Problem Set 2 Solution

CS231A: Computer Vision
Stanford University
Spring 2017

Chi Zhang
SUID: 06116342
Date: May 4, 2017

1 Fundamental Matrix Estimation From Point Correspondences

The following code includes implementations of `lls_eight_point_alg()`, `normalized_eight_point_alg()`, `plot_epipolar_lines_on_images()` and `compute_distance_to_epipolar_lines()`:

```
'''
LLS_EIGHT_POINT_ALG  computes the fundamental matrix from matching points using
linear least squares eight point algorithm
Arguments:
    points1 - N points in the first image that match with points2
    points2 - N points in the second image that match with points1

    Both points1 and points2 are from the get_data_from_txt_file() method
Returns:
    F - the fundamental matrix such that (points2)^T * F * points1 = 0
Please see lecture notes and slides to see how the linear least squares eight
point algorithm works
'''
def lls_eight_point_alg(points1, points2):
    points_num = points2.shape[0]

    W = np.zeros((points_num, 9))
    for i in xrange(points_num):
        u1 = points1[i][0]
        v1 = points1[i][1]
        u2 = points2[i][0]
        v2 = points2[i][1]
        W[i] = np.array([u1*u2, u2*v1, u2, v2*u1, v1*v2, v2, u1, v1, 1])

    # compute F_hat
    U, s, VT = np.linalg.svd(W, full_matrices=True)
    f = VT[-1, :]
    F_hat = np.reshape(f, (3, 3))

    # compute F
    U, s_hat, VT = np.linalg.svd(F_hat, full_matrices=True)
    s = np.zeros((3, 3))
    s[0][0] = s_hat[0]
    s[1][1] = s_hat[1]
    F = np.dot(U, np.dot(s, VT))
```

```
    return F

'''
NORMALIZED_EIGHT_POINT_ALG  computes the fundamental matrix from matching points
using the normalized eight point algorithm
Arguments:
    points1 - N points in the first image that match with points2
    points2 - N points in the second image that match with points1

    Both points1 and points2 are from the get_data_from_txt_file() method
Returns:
    F - the fundamental matrix such that (points2)^T * F * points1 = 0
Please see lecture notes and slides to see how the normalized eight
point algorithm works
'''
def normalized_eight_point_alg(points1, points2):
    N = points1.shape[0]
    points1_uv = points1[:, 0:2]
    points2_uv = points2[:, 0:2]

    # normalization
    mean1 = np.mean(points1_uv, axis=0)
    mean2 = np.mean(points2_uv, axis=0)

    points1_center = points1_uv - mean1
    points2_center = points2_uv - mean2

    scale1 = np.sqrt(2/(np.sum(points1_center**2)/N * 1.0))
    scale2 = np.sqrt(2/(np.sum(points2_center**2)/N * 1.0))

    T1 = np.array([[scale1, 0, -mean1[0] * scale1],
                   [0, scale1, -mean1[1] * scale1],
                   [0, 0, 1]])

    T2 = np.array([[scale2, 0, -mean2[0] * scale2],
                   [0, scale2, -mean2[0] * scale2],
                   [0, 0, 1]])

    q1 = T1.dot(points1.T).T; q2 = T2.dot(points2.T).T
    Fq = lls_eight_point_alg(q1, q2)
    F = T2.T.dot(Fq).dot(T1)

    return F

'''
```

```
PLOT_EPIPOLAR_LINES_ON_IMAGES given a pair of images and corresponding points,
draws the epipolar lines on the images
Arguments:
    points1 - N points in the first image that match with points2
    points2 - N points in the second image that match with points1
    im1 - a HxW(xC) matrix that contains pixel values from the first image
    im2 - a HxW(xC) matrix that contains pixel values from the second image
    F - the fundamental matrix such that (points2)^T * F * points1 = 0

    Both points1 and points2 are from the get_data_from_txt_file() method
Returns:
    Nothing; instead, plots the two images with the matching points and
    their corresponding epipolar lines. See Figure 1 within the problem set
    handout for an example
'''
def plot_epipolar_lines_on_images(points1, points2, im1, im2, F):
    plt.subplot(1,2,1)
    ln1 = F.T.dot(points2.T)
    for i in xrange(ln1.shape[1]):
        plt.plot([0, im1.shape[1]], [-ln1[2][i]*1.0/ln1[1][i],
                 -(ln1[2][i]+ln1[0][i]*im1.shape[1])*1.0/ln1[1][i]], 'r')
        plt.plot([points1[i][0]], [points1[i][1]], 'b*')
    plt.imshow(im1, cmap='gray')

    plt.subplot(1,2,2)
    ln2 = F.dot(points1.T)
    for i in xrange(ln2.shape[1]):
        plt.plot([0, im2.shape[1]], [-ln2[2][i]*1.0/ln2[1][i],
                 -(ln2[2][i]+ln2[0][i]*im2.shape[1])/ln2[1][i]], 'r')
        plt.plot([points2[i][0]], [points2[i][1]], 'b*')
    plt.imshow(im2, cmap='gray')

'''
COMPUTE_DISTANCE_TO_EPIPOLAR_LINES  computes the average distance of a set a
points to their corresponding epipolar lines
Arguments:
    points1 - N points in the first image that match with points2
    points2 - N points in the second image that match with points1
    F - the fundamental matrix such that (points2)^T * F * points1 = 0

    Both points1 and points2 are from the get_data_from_txt_file() method
Returns:
    average_distance - the average distance of each point to the epipolar line
'''
def compute_distance_to_epipolar_lines(points1, points2, F):
```

```
    l = F.T.dot(points2.T)
    dist_sum = 0.0
    points_num = points1.shape[0]

    for i in xrange(points_num):
        dist_sum += np.abs(points1[i][0]*l[0][i] + points1[i][1]*l[1][i] + l[2][i])
                    / np.sqrt(l[0][i]**2 + l[1][i]**2)

    return dist_sum / points_num
```

OUTPUT

```
--------------------------------------------------------------------------------
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]]
Distance to lines in image 1 for LLS: 28.0256629375
Distance to lines in image 2 for LLS: 25.1628758
p'^T F p = 0.0302958533132
Fundamental Matrix from normalized 8-point algorithm:
[[  6.52113484e-07  -5.33615067e-06   8.80860210e-05]
 [ -4.91237449e-06  -3.40420428e-07  -6.43807393e-03]
 [ -8.56136054e-04   8.84208000e-03   1.45953063e-01]]
Distance to lines in image 1 for normalized: 0.890641627244
Distance to lines in image 2 for normalized: 0.828772911822
--------------------------------------------------------------------------------
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]]
Distance to lines in image 1 for LLS: 9.70143882945
Distance to lines in image 2 for LLS: 14.5682271905
p'^T F p = 0.0331366910623
Fundamental Matrix from normalized 8-point algorithm:
[[ -1.51007608e-07   2.51618737e-06  -1.56134009e-04]
 [  3.63462620e-06   3.22311660e-07   7.02588719e-03]
 [  2.36155133e-04  -8.53003408e-03  -2.45880925e-03]]
Distance to lines in image 1 for normalized: 0.889513454057
Distance to lines in image 2 for normalized: 0.89173437234
```

(a) For dataset 1,

$$F = \begin{bmatrix} 1.552 \times 10^{-6} & -8.182 \times 10^{-6} & -1.504 \times 10^{-3} \\ -5.870 \times 10^{-6} & -3.029 \times 10^{-7} & -1.136 \times 10^{-2} \\ -3.523 \times 10^{-3} & 1.415 \times 10^{-2} & 9.998 \times 10^{-1} \end{bmatrix}$$

For dataset 2,

$$F = \begin{bmatrix} -5.631 \times 10^{-6} & 2.750 \times 10^{-5} & -6.427 \times 10^{-3} \\ -2.776 \times 10^{-5} & -6.747 \times 10^{-6} & 1.522 \times 10^{-2} \\ 1.076 \times 10^{-2} & -1.225 \times 10^{-2} & -9.997 \times 10^{-1} \end{bmatrix}$$

(b) For dataset 1,

$$F_{norm} = \begin{bmatrix} 6.521 \times 10^{-7} & -5.336 \times 10^{-6} & 8.809 \times 10^{-5} \\ -4.912 \times 10^{-6} & -3.404 \times 10^{-7} & -6.438 \times 10^{-3} \\ -8.561 \times 10^{-4} & 8.842 \times 10^{-3} & 1.460 \times 10^{-1} \end{bmatrix}$$

For dataset 2,

$$F_{norm} = \begin{bmatrix} -1.510 \times 10^{-7} & 2.516 \times 10^{-6} & -1.561 \times 10^{-4} \\ 3.635 \times 10^{-6} & 3.223 \times 10^{-7} & 7.026 \times 10^{-3} \\ 2.361 \times 10^{-4} & -8.530 \times 10^{-3} & -2.459 \times 10^{-3} \end{bmatrix}$$

(c) See figures below:

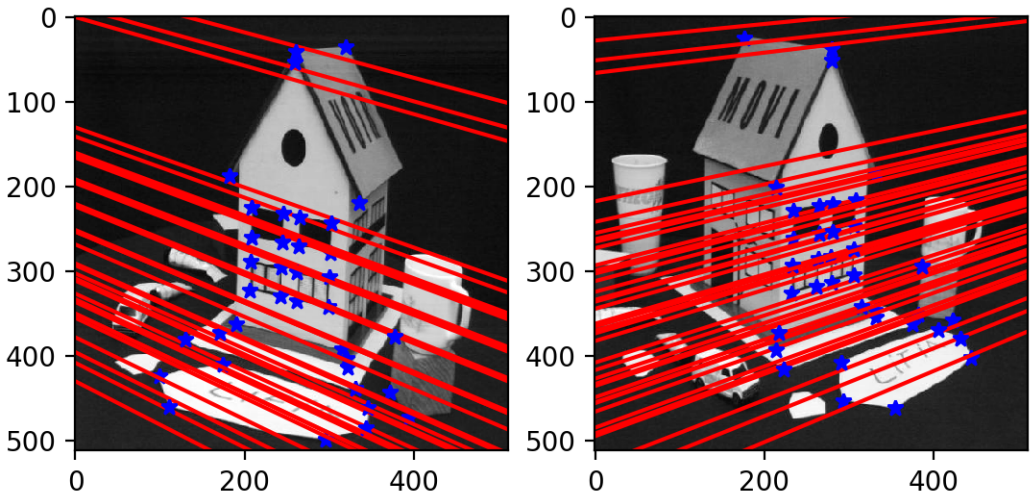


Figure 1: Plot of dataset 1 (without normalization)

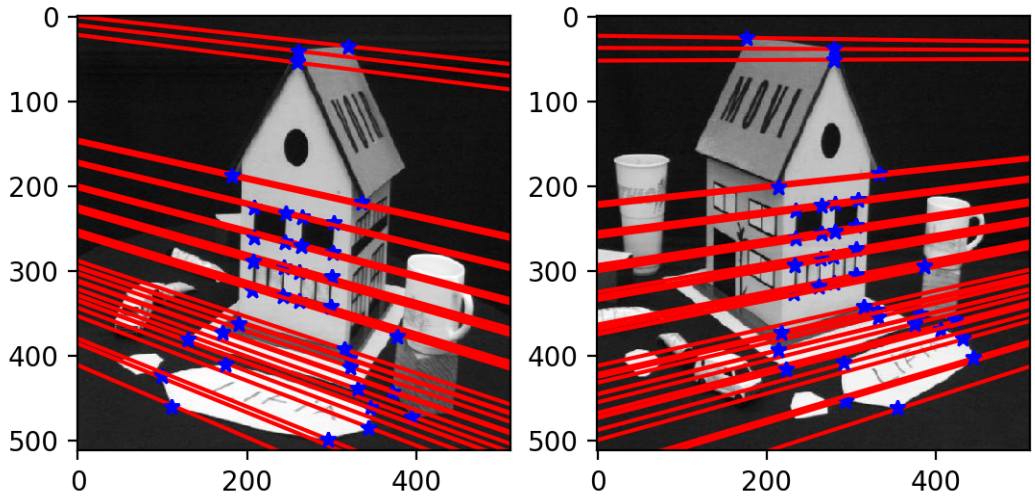


Figure 2: Plot of dataset 1 (normalized)

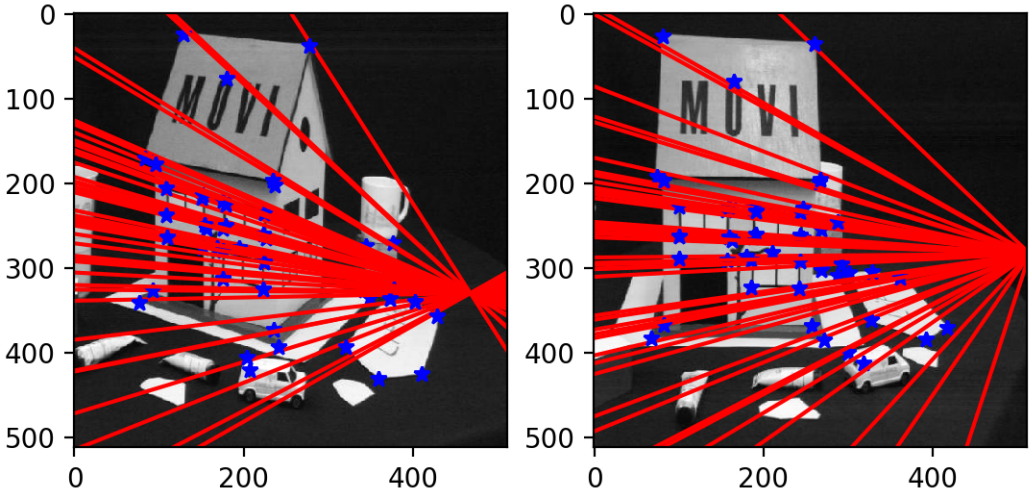


Figure 3: Plot of dataset 2 (without normalization)

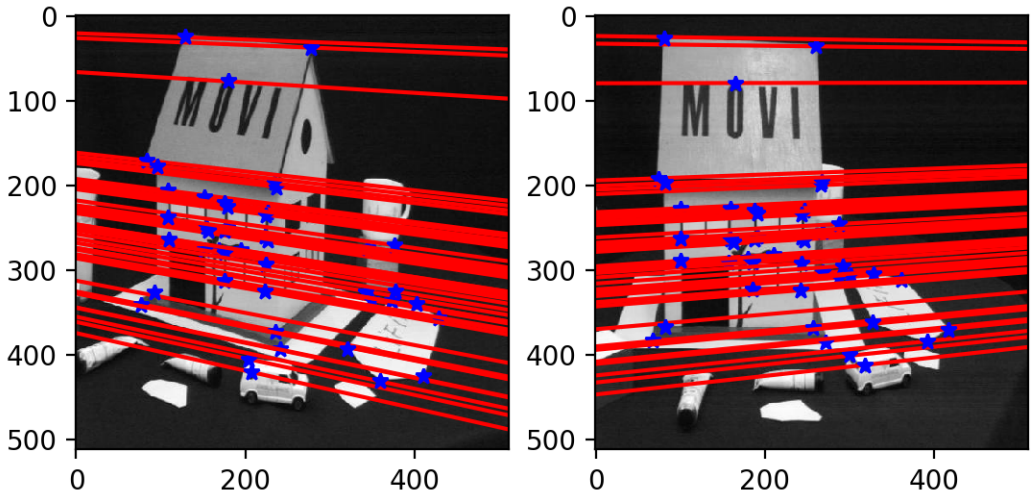


Figure 4: Plot of dataset 2 (normalized)

(d) Refer to `compute_distance_to_epipolar_lines()` and the output attached above.

2 Matching Homographies for Image Rectification

The following code includes implementations of `compute_epipole()` and `compute_matching_homographies()`:

```
'''
COMPUTE_EPIPOLE computes the epipole in homogenous coordinates
given matching points in two images and the fundamental matrix
Arguments:
    points1 - N points in the first image that match with points2
    points2 - N points in the second image that match with points1
    F - the Fundamental matrix such that (points1)^T * F * points2 = 0

    Both points1 and points2 are from the get_data_from_txt_file() method
Returns:
    epipole - the homogenous coordinates [x y 1] of the epipole in the image
'''
def compute_epipole(points1, points2, F):
    l = F.T.dot(points2.T).T
    U, s, VT = np.linalg.svd(l)
    e = VT[-1, :]
    # normalize
    e /= e[2]
    return e

'''
COMPUTE_MATCHING_HOMOGRAPHIES determines homographies H1 and H2 such that they
```

```
    rectify a pair of images
    Arguments:
        e2 - the second epipole
        F - the Fundamental matrix
        im2 - the second image
        points1 - N points in the first image that match with points2
        points2 - N points in the second image that match with points1
    Returns:
        H1 - the homography associated with the first image
        H2 - the homography associated with the second image
    '''
    def compute_matching_homographies(e2, F, im2, points1, points2):
        # calculate H2
        width = im2.shape[1]
        height = im2.shape[0]

        T = np.identity(3)
        T[0][2] = -1.0 * width / 2
        T[1][2] = -1.0 * height / 2

        e = T.dot(e2)
        e1_prime = e[0]
        e2_prime = e[1]
        if e1_prime >= 0:
            alpha = 1.0
        else:
            alpha = -1.0

        R = np.identity(3)
        R[0][0] = alpha * e1_prime / np.sqrt(e1_prime**2 + e2_prime**2)
        R[0][1] = alpha * e2_prime / np.sqrt(e1_prime**2 + e2_prime**2)
        R[1][0] = - alpha * e2_prime / np.sqrt(e1_prime**2 + e2_prime**2)
        R[1][1] = alpha * e1_prime / np.sqrt(e1_prime**2 + e2_prime**2)

        f = R.dot(e)[0]
        G = np.identity(3)
        G[2][0] = - 1.0 / f

        H2 = np.linalg.inv(T).dot(G.dot(R.dot(T)))

        # calculate H1
        e_prime = np.zeros((3, 3))
        e_prime[0][1] = -e2[2]
        e_prime[0][2] = e2[1]
        e_prime[1][0] = e2[2]
```

```
    e_prime[1][2] = -e2[0]
    e_prime[2][0] = -e2[1]
    e_prime[2][1] = e2[0]

    v = np.array([1, 1, 1])
    M = e_prime.dot(F) + np.outer(e2, v)

    points1_hat = H2.dot(M.dot(points1.T)).T
    points2_hat = H2.dot(points2.T).T

    W = points1_hat / points1_hat[:, 2].reshape(-1, 1)
    b = (points2_hat / points2_hat[:, 2].reshape(-1, 1))[:, 0]

    # least square problem
    a1, a2, a3 = np.linalg.lstsq(W, b)[0]
    HA = np.identity(3)
    HA[0] = np.array([a1, a2, a3])

    H1 = HA.dot(H2).dot(M)
    return H1, H2
```

OUTPUT

```
e1 [ -1.30017381e+03  -1.42178443e+02   1.00000000e+00]
e2 [  1.65402039e+03   4.53130376e+01   1.00000000e+00]
H1:
[[ -1.19980175e+01  -4.15391888e+00  -1.23134027e+02]
 [  1.40770937e+00  -1.48663913e+01  -2.83413506e+02]
 [ -9.21951815e-03  -2.19162343e-03  -1.22985776e+01]]

H2:
[[  8.09784464e-01  -1.22037583e-01   7.99367985e+01]
 [ -3.00285336e-02   1.01581747e+00   3.63803136e+00]
 [ -6.99412356e-04   1.05404089e-04   1.15206612e+00]]
```

(a) The epipoles are

$$e_1 = \begin{pmatrix} -1300.174 \\ -142.178 \\ 1 \end{pmatrix}, \; e_2 = \begin{pmatrix} 1654.020 \\ 45.313 \\ 1 \end{pmatrix}$$

(b) The homographies are

$$H_1 = \begin{bmatrix} -11.998 & -4.154 & -123.134 \\ 1.408 & -14.866 & -283.414 \\ -9.220 \times 10^{-3} & -2.192 \times 10^{-3} & -12.229 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 0.810 & -0.122 & 79.937 \\ -0.030 & 1.016 & 3.638 \\ -6.994 \times 10^{-4} & 1.054 \times 10^{-4} & 1.152 \end{bmatrix}$$

(c) The rectified images are shown below:

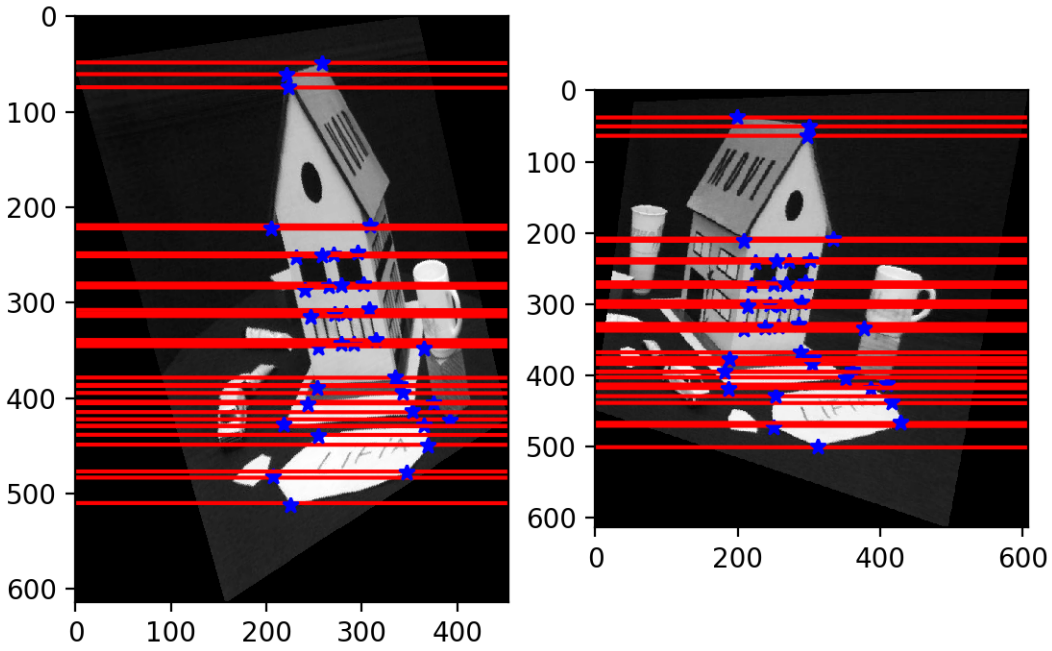


Figure 5: The rectified images

3 The Factorization Method

The following code includes the implementation of `factorization_method()`:

```
'''
FACTORIZATION_METHOD The Tomasi and Kanade Factorization Method to determine
the 3D structure of the scene and the motion of the cameras.
Arguments:
    points_im1 - N points in the first image that match with points_im2
    points_im2 - N points in the second image that match with points_im1

    Both points_im1 and points_im2 are from the get_data_from_txt_file() method
Returns:
    structure - the structure matrix
    motion - the motion matrix
'''
def factorization_method(points_im1, points_im2):
    N = points_im1.shape[0]
    points_set = [points_im1, points_im2]
    D = np.zeros((4, N))

    for i in range(len(points_set)):
```

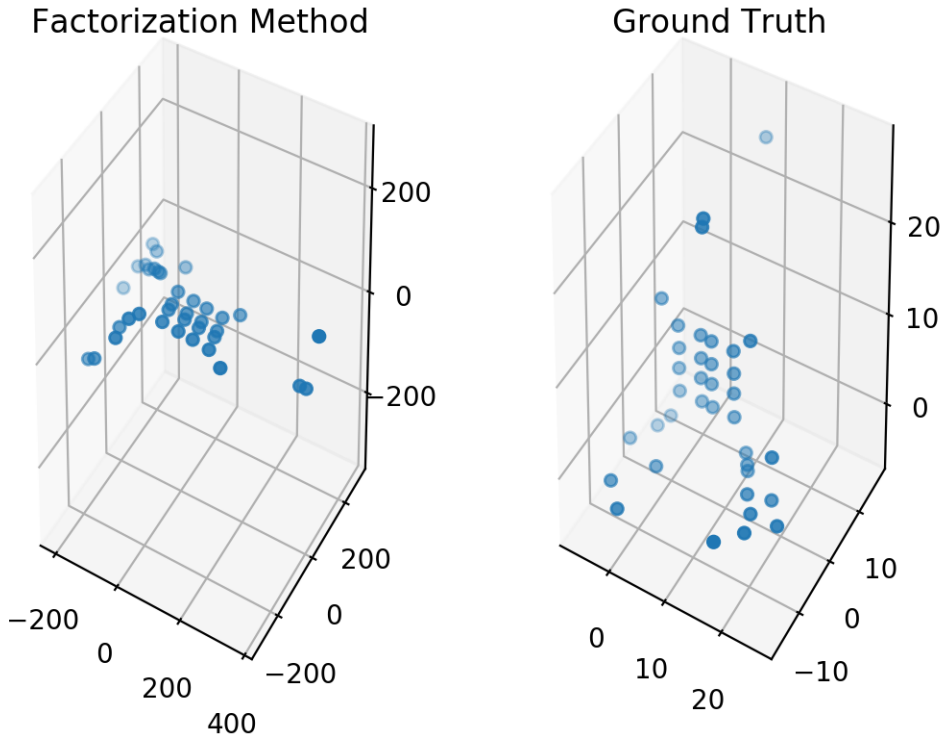
```
        # normalize points
        points = points_set[i]
        centroid = 1.0 / N * points.sum(axis=0)
        points[:, 0] -= centroid[0] * np.ones(N)
        points[:, 1] -= centroid[1] * np.ones(N)
        # construct D
        D[2*i:2*i+2, :] = points[:, 0:2].T

    U, s, VT = np.linalg.svd(D)
    print U.shape, s.shape, VT.shape
    print s        # print sigular values
    M = U[:, 0:3] # motion
    S = np.diag(s)[0:3, 0:3].dot(VT[0:3, :]) # structure
    return S, M
```

OUTPUT

```
s1 = [ 959.5852216   540.47613178  184.43174791   27.9151956 ]
s2 = [ 264.54396508  210.06072009    7.21921783    5.12857709]
```

(a) Refer to code above.

(b) The resulting 3D points look identical to the provided ground truth.



Define a common matrix H for scaling and rotation, and we know

$$D = MS$$

$$D = (MH^{-1})(HS) = M'S'$$

which means new structure S' can be obtained by scaling and rotating S and their 3D structures will be identical. But we should notice choice there are many choices of S, so the problem is underdetermined. This is the similarity ambiguity problem, which occurs when a reconstruction is correct up to a similarity transform (rotation, translation and scaling).

(c) The four singular values are:

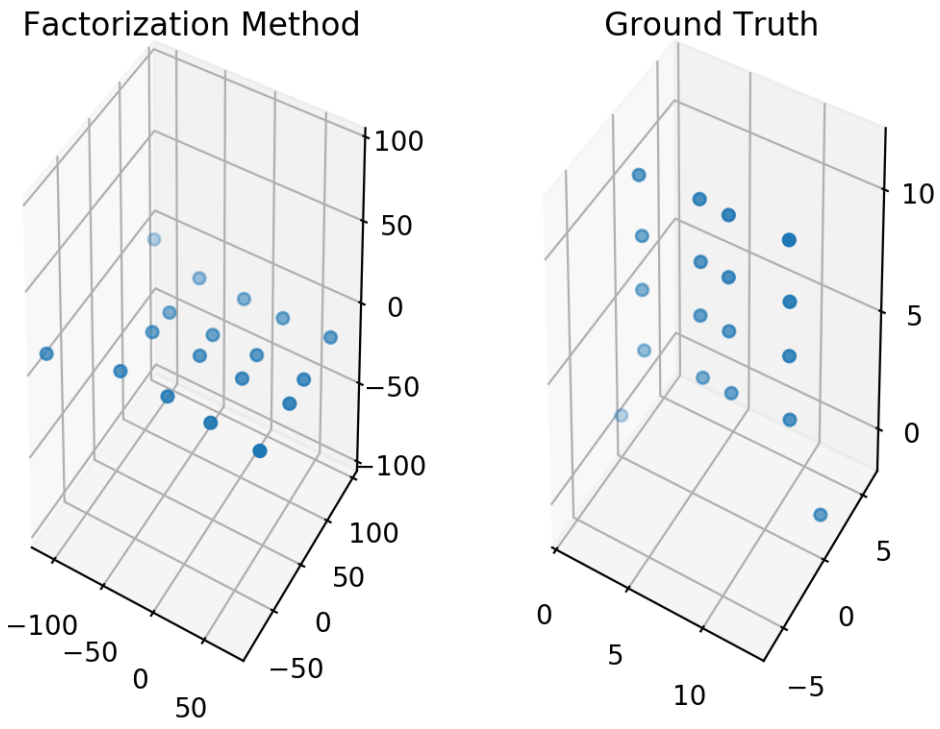
$$959.585 \quad 540.476 \quad 184.432 \quad 27.915$$

Ideally, we expect three non-zero sigular values because $rank(D) = 3$. Even in this case, four non-zero singular values are found by SVD, we should notice the last singular value is much smaller than the other three. The measurement noise and affine camera approximation results in the fourth non-zero value (which should be zero). But we can simply ignore the fourth singular value and estimate

$$D = U_3 \Sigma_3 V_3^T$$

and this will still the best approximation of D.

(d) By loading only a subset of the original correspondences, the new results are



(e) The four singular values are:

$$264.544 \quad 210.061 \quad 7.219 \quad 5.129$$

Similarly, we get four non-zero singular values but some of them are much smaller than the others. Concretely in this case, the third and fourth value should result from measurement noise and affine approximation. The reason why only two non-zero singular values are expected is $rank(D) = 2$, which means points of the subset are located on the same plane and it is impossible to implement 3D reconstruction via this subset of correspondences.

4 Triangulation in Structure From Motion

The following code includes implementations of `estimate_initial_RT()`, `linear_estimate_3d_point()`, `reprojection_error()`, `jacobian()`, `nonlinear_estimate_3d_point()` and `estimate_RT_from_E()`:

```
'''
ESTIMATE_INITIAL_RT from the Essential Matrix, we can compute 4 initial
guesses of the relative RT between the two cameras
Arguments:
    E - the Essential Matrix between the two cameras
Returns:
    RT: A 4x3x4 tensor in which the 3x4 matrix RT[i,:,:] is one of the
        four possible transformations
'''
def estimate_initial_RT(E):
    U, s, VT = np.linalg.svd(E)
    # compute R
    Z = np.array([[0, 1, 0],
                  [-1, 0, 0],
                  [0, 0, 0]])
    W = np.array([[0, -1, 0],
                  [1, 0, 0],
                  [0, 0, 1]])

    M = U.dot(Z).dot(U.T)
    Q1 = U.dot(W).dot(VT)
    R1 = np.linalg.det(Q1) * 1.0 * Q1

    Q2 = U.dot(W.T).dot(VT)
    R2 = np.linalg.det(Q2) * 1.0 * Q2

    # compute T
    T1 = U[:, 2].reshape(-1, 1)
    T2 = -U[:, 2].reshape(-1, 1)

    R_set = [R1, R2]
```

```
    T_set = [T1, T2]
    RT_set = []
    for i in range(len(R_set)):
        for j in range(len(T_set)):
            RT_set.append(np.hstack((R_set[i], T_set[j])))

    RT = np.zeros((4, 3, 4))
    for i in range(RT.shape[0]):
        RT[i, :, :] = RT_set[i]

    return RT

'''
LINEAR_ESTIMATE_3D_POINT given a corresponding points in different images,
compute the 3D point is the best linear estimate
Arguments:
    image_points - the measured points in each of the M images (Mx2 matrix)
    camera_matrices - the camera projective matrices (Mx3x4 tensor)
Returns:
    point_3d - the 3D point
'''
def linear_estimate_3d_point(image_points, camera_matrices):
    N = image_points.shape[0]
    A = np.zeros((2*N, 4))
    A_set = []

    for i in xrange(N):
        pi = image_points[i]
        Mi = camera_matrices[i]
        Aix = pi[0]*Mi[2] - Mi[0]
        Aiy = pi[1]*Mi[2] - Mi[1]
        A_set.append(Aix)
        A_set.append(Aiy)

    for i in xrange(A.shape[0]):
        A[i] = A_set[i]

    U, s, VT = np.linalg.svd(A)
    P_homo = VT[-1]
    P_homo /= P_homo[-1]
    P = P_homo[:3]

    return P

'''
```

```
REPROJECTION_ERROR given a 3D point and its corresponding points in the image
planes, compute the reprojection error vector and associated Jacobian
Arguments:
    point_3d - the 3D point corresponding to points in the image
    image_points - the measured points in each of the M images (Mx2 matrix)
    camera_matrices - the camera projective matrices (Mx3x4 tensor)
Returns:
    error - the 2Mx1 reprojection error vector
'''
def reprojection_error(point_3d, image_points, camera_matrices):
    N = image_points.shape[0]
    error_set = []
    point_3d_homo = np.hstack((point_3d, 1))

    for i in xrange(N):
        pi = image_points[i]
        Mi = camera_matrices[i]
        Yi = Mi.dot(point_3d_homo)
        # compute error
        pi_prime = 1.0 / Yi[2] * np.array([Yi[0], Yi[1]])
        error_i = (pi_prime - pi)
        error_set.append(error_i[0])
        error_set.append(error_i[1])

    error = np.array(error_set)
    return error

'''
JACOBIAN given a 3D point and its corresponding points in the image
planes, compute the reprojection error vector and associated Jacobian
Arguments:
    point_3d - the 3D point corresponding to points in the image
    camera_matrices - the camera projective matrices (Mx3x4 tensor)
Returns:
    jacobian - the 2Mx3 Jacobian matrix
'''
def jacobian(point_3d, camera_matrices):
    J = np.zeros((2*camera_matrices.shape[0], 3))
    point_3d_homo = np.hstack((point_3d, 1))
    J_set = []

    for i in xrange(camera_matrices.shape[0]):
        Mi = camera_matrices[i]
        pi = Mi.dot(point_3d_homo)
        Jix = (pi[2]*np.array([Mi[0, 0], Mi[0, 1], Mi[0, 2]]) \
```

```
                - pi[0]*np.array([Mi[2, 0], Mi[2, 1], Mi[2, 2]])) / pi[2]**2
        Jiy = (pi[2]*np.array([Mi[1, 0], Mi[1, 1], Mi[1, 2]]) \
                - pi[1]*np.array([Mi[2, 0], Mi[2, 1], Mi[2, 2]])) / pi[2]**2
        J_set.append(Jix)
        J_set.append(Jiy)

    for i in range(J.shape[0]):
        J[i] = J_set[i]
    return J

'''
NONLINEAR_ESTIMATE_3D_POINT given a corresponding points in different images,
compute the 3D point that iteratively updates the points
Arguments:
    image_points - the measured points in each of the M images (Mx2 matrix)
    camera_matrices - the camera projective matrices (Mx3x4 tensor)
Returns:
    point_3d - the 3D point
'''
def nonlinear_estimate_3d_point(image_points, camera_matrices):
    P = linear_estimate_3d_point(image_points, camera_matrices)

    for i in xrange(10):
        e = reprojection_error(P, image_points, camera_matrices)
        J = jacobian(P, camera_matrices)
        P -= np.linalg.inv(J.T.dot(J)).dot(J.T).dot(e)

    return P

'''
ESTIMATE_RT_FROM_E from the Essential Matrix, we can compute  the relative RT
between the two cameras
Arguments:
    E - the Essential Matrix between the two cameras
    image_points - N measured points in each of the M images (NxMx2 matrix)
    K - the intrinsic camera matrix
Returns:
    RT: The 3x4 matrix which gives the rotation and translation between the
        two cameras
'''
def estimate_RT_from_E(E, image_points, K):
    RT = estimate_initial_RT(E)
    count = np.zeros((1, 4))
    I0 = np.array([[1.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0],
```

```
                    [0.0, 0.0, 1.0, 0.0]])
    M1 = K.dot(I0)

    camera_matrices = np.zeros((2, 3, 4))
    camera_matrices[0] = M1
    for i in range(RT.shape[0]):
        RTi = RT[i] # 3x4 matrix
        M2i = K.dot(RTi)
        camera_matrices[1] = M2i
        for j in range(image_points.shape[0]):
            pointj_3d = nonlinear_estimate_3d_point(image_points[j], camera_matrices)
            Pj = np.vstack((pointj_3d.reshape(3, 1), [1]))
            Pj_prime = camera1tocamera2(Pj, RTi)
            if Pj[2] > 0 and Pj_prime[2] > 0:
                count[0, i] += 1

    maxIndex = np.argmax(count)
    maxRT = RT[maxIndex]
    return maxRT

def camera1tocamera2(P, RT):
    P_homo = np.array([P[0], P[1], P[2], 1.0])
    A = np.zeros((4, 4))
    A[0:3, :] = RT
    A[3, :] = np.array([0.0, 0.0, 0.0, 1.0])
    P_prime_homo = A.dot(P_homo.T)
    P_prime_homo /= P_prime_homo[3]
    P_prime = P_prime_homo[0:3]
    return P_prime
```

OUTPUT

```
--------------------------------------------------------------------------------
Part A: Check your matrices 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:
[[[ 0.98305251 -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]]]
--------------------------------------------------------------------------------
Part B: Check that the difference from expected point
is near zero
--------------------------------------------------------------------------------
Difference:  0.00292430530368
--------------------------------------------------------------------------------
Part C: Check that the difference from expected error/Jacobian
is near zero
--------------------------------------------------------------------------------
Error Difference:  8.30130027278e-07
Jacobian Difference:  1.81711463654e-08
--------------------------------------------------------------------------------
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
--------------------------------------------------------------------------------
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]]
--------------------------------------------------------------------------------
Part F: Run the entire SFM pipeline
--------------------------------------------------------------------------------
```

(a) Refer to `Estimated RT` in **Part A** in the output above. For each matrix,

$$RT = \begin{bmatrix} R & T \end{bmatrix}$$

and you will find the third matrix matches the `Example RT`.

(b) Refer to **Part B** in the output above.

(c) Refer to **Part C** in the output above.

(d) Refer to **Part D** in the output above.

(e) Refer to **Part E** in the output above.

(f) The final plot of the reconstructed statue is shown below:

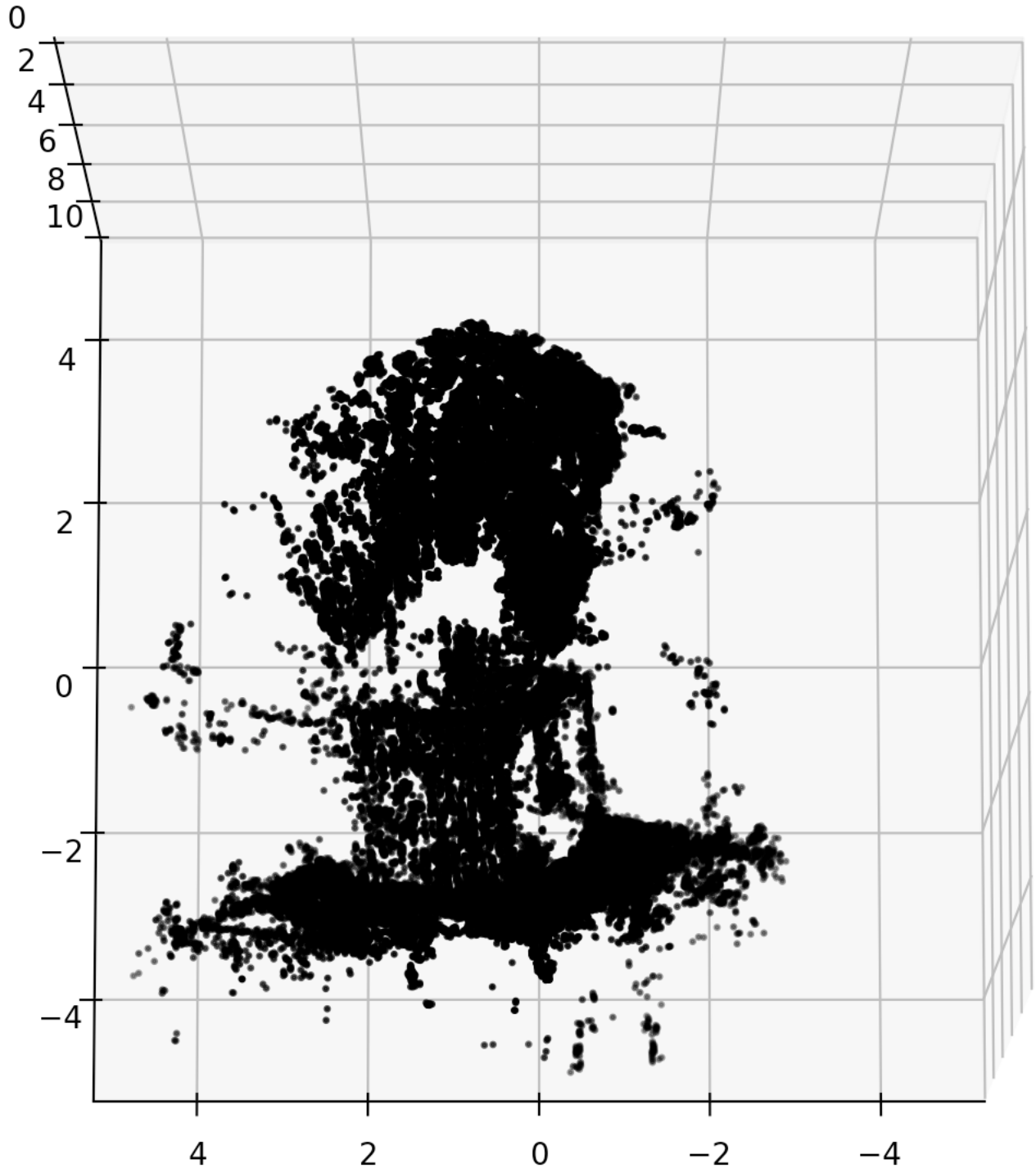


Figure 6: The reconstructed statue