Q1.1

For fundamental matrix:

X2TFX1 = 0

And X1=X2=(0, 0), so:

[0 0 1][[f11 f12 f13], [f21 f22 f23], [f31 f32 f33]][0 0 1]T = 0

[f31 f32 f33] [0 0 1]T = 0

Thus, f33 = 0

Q1.2

Since the pure translation is parallel to x-axis, ty=tz=0, so cross product matrix can be:

t_x = [[0 0 0], [0 0 -tx], [0 tx 0]]

Rotation matrix:

R = [[1 0 0], [0 1 0], [0 0 1]]

Essential matrix = t_x R = [[0 0 0], [0 0 -tx], [0 tx 0]]

Epipolar lines:

$l_1^T = X_2^T E$ = [x2 y2 1]E = [0 tx –tx*y2]

$l_2^T = X_1^T E^T$ = [x1 y1 1]E = [0 -tx tx*y1]

Q1.3

Time stamp 1: $w1 = R1w+t1$

Time stamp 2: $w2 = R2w+t2$

Then use $w=R1^{-1}(w1-t1)$ to express w2:

$w2 = R2R1^{-1}w1- R2R1^{-1}t1+t2$

Thus,

$R_{rel} = R2R1^{-1}$

$t_{rel} = t2- R2R1^{-1}t1$

$E = t_{rel}R_{rel}$

$F = [K^{-1}]^{T}t_{rel}R_{rel}K^{-1}$

Q1.4

Suppose p is a point of the object in 3-D space and x1, x2 are the 2D coordinates of point p in different images. Because all points on the object are of equal distance to the mirror, point p can represent other points for the distance to mirror.

$X_2^T F X_1 = 0$ 1

$M_2 = TM_1$ 2

The relationship of two cameras is reflection: $T^T T=I$

Thus, $X_1 = KM_1 p$ 3

Combine 2and 3: $X_2 = KTM_1 p$ 4

We can know from 1 that:

$X_2^T F X_1 + X_1^T F^T X_2 = 0$

Substitute X_1 and X_2 by 3, 4:

$p^T M_1^T T^T K^T F K M_1 p + p^T M_1^T K^T F^T K T M_1 p = 0$

Remove common terms: $p^T M_1^T$ and $M_1 p$:

$T^T K^T F K + K^T F^T K T = 0$, and we know $T^T T=I$:

$K^T (F+F^T) K =0$

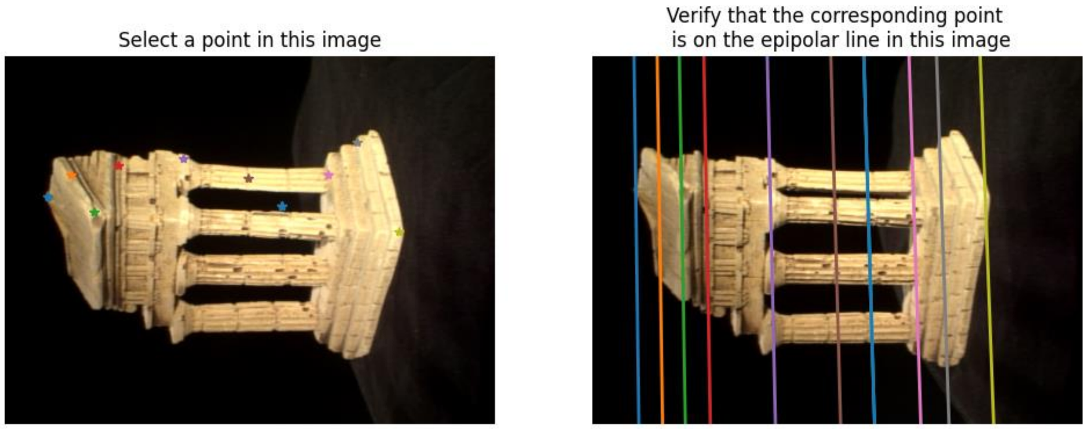
In conclusion, $F = - F^T$, so the fundamental matrix F is skew-symmetric.

Q2.1

F = [[9.80213863e-10 -1.32271663e-07 1.12586847e-03]

 [-5.72416248e-08 2.97011941e-09 -1.17899320e-05]

 [-1.08270296e-03 3.05098538e-05 -4.46974798e-03]]




```
def eightpoint(pts1, pts2, M):
    # Normalize the input pts1 and pts2 using the matrix T.
    pts1, pts2 = pts1/M, pts2/M
    x1, y1 = pts1[:, 0], pts1[:, 1]
    x2, y2 = pts2[:, 0], pts2[:, 1]

    T = np.array([[1 / M, 0, 0], [0, 1 / M, 0], [0, 0, 1]])
    A = np.array((x2 * x1, x2 * y1, x2, y2 * x1, y2 * y1, y2, x1, y1,
np.ones(pts1.shape[0]))).T

    # Solve for the least square solution using SVD.
    u, s, vh = np.linalg.svd(A)
    F = vh[-1].reshape(3, 3)

    # Use the function `_singularize` (provided) to enforce the
singularity condition.
    F = _singularize(F)

    # Use the function `refineF` (provided) to refine the computed
fundamental matrix.
    F = refineF(F, pts1, pts2)
    # Unscale the fundamental matrix
    F = T.T @ F @ T

    return F
```

Q3.1

```
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    E = K2.T @ F @ K1
    return E
```

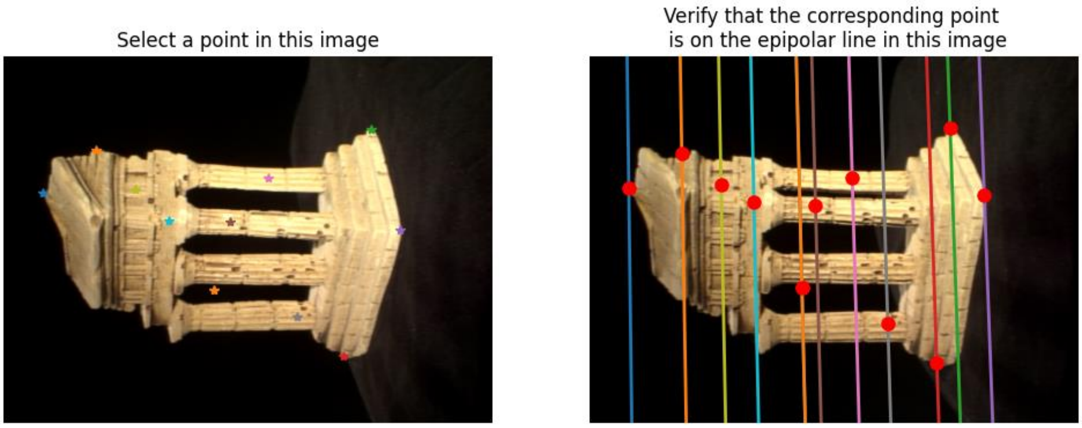
Q3.2

```
def triangulate(C1, pts1, C2, pts2):
    # For every input point, form A using the corresponding points
from pts1 & pts2 and C1 & C2
    x1, y1 = pts1[:, 0], pts1[:, 1]
    x2, y2 = pts2[:, 0], pts2[:, 1]
    A1 = np.array((C1[0, 0]-C1[2, 0]*x1, C1[0, 1]-C1[2, 1]*x1, C1[0,
2]-C1[2, 2]*x1, C1[0, 3]-C1[2, 3]*x1)).T
    A2 = np.array((C1[1, 0] - C1[2, 0] * y1, C1[1, 1] - C1[2, 1] *
y1, C1[1, 2] - C1[2, 2] * y1, C1[1, 3] - C1[2, 3] * y1)).T
    A3 = np.array((C2[0, 0] - C2[2, 0] * x2, C2[0, 1] - C2[2, 1] *
x2, C2[0, 2] - C2[2, 2] * x2, C2[0, 3] - C2[2, 3] * x2)).T
    A4 = np.array((C2[1, 0] - C2[2, 0] * y2, C2[1, 1] - C2[2, 1] *
y2, C2[1, 2] - C2[2, 2] * y2, C2[1, 3] - C2[2, 3] * y2)).T

    w = np.zeros((pts1.shape[0], 3))    # wi = [xi, yi, zi].T, Nx3
    for i in range(pts1.shape[0]):
        A = np.vstack((A1[i, :], A2[i, :], A3[i, :], A4[i, :]))     #
4x4
        u, s, vh = np.linalg.svd(A)
        p = vh[-1, :]
        w[i, :] = [p[0]/p[-1], p[1]/p[-1], p[2]/p[-1]]

    # convert w from homogeneous coordinates to non-homogeneous ones,
4x1
    w_homo = np.hstack((w, np.ones((pts1.shape[0], 1))))
    err = 0
    for i in range(pts1.shape[0]):
        proj1 = C1 @ w_homo[i, :].T
        proj2 = C2 @ w_homo[i, :].T
        proj1 = (proj1[:2]/proj1[-1]).T
        proj2 = (proj2[:2]/proj2[-1]).T
        # Calculate the reprojection error
        err += np.sum((pts1[i]-proj1)**2+(pts2[i]-proj2)**2)

    return w, err
```

Q3.3

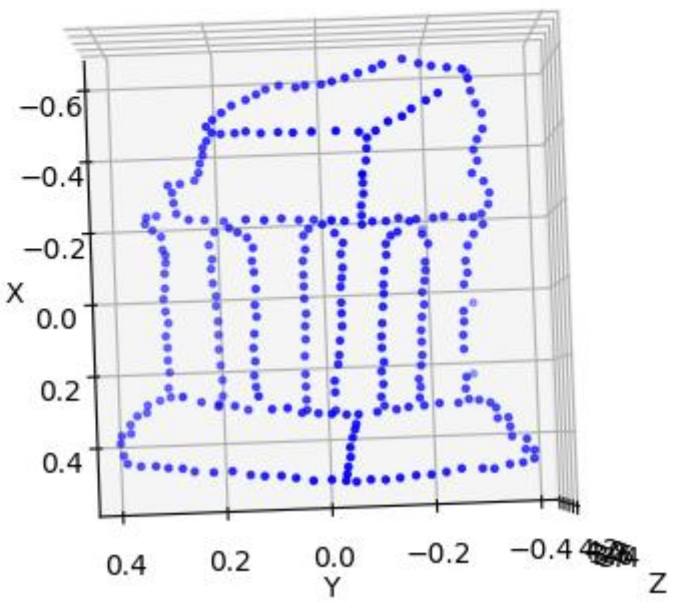
```
def findM2(F, pts1, pts2, intrinsics, filename='../data/q3_3.npz'):

    K1, K2 = intrinsics['K1'], intrinsics['K2']
    E = essentialMatrix(F, K1, K2)
    M2s = camera2(E)
    M1 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
    C1 = K1 @ M1

    err = float('inf')
    M2, C2, P = None, None, None
    for i in range(M2s.shape[2]):
        cur_C2 = K2 @ M2s[:, :, i]
        cur_w, cur_err = triangulate(C1, pts1, cur_C2, pts2)

        if cur_err < err and np.min(cur_w[:, 2]) >= 0:     # valid
points with all coordinates>=0
            err = cur_err
            M2 = M2s[:, :, i]
            C2 = cur_C2
            P = cur_w

    if filename=='../data/q4_2.npz':
        np.savez(filename, F=F, M1=M1, M2=M2, C1=C1, C2=C2)
    elif filename == '../data/q3_3.npz':
        np.savez(filename, M2=M2, C2=C2, P=P)
    return M2, C2, P
```

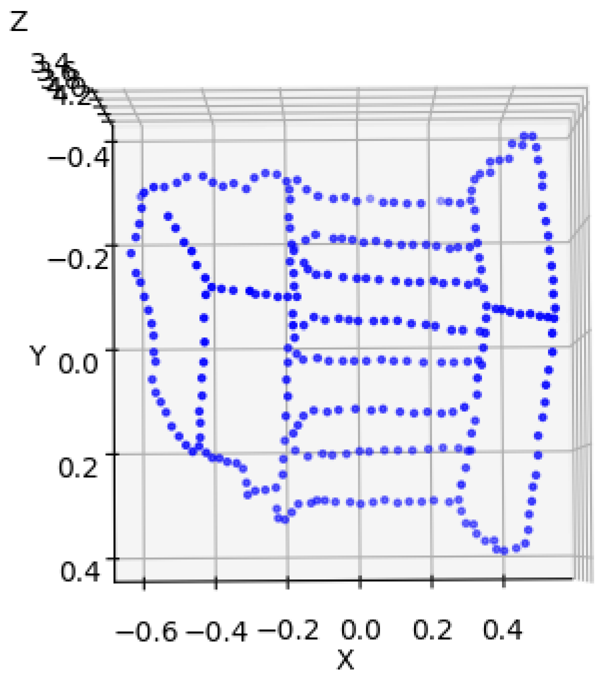
Q4.1

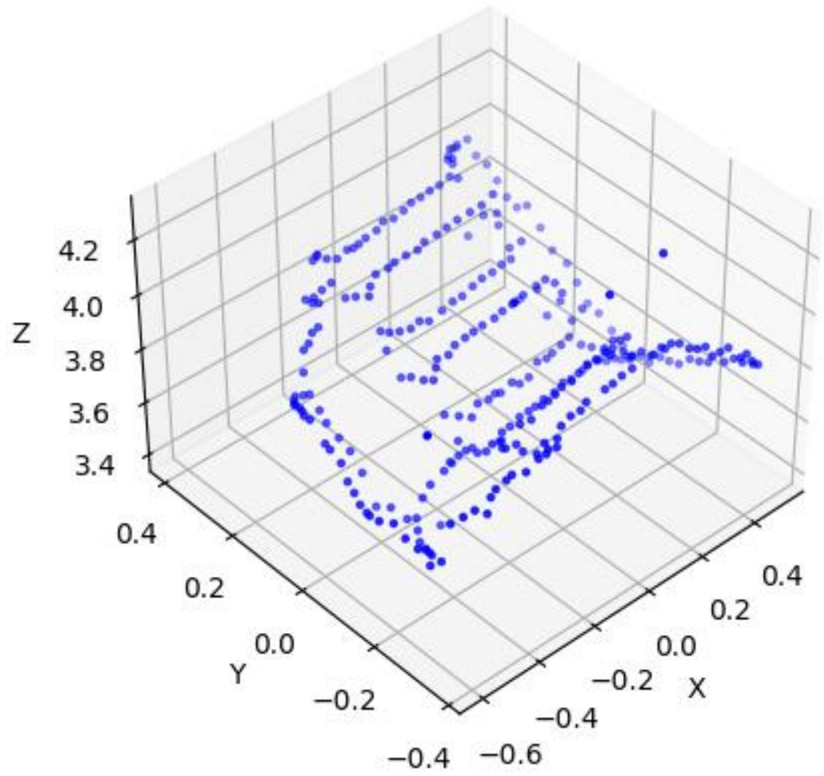



```
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Parameters about window
    win_size = 11
    center = win_size//2
    search = 50      # the search range over the set of pixels that lie along the epipolar line

    x1, y1 = int(x1), int(y1)
    epip_line = F @ np.array([x1, y1, 1])   # epipolar line

    # get window1 around the pixel in image1
    win1 = im1[y1 - center: y1 + center + 1, x1 - center: x1 + center + 1]
    # get the set of points to search in image2
    y2_range = np.array(range(y1 - search, y1 + search))
    x2_range = np.round(-(epip_line[1] * y2_range + epip_line[2]) / epip_line[0]).astype(int)
    valid = (x2_range >= center) & (x2_range < im2.shape[1] - center) & (y2_range >= center) & (y2_range < im2.shape[0] - center)
    x2_range, y2_range = x2_range[valid], y2_range[valid]

    err = float('inf')
    x2, y2 = None, None
    for i in range(len(x2_range)):
        win2 = im2[y2_range[i]-center:y2_range[i]+center+1, x2_range[i]-center:x2_range[i]+center+1]
        cur_err = np.sum(gaussian_filter(np.linalg.norm((win1-win2)), sigma=3))
        if cur_err<err:
            err = cur_err
            x2, y2 = x2_range[i], y2_range[i
    return x2, y2
```

Q4.2


X
Y
Z
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.4
0.2
0.0
−0.2
−0.4


Z
Y
X
−0.4
−0.2
0.0
0.2
0.4
−0.6
−0.4
−0.2
0.0
0.2
0.4




```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):

    # ----- TODO -----
    # get [x2, y2]
    x1, y1 = temple_pts1['x1'], temple_pts1['y1']
    points1, points2 = [], []
    for i in range(x1.shape[0]):
        tmp_x2, tmp_y2 = epipolarCorrespondence(im1, im2, F, x1[i], 
y1[i])
        points1.append([x1[i][0], y1[i][0]])
        points2.append([tmp_x2, tmp_y2])
    points1, points2 = np.array(points1), np.array(points2)

    M2, C2, P = findM2(F, points1, points2, intrinsics)

    return P
```

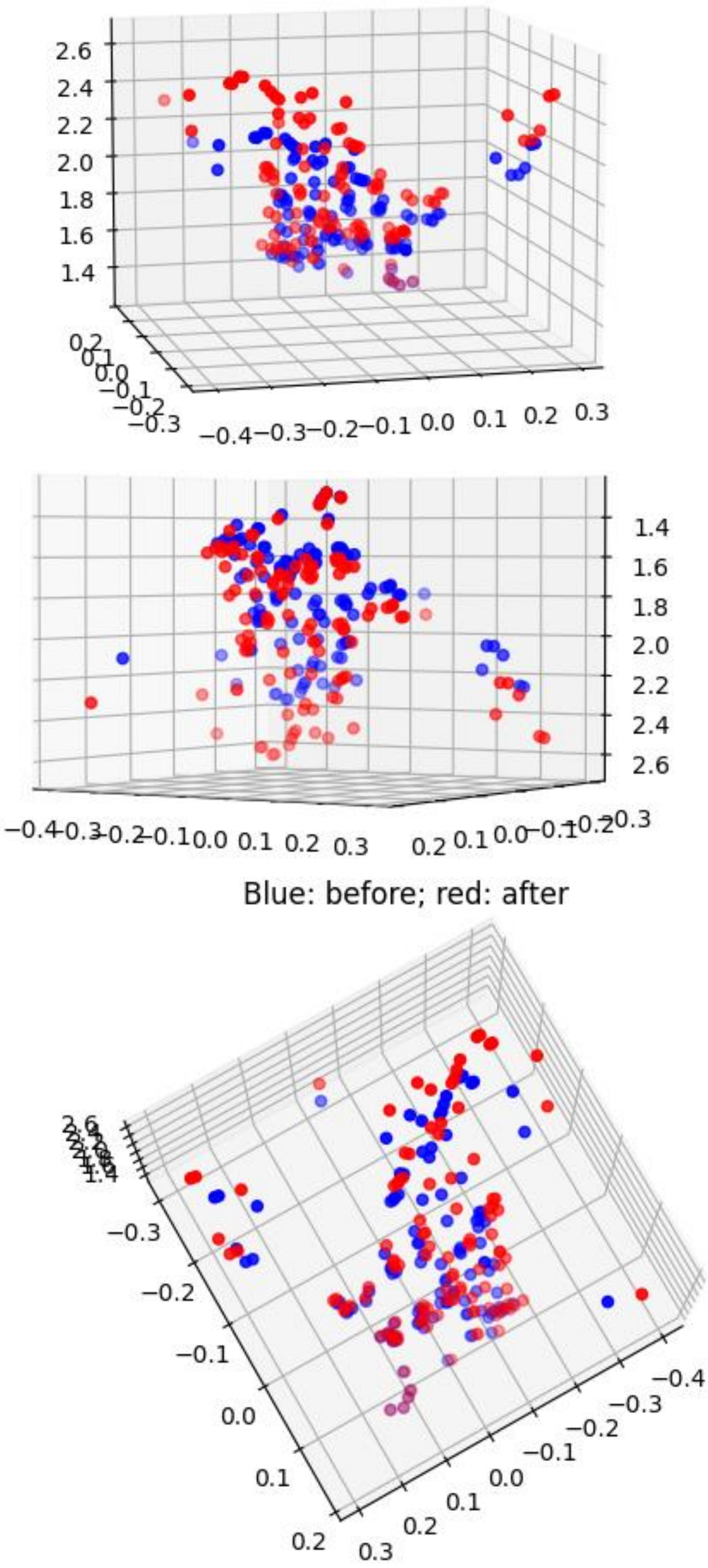
Q5.1

```
def ransacF(pts1, pts2, M, nIters=1000, tol=10):
    # homography matrix
    pts1_hom = np.vstack((pts1.T, np.ones([1, pts1.shape[0]])))
    pts2_hom = np.vstack((pts2.T, np.ones([1, pts1.shape[0]])))

    max_inliers = 0
    inliers, best_F = None, None
    for i in range(nIters):
        # randomly choose 8 points
        rand_index = np.random.choice(pts1.shape[0], 8)
        rand_p1, rand_p2 = pts1[rand_index, :], pts2[rand_index, :]

        # get predicted points
        F = eightpoint(rand_p1, rand_p2, M)
        pred_p2_hom = F @ pts1_hom
        pred_p2 = pred_p2_hom/np.linalg.norm(pred_p2_hom[:2, :],
axis=0)

        # calculate error
        err = abs(np.sum(pts2_hom*pred_p2, axis=0))
        cur_inliers = (err<tol).T
        if cur_inliers[cur_inliers].shape[0] > max_inliers:
            max_inliers = cur_inliers[cur_inliers].shape[0]
            best_F = F
            inliers = cur_inliers

    return best_F, inliers
```

Q5.2

```
def rodrigues(r):
    theta = np.linalg.norm(r)
    if theta == 0:
        return np.eye(3)
    else:
        # R = I*cos(θ) + (1 - cos θ)*u@u.T + ux*sin(θ)
        u = r/theta
        ux = np.array([[0, -u[2], u[1]], [u[2], 0, -u[0]], [-u[1],
u[0], 0]])
        R = np.eye(3)*np.cos(theta)+(1-np.cos(theta))*(u @
u.T)+ux*np.sin(theta)
        return R

def invRodrigues(R):
    A = (R-R.T)/2
    ro = np.array([[A[2][1], A[0][2], A[1][0]]]).T
    s = np.linalg.norm(ro)
    c = (R[0][0]+R[1][1]+R[2][2]-1)/2

    r = None
    if s == 0. and c == 1.:
        r = np.zeros((3, 1))
    elif s == 0. and c == -1.:
        # v = a nonzero column of R + I
        R_I = R+np.eye(3)
        for i in range(3):
            if np.count_nonzero(R_I[:, i]) > 0:
                v = R_I[:, i]
                break

        u = v/np.linalg.norm(v)
        tmp_r = u*np.pi
        if (np.linalg.norm(tmp_r) == np.pi) and ((tmp_r[0][0] ==
tmp_r[1][0] == 0. and tmp_r[2][0] < 0) or (tmp_r[0][0] == 0. and
tmp_r[1][0] < 0.) or (tmp_r[0][0] < 0.)):
            r = -tmp_r
        else:
            r = tmp_r
    else:
        u = ro/s
        theta = np.arctan2(s, c)
        r = u*theta
    return r
```

Q5.3



The reprojection error with the initial M2 and w: 18569.213306384438

The reprojection error with the optimized matrices: 1

```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    P = x[:-6].reshape(-1, 3)
    r2 = x[-6:-3].reshape(3, 1)
    t2 = x[-3:].reshape(3, 1)

    R2 = rodrigues(r2)
    M2 = np.hstack((R2, t2))
    P_hom = np.vstack((P.T, np.ones((1, P.shape[0]))))

    x1_hom = np.dot(C1, P_hom)
    x2_hom = np.dot(C2, P_hom)
    p1_hat = np.zeros((2, P_hom.shape[1]))
    p2_hat = np.zeros((2, P_hom.shape[1]))

    p1_hat[:2, :] = (x1_hom[:2, :] / x1_hom[2, :])
    p2_hat[:2, :] = (x2_hom[:2, :] / x2_hom[2, :])
    p1_hat = p1_hat.T
    p2_hat = p2_hat.T
    residuals = np.concatenate([(p1 - p1_hat).reshape([-1]), (p2 -
p2_hat).reshape([-1])])

    return residuals
```

```python
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    obj_start = obj_end = 0
    # ----- TODO -----
    # YOUR CODE HERE
    R2_init = M2_init[:, :3]
    t2_init = M2_init[:, 3]
    r2_init = invRodrigues(R2_init)
    x_init = np.concatenate([P_init.flatten(), r2_init.flatten(),
t2_init.flatten()])
    obj_start = np.sum(rodriguesResidual(K1, M1, p1, K2, p2,
x_init)**2)

    func = lambda x: (rodriguesResidual(K1, M1, p1, K2, p2, x))
    x_optimized, obj_end = leastsq(func, x_init)
    P2 = x_optimized[:-6].reshape(-1, 3)
    r2 = x_optimized[-6:-3].reshape(3, 1)
    t2 = x_optimized[-3:].reshape(3, 1)
    R2 = rodrigues(r2)
    M2 = np.hstack((R2, t2))

    return M2, P2, obj_start, obj_end
```