Assignment 2: Multi-View Geometry

Comment: most of the written problems are designed to help with the coding part (structure-from-motion). Thus, they should be solved first. Your coding part will require figuring out how to work with numpy arrays (e.g. slicing, broadcasting). If you are new to numpy, this could be tricky. I can recommend working on small test cases. For debugging, print arrays (letter and after slicing [ ]) to verify that the result is correct.

Problem 1

Prove that epipoles in two images obtained by the same camera during "translating" motion (no rotation) have exactly the same location in both images.

Details:

(a) represents the camera centre before translation and C_1' represents the camera centre after translation. Let C_1 be the world coordinate. Then the coordinate of C_1 in world coordinate system will be $C_1 = [0, 0, 0]^T$. Since there is only translation between the two camera centre. Then the coordinate of C_1 in world coordinate system is $C_1' = [t_x, t_y, t_z]^T$. T is called T in the transition vector.

Similarly we know that epipoles e_1 and e_2 can be computed by $e_1 = P_1 C_2$ and $e_2 = P_2 C_1$. Given the P_1, the projection matrix one camera centre 1, since there is no translation nor rotation for the camera centre, $P_1 = [I|0]$. Similarly the projection matrix P_2 one camera centre 2 is $P_2 = [I|T]$.

Then given these information, $e_1 = [I|0]C_1' = [I \times 0 + T] = [t_x, t_y, t_z]^T$ and $e_2 = [I|T]C_1 = [0 \times I + T] = [t_x, t_y, t_z]^T = T$. We can see that without translation, the epipoles in two images are at the same coordinate in the image plane.

Problem 2

Assuming a calibrated camera (that is, $K = I$) and its two views corresponding to projection matrices $P_1 = [I|0]$ and $P_2 = [R|T]$ w.r.t. some world coordinate system, show formulas for coordinates of the following 3D points (in the same world coordinate system):

Assuming T is column vector, and all coordinates are represented in row vector.

(a) optical center for the first view:

$C_1 = [0, 0, 0]$

(b) image center for the first view:

Given $K[I \mid 0]X = [0, 0, 0]^T$. So the result represents a line of z axis.

And since $K = I$, we know that the focal length is 1. Thus, the coordinate of $Q = [0, 0, 1]$

(c) optical center for the second view: $C_2 = -R[0, 0, 0]^T + T = [T]^T$

(d) image center for the second view: $Q_2 = -R[0, 0, 1]^T + [T]^T$

Problem 3

Using the same set up in problem 2, show formulas for normalized coordinates of the following image points:

(a) epipole in the first camera image: $e_1 = P_1C_2$.

(b) epipole in the second camera image: $e_2 = P_2C_1$.

Problem 4 (homogeneous and non-homogeneous line representations)

Lines in 2D images can be represented "homogeneously" as 3-vectors $l = [l_1, l_2, l_3]^T$ that give equation $l^T x = 0$ for homogeneous points $x = [x_1, x_2, x_3]^T = [\frac{x}{z}, \frac{y}{z}, 1]^T$ forming a line. Given l, what are the values of scalar parameters a, b in the line equation $ax + by = 0$ for the same 2D points based on their regular (non-homogeneous) representation $(x, y) = [\frac{x_1}{x_3}, \frac{x_2}{x_3}]$ in $\mathbb{R}^2$?

First given the equation $l^T x = 0$, we know that $l_1 \cdot x_1 + l_2 \cdot x_2 + l_3 \cdot x_3 = 0$

Divide x_3 l_3 to obtain non-homogeneous representation $\frac{l_1}{l_3} \cdot x_1 + \frac{l_2}{l_3} \cdot x_2 + 0$

Substituting a and b into the equation, we will get $l_1 \cdot x_1 + l_2 \cdot x_2 + 0 = 0$

We will then obtain the following $a = -\frac{l_1}{l_2}$ $b = -\frac{l_2}{l_2}$

$$a = -\frac{l_1}{l_2}$$

$$b = -\frac{l_2}{l_2}$$

Problem 5 (epipolar lines in normalized and non-normalized images)

Given a matrix of intrinsic camera parameters K and essential matrix E between two views (K_1 and E) such that $x_2^T E x_1 = 0$ for any corresponding points, write expressions for the following:

(a) given homogeneous normalized point x_1' in image 1, specify 3-vector l_2' describing the corresponding epipolar line of normalized points in image 2:

$l_2' = Ex_1'$

(b) given homogeneous normalized point x_2' in image 2, specify 3-vector l_1' describing the corresponding epipolar line of normalized points in image 1:

$l_2' = x_2'^T E$

(c) assuming the 3-vectors l_1' of normalized image points, what is a 3-vector representation l for the line formed by the corresponding points on the two (unnormalized) camera images:

$l = K^{-T} l'$

Problem 6 (least squares for triangulation)

Describe your approach to triangulation using two matched feature points $x_1 = [u_1, v_1, 1]^T$ and $x_2 = [u_2, v_2, 1]^T$ in two views with given projection matrices P_1 and P_2. You should find 3D point $X = [X_1, X_2, X_3, 1]^T$ and two scalars u_1, u_2 such that $P_1 X = u_1 x_1$ and $P_2 X = u_2 x_2$. Be specific as you will need this for your programming part (see Use notation $M_i^T q$ to denote the i-th row vector of matrix M.)

You should use the first approach described for homography estimation in topic 6. In particular, you can formulate the problem as $AX = 0$; define matrix A in terms of x_1, x_2, P_1, and P_2; write equations in an overdetermined system of 4 linear equations $A_{11}[X_1, X_2, X_3]^T = -A_{14}$ and specify the solution minimizing the sum of squared errors.

Can you characterize geometrically the case when your solution satisfies $A_{11}[X_1, X_2, X_3]^T = -A_{14}$ exactly?

Solution:

Given $X = [X_1, X_2, X_3]^T$ and P_1, P_2 we can expand it to obtain the following equations:

1. $P_1[1,1] \cdot u_1 x_1$
2. $P_1[1,2] \cdot u_1 x_2$
3. $P_1[1,3] \cdot u_1 x_3$
4. $P_1[1,4] \cdot u_1 x_4$
5. $P_1[2,1] \cdot u_1 x_1$
6. $P_1[2,2] \cdot u_1 x_2$
7. $P_1[2,3] \cdot u_1 x_3$
8. $P_1[2,4] \cdot u_1 x_4$
9. $P_1[3,1] \cdot u_1 x_1$
10. $P_1[3,2] \cdot u_1 x_2$
11. $P_1[3,3] \cdot u_1 x_3$
12. $P_1[3,4] \cdot u_1 x_4$
13. $P_2[1,1] \cdot u_2 x_1$
14. $P_2[1,2] \cdot u_2 x_2$
15. $P_2[1,3] \cdot u_2 x_3$
16. $P_2[1,4] \cdot u_2 x_4$
17. $P_2[2,1] \cdot u_2 x_1$
18. $P_2[2,2] \cdot u_2 x_2$
19. $P_2[2,3] \cdot u_2 x_3$
20. $P_2[2,4] \cdot u_2 x_4$
21. $P_2[3,1] \cdot u_2 x_1$
22. $P_2[3,2] \cdot u_2 x_2$
23. $P_2[3,3] \cdot u_2 x_3$
24. $P_2[3,4] \cdot u_2 x_4$

$A_{11} = [A_1[1], A_1[2], A_1[3]]^T$ and $A_{14} = [A_1[1], A_1[2], A_1[3]]^T$

Thus it can be written as so that matrix

$$A = \begin{bmatrix} A_1[1] & A_1[2] & -u_1 \\ A_1[1] & A_1[2] & -u_1 \\ A_1[1] & A_1[2] & -u_1 \\ A_1[1] & A_1[2] & -u_1 \end{bmatrix}$$

Thus a solution to this least square problem can be obtained by:

$$\min_{\|X\|=1} \|AX - b\|^2 \Longleftrightarrow X = (A_{11})^{-1} \cdot -A_{14}$$

If by solution satisfies $A_{11}[X_1, X_2, X_3]^T = -A_{14}$ exactly, this implies that the points u_1, u_2 are exactly on the corresponding epipolar lines and the corresponding rays intersect in 3D perfectly.

Problem 7 (the programming part)

Structure from Motion

NOTE: Steps 0-3 and 10 are given, other steps needs to be implemented.

Step 0: Loading two camera views and camera's intrinsic matrix K

```
%matplotlib inline
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d
import mpl_toolkits.mplot3d.proj3d
import skimage.feature.peak_local_max
from skimage.feature.corner_harris, corner_peaks, plot_matches, BRIEF, match_descriptors
from skimage.transform.ransac, AffineTransform, EuclideanTransform, SimilarityTransform, ProjectiveTransform
from skimage.measure.fit import LineModel
from skimage.io import imread, imshow, imsave
# Calibrated (K) intrinsic parameters for images 1 and image 2
K = np.array([[2759.48, 0.00000000000000, 1520.69], # corresponding matrix K (intrinsic parameters)
[0.00000000000000, 2759.48, 1006.81],
[0.00000000000000, 0.00000000000000, 1.00000000000000]])
imsize = (2704,3384)
# Load images
im1 = io.imread('./gate_K_small/DSC_0349.JPG')
im2 = io.imread('./gate_K_small/DSC_0350.JPG')

# K = 3x3 matrix of the intrinsic parameters
K = np.array([[2759.48, 0.00000000000000, 1520.69],
[0.00000000000000, 2759.48, 1006.81],
[0.00000000000000, 0.00000000000000, 1.00000000000000]])

plot_figure(1,figsize=(10, 6))
subplot(1,2,1,xticks=[],yticks=[])
imshow(im1)
subplot(1,2,2,xticks=[],yticks=[])
imshow(im2)
plot_save()
```

Step 1: Feature detection (e.g. corners)

```
# Define corner_peaks and match_descriptors parameters that have proven great preselection on gate images
# Feel free to experiment with other values and other functions (e.g. corner_harris)
keypoints1 = corner_peaks(corner_harris(rgb2gray(im1)), threshold_rel=0.001, min_distance=5)
keypoints2 = corner_peaks(corner_harris(rgb2gray(im2)), threshold_rel=0.001, min_distance=5)

print ("The number of keypoints in image1 is: {0}".format(len(keypoints1)))
print ("The number of keypoints in image2 is: {0}".format(len(keypoints2)))

fig,ax=plot_figure(2,figsize (10, 6))
ax0 = ax[0].axis('off')
ax1 = ax[1].axis('off')
ax[0].imshow(im1)
ax[0].scatter(keypoints1[:, 1], keypoints1[:, 0],marker='+')
ax[1].imshow(im2)
ax[1].scatter(keypoints2[:, 1], keypoints2[:, 0],marker='+')

plot_save()
The number of keypoints in image1 is: 1274 and 1241
```

Step 2: Feature matching (e.g. BRIEF descriptor, a variant of SURF, SIFT, etc)

```
extractor1 = BRIEF()
extractor1.extract(rgb2gray(im1), keypoints1)
descriptors1 = keypoints1[extractor1.mask]
extractor2 = BRIEF(sigma=2.0, sample_seed=1)
extractor2.extract(rgb2gray(im2), keypoints2)
descriptors2 = keypoints2[extractor2.mask]

matches12 = match_descriptors(descriptors1, descriptors2, cross_check=True)

print ("The number of matches is: {0}".format(matches12.shape[0]))

fig,ax=plot_figure(2,figsize (10, 6))
ax0 = ax[0].axis('off')
ax1 = ax[1].axis('off')
plot_matches(ax0,ax1,im1, im2, keypoints1, keypoints2, matches12, c='b', matches_color = 'r')

plot_save()
The number of matches is: 1000
```

Step 3: Fundamental Matrix estimation using RANSAC

```
model, inliers = ransac((keypoints1[matches12[:, 0]],
keypoints2[matches12[:, 1]]),
EuclideanTransform, 4, 0.01, max_trials=1000)
print ("The number of inliers is: {0}".format(inliers.shape[0]))

# Compute inliers using estimated ratio(inliers/total matches)
inlier_ratio = float(inliers.shape[0]) / matches12.shape[0]
print ("Ratio: {0}".format(inlier_ratio))
# Compute essential matrix
E = K.T.dot(model.params).dot(K)

# Compute inliers keypoints coordinates
inlier_keypoints_left = keypoints1[matches12[inliers, 0], :][:, [1, 0]]
inlier_keypoints_right = keypoints2[matches12[inliers, 0], :][:, [1, 0]]

# Add ones to make them homogeneous
inlier_keypoints_left = np.hstack([inlier_keypoints_left, np.ones(inlier_keypoints_left.shape[0])[:, np.newaxis]])
inlier_keypoints_right = np.hstack([inlier_keypoints_right, np.ones(inlier_keypoints_right.shape[0])[:, np.newaxis]])

# Compute epipolar lines
epilines_right = np.dot(E, inlier_keypoints_left.T).T
epilines_left = np.dot(E.T, inlier_keypoints_right.T).T

# Define a function to draw epipolar lines
def draw_epilines(lines, shape, axis):
    for r in lines:
        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [shape[1], -(r[2]+r[0]*shape[1])/r[1]])
        axis.plot([x0, x1], [y0, y1], 'b')

fig,ax = plot_figure(2,figsize=(10, 6))
ax0 = ax[0].axis('off')
ax1 = ax[1].axis('off')
ax[0].imshow(im1)
ax[0].scatter(inlier_keypoints_left[:, 1], inlier_keypoints_left[:, 0],marker='+')
draw_epilines(epilines_right, im2.shape[:2], ax[1])
ax[1].imshow(im2)
ax[1].scatter(inlier_keypoints_right[:, 1], inlier_keypoints_right[:, 0],marker='+')

plot_save()
The number of inliers is 715
```

Step 4: Epipolar lines from F

```
# Compute epipole as nullspace of E(working in the images from the set of inlier pts)
inv_model = np.linalg.inv(model.inverse[0, 1:3, 1:3])
# Compute epipole using formula from Hartley and Zisserman
e = np.dot(inv_model, model.inverse[0, 1:3, 0])
print e.astype(int)
print inlier_keypoints_left[0].astype(int)
print inlier_keypoints_right[0].astype(int)
# Define a function to draw epipolar lines
def plot_epilines(epi, ax, pts, shape):
    for i, (r, p) in enumerate(zip(epi, pts)):
        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [shape[1], -(r[2]+r[0]*shape[1])/r[1]])
        ax.plot([x0, x1], [y0, y1], 'b')
        ax.scatter(p[1],p[0],marker='+')
# Compute epipolar lines
epilines_right = np.dot(E, inlier_keypoints_left.T).T
epilines_left = np.dot(E.T, inlier_keypoints_right.T).T

# Plot
fig,ax = plot_figure(2,figsize=(10, 6))
ax0 = ax[0].axis('off')
ax1 = ax[1].axis('off')
plot_epilines(epilines_right, ax[0], inlier_keypoints_left, im2.shape[:2])
ax[0].imshow(im1)
plot_epilines(epilines_left, ax[1], inlier_keypoints_right, im1.shape[:2])
ax[1].imshow(im2)
plot_save()
[ 1 -1 -1]
[1359 1000 1]
[1359 1000 1]
```

singular values for F

```
# Check singular values
U,S,V = np.linalg.svd(F)
print S
array([ 1., 1., 0.])
```

Step 6: Epipolar lines from F

```
# Compute epipole as nullspace of E(working in the images from the set of inlier pts)
inv_model = np.linalg.inv(model.inverse[0, 1:3, 1:3])
# Compute epipole using formula from Hartley and Zisserman
e = np.dot(inv_model, model.inverse[0, 1:3, 0])
print e.astype(int)
print inlier_keypoints_left[0].astype(int)
print inlier_keypoints_right[0].astype(int)
# Define a function to draw epipolar lines
def plot_epilines(epi, ax, pts, shape):
    for i, (r, p) in enumerate(zip(epi, pts)):
        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [shape[1], -(r[2]+r[0]*shape[1])/r[1]])
        ax.plot([x0, x1], [y0, y1], 'b')
        ax.scatter(p[1],p[0],marker='+')
# Compute epipolar lines
epilines_right = np.dot(E, inlier_keypoints_left.T).T
epilines_left = np.dot(E.T, inlier_keypoints_right.T).T

# Plot
fig,ax = plot_figure(2,figsize=(10, 6))
ax0 = ax[0].axis('off')
ax1 = ax[1].axis('off')
plot_epilines(epilines_right, ax[0], inlier_keypoints_left, im2.shape[:2])
ax[0].imshow(im1)
plot_epilines(epilines_left, ax[1], inlier_keypoints_right, im1.shape[:2])
ax[1].imshow(im2)
plot_save()
[ 1 -1 -1]
[1359 1000 1]
[1359 1000 1]
```


Step 5: Camera Normalization and Essential Matrix estimation using RANSAC

```
# Compute the epipole as the nullspace of the transpose of E
# Compute epipolar lines using the formula from Hartley and Zisserman
# Define a function to draw epipolar lines
def plot_epilines(epi, ax, pts, shape):
    for i, (r, p) in enumerate(zip(epi, pts)):
        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [shape[1], -(r[2]+r[0]*shape[1])/r[1]])
        ax.plot([x0, x1], [y0, y1], 'b')
        ax.scatter(p[1],p[0],marker='+')
# Compute epipolar lines
epilines_right = np.dot(E, inlier_keypoints_left.T).T
epilines_left = np.dot(E.T, inlier_keypoints_right.T).T

# Plot
fig,ax = plot_figure(2,figsize=(10, 6))
ax0 = ax[0].axis('off')
ax1 = ax[1].axis('off')
plot_epilines(epilines_right, ax[0], inlier_keypoints_left, im2.shape[:2])
ax[0].imshow(im1)
plot_epilines(epilines_left, ax[1], inlier_keypoints_right, im1.shape[:2])
ax[1].imshow(im2)
plot_save()
```

singular values for E

```
# Check singular values
U,S,V = np.linalg.svd(E)
print S
array([ 1.00000000e+00, 9.99999999e-01, 0.00000000e+00])
```

Step 7: Camera rotation and translation (four solutions)

Factorize essential matrix $E = [T]_\times R$ where E is rotation and T is a translation. Find solutions R_1, R_2 and T_1, T_2. Use camera 1 for world coordinates. Define projection matrix for camera 1 as $P_1 = [I|0]$ and compute four projection matrices for the second camera P_1, P_2, P_3, P_4.

Hint 1: for any multiplication out EA or multiply E, never $+$

Hint 2: Apply $\text{svd}(E)$ and $\text{diag}(1,1,0)$ returns $\text{diag}(1,1,0)$ only if the first orthogonal matrix is not decomposition $E = U\text{diag}(1,1,0)V^T$.

Warning: remember that python uses 0 as a starting index for the rows or columns in arrays. For example, $A[0]$ denotes the first row of matrix A, while $P_1[:,3]$ stands for the 3rd row of the corresponding projection matrix and $B[:,1]$ is the second column of the matrix.

```
# Recovery of the camera matrix and extrinsic parameters
W = np.array([[0,-1,0],[1,0,0],[0,0,1]])
U,S,V = np.linalg.svd(E)
# Four possibilities
P1 = np.hstack((np.eye(3), np.zeros((3,1))))
# Define variables
tx = np.dot(np.dot(U,W),np.dot(np.diag([1,1,0]),U.T))
R = np.dot(np.dot(U,W),V)
tx = np.dot(np.dot(U,W.T),np.dot(np.diag([1,1,0]),U.T))
R = np.dot(np.dot(U,W.T),V)
# Special empty matrix
P = np.zeros((3,4,4))
P[0] = np.dot(np.hstack((R, tx[:,2:])), P1)
P[1] = np.dot(np.hstack((R, -tx[:,2:])), P1)
P[2] = np.dot(np.hstack((R, tx[:,1:2])), P1)
P[3] = np.dot(np.hstack((R, -tx[:,1:2])), P1)
# Print
for i in range(4):
    print P[i]
# Force camera matrix
P = np.zeros((3,4,4))
P[0] = np.dot(np.hstack((R, tx[:,2:])), P1)
P[1] = np.dot(np.hstack((R, -tx[:,2:])), P1)
P[2] = np.dot(np.hstack((R, tx[:,1:2])), P1)
P[3] = np.dot(np.hstack((R, -tx[:,1:2])), P1)
# Print
for i in range(4):
    print P[i]
```

Summary of Structure-from-Motion (the remaining steps 8-11)

In them, 3D reconstruction steps are should be the same for all four possible solutions consistent with the projection matrices estimated in step 7. In all steps you should obtain 3D points corresponding to pairs of matched features that are inliers for estimated E (i.e. consistent with the epipolar geometry). Make sure to use normalized coordinates for these computations (i.e. pre-multiply image coordinates by K^{-1}). Compute the 3D points in the coordinate system of the first camera. Use the provided code to plot the 3D point clouds (visualization step 10 of the assignment). Your plot should contain 4D (left camera positions, right camera positions, image centers and triangulated points (like in the lecture slides). Use different colors for the 3D points that correspond to different solutions. Use the provided code to project the 3D points back into each camera, convert to uncalibrated coordinates, and display these projected points (over-lay) together with the original features (use blue). Observe if the one test that has points are in front of each image.

Step 8: Triangulation (four solutions)

```
# Compute 3D points using the formula from Hartley and Zisserman (DLT method)
# Define variables or appropriate data structures for storing the 3D points
# Loop over all four solutions
# Loop over all inlier matches
# Define the matrix A for the DLT method (Slide 24 of
# lecture 6)
# Define the matrix U from the SVD of A
# Define the 3D point for this match
# Store the 3D point
# Print the shapes of the 3D point cloud for each solution
# Example:
# print "Shape of the 3D point cloud for solution 1: ", P_1.shape
# Keep only those points that are in front of both cameras
# Example:
# P_1 = P_1[:,:,P_1[:,2,3] > 0]
# Return the 3D point clouds for all four solutions
# Example:
# return P_1, P_2, P_3, P_4
# Call the function
P_1, P_2, P_3, P_4 = triangulate(inlier_keypoints_left, inlier_keypoints_right, P1, P[0], P[1], P[2], P[3])
Shape of the 3D point cloud for solution 1: (3, 715)
Shape of the 3D point cloud for solution 2: (3, 715)
Shape of the 3D point cloud for solution 3: (3, 715)
Shape of the 3D point cloud for solution 4: (3, 715)
```

Step 9: Triangulation (four solutions)

```
# Define variables or appropriate data structures for storing the 3D points
# Loop over all four solutions
# Loop over all inlier matches
# Define the matrix A for the DLT method (Slide 24 of
# lecture 6)
# Define the matrix U from the SVD of A
# Define the 3D point for this match
# Store the 3D point
# Print the shapes of the 3D point cloud for each solution
# Example:
# print "Shape of the 3D point cloud for solution 1: ", P_1.shape
# Keep only those points that are in front of both cameras
# Example:
# P_1 = P_1[:,:,P_1[:,2,3] > 0]
# Return the 3D point clouds for all four solutions
# Example:
# return P_1, P_2, P_3, P_4
# Call the function
P_1, P_2, P_3, P_4 = triangulate(inlier_keypoints_left, inlier_keypoints_right, P1, P[0], P[1], P[2], P[3])
Shape of the 3D point cloud for solution 1: (3, 715)
Shape of the 3D point cloud for solution 2: (3, 715)
Shape of the 3D point cloud for solution 3: (3, 715)
Shape of the 3D point cloud for solution 4: (3, 715)
```

Step 10: Visualization

```
# Define a function to draw the 3D point cloud
def plot_3d(points,c='b',marker='o'):
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(points[0,:], points[1,:], points[2,:],c=c,marker=marker)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    return fig,ax
# Example:
# fig,ax = plot_3d(P_1,'r','o')
# Plot the 3D point clouds for all four solutions
# Example:
# fig,ax = plot_3d(P_1,'r','o')
# fig,ax = plot_3d(P_2,'b','o',ax)
# fig,ax = plot_3d(P_3,'g','o',ax)
# fig,ax = plot_3d(P_4,'y','o',ax)
fig,ax = plot_3d(P_1,'r','o')
fig,ax = plot_3d(P_2,'b','o',ax)
fig,ax = plot_3d(P_3,'g','o',ax)
fig,ax = plot_3d(P_4,'y','o',ax)
```

Step 11: Visualization

```
# Define variables or appropriate data structures for storing the projected points
# Loop over all four solutions
# Loop over all inlier matches
# Define the matrix A for the DLT method (Slide 24 of
# lecture 6)
# Define the matrix U from the SVD of A
# Define the 3D point for this match
# Store the 3D point
# Print the shapes of the 3D point cloud for each solution
# Example:
# print "Shape of the 3D point cloud for solution 1: ", P_1.shape
# Keep only those points that are in front of both cameras
# Example:
# P_1 = P_1[:,:,P_1[:,2,3] > 0]
# Return the 3D point clouds for all four solutions
# Example:
# return P_1, P_2, P_3, P_4
# Call the function
P_1, P_2, P_3, P_4 = triangulate(inlier_keypoints_left, inlier_keypoints_right, P1, P[0], P[1], P[2], P[3])
Shape of the 3D point cloud for solution 1: (3, 715)
Shape of the 3D point cloud for solution 2: (3, 715)
Shape of the 3D point cloud for solution 3: (3, 715)
Shape of the 3D point cloud for solution 4: (3, 715)
```

```
Ad = np.zeros((n, 4, 4))
for i in range(x1.shape[0]):
    temp = np.array([identity[0], identity[1], R2[0], R2[1]])
    Ad[i] = np.append(temp, np.array([[x1[i][0]], [x1[i][1]], T2[0]-x2[i][0], T2[1]-x2[i][1]]), 1)
```

Solution using least squares: assume homogeneous 3D point $X = [X_1, X_2, X_3, 1]$. Then, $AX = 0$ gives 4 equations for 3 unknowns. Use approach 1 (inhomogeneous least squares) discussed for homography estimation (Topic 6).

In [12]:
```
# least squares for solving linear system A_{0:2} X_{0:2} = - A_3
Aa_02 = Aa[:, :, :3]        # the first 3 columns of 3x4 matrix A
Aa_3  = Aa[:, :, 3]         # the last column on 3x4 matrix A

Ab_02 = Ab[:, :, :3]
Ab_3  = Ab[:, :, 3]

Ac_02 = Ac[:, :, :3]
Ac_3  = Ac[:, :, 3]

Ad_02 = Ad[:, :, :3]
Ad_3  = Ad[:, :, 3]


# # Nx3 matrices: N rows with 3D point coordinates for N reconstructed points (N=num_inliers)
Xa = np.zeros((n, 3))
for i in range(n):
    Xa[i] = np.dot(np.linalg.pinv(Aa_02[i]), Aa_3[i])

Xb = np.zeros((n, 3))
for i in range(n):
    Xb[i] = np.matmul(np.linalg.pinv(Ab_02[i]), Ab_3[i])

Xc = np.zeros((n, 3))
for i in range(n):
    Xc[i] = np.matmul(np.linalg.pinv(Ac_02[i]), Ac_3[i])

Xd = np.zeros((n, 3))
for i in range(n):
    Xd[i] = np.matmul(np.linalg.pinv(Ad_02[i]), Ad_3[i])
```

Step 9: Camera positioning in 3D (four solutions)

In this step you will compute location of each cameras' optical center and its (calibrated) image center as points in 3D (world coordinate system). The next step 10 visualaizes the computed cameras' optical centers in red and image centers in green.

In [13]:
```
# camera's optical centers (for pair of cameras) as points in 3D world coordinate system
# rxt matrices: two rows with 3D point coordinates for the first and second camera
Ca = np.zeros((2,3))
Ca[1] = np.dot(R1, Ca[0])+T1.transpose()
Cb = np.zeros((2,3))
Cb[1] = np.dot(R1, Cb[0])+T2.transpose()
Cc = np.zeros((2,3))
Cc[1] = np.dot(R2, Cc[0])+T1.transpose()
Cd = np.zeros((2,3))
Cd[1] = np.dot(R2, Cd[0])+T2.transpose()

# calibrated/normalized image centers (for pair of cameras) as points in 3D world coordinate system
# 2x3 matrices: two rows with 3D pnt## Step 9: Cameras positions/orientations in 3D  (four solutions)int coordinates for the first and second camera
Qa = np.zeros((2,3))
Qa[0][-1] = 1
Qa[1] = np.dot(R1, Qa[0])+T1.transpose()

Qb = np.zeros((2,3))
Qb[0][-1] = 1
Qb[1] = np.dot(R1, Qb[0])+T2.transpose()

Qc = np.zeros((2,3))
Qc[0][-1] = 1
Qc[1] = np.dot(R2, Qc[0])+T1.transpose()

Qd = np.zeros((2,3))
Qd[0][-1] = 1
Qd[1] = np.dot(R2, Qd[0])+T2.transpose()

print (Cc)
print (Qc)
```

```
[[0.        0.        0.        ]
 [0.91563089 0.13672293 0.37805677]]
[[ 0.          0.          1.          ]
 [ 1.67387752  0.25765734 -0.26259673]]
```

Step 10 (fully implemented): 3D visualization of cameras and triangulated points (four solutions)

In [14]:
```
# visualization part
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(10,figsize = (10, 10))

axl0_1 = plt.subplot(221, projection='3d')
plt.title('Solution a')
axl0_1.scatter(Xa[:,0],Xa[:,1],Xa[:,2], c='b', marker='p')
axl0_1.scatter(Ca[:,0],Ca[:,1],Ca[:,2], c='r', marker='p')
axl0_1.scatter(Qa[:,0],Qa[:,1],Qa[:,2], c='g', marker='p')

axl0_2 = plt.subplot(222, projection='3d')
plt.title('Solution b')
axl0_2.scatter(Xb[:,0],Xb[:,1],Xb[:,2], c='b', marker='p')
axl0_2.scatter(Cb[:,0],Cb[:,1],Cb[:,2], c='r', marker='p')
axl0_2.scatter(Qb[:,0],Qb[:,1],Qb[:,2], c='g', marker='p')

axl0_3 = plt.subplot(223, projection='3d')
plt.title('Solution c')
axl0_3.scatter(Xc[:,0],Xc[:,1],Xc[:,2], c='b', marker='p')
axl0_3.scatter(Cc[:,0],Cc[:,1],Cc[:,2], c='r', marker='p')
axl0_3.scatter(Qc[:,0],Qc[:,1],Qc[:,2], c='g', marker='p')

axl0_4 = plt.subplot(224, projection='3d')
plt.title('Solution d')
axl0_4.scatter(Xd[:,0],Xd[:,1],Xd[:,2], c='b', marker='p')
axl0_4.scatter(Cd[:,0],Cd[:,1],Cd[:,2], c='r', marker='p')
axl0_4.scatter(Qd[:,0],Qd[:,1],Qd[:,2], c='g', marker='p')

plt.show()
```

Step 11: Reprojection errors

In [15]:
```
# Randomly select N=50 matches (pairs of features in two images) from the set of inliers for E
N = 50
ind_sample2 = np.random.choice(num_inliers, N, replace = False)

# Indicate (E) inlier matches in image 1 and image 2
plt.figure(11,figsize = (10, 4))
axl1_1 = plt.subplot(121)
plt.imshow(imL)
plt.plot(ptsL[ind[E_inliers][ind_sample2], 0], ptsL[ind[E_inliers][ind_sample2], 1], 'ob')
axl1_2 = plt.subplot(122)
plt.imshow(imR)
plt.plot(ptsR[ind[E_inliers][ind_sample2], 0], ptsR[ind[E_inliers][ind_sample2], 1], 'ob')

xl = ptsL[ind[E_inliers][ind_sample2], :]
xr = ptsR[ind[E_inliers][ind_sample2], :]

# project reconstructed 3D points onto both images and display them in red color
Ma = np.zeros((N, 4, 4))
for i in range(x1.shape[0]):
    temp = np.array([identity[0], identity[1], R1[0], R1[1]])
    Ma[i] = np.append(temp, np.array([[x1[i][0]], [x1[i][1]], T1[0]-x2[i][0], T1[1]-x2[i][1]]), 1)

Ma_02 = Ma[:, :, :3]
Ma_3  = Ma[:, :, 3]

Xcorrect = np.zeros((N, 3))
for i in range(N):
    Xcorrect[i] = np.matmul(np.linalg.pinv(Ma_02[i]), Ma_3[i])

# a. convert correct points (Xa, Xb, Xc, or Xd) to homogeneous 4 vectors
Xpoints = np.zeros((N, 4))
for i in range(N):
    Xpoints[i] = np.append(Xcorrect[i], 1)

# b. project homogeneous 3D points (onto uncalibrated cameras) using correct Projection matrices (KPw and, e.g. KPa)
c1 = np.zeros((N, 3))
c2 = np.zeros((N, 3))
for i in range(N):
    c1[i] = np.dot(K, np.dot(Pw, Xpoints[i]))
    c2[i] = np.dot(K, np.dot(Pb, Xpoints[i]))

# c. convert to regular (inhomogeneous) point
ptsL_proj = np.zeros((N,2))
ptsR_proj = np.zeros((N,2))
for i in range(n):
    ptsL_proj[i] = np.array([c1[i][0]/c1[i][2], c1[i][1]/c1[i][2]])
    ptsR_proj[i] = np.array([c2[i][0]/c2[i][2], c2[i][1]/c2[i][2]])

axl1_1.plot(ptsL_proj[:,0], ptsL_proj[:,1], '.r')
axl1_2.plot(ptsR_proj[:,0], ptsR_proj[:,1], '.r')

plt.show()
```

Question: how different are projected points for SfM solutions a, b, c, and d? Explain.

Answer: