1

(a)

```
w0 = (points1[:,0] * points2[:,0]).reshape(points2.shape[0], 1)
w1 = (points1[:,1] * points2[:,0]).reshape(points2.shape[0], 1)
w2 = (points2[:,0]).reshape(points2.shape[0], 1)
w3 = (points1[:,0] * points2[:,1]).reshape(points2.shape[0], 1)
w4 = (points1[:,1] * points2[:,1]).reshape(points2.shape[0], 1)
w5 = (points2[:,1]).reshape(points2.shape[0], 1)
w6 = (points1[:,0]).reshape(points2.shape[0], 1)
w7 = (points1[:,1]).reshape(points2.shape[0], 1)
w8 = np.ones(w7.shape)
W = np.concatenate([w0,w1,w2,w3,w4,w5,w6,w7,w8], 1)
_, _, V_T = np.linalg.svd(W)
F11,F12,F13,F21,F22,F23,F31,F32,F33 = V_T[-1]
F_hat = np.array([[F11,F12,F13], [F21,F22,F23], [F31,F32,F33]])
U, S, V_T = np.linalg.svd(F_hat)
S_new = np.array([[S[0],0,0], [0,S[1],0], [0,0,0]])
F = U.dot(S_new).dot(V_T)
return F
```

```
--------------------------------------------------------------------------------
Set: data/set1
--------------------------------------------------------------------------------
Fundamental Matrix from LLS  8-point algorithm:
[[  1.55218081e-06  -8.18161523e-06  -1.50440111e-03]
 [ -5.86997052e-06  -3.02892219e-07  -1.13607605e-02]
 [ -3.52312036e-03   1.41453881e-02   9.99828068e-01]]

--------------------------------------------------------------------------------
Set: data/set2
--------------------------------------------------------------------------------
Fundamental Matrix from LLS  8-point algorithm:
[[ -5.63087200e-06   2.74976583e-05  -6.42650411e-03]
 [ -2.77622828e-05  -6.74748522e-06   1.52182033e-02]
 [  1.07623595e-02  -1.22519240e-02  -9.99730547e-01]]
```

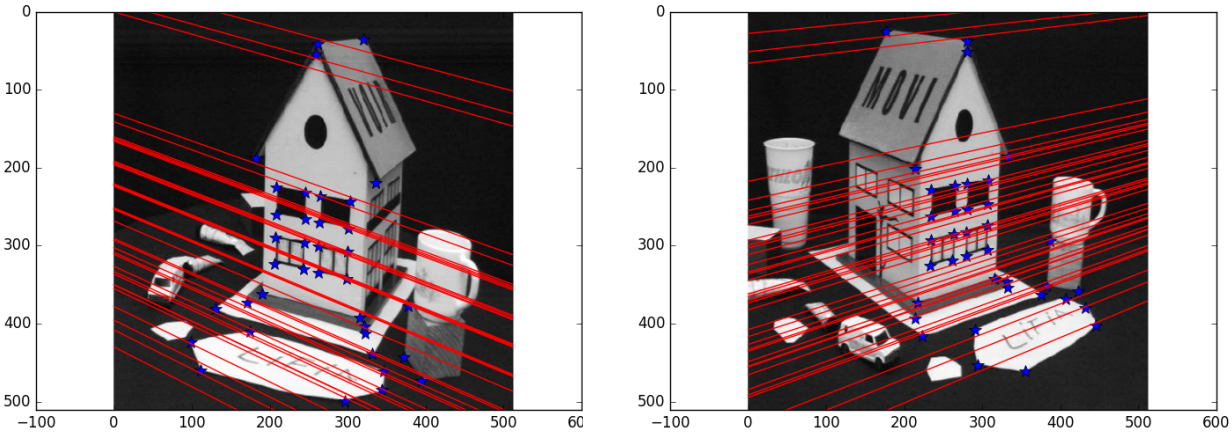
(b)

```
centroid1 = np.mean(points1, 0)
centroid1[2] = 0
centroid2 = np.mean(points2, 0)
centroid2[2] = 0
p1 = points1 - centroid1
p2 = points2 - centroid2
tx1 = -centroid1[0]
```

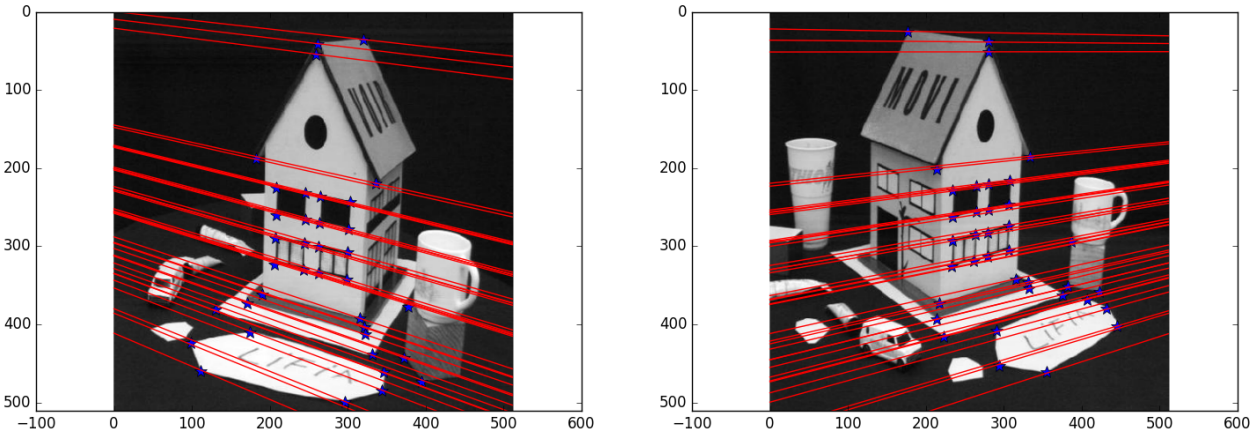
```
        ty1 = -centroid1[1]
        tx2 = -centroid2[0]
        ty2 = -centroid2[1]
        temp1 = p1 * p1
        temp1 = np.mean(temp1, 0)
        temp1 = np.sqrt(2 / temp1)
        temp1[2] = 1
        sx1 = temp1[0]
        sy1 = temp1[1]
        p1 *= temp1
        #print 'p1:', p1
        temp2 = p2 * p2
        temp2 = np.mean(temp2, 0)
        temp2 = np.sqrt(2 / temp2)
        temp2[2] = 1
        sx2 = temp2[0]
        sy2 = temp2[1]
        p2 *= temp2
        T1t = np.array([[1,0,tx1], [0,1,ty1], [0,0,1]])
        T1s = np.array([[sx1,0,0], [0,sy1,0], [0,0,1]])
        T1 = T1s.dot(T1t)
        T2t = np.array([[1,0,tx2], [0,1,ty2], [0,0,1]])
        T2s = np.array([[sx2,0,0], [0,sy2,0], [0,0,1]])
        T2 = T2s.dot(T2t)
        Fq = lls_eight_point_alg(p1, p2)
        F = T2.T.dot(Fq).dot(T1)
        return F

--------------------------------------------------------------------------------
Set: data/set1
--------------------------------------------------------------------------------
Fundamental Matrix from normalized 8-point algorithm:
[[  8.32296606e-07  -6.53427327e-06   1.29643474e-04]
 [ -5.85973613e-06  -3.83263853e-07  -7.78370990e-03]
 [ -1.06519134e-03   1.06779492e-02   1.77016125e-01]]

--------------------------------------------------------------------------------
Set: data/set2
--------------------------------------------------------------------------------
Fundamental Matrix from normalized 8-point algorithm:
[[ -2.20751997e-07   3.66585191e-06  -2.28591912e-04]
 [  5.27635069e-06   4.67194848e-07   1.02087044e-02]
 [  3.44169986e-04  -1.23948379e-02  -3.40804152e-03]]
```

(c)

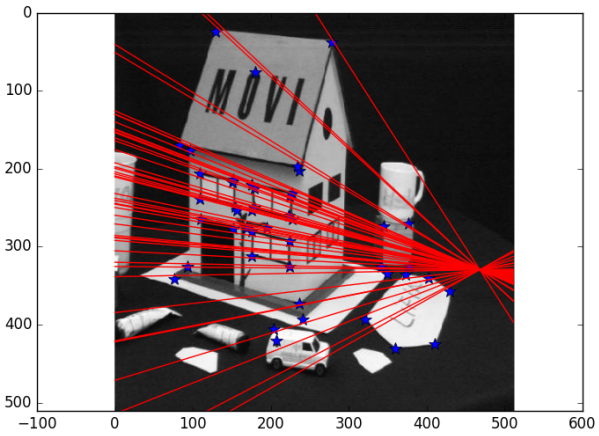
Set1 LLS

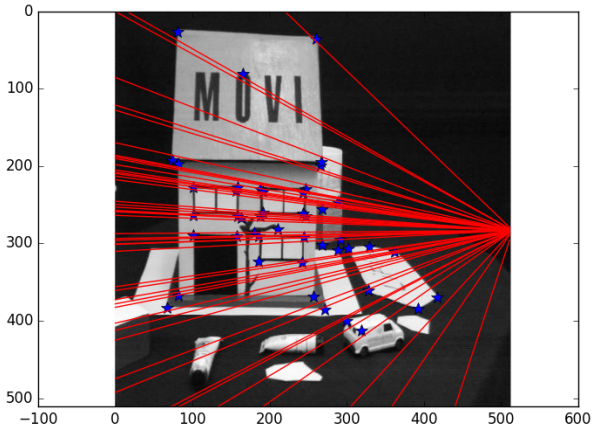


Set1 normalized

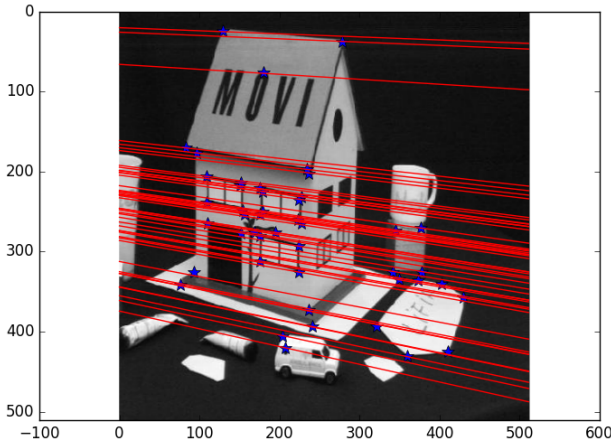


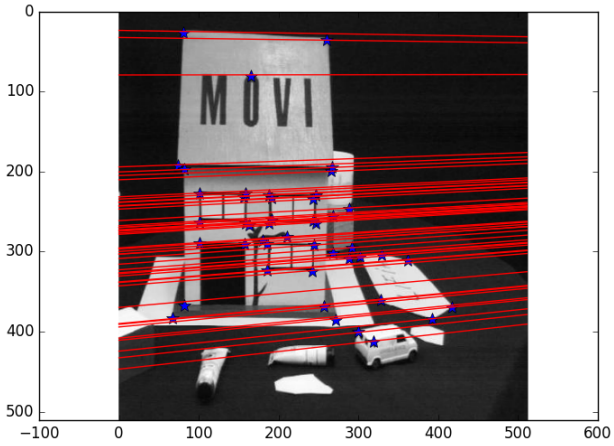
Set2 LLS





Set2 normalized





code:

```
l1 = (F.T.dot(points2.T)).T
l2 = (F.dot(points1.T)).T
plt.imshow(im1, cmap=plt.cm.gray)
```

```
        for point in points1:
                plt.plot(point[0], point[1], marker='*', markersize=10, color="blue")
        for line in l1:
                x = np.array(range(512))
                a, b, c = line
                y = -a/b*x - c/b
                plt.ylim((511,0))
                plt.plot(x, y, lw=1, color="red")
        plt.show()
        plt.imshow(im2, cmap=plt.cm.gray)
        for point in points2:
                plt.plot(point[0], point[1], marker='*', markersize=10, color="blue")
        for line in l2:
                x = np.array(range(512))
                a, b, c = line
                y = -a/b*x - c/b
                plt.ylim((511,0))
                plt.plot(x, y, lw=1, color="red")
        plt.show()
```

(d)

```
        l1 = (F.T.dot(points2.T)).T
        d_tot = 0
        for i in range(len(l1)):
                a, b, c = l1[i]
                x = points1[i,0]
                y = points1[i,1]
                d_tot += np.absolute(a*x + b*y + c) / np.sqrt(a**2 + b**2)
        average_distance = d_tot/len(l1)
        return average_distance
```

```
--------------------------------------------------------------------------------
Set: data/set1
--------------------------------------------------------------------------------
Distance to lines in image 1 for LLS: 28.0256629375
Distance to lines in image 2 for LLS: 25.1628758
p'^T F p = 0.0364469608237
Distance to lines in image 1 for normalized: 0.884401605649
Distance to lines in image 2 for normalized: 0.824224706542

--------------------------------------------------------------------------------
Set: data/set2
--------------------------------------------------------------------------------
Distance to lines in image 1 for LLS: 9.70143882946
Distance to lines in image 2 for LLS: 14.5682271905
p'^T F p = 0.0480279237456
```

Distance to lines in image 1 for normalized: 0.891400776488
Distance to lines in image 2 for normalized: 0.893528896442

2

(a)

```
    _, _, V_T = np.linalg.svd(F)
    epipole = V_T[-1]
    z = epipole[2]
    epipole /= z
    return epipole
```

e1 [-1.31864989e+03 -1.48121193e+02   1.00000000e+00]
e2 [1.63121737e+03   4.99110086e+01   1.00000000e+00]

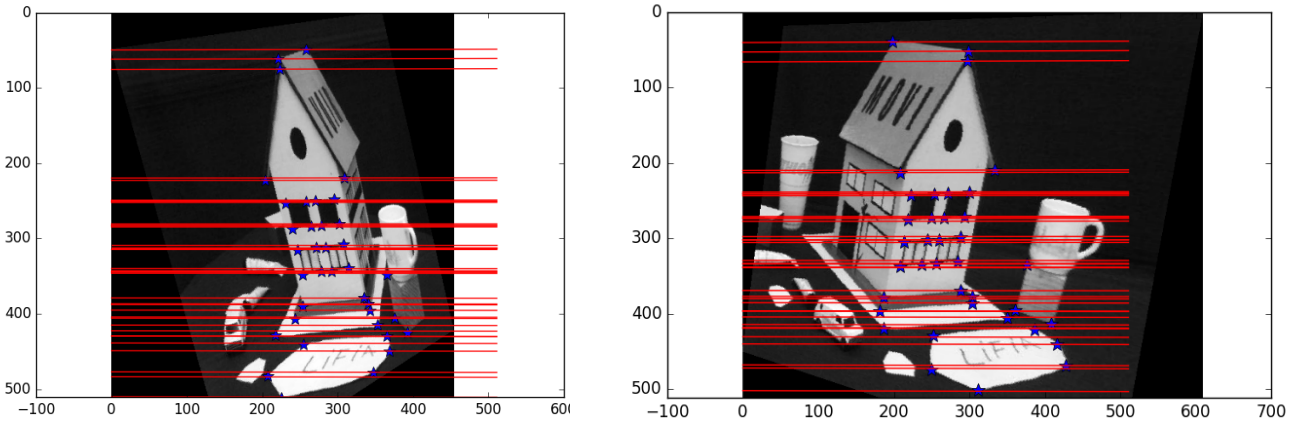
(b)

```
    t = im2.shape[0]/2
    T = np.array([[1,0,-t], [0,1,-t], [0,0,1]])
    x, y, _ = T.dot(e2)
    r1 = x / np.sqrt(x**2 + y**2)
    r2 = y / np.sqrt(x**2 + y**2)
    R = np.array([[r1,r2,0], [-r2,r1,0], [0,0,1]])
    f, _, _ = R.dot(T).dot(e2)
    G = np.array([[1,0,0], [0,1,0], [-1/f,0,1]])
    H2 = inv(T).dot(G).dot(R).dot(T)

    ex = np.array([[0,-e2[2],e2[1]], [e2[2],0,-e2[0]], [-e2[1],e2[0],0]])
    M = ex.dot(F) + e2.reshape(3,1)
    W = (H2.dot(M).dot(points1.T)).T
    W /= W[:,2].reshape(len(W), 1)
    b = (H2.dot(points2.T)).T
    b /= b[:,2].reshape(len(b), 1)
    b = b[:,0].reshape(len(b), 1)
    a = np.linalg.lstsq(W, b)[0]
    HA = np.array([[a[0,0], a[1,0], a[2,0]], [0,1,0], [0,0,1]])
    H1 = HA.dot(H2).dot(M)
    return H1, H2
```

H1:
[[-1.42788129e+01 -4.96476665e+00 -1.51088638e+02]
 [1.72427867e+00 -1.77203381e+01 -3.51037741e+02]
 [-1.08629453e-02 -2.58074561e-03 -1.47066847e+01]]

H2:
[[8.06893126e-01 -1.20920368e-01 8.03909740e+01]
[-3.38593618e-02 1.01624068e+00 4.51038177e+00]
[-7.11186107e-04 1.06577789e-04 1.15477973e+00]]

(c)

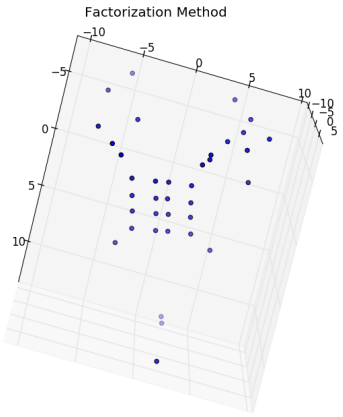


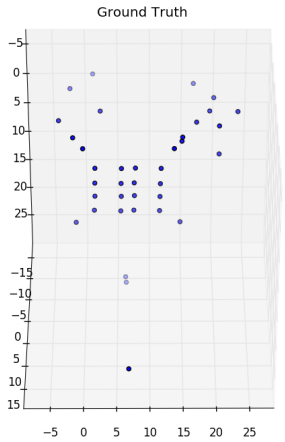
3

(a)

```
centroid1 = np.mean(points_im1, axis=0)
centroid2 = np.mean(points_im2, axis=0)
im1 = points_im1 - centroid1
im2 = points_im2 - centroid2
im1 = im1.T[:2]
im2 = im2.T[:2]
D = np.concatenate([im1, im2], 0)
U, W, V_T = np.linalg.svd(D)
M = U[:,:3]
W3 = np.array([[W[0],0,0], [0,W[1],0], [0,0,W[2]]])
S = W3.dot(V_T[:3])
return S, M
```

(b)



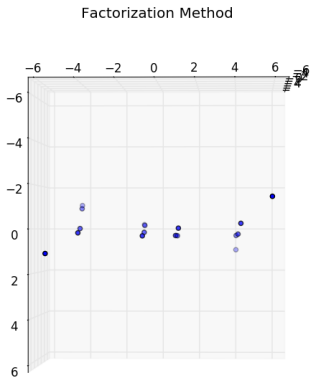


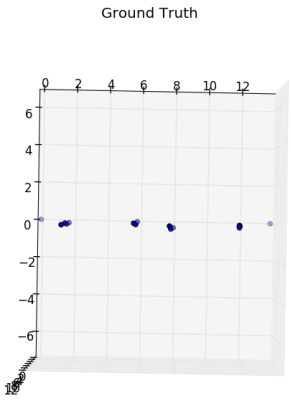
The results look the same except for a scaling and rotation. This is because the structure and motion matrices obtained are not a unique solution. D = M*S, but also D = (M*H)*(inv(H)*S), where H is an arbitrary affine transformation matrix. Therefore the solution can be obtained only up to an affine transformation.

(c)

4 singular values: [959.5852216 540.47613178 184.43174791 27.9151956]
The 4 non-zero singular values are due to measurement noise and affine camera approximation. In the ideal world, you expect to get just 3 non-zero singular values because matrix D has rank 3.

(d)





The ground truth has all the points approximately on the same plane, whereas the calculated result has the points on a twisted surface (not on the same plane), so they are no longer similar. We took the ground truth 3D points all from the same plane which is perpendicular to the y-axis, as shown below. This means that we only have 2D information and we don't have information regarding the y-dimension, since all the points have y-dimension approximately 0. This resulted in not enough information to accurately predict the structure and motion matrices up to affine transformation.

3D points coordinates:

11.574016 -0.102297 8.170404
1.724139 -0.155021 8.133060
1.610283 -0.236623 10.606374
1.781023 -0.143071 3.385768
7.611127 -0.201772 10.578166
7.871800 -0.300644 3.468180
7.683315 -0.261206 5.948648
5.696035 -0.165845 5.976460
13.700000 0.000000 0.000000
11.743422 -0.256549 3.503563
11.515773 -0.148995 10.624404
5.624941 -0.097076 10.633288
5.715866 -0.154460 8.168373
5.771224 -0.022592 3.346165
0.000000 0.000000 0.000000
11.632162 -0.190778 6.043319
1.731994 -0.207632 5.938620
7.704337 -0.354395 8.192640

(e)

new singular values: [264.54396508 210.06072009 7.21921783 5.12857709]

In the new singular values, the 3rd and 4th singular values are very low and close to 0, whereas in the original singular values the 3rd value was 184 which is very far from 0. This indicates that the D matrix has roughly just rank of 2 instead of 3. This is due to the fact that all points are taken from the same plane which resulted in the loss of information about the y-dimension.

4

(a)

```
U, D, V_T = np.linalg.svd(E)
W = np.array([[0,-1,0], [1,0,0], [0,0,1]])
Q1 = U.dot(W).dot(V_T)
Q2 = U.dot(W.T).dot(V_T)
```

```
R1 = np.linalg.det(Q1) * Q1
R2 = np.linalg.det(Q2) * Q2
T1 = U[:,-1].reshape(U.shape[0], 1)
T2 = -T1
RT1 = np.concatenate([R1, T1], 1)
RT2 = np.concatenate([R1, T2], 1)
RT3 = np.concatenate([R2, T1], 1)
RT4 = np.concatenate([R2, T2], 1)
return RT1, RT2, RT3, RT4
```

Estimated RT:

```
[[ 0.98305251, -0.11787055, -0.14040758,  0.99941228],
 [-0.11925737, -0.99286228, -0.00147453, -0.00886961],
 [-0.13923158,  0.01819418, -0.99009269,  0.03311219]]

[[ 0.98305251, -0.11787055, -0.14040758, -0.99941228],
 [-0.11925737, -0.99286228, -0.00147453,  0.00886961],
 [-0.13923158,  0.01819418, -0.99009269, -0.03311219]]

[[ 0.97364135, -0.09878708, -0.20558119,  0.99941228],
 [ 0.10189204,  0.99478508,  0.00454512, -0.00886961],
 [ 0.2040601 , -0.02537241,  0.97862951,  0.03311219]]

[[ 0.97364135, -0.09878708, -0.20558119, -0.99941228],
 [ 0.10189204,  0.99478508,  0.00454512,  0.00886961],
 [ 0.2040601 , -0.02537241,  0.97862951, -0.03311219]]
```

(b)

```
P = image_points
M = camera_matrices
for i in range(P.shape[0]):
        ai = np.array([P[i,0]*M[i,2] - M[i,0], P[i,1]*M[i,2] - M[i,1]])
        if i == 0: A = ai
        if i > 0:
                A = np.concatenate([A, ai], 0)
_, _, V_T = np.linalg.svd(A)
P3D = V_T[-1]
P3D /= P3D[-1]
P3D = P3D[:3]
return P3D
```

```
--------------------------------------------------------------------------------
Part B: Check that the difference from expected point
is near zero
--------------------------------------------------------------------------------
Difference:  0.00292430530368
```

(c)

```
def reprojection_error(point_3d, image_points, camera_matrices):
    # TODO: Implement this method!
        P3d = point_3d
        P = image_points
        M = camera_matrices
        P3d = np.concatenate([P3d, [1]], 0).reshape(4,1)
        for i in range(P.shape[0]):
            pi = M[i].dot(P3d)
            pi = (pi/pi[-1])[:2].T
            if i == 0: P_prime = pi
            else: P_prime = np.concatenate([P_prime, pi], 0)
        error = P_prime - P
        error = error.reshape(-1)
        return error


def jacobian(point_3d, camera_matrices):
    # TODO: Implement this method!
        P3d = point_3d
        M = camera_matrices
        P3d = np.concatenate([P3d, [1]], 0).reshape(4,1)
        for i in range(M.shape[0]):
            pi = M[i].dot(P3d)
            jix = ((pi[2]*M[i,0,:3] - pi[0]*M[i,2,:3]) / pi[2]**2).reshape(1,3)
            jiy = ((pi[2]*M[i,1,:3] - pi[1]*M[i,2,:3]) / pi[2]**2).reshape(1,3)
            if i == 0: J = np.concatenate([jix, jiy], 0)
            else: J = np.concatenate([J, jix, jiy], 0)
        return J
```

```
Part C: Check that the difference from expected error/Jacobian
is near zero
--------------------------------------------------------------------------------
error: [-0.00954584 -0.51714071  0.00593078  0.5016317 ]
Jacobian:
[[ 154.33943931    0.          -22.42541691]
 [   0.          154.33943931   36.51165089]
 [ 141.87950588  -14.27738422  -56.20341644]
 [  21.9792766   149.50628901   32.23425643]]

Error Difference:  8.30130013067e-07
Jacobian Difference:  1.81711570235e-08
```

(d)

```
P3d = linear_estimate_3d_point(image_points, camera_matrices)
for i in range(10):
        J = jacobian(P3d, camera_matrices)
        e = reprojection_error(P3d, image_points, camera_matrices)
        P3d = P3d - inv(J.T.dot(J)).dot(J.T).dot(e)
return P3d
```

Part D: Check that the reprojection error from nonlinear method
is lower than linear method

Linear method error: 98.7354235689
Nonlinear method error: 95.5948178485

(e)

```
def estimate_RT_from_E(E, image_points, K):
  # TODO: Implement this method!
    P = image_points
    RTs = estimate_initial_RT(E)
    zeros = np.zeros((3,1))
    M1 = np.concatenate([K, zeros], 1)
    max_count = 0
    for RT in RTs:
        positive_count = 0
        M2 = K.dot(RT)
        for pt in P:
            M = np.array([M1, M2])
            P3d1 = nonlinear_estimate_3d_point(pt, M)
            P3d1h = np.concatenate([P3d1, [1]], 0).reshape(4,1)
            P3d2 = RT.dot(P3d1h)
            if P3d1[2] >= 0 and P3d2[2] >= 0:
                positive_count += 1
                if positive_count > max_count:
                    max_count = positive_count
                    RT_true = RT
    return RT_true
```

Part E: Check your matrix against the example R,T

Example RT:

```
[[ 0.9736 -0.0988 -0.2056  0.9994]
 [ 0.1019  0.9948  0.0045 -0.0089]
 [ 0.2041 -0.0254  0.9786  0.0331]]
```

Estimated RT:
[[0.97364135 -0.09878708 -0.20558119 0.99941228]
 [0.10189204 0.99478508 0.00454512 -0.00886961]
 [0.2040601 -0.02537241 0.97862951 0.03311219]]

(f)