CS231A: Computer Vision, From 3D Reconstruction to Recognition
Homework #2 Solution

(Winter 2021)

Shubham Shrivastava email: shubhams@stanford.edu

1 Fundamental Matrix (20 points)

In this question, you will explore some properties of fundamental matrix and derive a minimal parameterization for it.

a Show that two 3×4 camera matrices M and M' can always be reduced to the following canonical forms by an appropriate projective transformation in 3D space, which is represented by a 4×4 matrix H. Here, we assume $e_3^T(-A'A^{-1}b+b') \neq 0$, where $e_3 = (0,0,1)^T$, $M = [A, b]$ and $M' = [A', b']$.
Note: You don't have to show the explicit form of H for the proof. **[10 points]**
Hint: The camera matrix has rank 3. Block matrix multiplication may be helpful. If you construct a projective transformation matrix H_0 that reduces M to $\hat{M}$, (i.e., $\hat{M} = MH_0$) can a H_1 be constructed such that not only does it not affect the reduction of M to $\hat{M}$ (i.e., $\hat{M} = MH_0H_1$), but it also reduces M' to $\hat{M}'$? (i.e., $\hat{M}' = M'H_0H_1$)

$$\hat{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \text{ and } \hat{M}' = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

a) Camera Matrices: $M = [A \; b]$

$M' = [A' \; b']$

Canonical form of camera matrix: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \hat{M}$

We have to prove that an "H_0" exists such that

$$MH_0 = \hat{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and, another "H_1" exists such that

$$MH_0H_1 = \hat{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \& \quad M'H_0H_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$MH_0 = [A \; b]H_0$

Let's construct H_0 as $\begin{bmatrix} A_0 & b_0 \\ 0 & 1 \end{bmatrix}$

$$MH_0 = [A \; b]\begin{bmatrix} A_0 & b_0 \\ 0 & 1 \end{bmatrix}$$

$$= [AA_0 \quad Ab_0 + b] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

AA_0 $\quad$ $Ab_0 + b$

$\Rightarrow Ab_0 + b = 0_{3\times 1} \Rightarrow \boxed{b_0 = -A^{-1}b}$

and, $AA_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I_{3\times 1}$

$\Rightarrow AA_0 = I$

$\Rightarrow \boxed{A_0 = \bar{A}^{-1}}$

This results in $\boxed{H_0 = \begin{bmatrix} \bar{A}^{-1} & -\bar{A}^{-1}b \\ 0 & 1 \end{bmatrix}}$

Also, given is that, $\hat{M} = MH_0H_1 = \underbrace{MH_0}_{3\times 4}$

$\Rightarrow [I_{3\times 3} \;\; 0]H_1 = [I_{3\times 3} \;\; 0]$

$\Rightarrow H_1 = \begin{bmatrix} I_{3\times 3} & 0 \\ m_{1\times 3} & n_{1\times 1} \end{bmatrix}$ $\Bigg| \; H_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ m_1 & m_2 & m_3 & n_1 \end{bmatrix}$

$M'H_0H_1 = \underbrace{[A' \;\; b']}_{3\times 4} \underbrace{\begin{bmatrix} \bar{A}^{-1} & -\bar{A}^{-1}b \\ 0 & 1 \end{bmatrix}}_{4\times 4} H_1$

$\Rightarrow M'H_0H_1 = \begin{bmatrix} \underbrace{A'\bar{A}^{-1}}_{3\times 3} & \underbrace{-A'\bar{A}^{-1}b + b'}_{3\times 1} \end{bmatrix} \begin{bmatrix} I_{3\times 3} & 0_{3\times 1} \\ m_{1\times 3} & n_{1\times 1} \end{bmatrix}$

Given constraint $e_3 = (0, 0, 1)^T$; where, $e_3^T = (-A'\bar{A}^{-1}b + b')$

Let's represent $M'H_0$ as

$$M'H_0 = \begin{bmatrix} A'A^{-1} & -A'A^{-1}b+b' \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & 0 \\ p_{21} & p_{22} & p_{23} & 0 \\ p_{31} & p_{32} & p_{33} & 1 \end{bmatrix}$$

($A'A^{-1}$)

$$M'H_0H_1 = \begin{bmatrix} p_{11} & p_{12} & p_{13} & 0 \\ p_{21} & p_{22} & p_{23} & 0 \\ p_{31} & p_{32} & p_{33} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ m_1 & m_2 & m_3 & n_1 \end{bmatrix}$$

$$\Rightarrow M'H_0H_1 = \begin{bmatrix} p_{11} & p_{12} & p_{13} & 0 \\ p_{21} & p_{22} & p_{23} & 0 \\ p_{31}+m_1 & p_{32}+m_2 & p_{33}+m_3 & n_1 \end{bmatrix}$$

If $m_1 = -p_{31}$, $m_2 = -p_{32}$, $m_3 = -p_{33}$, and $n_1 = 1$

Then we arrive at:

$$M'H_0H_1 = \begin{bmatrix} p_{11} & p_{12} & p_{13} & 0 \\ p_{21} & p_{22} & p_{23} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{for} \quad H_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_{31} & -p_{32} & -p_{33} & 1 \end{bmatrix}$$

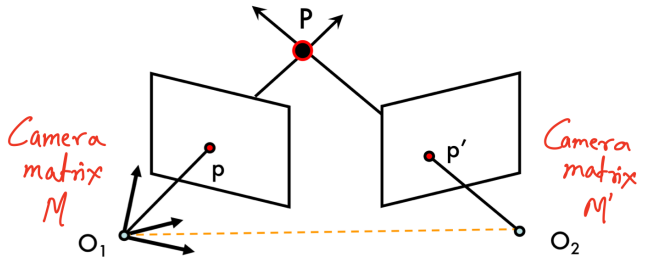
This is of the canonical form

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & 1 \end{bmatrix} \quad \text{with} \quad b_1 = b_2 = 0$$

The projective transform H_1 corresponding to the canonical matrix shown above is:

$$H_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a_{31} & -a_{32} & -a_{33} & 1 \end{bmatrix}$$

b Given a 4×4 matrix H representing a projective transformation in 3D space, prove that the fundamental matrices corresponding to the two pairs of camera matrices (M, M') and $(MH, M'H)$ are the same. **[5 points]**
(Hint: Think about point correspondences)



$$p^T F p' = 0 \qquad F = K^{-T} \cdot [T_\times] \cdot R \, K'^{-1}$$

F = Fundamental Matrix
(Faugeras and Luong, 1992)

Points in Camera 1 and Camera 2 are related by the fundamental matrix F as

$$p^T F p' = 0$$
$$\Rightarrow (Mp)^T F (M'p) = 0$$
$$\Rightarrow p^T M^T F M' p = 0 \quad \text{(i)}$$

If a projective transformation in 3D, H is applied to the points, then they are related by the fundamental matrix as

$$(MHP)^T F (M'HP) = 0$$

(MHP, $M'HP$: 3D points)

We can reformulate the camera matrices as: $M_1 \rightarrow MH^{-1}$, $M_1' \rightarrow M'H^{-1}$

This gives us:

$$(M_1 HP)^T F (M_1' HP) = 0$$

$$(M H^{-1} HP)^T F (M' H^{-1} HP) = 0$$

$$\Rightarrow (MP)^T F M' P = 0$$

$$\Rightarrow P^T M^T F M' P = 0 \quad \text{— (ii)}$$

From equations (i) and (ii), it can be seen that the fundamental matrix corresponding to the two pairs of camera matrices are the same.

c Using the conclusions from (a) and (b), derive the fundamental matrix F of the camera pair (M, M') using a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, b_1, b_2. Then use the fact that F is only defined up to a scale factor to construct a seven-parameter expression for F. *(Hint: The fundamental matrix corresponding to a pair of camera matrices $M = [I \mid 0]$ and $M' = [A \mid b]$ is equal to $[b]_\times A$.)* **[5 points]**

Given two camera matrices M_1 and M_2, we have seen from (a), that a trasformation matrix H_0 exists which will transform M_1 to

$$\hat{M} = M H_0 = [I \;\; 0]$$

Another matrix H_1 exists which will not effect $\hat{M}$ but will reduce M' to:

$$\hat{M}' = M' H_0 H_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [A \;\; b]$$

$$\hat{M} = M H_0 = M H_0 H_1$$

Let's define another transformation matrix H such that: $H^{-1} = H_0 H_1$

$$\Rightarrow \quad \hat{M} = M H^{-1} = [I \;\; 0]$$

and, $\hat{M}' = M' H^{-1} = [A \;\; b]$

Applying this transformation on 3D point X, we get

$$\hat{X} = HX$$

So, the 2D projections in each camera can be given as $x = MX = M H^{-1} H X = \hat{M}\hat{X} = [I \;\; 0]\hat{X} = \hat{X}$ — (i)

$$x' = M'X = M'H^{-1}HX = \hat{M}'\hat{X} = [A \;\; b]\hat{X} \quad \text{— (ii)}$$

In homogeneous coordinates $\hat{X} = [\hat{X}_1 \;\; \hat{X}_2 \;\; \hat{X}_3 \;\; 1]^T$

So, we can rewrite (ii) as

$$x' = [A \;\; b]\hat{X} = [A \;\; b]\begin{bmatrix}\hat{X}_1\\ \hat{X}_2\\ \hat{X}_3\\ 1\end{bmatrix} = A[I \;\; 0]\begin{bmatrix}\hat{X}_1\\ \hat{X}_2\\ \hat{X}_3\\ 1\end{bmatrix} + b$$

$$\Rightarrow x' = A[I \;\; 0]\hat{X} + b = Ax + b \quad \text{using equation (i)} \quad \text{— (iii)}$$

Let's formulate the following equations:

$$x' \times b = (Ax + b) \times b = (Ax \times b) + \underbrace{(b \times b)}_{=0}$$

$$= Ax \times b \quad \text{— (iv)}$$

$$x'^T \cdot (x' \times b) = x'^T \cdot (Ax \times b) = 0 \quad \text{— (v)}$$

$(x' \times b)$: Perpendicular to the plane containing x'

This also results in: $x'^T \cdot (b \times Ax) = 0$ — (vi)

A cross product can be represented in the dot product form as follows:

$a \times b = [a_x] \cdot b$ where $[a_x] = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$

So, equation (vi) can be represented as

$$x'^T \cdot (b \times Ax) = x'^T [b_x] A x = 0 \quad \text{— (vii)}$$

This is of the form: $x'^T F x = 0$

where, F = fundamental matrix

x = 2D points in Camera 1

x' = Corresponding 2D points in Camera 2

Hence, we obtain an expression for the fundamental matrix, $\boxed{F = [b_x] A}$

Expanding this fundamental matrix:

$F = [b_x] A$; we have $\vec{b} = [b_1 \; b_2 \; 1]$

and $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 0 \end{bmatrix}$

$$\Rightarrow F = \begin{bmatrix} 0 & -1 & b_2 \\ 1 & 0 & -b_1 \\ b_2 & b_1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 0 \end{bmatrix}$$

$$\Rightarrow F = \begin{bmatrix} -a_{21} & -a_{22} & -a_{23} \\ a_{11} & a_{12} & a_{13} \\ a_{11}b_2 + a_{21}b_1 & a_{12}b_2 + a_{22}b_1 & a_{13}b_2 + a_{23}b_1 \end{bmatrix}$$

Fundamental matrix is only known up to scale, hence the last element of F (i.e. row=3, column=3) must be equal to 1. Using this constraint, we obtain:

$$F = \frac{1}{a_{13}b_2 + a_{23}b_1} \begin{bmatrix} -a_{21} & -a_{22} & -a_{23} \\ a_{11} & a_{12} & a_{13} \\ a_{11}b_2 + a_{21}b_1 & a_{12}b_2 + a_{22}b_1 & a_{13}b_2 + a_{23}b_1 \end{bmatrix}$$

2 Fundamental Matrix Estimation From Point Correspondences (30 points)

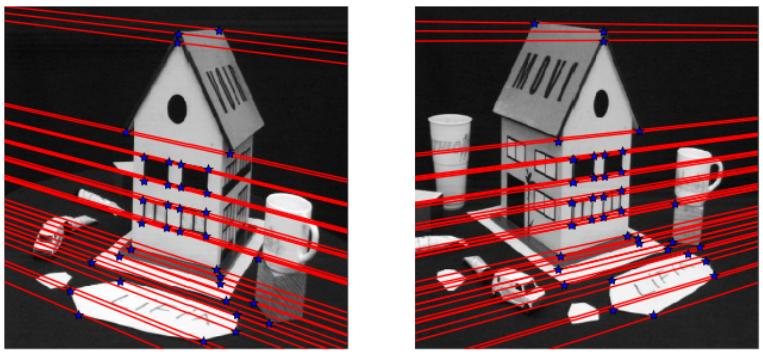


Figure 1: Example illustration, with epipolar lines shown in both images (Images courtesy Forsyth & Ponce)

This problem is concerned with the estimation of the fundamental matrix from point correspon-

dences. In this problem, you will implement both the linear least-squares version of the eight-point algorithm and its normalized version to estimate the fundamental matrices. You will implement the methods in `p2.py` and complete the following:

(a) Implement the linear least-squares eight point algorithm in `lls_eight_point_alg()` and report the returned fundamental matrix. Remember to enforce the rank-two constraint for the fundamental matrix via singular value decomposition. Attach a copy of your code. **[20 points]**

```
'''
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):
    # points1 and points2 are in homogeneous coordinate system.
    # let's build the set of linear equations [Wf = 0]
    # where W is composed of W = [uu', uv', u, vu', vv', v, u', v', 1]
    # For a set of points p and p', fundamental matrix relates
    # them as [pT.F.p' = 0]
    # Here, p = points2; p' = points1
    W = []
    for i, p in enumerate(points2):
        p_prime = points1[i]
        W.append([p[0]*p_prime[0],p[0]*p_prime[1],p[0],p[1]*p_prime[0],p[1]*
    p_prime[1],p[1],p_prime[0],p_prime[1],p_prime[2]])
    W = np.array(W, dtype=np.float64)
    u, s, v_t = np.linalg.svd(W, full_matrices=True)
    # fundamental matrix can be obtained as the last column of v or last row
    of v_transpose
    F_hat = v_t.T[:,-1]
    F_hat = np.reshape(F_hat, (3,3))
    # this fundamental matrix may be full rank i.e rank=3. but the rank of
    fundamental matrix should be rank(f)=2
    # let's use SVD on F_hat again and then obtain a rank2 fundamental matrix
    u, s, v_t = np.linalg.svd(F_hat, full_matrices=True)
    # let's build a matrix from the first two singular values
    s_mat = np.diag(s)
    s_mat[-1,-1] = 0.
    # let's compose our rank(2) fundamental matrix
    F = np.dot(u, np.dot(s_mat, v_t))
    # return the fundamental matrix

    return F
```

Listing 1: Code Snippet

```
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) Implement the normalized eight point algorithm in `normalized_eight_point_alg()` and report the returned fundamental matrix. Remember to enforce the rank-two constraint for the fundamental matrix via singular value decomposition. Attach a copy of your code. **[5 points]**

```
'''
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
'''
def normalized_eight_point_alg(points1, points2):
    # compute the centroid of all points in camera1 and camera2
    centroid1 = np.array([np.mean(points1[:,0]), np.mean(points1[:,1]), np.mean(points1[:,2])])
    centroid2 = np.array([np.mean(points2[:,0]), np.mean(points2[:,1]), np.mean(points2[:,2])])
    # let's compute mean-squared distance between each point
    # and centroid of all points in respective cameras
    # camera 1
    squared_dist1 = []
    for pt in points1:
        squared_dist1.append((pt[0] - centroid1[0]) ** 2 + \
                             (pt[1] - centroid1[1]) ** 2 + \
                             (pt[2] - centroid1[2]) ** 2)
    mean_squared_dist1 = np.sqrt(np.mean(squared_dist1))

    # camera 2
    squared_dist2 = []
    for pt in points2:
        squared_dist2.append((pt[0] - centroid2[0]) ** 2 + \
                             (pt[1] - centroid2[1]) ** 2 + \
                             (pt[2] - centroid2[2]) ** 2)
    mean_squared_dist2 = np.sqrt(np.mean(squared_dist2))

    # build a transformation matrix which will first translate these
    # points to the centroid and then scale them so that the
    # points are centered at the centroid with a mean-squared distance
    # of 2 pixels
    translation1 = np.array([[1., 0., -centroid1[0]],
                             [0., 1., -centroid1[1]],
                             [0., 0., 1.]], dtype=np.float64)
    scaling1 = np.array([[np.sqrt(2.)/mean_squared_dist1, 0., 0.],
                         [0., np.sqrt(2.)/mean_squared_dist1, 0.],
                         [0., 0., 1.0]], dtype=np.float64)
    T1 = np.dot(scaling1, translation1)
    normalized_points1 = np.dot(T1, points1.T).T

    # normalize points in second camera
    translation2 = np.array([[1., 0., -centroid2[0]],
                             [0., 1., -centroid2[1]],
                             [0., 0., 1.]], dtype=np.float64)
    scaling2 = np.array([[np.sqrt(2.)/mean_squared_dist2, 0., 0.],
                         [0., np.sqrt(2.)/mean_squared_dist2, 0.],
                         [0., 0., 1.0]], dtype=np.float64)
    T2 = np.dot(scaling2, translation2)
```

```
    normalized_points2 = np.dot(T2, points2.T).T

    # compute fundamental matrix for normalized points
    F_normalized = lls_eight_point_alg(normalized_points1, normalized_points2)

    # denormalize fundamenta matrix
    F = np.dot(T2.T, np.dot(F_normalized, T1))

    # return fundamental matrix
    return F
```

Listing 2: Code Snippet

```
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]]
```

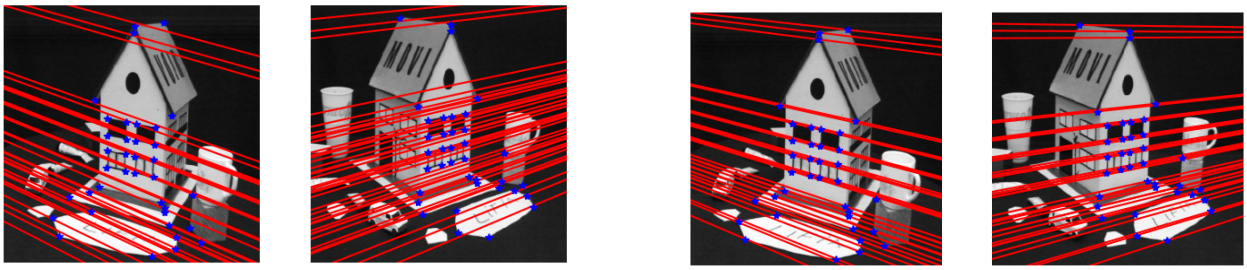
(c) After implementing methods to determine the Fundamental matrix, we can now determine epipolar lines. Specifically to determine the accuracy of our Fundamental matrix, we will compute the average distance between a point and its corresponding epipolar line in `compute_distance_to_epipolar_lines()`. Attach a copy of your code.
[5 points]

```
'''
COMPUTE_DISTANCE_BETWEEN_POINTS_AND_EPIPOLAR_LINES
computes distance between points in each camera to the epipolar
lines computed from the other camera using fundamental matrix.
Arguments:
    points - N points in the image
    lines - lines computed from the other camera using fundamental matrix.

Returns:
    distance for each point and corresponding line
'''
def point2line_dist(points, lines):
    distances = []
    for i, pt in enumerate(points):
        dist = abs(lines[i][0]*pt[0] + lines[i][1]*pt[1] + lines[i][2]) / np.
    sqrt((lines[i][0] ** 2) + (lines[i][1] ** 2))
        distances.append(dist)
    distances = np.array(distances, dtype=np.float64)
    return distances

'''
COMPUTE_DISTANCE_TO_EPIPOLAR_LINES  computes the average distance of a set a
points to their corresponding epipolar lines. Compute just the average
    distance
from points1 to their corresponding epipolar lines (which you get from points2
    ).
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):
    # compute epipolar line
```

```
    epipolar_lines1 = np.dot(F.T, points2.T).T
    distances1 = point2line_dist(points1, epipolar_lines1)
    average_distance = np.mean(distances1)
    # return average distance
    return average_distance
```

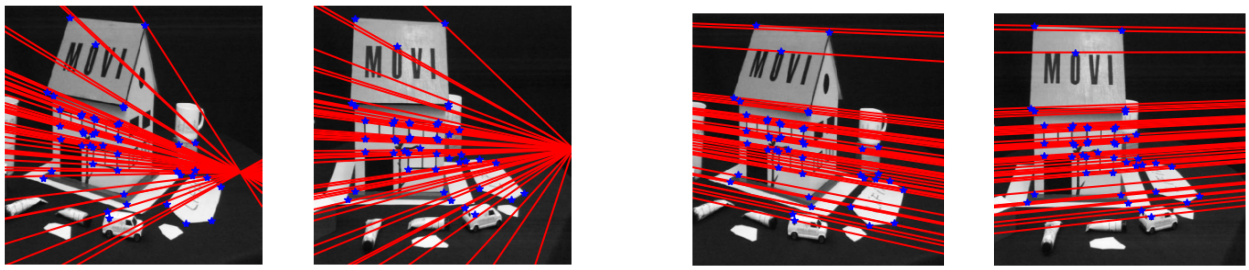
Listing 3: Code Snippet

```
Distance to lines in image 1 for LLS: 9.70143882944614
Distance to lines in image 2 for LLS: 14.568227190516817
Distance to lines in image 1 for normalized: 0.8895134540568704
Distance to lines in image 2 for normalized: 0.8917343723800033
```



(a) Linear Least-Squares Method (b) Normalized Linear Least-Squares Method

Figure 2: Landmarks and corresponding epipolar lines using linear least-squares version of the eight-point algorithm (2a), and normalized linear least-squares version of the eight-point algorithm (2b) for computing Fundamental Matrix for Set 1.



(a) Linear Least-Squares Method (b) Normalized Linear Least-Squares Method

Figure 3: Landmarks and corresponding epipolar lines using linear least-squares version of the eight-point algorithm (3a), and normalized linear least-squares version of the eight-point algorithm (3b) for computing Fundamental Matrix for Set 2.

3 The Factorization Method (15 points)

In this question, you will explore the factorization method, initially presented by Tomasi and Kanade, for solving the affine structure from motion problem. You will implement the methods in `p3.py` and complete the following:

(a) Implement the factorization method as described in lecture and in the course notes. Complete the function `factorization_method()`. Submit a copy of your code.
[5 points]

```
'''
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):
    # let's find centroid of points in their respective cameras.
    points_im1_centroid = np.array((np.mean(points_im1[:,0]), np.mean(
    points_im1[:,1])), dtype=np.float32)
    points_im2_centroid = np.array((np.mean(points_im2[:,0]), np.mean(
    points_im2[:,1])), dtype=np.float32)

    # compute centered points
    x1j_hat = points_im1[:,:2] - points_im1_centroid
    x2j_hat = points_im2[:,:2] - points_im2_centroid

    # build the measurement matrix
    D = []
    # camera 1
    D.append([x1j_hat[j,0] for j in range(x1j_hat.shape[0])])
    D.append([x1j_hat[j,1] for j in range(x1j_hat.shape[0])])
    # camera 2
    D.append([x2j_hat[j,0] for j in range(x2j_hat.shape[0])])
    D.append([x2j_hat[j,1] for j in range(x2j_hat.shape[0])])
    D = np.array(D, dtype=np.float32)

    # This D can be factorized as [D=MS]; where M=Motion, S=Structure
    # let's decompose D using SVD
    u, s, v_t = np.linalg.svd(D, full_matrices=True)
    # For D=MS, M captures camera parameters (2m x 3), and S captures 3D
    points ( 3 x n)
    motion = u[:,:3]
    s_mat = np.diag(s[:3])
    structure = np.dot(s_mat, v_t[:3,:])

    # return motion and structure
    return structure, motion
```

Listing 4: Code Snippet

(b) Run the provided code that plots the resulting 3D points. Compare your result to the ground truth provided. The results should look identical, except for a scaling and rotation. Explain why this occurs.
[3 points]

The 3D points obtained using Factorization Method is true up to an affine transformation. This happens because we make affine assumption about the camera model (weak perspective model) in order to reduce a set of non-linear equations to a set of linear equations. This method of obtaining structure from motion results in *Affine* and *Similarity* ambiguity. Resulting 3D points are shown in Figure 4. Notice that the obtained set of points look similar but the scaling and rotation is different.

(c) Report the 4 singular values from the SVD decomposition. Why are there 4 non-zero singular

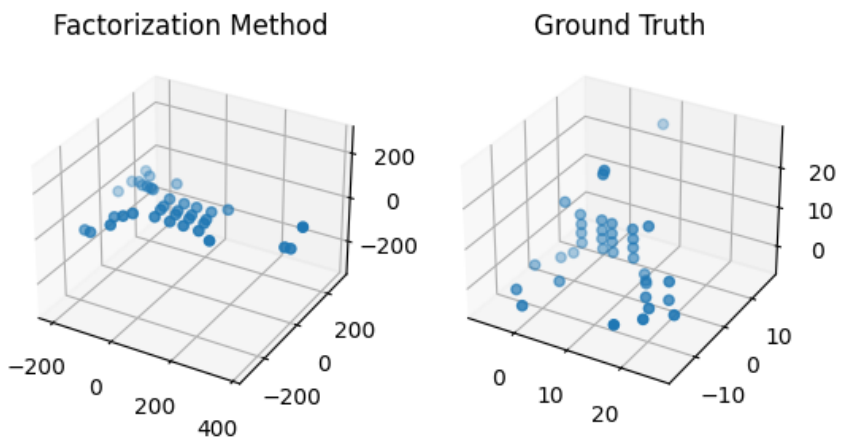


Figure 4: Factorization method for solving affine structure from motion. Plots of estimated 3D points up to an affine transformation for Set 1.

values? How many non-zero singular values would you expect to get in the idealized version of the method, and why?
[2 points]

The 4 singular values are: $\sigma_1 = 959.5852$, $\sigma_2 = 540.47614$, $\sigma_3 = 184.43175$, $\sigma_4 = 27.915195$.

Ideally, the rank of Measurement Matrix (D) is 3 because D is the product of two matrices whose max dimension is 3; i.e. D can be expressed as the product of $2m$x3 motion matrix M and the 3xn structure matrix. However, in practice, because of measurement noise and the affine camera approximation, $rank(D) > 3$.

(d) The next part of the code will now only load a subset of the original correspondences. Compare your new results to the ground truth, and explain why they no longer appear similar.
[3 points]

Simlar to part (b), the 3D points obtained using Factorization Method is true up to an affine transformation because we make affine approximation for the camera model. This method of obtaining structure from motion results in *Affine* and *Similarity* ambiguity. This causes the points to look similar except for the scaling and rotation.

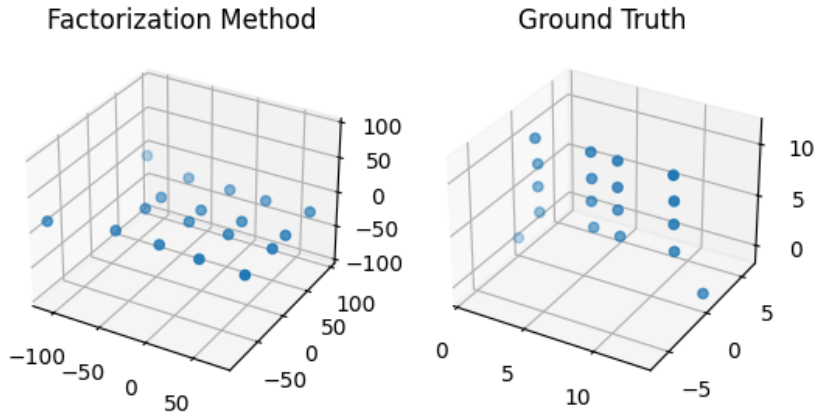


Figure 5: Factorization method for solving affine structure from motion. Plots of estimated 3D points up to an affine transformation for Set 1 Subset.

(e) Report the new singular values, and compare them to the singular values that you found previously. Explain any major changes.
[2 points]

The 4 singular values are: $\sigma_1 = 264.54398$, $\sigma_2 = 210.06071$, $\sigma_3 = 7.219218$, $\sigma_4 = 5.128577$.

This looks different from the singular values obtained on Set 1 because the solution found with the factorization method is not unique and there are many ways one could form the $D = MS$ product. Set 1 subset also removes some of the 3D points farther away from the origin $(0, 0)$, which ends up removing some outliers from the set as well as reduces the overall scale of points. This causes the singular values to be of a smaller scale compared to the ones found before and σ_4 to be relatively smaller.

4 Triangulation in Structure From Motion (35 points)

Figure 6: The set of images used in this structure from motion reconstruction.

Structure from motion is inspired by our ability to learn about the 3D structure in the surrounding environment by moving through it. Given a sequence of images, we are able to simultaneously estimate both the 3D structure and the path the camera took. In this problem, you will implement significant parts of a structure from motion framework, estimating both R and T of the cameras, as well as generating the locations of points in 3D space. Recall that in the previous problem we triangulated points assuming affine transformations. However, in the actual structure from motion problem, we assume projective transformations. By doing this problem, you will learn how to solve this type of triangulation. In Course Notes 4, we go into further detail about this process. You will implement the methods in `p4.py` and complete the following:

(a) Given correspondences between pairs of images, we compute the respective Fundamental and Essential matrices. Given the Essential matrix, we must now compute the R and T between the two cameras. However, recall that there are four possible R, T pairings. In this part, we seek to find these four possible pairings, which we will later be able to decide between. In the course notes, we explain in detail the following process:

1. To compute R: Given the singular value decomposition $E = UDV^T$, we can rewrite $E = MQ$ where $M = UZU^T$ and $Q = UWV^T$ *or* UW^TV^T, where

$$Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

   Note that this factorization of E only guarantees that Q is orthogonal. To find a rotation, we simply compute $R = (\det Q)Q$.
2. To compute T: Given that $E = U\Sigma V^T$, T is simply either u_3 or $-u_3$, where u_3 is the third column vector of U.

Implement this in the function `estimate_initial_RT()`. We provide the correct R, T, which should be contained in your computed four pairs of R, T. Submit the four pairs of R, T and a copy of your code.
[5 points]

```
'''
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):
    # decompose E using SVD into U.Sigma.V^T
    U, Sigma, V_t = np.linalg.svd(E, full_matrices=True)
    # build 'Z' and 'W'
    Z = np.array([[0, 1, 0],
                  [-1, 0, 0],
                  [0, 0, 0]], dtype=np.float32)
    W = np.array([[0, -1, 0],
                  [1, 0, 0],
                  [0, 0, 1]], dtype=np.float32)
    # E can be re-written as E=MQ; where M=U.Z.U^T and Q=U.W.V^T or Q=U.W^T.V^
    T
    M = np.dot(U, np.dot(Z, U.T))
    Q1 = np.dot(U, np.dot(W, V_t))
    Q2 = np.dot(U, np.dot(W.T, V_t))
    # R can be computed as R=(det Q)Q
    R1 = np.dot(np.linalg.det(Q1), Q1)
    R2 = np.dot(np.linalg.det(Q2), Q2)
    # E = U  V^T, T is simply either u3 or   u3  , where u3 is the third column
     vector of U
    T1 = np.array(U[:,2])
    T2 = np.array(-U[:,2])

    # compose 4 possible RT
    RT = np.zeros((4,3,4), dtype=np.float32)
    for i, (R, T) in enumerate(zip((R1, R1, R2, R2), (T1, T2, T1, T2))):
        RT[i][:,:3] = R
        RT[i][:,3] = T

    # return initial RT
    return RT
```

Listing 5: Code Snippet

The code snippet in Listing 5 computes 4 possible R, T pairs and are given below.

```
RT #1:
 [[ 0.9830525  -0.11787055 -0.14040758  0.9994123 ]
  [-0.11925737 -0.9928623  -0.00147453 -0.00886961]
  [-0.13923158  0.01819418 -0.9900927   0.03311219]]

RT #2:
 [[ 0.9830525  -0.11787055 -0.14040758 -0.9994123 ]
  [-0.11925737 -0.9928623  -0.00147453  0.00886961]
  [-0.13923158  0.01819418 -0.9900927  -0.03311219]]

RT #3:
 [[ 0.97364134 -0.09878708 -0.20558119  0.9994123 ]
  [ 0.10189205  0.9947851   0.00454512 -0.00886961]
```

```
  [ 0.20406011 -0.02537241  0.9786295   0.03311219]]

RT #4:
 [[ 0.97364134 -0.09878708 -0.20558119 -0.9994123 ]
  [ 0.10189205  0.9947851   0.00454512  0.00886961]
  [ 0.20406011 -0.02537241  0.9786295  -0.03311219]]
```

(b) In order to distinguish the correct R, T pair, we must first know how to find the 3D point given matching correspondences in different images. The course notes explain in detail how to compute a linear estimate (DLT) of this 3D point:

1. For each image i, we have $p_i = M_i P$, where P is the 3D point, p_i is the homogenous image coordinate of that point, and M_i is the projective camera matrix.
2. Formulate matrix
$$A = \begin{bmatrix} p_{1,1}m^{3\top} - m^{1\top} \\ p_{1,2}m^{3\top} - m^{2\top} \\ \vdots \\ p_{n,1}m^{3\top} - m^{1\top} \\ p_{n,2}m^{3\top} - m^{2\top} \end{bmatrix}$$
where $p_{i,1}$ and $p_{i,2}$ are the xy coordinates in image i and $m^{k\top}$ is the k-th row of M.
3. The 3D point can be solved for by using the singular value decomposition.

Implement the linear estimate of this 3D point in `linear_estimate_3d_point()`. Like before, we print out a way to verify that your code is working. Submit the output generated by this part of the code and a copy of your code.
[5 points]

```
'''
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):
    # We can re-write [p x MP = 0] in the form [AP = 0] and solve for P
    # by decomposing A using SVD
    # let's formulate the A matrix
    A = []
    for i, M in enumerate(camera_matrices):
        im_pt = image_points[i]
        A.append(im_pt[0]*M[2] - M[0])
        A.append(im_pt[1]*M[2] - M[1])
    A = np.array(A, dtype=np.float64)
    # solve for P
    u, s, v_t = np.linalg.svd(A, full_matrices=True)
    # P can be obtained as the last column of v or last row of v_transpose
    P = v_t[-1]
    # homogeneous to euclidean conversion
    P = (P / P[-1])[:3]
    # convert to float64
    P = np.array(P, dtype=np.float64)

    # return the 3D point
```

```
30      return P
```

Listing 6: Code Snippet

```
1 Difference between the estimated and expected 3d point: 0.0029243053036863698
```

(c) However, we can do better than linear estimates, but usually this falls under some iterative nonlinear optimization. To do this kind of optimization, we need some residual. A simple one is the reprojection error of the correspondences, which is computed as follows:
For each image i, given camera matrix M_i, the 3D point P, we compute $y = M_i P$, and find the image coordinates

$$p_i' = \frac{1}{y_3} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Given the ground truth image coordinates p_i, the reprojection error e_i for image i is

$$e_i = p_i' - p_i$$

The Jacobian is written as follows:

$$J = \begin{bmatrix} \frac{\partial e_1}{\partial P_1} & \frac{\partial e_1}{\partial P_2} & \frac{\partial e_1}{\partial P_3} \\ \vdots & \vdots & \vdots \\ \frac{\partial e_m}{\partial P_1} & \frac{\partial e_m}{\partial P_2} & \frac{\partial e_m}{\partial P_3} \end{bmatrix}$$

Recall that each e_i is a vector of length two, so the whole Jacobian is a $2K \times 3$ matrix, where K is the number of cameras. Fill in the methods `reprojection_error()` and `jacobian()`, which computes the reprojection error and Jacobian for a 3D point and its list of images. Like before, we print out a way to verify that your code is working. Submit the output generated by this part of the code and a copy of your code.
[5 points]

Gauss-Newton algorithm can be used to update our estimate of triangulated 3D point by correcting our estimate towards even a better one that minimizes the reprojection error.

Reprojection error is given as: $e_i = M\hat{P}_i - p_i$; where $\hat{P}_i$ is the estimate of triangulated 3D point, M is the camera projection matrix, and p_i is the measured 2D point.

$$e_i = \left[M\hat{P}_i\right]_e - p_i \quad \Big| \quad M\hat{P}_i = \begin{bmatrix} m_1\hat{P}_i \\ m_2\hat{P}_i \\ m_3\hat{P}_i \end{bmatrix} \Rightarrow \left[M\hat{P}_i\right]_e = \frac{1}{m_3\hat{P}_i}\begin{bmatrix} m_1\hat{P}_i \\ m_2\hat{P}_i \end{bmatrix} \quad \text{— (i)}$$

$$= p_i' - p_i$$

where, $p_i' = \frac{1}{y_3}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ in euclidean coordinates

We need to compute Jacobian, "J", in order to update our estimate of the 3D point.

i.e. $\hat{P} = \hat{P} - (J^T J)^{-1} J^T e$; for $\hat{P} = \begin{bmatrix} \hat{P}_1 \\ \hat{P}_2 \\ \hat{P}_3 \end{bmatrix}$

where,

$$J = \begin{bmatrix} \frac{\partial e_1}{\partial \hat{P}_1} & \frac{\partial e_1}{\partial \hat{P}_2} & \frac{\partial e_1}{\partial \hat{P}_3} \\ \vdots & & \\ \frac{\partial e_m}{\partial \hat{P}_1} & \frac{\partial e_m}{\partial \hat{P}_2} & \frac{\partial e_m}{\partial \hat{P}_3} \end{bmatrix}$$

m = No. of Cameras

each e_i is vector of length 2

Let's compute the Jacobian:

The camera projection matrix can be given as:

$$M = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \quad \text{(ii)}$$

$$\Rightarrow e_i = \frac{1}{m_3 \hat{P}} \begin{bmatrix} m_1 \hat{P} \\ m_2 \hat{P} \end{bmatrix} - p_i$$

$$\text{where, } p_i = \frac{1}{m_3 P} \begin{bmatrix} m_1 P \\ m_2 P \end{bmatrix} \Rightarrow \frac{\partial}{\partial \hat{P}}(p_i) = 0$$

Let's compute $\frac{\partial e_i}{\partial P_1}$:

$$\frac{\partial e_i}{\partial P_1} = \frac{\partial}{\partial P_1}\left(\frac{1}{m_3 \hat{P}} \begin{bmatrix} m_1 \hat{P} \\ m_2 \hat{P} \end{bmatrix}\right)$$

$$= \begin{bmatrix} \frac{\partial}{\partial P_1}\left(\frac{m_1 \hat{P}}{m_3 \hat{P}}\right) \\ \frac{\partial}{\partial P_1}\left(\frac{m_2 \hat{P}}{m_3 \hat{P}}\right) \end{bmatrix}$$

$$\begin{aligned} m_1 \hat{P} &= m_{11} \hat{P}_1 + m_{12} \hat{P}_2 + m_{13} \hat{P}_3 \\ m_2 \hat{P} &= m_{21} \hat{P}_1 + m_{22} \hat{P}_2 + m_{23} \hat{P}_3 \\ m_3 \hat{P} &= m_{31} \hat{P}_1 + m_{32} \hat{P}_2 + m_{33} \hat{P}_3 \end{aligned}$$

Recall the quotient rule:

$$\left(\frac{f}{g}\right)' = \left(\frac{f'g - fg'}{g^2}\right)$$

$$\Rightarrow \frac{\partial e_i}{\partial P_1} = \begin{bmatrix} \dfrac{(m_{11}\hat{P}_1)(m_3\hat{P}) - (m_1\hat{P})(m_{31}\hat{P}_1)}{(m_3\hat{P})^2} \\ \dfrac{(m_{21}\hat{P}_1)(m_3\hat{P}) - (m_2\hat{P})(m_{31}\hat{P}_1)}{(m_3\hat{P})^2} \end{bmatrix} \quad \text{— (iii)}$$

Similarly,

$$\frac{\partial e_i}{\partial P_2} = \begin{bmatrix} \dfrac{(m_{12}\hat{P}_2)(m_3\hat{P}) - (m_1\hat{P})(m_{32}\hat{P}_2)}{(m_3\hat{P})^2} \\ \dfrac{(m_{22}\hat{P}_2)(m_3\hat{P}) - (m_2\hat{P})(m_{32}\hat{P}_2)}{(m_3\hat{P})^2} \end{bmatrix} \quad \text{— (iv)}$$

$$\frac{\partial e_i}{\partial P_3} = \begin{bmatrix} \dfrac{(m_{13}\hat{P}_3)(m_3\hat{P}) - (m_1\hat{P})(m_{33}\hat{P}_3)}{(m_3\hat{P})^2} \\ \dfrac{(m_{23}\hat{P}_3)(m_3\hat{P}) - (m_2\hat{P})(m_{33}\hat{P}_3)}{(m_3\hat{P})^2} \end{bmatrix} \quad \text{— (v)}$$

```
'''
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 2M reprojection error vector
'''
def reprojection_error(point_3d, image_points, camera_matrices):
    error = []
    # compute the projected 2d points
    for i, M in enumerate(camera_matrices):
        # euclidean to homogeneous conversion
        pt_3d_homogeneous = np.append(point_3d, 1.0)
        # compute reprojected 2d point
        projected_2d_pt_homogeneous = np.dot(M, pt_3d_homogeneous)
        # homogeneous to euclidean conversion
        projected_2d_pt = (projected_2d_pt_homogeneous /
    projected_2d_pt_homogeneous[-1])[:2]
        # compute reprojection error
        error.extend(projected_2d_pt - image_points[i])
    error = np.array(error, dtype=np.float64)
    # return the reprojection error
    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):
    point_3d_homogeneous = np.append(point_3d, 1.0)
    # compute the jacobian matrix
    J = []
    for M in camera_matrices:
        m1P_hat = np.dot(M[0,:], point_3d_homogeneous.T)
        m2P_hat = np.dot(M[1,:], point_3d_homogeneous.T)
        m3P_hat = np.dot(M[2,:], point_3d_homogeneous.T)

        J.append([((M[0,0]*m3P_hat - M[2,0]*m1P_hat) / (m3P_hat ** 2)), \
                  ((M[0,1]*m3P_hat - M[2,1]*m1P_hat) / (m3P_hat ** 2)), \
                  ((M[0,2]*m3P_hat - M[2,2]*m1P_hat) / (m3P_hat ** 2))])
        J.append([((M[1,0]*m3P_hat - M[2,0]*m2P_hat) / (m3P_hat ** 2)), \
                  ((M[1,1]*m3P_hat - M[2,1]*m2P_hat) / (m3P_hat ** 2)), \
                  ((M[1,2]*m3P_hat - M[2,2]*m2P_hat) / (m3P_hat ** 2))])
    J = np.array(J, dtype=np.float32)

    # return the jacobian matrix
    return J
```

Listing 7: Code Snippet

```
Difference between obtained and expected Error :  8.301300130674275e-07
Difference between obtained and expected Jacobian:  2.517187985695557e-05
```

(d) Implement the Gauss-Newton algorithm, which finds an approximation to the 3D point that minimizes this reprojection error. Recall that this algorithm needs a good initialization, which we have from our linear estimate in part (b). Also recall that the Gauss-Newton algorithm is not guaranteed to converge, so, in this implementation, you should update the estimate of the point $\hat{P}$ for 10 iterations (for this problem, you do not have to worry about convergence criteria for early termination):

$$\hat{P} = \hat{P} - (J^T J)^{-1} J^T e$$

where J and e are the Jacobian and error computed from the previous part. Implement the Gauss-Newton algorithm to find an improved estimate of the 3D point in the `nonlinear_estimate_3d_point()` function. Like before, we print out a way to verify that your code is working. Submit the output generated by this part of the code and a copy of your code.
[5 points]

```
'''
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):
    # compute linear estimate of 3D points
    point_3d = linear_estimate_3d_point(image_points, camera_matrices)
    # optimize for 'n_iter' iterations
    n_iter = 10
    for iter in range(n_iter):
        # compute pre-optimization reprojection error
        err_pre_optim = reprojection_error(point_3d, image_points,
    camera_matrices)
        # compute Jacobian
        J = jacobian(point_3d, camera_matrices)
        point_3d = point_3d - np.dot(np.linalg.inv(np.dot(J.T, J)),np.dot(J.T,
     err_pre_optim))
        # compute post-optimization reprojection error
        err_post_optim = np.linalg.norm(reprojection_error(point_3d,
    image_points, camera_matrices))
        # print('Iter {:2d} | pre-optim error: {:.4f} | post-optim error: {:.4
    f}'.format(iter, np.linalg.norm(err_pre_optim), err_post_optim))

    # return the non-linear estimate of 3d point
    return point_3d
```

Listing 8: Code Snippet

```
Reprojection error from Linear method : 98.73542356894183
Reprojection error from Nonlinear method : 95.59481784846044
```

(e) Now finally, go back and distinguish the correct R, T pair from part (a) by implementing the method `estimate_RT_from_E()`. You will do so by:

1. First, compute the location of the 3D point of each pair of correspondences given each R, T pair
2. Given each R, T you will have to find the 3D point's location in that R, T frame. The correct R, T pair is the one for which the most 3D points have positive depth (z-coordinate)

with respect to both camera frames. When testing depth for the second camera, we must transform our computed point (which is the frame of the first camera) to the frame of the second camera.
[5 points]

```
'''
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):
    # estimate 4 possible RT from Essential Matrix
    RT_init = estimate_initial_RT(E)
    # create a list of all triangulated 3d points for each possible RT
    # for N measured points, in M cameras, shape will be -> 4 x N x M x 3
    points_3d_rt = np.zeros((4, image_points.shape[0], image_points.shape[1],
    3), dtype=np.float32)
    # go through all possible RT
    for i, RT in enumerate(RT_init):
        # go through all 2D points tracked in both cameras
        for j, image_point in enumerate(image_points):
            # image_point_cam1 = image_point[0]
            # image_point_cam2 = image_point[1]
            # build a 3x4 matrix for camera 1
            camera_mtx_1 = np.zeros((3,4), dtype=np.float32)
            camera_mtx_1[:,:3] = K
            camera_mtx_1[-1,-1] = 1.0
            # build a projection matrix for camera 2
            camera_mtx_2 = np.dot(K, RT)
            # combine the two camera matrices
            camera_matrices = np.zeros((2, 3, 4), dtype=np.float32)
            camera_matrices[0] = camera_mtx_1
            camera_matrices[1] = camera_mtx_2
            # get non-linear estimate of 3d point
            point_3d = nonlinear_estimate_3d_point(image_point,
    camera_matrices)
            point_3d_homogeneous = np.append(point_3d, 1.0)
            # add to the complete list
            points_3d_rt[i,j,0] = point_3d # camera 1
            points_3d_rt[i,j,1] = np.dot(RT, point_3d_homogeneous) # camera 2

    # for each RT, count number of points that fall in front of all cameras.
    # the one with maximum number of points will correspond to the correct RT
    correct_RT = None
    max_count = 0
    for i in range(RT_init.shape[0]):
        count = np.sum(points_3d_rt[i,:,:,2] > 0.)
        if count > max_count:
            max_count = count
            correct_RT = RT_init[i]

    # return correct RT
    return correct_RT
```

Listing 9: Code Snippet

(f) Congratulations! You have implemented a significant portion of a structure from motion pipeline. Your code is able to compute the rotation and translations between different cameras, which provides the motion of the camera. Additionally, you have implemented a robust method to triangulate 3D points, which enable us to reconstruct the structure of the scene. In order to run the full structure from motion pipeline, please change the variable `run_pipeline` at the top of the main function to `True`. Submit the final plot of the reconstructed statue. Hopefully, you can see a point cloud that looks like the frontal part of the statue in the above sequence of images.

Note: Since the class is using Python, the structure from motion framework we use is not the most efficient implementation. It will be common that generating the final plot may take a few minutes to complete. Furthermore, Matplotlib was not built to be efficient for 3D rendering. Although it's nice to wiggle the point cloud to see the 3D structure, you may find that the GUI is laggy. If we used better options that incorporate OpenGL (see Glumpy), the visualization would be more responsive. However, for the sake of the class, we will only use the numpy-related libraries.
[10 points]

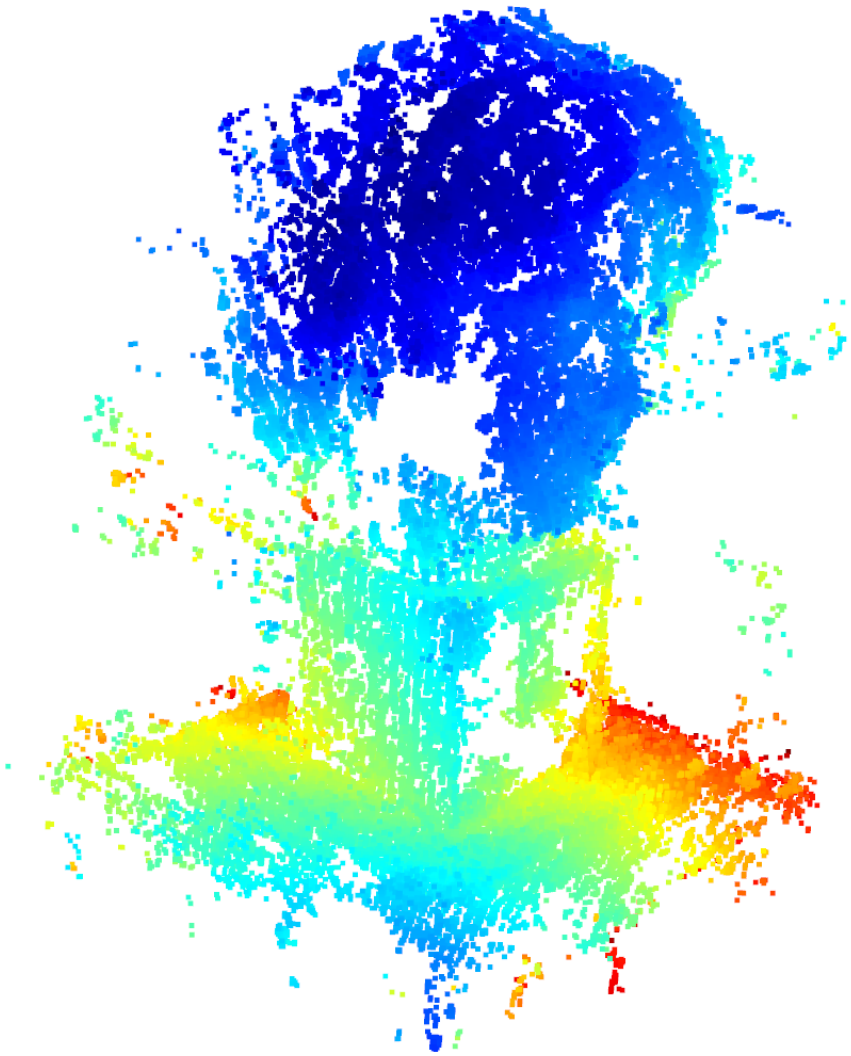


Figure 7: Point-cloud obtained from SFM pipeline.