Keerthi Raj Nagaraja (nagaraj1@purdue.edu)

11-25-2014

1 Projective Reconstruction

In this assignment, we use the concept of Fundamental matrix and stereo reconstruction to reconstruct a 3D scene using point correspondences between two 2D images of the same scene captured by an uncalibrated camera. The steps for reconstruction includes, estimating the Fundamental matrix and camera projection matrices using manual set of correspondences, optimizing them using non-linear least squares optimization (Levenberg-Marquadt optimization). We then rectify the images using hartley's rectification algorithm. The interest points or salient edges in two rectified images are obtained using Canny edge detector and are projected back to world 3D to reconstruct the shapes in the images. These are described within two broad-level tasks - Image Rectification and 3D Reconstruction.

1.1 Image Rectification

The goal of the image rectification is to make the epipolar lines in the two images parallel to image x-axis so that the 2D search for correspondences between two images is reduced to search in 1D (1 row ideally). We proceed to do that with following steps

(a) Estimate the fundamental matrix (F) using the 12 (minimum of 8) correspondences selected manually. Let a correspondence be $(\vec{x}_i, \vec{x}'_i)$, then

$$\vec{x'}_i^T F \vec{x}_i = 0$$

$$\begin{bmatrix} x'x & x'y & x' & y'x & y'y & y' & x & y & 1 \end{bmatrix} \vec{f} = 0$$

If N correspondences are considered then, to find F, we need to solve for

$$A\vec{f} = 0$$

where $\vec{f}$ contains elements of F and A is $N \times 9$ matrix . This is solved by taking SVD of A and equating $\vec{f}$ to the last row of V, if $A = UDV^T$. The Fundamental matrix F hence found may not be of rank 2. We force F to be a rank 2 matrix by taking SVD of $F(F = U_f D_f V_f^T)$ and assigning the last singular value in D_f to zero and recompute as $F = U_f D'_f V_f^T$

(b) Compute the epipoles $(\vec{e}, \vec{e'})$ of the two images using the fundamental matrix F. $\vec{e}$ and $\vec{e'}$ are the right and left null vectors of F respectively. Through SVD decomposition of $F(F = UDV^T)$ we can find these. $\vec{e}$ is the last row of V and $\vec{e'}$ is the last column of U.

(c) Compute the camera projection matrices P and P' for image 1 and image 2 respectively. These represent the relationship between pixel coordinates and world

3D coordinates. Since we assume the canonical configuration of the two cameras, the camera 1 is located at world origin. Therefore, we set

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$P' = \left[sF | \vec{e'} \right]$$

where $S = \begin{bmatrix} 0 & -e'_3 & e'_2 & e'_1 \\ e'_3 & 0 & -e'_1 & e'_2 \\ -e'_2 & -e'_1 & 0 & e'_3 \end{bmatrix}$

(d) Refine P' using the Levenberg-Marquardt non-linear optimization method. The goal is to minimize the geometric error between the image points and the re-projected world-points to image plane. Mathematically, given a correspondence $(\vec{x}, \vec{x'})$ and a pair of cameras (P, P'), we want to find the best value of P'(Since we are fixing P, only P' is optimized) such that $\vec{x} = PX$ and $\vec{x'} = P'X$. If we express these in homogeneous system of equations, then $AX = 0$ where

$$A = \begin{bmatrix} x\vec{p_3}^T - \vec{p_1}^T \\ y\vec{p_3}^T - \vec{p_2}^T \\ x'\vec{p'_3}^T - \vec{p'_1}^T \\ y'\vec{p'_3}^T - \vec{p'_2}^T \end{bmatrix}$$

and X is the world point in homogeneous coordinate for the correspondence $(\vec{x}, \vec{x'})$. This method is called the triangulation method. The world point is reprojected back to image planes using P and P' to obtain estimates $(\hat{\vec{x}}, \hat{\vec{x'}})$. Now the goal of LM is to minimize

$$d^2_{geom} = \sum_i (||\vec{x_i} - \hat{\vec{x_i}}||^2 + ||\vec{x'_i} - \hat{\vec{x'_i}}||^2)$$

Once we find optimized P', we calculate optimized e' which will be used further in the image rectification.

(e) Find homographies H_1 and H_2 which rectify the images 1 and 2 respectively. Rectification involves making the epipolar lines parallel to x-axis and this is done through a homography which takes the epipole in an image to infinity. The basic idea involved in the procedure is described below:

i. Suppose that the estimated epipole e' is (u, v, w) in homogeneous coordinates. The first step is to perform a rotation on the image plane to bring (u, v, w) to a point $(u', 0, w')$ on the x-axis of the image plane. However, for this to work the origin of the image coordinate system should be at the centre of the image. So, we translate the image through translation T. Next, we find the 3×3 rotation matrix R that makes the second component of the epipole e' vanish: $R(u, v, w)^T = (u', 0, w')^T$. Such a matrix is given by

$$R = \begin{bmatrix} \frac{e'_1}{d} & \frac{e'_2}{d} & 0 \\ -\frac{e'_2}{d} & \frac{e'_1}{d} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where, $d = \sqrt{(\frac{e'_1}{e'_3})^2 + (\frac{e'_2}{e'_3})^2}$

ii. The next step is then to find a 3×3 transformation matrix G that would transform $(u', 0, w')$ to a point at infinity along the x-axis, i.e. find a matrix G such that $G(u', 0, w')^T = (u'', 0, 0)^T$. Such a matrix is given by

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{e'_3}{e'_1} & 0 & 1 \end{bmatrix}$$

iii. The epipole thus obtained will be at infinity. However, due to effect of translation T, all the pixels are now referred with respect to center of the image. To nullify the effect of transformation T, we include another transformation T_2 which brings back all the pixels with respect to the image origin. Finally, The 3×3 homography H_2 that rectifies image 2 is then the product of T_2, G, R and T. Therefore,

$$H_2 = T_2 G R T$$

iv. Now since we know H_2, we can find H_1 such that the distance between rectified point in image 2 is close to corresponding rectified point of image 1. Since we know that the rectification already makes sure the corresponding pixels are in same row (height), we just need to reduce the distance in terms width (column). This is done by first taking a correspondence $(\vec{x}, \vec{x'})$ and finding their rectified set points $(\hat{\vec{x}}, \hat{\vec{x'}})$ using the equations

$$\hat{\vec{x}} = H_0 H_2 P' P^{\dagger} \vec{x}$$

$$\hat{\vec{x'}} = H_2 \vec{x'}$$

Where, H_0 is the homography between rectified image 2 to rectified image 1. The goal is to find H_0 which minimizes $||\hat{\vec{x}} - \hat{\vec{x'}}||^2$ or $||H_0 H_2 P' P^{\dagger} \vec{x} - \hat{\vec{x'}}||^2$. Given that we know everything except H_0, this can be posed as minimization of

$$||H_0 \vec{m} - \hat{\vec{x'}}||^2$$

and H_0 is forced to be of the form $\begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ since this homography shouldn't change the row number of the point as it would nullify the advantage of rectification. Since there are 3 unknown variables (a, b, c) and we have 12 manually obtained correspondences, this can be solved using linear least squares method. Finally, H_1 is calculated as

$$H_1 = H_0 H_2 P' P^{\dagger}$$

(f) As a final step, we use H_1 and H_2 to obtain two rectified images from the original images. These will now be used for 3D reconstruction.

1.2 3D Reconstruction

The steps involved in 3D reconstruction are explained below:

(a) First, we need to find large number of interest points in the rectified images which represents edges or corners so that the reconstructed figure can have sufficient information in order to make sense of any shapes in 3D. For this, we use Canny edge detection to obtain binary edge-images corresponding to each rectified image. Parameters are Canny Low Threshold = 70, Canny Low Threshold = 140 and kernel size = 3.

(b) Each pixel in binary edge-image whose value is 1 is considered as an interest point and the corresponding interest point in second rectified image is searched within +/- 3 rows of the current pixel in first image (Not the whole image, thanks to rectification). The best correspondence is the one which has maximum NCC value and NCC threshold of 0.93 is used to eliminate false correspondences. The NCC metric has been explained in detail in Homework 4.

(c) Once large number of correspondences $(\vec{x}, \vec{x'})$ are found in rectified images, we project these back to world coordinates using the triangulation method explained in previous section and obtain 3D coordinates.

(d) These 3D points are plotted in 3D plot figures using *matplotlib* library in python.

2 Observations

(a) Observed that the image rectification is dependent on careful selection of the manual correspondences and a bad set of input points may result in bad rectification. It also depends on number of correspondences.

(b) SURF/SIFT feature extraction has been tried but led to fewer interest points and hence bad 3D reconstruction since its difficult to extract 3D structure with less points. However, SURF/SIFT method is faster than Edge-based NCC method.

(c) The rectification led to correspondences in two images lying within +/- 3 rows.

(d) LM optimization improves P' very well. I tried finding H_1 and H_2 before LM and it didn't rectify images at all for many datasets.

3 Results

Initial Fundamental Matrix: $F = \begin{bmatrix} 0. & 0. & -0.00000751 \\ -0. & 0. & -0.00031715 \\ 0.00002377 & 0.00026122 & -0.99673064 \end{bmatrix}$

Initial Epipole 1 : $e1 = \begin{bmatrix} -469342.90762852 \\ 46515.12238158 \\ 1. \end{bmatrix}$

Initial Epipole 2: $e2 = \begin{bmatrix} -191691.43151845 \\ 1398.69726226 \\ 1. \end{bmatrix}$

Projection Matrix, P1: $P1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

Initial Projection Matrix, P2: $P2 = \begin{bmatrix} 0.03324034 & 0.36537021 & -1394.12409809 & -191691.43151845 \\ 4.55558732 & 50.0739803 & -191064.72293425 & 1398.69726226 \\ 0.00012727 & -0.00002303 & 60.80624194 & 1. \end{bmatrix}$

Optimized Projection Matrix, P2: $P2 = \begin{bmatrix} -0.00009638 & 0.05683195 & -5751.40317238 & -1554.01975257 \\ 0.01574014 & 0.14679969 & 130.38886741 & 36.54653789 \\ 0.00005128 & -0.00004527 & 3.87252536 & 1. \end{bmatrix}$

Optimized Epipole 2:

$e2 = \begin{bmatrix} -1554.01975257 \\ 36.54653789 \\ 1. \end{bmatrix}$

Rectification Homography 2: $H2 = \begin{bmatrix} 1.20193367 & 0.11838895 & -110.02226135 \\ 0.01795715 & 1.00660805 & -8.88228047 \\ 0.00050427 & 0.00004967 & 0.78182632 \end{bmatrix}$

Epipole forced to infinity by multiplying H2: $e2 = \begin{bmatrix} 1973.52421584 \\ -0. \\ -0. \end{bmatrix}$

Rectification Homography 1: $H1 = \begin{bmatrix} 0.16390542 & -0.00307337 & -4.69065735 \\ 0.01551493 & 0.14919411 & -6.00540486 \\ 0.00004083 & 0.00000056 & 0.13387323 \end{bmatrix}$

(a) Image 1

(b) Image 2

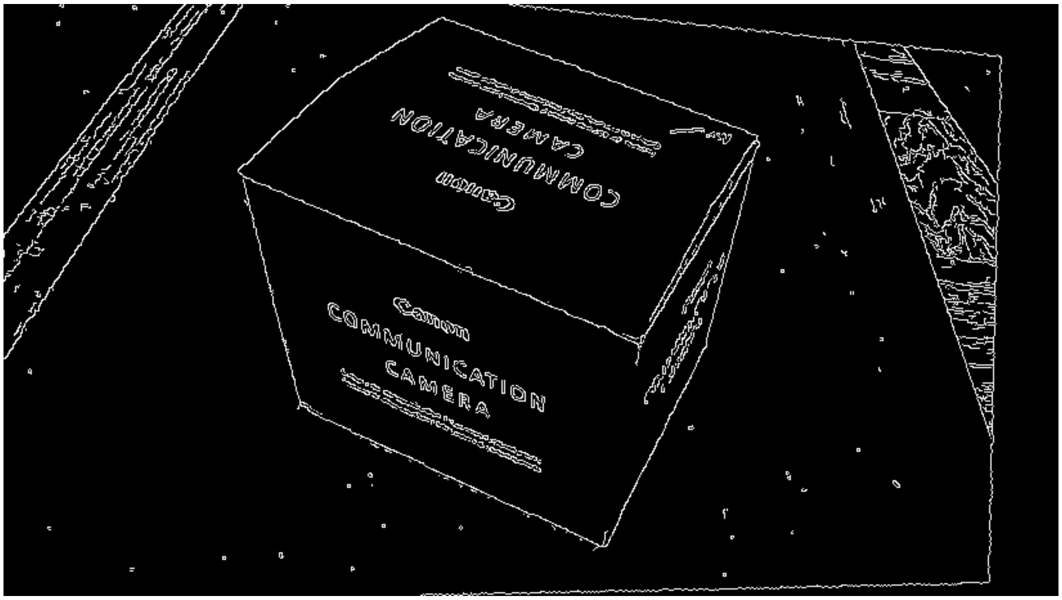
Figure 1: Input Images - With 12 correspondences chosen manually shown as blue points

Points Chosen: [(188, 144)–(168, 116)] [(336, 16)–(344, 10)] [(597, 92)–(571, 110)] [(497, 261)–(401, 259)] [(473, 425)–(407, 421)] [(236, 315)–(222, 282)] [(555, 261)–(539, 277)] [(314, 286)–(271, 263)] [(421, 316)–(352, 305)] [(300, 98)–(275, 84)][(424, 110)–(385, 108)] [(443, 130)–(395, 130)]

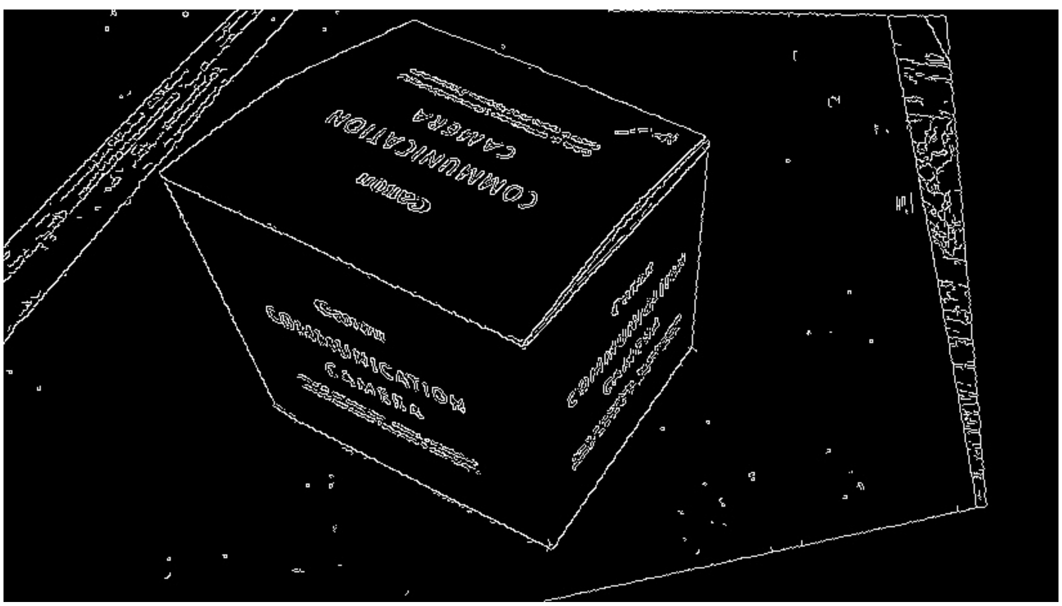
(a) Rectified Image 1

(b) Rectified Image 2

Figure 2: Rectified Images



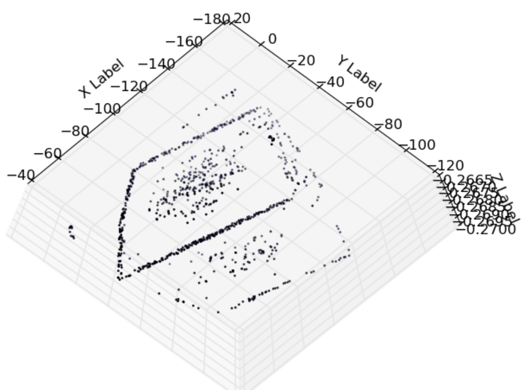
(a) Rectified Edge Image 1



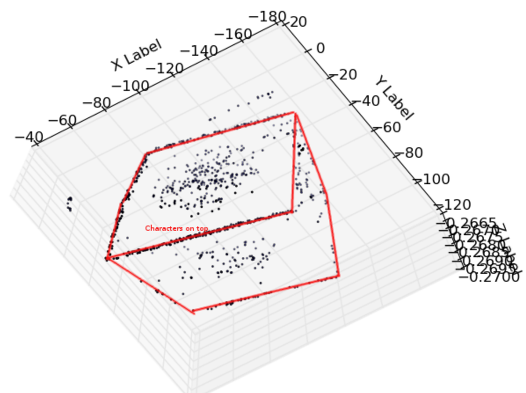
(b) Rectified Edge Image 2

Figure 3: Rectified Edge Images

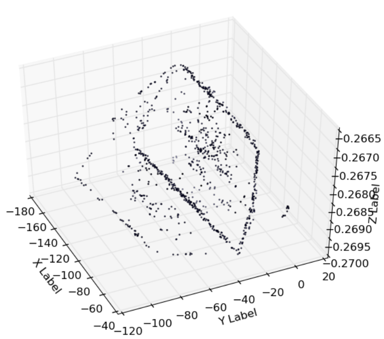
(a) Matched Correspondences between two rectified images



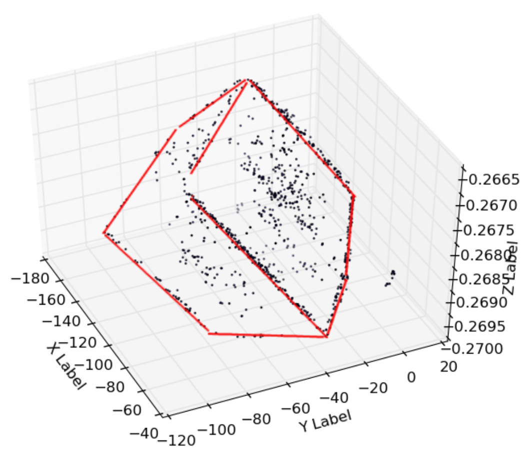
(b) 3D Image - View 1



(c) 3D Image - View 1 - With Markers

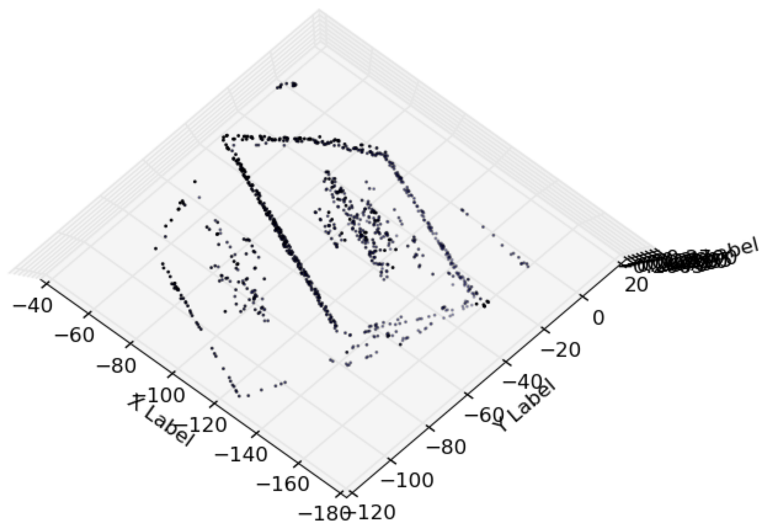


(d) 3D Image - View 2

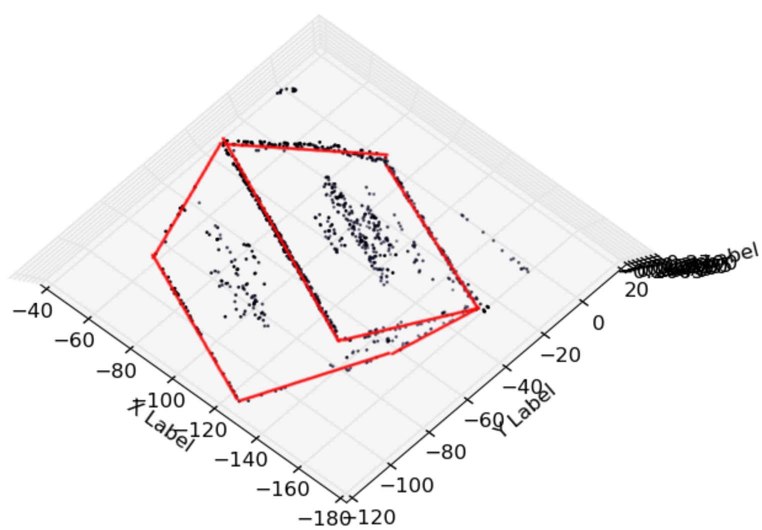


(e) 3D Image - View 2 - With Markers

Figure 4: 3D Plots



(a) 3D Image - View 3



(b) 3D Image - View 3 - With Markers

Figure 5: 3D Plots

4 Appendix

4.1 ImageRectification.py

```python
# Importing libraries
import cv2
import cv
from math import *
from numpy import *
#from sympy import Symbol,cos,sin
from operator import *
from numpy.linalg import *
import time
import ctypes
from scipy.optimize import leastsq
from matplotlib import pyplot as plt
# Prints the numbers in float instead of scientific format
set_printoptions(suppress=True)

filename='Dataset 5/' # Dataset Used
#-----------------------------------------------------------------------------------------------
#This function reads the manual correspondences saved in a text file.
def readmatches(filename):
    f = open(filename).read()
    rows = []
    for line in f.split('\n'):
        rows.append(line.split('\t'))
    rows.pop()
    for loopVar1 in range(0, len(rows)):
        for loopVar2 in range(0, len(rows[loopVar1])):
            rows[loopVar1][loopVar2]=float(rows[loopVar1][loopVar2])
    return rows

#----------------------------------------------------------------------------------------------#
#This function saves the homographies to a text file
def save_matrix(filename, H):
    fo = open(filename, 'w', 0)
    for loopVar1 in range(H.shape[0]):
        for loopVar2 in range(H.shape[1]):
            fo.write(str(H[loopVar1, loopVar2]))
            if loopVar2!=H.shape[1]-1:
                fo.write('\t')
        if loopVar1!=H.shape[0]-1:
            fo.write('\n')
    fo.close()

#-----------------------------------------------------------------------------------------------
# This function takes in P1, P2, F and epipoles to find H1 and H2
def Rectify_Images(correspondences, F, P1, P2, e1, e2, H1, H2):
    img_1 = cv2.imread(filename+'Pic_1.jpg',1) # Read two images
    img_2 = cv2.imread(filename+'Pic_2.jpg',1)
    image_width=img_1.shape[1]          # Get their size
```

```
    image_height=img_1.shape[0]

    G=matrix(identity(3))                # Initialize required matrices
    R=matrix(zeros((3,3)))
    T=matrix(identity(3))
    H2=matrix(zeros((3,3)))

    T[0,2]=(-image_width/2)              # Calculate T
    T[1,2]=(-image_height/2)

    e2=T*e2                      # Find translated epipole
    mirror = e2[0,0] < 0

    d=sqrt(pow((e2[1,0]),2)+pow((e2[0,0]),2))   # Find distance ...
        between origin and epipole
    alpha=e2[0,0]/d                  # Cos theta
    beta=e2[1,0]/d                   # -Sin theta

    R[0,0]=alpha#cos(theta)              # Calculate R
    R[0,1]=beta#-sin(theta)
    R[1,0]=-beta#sin(theta)
    R[1,1]=alpha#cos(theta)
    R[2,2]=1

    e2=R*e2                      # Find rotated epipole

    f=e2[0,0]/e2[2,0]                # Calculate G
    G[2,0]=(-1/f)
    print f

    H2=G*R*T                     # Calculate H2
    print 'H2', H2
    print 'e2 after H2', G*e2            # Check Epipole after sending ...
        to infinity
    print H2*transpose(matrix([image_width/2, image_height/2, 1]))

    center_point = matrix(zeros((3,1)))     # Calculate T2
    center_point[0,0]=image_width/2
    center_point[1,0]=image_height/2
    center_point[2,0]=1
    new_center=H2*center_point

    #print center_point, new_center
    T2=matrix(identity(3))
    T2[0,2]=(image_width/2)-(new_center[0,0]/new_center[2,0])
    T2[1,2]=(image_height/2)-(new_center[1,0]/new_center[2,0])
    #print T2
    H2=T2*H2                     # Calculate H2 after correcting for T2

    if mirror:                   # Mirror H2 if epipole is along ...
        negative x-axis
        mm = array([[-1, 0, image_width],
                    [0, -1, image_height],
                    [0, 0, 1]], dtype=float)
```

```
            #H1 = mm.dot(H1)
            H2 = mm.dot(H2)

    print 'H2', H2
    print 'center after H2', H2*transpose(matrix([image_width/2, ...
        image_height/2, 1]))
    #----------------------------------------------------------------------------------#
    H_temp1=H2*P2*pinv(P1)              # Find Temporary H1
    A=[]
    b=[]
    for loopVar1 in range(len(correspondences)):    # For all ...
        correspondences, find A and b for least squares estimation of H0
        x1=[correspondences[loopVar1, 0], correspondences[loopVar1, ...
            1], 1]
        new_x1=H_temp1*transpose(matrix(x1))
        new_x1=asarray(new_x1/new_x1[2,0])
        x2=[correspondences[loopVar1, 2], correspondences[loopVar1, ...
            3], 1]
        new_x2=H2*transpose(matrix(x2))
        new_x2=asarray(new_x2/new_x2[2,0])
        #print new_x1, new_x2
        A.append([new_x1[0][0], new_x1[1][0], new_x1[2][0]])
        b.append(new_x2[0][0])
    h=linalg.lstsq(A, b)[0]             # Calculate H0
    #print h
    H_temp2=matrix(identity(3))
    H_temp2[0,0]=h[0]
    H_temp2[0,1]=h[1]
    H_temp2[0,2]=h[2]

    #print H_temp2
    H1=H_temp2*H_temp1              # Calculate H1
    #print H1

    center_point = matrix(zeros((3,1)))     # Calculate T2
    center_point[0,0]=image_width/2
    center_point[1,0]=image_height/2
    center_point[2,0]=1
    new_center=H1*center_point

    T2=matrix(identity(3))
    T2[0,2]=(image_width/2)-(new_center[0,0]/new_center[2,0])
    T2[1,2]=(image_height/2)-(new_center[1,0]/new_center[2,0])
    #print T2
    H1=T2*H1                    # Calculate H1 after correction for T2

    if mirror:                  # Mirror H1 if epipole is along ...
        negative x-axis
            mm = array([[-1, 0, image_width],
                        [0, -1, image_height],
                        [0, 0, 1]], dtype=float)
            #H1 = mm.dot(H1)
            #H2 = mm.dot(H2)
    print 'H1', H1
```

```
    cv2.imwrite(filename+'Pic_2_corrected.jpg', ...
        Apply_Homography(img_2, H2)) #Save the rectified images
    cv2.imwrite(filename+'Pic_1_corrected.jpg', ...
        Apply_Homography(img_1, H1))
    return 0

#--------------------------------------------------------------------------
# This function takes in an image and homography matrix and applies ...
    the given homography to produce new image
def Apply_Homography(image, H):
    #Declare two empty arrays for storing points while applying homography
    old_point=[]
    new_point=[]
    image_width=image.shape[1]  #width
    image_height=image.shape[0] #height
    inv_H=inv(H)
    print inv_H, 'inv_H'

    #----------------------------------------------------------------------
    #This section of code finds out minimum and maximum indices in ...
        both x and y.
    #Later, finds out the scaling factor to scale the final output image
    #Creates an empty output image with scaled-down dimensions
    min_x=0
    min_y=0
    max_x=0
    max_y=0

    a=image.shape[0]    #height
    b=image.shape[1]    #width
    corners=[[0, 0],[b, 0],[0, a],[b, a]]
    for loopVar1 in range(0,len(corners)):
        old_point=[[corners[loopVar1][0]],[corners[loopVar1][1]],[1]]
        print old_point
        new_point=H*old_point
        new_point=new_point*(1/new_point[2][0])
        old_point=array(old_point)
        new_point=array(new_point)
        if (loopVar1==0):
            min_x=new_point[0][0]
            min_y=new_point[1][0]
            max_x=new_point[0][0]
            max_y=new_point[1][0]
        else:
            if(new_point[0][0]<min_x):
                min_x=new_point[0][0]
            if(new_point[1][0]<min_y):
                min_y=new_point[1][0]
            if(new_point[0][0]>max_x):
                max_x=new_point[0][0]
            if(new_point[1][0]>max_y):
                max_y=new_point[1][0]
        print loopVar1
```

```

    print min_x
    print min_y
    print max_x
    print max_y

    min_x=0
    min_y=0
    b=image.shape[1]
    a=image.shape[0]
    scaling_x=1
    output_img = zeros((a,b,3), uint8) # Output Image with all pixels ...
        set to black
    #

    #
    #This loop applies inverse homography to all points in output ...
        image and gets original pixel coordinates.
    #Used Inverse transformation to avoid having empty pixels in the ...
        output image
    for loopVar1 in range(0,a):
        for loopVar2 in range(0,b):
            new_point=matrix([[(loopVar2*scaling_x)+min_x],[(loopVar1*scaling_x)+min_y],[1]])
            old_point=inv_H*new_point
            old_point=old_point*(1/old_point[2][0])
            old_point=array(old_point)
            new_point=array(new_point)

        #When indices are positive, copy from original image.
            if ((old_point[0][0]>0)and(old_point[1][0]>0)):
                try:
                    output_img[loopVar1][loopVar2]=image[old_point[1][0]][old_point[0][0]]
                #When indices exceed the available image size,keep the ...
                    black pixel as it is in the output image.
                except IndexError:
                    output_img[loopVar1][loopVar2]=output_img[loopVar1][loopVar2]
            #When indices are negative, keep the black pixel as it is ...
                in the output image.
            else:
                output_img[loopVar1][loopVar2]=output_img[loopVar1][loopVar2]
        print loopVar1
    return output_img
#
# This function is the cost function used by Lev-Mar optimization. It ...
    takes in a parameter vector and returs the current cost vector
def CostFunction(p):
    P2=matrix(array(p).reshape((3,4)))      # Find P2 from parameter p
    est_x=[]
    for loopVar1 in range(len(op)):         # For all correspondences, ...
        do triangulation
        A=matrix(zeros((4,4)))          # Find Matrix A
        A[0,:]=(op[loopVar1, 0]*P1[2,:])-(P1[0,:])
        A[1,:]=(op[loopVar1, 1]*P1[2,:])-(P1[1,:])
        A[2,:]=(op[loopVar1, 2]*P2[2,:])-(P2[0,:])
```

```
        A[3,:]=(op[loopVar1, 3]*P2[2,:])-(P2[1,:])

        world_X=transpose(matrix((linalg.svd(transpose(matrix(A))*matrix(A))[2][3]).tolist()[0
            # Find world point

        proj_x=P1*world_X               # Project the world point back ...
            to Image 1 and Image 2
        proj_x=proj_x/proj_x[2,0]
        proj_x_bar=P2*world_X
        proj_x_bar=proj_x_bar/proj_x_bar[2,0]

        est_x.append(proj_x[0,0])           # Take the projected ...
            points as estimated
        est_x.append(proj_x[1,0])
        est_x.append(proj_x_bar[0,0])
        est_x.append(proj_x_bar[1,0])

    cost=subtract(X,est_x)                  # Find the cost by ...
        subtrating estimates from known image points
    return cost                     # Return the cost vector

#
# This function normalizes the selected points so that the ...
    rectification works for all scales
def normalize(correspondences):
    mean_x_1 = 0.0
    mean_y_1 = 0.0
    mean_x_2 = 0.0
    mean_y_2 = 0.0

    for loopVar1 in range(NUM_OF_POINTS):           # For all ...
        correspondences, find means
        mean_x_1+=correspondences[loopVar1, 0]
        mean_y_1+=correspondences[loopVar1, 1]
        mean_x_2+=correspondences[loopVar1, 2]
        mean_y_2+=correspondences[loopVar1, 3]

    mean_x_1/=float(NUM_OF_POINTS)
    mean_y_1/=float(NUM_OF_POINTS)
    mean_x_2/=float(NUM_OF_POINTS)
    mean_y_2/=float(NUM_OF_POINTS)

    variance_1 = 0.0
    variance_2 = 0.0
    for loopVar1 in range(NUM_OF_POINTS):           # For all ...
        correspondences, find variances
        variance_1+=sqrt((correspondences[loopVar1, 0] - mean_x_1)**2 ...
            + (correspondences[loopVar1, 1] - mean_y_1)**2)
        variance_2+=sqrt((correspondences[loopVar1, 2] - mean_x_2)**2 ...
            + (correspondences[loopVar1, 3] - mean_y_2)**2)
    variance_1/=float(NUM_OF_POINTS)
    variance_2/=float(NUM_OF_POINTS)

    scale_1 = sqrt(2)/variance_1            # Find Scales
```

```
    scale_2 = sqrt(2)/variance_2

    translate_x_1 = -scale_1*mean_x_1       # Find translation factors
    translate_y_1 = -scale_1*mean_y_1

    translate_x_2 = -scale_2*mean_x_2
    translate_y_2 = -scale_2*mean_y_2

    T1 = matrix(zeros((3,3)))           # Initialize T1 and T2
    T2 = matrix(zeros((3,3)))

    T1[0, 0]= scale_1               # Calculate T1
    T1[0, 2]= translate_x_1
    T1[1, 2]= translate_y_1
    T1[1, 1]= scale_1
    T1[2, 2]= 1

    T2[0, 0]= scale_2               # Calculate T2
    T2[0, 2]= translate_x_2
    T2[1, 2]= translate_y_2
    T2[1, 1]= scale_2
    T2[2, 2]= 1

    return T1, T2                   # Return the T1 and T2 matrices

#-----------------------------------------------------------------------------------------


# Main Code starts
op=matrix(readmatches(filename+'manual_correspondences.txt'))       # ...
    Read the manual correspondences from the file
NUM_OF_POINTS=op.shape[0]                              # Find the number of ...
    points
T1, T2=normalize(op)                                   # Find normalization ...
    matrices T1 and T2

#Calculate the F Matrix from AF=0
A=[] # A Matrix

#This loop fills in Matrix A
for loopVar1 in range(0,NUM_OF_POINTS):
    A.append([(op[loopVar1,2]*op[loopVar1,0]), ...
        (op[loopVar1,2]*op[loopVar1,1]), op[loopVar1,2], ...
        (op[loopVar1,3]*op[loopVar1,0]), ...
        (op[loopVar1,3]*op[loopVar1,1]) ,op[loopVar1,3] , ...
        op[loopVar1,0] , op[loopVar1,1], 1])

#Find out least squares solution and fill in F matrix
U,D,V=linalg.svd(A)
f=asarray(V[8])
F=matrix([[f[0],f[1],f[2]],[f[3],f[4],f[5]],[f[6],f[7],f[8]]])      # ...
    Initial F Matrix

U,D,V=linalg.svd(F)                               # Impose Rank 2 restriction
```

```
D_low_rank=zeros((3,3))
D_low_rank[0][0]=D[0]
D_low_rank[1][1]=D[1]
D_low_rank[2][2]=0
F=matrix(U)*matrix(D_low_rank)*matrix(V)
#F=F/float(F[2,2])
F=transpose(T2)*F*T1                          # Apply normalization
print 'F=', F

U,D,V=linalg.svd(F)                         # Get Epipoles from SVD
e_bar=matrix(U)[:,2]
e_bar=e_bar/e_bar[2,0]
e=transpose(matrix(V)[2,:])
e=e/e[2,0]
print 'e1=', e
print 'e2=', e_bar

P=c_[matrix(identity(3)), zeros((3,1))]                 # Find P1 and P2
s=matrix([[0, -e_bar[2,0], e_bar[1,0]], [e_bar[2,0], 0, -e_bar[0,0]], ...
    [-e_bar[1,0], e_bar[0,0], 0]])
P_bar=c_[s*F, e_bar]
print 'P1=', P
print 'P2=', P_bar
P1=P

#-------------------------------------------------------------------------------
#Optimization Code starts

X=[x for sublist in asarray(op) for x in sublist]           # Get all ...
    correspondence points as 1D list
p=hstack(asarray(P_bar)).tolist()                       # Convert P2 ...
    matrix to parameter vector
cost=CostFunction(p)                          # Check initial cost
print sum(square(cost))
#print type(cost), len(cost)
#print cost
optimal_p=leastsq(CostFunction, p)[0]                   # Run Lev-Mar ...
    optimization
#print optimal_p
P_bar=matrix(array(optimal_p).reshape((3,4)))               # Get ...
    Optimal P2 matrix
P_bar=P_bar/P_bar[2, 3]
cost=CostFunction(optimal_p)                        # Find optimal cost
print sum(square(cost))

print P_bar
e_bar=P_bar[:,3]                            # Get optimal epipole 2
print e_bar
H1=asarray(cv.CreateMat(3, 3, cv.CV_64FC1))             # Initialize ...
    H1 and H2 matrices
H2=asarray(cv.CreateMat(3, 3, cv.CV_64FC1))
Rectify_Images(op, F, P, P_bar, e, e_bar, H1, H2)           # Rectify ...
    the images by finding H1 and H2

```

```
save_matrix(filename+'F.txt', F)                    # Save all ... important matrices to file
save_matrix(filename+'P1.txt', P)
save_matrix(filename+'P2.txt', P_bar)
save_matrix(filename+'H1.txt', H1)
save_matrix(filename+'H2.txt', H2)
```

4.2 ProjectiveReconstruction.py

```
# Importing libraries
import cv2
import cv
from math import *
from numpy import *
from sympy import Symbol,cos,sin

from operator import *
from numpy.linalg import *
import time
import ctypes
import matplotlib.pyplot as plt
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
# Prints the numbers in float instead of scientific format
set_printoptions(suppress=True)

filename='Dataset 5/'  # Dataset Used
#------------------------------------------------------------
#Parameters used in the method
SEARCH_WINDOW=3     # Searching window of rows +/- 3 rows

# Canny parameters
canny_lowThreshold=70
canny_threshold_ratio=2
canny_kernel_size=3

# NCC parameters
w=5         # NCC neighbor window size
center_of_op=(w/2)
ncc_th=0.9
#ncc_r_th=0.8

#------------------------------------------------------------
# This function takes in two images and the matched set of points. ... Draws lines between matched points on a concatenated image
def draw_lines(img1, img2, matches):
    offset=img1.shape[1]
    #print offset

    final_image=array(concatenate((img1, img2), axis=1))        # ... Concatenate the two images
```

```

    for loopVar1 in range(0, len(matches)):     # For each pair of ...
        point in the matched set
        #print matches[loopVar1]
        cv2.line(final_image, (int(matches[loopVar1][0][0]), ...
            int(matches[loopVar1][0][1])), ...
            (int(matches[loopVar1][1][0])+offset, ...
            int(matches[loopVar1][1][1])), (0,0,255), 1)      # Draw a ...
            line
        cv2.circle(final_image, (int(matches[loopVar1][0][0]), ...
            int(matches[loopVar1][0][1])), 2, (255,0,0), -1) # Draw a ...
            point
        cv2.circle(final_image, (int(matches[loopVar1][1][0])+offset, ...
            int(matches[loopVar1][1][1])), 2, (255,0,0), -1) # Draw a ...
            point
    cv2.imwrite(filename+"matched_points.jpg", final_image)     # Save ...
        the resultant image
    return final_image

#---------------------------------------------------------------------------
#This function saves the correspondences to a text file to be read by ...
    3D plotting routine
def save_matches(filename, matches):
    fo = open(filename, 'w', 0)
    for loopVar1 in range(0, len(matches)):
        fo.write(str(matches[loopVar1][0][0]))
        fo.write('\t')
        fo.write(str(matches[loopVar1][0][1]))
        fo.write('\t')
        fo.write(str(matches[loopVar1][1][0]))
        fo.write('\t')
        fo.write(str(matches[loopVar1][1][1]))
        if loopVar1 !=len(matches)-1:
            fo.write('\n')
    fo.close()

#---------------------------------------------------------------------------#
#This function reads the matrices or points saved in a text file
def readmatches(filename):
    f = open(filename).read()
    rows = []
    for line in f.split('\n'):
        rows.append(line.split('\t'))

    for loopVar1 in range(0, len(rows)):
        for loopVar2 in range(0, len(rows[loopVar1])):
            rows[loopVar1][loopVar2]=float(rows[loopVar1][loopVar2])
    return rows

#---------------------------------------------------------------------------#
# This function takes in a binary edge image and extracts the interest ...
    points just by checking if the pixel value is 1 or not
def get_points(image, x_range, y_range):
    points=[]
```

```
    for loopVar1 in range(y_range[0], y_range[1]+1):    # For all ...
        pixels within specified window
        for loopVar2 in range(x_range[0], x_range[1]+1):
            if image[loopVar1, loopVar2]==1:        # Check if the ...
                pixel value is 1
                points.append([loopVar2, loopVar1]) # If yes, save the ...
                    point
    return points

#-----------------------------------------------------------------------------------------------
# This funtion takes in an image, a point and window size to find the ...
    f and m values in the neighbor -> used by NCC
def find_fm_neighbor(point, image, w):
    f=[]                    # Initialize f and m matrices
    m = zeros((w,w),float)
    center_of_op=(w/2)
    image_height=image.height        # Get size of image
    image_width=image.width
    for loopVar3 in range(-center_of_op, center_of_op+1): # Find f and ...
        m by looping through the window
        temp1=[]
        for loopVar4 in range(-center_of_op, center_of_op+1):
            if ((point[0]+loopVar3)>=0 and ...
                (point[0]+loopVar3)<image_width) and ...
                ((point[1]+loopVar4)>=0 and ...
                (point[1]+loopVar4)<image_height):
                temp1.append(image[(point[1]+loopVar4), ...
                    (point[0]+loopVar3)])
        if len(temp1)==w:
            f.append(temp1)
    m.fill(mean(f))             # Fill in mean matrix m
    fm=subtract(f,m)            # Subtract mean matrix from f matric
    return fm               # Return the (f-m) window

#-----------------------------------------------------------------------------------------------
# This function takes in two points and finds the ncc between those two
def find_ncc(point1, point2):
    fm1=find_fm_neighbor(point1, input_img_1, w)    # Finds (f-m) ...
        window for point 1
    fm2=find_fm_neighbor(point2, input_img_2, w)    # Finds (f-m) ...
        window for point 2
    sum_of_squares_1=sum(square(fm1))       # Find sum of squares
    sum_of_squares_2=sum(square(fm2))
    prod=multiply(fm1,fm2)              # Find product of (f-m) windows
    sum_of_prod=sum(prod)               # Find sum of product
    ncc=sum_of_prod/sqrt(sum_of_squares_1*sum_of_squares_2) # ...
        Calculate NCC
    return ncc                  # Return the NCC score

#-----------------------------------------------------------------------------------------------
# This function takes in two sets of points of two images and ncc ...
    parameters. It finds the matched points using NCC metric.
def find_correspondences(points1, points2, ncc_th, r):
    matches=[]
```

```
    for loopVar1 in range(len(points1)):        # For each point in ...
        image 1
        max_ncc=0
        flag=0
        for loopVar2 in range(len(points2)):    # For each point in ...
            image 2
            if ...
                abs(points2[loopVar2][1]-points1[loopVar1][1])<SEARCH_WINDOW: ...
                # If the point 2 is within search window
                ncc=find_ncc(points1[loopVar1], points2[loopVar2])  # ...
                    Find NCC
                if ncc > ncc_th:                # If NCC exceeds threshold
                    if ncc > max_ncc:           # Check if its greater ...
                        than current maximum score
                        max_ncc=ncc         # If yes, update the ...
                            maximum and update the best point
                        flag=1
                        current_match=points2[loopVar2]

        if flag==1:                         # If there is a match found
            matches.append([points1[loopVar1],current_match])   # Save ...
                the matched pair
        print loopVar1
    return matches                      # Return the set of matched points

#-----------------------------------------------------------------------------
# This function implements triangulation. It takes in set of image ...
    points and project matrices. Finds the correspondings world points
def ProjectToWorld(op, P1, P2):
    world_points=[]
    for loopVar1 in range(len(op)):         # For all image ...
        correspondences
        A=matrix(zeros((4,4)))              # Find A matrix
        A[0,:]=(op[loopVar1, 0]*P1[2,:])-(P1[0,:])
        A[1,:]=(op[loopVar1, 1]*P1[2,:])-(P1[1,:])
        A[2,:]=(op[loopVar1, 2]*P2[2,:])-(P2[0,:])
        A[3,:]=(op[loopVar1, 3]*P2[2,:])-(P2[1,:])

        world_X=transpose(matrix((linalg.svd(transpose(matrix(A))*matrix(A)))[2][3]).tolist()[0
            # Find world point
        world_X=world_X/world_X[3,0]
        world_points.append([world_X[0, 0], world_X[1, 0], world_X[2, ...
            0]])      # Save the world points
    return world_points                                 # Return them

#-----------------------------------------------------------------------------
# This function takes in world points and draws 3D plot using plotting ...
    tools
def draw3D(world_points):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')          # Create a ...
        figure and add 3D sub plot
    x = []
    y = []
```

```
    z = []
    count = 0
    for loopVar1 in range(len(world_points)):          # For all ...
        world points
        if abs(world_points[loopVar1][2]) < 20:        # get rid of ...
            outliers!
            x.append(world_points[loopVar1][0])      # Extract x, y and ...
                z coordinates
            y.append(world_points[loopVar1][1])
            z.append(world_points[loopVar1][2])
            count+=1
    ax.scatter(x, y, z, zdir='z', s=1)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()

#----------------------------------------------------------------------------
# Main Code starts
# Load both the images in color as well as gray scale
                                        # Load image 1
orig_img_1 = cv2.imread(filename+'Pic_1_corrected.jpg',1)
input_img_1 = cv.LoadImage(filename+'Pic_1_corrected.jpg',0)

                                        # Load image 2
orig_img_2 = cv2.imread(filename+'Pic_2_corrected.jpg',1)
input_img_2 = cv.LoadImage(filename+'Pic_2_corrected.jpg',0)

                                        # Apply Canny Edge detector ...
                                            for Image 1
detected_edges=cv.CreateImage((orig_img_1.shape[1], ...
    orig_img_1.shape[0]), cv.IPL_DEPTH_8U, 1)
cv.Canny(input_img_1, detected_edges, canny_lowThreshold, ...
    canny_lowThreshold*canny_threshold_ratio, canny_kernel_size ) # ...
    Apply Canny detector
cv.SaveImage(filename+'Pic_1_edge.jpg', detected_edges)
edge_image_1 = cv2.imread(filename+'Pic_1_edge.jpg',0)
                                        # Apply Canny Edge detector ...
                                            for Image 2
detected_edges=cv.CreateImage((orig_img_2.shape[1], ...
    orig_img_2.shape[0]), cv.IPL_DEPTH_8U, 1)
cv.Canny(input_img_2, detected_edges, canny_lowThreshold, ...
    canny_lowThreshold*canny_threshold_ratio, canny_kernel_size ) # ...
    Apply Canny detector
cv.SaveImage(filename+'Pic_2_edge.jpg', detected_edges)
edge_image_2 = cv2.imread(filename+'Pic_2_edge.jpg',0)

points1=get_points(edge_image_1, (170, 600), (5, 430))              # ...
    Extract interest points in Image 1
points2=get_points(edge_image_2, (115, 550), (5, 430))              # ...
    Extract interest points in Image 2

print len(points1), len(points2), points1[len(points1)-1]
```

```
matches=find_correspondences(points1, points2, ncc_th, ncc_r_th)    ...
        # Find matches using NCC method
print len(matches)
draw_lines(orig_img_1, orig_img_2, matches)                 # Draw ...
    lines between matched points in 2D images
save_matches(filename+'matches.txt', matches)                   # Save ...
    the matches

correspondences=matrix(readmatches(filename+'matches.txt'))         # ...
    Read the matches
P1=matrix(readmatches(filename+'P1.txt'))                   # Read P1 ...
    and P2 matrices
P2=matrix(readmatches(filename+'P2.txt'))
print len(correspondences), P1, P2
world_points=ProjectToWorld(correspondences, P1, P2)                # ...
    Project points to world coordinates
print len(world_points)
draw3D(world_points)                                # Draw 3D plots of ...
    the world coordinates

'''                                     # SURF Code (TRIED - NOT USED ...
    FINALLY)
#--------------------------------------------------------------------------------------------------
# Use OpenCV's built in SURF function to get the corner points and ...
    their descriptors
(points1, desc1)=cv.ExtractSURF(input_img_1, None, ...
    cv.CreateMemStorage(), (0, surf_th, 3, 1))
(points2, desc2)=cv.ExtractSURF(input_img_2, None, ...
    cv.CreateMemStorage(), (0, surf_th, 3, 1))


for loopVar1 in range(len(points1)):
    points1[loopVar1]=points1[loopVar1][0]
for loopVar1 in range(len(points2)):
    points2[loopVar1]=points2[loopVar1][0]

points1=sorted(points1, key=itemgetter(1))
points2=sorted(points2, key=itemgetter(1))
print points2
exit()
matches=[]
find_matches(points1, points2, desc1, desc2, matches, score_th, ...
    ratio_th) # Use the descriptors to find the matches
draw_lines(orig_img_1, orig_img_2, matches) # Draw the lines between ...
    matches points
print len(matches)              # print the number of matched points
print len(points1), len(points2)'''
```