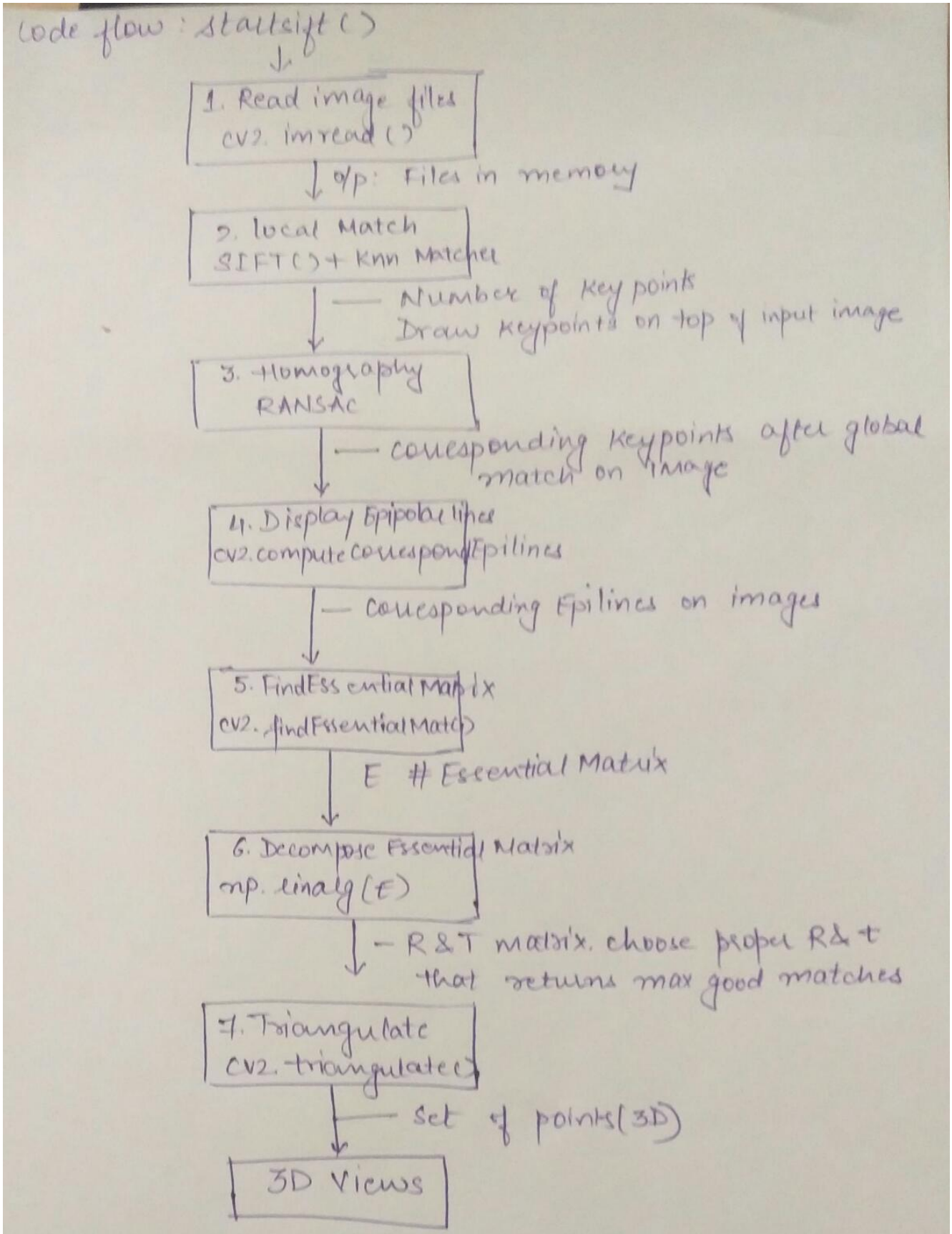
Table of Contents

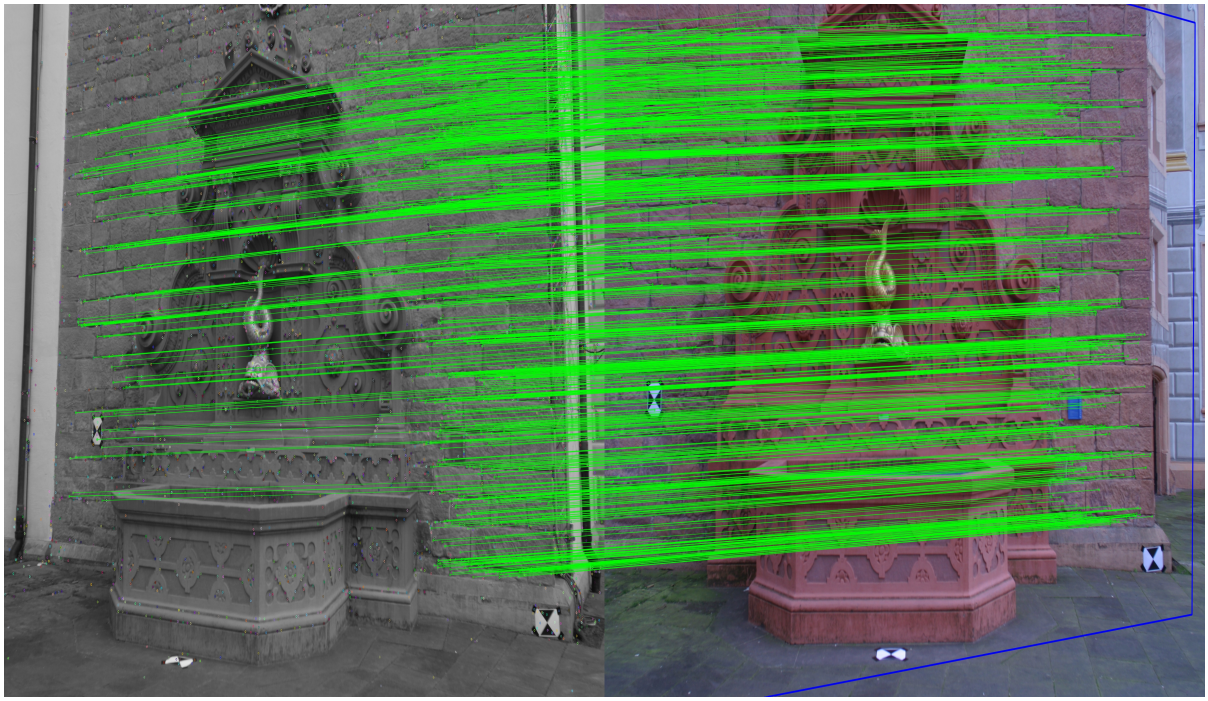

code flow : stallsift()
1. Read image files
cv2. imread()
o/p: Files in memory
2. local Match
SIFT() + Knn Matcher
— Number of key points
Draw keypoints on top of input image
3. Homography
RANSAC
— corresponding keypoints after global match on image
4. Display Epipolar lines
cv2. computeCorrespondEpilines
— corresponding Epilines on images
5. FindEssentialMatrix
cv2. findEssentialMat()
E # Essential Matrix
6. Decompose Essential Matrix
np. linalg(E)
— R&T matrix. choose proper R&t that returns max good matches
7. Triangulate
cv2. triangulatePoints()
— set of points(3D)
3D Views

Steps For Structure from Motion.

Intermediate Step Results

Key Points Found before applying homography

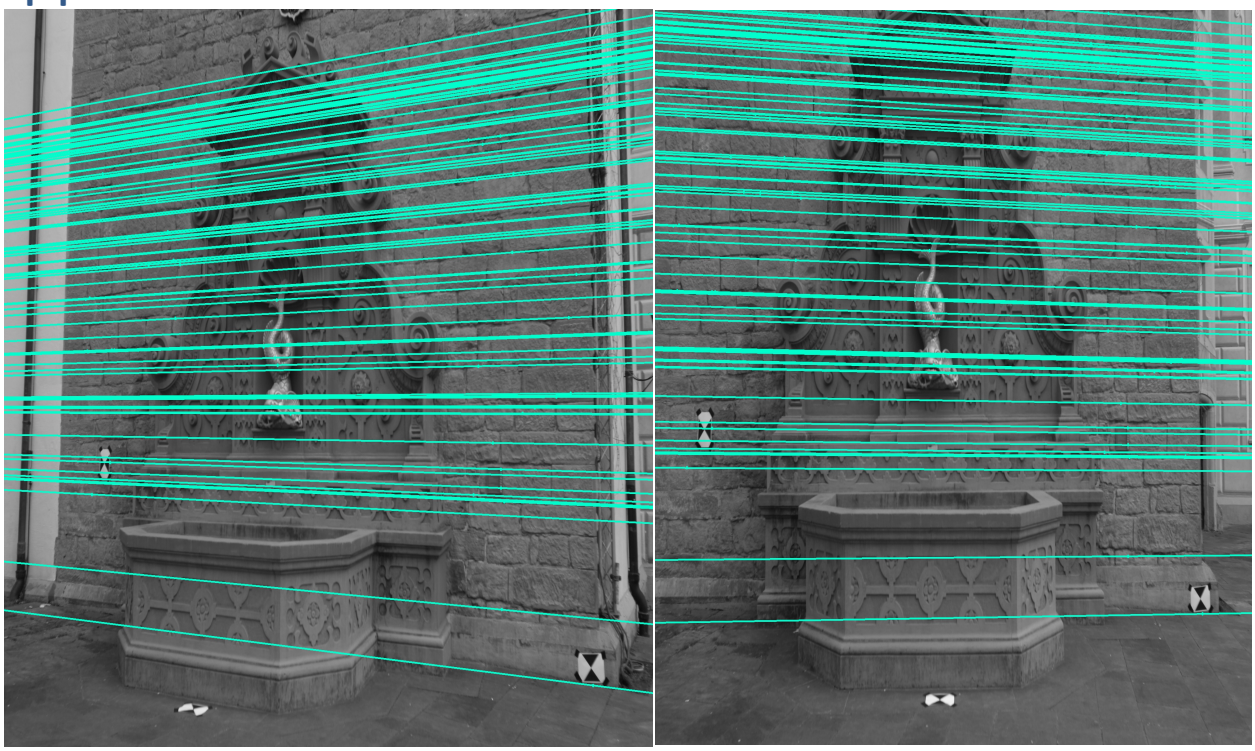
Key Points Match After Applying Homography



Number of KeyPoints

4617

Epipolar Lines



Essential Matrix

```
[ [ 0.06830577        0.53381523       -0.1486675   ]
  [-0.68842412       -0.01992625       -0.13473866]
  [-0.0545068         0.41682117       -0.13814262] ]
```

How to find the exact Rotation and Translation matrix?

So once we perform SVD on essential Matrix(E) using numpy function linalg such that we get three components such that $E = USV^T$

Rotation and Translation matrix may be given as any of the following 4 options:

Option 1
$R = U.dot(W).dot(V^T)$
$T = U[:, 2]$

Option 2
$R = U.dot(W).dot(V^T)$
$T = - U[:, 2]$

Option 3:
$R = U.dot(W.T).dot(V^T)$
$T = U[:, 2]$

Option 4:
$R = U.dot(W.T).dot(V^T)$
$T = - U[:, 2]$

On comparing the good matches found with all four above mentioned options, by plotting them on 3D visualization graph, Option 2 was the best available.

Rotation Matrix

```
[[-0.88668161        0.22582744     0.40348196  ]
 [-0.05159609       -0.91549294     0.39901194  ]
 [-0.45949273       -0.33297846    -0.82340256 ]]
```

Translation Matrix

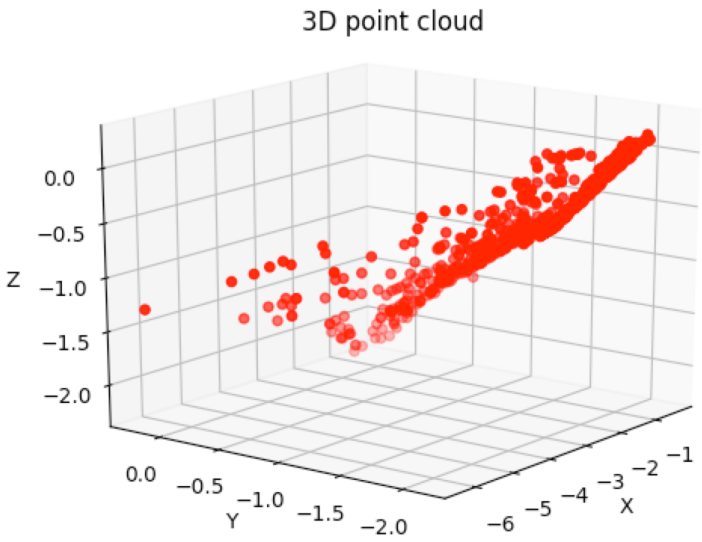
```
[[ 0.61363441  0.12264312  -0.78000736]]
```

Rotation and Translation Matrix stacked together – Horizontally.

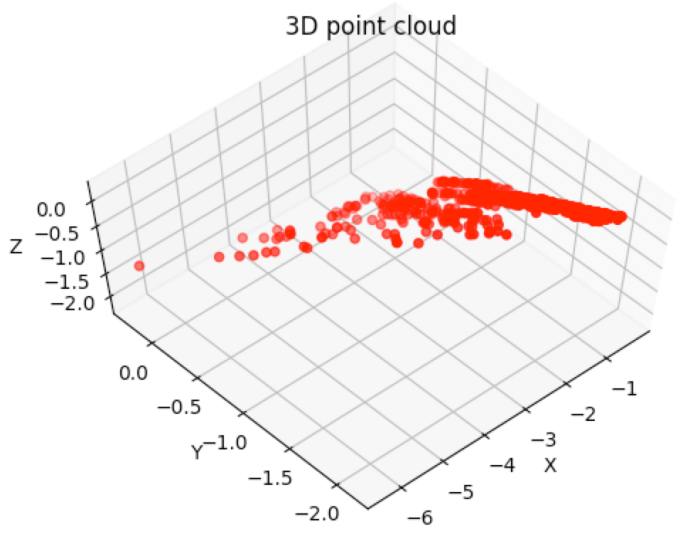
```
[[-0.88668161        0.22582744     0.40348196     0.61363441]
 [-0.05159609       -0.91549294     0.39901194     0.12264312]
 [-0.45949273       -0.33297846    -0.82340256    -0.78000736]]
```

3D Plots

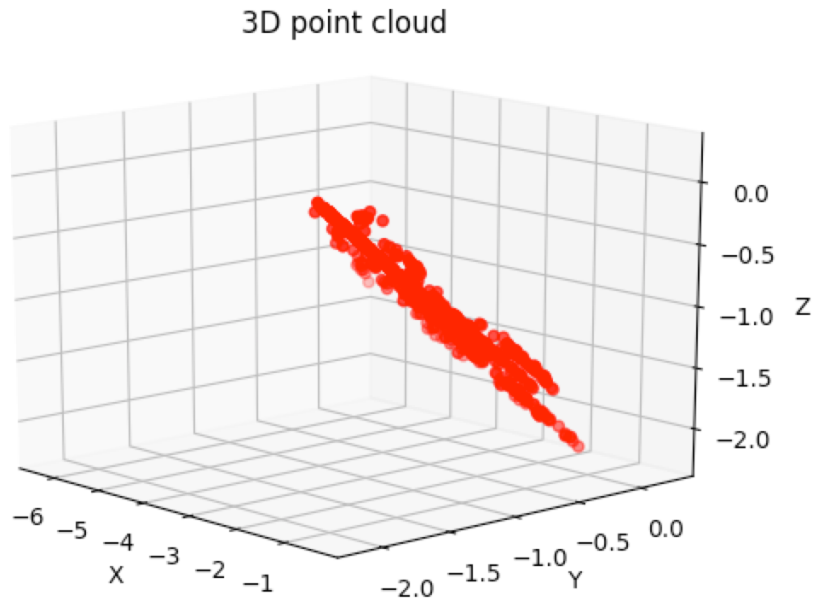
Right side Camera (View 1)



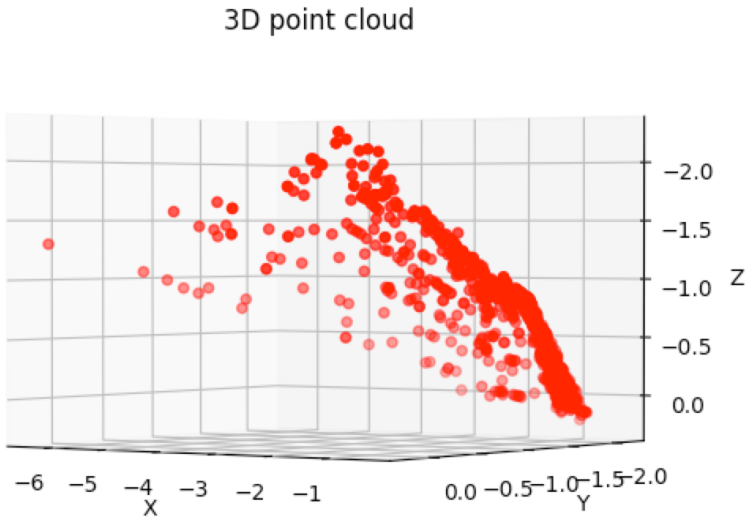
Right Side Camera (View 2)



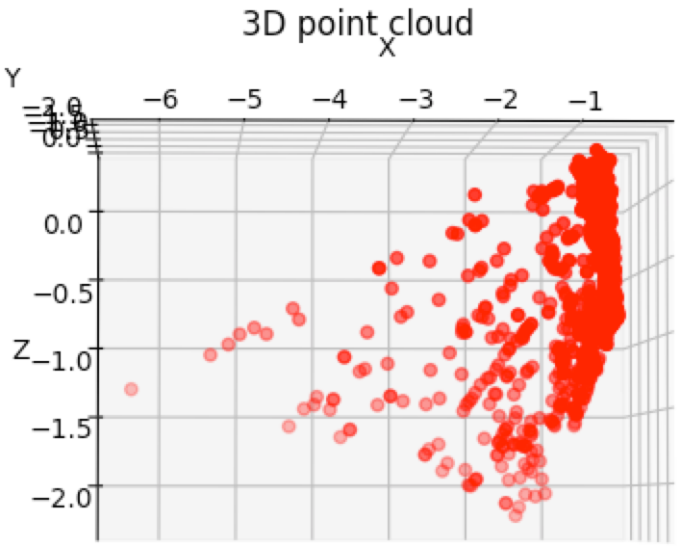
Left Side Camera (View 1)



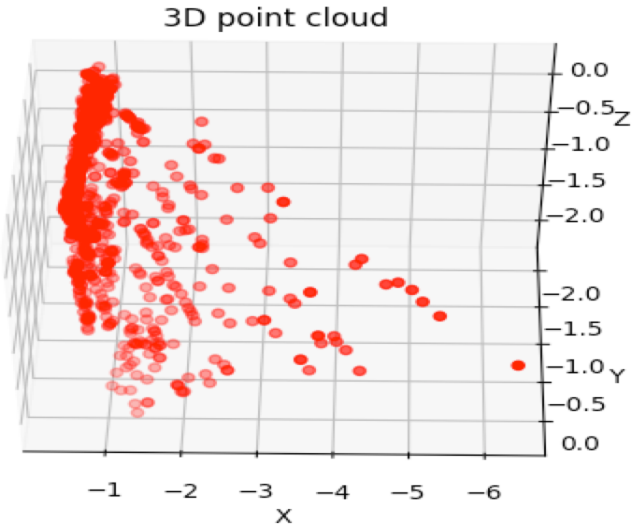
Left Side Camera (View 2)



Top View



Center View



My Comments

The algorithm performs pretty well. The points to support this statement are

- Clearly most of the points that lie on the same plane, namely the wall behind the fountain, and that the fountain itself extends from that wall in negative *z* coordinates and shown in the diagrams.
- Similar *x* coordinates corresponds to the wall behind the fountain
- Between *z* coordinates from *-0.5* to *-1.0*, the *x* coordinate is significantly different. almost act as outliers, which shows different keypoints that belong to the surface of the fountain.

References Used

1. Wikipedia.com
2. OpenCV official website.
3. OpenCV: Computer Vision Projects with Python by Joseph Howse
4. github.com to find utilities to plot 3D points using matplotlib

Code (with step wise comments)

```
__author__ = 'deepika'
#https://github.com/FantasyChen/LinearSFM/blob/30f38eca86cd28544829160d9a1249ea43c2c84f/SIFT_matching.py

import cv2
import matplotlib
matplotlib.use('TKAgg')

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

#Define global variables here
Fmask = None
F = None
focal=2760/4
principle=(1520/4, 1006/4)
K = np.matrix([[focal, 0, principle[0]], [0, focal, principle[1]], [0, 0, 1]])
K_inv = np.linalg.inv(K)

def to_gray(color_img):
    gray = cv2.cvtColor(color_img, cv2.COLOR_BGR2GRAY)
    return gray
```

```
def gen_sift_features(gray_img):
    sift = cv2.xfeatures2d.SIFT_create()
    kp, desc = sift.detectAndCompute(gray_img, None)
    return kp, desc

def read_images():
    return cv2.imread('image1.jpg'), cv2.imread('image2.jpg')

def run_knn_matcher(kp1, desc1, kp2, desc2):
    bf = cv2.BFMatcher()  # Brute-Force Matcher (FLAAN matcher is an alternative)
    matches = bf.knnMatch(desc1, desc2, k=2)
    good = []
    pts1 = []
    pts2 = []
    for m, n in matches:
        if m.distance < 0.7 * n.distance:
            good.append(m)
            pts2.append(kp2[m.trainIdx].pt) #Right
            pts1.append(kp1[m.queryIdx].pt) #left
    pts1 = np.float32(pts1)
    pts2 = np.float32(pts2)
    return pts1, pts2, good

def auto_points(kp1, desc1, kp2, desc2):
    bf = cv2.BFMatcher()  # Brute-Force Matcher (FLAAN matcher is an alternative)
    matches = bf.knnMatch(desc1, desc2, k=2)
    good = []
    for m, n in matches:
        if m.distance < 0.7 * n.distance:
            good.append((m.trainIdx, m.queryIdx))

    #naxN = max([x[1] for x in good])
    auto_pts1 = np.zeros((1,len(good),2))
    auto_pts2 = np.zeros((1,len(good),2))
    print "Initially: ", auto_pts1.shape, "kp1: ", len(kp1), "kp2: ", len(kp2)

    for idx in range(len(good)):
        match = good[idx]
        if match[1] < len(kp2):
            auto_pts1[0, idx, :] = kp1[match[0]].pt
            auto_pts2[0, idx, :] = kp2[match[1]].pt

    return auto_pts1, auto_pts2
```

```
def apply_homography(src_pts, dst_pts, matches, source, source_kp, destination,
destination_kp):

    H, status =  cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 4.0)
    matchesMask = status.ravel().tolist() #convert numpy nd.array to List

    h, w = source.shape[0], source.shape[1]
    pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2)
    dst = cv2.perspectiveTransform(pts, H)
    destination = cv2.polylines(destination, [np.int32(dst)], True, 255, 3, cv2.LINE_AA)

    drawParameters = dict(matchColor=(0, 255, 0), singlePointColor=None,
matchesMask=matchesMask, flags=2)

    img3 = cv2.drawMatches(
        source, source_kp,
        destination, destination_kp,
        matches, None, **drawParameters)

    return img3

def find_essential_matrix(pts1, pts2, focal, principle):
    return cv2.findEssentialMat(pts1, pts2, focal, principle)

def decompose_essential_matrix(ess_matrix, matches, first_key_points,
second_key_points):

    first_match_points = np.zeros((len(matches), 2), dtype=np.float32)
    second_match_points = np.zeros_like(first_match_points)
    for i in range(len(matches)):
        first_match_points[i] = first_key_points[matches[i].queryIdx].pt
        second_match_points[i] = second_key_points[matches[i].trainIdx].pt

    match_pts1 = first_match_points
    match_pts2 = second_match_points

    global Fmask, K_inv
    U, S, Vt = np.linalg.svd(ess_matrix)
    W = np.array([0.0, -1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
                  1.0]).reshape(3, 3)
```

```
    # iterate over all point correspondences used in the estimation of the
    # fundamental matrix
    first_inliers = []
    second_inliers = []
    for i in range(len(Fmask)):
        if Fmask[i]:
            # normalize and homogenize the image coordinates
            first_inliers.append(K_inv.dot([match_pts1[i][0],
                                match_pts1[i][1], 1.0]))
            second_inliers.append(K_inv.dot([match_pts2[i][0],
                                 match_pts2[i][1], 1.0]))
    # option 1
    #R = U.dot(W).dot(Vt)
    #T = U[:, 2]

    # option 2
    R = U.dot(W).dot(Vt)
    T = - U[:, 2]

    # option 3:
    #R = U.dot(W.T).dot(Vt)
    #T = U[:, 2]

    # option 4:
    #R = U.dot(W.T).dot(Vt)
    #T = - U[:, 2]

    match_inliers1 = first_inliers
    match_inliers2 = second_inliers
    Rt1 = np.hstack((np.eye(3), np.zeros((3, 1))))
    Rt2 = np.hstack((R, T.reshape(3, 1)))

    return match_inliers1, match_inliers2, Rt1, Rt2

def draw_lines(img_left, img_right, lines, pts_left, pts_right):
    h,w = img_left.shape[0],img_left.shape[1]
    img_left = cv2.cvtColor(img_left, cv2.COLOR_GRAY2BGR)
    img_right = cv2.cvtColor(img_right, cv2.COLOR_GRAY2BGR)

    for line, pt_left, pt_right in zip(lines, pts_left, pts_right):
        x_start,y_start = map(int, [0, -line[2]/line[1] ])
        x_end,y_end = map(int, [w, -(line[2]+line[0]*w)/line[1] ])
        #color = tuple(np.random.randint(0,255,2).tolist())
```

```
        color = (204,255,0)
        #print "Value of color: ", color
        cv2.line(img_left, (x_start,y_start), (x_end,y_end), color,3)
        cv2.circle(img_left, tuple(pt_left), 5, color, -1)
        cv2.circle(img_right, tuple(pt_right), 5, color, -1)

    return img_left, img_right

def display_epipolar_lines(pts_left_image, pts_right_image, img_left, img_right):

    print "Initialized F"
    global F, Fmask
    F, Fmask = cv2.findFundamentalMat(pts_left_image, pts_right_image,
cv2.FM_LMEDS)
    take_top = 100
    # Selecting only the inliers
    pts_left_image = pts_left_image[Fmask.ravel()==1]
    pts_right_image = pts_right_image[Fmask.ravel()==1]

    lines1 = cv2.computeCorrespondEpilines(pts_right_image.reshape(-1,1,2), 2, F)
    lines1 = lines1.reshape(-1,3)
    lines1 = lines1[0:take_top]
    img_left_lines, img_right_pts = draw_lines(img_left, img_right, lines1,
pts_left_image, pts_right_image)

    # Drawing the lines on right image and the corresponding feature points on the left
image
    lines2 = cv2.computeCorrespondEpilines(pts_left_image.reshape(-1,1,2), 1,F)
    lines2 = lines2.reshape(-1,3)
    lines2 = lines2[0:take_top]
    img_right_lines, img_left_pts = draw_lines(img_right, img_left, lines2,
pts_right_image, pts_left_image)

    cv2.imwrite('Epi lines on left image.jpg',img_left_lines)
    cv2.imwrite('Feature points on right image.jpg',img_right_pts)
    cv2.imwrite('Epi lines on right image.jpg',img_right_lines)
    cv2.imwrite('Feature points on left image.jpg',img_left_pts)

    print "=------Done with epipole lines=-----"

def startSift():

    #Step 1:
```

```
    #       Read the image files and find SIFT features.
    source, destination = read_images()
    print "Read imaged: ", type(source), source.shape

    #Step 2:
    #       Figure out local matches
    source_gray = to_gray(source)
    destination_gray = to_gray(destination)

    source_kp, source_desc = gen_sift_features(source_gray)
    destination_kp, destination_desc = gen_sift_features(destination_gray)
    print "Number of keyPoints found: ", len(source_kp)

    img = cv2.drawKeypoints(source_gray, source_kp, source)
    cv2.imwrite('KeyPointsMatchBeforeHomography.jpg',img)

    src_pts, dst_pts, matches = run_knn_matcher(source_kp, source_desc,
destination_kp, destination_desc)

    #Step 3:
    #       Run for global matches. That is homography
    global_match_img = apply_homography(src_pts, dst_pts, matches, source,
source_kp, destination, destination_kp)

cv2.imwrite('KeyPointsMatchAfterApplyingHomography.jpg',global_match_img)

    #Step 4:
    #       Display Epipolar lines
    print "-----------starting Epipolar lines"
    display_epipolar_lines(src_pts, dst_pts, source_gray, destination_gray)

    #Step 5:
    #       Find Essential Matrix
    ess_matrix, mask = find_essential_matrix(src_pts, dst_pts, focal, principle)
    print "Essential Matrix from openCv function: ", ess_matrix

    #Step 6:
    #       Decompose Essential Matrix
    match_inliers1, match_inliers2, Rt1, Rt2 = decompose_essential_matrix(ess_matrix,
matches, source_kp, destination_kp)
    print "Step 5 results:"
    print "Rt1: ", Rt1
    print "Rt2: ", Rt2
```

```
    #Step 7:
    #         Triangulate
    first_inliers = np.array(match_inliers1).reshape(-1, 3)[:, :2]
    second_inliers = np.array(match_inliers2).reshape(-1, 3)[:, :2]
    pts4D = cv2.triangulatePoints(Rt1, Rt2, first_inliers.T, second_inliers.T).T

    # convert from homogeneous coordinates to 3D
    pts3D = pts4D[:, :3]/np.repeat(pts4D[:, 3], 3).reshape(-1, 3)

    # plot with matplotlib
    Ys = pts3D[:, 0]
    Zs = pts3D[:, 1]
    Xs = pts3D[:, 2]

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(Xs, Ys, Zs, c='r', marker='o')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.title('3D point cloud')
    plt.show()
    raw_input(">Hit Enter To Close")
    plt.close()

startSift()
```