Q1.1

Fundamental matrix maps a point in an image to a line in another image. This characteristic of fundamental matrix allows the equation below to be true.

$$x_2 F x_1 = 0$$

F is the fundamental matrix, x_1 and x_2 are corresponding pixels coordinates.

In the given figure, the corresponding point for each plane are its respective origin. Because F is a 3x3 matrix that can be represented as below, the above equation can be simplified after substituting origin coordinates.

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

$$[u_1 \quad v_1 \quad 1] \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix} = 0$$

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

Solving the above, below can be derived.

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

Q1.2

For unnormalized cameras with same intrinsic matrix, the relationship between the corresponding points on different image planes can be described using an essential matrix.

$$x_2^T E x_1 = 0$$

E is the essential matrix and x_1 and x_2 are corresponding homogenous image coordinates at image plane 1 and 2.

Given a rotation matrix and skew symmetric translation matrix, essential matrix can be found. In this case, because there is no rotation and only translation in x axis, the rotation matrix is just identity.

$$E = \hat{T} R$$

R is the rotation matrix from image plane 1 to 2, $\hat{T}$ is the skew symmetric translation matrix which can be converted from translation vector as below.

$$T = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} = \hat{T}$$

Therefore, the equation $x_2^T E x_1 = 0$ can be represented as the following (*dx* is translation in x axis).

$$x_2^T \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -dx \\ 0 & dx & 0 \end{bmatrix} x_1$$

In the equation above, if a pair of corresponding points in homogenous coordinates is given, the above can be solved to be

$$[x_2 \quad y_2 \quad 1] \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -dx \\ 0 & dx & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = 0$$

$$[0 \quad dx \quad -y_2 dx] \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = 0$$

$$y_1 dx - y_2 dx = 0$$

$$y_1 = y_2$$

The above is sufficient to demonstrate that the epipolar lines are parallel; for any given corresponding points, the y-coordinates of the points are always equal.

Q1.3

The relative rotation R_{rel} from frame i to frame j can be computed by multiplying the rotation matrices at each frame.

$$R_{rel} = R_j R_i^T$$

The relative translation t_{rel} from frame i to frame j can be found by first finding the different between the translation vector at each frame and multiplying the rotation matrix of previous frame.

$$t_{rel} = R_i^T (t_j - t_i)$$

From the relative rotation and translation matrices, the essential matrix $\boldsymbol{E}$ can be computed as in the previous question. $\widehat{\boldsymbol{t_{rel}}}$ is a skew symmetric form of relative translation vector.

$$\boldsymbol{E} = \widehat{\boldsymbol{t_{rel}}} \boldsymbol{R_{rel}}$$

The fundamental matrix is found by multiplying camera intrinsics as below.

$$\boldsymbol{F} = \boldsymbol{K^{-T}} \widehat{\boldsymbol{t_{rel}}} \boldsymbol{R_{rel}} \boldsymbol{K^{-1}}$$

Q2.1

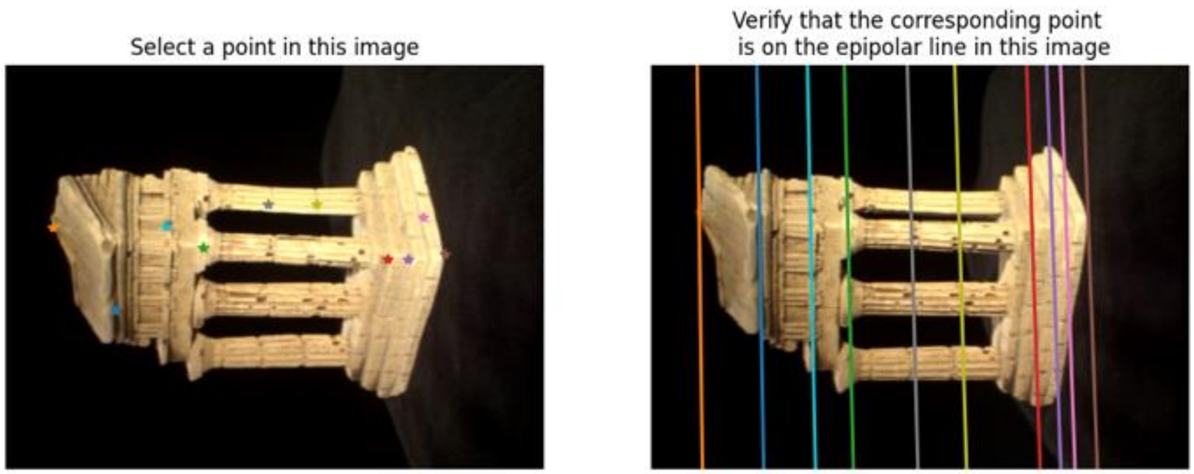
From the code, F matrix is found from the terminal output.

$$F = \begin{bmatrix} 0 & 0 & -0.251 \\ 0 & 0 & 0.003 \\ 0.242 & -0.007 & 1 \end{bmatrix}$$

```
Optimization terminated successfully.
         Current function value: 0.000107
         Iterations: 8
         Function evaluations: 819
[[-2.19299582e-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]]
```

[Terminal output of eightpoint.py with print(F)]

From displayEpipolarF, the following image was outputted from the code. Note that the points were selected arbitrarily.



[Output of eightpoint.py]

```python
def eightpoint(pts1, pts2, M):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE

    # scale the given points by M
    # so that difference in magnitude of points is not too large -> better SVD
    T =  np.diag([ 1/M , 1/M , 1.0])

    # use matrix T to scale the points (use homogenous coordinates)
    pts1_s = T.dot(toHomogenous(pts1).T).T
    pts2_s = T.dot(toHomogenous(pts2).T).T

    # de-homogenize the points
    pts1_s = pts1_s[:, :2]
    pts2_s = pts2_s[:, :2]

    # compute eight point algorithm's equation
    AF = np.zeros((pts1_s.shape[0], 9))
    for i in range(pts1_s.shape[0]):
        # AF = [x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]
        AF[i, :] = [ pts2_s[i,0]*pts1_s[i,0] , pts2_s[i,0]*pts1_s[i,1] , pts2_s[i,0],
                    pts2_s[i,1]*pts1_s[i,0] , pts2_s[i,1]*pts1_s[i,1] , pts2_s[i,1],
                    pts1_s[i,0], pts1_s[i,1], 1  ]

    # solve for F matrix using SVD
    u, s, vt = np.linalg.svd(AF)
    # f is the last column of v, so the last raw of vt
    f = vt[-1, :].reshape(3, 3)

    f = refineF(f, pts1_s, pts2_s)

    F = T.T.dot(f).dot(T)

    #scale F so that F[2,2] = 1
    F = F / F[2, 2]

    np.savez("q2_1.npz", F, M)

    return F
```

[Screenshot of function eightpoint]

Q2.2

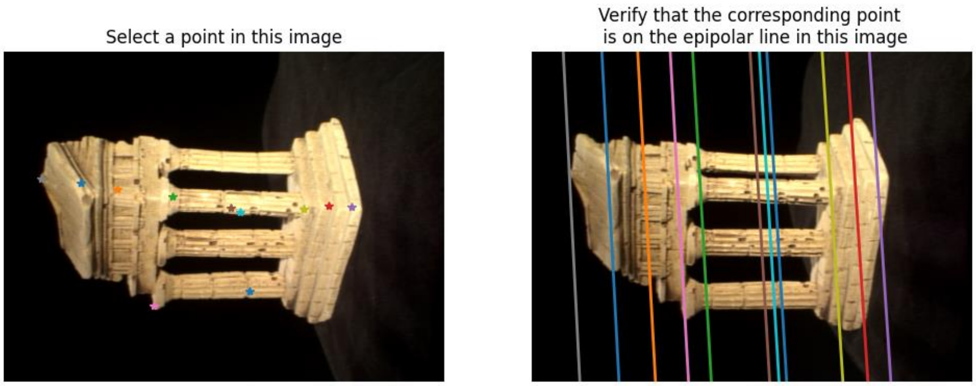
After constructing the code, one of the F matrix is found from the terminal output.

$$F = \begin{bmatrix} 0 & 0 & -0.251 \\ 0 & 0 & -0.008 \\ 0.242 & 0.002 & 1 \end{bmatrix}$$

```
Optimization terminated successfully.
         Current function value: 3.410082
         Iterations: 14
         Function evaluations: 2148
[[-3.09074108e-06  5.19535499e-06 -2.50832038e-01]
 [ 4.07719922e-05  4.70460905e-07 -7.89122701e-03]
 [ 2.42490342e-01  1.80080816e-03  1.00000000e+00]]
Error: 0.5452895832990327
```

[Terminal output of sevenpoint.py]

From displayEpipolarF, the following image was outputted from the code. Note that the points were selected arbitrarily.



[Output of sevenpoint.py]

```python
def sevenpoint(pts1, pts2, M):
    Farray = []

    # assert len(pts1) == len(pts2) == 7

    # Compute A matrix using the same method as in eightpoint
    AF = np.zeros((pts1.shape[0], 9))
    for i in range(pts1.shape[0]):
        # AF = [x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1]
        AF[i, :] = [ pts2[i,0]*pts1[i,0] , pts2[i,0]*pts1[i,1] , pts2[i,0],
                    pts2[i,1]*pts1[i,0] , pts2[i,1]*pts1[i,1] , pts2[i,1],
                    pts1[i,0], pts1[i,1], 1  ]

    # Solving for nullspace of A to get two Fs
    _, _, vt = np.linalg.svd(AF)
    fvec1 = vt[7]
    fvec2 = vt[8]

    Fmat = [np.array([  [fvec1[0], fvec1[3], fvec1[6]],  [fvec1[1], fvec1[4], fvec1[7]], [fvec1[2], fvec1[5], fvec1[8]] ]),
            np.array([  [fvec2[0], fvec2[3], fvec2[6]],  [fvec2[1], fvec2[4], fvec2[7]], [fvec2[2], fvec2[5], fvec2[8]] ])]

    # Find F that meets the singularity constraint
    D = np.zeros((2, 2, 2))
    for i1 in range(2):
        for i2 in range(2):
            for i3 in range(2):
                Dtmp = np.array([Fmat[i1][:, 0], Fmat[i2][:, 1], Fmat[i3][:, 2]]).T
                D[i1, i2, i3] = np.linalg.det(Dtmp)

    coefficients = np.array([
        -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],
        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],
        D[1, 1, 0] + D[0, 1, 1] + D[1, 0, 1] - 3*D[1, 1, 1],
        D[1, 1, 1]
    ])

    roots = np.roots(coefficients)

    for r in roots:
        if np.isreal(r):
            Ftmp = r.real * Fmat[0] + (1 - r.real) * Fmat[1]
            Ftmp = np.array(Ftmp)
            F = refineF(Ftmp, pts1, pts2)
            F = F / F[2, 2]
            Farray.append(F)

    return Farray
```

[Screenshot of function sevenpoint]

Q3.1

From the output of the program as shown in the figure below, the essential matrix is estimated to be

$$E = \begin{bmatrix} -0.507 & 68.654 & -371.96 \\ 29.71 & -1.547 & 9.682 \\ 372.99 & 2.985 & 0.150 \end{bmatrix}$$

```
         Current function value: 0.000107
         Iterations: 8
         Function evaluations: 819
[[-5.06936458e-01  6.86542948e+01 -3.71961460e+02]
 [ 2.97106977e+01 -1.54718776e+00  9.68232563e+00]
 [ 3.72991091e+02  2.98549846e+00  1.50354606e-01]]
```

Given a fundamental matrix and intrinsic camera matrices, the essential matrix can be computed from the following equation.

$$E = K_2^T F K_1$$

This is implemented in the program.

```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE
    E = K2.T.dot(F).dot(K1)
    return E
```

[Screenshot of essentialMatrix function]

Q3.2

Given two camera matrices C_1 and C_2, the following formula must be true.

$$x \times Cw = 0$$

Using the above constraint, the A_i matrix for each point can be derived as the following.

$$\begin{bmatrix} yC_3^T - C_2^T \\ C_1^T - xC_3^T \end{bmatrix} w = 0$$

In the matrix above, C_i^T refers to the i^{th} row of camera matrix C.

For two set of corresponding 2D points, the matrix can be expanded to be a 4x4 matrix. Note that each row is 1x4.

$$A_i = \begin{bmatrix} y_1 C_{1,3}^T - C_{1,2}^T \\ C_{1,1}^T - x_1 C_{1,3}^T \\ y_2 C_{2,3}^T - C_{2,2}^T \\ C_{2,1}^T - x_2 C_{2,3}^T \end{bmatrix}$$

```python
def triangulate(C1, pts1, C2, pts2):
    #initialize array for P and error
    Ps = list()
    err = 0

    #loop through all points
    N = pts1.shape[0]
    for i in range(N):
        x1, y1 = pts1[i, :]
        x2, y2 = pts2[i, :]

        # Compute A using derived formula
        A0 =  y1*C1[2, :] - C1[1, :]
        A1 = C1[0, :] - x1*C1[2, :]
        A2 =  y2*C2[2, :] - C2[1, :]
        A3 = C2[0, :] - x2*C2[2, :]
        A = np.stack((A0, A1, A2, A3), axis=0)

        # solve for lstsq solution using SVD
        __, __, Vt = np.linalg.svd(A)

        w_raw = Vt[-1, :]
        w_3d = w_raw[0:3] / w_raw[3] #normalize by last element
        Ps.append(w_3d)

        # calculate reprojection error using homogenous coordinates
        w_homo = np.zeros((4, 1), dtype=np.float32)
        w_homo[0:3, 0] = w_3d
        w_homo[3, 0] = 1

        p1_rep = C1 @ w_homo
        p2_rep = C2 @ w_homo

        # normalize reprojected points
        x1_rep, y1_rep = p1_rep[0:2, 0] / p1_rep[2, 0]
        x2_rep, y2_rep = p2_rep[0:2, 0] / p2_rep[2, 0]

        # calculate error and add each iteration
        err += (x1_rep-x1)**2 + (y1_rep-y1)**2 + (x2_rep-x2)**2 + (y2_rep-y2)**2

    P = np.stack(Ps, axis=0)
    return P, err
```

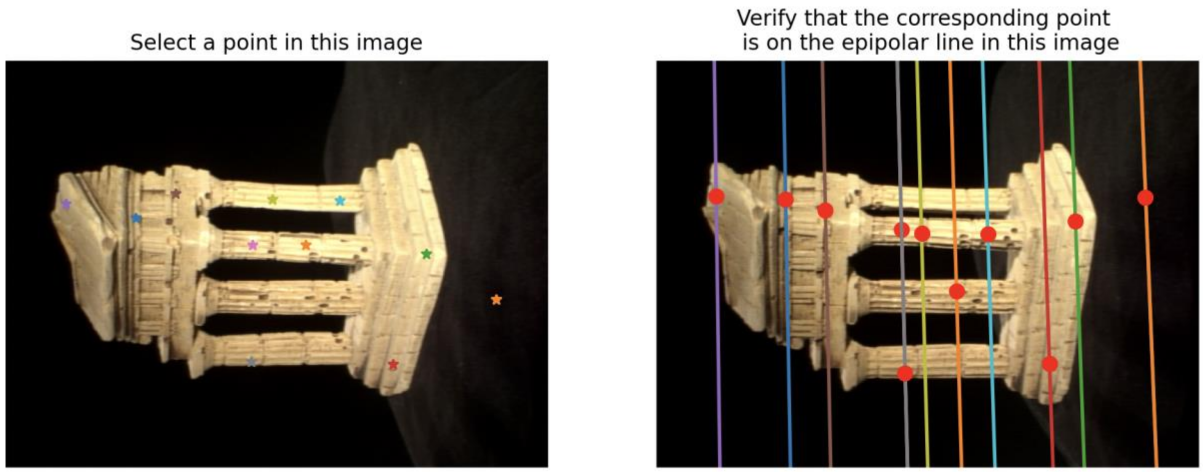
[Screenshot of function triangulate]

Q3.3

```python
def findM2(F, pts1, pts2, intrinsics, filename="q3_3.npz"):
    K1 = intrinsics['K1']
    K2 = intrinsics['K2']

    # CALCULATE E
    E = essentialMatrix(F, K1, K2)

    # CALCULATE M1 and M2
    M1 = np.array([ [ 1,0,0,0 ],
                    [ 0,1,0,0 ],
                    [ 0,0,1,0 ]  ])

    M2_list = camera2(E)

    #  TRIANGULATION
    C1 = K1.dot(M1)

    P = np.zeros( (pts1.shape[0],3) )
    M2 = np.zeros( (3,4) )
    C2 = np.zeros( (3,4) )
    prev_err = np.inf
    for i in range(M2_list.shape[2]):
        M2 = M2_list[:, :, i]
        C2 = K2.dot(M2)
        P_i, err = triangulate(C1, pts1, C2, pts2)
        if ( err<prev_err and np.min(P_i[:, 2])>=0):
            P = P_i
            M2 = M2
            C2 = C2
            prev_err = err

    np.savez(filename, M2=M2, C2=C2, P=P)

    return M2, C2, P
```

[Screenshot of function findM2]

Q4.1

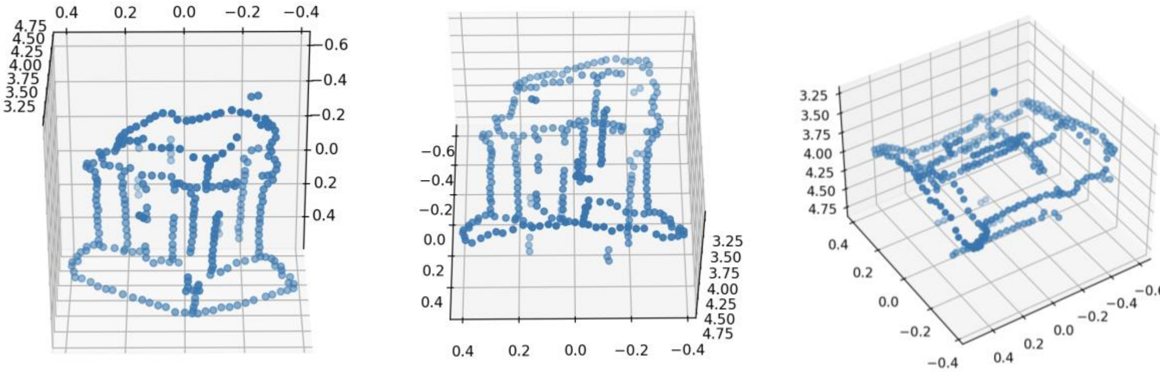


[Output of epipolarCorrespondence.py]

```
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # define gaussian filter kernel size and sigma
    kernel_size = 51
    sigma = 31

    #make window for gaussian filter
    window = np.zeros( (kernel_size, kernel_size) )
    window[kernel_size//2, kernel_size//2]=1

    #use ndimage gaussian filter to get gaussian kernel
    ker = gaussian_filter( window, sigma)
    ker = np.sum(ker)
    ker = np.asarray(ker)
    ker = np.dstack( ( ker, ker, ker )  )


    # Find epipolar line
    v = np.array([x1, y1, 1])
    l = F.dot(v)
    s = np.sqrt(l[0] ** 2 + l[1] ** 2)

    # line equation in normal form
    l = l / s

    # find intersection points of epipolar line with image borders
    sy, sx, __ = im2.shape
    if l[0] != 0:
        ye = sy - 1
        ys = 0
        xe = -(l[1] * ye + l[2]) / l[0]
        xs = -(l[1] * ys + l[2]) / l[0]
    else:
        xe = sx - 1
        xs = 0
        ye = -(l[0] * xe + l[2]) / l[1]
        ys = -(l[0] * xs + l[2]) / l[1]

    # find points on epipolar line
    N = max( (ye-ys), (xe-xs) )

    x2_list = np.linspace(xs, xe, N)
    y2_list = np.linspace(ys, ye, N)
    x2_list = np.rint(x2_list).astype(int)
    y2_list = np.rint(y2_list).astype(int)

    min_error = np.inf
    x2_min_error = None
    y2_min_error = None

    # find best match using gaussian weighted sum of squared differences
    k_half = kernel_size // 2 # for rounding
    k_half_1 = (kernel_size-1) // 2

    # check if points are within image borders
    if x1 >= k_half and y1 >= k_half and x1 <= sx-1-k_half_1 and y1 <= sy-1-k_half_1:
        patch_1 = im1[y1 - k_half: y1 - k_half + kernel_size, x1 - k_half: x1 - k_half + kernel_size, :]
        patch_1 = np.asarray(patch_1)

    # loop through all points on epipolar line
        for i in range(x2_list.shape[0]):
            x2 = x2_list[i]
            y2 = y2_list[i]

            if x2 >= k_half and y2 >= k_half and x2 <= sx-1-k_half_1 and y2 <= sy-1-k_half_1:
                patch_2 = im2[y2-k_half: y2-k_half+kernel_size, x2-k_half: x2-k_half+kernel_size, :]
                patch_2 = np.asarray(patch_2)

                diff = patch_1 - patch_2
                diff_gaussian = np.multiply(ker, diff)
                err = np.linalg.norm(diff_gaussian)

    # update best match
                if err<min_error:
                    min_error = err
                    x2_min_error = x2
                    y2_min_error = y2

    return x2_min_error, y2_min_error
```

[Screenshot of function epipolarCorredpondence]

Q4.2



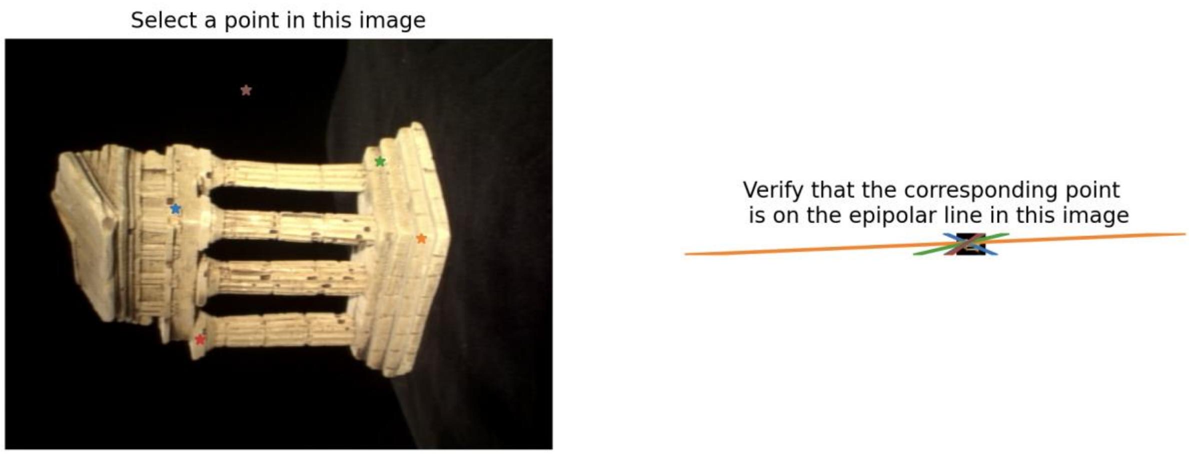
[3D Scatter Plot from visualize.py]

```python
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    # Initialize template points
    x1 = temple_pts1[:,0].astype(int).flatten()
    y1 = temple_pts1[:,1].astype(int).flatten()

    M1 = np.array([ [ 1,0,0,0 ],
                    [ 0,1,0,0 ],
                    [ 0,0,1,0 ]  ])

    C1 = K1.dot(M1)

    # FIND EPIPOLAR PTS2 CORRESPONDANCES
    pts1_new = []
    pts2_new = []

    # for each point in temple_pts1, find the corresponding point in temple_pts2
    for i in range(x1.shape[0]):
        x2, y2 = epipolarCorrespondence(im1, im2, F, x1[i], y1[i])
        if x2 is not None:
            pts1_new.append([ x1[i], y1[i] ])
            pts2_new.append([ x2, y2 ])

    # convert to numpy array
    pts1_new = np.asarray(pts1_new)
    pts2_new = np.asarray(pts2_new)

    # Find 3D points using triangulation
    M2_best, C2_best, P = findM2(F, pts1_new, pts2_new, intrinsics)

    # Save to npz
    np.savez('q4_2.npz', F=F, M1=M1, M2=M2_best, C1=C1 , C2=C2_best )

    return P
```

[Screenshot of function compute3D_pts]

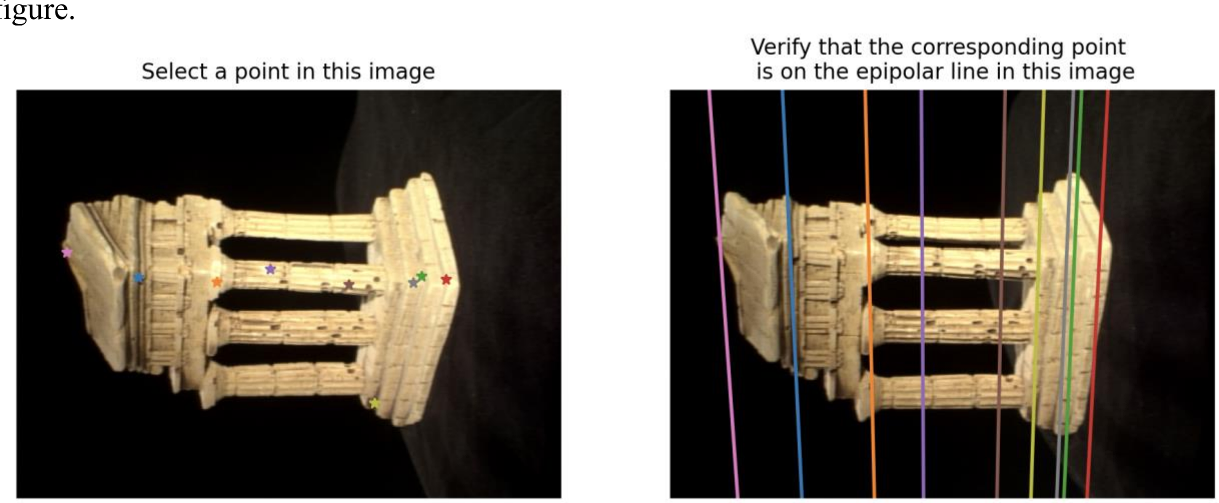
Q5.1

When give noisy data, the eight-point algorithm outputs a fundamental matrix that results in the following figure.



[Output of eightpoint.py with noisy correspondence data]

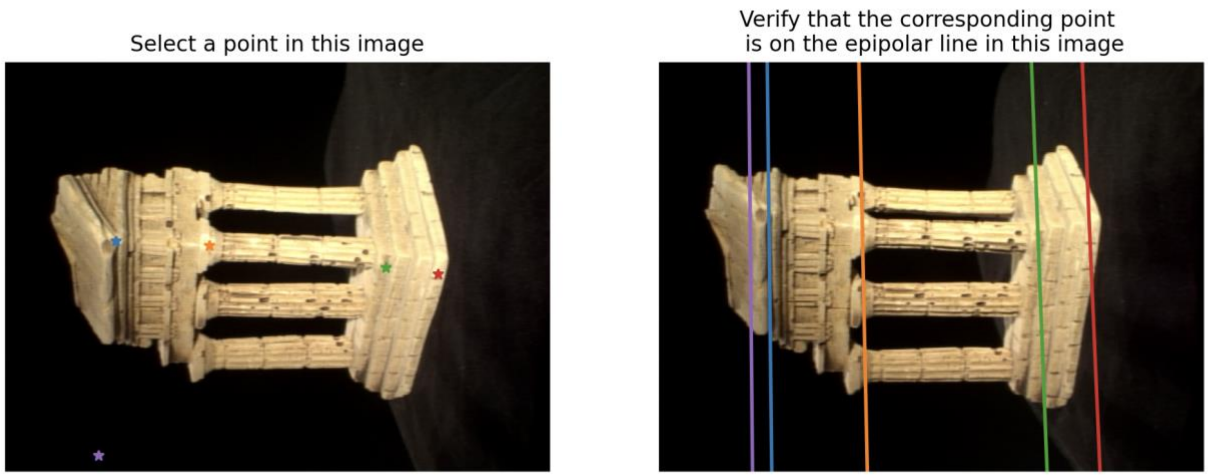
For RANSAC implementation with seven-point algorithm, the output results in the following figure.



[Output of ransacF function with noisy correspondence data (default parameter)]

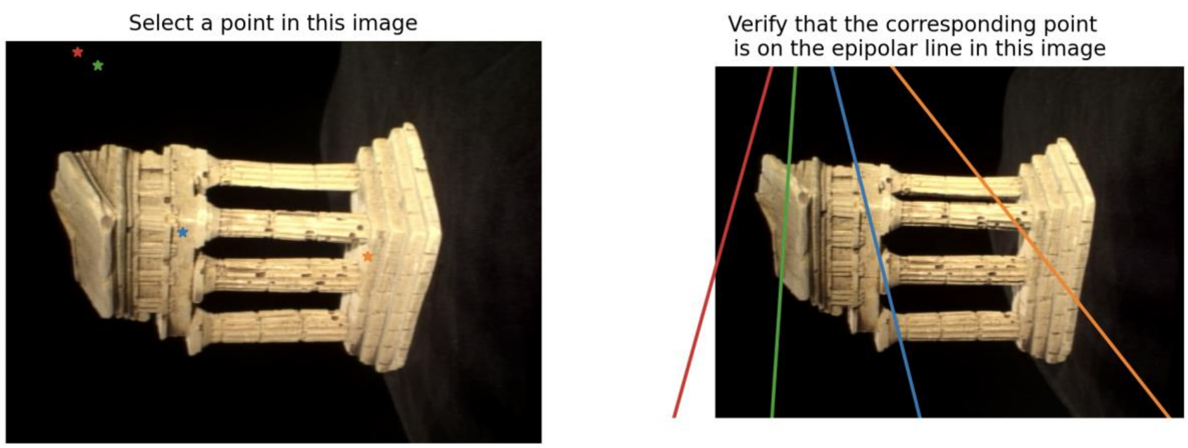
The error between the points and the epipolar line was computed using the helper function calc_epi_error, which essentially calculates the sum of the squared distance between the points and the estimated epipolar line. In the code, the computed error is compared iteratively with the input tolerance value and constructs output vector that identifies which points are the inliers. For a default parameter of tolerance = 10 and nIters = 1000, the resulting number of inliers are computed to be 110.

Intuitively, by varying the nIters parameter, the RANSAC algorithm can output different results every time it is run, because decreasing the number of iterations implies less sampling. As shown in the result below, when nIters = 100 parameter is given, it outputs a different result when compared to the output above. Increasing the number of iterations results in slower execution of the program.



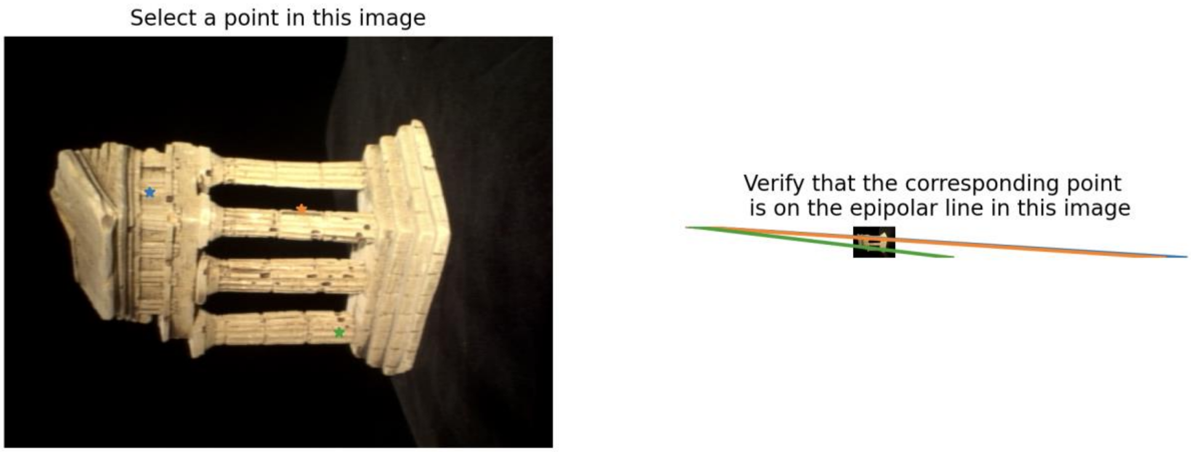
[Output of ransacF function with noisy correspondence data (nIters=100)]

On the other hand, lowering the tolerance parameter results in fewer inliers, which may produce inaccurate F matrix. For example, if tolerance is lowered to 1, the program outputs an F matrix that produces the following.



[Output of ransacF function with noisy correspondence data (tol=1)]

Also, if tolerance parameter is increased, the algorithm may take noisy data into account when constructing the fundamental matrix and produce inaccurate result as below.



[Output of ransacF function with noisy correspondence data (tol=1000)]

```python
def ransacF(pts1, pts2, M, nIters=10, tol=1000):

    max_inliers = -np.inf

    for __ in range(nIters):
        ran_points = random.sample(range(0, pts1.shape[0]), 7)

        pts1_sample = pts1[ran_points]
        pts2_sample = pts2[ran_points]

        F_list = sevenpoint(pts1_sample, pts2_sample, M)

        for F_tmp in F_list:
            total_inliers = 0
            inlier_tmp = np.zeros(pts1.shape[0], dtype=bool)
            for k in range(pts1.shape[0]):
                #make homogenous points
                x1 = np.array(  [pts1[k,0], pts1[k,1], 1] ).reshape(1,3)
                x2 = np.array(  [pts2[k,0], pts2[k,1], 1] ).reshape(1,3)

                # use epipolar constraint to check if point is inlier
                if calc_epi_error(x1, x2, F_tmp) < tol:
                    total_inliers = total_inliers +1
                    inlier_tmp[k] = True
                else:
                    inlier_tmp[k] = False

            if total_inliers > max_inliers:
                max_inliers = total_inliers
                inliers = inlier_tmp
                F = F_tmp

    print("max inliers: ", max_inliers)

    return F, inliers
```

[Screenshot of function ransacF]

Q5.2

```python
def rodrigues(r):
    # TODO: Replace pass by your implementation
    zero = 1e-30 # threshold for checking if theta is close to 0
    theta = np.linalg.norm(r) # theta is the length of r

    if np.abs(theta) < zero:
        return np.eye(3, dtype=np.float32) # if ~0 then return identity matrix
    else:
        u = r / theta
        u_cross = np.array([[0, -u[2], u[1]], [u[2], 0, -u[0]], [-u[1], u[0], 0]], dtype=np.float32)
        u = u.reshape(3,1)

        # Rodrigues formula
        R = np.eye(3, dtype=np.float32) * np.cos(theta) + (1 - np.cos(theta)) * (u @ u.transpose()) + u_cross * np.sin(theta)

        return R
```

[Screenshot of function rodrigues]

```python
def invRodrigues(R):
        # Arctan as defined in pdf
    def arctan2(y, x):
        if isgreater(x, 0):
            return np.arctan(y / x)
        elif isgreater(0, x):
            return np.pi + np.arctan(y / x)
        elif isequal(x, 0) and isgreater(y, 0):
            return np.pi*0.5
        elif isequal(x, 0) and isgreater(0, y):
            return -np.pi*0.5

    def isequal(a,b): # to check if close to 0
        zero = 0.001
        return np.abs(a - b) < zero

    def isgreater(a,b): # to check if greater than 0
        zero = 0.001
        return a - b > zero

    def S_half(r): # function for half sphere
        length = np.sum(r**2)**0.5
        r1, r2, r3 = r[0], r[1], r[2]
        if (isequal(length, np.pi) and isequal(r1, r2) and isequal(r1, 0) and isgreater(0, r3)) or (isequal(r1, 0) and isgreater(0, r2)) or  isgreater(0, r1):
            return -r
        else:
            return r

    zero = 0.0001
    A = (R - R.transpose()) / 2
    a32, a13, a21 = A[2, 1], A[0, 2], A[1, 0]
    rho = np.array([[a32], [a13], [a21]], dtype=np.float32).T
    s = np.sum(rho**2)**0.5
    c = (R[0, 0]+R[1, 1]+R[2, 2] - 1) / 2.0
    if isequal(s, 0) and isequal(c, 1):
        return np.zeros((3, 1), dtype=np.float32)
    elif isequal(s, 0) and isequal(c, -1):
        V = R+np.eye(3, dtype=np.float32)
        # find a nonzero column of V
        mark = np.where(np.sum(V**2, axis=0) > zero)[0]
        v = V[:, mark[0]]
        u = v / (np.sum(v**2)**0.5)

        r = S_half(u*np.pi)
        return r
    elif not isequal(s, 0):
        u = rho / s
        theta = arctan2(s, c)
        return u*theta
```

[Screenshot of function invRodrigues]

Q5.3

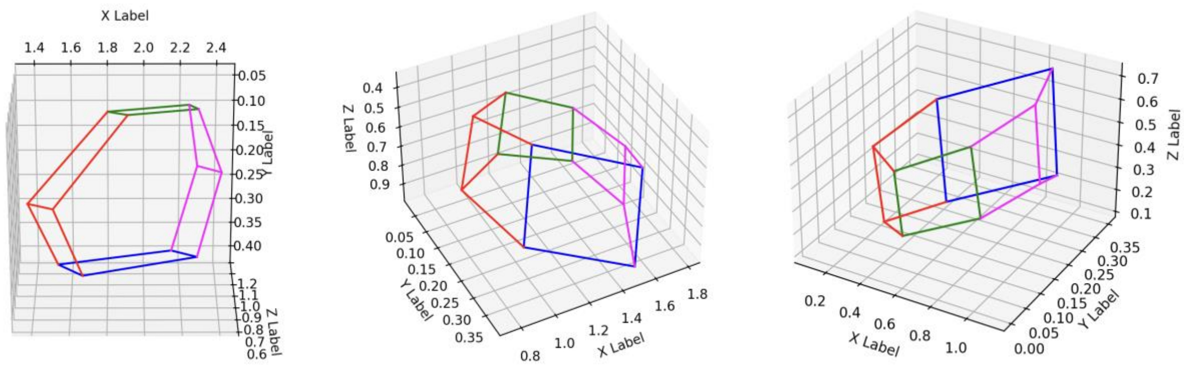
```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    n = p1.shape[0]
    P = x[0:3*n].reshape(n, 3)
    r2 = x[3*n:3*n+3]
    t2 = x[3*n+3:3*n+6]

    R2 = rodrigues(r2)

    t2 = t2.reshape(3,1)
    M2 = np.concatenate((R2, t2), axis=1)

    P_h = np.concatenate( ( P, np.ones( (P.shape[0], 1) ) ), axis=1 ).transpose()

    p1_rep_h = K1 @ M1 @ P_h
    p1_rep = p1_rep_h[0:2, :] / p1_rep_h[2, :]
    p2_rep_h = K2 @ M2 @ P_h
    p2_rep = p2_rep_h[0:2, :] / p2_rep_h[2, :]

    p1_hat = p1_rep.transpose()
    p2_hat = p2_rep.transpose()
    e1 = (p1 - p1_hat).reshape(-1)
    e2 = (p2 - p2_hat).reshape(-1)

    residuals = np.concatenate((e1, e2), axis=0)

    return residuals
```

```python
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    obj_start = obj_end = 0

    R_init = M2_init[:, 0:3]
    r_init = invRodrigues(R_init)
    t_init = M2_init[:, 3].reshape([-1])

    # Ensure P_init is reshaped properly
    P_init_flattened = P_init.reshape(-1)

    # Construct the initial concatenated vector
    x_init  = np.hstack([P_init_flattened, r_init.ravel(), t_init])

    func = lambda x: (rodriguesResidual(K1, M1, p1, K2, p2, x)** 2).sum()
    res = scipy.optimize.minimize(func, x_init, options={'disp': True})

    x_new = res.x

    n = p1.shape[0]
    P_new = x_new[0:3*n].reshape(n, 3)
    r_new = x_new[3*n:3*n+3]
    t_new = x_new[3*n+3:3*n+6, None]

    R_new = rodrigues(r_new)

    # Construct the final optimized M2
    M2_new = np.hstack([R_new, t_new])

    # Objective function values
    obj_start = func(x_init)
    obj_end = func(x_new)

    return M2_new, P_new, obj_start, obj_end
```

[Screenshot of functions rodriguesResidual and bundleAdjustment]

Q6.1

In order to compute the 3D points, the triangulate function that was made for Q3 was modified to take another set of points. First, the points were compared against the threshold inputted to the function and those who had greater confidence values were chosen to construct a 6x6 A_i matrix. Then similarly as in 2 point reconstruction, SVD is used to compute for a 3D point.

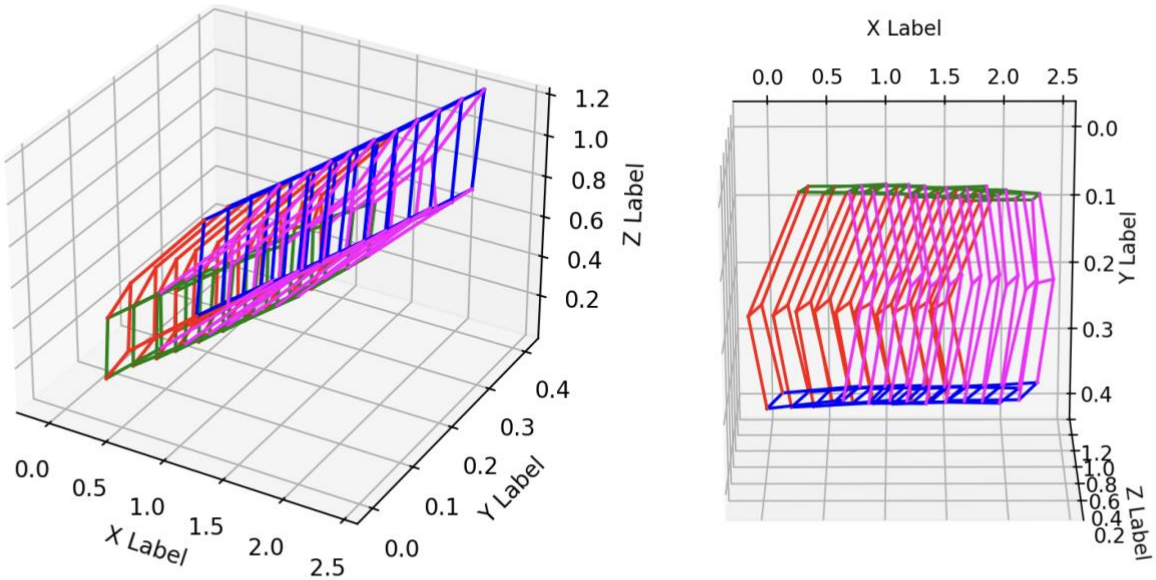


[Output of plot_3D_keypoint at frames 0, 4, 8]

```python
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres=100):
    # assert pts1.shape[0] == pts2.shape[0]

    Ps = list()
    err = 0
    new_pts1 = []
    new_pts2 = []
    new_pts3 = []

    N = pts1.shape[0]
    for k in range(N):
        if pts1[k, 2] > Thres:
            new_pts1.append(pts1[k, :])
        if pts2[k, 2] > Thres:
            new_pts2.append(pts2[k, :])
        if pts2[k, 2] > Thres:
            new_pts3.append(pts3[k, :])

    new_pts1 = np.asarray(new_pts1)
    new_pts2 = np.asarray(new_pts2)
    new_pts3 = np.asarray(new_pts3)

    for i in range(N):
        x1, y1, __ = new_pts1[i, :]
        x2, y2, __ = new_pts2[i, :]
        x3, y3, __ = new_pts3[i, :]

        # construct A
        A0 =  y1*C1[2, :] - C1[1, :]
        A1 = C1[0, :] - x1*C1[2, :]
        A2 =  y2*C2[2, :] - C2[1, :]
        A3 = C2[0, :] - x2*C2[2, :]
        A4 =  y3*C3[2, :] - C3[1, :]
        A5 = C3[0, :] - x3*C3[2, :]
        A = np.stack((A0, A1, A2, A3, A4, A5), axis=0)

        # solve w, just find the null space
        U, s, Vt = np.linalg.svd(A)
        w_raw = Vt[-1, :] #(4,)
        w_3d = w_raw[0:3] / w_raw[3] #(3,)
        Ps.append(w_3d)

        # get reproject error
        w_homo = np.zeros((4, 1), dtype=np.float32)
        w_homo[0:3, 0] = w_3d
        w_homo[3, 0] = 1
        p1_rep = C1 @ w_homo
        p2_rep = C2 @ w_homo
        p3_rep = C3 @ w_homo

        x1_rep, y1_rep = p1_rep[0:2, 0] / p1_rep[2, 0]
        x2_rep, y2_rep = p2_rep[0:2, 0] / p2_rep[2, 0]
        x3_rep, y3_rep = p3_rep[0:2, 0] / p3_rep[2, 0]

        err += (x1_rep-x1)**2 + (y1_rep-y1)**2 + (x2_rep-x2)**2 + (y2_rep-y2)**2 + (x3_rep-x3)**2 + (y3_rep-y3)**2

    P = np.stack(Ps, axis=0)
    return P, err
```

[Screenshot of function MultiviewReconstruction]

Q6.2



[Graphical output of function plot_3d_keypoint_video]

```python
def plot_3d_keypoint_video(pts_3d_video):

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    for i in range(pts_3d_video.shape[0]):
        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()
```

[Screenshot of function plot_3d_keypoint_video]