090758e+02],
 [0.00000000e+00, 1.66268979e+03, 6.06617825e+02],
 [0.00000000e+00, 0.00000000e+00, 1.00000000e+00]])
```

3.1 Estimate the Essential Matrix

Write code which loads in the pair of test images, use your SIFT matching routine to return a set of matching points.

Normalize these point coordinates using K^{-1} and then modify your code from Problem 2 to recover the essential matrix E.

Remember that you can still use the 8 point algorithm but now you will need to set the two singular values to be equal instead of just zeroing out the smallest singular value.

In [38]:
```
def getEssentialMatrix(pts2L,pts2R, K):

    if pts2L.shape[1] != pts2R.shape[1]:
        return np.zeros((3,3)), 99999

    numberOfPoints = pts2L.shape[1]

    pts2L = np.vstack((pts2L, np.ones((1, numberOfPoints))))
    pts2R = np.vstack((pts2R, np.ones((1, numberOfPoints))))

    invK = np.linalg.inv(K)

    pts2LN = invK @ pts2L
    pts2RN = invK @ pts2R

    A = np.zeros((numberOfPoints, 9))
    for i in range(numberOfPoints):
        A[i] = [pts2LN[0,i]*pts2RN[0,i], pts2LN[0,i]*pts2RN[1,i], pts2LN[0,i]*pts2RN[2,i
],
                pts2LN[1,i]*pts2RN[0,i], pts2LN[1,i]*pts2RN[1,i], pts2LN[1,i]*pts2RN[2,i
],
                pts2LN[2,i]*pts2RN[0,i], pts2LN[2,i]*pts2RN[1,i], pts2LN[2,i]*pts2RN[2,i
]]

    U, S, Vt = np.linalg.svd(A)
    E = Vt[-1].reshape(3,3)

    U, S, Vt = np.linalg.svd(E)
    S[-1] = 0
    S[0] = 1
    S[1] = 1
    E = U @ np.diag(S) @ Vt
    E = E/E[2,2]

    return E



img1 = opencv.imread('sfm/IMG_1434.JPG')
img1 = opencv.cvtColor(img1,opencv.COLOR_RGB2BGR)
gray1= opencv.cvtColor(img1,opencv.COLOR_BGR2GRAY)
dim = (0.4*np.asarray(gray1.shape)).astype(int)
gray1 = opencv.resize(gray1,(dim[1],dim[0]))

img2 = opencv.imread('sfm/IMG_1435.JPG')
img2 = opencv.cvtColor(img2,opencv.COLOR_RGB2BGR)
gray2= opencv.cvtColor(img2,opencv.COLOR_BGR2GRAY)
dim = (0.4*np.asarray(gray2.shape)).astype(int)
gray2 = opencv.resize(gray2,(dim[1],dim[0]))

pts1, pts2, kpts1, kpts2, good = getMatches(gray1, gray2)

E = getEssentialMatrix(pts1, pts2, K)

print(E)
```

```
[[  9.66992657 -84.03919155 -20.13431751]
 [ 82.96511082   9.44030306  30.02191571]
 [ 24.29313942 -25.3019361    1.        ]]
```

3.2 Camera Pose from the Esential Matrix

Using the SVD technique we described in class, recover the two possible rotation matrices R and translation vectors t associated with E. Discuss your results. Is the recovered translation reasonable based on what you see in the images?

In [29]:
```
def decomposeEssentialMatrix(E):
    Rz_pos = np.zeros((3,3))
    Rz_pos[0,1] = 1
    Rz_pos[1,0] = -1
    Rz_pos[2,2] = 1

    Rz_neg = np.zeros((3,3))
    Rz_neg[0,1] = -1
    Rz_neg[1,0] = 1
    Rz_neg[2,2] = 1

    U, S, Vt = np.linalg.svd(E)

    R1 = U @ Rz_pos @ Vt
    R2 = U @ Rz_neg @ Vt

    T1 = U @ Rz_pos @ S @ U.T
    T2 = U @ Rz_neg @ S @ U.T

    return R1, T1, R2, T2

R1, T1, R2, T2 = decomposeEssentialMatrix(E)

print("Possible rotations:")
print(R1)
print(R2)
print("Possible translations:")
print(T1)
print(T2)
```

```
Possible rotations:
[[ 0.99379725  0.10794079  0.02675481]
 [-0.10755828  0.99408028 -0.01534993]
 [-0.02825331  0.01237702  0.99952417]]
[[-0.80922241  0.04718777 -0.58560431]
 [ 0.24999428 -0.87436791 -0.41591299]
 [-0.53165963 -0.48296383  0.69576143]]
Possible translations:
[  60.94180659 -113.34765178  -10.24104166]
[-60.94180659 113.34765178  10.24104166]
```

In [35]:
```
import math

def rotationMatrixToEulerAngles(R) :
    sy = np.sqrt(R[0,0] * R[0,0] +  R[1,0] * R[1,0])
    singular = sy < 1e-6

    if  not singular :
        x = math.atan2(R[2,1] , R[2,2])
        y = math.atan2(-R[2,0], sy)
        z = math.atan2(R[1,0], R[0,0])
    else :
        x = math.atan2(-R[1,2], R[1,1])
        y = math.atan2(-R[2,0], sy)
        z = 0
    return np.array([x, y, z])

angles1 = rotationMatrixToEulerAngles(R1)
angles2 = rotationMatrixToEulerAngles(R2)

print(angles1*180/np.pi)
print(angles2*180/np.pi)
```

```
[ 0.70945222  1.619011   -6.17705573]
[-34.76650566  32.11765803 162.83243479]
```

In order to visualize the rotation matrix, I implemented a code from https://www.learnopencv.com/rotation-matrix-to-euler-angles/ (https://www.learnopencv.com/rotation-matrix-to-euler-angles/) and calculated the euler angles from the rotation matrix.

For this case, I assumed the first camera is in the origin with pointing towards positive z direction, Second rotation matrix (R2) has negative (clockwise rotation) around x axis, anti-clockwise around y and clockwise around z again. This makes sense from the test images, since in order to transform from the left to the right one, the left one is sort of a top view of the table whereas the right one is a right side view of the table.

In case of translation the first translation [60.94180659 -113.34765178 -10.24104166] makes sense, because it supports going on the right along x axis, down along y axis and towards positive z.