Homework 10

Fangda Li
li1208@purdue.edu
ECE 661 - Computer Vision

November 29, 2016

1 Methodology

This section describes steps needed to reconstruct an object in 3D from two stereo images using epipolar geometry.

1. To begin, take two images of an object from two different angles. Then, manually pick 8 corresponding pairs of points on each image.
2. Using the 8 point pairs, find the initial estimation of the fundamental matrix $\mathbf{F}$.
   (a) The coordinates $\vec{x_i}$ and $\vec{x_i}'$ of projection of a world point onto the two stereo image planes have to satisfy the following constraint:
   $$\vec{x_i}'^T \mathbf{F} \vec{x_i} = 0. \tag{1}$$
   (b) Subsequently, a linear homogeneous system can be obtained by stacking up the following equation,
   $$x_i' x_i f_{11} + x_i' y_i f_{12} + x_i' f_{13} + y_i' x_i f_{21} + y_i' y_i f_{22} + y_i' f_{23} + x_i f_{31} + y_i f_{32} + f_{33} = 0, \tag{2}$$
   where $f_{k,l}$ is the element on the kth row and lth column of $\mathbf{F}$.
   (c) Solving the homogeneous system with linear least squares method shall give the initial estimation of the unconditioned $\mathbf{F}$. In order to condition $\mathbf{F}$ to be rank 2, use SVD to decompose $\mathbf{F}$ and re-set its last eigenvalue to 0.
3. Find the initial estimation of epipoles $\vec{e}$ and $\vec{e}'$ of the two image planes.
   (a) The two epipoles $\vec{e}$ and $\vec{e}'$ are the right and left null vector of $\mathbf{F}$, respectively.
4. Obtain the canonical camera projection matrices $\mathbf{P}$ and $\mathbf{P}'$ for the two stereo image planes using the following equation:
   $$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{3}$$

$$P' = \left[[\vec{e}']_X \mathbf{F} \mid \vec{e}' \right], \tag{4}$$

where $[\vec{e}']_X$ is the cross-product equivalent matrix of $\vec{e}'$.

5. Refine the fundamental matrix using the non-linear optimization algorithm, Levenberg-Marquardt, with the 8 manual correspondences.

   (a) The geometric error used in LM is given by

   $$d^2_{geom} = \sum_i (||\vec{x}_i - \hat{\vec{x}}_i||^2 + ||\vec{x}_i{}' - \hat{\vec{x}}_i'||^2), \tag{5}$$

   where $\vec{x}_i$ and $\vec{x}_i{}'$ are the re-projected coordinates of the world point $\vec{X}_i$, which is obtained by the triangulation process described in the following steps.

   (b) Given two corresponding points $\vec{x}_i$ and $\vec{x}_i{}'$ on the stereo image planes, its back-projected physical coordinate $\vec{X}_i$ is obtained by solving the following homogeneous system with linear least squares method,

   $$\begin{bmatrix} x_i \vec{P_3}^T - \vec{P_1}^T \\ y_i \vec{P_3}^T - \vec{P_2}^T \\ x_i' \vec{P_3}'^T - \vec{P_1}'^T \\ y_i' \vec{P_3}'^T - \vec{P_2}'^T \end{bmatrix} \vec{X}_i = 0. \tag{6}$$

   , where $\vec{P_k}$ is the kth row of $\mathbf{P}$ and $\vec{P_k}'$ is the kth row of $\mathbf{P}'$.

6. Rectify the two images by bringing the two epipoles to infinity along x-axis.

   (a) The homography matrix $\mathbf{H}'$ for rectifying the right image is given by

   $$\mathbf{H}' = \mathbf{T}_2\mathbf{G}\mathbf{R}\mathbf{T}_1, \tag{7}$$

   where $\mathbf{T}_1$ translates the image center to origin, $\mathbf{R}$ rotates the epipole onto x-axis, $\mathbf{G}$ takes the epipole to infinity and $\mathbf{T}_2$ translates the image back to its original image center.

   (b) After $\mathbf{H}'$ is obtained, the homography matrix that rectifies the left image can be computed by the following procedure:

      i. Let $\mathbf{M} = \mathbf{P}'\mathbf{P}^\dagger$.

      ii. Let $\mathbf{H}_0 = \mathbf{H}'\mathbf{M}$ and $\mathbf{H}_A = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

      iii. Let $\hat{\vec{x}}_i = \mathbf{H}_0\vec{x}_i$ and $\hat{\vec{x}}_i' = \mathbf{H}'\vec{x}_i{}'$.

      iv. a, b and c are obtained by

   $$\arg\min_{a,b,c} \sum_i (a\hat{x}_i + b\hat{y}_i + c - \hat{x}_i{}')^2 \tag{8}$$

v. Finally, $\mathbf{H} = \mathbf{H}_A\mathbf{H}_0$.

7. Obtain the features on the two rectified images and establish correspondences between the features with constraints.

   (a) The difference in row coordinate of the two matching points has to be smaller than a threshold.

   (b) The Euclidean distance between the two descriptors also has to be smaller than a certain threshold.

8. Further refine $\mathbf{F}$ using Levenberg-Marquardt with the automatic correspondences found in the previous step. The refined $\mathbf{F}$ shall be the final fundamental matrix in this experiment. Subsequently, the canonical camera projection matrices obtained using the final fundamental matrix are the final projection matrices in this experiment.

9. Triangulate all the automatic correspondences to form the point cloud in world 3D of the object.

2 Results

Figure 1: Input images

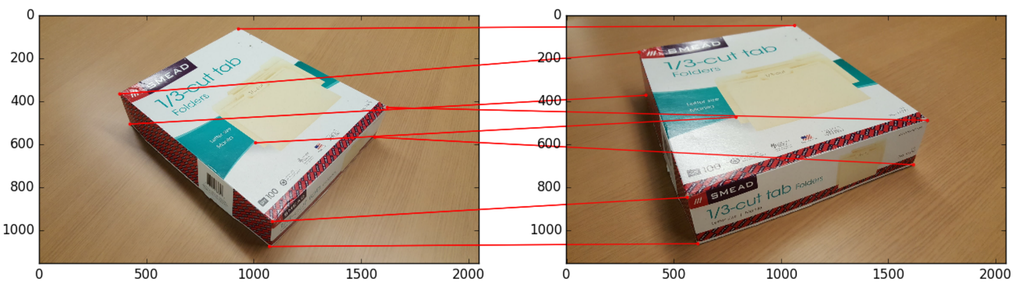


Figure 2: Manual correspondences

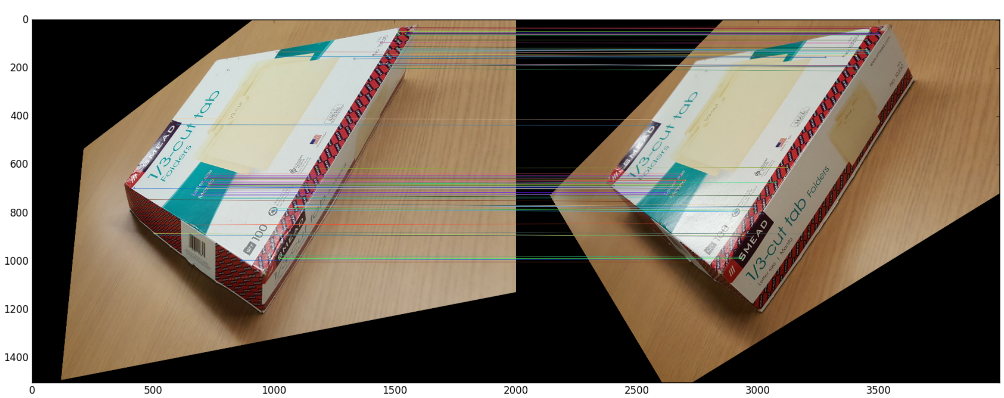


Figure 3: Automatic correspondences generated using SIFT on rectified images. Note that all the line connections are horizontal because of the rectification process. Moreover, without using Levenberg-Marquardt optimization on the initial estimation of **F**, my rectified images could not show up properly.

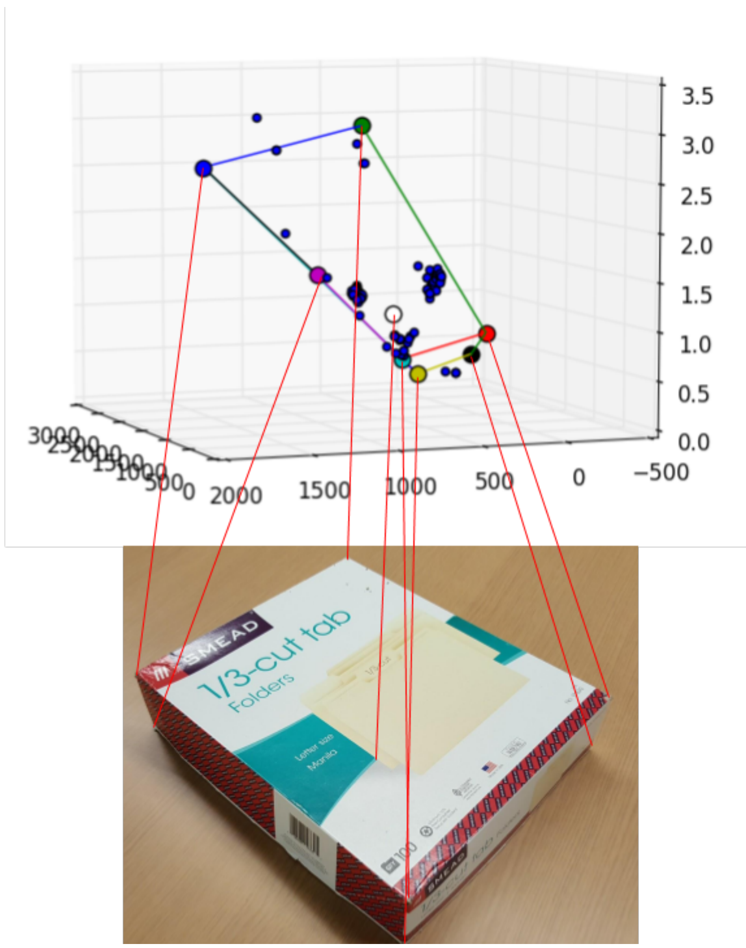


Figure 4: Correspondences between the reconstructed 3D points and the input image. The projective distortion can be clearly observed.

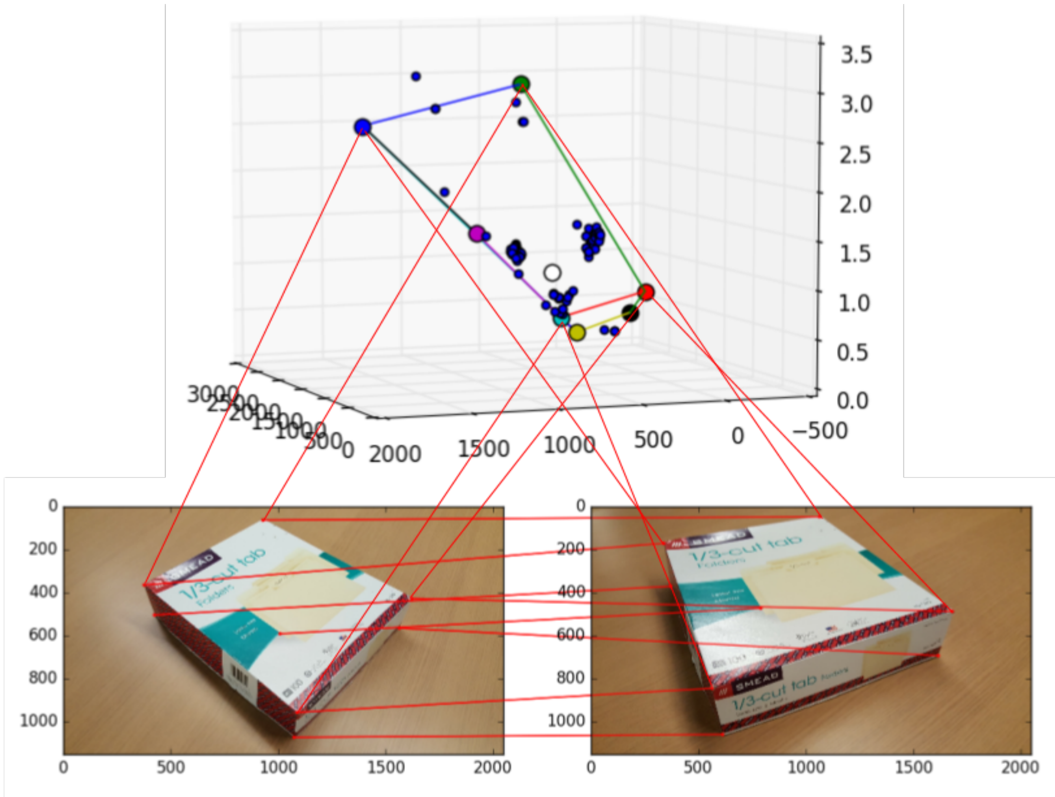
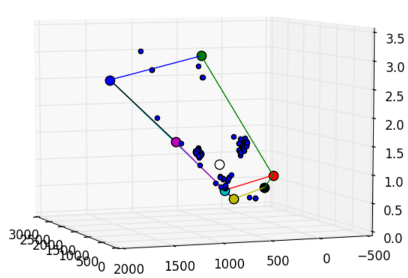
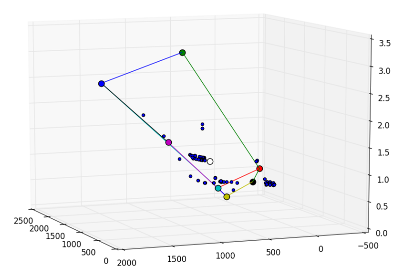


Figure 5: Correspondences between the reconstructed 3D points and the two input images.



(a) With Levenberg-Marquardt



(b) Without Levenberg-Marquardt

Figure 6: Comparison on the reconstructed 3D points between using and not using Levenberg-Marquardt on the SIFT generated correspondences. Note that without nonlinear optimization, the points tend to fall out of the wireframe of the input object.

3 Source Code

3.1 rectification.py

```
#!/usr/bin/python
from pylab import *
import cv2
from scipy.optimize import leastsq

def apply_transformation_on_points(points, H):
	'''
		Apply the given transformation matrix on all points from input.
		@coords: list of input points, each is represented by (row,col)
		@return: list of points after transformation, each is represented by (
	'''
	l = []
	for point in points:
		p = array([point[0], point[1], 1.])
		p = dot(H, p)
		p = p / p[-1]
		l.append( (p[0], p[1]) )
	return l

def solve_homo_system_with_llsm(pts1, pts2):
	'''
		Obtain the the solution to a homogeneous linear least square problem.
		@pts1,pts2: list of matching points
		@return: ndarray of H
	'''
	n = len(pts1)
	A = zeros((n,9))
	for i in range(n):
		x,y,w = pts1[i][0], pts1[i][1], 1.
		xp,yp,wp = pts2[i][0], pts2[i][1], 1.
		A[i] = [xp*x, xp*y, xp, yp*x, yp*y, yp, x, y, 1]
	# Solution is the right eigenvector corresponding to the smallest eigenvalue
	U, s, Vt = svd( dot(A.T, A) )
	v = Vt[-1,:]
	return v

def condition_fundamental_matrix(F):
	'''
		Condition the fundamental matrix using SVD such that it's of rank 2.
	'''
	U, s, Vt = svd(F)
```

```
        s[-1] = 0.
        D = diag(s)
        F = dot( U, dot( D, Vt ) )
        F = F / F[-1,-1]
        return F

def get_normalization_homography_matrix():
        '''
                Find the homography matrix that transforms the given coordinates to ze
        '''
        pass

def get_right_null_space(A, eps=1e-5):
        '''
                Return the null space vector(s) of a matrix.
        '''
        U, s, Vt = svd(A)
        null_space = compress(s <= eps, Vt, axis=0)
        return null_space.T

def get_cross_product_equiv_matrix(w):
        '''
                Get the skew-symmetric cross-product equivalent matrix of a vector
        '''
        x,y,z = w[0],w[1],w[2]
        return array([[0., -z, y],
                                [z, 0., -x],
                                [-y, x, 0.]])

def triangulate_point(P, Pp, pt1, pt2):
        '''
                Given two corresponding points on the two image planes, return its phy
        '''
        A = zeros((4,4))
        A[0,:] = pt1[0] * P[2,:] - P[0,:]
        A[1,:] = pt1[1] * P[2,:] - P[1,:]
        A[2,:] = pt2[0] * Pp[2,:] - Pp[0,:]
        A[3,:] = pt2[1] * Pp[2,:] - Pp[1,:]
        # Solution is the right eigenvector corresponding to the smallest eigenvalue
        U, s, Vt = svd( dot(A.T, A) )
        v = Vt[-1,:]
        return v / v[-1]

def triangulate_points(P, Pp, pts1, pts2):
        '''
```

```
            Convenience function.
        '''
        pts = []
        for pt1,pt2 in zip(pts1,pts2):
                pt = triangulate_point(P, Pp, pt1, pt2)
                pts.append(pt)
        return pts

def get_fundamental_matrix_from_projection(P, Pp):
        '''
                Extract F from the secondary canonical camera projection matrix.
        '''
        ep = Pp[:,3]
        s = get_cross_product_equiv_matrix(ep)
        F = dot( s, dot ( Pp, dot( P.T, inv( dot(P, P.T) ) ) ) )
        return F / F[-1,-1]

def nonlinear_optimization(pts1, pts2, P, Pp):
        '''
                Optimize the secondary camera matrix in canonical configuration.
        '''
        nPts = len(pts1)
        array_meas = hstack((array(pts1).T, array(pts2).T))
        array_reprj = zeros(array_meas.shape)
        p_guess = Pp.flatten()
        def error_function(p):
                '''
                        Geometric distance as cost function for LevMar.
                '''
                Pp = p.reshape(3,4)
                array_reprj.fill(0.)
                for i in range(nPts):
                        pt1, pt2 = pts1[i], pts2[i]
                        pt_world = triangulate_point(P, Pp, pt1, pt2)
                        pt1_reprj = dot(P, pt_world)
                        pt1_reprj = pt1_reprj / pt1_reprj[-1]
                        pt2_reprj = dot(Pp, pt_world)
                        pt2_reprj = pt2_reprj / pt2_reprj[-1]
                        array_reprj[:,i] = pt1_reprj[:2]
                        array_reprj[:,i+nPts] = pt2_reprj[:2]
                error = array_meas - array_reprj
                return error.flatten()
        print "Optimizing..."
        p_refined, _ = leastsq(error_function, p_guess)
        Pp_refined = p_refined.reshape(3,4)
```

```python
        Pp_refined = Pp_refined / Pp_refined[-1,-1]
        P_refined = P
        return P_refined, Pp_refined

def get_epipoles(F):
        '''
                Given fundamental matrix, return the left and right epipole.
        '''
        e = get_right_null_space(F)
        assert e.shape[1] == 1, "More than one left epipoles have been found."
        ep = get_right_null_space(F.T)
        assert ep.shape[1] == 1, "More than one right epipoles have been found."
        return e/e[-1], ep/ep[-1]

def get_canonical_projection_matrices(F, ep):
        '''
                Given fundamental matrix and epipole of the right image plane, find bo
        '''
        P = hstack(( eye(3), zeros((3,1)) ))
        s = get_cross_product_equiv_matrix(ep)
        Pp = hstack(( dot(s, F), ep ))
        return P, Pp

def get_fundamental_matrix(pts1, pts2):
        '''
                Given point correspondences, make an initial estimate of fundamental m
        '''
        get_normalization_homography_matrix()
        f = solve_homo_system_with_llsm(pts1, pts2)
        F = f.reshape(3,3)
        F = condition_fundamental_matrix(F)
        return F

def get_rectification_homographies(image1, image2, pts1, pts2, e, ep, P, Pp):
        '''
                Find the homography matrices that align the corresponding epipolar lin
        '''
        # Start with the second image first
        h2, w2 = image2.shape[0], image2.shape[1]
        # Translational matrix that shifts the image to be origin-centered
        T1 = array([[1., 0., -w2/2.],
                                [0., 1., -h2/2.],
                                [0., 0., 1.]])
        # Rotational matrix that rotates the epipole onto x-axis
        theta = arctan( (ep[1] - h2/2.) / (ep[0] - w2/2.) )
```

```
        # Since we want to rotate to positive x-axis
        theta = -theta[0]
        R = array([[cos(theta), -sin(theta), 0.],
                                [sin(theta), cos(theta), 0.],
                                [0., 0., 1.]])
        # Homography that takes epipole to infinity
        f = norm(array([ep[1] - h2/2., ep[0] - w2/2.]))
        G = array([[1., 0., 0.],
                                [0., 1., 0.],
                                [-1./f, 0., 1.]])
        # Translate back to original center
        T2 = array([[1., 0., w2/2.],
                        [0., 1., h2/2.],
                        [0., 0., 1.]])
        # The final homography for the second image
        Hp = dot( T2, dot( G, dot(R, T1) ) )
        ####
        # Now the first image
        M = dot( Pp, dot( P.T, inv( dot(P, P.T) ) ) )
        H0 = dot( Hp, M )
        pts1h = apply_transformation_on_points(pts1, H0)
        pts2h = apply_transformation_on_points(pts2, Hp)
        # Construct inhomogeneous system
        n = len(pts1)
        A = zeros((n,3))
        b = zeros((n,1))
        for i in range(n):
                xh,yh = pts1h[i][0], pts1h[i][1]
                xph = pts2h[i][0]
                A[i] = [xh, yh, 1.]
                b[i] = xph
        # h is pseudo-inverse multiplied by b
        h = dot( dot( inv( dot(A.T, A) ), A.T ), b )
        h = h.flatten()
        # Obtain the homography for the first image
        HA = array([[h[0], h[1], h[2]],
                        [0., 1., 0.],
                        [0., 0., 1.]])
        H = dot(HA, H0)
        return H, Hp
```

3.2 features.py

```
import cv2
```

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

def get_sift_matchings(kp1, des1, kp2, des2, rowDiff=10, maxDist=300):
        '''
                Get matchings from SIFT features with constrains and return list of ma
        '''
        bf = cv2.BFMatcher()
        matches = bf.match(des1, des2)
        pts1, pts2 = [], []
        good = []
        for m in matches:
                pt1 = kp1[m.queryIdx].pt
                pt2 = kp2[m.trainIdx].pt
                # Reinforce constraint from rectification
                # Matches have to reside in approximately the same row
                if abs(pt1[1] - pt2[1]) > rowDiff or m.distance > maxDist: continue
                pts1.append(pt1)
                pts2.append(pt2)
                good.append([m])
        return pts1, pts2, good
```

3.3 main.py

```
#!/usr/bin/python
from pylab import *
import cv2
from rectification import *
from features import *
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.patches import Circle, ConnectionPatch

def pick_points(image):
        '''
                Prompt user to click points on image.
        '''
        points = []
```

```
        fig = plt.figure()
        def onclick(event):
                if len(points) == 8:
                        return
                x = int(event.xdata)
                y = int(event.ydata)
                points.append((x,y))
                print "Mouse clicked at (x,y)...", (x,y), "; Total number of points clic
                print points
        imshow(image)
        cid = fig.canvas.mpl_connect('button_press_event', onclick)
        show()

def main():
        nfeatures = 1000 # Number of SIFT features
        w,h = 2000, 1500 # Size of the canvas
        image1 = imread('./images/3.jpg')
        image2 = imread('./images/4.jpg')
        pts1_manual = [(376, 365), (929, 63), (1623, 428), (1088, 958), (424, 505), (107
        pts2_manual = [(339, 171), (1064, 47), (1683, 488), (564, 846), (371, 371), (613
        figure()
        subplot(1,2,1)
        imshow(image1)
        subplot(1,2,2)
        imshow(image2)
        # Uncomment to pick points if necessary
        # pick_points(image1)
        # pick_points(image2)
        # Plot the manual correspondences
        figure()
        fig, axes = subplots(1,2)
        axes[0].set_aspect('equal')
        axes[0].imshow(image1)
        axes[1].set_aspect('equal')
        axes[1].imshow(image2)
        for i in range(8):
                # color = np.random.rand(3,1)
                color = (1., 0, 0)
                pt1 = pts1_manual[i]
                pt2 = pts2_manual[i]
                axes[0].add_patch( Circle(pt1, 5, fill=False, color=color, clip_on=False
                axes[1].add_patch( Circle(pt2, 5, fill=False, color=color, clip_on=False
                # Draw lines for matching pairs
                line1 = ConnectionPatch(xyA=pt1, xyB=pt2, coordsA='data', coordsB='data'
                line2 = ConnectionPatch(xyA=pt2, xyB=pt1, coordsA='data', coordsB='data'
```

```
        axes[0].add_patch(line1)
        axes[1].add_patch(line2)
F = get_fundamental_matrix(pts1_manual, pts2_manual)
print "F = ", F
e, ep = get_epipoles(F)
print "e = ", e
print "ep = ", ep
P, Pp = get_canonical_projection_matrices(F, ep)
print "P = ", P
print "Pp = ", Pp
print "======== First Nonlinear Optimization ========"
P_refined, Pp_refined = nonlinear_optimization(pts1_manual, pts2_manual, P, Pp)
print "P_refined = ", P_refined
print "Pp_refined = ", Pp_refined
F_refined = get_fundamental_matrix_from_projection(P_refined, Pp_refined)
print "F_refined = ", F_refined
e_refined, ep_refined = get_epipoles(F_refined)
print "e_refined = ", e_refined
print "ep_refined = ", ep_refined
print "======== Rectification ========"
H, Hp =        get_rectification_homographies(image1, image2, pts1_manual, pts2_
print "H = ", H
print "Hp = ", Hp
print "e = ", dot(H, e_refined)
print "ep = ", dot(Hp, ep_refined)
rectified1 = cv2.warpPerspective(image1, H, (w,h))
rectified2 = cv2.warpPerspective(image2, Hp, (w,h))
figure()
subplot(1,2,1)
imshow(rectified1)
subplot(1,2,2)
imshow(rectified2)
print "======== Feature Matching ========"
kp1, des1 = get_sift_kp_des(rectified1, nfeatures=nfeatures)
kp2, des2 = get_sift_kp_des(rectified2, nfeatures=nfeatures)
pts1_ft, pts2_ft, good = get_sift_matchings(kp1, des1, kp2, des2)
for i,pt in enumerate(pts1_ft):
        pts1_ft[i] = (pt[1],pt[0])
for i,pt in enumerate(pts2_ft):
        pts2_ft[i] = (pt[1],pt[0])
matchings = cv2.drawMatchesKnn(rectified1,kp1,rectified2,kp2,good,array([]),flag
figure()
imshow(matchings)
print "======== Final Nonlinear Optimization ========"
P_final, Pp_final = nonlinear_optimization(pts1_ft, pts2_ft, P_refined, Pp_refin
```

```
        print "P_final = ", P_final
        print "Pp_final = ", Pp_final
        F_final = get_fundamental_matrix_from_projection(P_final, Pp_final)
        print "F_final = ", F_final
        e_final, ep_final = get_epipoles(F_final)
        print "e_final = ", e_final
        print "ep_final = ", ep_final
        print "======== Final Triangulation ========"
        pts_world = triangulate_points(P_final, Pp_final, pts1_ft, pts2_ft)
        pts_world = array(pts_world)
        fig = figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(pts_world[:,1], pts_world[:,0], pts_world[:,2], c='b', depthshade=Fal
        # Also plot the original manual points
        pts1_manual_rect = apply_transformation_on_points(pts1_manual, H)
        pts2_manual_rect = apply_transformation_on_points(pts2_manual, Hp)
        pts_manual_world = triangulate_points(P_final, Pp_final, pts1_manual_rect, pts2_
        pts = pts_manual_world
        # 9 lines in total
        for s,e in [(0,1), (1,2), (2,3), (3,0), (4,5), (5,6), (0,4), (3,5), (2,6)]:
                ax.plot([pts[s][0], pts[e][0]], [pts[s][1], pts[e][1]], zs=[pts[s][2], p
        # pts_manual_world = array(pts_manual_world)
        for i,c in enumerate(['b', 'g', 'r', 'c', 'm', 'y', 'k', 'w']):
                pt = pts_manual_world[i]
                ax.scatter(pt[0], pt[1], pt[2], c=c, s=80, depthshade=False)
        show()

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