Computer Vision HW4

Jiyoon Park (jiyoonp)

Part I

Theory

Q1.1

We know that

$$x_2^T F x_1 = 0$$

$$x_2^T \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} x_1 = 0$$

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

$$\begin{bmatrix} f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0$$

$$\begin{bmatrix} f_{33} \end{bmatrix} = 0$$

So, $\mathrm{F}_{33} = 0$

Q1.2

Let

$$t = \begin{bmatrix} t_1 \\ 0 \\ 0 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$E = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix}$$

$$x_1 = \begin{bmatrix} x_{11} \\ x_{12} \\ 1 \end{bmatrix}$$

$$x_2 = \begin{bmatrix} x_{21} \\ x_{22} \\ 1 \end{bmatrix}$$

Then

$$l_1^T = \begin{bmatrix} x_{21} \\ x_{22} \\ 1 \end{bmatrix} E = \begin{bmatrix} 0 & t_1 & x_{22}t_1 \end{bmatrix}$$

$$l_2^T = \begin{bmatrix} x_{11} \\ x_{12} \\ 1 \end{bmatrix} E = \begin{bmatrix} 0 & -t_1 & x_{12}t_1 \end{bmatrix}$$

So, the epipolar lines are

$$t_1 y_1 - x_{22} t_1 = 0$$

$$-t_1 y_1 + x_{12} t_1 = 0$$

it is clear that the lines do not have an x component in it which means they are parallel to the x axis.

Q1.3

Let P be the 3D coordinate and

$$x_1, x_2$$

the homogenous points on the image.

$$x_1 = K(R_1P + t_1)$$
$$P = R_1^{-1}(K^{-1}x_1 - t_1)$$

$$\begin{aligned} x_2 = K(R_2P + t_2) &= K(R_2(R_1^{-1}(K^{-1}x_1 - t_1)) + t_2) \\ &= K(R_2(R_1^{-1}K^{-1}x_1 - R_1^{-1}t_1) + t_2) \\ &= K(R_2R_1^{-1}K^{-1}x_1 - R_2R_1^{-1}t_1 + t_2) \\ &= KR_2R_1^{-1}K^{-1}x_1 - KR_2R_1^{-1}t_1 + Kt_2 \end{aligned}$$
$$R_{rel} = KR_2R_1^{-1}K^{-1}$$
$$t_{rel} = -KR_2R_1^{-1}t_1 + Kt_2$$
$$E = t_{rel} \times R_{rel}$$

$$F = K^{-T}EK^{-1} = K^{-T}t_{rel} \times R_{rel}K^{-1}$$

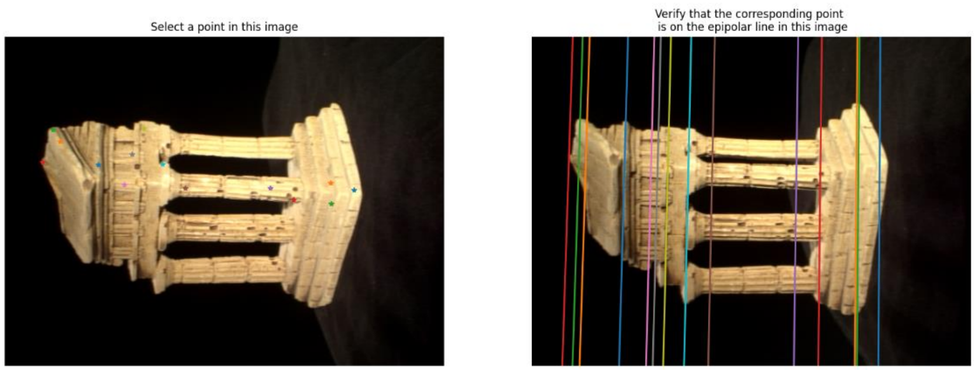
Q1.4

Part II

Question 2

Q2.1

$$F = \begin{bmatrix} -2.19299581e-07 & 2.95926445e-05 & -2.51886343e-01 \\ 1.28064547e-05 & -6.64493709e-07 & 2.63771740e-03 \\ 2.42229086e-01 & -6.82585550e-03 & 1.00000000e+00 \end{bmatrix}$$

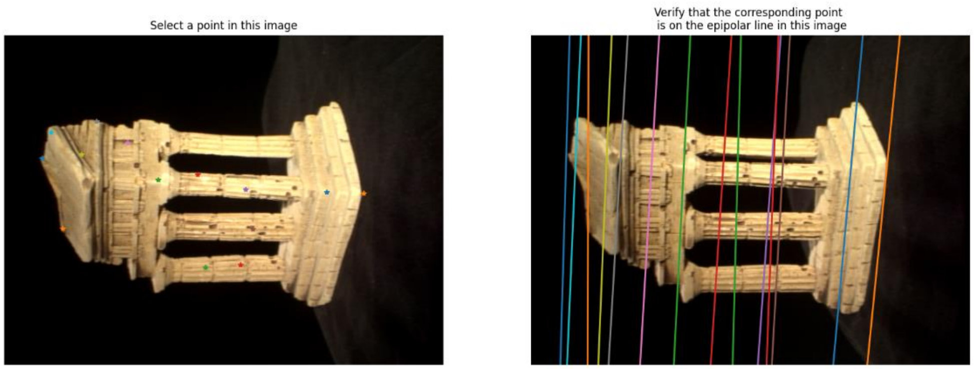



```
def eightpoint(pts1, pts2, M):
    N = pts1.shape[0]

    T = np.diag([1 / M, 1 / M, 1])

    npts1 = pts1 / M
    npts2 = pts2 / M

    x1 = npts1
    x2 = npts2

    A = [
        [x2[i, 0] * x1[i, 0], x2[i, 0] * x1[i, 1], x2[i, 0], x2[i, 1] * x1[i,
0], x2[i, 1] * x1[i, 1], x2[i, 1],
         x1[i, 0], x1[i, 1], 1]
        for i in range(N)
    ]

    A = np.array(A).reshape(N, -1)
    u, s, vh = np.linalg.svd(A)
    F = vh[-1, :].reshape(3, 3)
```

```
    F = _singularize(F)
    F = refineF(F, npts1, npts2)

    F = T.T @ F @ T
    F = F / F[-1, -1]

    np.savez('results/q2_1.npz', F=F, M=M)

    return F
```

Q2.2

```
Final F:
 [[-6.24506676e-07 -7.57931706e-06 -1.36008881e-01]
 [ 3.23962377e-05  7.73246862e-06 -1.06791029e-02]
 [ 1.30557801e-01  4.07163434e-03  1.00000000e+00]]
```




```
def sevenpoint(pts1, pts2, M):
    Farray = []
    # ----- TODO -----
    N = pts1.shape[0]
    T = np.diag([1 / M, 1 / M, 1])

    npts1 = pts1 / M
    npts2 = pts2 / M

    x1 = npts1
    x2 = npts2
    A = [
        [x1[i, 0] * x2[i, 0], x1[i, 0] * x2[i, 1], x1[i, 0], x1[i, 1] * x2[i, 0],
x1[i, 1] * x2[i, 1], x1[i, 1],
         x2[i, 0], x2[i, 1], 1]
        for i in range(N)
    ]
    A = np.array(A).reshape(N, -1)
    u, s, vh = np.linalg.svd(A)

    fvec1 = vh[-2, :].reshape(3, 3)
    fvec2 = vh[-1, :].reshape(3, 3)

    Fmat = np.zeros((3, 3, 2))
    Fmat[:, :, 0] = fvec1.T
    Fmat[:, :, 1] = fvec2.T

    D = np.zeros((2, 2, 2))
    for i1 in range(2):
        for i2 in range(2):
```

```
            for i3 in range(2):
                Dtmp = np.zeros((3, 3))
                Dtmp[:, 0] = Fmat[:, :, i1][:, 0]
                Dtmp[:, 1] = Fmat[:, :, i2][:, 1]
                Dtmp[:, 2] = Fmat[:, :, i3][:, 0]
                D[i1, i2, i3] = np.linalg.det(Dtmp)
    coefficients = np.zeros(4)
    coefficients[0] = -D[1][0][0] + D[0][1][1] + D[0][0][0] + D[1][1][0] + D[1][0][1]
- D[0][1][0] - D[0][0][1] - \
                      D[1][1][1]
    coefficients[1] = D[0][0][1] - 2 * D[0][1][1] - 2 * D[1][0][1] + D[1][0][0] - 2 *
D[1][1][0] + D[0][1][0] + 3 * \
                      D[1][1][1]
    coefficients[2] = D[1][1][0] + D[0][1][1] + D[1][0][1] - 3 * D[1][1][1]
    coefficients[3] = D[1][1][1]

    roots = np.polynomial.polynomial.polyroots(coefficients)

    for i in range(len(roots)):
        Ftmp = roots[i] * Fmat[:, :, 0] + (1 - roots[i]) * Fmat[:, :, 1]
        F = T.T @ Ftmp @ T
        Farray.append(F)
    # save
    np.savez('results/q2_2.npz', F=F, M=M, pts1=pts1, pts2=pts2)

    return Farray
```

Question 3

Q3.1

```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    E = K2.T @ F @ K1
    E = E / E[-1, -1]
    # save
    # np.savez('results/q3_1.npz', F=F, E=E)
    return E
```

Q3.2

$$A = \begin{bmatrix} y_1 c_{13} - c_{12} \\ c_{11} - x_1 c_{13} \\ y_2 c_{23} - c_{22} \\ c_{21} - x_2 c_{23} \end{bmatrix}$$

Where

$$C_1 = \begin{bmatrix} C_{11} \\ C_{12} \\ C_{13} \end{bmatrix}$$

$$C_2 = \begin{bmatrix} C_{21} \\ C_{22} \\ C_{23} \end{bmatrix}$$

and

$$(x_1, y_1)$$

is the image projection to img1, and

$$(x_2, y_2)$$

is the projection to img 2

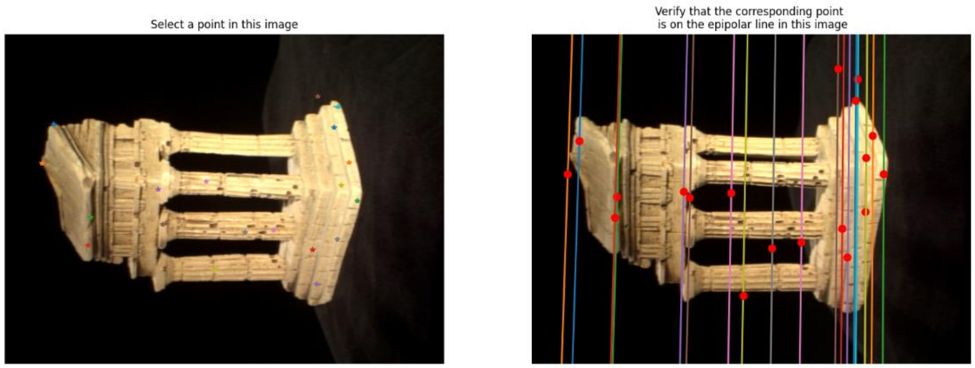
Q3.3

```
def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation

    N = pts1.shape[0]

    X = np.zeros((N, 3))
    error = 0

    Proj1_n_list = np.zeros((N, 2))
    Proj2_n_list = np.zeros((N, 2))
    for i in range(N):
        A = np.array([
            pts1[i, 1] * C1[2, :] - C1[1, :],
            C1[0, :] - pts1[i, 0] * C1[2, :],
            pts2[i, 1] * C2[2, :] - C2[1, :],
            C2[0, :] - pts2[i, 0] * C2[2, :]
        ])
        u, s, vh = np.linalg.svd(A)
        P = vh[-1, :].reshape(4, -1)
        P /= P[-1, :]
        P = P.reshape(4)
        X[i, :] = P[:3]

        pt3D = P
        Proj1 = C1 @ pt3D.T  # 3xN
        Proj1_n = (Proj1 / Proj1[2])[:2]  # 2xN
        Proj1_n_list[i, :] = Proj1_n

        Proj2 = C2 @ pt3D.T  # 3xN
        Proj2_n = (Proj2 / Proj2[2])[:2]  # 2xN
        Proj2_n_list[i, :] = Proj2_n

        e1 = (np.linalg.norm(Proj1_n - pts1[i, :])) ** 2
        e2 = (np.linalg.norm(Proj2_n - pts2[i, :])) ** 2
        # print(e1, e2)
        error += (e1 + e2)

    # err = np.sum((pts1[:, 0] - Proj1_n_list[:, 0]) ** 2 + (pts1[:, 1] -
Proj1_n_list[:, 1]) ** 2 + (
    #          pts2[:, 0] - Proj2_n_list[:, 0]) ** 2 + (pts2[:, 1] -
Proj2_n_list[:, 1]) ** 2)
    # print(error, err)
    return X, error
```

```python
def findM2(F, pts1, pts2, intrinsics, filename='q3_3.npz'):
    '''
    Q2.2: Function to find the 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)

    M2s = camera2(E)
    min_p = -10000000
    M2 = None

    M1 = np.hstack((np.identity(3), np.zeros(3)[:, np.newaxis]))
    C1 = K1.dot(M1)

    Pf = None
    C2f = None

    for i in range(M2s.shape[-1]):
        m = M2s[:, :, i]
        C2 = K2.dot(m)
        P, err = triangulate(C1, pts1, C2, pts2)

        if np.min(P[:, -2]) > min_p or np.min(P[:, -2]) > 0:
            min_p = np.min(P[:, -2])
            M2 = m
            Pf = P
            C2f = C2

    # save
    # np.savez('results/' + filename, M2=M2, C2=C2, P=Pf)

    return M2, C2f, Pf
```

Question 4

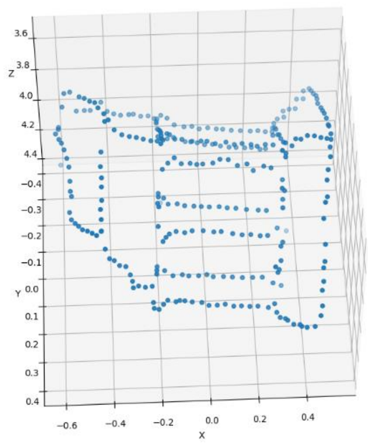
Q4.1

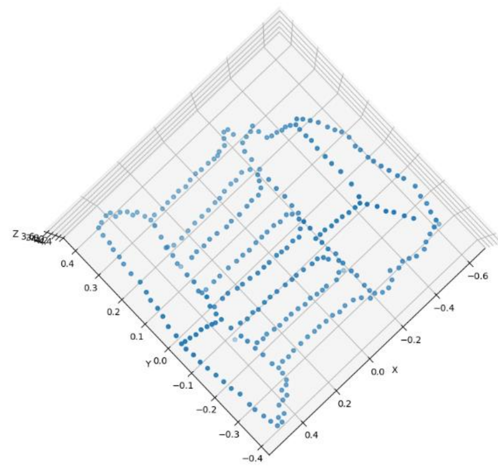



```
def epipolarCorrespondence(im1, im2, F, x1, y1):
    x1, y1 = int(x1), int(y1)

    # Replace pass by your implementation
    A = list(F @ np.array([x1, y1, 1]).T)

    [a, b, c] = [float(i) for i in A]

    w_size = 10
    search = 20

    xs = np.array([i for i in range(w_size * 2, im2.shape[0] - w_size * 2)])
    ys = -b / a * xs - c / a

    xs = [int(i) for i in xs]
    ys = [int(i) for i in ys]
    zs = zip(xs, ys)

    fzs = []
    for (zx, zy) in zs:
        if y1 - search < zx < y1 + search:
            fzs.append([zx, zy])

    img_g1 = ndimage.gaussian_filter(im1, sigma=2)
    img_g2 = ndimage.gaussian_filter(im2, sigma=2)

    # d1 = getDensity(img_g1, x1, y1, w_size)
    d1 = img_g1[y1 - w_size: y1 + w_size, x1 - w_size: x1 + w_size]

    min_d = 1000
```

```
    [x2, y2] = [0, 0]

    for (y, x) in fzs:
        if not possible(img_g2, w_size, x, y):
            continue
        d2 = img_g2[y - w_size: y + w_size, x - w_size: x + w_size]
        d = np.sqrt(np.sum((d1 - d2) ** 2))
        if d < min_d:
            min_d = d
            [x2, y2] = [x, y]

    return x2, y2


def possible(img, w_size, x, y):
    xx = img.shape[1]
    yy = img.shape[0]

    if not (0 <= x - w_size <= xx) or not (0 <= x + w_size <= xx):
        return False
    if not (0 <= y - w_size <= yy) or not (0 <= y + w_size <= yy):
        return False
    else:
        return True
```

Q4.2





```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    # ----- TODO -----
    # YOUR CODE HERE
    K1, K2 = intrinsics['K1'], intrinsics['K2']
    M1 = np.hstack((np.identity(3), np.zeros(3)[:, np.newaxis]))
    C1 = K1.dot(M1)

    fig = plt.figure()
    ax = plt.axes(projection='3d')

    pts2 = []

    for (x1, y1) in temple_pts1:
        x2, y2 = epipolarCorrespondence(im1, im2, F, x1, y1)
        pts2.append([x2, y2])

    pts2 = np.array(pts2)

    M2, C2, P = findM2(F, temple_pts1, pts2, intrinsics, filename='q3_3.npz')

    C2 = K2.dot(M2)

    filename = "q4_2.npz"
    np.savez('results/' + filename, F=F, M1=M1, M2=M2, C1=C1, C2=C2)

    ax.scatter3D(P[:, 0], P[:, 1], P[:, 2])

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.show()

    return P
```

Question 5

Q5.1

When I run the ransac with the noisy points, I have to iterate more to get a good F where as to running the eight point algorithm will give me a good value right away. But overall with the right tuning (parameters niters/tolerance) the results are both good.

I used the error calculation matrix in helper function which calculates the error distances of the reprojection and returns the error.

I checked for every point and decided the inliers based on if the reprojection error for that set of points was below the tolerance.

Overall, when I increase the nIters, the F got closer to the F I got when I used the eightpoint algorithm. This is natural as the random sampling can be better as we sample more.

For the tolerance, I found that 5 was most optimal. If I decreased it down to 1, most of the points did not make it to the inlier category and my F generated a 3D point that was on a plane. When I increased it to 20, the they determined that most of the points were inliers and gave me the wrong F that made my 3D points not in the shape of the temple.

```
def ransacF(pts1, pts2, M, nIters=500, tol=5):
    # Replace pass by your implementation
    max_iters = nIters
    inlier_tol = tol

    iters = 0
    max_right = 0

    rand_len = pts1.shape[0]
    sampled = 8

    M1 = np.hstack((np.identity(3), np.zeros(3)[:, np.newaxis]))

    bestF = None

    while iters < max_iters:
        iters += 1

        rand_points = np.random.choice(range(rand_len), sampled)

        pts1_c = pts1[rand_points, :]
        pts2_c = pts2[rand_points, :]

        F = eightpoint(pts1_c, pts2_c, M)

        right = np.zeros((rand_len, 1))
```

```
        pts1_homo, pts2_homo = toHomogenous(pts1), toHomogenous(pts2)
        res = calc_epi_error(pts1_homo, pts2_homo, F)

        for i in range(rand_len):
            err = res[i]
            if err < inlier_tol:
                right[i] = 1
            else:
                right[i] = 0
        if np.sum(right) >= max_right:
            bestF = F
            max_right = np.sum(right)
            inliers = right

    print(iters, "::", np.sum(max_right))

    idx = np.where(inliers)[0]

    pts1 = noisy_pts1[idx, :]
    pts2 = noisy_pts2[idx, :]

    bestF = eightpoint(pts1, pts2, M)

    np.savez('results/q5_1.npz', bestF=bestF, inliers=inliers)

    return bestF, inliers
```

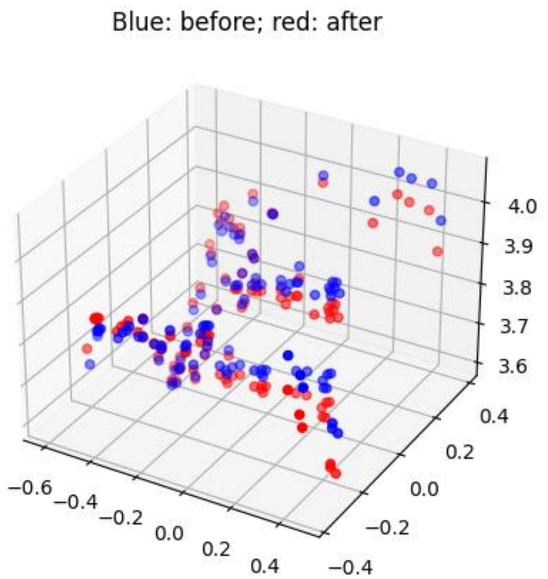
Q5.2

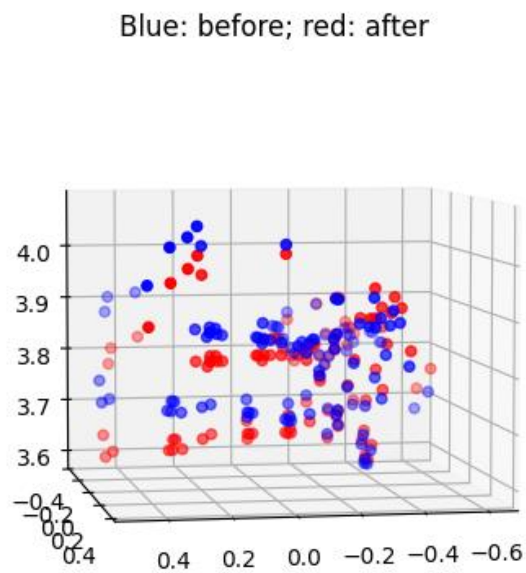
```
def rodrigues(r):
    # Replace pass by your implementation
    theta = np.linalg.norm(r)
    I = np.diag([1, 1, 1])

    if theta == 0:
        return I
    u = r / theta
    u = u.reshape(3)
    [u1, u2, u3] = u[0], u[1], u[2]
    ux = np.array(
        [[0, -u3, u2],
         [u3, 0, -u1],
         [-u2, u1, 0]]
    )
    R = math.cos(theta) * I + (1 - math.cos(theta)) * u @ u.T + math.sin(theta) * ux
    return R
```

```
def invRodrigues(R):
    # Replace pass by your implementation
    c = (R[0, 0] + R[1, 1] + R[2, 2] - 1) / 2

    A = (R - R.T) / 2
    p = np.array([A[2, 1], A[0, 2], A[1, 0]]).T
    s = np.linalg.norm(p)
    if s == 0 and c == 1:
        return np.zeros((3, 1))
    elif s == 0 and c == -1:
        t = R + np.diag([1, 1, 1])
        for i in range(3):
            if np.sum(t[:, i]) != 0:
                v = t[:, i]
                break
        u = v / np.linalg.norm(v)
        up = u * math.pi

        if (np.linalg.norm(up) == math.pi and (u[0] == u[1] and u[2] < 0)) or ((u[0]
== 0 and u[1] < 0) or u[0] < 0):
            up = -up
        r = up
    theta = np.arctan2(s, c)
    if math.sin(theta) != 0:
        u = p / s
        r = theta * u
    return r
```

Q5.3

```
init error: 491972.47419176035
after error 22.905571686573936
```






```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Replace pass by your implementation
    P, r2, t2 = x[:-6], x[-6:-3], x[-3:]
    N = P.shape[0] // 3
    P = P.reshape((N, 3))
    r2 = r2.reshape((3, 1))
    t2 = t2.reshape((3, 1))
    R2 = rodrigues(r2)
    M2 = np.hstack((R2, t2))

    P = np.vstack((P.T, np.ones((1, N))))
    p1_hat = K1 @ M1 @ P
    p1_hat /= p1_hat[2, :]

    p2_hat = K2 @ M2 @ P
    p2_hat /= p2_hat[2, :]

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


def bundleAdjustment(K1, M1, pts1, K2, M2_init, pts2, P_init):
    # ----- TODO -----

    R2_init, T2_init = M2_init[:, :3], M2_init[:, 3:]
    r2_init = invRodrigues(R2_init)
    x = np.concatenate([P_init[:, :3].reshape((-1, 1)), r2_init.reshape((-1, 1)), T2_init]).reshape((-1, 1))
    obj_start = np.linalg.norm(rodriguesResidual(K1, M1, pts1, K2, pts2, x)) ** 2
```

```
    def f(x):
        r = np.sum(rodriguesResidual(K1, M1, pts1, K2, pts2, x) ** 2)
        return r

    t = scipy.optimize.minimize(f, x)
    f = t.x

    P, r2, t2 = f[:-6], f[-6:-3], f[-3:]
    P, r2, t2 = P.reshape((-1, 3)), r2.reshape((3, 1)), t2.reshape((3, 1))
    R2 = rodrigues(r2).reshape((3, 3))
    M2 = np.hstack((R2, t2))

    obj_end = t.fun

    np.savez('results/q5.npz', f=f, M2=M2, P=P, obj_start=obj_start,
obj_end=obj_end)

    return M2, P, obj_start, obj_end
```

Question 6

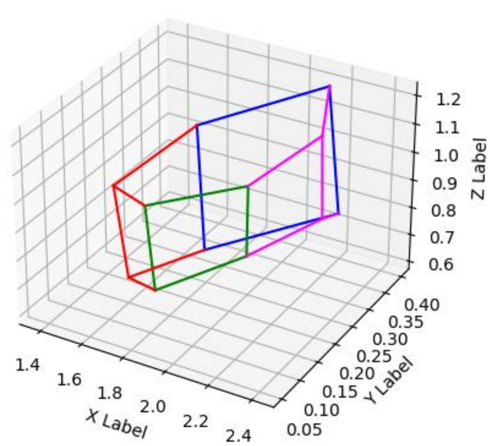
Q6.1

Since I have 3 images instead of 2, for every point in P(world 3D space) I saw if the points were really valid (above threshold) and calculated the 3D points with only the confident values. For example if pts1, pts2's confidence was over the threshold and pts3's confidence was not, then I used triangulate on the two points. On the other hand, when all three points were confident, then I used all 3 points in triangulate. For that I needed to make a triangulate_with_three_points function. In it, I just modified A to be

$$A = \begin{bmatrix} y_1 c_{13} - c_{12} \\ c_{11} - x_1 c_{13} \\ y_2 c_{23} - c_{22} \\ c_{21} - x_2 c_{23} \\ y_3 c_{33} - c_{32} \\ c_{31} - x_3 c_{33} \end{bmatrix}$$

Everything else is the same (using svd, taking the last col of v and so on).

The reprojection error for the first frame is 2429.7574300839638

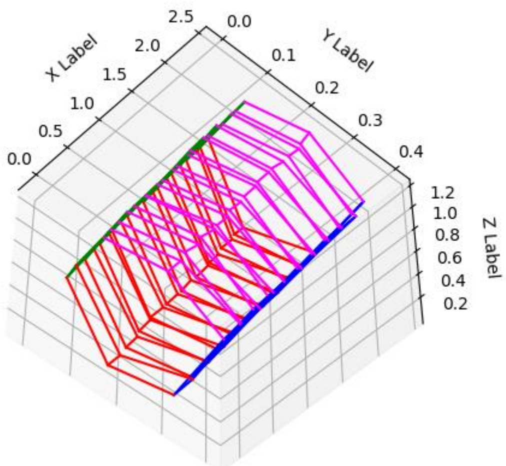


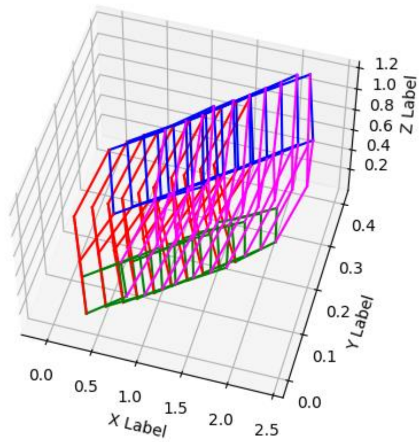
Regarding the thresholds, if I gave it a value lower than 300, the reprojection error went up very high as all of the points (pts1, pts2, pts3) were always used to calculate P. If I set it too high, then there would be times when there would be only one good point and thus cannot run the triangulate algorithm. So, after some experiments, I choose 300 as it did not run into any errors, and gave me a small reprojection error.

```python
def MultiviewReconstruction(K1, pts1, K2, pts2, K3, pts3, imgs, Thres=100):
    # Replace pass by your implementation

    thresh = 300

    realP = np.zeros((N, 3))
    for i in range(N):
        print("::", pts1[i, 2], pts2[i, 2], pts3[i, 2])
        if pts1[i, 2] > thresh and pts2[i, 2] > thresh and pts3[i, 2] >
thresh:
            X, error = triangulate3D(C1, pts1[i, :2].reshape(1, 2), C2,
pts2[i, :2].reshape(1, 2), C3,
                                     pts3[i, :2].reshape(1, 2))
        elif pts1[i, 2] > thresh and pts2[i, 2] > thresh:
            X, error = triangulate(C1, pts1[i, :2].reshape(1, 2), C2,
pts2[i, :2].reshape(1, 2))
        elif pts1[i, 2] > thresh and pts3[i, 2] > thresh:
            X, error = triangulate(C1, pts1[i, :2].reshape(1, 2), C3,
pts3[i, :2].reshape(1, 2))
        elif pts2[i, 2] > thresh and pts3[i, 2] > thresh:
            X, error = triangulate(C2, pts2[i, :2].reshape(1, 2), C3,
pts3[i, :2].reshape(1, 2))
        else:
            print("not working!!!")
        realP[i, :] = X

    return realP

def triangulate3D(C1, pts1, C2, pts2, C3, pts3):
    # (1)
    N = pts1.shape[0]

    X = np.zeros((N, 3))
    error = 0

    Proj1_n_list = np.zeros((N, 2))
    Proj2_n_list = np.zeros((N, 2))
    Proj3_n_list = np.zeros((N, 2))

    for i in range(N):
        A = np.array([
            pts1[i, 1] * C1[2, :] - C1[1, :],
            C1[0, :] - pts1[i, 0] * C1[2, :],
            pts2[i, 1] * C2[2, :] - C2[1, :],
            C2[0, :] - pts2[i, 0] * C2[2, :],
            pts3[i, 1] * C3[2, :] - C3[1, :],
            C3[0, :] - pts3[i, 0] * C3[2, :]
        ])
        u, s, vh = np.linalg.svd(A)
        P = vh[-1, :].reshape(4, -1)
        P /= P[-1, :]
        P = P.reshape(4)
        X[i, :] = P[:3]

        pt3D = P
        Proj1 = C1 @ pt3D.T  # 3xN
        Proj1_n = (Proj1 / Proj1[2])[:2]  # 2xN
```

```
        Proj1_n_list[i, :] = Proj1_n

        Proj2 = C2 @ pt3D.T  # 3xN
        Proj2_n = (Proj2 / Proj2[2])[:2]  # 2xN
        Proj2_n_list[i, :] = Proj2_n

        Proj3 = C3 @ pt3D.T  # 3xN
        Proj3_n = (Proj3 / Proj3[2])[:2]  # 2xN
        Proj3_n_list[i, :] = Proj3_n

        e1 = (np.linalg.norm(Proj1_n - pts1[i, :])) ** 2
        e2 = (np.linalg.norm(Proj2_n - pts2[i, :])) ** 2
        e3 = (np.linalg.norm(Proj3_n - pts3[i, :])) ** 2
        error += (e1 + e2 + e3)

    return X, error
```

Q6.2






```
def plot_3d_keypoint_video(pts_3d_video):
    # Replace pass by your implementation
    fig = plt.figure()
    num_points = len(pts_3d_video)
    ax = fig.add_subplot(111, projection='3d')
    for i in range(num_points):
        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)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()
```