16720 – hw4
mrohitth

## Problem 1.1

It is given that both principal points of two image planes coincide with coordinate (0,0), thusthe projected point x in the two image planes are $x^T$ = [0 0 1] and $x^T$ = [0 0 1]

According to the property of fundamental matrix when the points are at the origin we have,

$$[x_2 \;\; y_2 \;\; 1]\begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix}\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = 0$$

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

On simplifying the above matrices, all the terms that have the image point coordinates get cancelled at the origin and therefore, we get the following relation.

$$F_{33} = 0$$

## Problem 1.2

For pure translation along the x-axis, transformation matrices are as follows:

$$R = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} \quad and \quad t = \begin{matrix} t_1 \\ 0 \\ 0 \end{matrix}$$

The essential matrix is given by, $E = t_x R$

$t_x$ and essential matrix can be written as follows,

$$t_x = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{matrix} \quad and \quad E = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{matrix}$$

Let $x^T_1$ = [a1 a2 1] and $x^T_2$ = [b1 b2 1], we have2

$$l_1^T = x_2^T E$$

$$l_1^T = [b_1 \ b_2 \ 1] \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix} = [0 \quad t_1 \quad -b_2 t_1]$$

$$l_2^T = x_1^T E$$

$$l_2^T = [a_1 \ a_2 \ 1] \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t_1 \\ 0 & -t_1 & 0 \end{bmatrix} = [0 \quad -t_1 \quad a_2 t_1]$$

Thus, the epipolar line in the first camera is $t_1 y_1 - b_2 t_1 = 0$, and the epipolar line in the second camera is $-t_1 y_2 + a_2 t_1 = 0$. Both of them do not contain x components, so they are both parallel to the x-axis

## Problem 1.3

Let P be 3D world coordinates of the point in the image, $p_1$ be 2D coordinates on the image plane at time frame i and $p_2$ be 2D coordinates on the image plane at time frame i+1 World coordinate and image plane can be related by:

$$P = t_1 + R_1 p_1$$

$$p_1 = R_1^{-1}(P - t_1)$$

Similarly,

$$p_2 = R_2^{-1}(P - t_2)$$

Combining the above equations:

$$p_2 = R_2^{-1}(t_1 + R_1 p_1 - t_2)$$

$$p_2 = R_2^{-1} R_1 p_1 + R_2^{-1}(t_1 - t_2)$$

Comparing the above equation with $p_2 = R_{rel} p_1 + t_{rel}$

$$R_{rel} = R_2^{-1} R_1$$
$$t_{rel} = R_2^{-1}(t_1 - t_2)$$

Also. from the equations for essential matrix and fundamental matrix:

$$E = [t_{rel}]_x R_{rel}$$

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

## Problem 1.4

Let $C$ and $C^J$ be the camera in the real and virtual world respectively, its intrinsic matrixbe $K$. Let $P$ and $x$ be the 3D point in real world and the point in image plane and $P^J$ and$x^J$ be its reflection in the in the mirror and this point in the image plane. Given the mirror is flat, the transformation between these two points is a pure translation.

$$R = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} \quad and \quad t = \begin{matrix} t_1 \\ t_2 \\ t_3 \end{matrix}$$

$$P^J = P + t$$

$$\lambda_1 \; x = K \; P$$

$$\lambda_2 \; x^J = K \; P^J$$

With the help of the above equations, relationship between the two points is as follows:

$$\lambda_2 K^{-1} \; x^J = \lambda_1 K^{-1} \; x + t$$

We can simplify the equation and eliminate some terms by taking cross product with $t$ onboth sides, followed by dot product with $x^J$ to get the following:

$$x^{JT} K^{-T} \; t_\times K^{-1} \; x = 0$$

$$t_x = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}$$

We can see that, $t_\times$ is a skew-symmetric matrix here and we know the relation that $x^{JT} F x = 0$. Comparing this with above form, we get the following expression for $F$:

$$F = K^{-T} \; t_\times K^{-1}$$

Since, $t_\times$ is skew symmetric, it can be shown that $F$ here will also retain the property ofskew-symmetric for a given intrinsic matrix $K$.

$$F^T = - F$$

Therefore, we can conclude that the two images of the object are related by a skew-symmetricfundamental matrix.

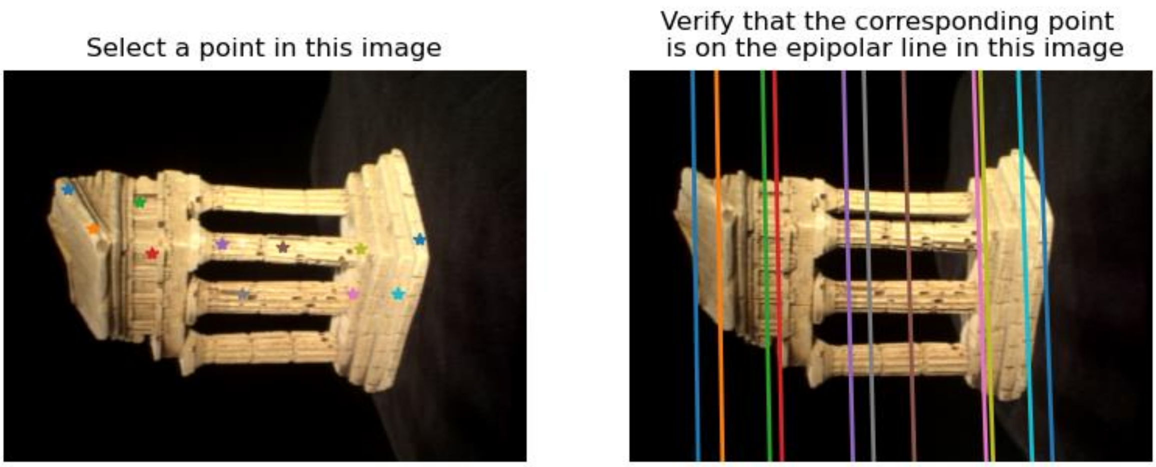
## Problem 2.1

q2_1_eightpoint.py

```python
def eightpoint(pts1, pts2, M):
    F = None            #fundamental matrix
    N = pts1.shape[0]   # Extrating the number of points

    pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)

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

    pts1_norm = pts1_homogenous @ T
    pts2_norm = pts2_homogenous @ T
    A = []
    for i in range(pts2_norm.shape[0]):
        h1 =[pts1_norm[i, 0]*pts2_norm[i, 0], pts1_norm[i, 0]*pts2_norm[i, 1], pts1_norm[i, 0],
            pts1_norm[i, 1]*pts2_norm[i, 0], pts1_norm[i, 1]*pts2_norm[i, 1], pts1_norm[i, 1],
            pts2_norm[i, 0], pts2_norm[i, 1], 1]
        A.append(h1)
    A = np.array(A)
    u, s, vh = np.linalg.svd(A)


    F = vh[-1,:]
    #print(h.shape)

    F = F.reshape((3,3))
    F = F.T
    F = refineF(F,pts1_norm[:, :-1],pts2_norm[:, :-1] )
    F = np.transpose(T)@ F @ T

    F = F/F[2,2] #Finding the unique fundamental matrix by setting the scale to 1.
    if(os.path.isfile('q2_1.npz')==False):
        np.savez('q2_1.npz',F = F, M = M)

    return F

```

The recovered matrix F from eight-point algorithm is

```
[[-2.18962367e-07   2.95584511e-05   -2.51851099e-01]
 [ 1.28367203e-05   -6.63934217e-07   2.63094865e-03]
 [ 2.42194841e-01   -6.81933857e-03   1.00000000e+00]]
```



## Problem 2.2

q2_2_sevenpoint.py

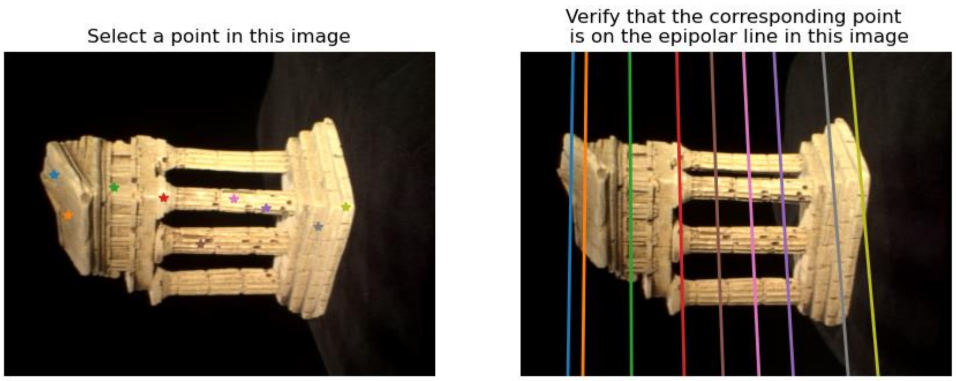
```python
    # YOUR CODE HERE
    pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)

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

    pts1_norm = pts1_homogenous @ T
    pts2_norm = pts2_homogenous @ T
    A = []
    for i in range(pts2_norm.shape[0]):
        h1 =[pts1_norm[i, 0]*pts2_norm[i, 0], pts1_norm[i, 0]*pts2_norm[i, 1], pts1_norm[i, 0],
            pts1_norm[i, 1]*pts2_norm[i, 0], pts1_norm[i, 1]*pts2_norm[i, 1],
            pts1_norm[i, 1], pts2_norm[i, 0], pts2_norm[i, 1], 1]
        A.append(h1)
    A = np.array(A)
    u, s, vh = np.linalg.svd(A)

    F1 = vh[-1,:]
    F2 = vh[-2,:]
    #prints(h.shape)

    F1 = F1.reshape((3,3))
    F2 = F2.reshape((3,3))

    c0 = np.linalg.det(F2)
    c2 = (np.linalg.det(2* F2 -F1) + np.linalg.det(F2))/2 - np.linalg.det(F2)
    c3 = (np.linalg.det(2*F1 -F2) - 2*c2 + c0 - 2* np.linalg.det(F1))/6
    c1 = np.linalg.det(F1) -c0 -c2 - c3

    alpha = np.polynomial.polynomial.polyroots((c0, c1, c2, c3))
    sol = [a.real for a in alpha if(a.imag == 0)]

    for i in range(len(sol)):
        a = sol[i]
        if a.imag == 0:
            r = a.real
            F = r*F1 + (1-r)*F2
            F = np.transpose(T)@ F @ T
            F = _singularize(F)
            Farray.append(F)
        else:
            continue

    Farray = np.stack(Farray, axis=-1)
    Farray /= Farray[2,2]

    return Farray.T
```

The recovered matrix F from seven-point algorithm is

[[ 2.17048089e-06   -1.37879287e-05   -1.95306038e-01]

[ 4.72931040e-05   1.56277138e-07   -6.67499589e-03]

[ 1.86889771e-01   2.55438738e-03   1.00000000e+00]]



## Problem 3.1

q3_1_essential_matrix.py

```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    # print(K2.shape, F.shape, K1.shape)
    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')


    # ----- TODO -----
    # YOUR CODE HERE

    F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
    E = essentialMatrix(F, K1, K2)
    # print(E, E.shape)

    if(os.path.isfile('q3_1.npz')==False):
        np.savez('q3_1.npz', F = F, E = E)

    # Simple Tests to verify your implementation:
    assert(E[2, 2] == 1)
    assert(np.linalg.matrix_rank(E) == 2)
```

## Problem 3.2

Let $C_{1i}$ be the $i^{th}$ row of $C_1$ and $C_{2i}$ be the $i^{th}$ row of $C_2$. If $W_i$ is a 4X1 vector of the 3Dcoordinates in the homogeneous form, we have

$$C_1 W_1 = x_{i1}$$

$$\begin{matrix} C_{11} \\ C_{12} \\ C_{13} \end{matrix} \quad x \quad \begin{matrix} u_i \\ v_i \\ w_i \\ 1 \end{matrix} \quad = \begin{matrix} x_{i1} \\ y_{i1} \\ 1 \end{matrix}$$

and,

$$C_2 W_2 = x_{i2}$$

$$\begin{matrix} C_{21} \\ C_{22} \\ C_{23} \end{matrix} \quad x \quad \begin{matrix} u_i \\ v_i \\ w_i \\ 1 \end{matrix} \quad = \begin{matrix} x_{i2} \\ y_{i2} \\ 1 \end{matrix}$$

$$C_{11} W_i = x_{i1} \quad C_{12} W_i = y_{i1} \quad C_{13} W_i = 1$$

$$C_{21} W_i = x_{i2} \quad C_{22} W_i = y_{i2} \quad C_{23} W_i = 1$$

On rearranging the terms, we get,

$$(x_{i1} C_{13} - C_{11})\ Wi = 0$$

$$(y_{i1} C_{13} - C_{12})\ Wi = 0$$

$$(x_{i2} C_{23} - C_{21})\ Wi = 0$$

$$(y_{i2} C_{23} - C_{22})\ Wi = 0$$

Thus, A can be written as,

$$A_i = \begin{bmatrix} x_{i1} C_{13} - C_{11} \\ y_{i1} C_{13} - C_{12} \\ x_{i2} C_{23} - C_{21} \\ y_{i2} C_{23} - C_{22} \end{bmatrix}$$

## Problem 3.3

q3_2_triangulate.py

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

    N = pts1.shape[0]
    P = np.zeros((N, 3))

    for i in range(0, N):
            a1 = np.multiply(pts1[i, 0], C1[2, :]) - C1[0, :]
            a2 = np.multiply(pts1[i, 1], C1[2, :]) - C1[1, :]
            a3 = np.multiply(pts2[i, 0], C2[2, :]) - C2[0, :]
            a4 = np.multiply(pts2[i, 1], C2[2, :]) - C2[1, :]
            A = np.vstack((a1,a2,a3,a4))

            u,s,v = np.linalg.svd(A)
            f1 = v[3, :]
            f1 = f1/f1[3]
            P[i, :] = f1[:3]

    a = np.ones((1, N))
    p_temp = np.vstack((P.T, a))
    pts1_new = np.matmul(C1, p_temp)
    pts1_new = pts1_new / pts1_new[2, :]

    pts2_new = np.matmul(C2, p_temp)
    pts2_new = pts2_new / pts2_new[2, :]

    pts1_new = pts1_new[:2, :]
    pts2_new = pts2_new[:2, :]

    # print((pts1.T).shape, pts1_new.shape, (pts2.T).shape, pts2_new.shape)

    error = np.power(np.subtract(pts1.T, pts1_new),2) + np.power(np.subtract(pts2.T, pts2_new), 2)
    error = np.sum(error)
    # print (error)


    return P, error
```

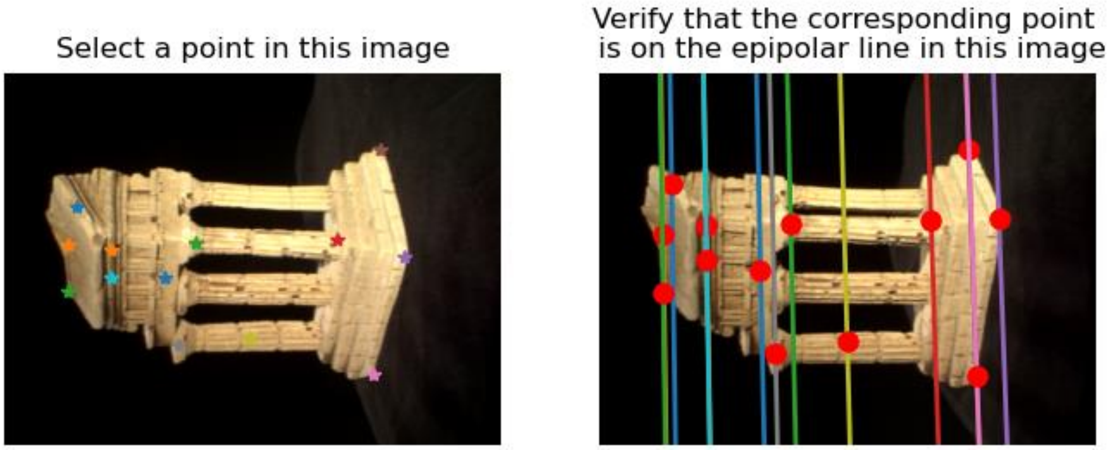
```python
def findM2(F, pts1, pts2, intrinsics, filename = 'q3_3.npz'):
    '''
    Q2.2: Function to find the camera2's projective matrix given correspondences
        Input:  F, the pre-computed fundamental matrix
                pts1, the Nx2 matrix with the 2D image coordinates per row
                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 th

    ***
    Hints:
    (1) Loop through the 'M2s' and use triangulate to calculate the 3D points and projection error.
        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 = intrinsics['K1']
    K2 = intrinsics['K2']

    E = essentialMatrix(F, K1, K2)

    M1 = np.hstack(((np.eye(3)), np.zeros((3, 1))))
    M2s = camera2(E)
    row, col, num = np.shape(M2s)

    C1 = np.matmul(K1, M1)

    # print(num)
    for i in range(num):
        M2 = M2s[:, :, i]
        C2 = np.matmul(K2, M2)
        P, err = triangulate(C1, pts1, C2, pts2)
        if (np.all(P[:,2] > 0)) :
            break
    #print("P is :",P)

    if(os.path.isfile('q3_3.npz')==False):
        np.savez('q3_3.npz', M2 = M2, C2 = C2, P = P)

    return M1, C1, M2, C2, P
```

16720 – hw4
mrohitth

## Problem 4.1

q4_1_epipolar_correspondence.py

```python
def Gaussian(shape, sigma):
    m, n = [(ss-1.)/2. for ss in shape]
    y, x = np.ogrid[-m:m+1, -n:n+1]
    h = np.exp( -(x*x + y*y) / (2.*sigma*sigma) )
    h[h < np.finfo(h.dtype).eps*h.max()] = 0
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h

def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Replace pass by your implementation
    P1 = np.vstack((x1, y1, 1))
    e = np.matmul(F, P1)
    e = e/np.linalg.norm(e)
    a = e[0][0]
    b = e[1][0]
    c = e[2][0]

    step = 10
    sigma = 5
    min_dis = np.inf

    #filter code here
    x1 = int(round(x1))
    y1 = int(round(y1))
    x2 = 0
    y2 = 0
    patch1 = im1[y1-step:y1+step+1, x1-step:x1+step+1]
    kernel = Gaussian((2*step+1, 2*step+1), sigma)

    for i in range(y1-sigma*step, y1+sigma*step):
        x2_curr = (-b*i-c)/a
        # x2_temp = round(x2_curr)
        x2_curr = int(round(x2_curr))

        s_h = i-step
        e_h = i+step+1
        s_w = x2_curr-step
        e_w = x2_curr+step+1
        if s_w > 0 and e_w < im2.shape[1] and s_h > 0 and e_h < im2.shape[0]:
            patch2 = im2[s_h:e_h, s_w:e_w]

            weightedDist = []
            for l in range(0, patch2.shape[2]):
                dist = np.subtract(patch1[:, :, l], patch2[:, :, l])
                weightedDist.append(np.linalg.norm(np.matmul(kernel, dist)))
            error = sum(weightedDist)

            if error < min_dis:
                min_dis = error
                x2 = x2_curr
                y2 = i
    # print(f"Best Error {error}")
    return x2, y2
```

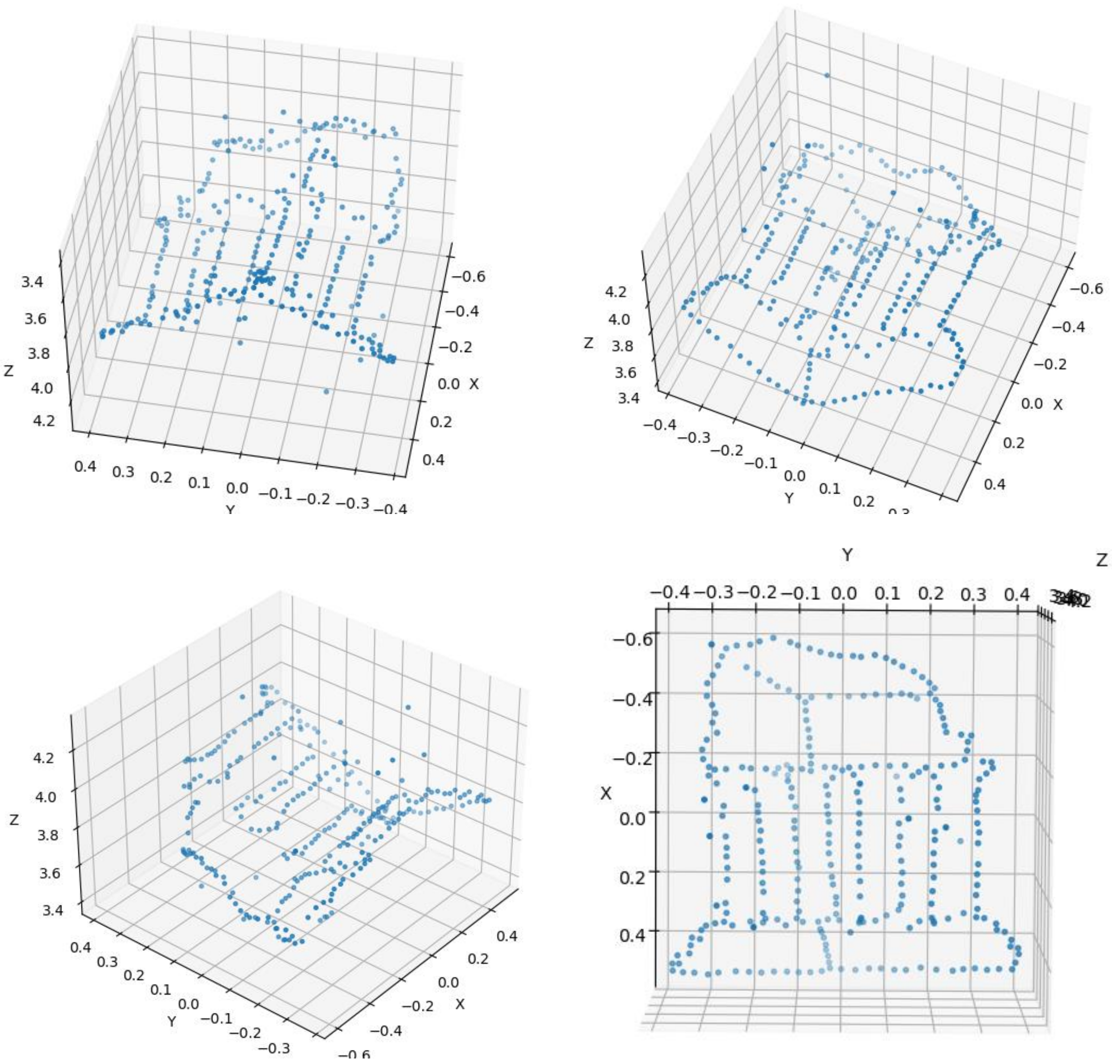



*Epipolar Correspondences*

## Problem 4.2

q4_2_visualize.py

```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2, C1, C2):

    # ----- TODO -----
    # YOUR CODE HERE
    pts2 = np.asarray([epipolarCorrespondence(im1, im2, F, temple_pts1[i, 0],
                        temple_pts1[i, 1]) for i in range(temple_pts1.shape[0]) ])

    P, _ = triangulate(C1, temple_pts1, C2, pts2)

    fig = plt.figure()
    ax = Axes3D(fig)
    ax.scatter(P[:, 0], P[:, 1], P[:, 2], s=7)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    plt.show()


    return P
```



## Problem 5.1

```python
'''
def ransacF(pts1, pts2, M, nIters=100, tol=10):
    # Replace pass by your implementation

    print("In ransac")

    N = pts1.shape[0]
    pts1_homo, pts2_homo = toHomogenous(pts1), toHomogenous(pts2)
    best_inlier = 0
    inlier_curr = np.zeros((pts1.shape[0],))
    # ----- TODO -----
    # YOUR CODE HERE

    choices = []
    ninlier = 0
    for i in range(nIters):
        print("Iterations :",i)
        try:
            choice = np.random.choice(range(pts1.shape[0]), 7)
            pts1_choice = pts1[choice, :]
            pts2_choice = pts2[choice, :]
            Fs = sevenpoint(pts1_choice, pts2_choice, M)
            for Fi in Fs:
                choices.append(choice)
                res = calc_epi_error(pts1_homo,pts2_homo, Fi)
                idx = np.where(res < tol)
                ninlier = np.array(idx).size
                if ninlier > best_inlier:
                    best_inlier = ninlier
                    F = Fi
                    idxmax=[]
                    idxmax.append(idx)
        except ValueError:
            print("Division by zero")

    inlier_curr[tuple(idxmax)] = 1
    inlier_curr = np.expand_dims(inlier_curr, axis = 1).astype(bool)

    return F, inlier_curr

```

If $error_i$ is smaller than a tolerance (i.e., 0.82), we consider this pair of points as an inlier. Effect of number of iterations and tolerance on fundamental matrix.

1) As the number of iterations is increased the fundamental matrix becomes more accuratein determining the points as it is a non-deterministic algorithm that produces result only with a certain probability, the probability increases with iterations. But after a point of threshold iteration the results provided by fundamental matrix stops changing value.

2) With the increase in the tolerance value more inliers and thus data points get included due to which estimation of fundamental matrix becomes more accurate. But higher tolerance will allow all the points to be included as inliers and defeat the purpose of RANSAC.

## Problem 5.2

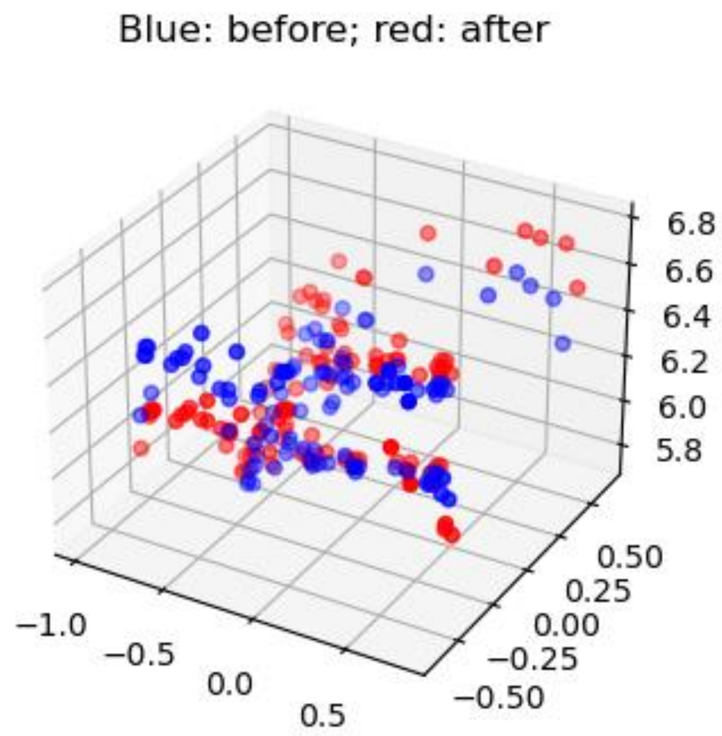
```
def rodrigues(r):

    theta = np.linalg.norm(r)
    if(theta == 0):
        return np.eye(3)
    u = r/theta
    #print(u.shape[0])
    u_cap = np.array([[0, -u[2], u[1]], [u[2], 0, -u[0]], [-u[1], u[0], 0]])
    u = u.reshape((u.shape[0],1))
    ## print(u.shape)
    R = np.eye(3)*np.cos(theta) + (1 - np.cos(theta))*(u@u.T) + u_cap * np.sin(theta)
    return R


'''
Q5.2: Inverse Rodrigues formula.
    Input:  R, a rotation matrix
    Output: r, a 3x1 vector
'''
def invRodrigues(R):

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

    if(s == 0 and c == 1):
        return np.zeros(3)
    elif(s == 0 and c == -1):
        v_ = R + np.eye(3)
        for i in range(3):
            if (np.count_nonzero(v_[:,i])) > 0:
                v = v_[:,i]
                print(v)
                break
        u = v/np.linalg.norm(v)
        r = u*np.pi

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

## Problem 5.3

```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Replace pass by your implementation
    residuals = None
    N = p1.shape[0]
    P = x[:-6].reshape((N,3))
    P = np.vstack((np.transpose(P), np.ones((1, N))))
    R2 = rodrigues(x[-6:-3].reshape((3,)))
    t2 = x[-3:].reshape((3,1))
    #print(R2.shape,t2.shape,R2,t2)
    M2 = np.hstack((R2, t2))

    C1 = K1 @ M1
    C2 = K2 @ M2

    p1_proj = C1 @ P
    p1_proj = p1_proj / p1_proj[2,:]
    p2_proj = C2 @ P
    p2_proj = p2_proj / p2_proj[2,:]
    p1_proj_coord = p1_proj[0:2,:].T
    p2_proj_coord = p2_proj[0:2,:].T
    residuals = np.concatenate([(p1-p1_proj_coord).reshape([-1]), (p2-p2_proj_coord).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
    R2init, t2init = M2_init[:,0:3], M2_init[:,3]

    x0 = np.concatenate((P_init.flatten(), invRodrigues(R2init).flatten(), t2init.flatten()))


    def func(x): #K1, M1, p1, K2, p2,
        return ((rodriguesResidual(K1, M1, p1, K2, p2, x))**2).sum()

    obj_start= func(x0)#**2).sum()
    x_upd = optimize.minimize(func, x0, method = 'CG' ).x  #leastsq



    obj_end= func(x_upd)#**2).sum()
    N = p1.shape[0]
    P = x_upd[:-6].reshape((N,3))
    R2 = rodrigues(x_upd[-6:-3].reshape((3,)))
    t2 = x_upd[-3:].reshape((3,1))
    M2 = np.hstack((R2, t2))

    return M2, P, obj_start, obj_end
```

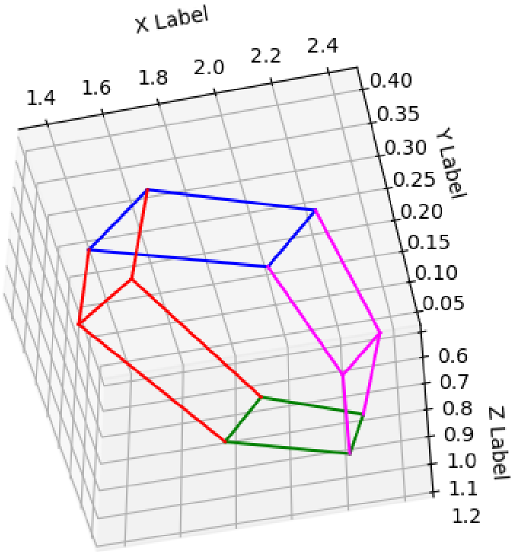
In this case, I optimized the F matrix again and ran the subsequent functions,
Re-projection Error for Inlier Points (Before Optimization) : 4941.746751245216
Re-projection Error for Inlier Points (After Optimization) : 13.103318510909448

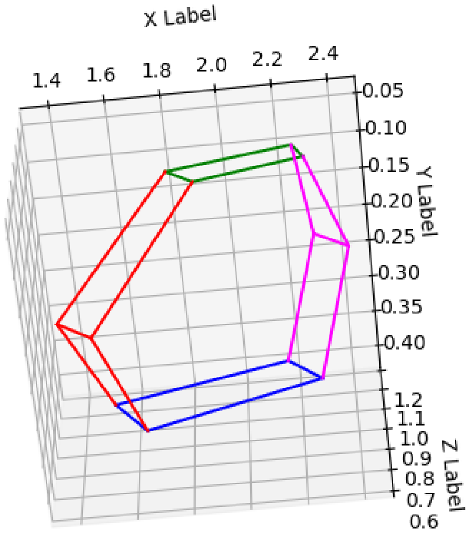

Blue: before; red: after
6.8
6.6
6.4
6.2
6.0
5.8
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5

## Problem 6

q6_ec_multiview_reconstruction.py

```
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres = 140):
    # Replace pass by your implementation
    pts1=pts1[pts1[:,2] > Thres]
    p1=pts1[:,:2]
    pts2=pts2[pts2[:,2] > Thres]
    p2=pts2[:,:2]
    pts3=pts3[pts3[:,2] > Thres]
    p3=pts3[:,:2]
    n, temp = p1.shape
    P = np.zeros((n,3))
    Phomo = np.zeros((n,4))
    for i in range(n):
        x1 = p1[i,0]
        y1 = p1[i,1]
        x2 = p2[i,0]
        y2 = p2[i,1]
        x3 = p3[i,0]
        y3 = p3[i,1]
        A1 = x1*C1[2,:] - C1[0,:]
        A2 = y1*C1[2,:] - C1[1,:]
        A3 = x2*C2[2,:] - C2[0,:]
        A4 = y2*C2[2,:] - C2[1,:]
        A5=  x3*C3[2,:] - C3[0,:]
        A6 = y3*C3[2,:] - C3[1,:]
        A = np.vstack((A1,A2,A3,A4,A5,A6))
        # print(A.shape)
        u, s, vh = np.linalg.svd(A)
        p = vh[-1, :]
        p = p/p[3]
        P[i, :] = p[0:3]
        Phomo[i, :] = p
        # print(p)
    p1_proj = np.matmul(C1,Phomo.T)
    lam1 = p1_proj[-1,:]
    p1_proj = p1_proj/lam1
    p2_proj = np.matmul(C2,Phomo.T)
    lam2 = p2_proj[-1,:]
    p2_proj = p2_proj/lam2
    err1 = np.sum((p1_proj[[0,1],:].T-p1)**2)
    err2 = np.sum((p2_proj[[0,1],:].T-p2)**2)
    err = err1 + err2
    # print(err)
    if(os.path.isfile('q6_1.npz')==False):
        np.savez('q6_1.npz',P=P)
    return P,err
    pass
```




In order to compute the 3D location, triangulate function was extended to three views bychanging the A matrix as follows:

$$A_i = \begin{bmatrix} x_{i1}C_{13} - C_{11} \\ y_{i1}C_{13} - C_{12} \\ x_{i2}C_{23} - C_{21} \\ y_{i2}C_{23} - C_{22} \\ x_{i3}C_{33} - C_{31} \\ y_{i3}C_{33} - C_{32} \end{bmatrix}$$

This modified A matrix is used to solve SVD and get the 3D location of points. From the given 2D points only the ones with confidence value greater than threshold is used for finding the 3D locations. After multiple iterations with the threshold, value of 140 gave the best results.