Q1.1

$x_1$: $projection\ of\ 3D\ point\ in\ camera\ 1's\ coordinate\ system$

$x_2$: $projection\ of\ 3D\ point\ in\ camera\ 2's\ coordinate\ system$

$$x_2^T\ F\ x_1\ =\ 0 \tag{a}$$

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = 0 \tag{b}$$

$$\begin{bmatrix} xx' & xy' & x & yx' & yy' & y & x' & y & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \tag{c}$$

From diagram $x_1$ and $x_2$ are (0,0,1) and $(0,0,1)^T$ subs in (c)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0$$

$$F_{33}\ =\ 0$$

Q1.2

$x_1$: *projection of 3D point in camera 1's coordinate system*

$x_2$: *projection of 3D point in camera 2's coordinate system*

E: *Essential matrix*

$\ell_2$: *epipolar line in camera 2's coordinate system*

$\ell_1$: *epipolar line in camera 1's coordinate system*

$$\ell_2 = x_1^T E \qquad (1)$$

$$\ell_1 = x_2^T E^T \qquad (2)$$

$$E = T \times R \qquad (3)$$

$$E = \hat{T} R \qquad (4)$$

We know that $a \times b = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \hat{a} b$

For translation in x,

$$T = [t \quad 0 \quad 0] \qquad (5)$$

$$\hat{T} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t \\ 0 & t & 0 \end{bmatrix} \qquad (6)$$

No rotation,

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (7)$$

From (4),

$$E = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t \\ 0 & t & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (8)$$

$$E = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t \\ 0 & t & 0 \end{bmatrix} \qquad (9)$$

$$E^T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t \\ 0 & -t & 0 \end{bmatrix} \qquad (10)$$

From (1) and (2),

$$\ell_2 = \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t \\ 0 & t & 0 \end{bmatrix} \qquad (11)$$

$$\ell_2 = \begin{bmatrix} 0 & t & -y_1 t \end{bmatrix} \qquad (12)$$

$$\ell_1 = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t \\ 0 & -t & 0 \end{bmatrix} \qquad (13)$$

$$\ell_1 = \begin{bmatrix} 0 & -t & y_2 t \end{bmatrix} \qquad (14)$$

(12) and (14) has no x component, hence parallel to x-axis

Q1.3

X: projection of 3D point in camera 1's coordinate system

$X_i$: projection of 3D point in camera 2's coordinate system at $i^{th}$ time stamp

$X_j$: projection of 3D point in camera 2's coordinate system at $j^{th}$ time stamp

$$X_i = K(R_i X + T_i) \qquad (1)$$

$$X_j = K\left(R_j X + T_j\right) \qquad (2)$$

Rearranging (1)

$$X = R_i^{-1}(K^{-1}X_i - T_i)$$

$$X = R_i^{-1}(K^{-1}X_i) - R_i^{-1}T_i \qquad (3)$$

Subs (3) in (2),

$$X_j = K\left(R_j(R_i^{-1}(K^{-1}X_i) - R_i^{-1}T_i) + T_j\right) \qquad (4)$$

$$X_j = (K\, R_j R_i^{-1} K^{-1})X_i + (-\, K\, R_j R_i^{-1} T_i + K\, T_j) \qquad (5)$$

$$X_j = R_{rel} X_i + T_{rel}$$

$$R_{rel} = K\, R_j R_i^{-1} K^{-1} \qquad (6)$$

$$T_{rel} = -\, K\, R_j R_i^{-1} T_i + K\, T_j \qquad (7)$$

$$E = T_{rel} \times R_{rel} \qquad (8)$$

$$E = (-\, K\, R_j R_i^{-1} T_i + K\, T_j) \times (K\, R_j R_i^{-1} K^{-1}) \qquad (9)$$

$$F = K^{-T} E\, K^{-1} \qquad (10)$$

$$F = K^{-T}(-\, K\, R_j R_i^{-1} T_i + K\, T_j) \times (K\, R_j R_i^{-1} K^{-1})\, K^{-1} \qquad (11)$$

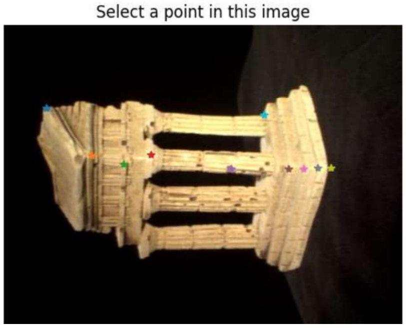
$$F = K^{-T}(T_{rel} \times R_{rel})\, K^{-1} \qquad (12)$$

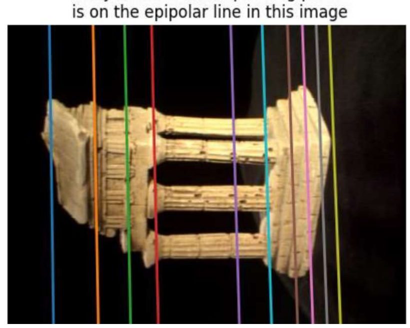
Q2.1

F = [[-2.18962367e-07 2.95584511e-05 -2.51851099e-01]

[ 1.28367203e-05 -6.63934216e-07 2.63094865e-03]

[ 2.42194841e-01 -6.81933857e-03 1.00000000e+00]]




```
import numpy as np
import matplotlib.pyplot as plt
from helper import displayEpipolarF, calc_epi_error, toHomogenous, refineF,
_singularize
"""
Q2.1: Eight Point Algorithm
    Input:  pts1, Nx2 Matrix
            pts2, Nx2 Matrix
            M, a scalar parameter computed as max (imwidth, imheight)
    Output: F, the fundamental matrix

    HINTS:
    (1) Normalize the input pts1 and pts2 using the matrix T.
    (2) Setup the eight point algorithm's equation.
    (3) Solve for the least square solution using SVD.
    (4) Use the function `_singularize` (provided) to enforce the singularity
condition.
    (5) Use the function `refineF` (provided) to refine the computed
fundamental matrix.
        (Remember to use the normalized points instead of the original points)
    (6) Unscale the fundamental matrix
"""
def eightpoint(pts1, pts2, M):
    npts1 = pts1/M
    npts2 = pts2/M

    A = np.zeros((npts1.shape[0], 9))
    A[:, 0] = npts1[:, 0] * npts2[:, 0]
    A[:, 1] = npts1[:, 1] * npts2[:, 0]
    A[:, 2] = npts2[:, 0]
    A[:, 3] = npts1[:, 0] * npts2[:, 1]
    A[:, 4] = npts1[:, 1] * npts2[:, 1]
```

```python
    A[:, 5] = npts2[:, 1]
    A[:, 6] = npts1[:, 0]
    A[:, 7] = npts1[:, 1]
    A[:, 8] = 1

    _, _, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)

    F = refineF(F, npts1, npts2)
    F = F / F[2, 2]

    scaleT = np.array([[1 / M, 0, 0], [0, 1 / M, 0], [0, 0, 1]])
    F = scaleT.T @ F @ scaleT
    return F


if __name__ == "__main__":
    correspondence = np.load("data/some_corresp.npz")  # Loading
correspondences
    intrinsics = np.load("data/intrinsics.npz")  # Loading the intrinscis of
the camera
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    pts1, pts2 = correspondence["pts1"], correspondence["pts2"]
    im1 = plt.imread("data/im1.png")
    im2 = plt.imread("data/im2.png")

    M = np.max([*im1.shape, *im2.shape])
    F = eightpoint(pts1, pts2, M)
    print("\n F = ", F)
    print("\n M", M)
    np.savez("submission/q2_1.npz", F, M)

    # Q2.1
    # displayEpipolarF(im1, im2, F)

    # Simple Tests to verify your implementation:
    pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)

    assert F.shape == (3, 3)
    assert F[2, 2] == 1
    assert np.linalg.matrix_rank(F) == 2
    assert np.mean(calc_epi_error(pts1_homogenous, pts2_homogenous, F)) < 1
```

Q3.1

E = [[-3.36615963e+00  4.56052787e+02 -2.47343036e+03]

[ 1.98055779e+02 -1.02807951e+01  6.44171617e+01]

[ 2.48028021e+03  1.98174709e+01  1.00000000e+00]]

```
import numpy as np
import matplotlib.pyplot as plt
from q2_1_eightpoint import eightpoint
"""
Q3.1: Compute the essential matrix E.
    Input:  F, fundamental matrix
            K1, internal camera calibration matrix of camera 1
            K2, internal camera calibration matrix of camera 2
    Output: E, the essential matrix
"""
def essentialMatrix(F, K1, K2):
    E = K2.T @ F @ K1
    E = E / E[2, 2]
    return E


if __name__ == "__main__":
    correspondence = np.load("data/some_corresp.npz")  # Loading
correspondences
    intrinsics = np.load("data/intrinsics.npz")  # Loading the intrinscis of
the camera
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    pts1, pts2 = correspondence["pts1"], correspondence["pts2"]
    im1 = plt.imread("data/im1.png")
    im2 = plt.imread("data/im2.png")

    F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
    E = essentialMatrix(F, K1, K2)
    np.savez("submission/q3_1.npz", E)
    print("\n E = ", E)

    # Simple Tests to verify your implementation:
    assert np.linalg.matrix_rank(E) == 2
```

Q3.2

X: 3D point

C1, C2: Camera matrices

$x_i, x_i'$: Matched points

$x = C\,X \qquad\qquad (homogenous\ form)$

$x = \alpha\,C\,X \qquad\qquad (inhomogenous\ form)$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \alpha \begin{bmatrix} \mathcal{C}_1 & \mathcal{C}_2 & \mathcal{C}_3 & \mathcal{C}_4 \\ \mathcal{C}_5 & \mathcal{C}_6 & \mathcal{C}_7 & \mathcal{C}_8 \\ \mathcal{C}_9 & \mathcal{C}_{10} & \mathcal{C}_{11} & \mathcal{C}_{12} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

To remove $\alpha$,

$x \times P\,X = 0$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \alpha \begin{bmatrix} -\boldsymbol{C}_1^T- \\ -\boldsymbol{C}_2^T- \\ -\boldsymbol{C}_3^T- \end{bmatrix} \begin{bmatrix} | \\ \boldsymbol{X} \\ | \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \alpha \begin{bmatrix} \boldsymbol{C}_1^T\boldsymbol{X} \\ \boldsymbol{C}_2^T\boldsymbol{X} \\ \boldsymbol{C}_3^T\boldsymbol{X} \end{bmatrix}$$

From property of cross products,

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \times \begin{bmatrix} \boldsymbol{C}_1^T\boldsymbol{X} \\ \boldsymbol{C}_2^T\boldsymbol{X} \\ \boldsymbol{C}_3^T\boldsymbol{X} \end{bmatrix} = \begin{bmatrix} y\boldsymbol{C}_3^T\boldsymbol{X} - \boldsymbol{C}_2^T\boldsymbol{X} \\ \boldsymbol{C}_1^T\boldsymbol{X} - x\boldsymbol{C}_3^T\boldsymbol{X} \\ x\boldsymbol{C}_2^T\boldsymbol{X} - y\boldsymbol{C}_1^T\boldsymbol{X} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Note: Third line is a linear combination of first and second lines (x times the first line plus y times the second line)

$$\begin{bmatrix} \boldsymbol{y}\boldsymbol{C}_3^T - \boldsymbol{C}_2^T \\ \boldsymbol{C}_1^T - \boldsymbol{x}\boldsymbol{C}_3^T \end{bmatrix} \boldsymbol{X} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Similar to $A_i X = 0$

Concatenate 2D points from both images,

$$\begin{bmatrix} \boldsymbol{y}\boldsymbol{C}_3^T - \boldsymbol{C}_2^T \\ \boldsymbol{C}_1^T - \boldsymbol{x}\boldsymbol{C}_3^T \\ \boldsymbol{y}'\boldsymbol{C}'^T_3 - \boldsymbol{C}'^T_2 \\ \boldsymbol{C}'^T_1 - \boldsymbol{x}'\boldsymbol{C}'^T_3 \end{bmatrix} \boldsymbol{X} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$AX = 0$

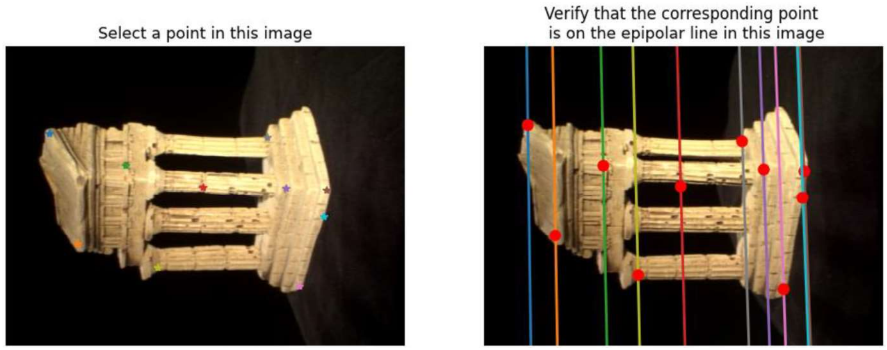
$$A = \begin{bmatrix} \boldsymbol{y}\boldsymbol{C}_3^T - \boldsymbol{C}_2^T \\ \boldsymbol{C}_1^T - \boldsymbol{x}\boldsymbol{C}_3^T \\ \boldsymbol{y}'\boldsymbol{C}'^T_3 - \boldsymbol{C}'^T_2 \\ \boldsymbol{C}'^T_1 - \boldsymbol{x}'\boldsymbol{C}'^T_3 \end{bmatrix}$$

```
def triangulate(C1, pts1, C2, pts2):

    P = np.zeros((pts1.shape[0], 3))
    err = 0
    A = np.zeros((4, 4))

    for i in range(pts1.shape[0]):
        A[0, :] =  (pts1[i, 1] * C1[2, :]) - C1[1, :]
        A[1, :] = -(pts1[i, 0] * C1[2, :]) + C1[0, :]
        A[2, :] =  (pts2[i, 1] * C2[2, :]) - C2[1, :]
        A[3, :] = -(pts2[i, 0] * C2[2, :]) + C2[0, :]

        _, _, V = np.linalg.svd(A)
        homoP = V[-1, :]
        homoP = homoP / homoP[3]
        P[i, :] = homoP[0:3]
        err += np.sum(np.linalg.norm(pts1[i] - (C1 @ homoP.T)[:2] / (C1 @
homoP.T)[2])**2 + np.linalg.norm(pts2[i] - (C2 @ homoP.T)[:2] / (C2 @
homoP.T)[2])**2)
    return P, err
```

Q3.3

```
def findM2(F, pts1, pts2, intrinsics, filename="q3_3.npz"):
    """
    Q2.2: Function to find camera2's projective matrix given correspondences
        Input:  F, the pre-computed fundamental matrix
                pts1, the Nx2 matrix with the 2D image coordinates per row
                pts2, the Nx2 matrix with the 2D image coordinates per row
                intrinsics, the intrinsics of the cameras, load from the .npz
file
                filename, the filename to store results
        Output: [M2, C2, P] the computed M2 (3x4) camera projective matrix, C2
(3x4) K2 * M2, and the 3D points P (Nx3)

    ***
    Hints:
    (1) Loop through the 'M2s' and use triangulate to calculate the 3D points
and projection error. Keep track
        of the projection error through best_error and retain the best one.
    (2) Remember to take a look at camera2 to see how to correctly reterive
the M2 matrix from 'M2s'.

    """
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    E = essentialMatrix(F, K1, K2)
    M1 = np.hstack((np.identity(3), np.zeros(3)[:, np.newaxis]))
    C1 = K1.dot(M1)

    M2s = camera2(E)
    least_error = np.inf
    best_M2 = np.zeros((3, 4))
    best_C2 = np.zeros((3, 4))
    best_P = np.zeros((pts1.shape[0], 3))

    for i in range(M2s.shape[2]):
        M2 = M2s[:, :, i]
        C2 = K2.dot(M2)
        P, err = triangulate(C1, pts1, C2, pts2)
        if np.all(P[:, -1] > 0):
            if err < least_error:
                least_error = err
                best_M2 = M2
                best_C2 = C2
                best_P = P

    return best_M2, best_C2, best_P
```

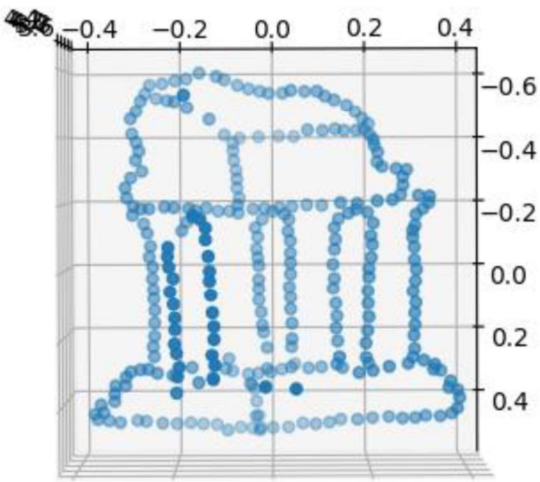
Q4.1

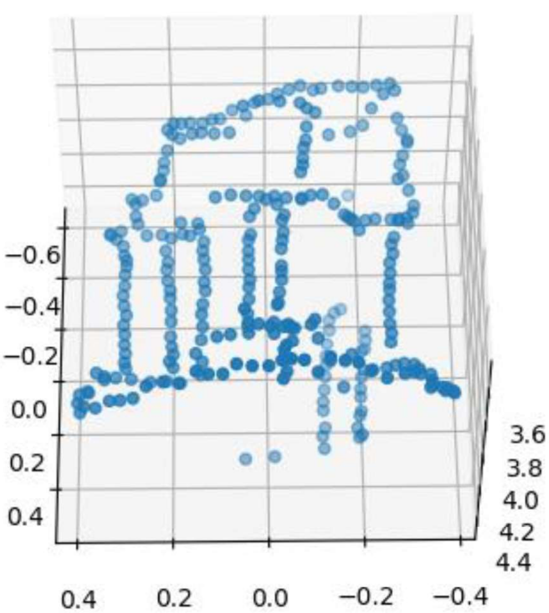


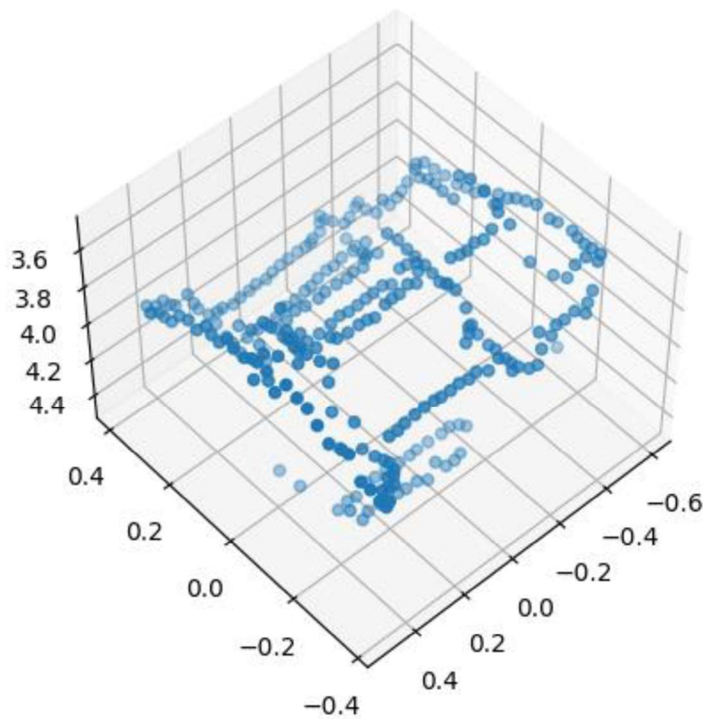

```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    line2 = F @ np.array([x1, y1, 1])
    window = 50
    bestSimilarity = float('inf')
    bestX, bestY = None, None

    def gaussian_kernel(size):
        k = (size-1)/2
        sigma = 1
        x, y = np.mgrid[-k:k+1, -k:k+1]
        kernel = np.exp(-(x**2 + y**2)/(2*sigma**2))
        return kernel/np.sum(kernel)

    guassian = gaussian_kernel(2*window+1)
    im1Convolve = guassian @ im1[y1-window:y1+window+1, x1-window:x1+window+1]

    for j in range(y1 - window, y1 + window + 1):
        x2 = int(-(line2[1]*j + line2[2])/line2[0])

        if x2 > window and x2 < im2.shape[1]-window and j > window and j <
im2.shape[0]-window:
            im2Convolve = guassian @ im2[j-window:j+window+1, x2-
window:x2+window+1]
            similarity = np.linalg.norm(im1Convolve - im2Convolve)

            if similarity < bestSimilarity:
                bestSimilarity = similarity
                bestX = x2
                bestY = j

    return bestX, bestY
```

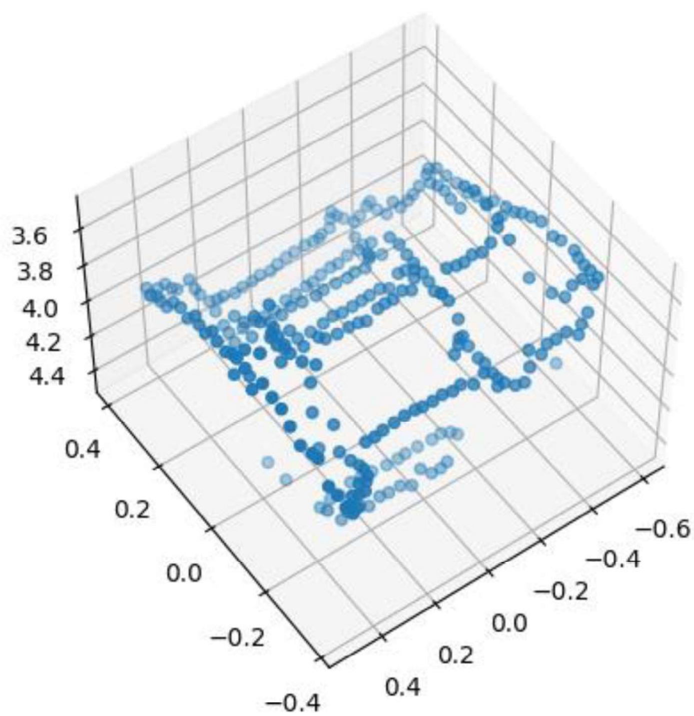
Q4.2

```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    x2s = []
    y2s = []

    for x1, y1 in temple_pts1:
        x2, y2 = epipolarCorrespondence(im1, im2, F, x1, y1)
        x2s.append(x2), y2s.append(y2)

    temple_pts2 = np.array([x2s, y2s], dtype="uint16").T
    M2, C2, P = findM2(F, temple_pts1, temple_pts2, intrinsics)
    return M2, C2, P
```




3.6
3.8
4.0
4.2
4.4
0.4
0.2
0.0
−0.2
−0.4
0.4
0.2
0.0
−0.2
−0.4
−0.6


3.6
3.8
4.0
4.2
4.4
0.4
0.2
0.0
−0.2
−0.4
0.4
0.2
0.0
−0.2
−0.4
−0.6

Q5.1

Eight Point F: [[-1.63581063e-04 1.03731856e-03 -1.69431210e-01]

[-1.15055897e-03 3.63406578e-04 2.43382903e-01]

[ 2.79695198e-01 -3.78989032e-01 1.00000000e+00]]

RANSAC F: [[ 3.99287317e-06 2.94089785e-05 -2.13537080e-01]

[ 3.61792432e-06 2.33934605e-07 -1.83515486e-03]

[ 2.03281383e-01 -1.20474074e-03 1.00000000e+00]]

**Error Metrics:** calc_epi_error. The sum of squared distance between the corresponding points and the estimated epipolar lines.

**Decision:** If epi_error within tolerance, count as inlier

**Study:**

**tol: 5 nIters:100**

F: [[ 5.09353477e-07 1.22735145e-05 -2.05408174e-01]

[ 2.25925735e-05 -5.60710954e-07 7.51172300e-04]

[ 1.97071610e-01 -4.55891889e-03 1.00000000e+00]]

Inlier Count: 105

**tol: 10 nIters:100**

F: [[-1.32511341e-05 6.27502584e-05 -4.75742815e-01]

[ 2.15104728e-05 -1.23640512e-06 4.56630022e-03]

[ 4.65163333e-01 -1.39252943e-02 1.00000000e+00]]

Inlier Count: 109

**tol: 50 nIters:100**

F: [[-2.78935720e-06 9.52701354e-05 -1.58808981e-01]

[-6.60294856e-05 -3.33586517e-06 2.60176215e-02]

[ 1.52364476e-01 -2.87204109e-02 1.00000000e+00]]

Inlier Count: 111

**tol: 10 nIters:10**

F: [[-2.27240346e-06 1.10926345e-04 -1.93705335e-01]

[-6.98899537e-05 -3.15022425e-06 2.07521008e-02]

[ 1.83850363e-01 -2.34657110e-02 1.00000000e+00]]

Inlier Count: 104

**tol:10 nIters:100**

F: [[-1.32511341e-05 6.27502584e-05 -4.75742815e-01]

[ 2.15104728e-05 -1.23640512e-06 4.56630022e-03]

[ 4.65163333e-01 -1.39252943e-02 1.00000000e+00]]

Inlier Count: 109

**tol:10 nIters:150**

F: [[-6.88237316e-06 2.00193964e-05 -3.69273679e-01]

[ 4.75430258e-05 -3.61279195e-07 -3.78966588e-03]

[ 3.58881604e-01 -4.32562777e-03 1.00000000e+00]]

Inlier Count: 110

**Conclusion:**

1. Decreasing tol led to decrease in inlier count. Because the number of points outside the tol increases.
2. Increasing nIters led to increase in inlier count as Ransac was able to find better solutions

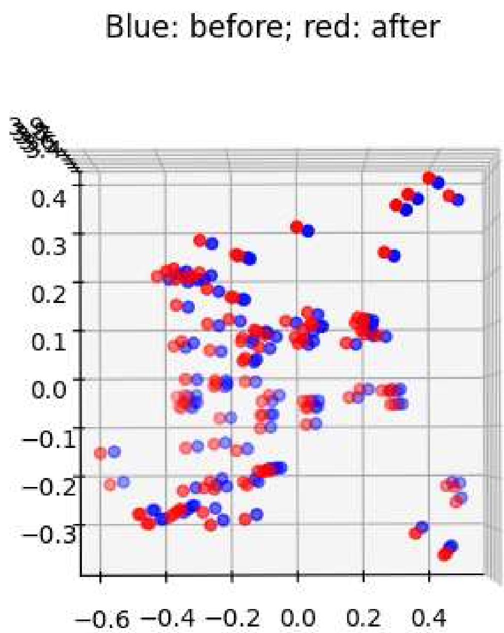
```
def ransacF(noisy_pts1, noisy_pts2, M, nIters=100, tol=10):
    best_inliers = None
    bestF = None
    maxCount = 0

    homogenous_pts1 = toHomogenous(noisy_pts1)
    homogenous_pts2 = toHomogenous(noisy_pts2)

    for i in range(int(nIters)):
        randomIndices = np.array(random.sample(range(len(noisy_pts1)), 10))

        selected_pts1 = noisy_pts1[randomIndices]
        selected_pts2 = noisy_pts2[randomIndices]
        F = eightpoint(selected_pts1, selected_pts2, M)

        epi_error = calc_epi_error(homogenous_pts1, homogenous_pts2, F)

        inliers = epi_error < tol
        inliers_count = np.count_nonzero(inliers)

        if inliers_count > maxCount:
            bestF = F
            maxCount = inliers_count
            best_inliers = inliers

    print('Inliers percentage: {:.2f}%'.format(maxCount / len(noisy_pts1) *
100))
    return bestF, best_inliers
```

Q5.2

```
def rodrigues(r):
    theta = np.linalg.norm(r)

    u = r/theta
    u_x = np.array([[0, -u[2], u[1]],
                    [u[2], 0, -u[0]],
                    [-u[1], u[0], 0]])
    u = u.reshape(-1, 1)
    return np.eye(3)*np.cos(theta) + np.sin(theta)*u_x  + (1-
np.cos(theta))*(u@u.T)
```

```
def invRodrigues(R):
    A = (R - R.T)/2
    rho = np.array([A[2, 1], A[0, 2], A[1, 0]]).T
    s = np.linalg.norm(rho)
    c = (np.trace(R) - 1)/2
    theta = np.arctan2(s, c)

    def S(r):
        if np.linalg.norm(r) == np.pi and ((r[0]==0 and r[1]==0 and r[2]<0) or
(r[0]==0 and r[1]<0) or (r[0]<0)):
            return -r
        else:
            return r

    if s == 0 and c == 1:
        return np.zeros(3)
    elif s == 0 and c == -1:
        v = np.diag(R) + 1
        u = v/np.linalg.norm(v)
        r = S(u*np.pi)
    elif np.sin(theta) != 0:
        u = rho/s
        r = u*theta

    return r
```

Q5.3




Initial reprojection error: 34622.60413453882

Optimised reprojection error: 10.88721491950835

```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    r2 = x[3*len(p1):3*len(p1)+3]
    t2 = x[3*len(p1)+3:]
    x = x[:3*len(p1)].reshape(len(p1), 3)

    P = np.hstack((x, np.ones((len(p1),1))))
    M2 = np.hstack((rodrigues(r2), t2.reshape(-1, 1)))

    C1 = K1 @ M1
    C2 = K2 @ M2

    p1_hat = C1 @ P.T
    p1_hat = p1_hat/p1_hat[-1]
    p2_hat = C2 @ P.T
    p2_hat = p2_hat/p2_hat[-1]

    residuals = np.concatenate([(p1 - p1_hat[:2].T).reshape([-1]), (p2 -
p2_hat[:2].T).reshape([-1])])
    return residuals
```

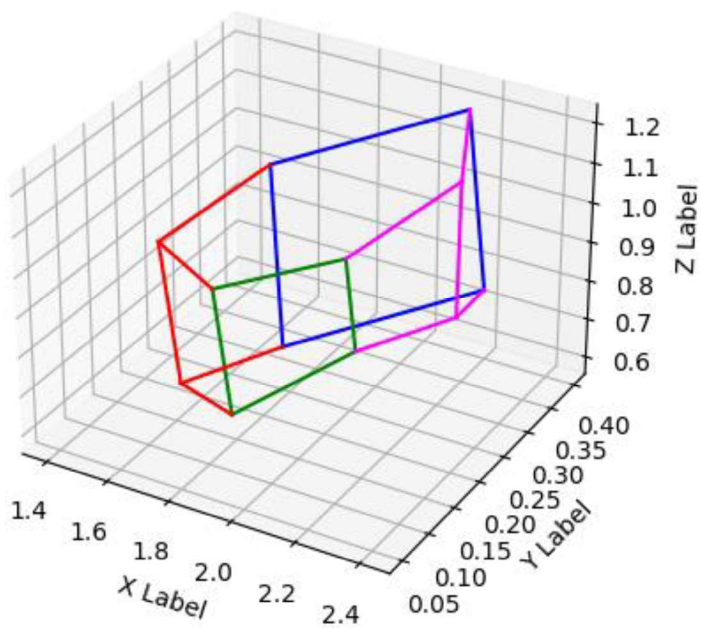
```
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    r2 = invRodrigues(M2_init[:, :3])
    t2 = M2_init[:, 3]
    obj_start = np.concatenate((P_init.flatten(), r2.flatten(), t2))

    print('Initial reprojection error: ', np.sum(rodriguesResidual(K1, M1, p1,
K2, p2, obj_start)**2))
    obj_end = opt.minimize(lambda x: np.sum(rodriguesResidual(K1, M1, p1, K2,
p2, x)**2), obj_start).x
    print('Optimised reprojection error: ', np.sum(rodriguesResidual(K1, M1,
p1, K2, p2, obj_end)**2))

    P = obj_end[:3*len(p1)].reshape(len(p1), 3)
    r2 = obj_end[3*len(p1):3*len(p1)+3]
    t2 = obj_end[3*len(p1)+3:]

    M2 = np.concatenate((rodrigues(r2), t2.reshape(-1, 1)), axis=1)
    return M2, P, obj_start, obj_end
```

Q6.1

**Method:**

Used the triangulate function for the highest confidence points. Also, the highest confidence should cross the threshold. If the conditions are satisfied, the corresponding 3D point and error is calculated.

```
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres=100):
    for i in range(pts1.shape[0]):
        confidence1 = pts1[i, 2]
        confidence2 = pts2[i, 2]
        confidence3 = pts3[i, 2]

        if(confidence1 < confidence2 and confidence1 < confidence3 and
confidence1 > Thres):
            p, e = triangulate(C2, np.array([pts2[i, :2]]), C3,
np.array([pts3[i, :2]]))
        elif(confidence2 < confidence1 and confidence2 < confidence3 and
confidence2 > Thres):
            p, e = triangulate(C3, np.array([pts3[i, :2]]), C1,
np.array([pts1[i, :2]]))
        elif(confidence3 > Thres):
            p, e = triangulate(C1, np.array([pts1[i, :2]]), C2,
np.array([pts2[i, :2]]))
        else:
            continue

        if i == 0:
            P = p
            err = e
        else:
            P = np.vstack((P, p))
            err = np.hstack((err, e))
    return P, err
```




Error: 46.21689976115511

Q6.2

```
def plot_3d_keypoint_video(pts_3d_video):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    ax.set_xlabel("X Label")
    ax.set_ylabel("Y Label")
    ax.set_zlabel("Z Label")

    for i in range(len(pts_3d_video)):
        pts_3d = pts_3d_video[i]
        for j in range(len(connections_3d)):
            index0, index1 = connections_3d[j]
            xline = [pts_3d[index0, 0], pts_3d[index1, 0]]
            yline = [pts_3d[index0, 1], pts_3d[index1, 1]]
            zline = [pts_3d[index0, 2], pts_3d[index1, 2]]
            ax.plot(xline, yline, zline, color=colors[j])
        np.set_printoptions(threshold=1e6, suppress=True)

    plt.show()
```