Computer Vision (16-720 B): Homework 4

3D Reconstruction

Name - Saharsh Agarwal
AndrewID - saharsh2

Question 1 : Theory

Question 1.1

Answer -

Let **x** be $[x_i, y_i, 1]^T$ and **x'** be $[x'_i, y'_i, 1]^T$ - both image points corresponding to the same point **P** in the 3D world $[X_i, Y_i, Z_i]^T$. x and x' is the homogeneous form in camera 1 and camera 2.

Since both x and x' pass through principal axes of respective cameras - $x_i, y_i, x'_i, y'_i = 0$.

Since $x^T F x' = 0$,

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

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

Thus, by matrix multiplication, it has been proved that in such a case F_{33} is 0. (Answer)

Question 1.2

Answer -

Assuming both the cameras have the same intrinsics (=K). Assume first camera is placed at [0,0,0] and the second camera is shifted by pure translation. T = $[T_x, T_y, T_z]$; where T_y and T_z are 0 and R = $I_{3\times3}$.

Essential matrix (E) -

$$E = \hat{T}R = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -T_x \\ 0 & T_x & 0 \end{bmatrix} \tag{3}$$

Let **x** be $[x_i, y_i, 1]^T$ and **x'** be $[x'_i, y'_i, 1]^T$ - both image points corresponding to the same point **P** in the 3D world $[X_i, Y_i, Z_i]^T$. x and x' is the homogeneous form in camera 1 and camera 2.

Epipolar line (L') associated with x' -

$$L' = Ex' = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -T_x \\ 0 & T_x & 0 \end{bmatrix} \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -T_x \\ T_x y' \end{bmatrix} \tag{4}$$

These are the coefficients of the line ax' + by' +c = 0. This shows that the y-component is constant and no dependence on x-component, implying the epipolar line is parallel to the x-axis. Similar derivation can be shown for the point x in camera 1. Thus, when 2 cameras differ only in pure translation in the x-direction, the epipolar lines are parallel to the x-axis. (y = y')

Question 1.3

Answer -

At time ti_1 : R_1, T_1, time ti_2 : R_2, T_2. Camera intrinsics remain constant (=K). If point in real world: X = $[X, Y, Z]^T$. The point in camera frame of reference:

$$x_1 = R_1 X + T_1, \quad => X = R_1^{-1}(x_1 - T_1) \tag{5}$$

$$x_2 = R_2 X + T_2 \quad => x_2 = R_2 R_1^{-1}(x_1 - T_1) + T_2 \quad => x_2 = (R_2 R_1^{-1})x_1 - (R_2 R_1^{-1} T_1 + T_2) \tag{6}$$

$$R_{rel} = (R_2 R_1^{-1}), \quad t_{rel} = -(R_2 R_1^{-1} T_1 + T_2) \tag{7}$$

Thus, E and F can be solved -

$$E = \hat{t_{rel}} R_{rel} = -(R_2 R_1^{-1} T_1 + T_2) \times (R_2 R_1^{-1}) \quad Cross - Product \tag{8}$$

$$F = K^{-T} E K^{-1} = K^{-T} \hat{t_{rel}} R_{rel} K^{-1} \tag{9}$$

Question 1.4

Viewing the mirror image is indeed like viewing the object from another position by the same camera (only translation involved and no rotation). Thus, translation T = $[T_x, T_y, T_z]$ and rotation R = $I_{3\times3}$. This translation is in the direction perpendicular to the mirror surface and magnitude is twice the perpendicular distance of the object from the mirror.

$$E = \hat{T}R = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \tag{10}$$

$$F = K^{-T} E K^{-1} = K^{-T} \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} K^{-1} \tag{11}$$

For checking if F is skew-symmetric - check $F^T = -F$

$$F^T = \left[K^{-T} \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} K^{-1} \right]^T = (K^{-1})^T \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}^T (K^{-T})^T \tag{12}$$

$$=> K^{-T} \begin{bmatrix} 0 & T_z & -T_y \\ -T_z & 0 & T_x \\ T_y & -T_x & 0 \end{bmatrix} K^{-1} = - \left[K^{-T} \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} K^{-1} \right] = -F \tag{13}$$

Hence proved => $F^T = -F$, thereby proving F is a skew-symmetric fundamental matrix in this case.

Question 2.1 - The 8-point Algorithm

Figure 1: Code for **q2_1_eightpoint.py**

```
def eightpoint(pts1, pts2, M):
    N = pts1.shape[0]
    pts1s = pts1/M #scaled1
    pts2s = pts2/M #scaled2

    points = []
    for i in range(N):
        pt = [ (pts1s[i,0]*pts2s[i,0]) , (pts1s[i,0]*pts2s[i,1]) , pts1s[i,0] ,
                    (pts1s[i,1]*pts2s[i,0]) , (pts1s[i,1]*pts2s[i,1]) , pts1s[i,1] ,
                    pts2s[i,0] , pts2s[i,1] , 1]
        points.append(pt)

    A = np.asarray(points)
    u,s,vt = np.linalg.svd(A)
    F = vt[-1,:].reshape((3,3)).T

    F = refineF(F,pts1s,pts2s)
    F = F/F[2,2]
    T = np.diag([1/M,1/M,1])
    us_F = T.T @ F @ T

    return us_F
```

The matrix **F** and scale **M** have been saved to **q2_1.npz** (and submitted in the zip).

Figure 2: Results for Display Epipolar in **q2_1_eightpoint.py**

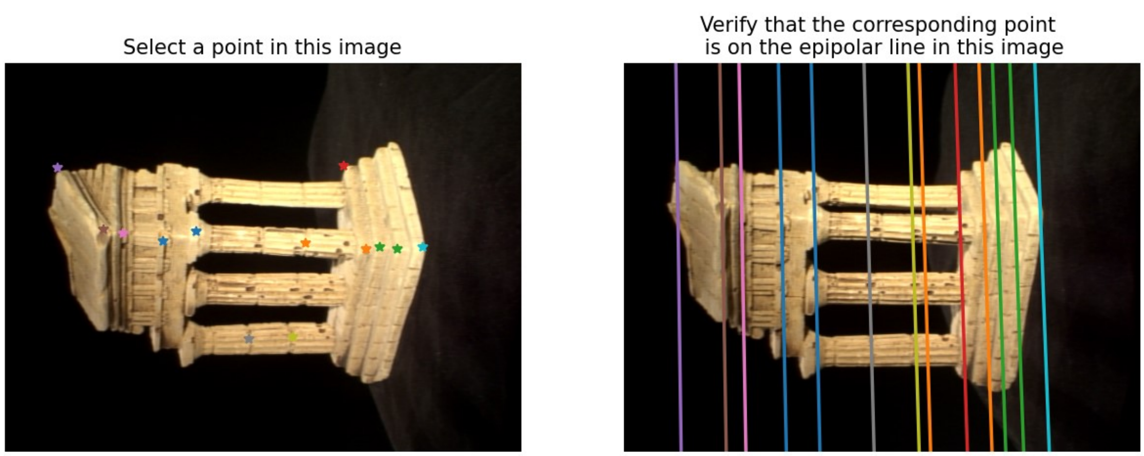


Figure 3: Results (F & M) for in **q2_1_eightpoint.py**

```
PS C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw4\hw4> python .\code\q2_1_eightpoint.py
Optimization terminated successfully.
         Current function value: 0.000107
         Iterations: 8
         Function evaluations: 893
M =  640
F =
[[-2.18962366e-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]]
PS C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw4\hw4>
```

Question 2.2 : Seven Point Algorithm

Figure 4: Code for **q2_1_sevenpoint.py**

```
def sevenpoint(pts1, pts2, M):

    Farray = []

    # ----- TODO -----
    # YOUR CODE HERE
    N = 7
    pts1s = pts1/M #scaled1
    pts2s = pts2/M #scaled2

    points = []
    for i in range(N):
        pt = [ (pts1s[i,0]*pts2s[i,0]) , (pts1s[i,0]*pts2s[i,1]) , pts1s[i,0] ,
                         (pts1s[i,1]*pts2s[i,0]) , (pts1s[i,1]*pts2s[i,1]) , pts1s[i,1] ,
                         pts2s[i,0] , pts2s[i,1] , 1]
        points.append(pt)

    A = np.asarray(points)
    u,s,vt = np.linalg.svd(A)
    F1 = vt[-1,:].reshape((3,3)).T
    F2 = vt[-2,:].reshape((3,3)).T

    #det(F(a)) = det(aF1 + (1-a)F2) = 0 (irrespective of a) = c0 + c1.a + c2.a^2 +c3.a^3

    c0 = np.linalg.det(F2) #putting a=0
    c2 = (np.linalg.det(F1)+np.linalg.det((2*F2)-F1)-(2*c0))/2 #a = 1 and a = -1 : both are added
    c1_c3 = (np.linalg.det(F1)-np.linalg.det((2*F2)-F1))*0.5 # when the above is subtracted
    c3 = (1/12)*(np.linalg.det(2*F1-F2)-np.linalg.det(3*F2-2*F1)-2*(np.linalg.det(F1) - np.linalg.det(2*F2-F1)))
    c1 = c1_c3-c3

    roots = npp.polyroots([c0,c1,c2,c3]) # computes roots of the polynomials with coefficients c
    T = np.diag([1/M,1/M,1])

    for i in roots:
        if i.imag == 0:
            a = i.real
            F = a*F1 + (1-a)*F2
            F = T.T @ F @ T # unscaling - as not refined, thus call singularise by hand
            F = _singularize(F)
            F = F/F[2,2] # making the last value 1
            Farray.append(F)
    Farray = np.asarray(Farray)
    #print(Farray.shape)
    return Farray
```

Figure 5: Results for Display Epipolar in **q2_1_sevenpoint.py**

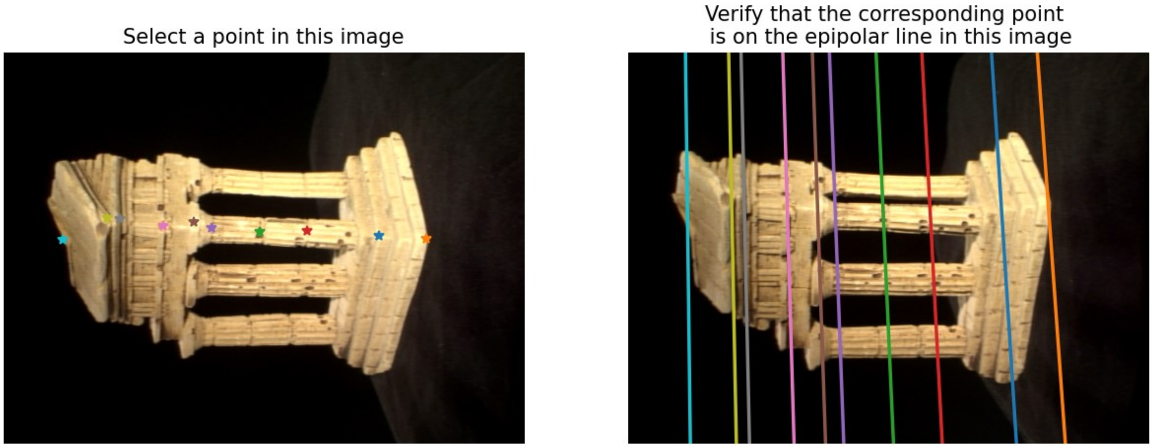


Figure 6: Results for in **q2_1_sevenpoint.py**

```
PS C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw4\hw4> python .\code\q2_2_sevenpoint.py
Error: 0.5668901239515978
F =
[[ 8.10457567e-07  8.90919506e-06 -2.01028424e-01]
 [ 2.63329748e-05 -6.00542594e-07  6.97429503e-04]
 [ 1.92182049e-01 -4.20123580e-03  1.00000000e+00]]
```

3 : Metric Reconstruction

Question 3.1

Figure 7: Code for **q3_1_essential_matrix.py**

```
def essentialMatrix(F, K1, K2):
    E = K2.T @ F @ K1
    #print("E : ", E)
    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",F=F,E=E)
    print("F =")
    print(F)
    print("E =")
    print(E)

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

The matrices **F** and **E** have been saved to **q3_1.npz** (and submitted in the zip).

Figure 8: Results in **q3_1_essential_matrix.py**

```
PS C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw4\hw4> python .\code\q3_1_essential_matrix.py
Optimization terminated successfully.
         Current function value: 0.000107
         Iterations: 8
         Function evaluations: 893
F =
[[-2.18962366e-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]]
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]]
```

Question 3.2

Let **x** be $[x_i, y_i, 1]^T$ and **x'** be $[x'_i, y'_i, 1]^T$ - both image points corresponding to the same point **X** in the 3D world $[X_i, Y_i, Z_i, 1]^T$. X is the homogeneous form of w_i. x and x' is the homogeneous form of $pts1_i$ in camera 1 and $pts2_i$ in camera 2.

$$x = \alpha C_1 X, \quad x' = \alpha C_2 X \tag{14}$$

where C_1 and C_2 are known Camera matrices (Dim = 3x4), and α is a scalar for homogeneous conversion. Showing for Camera 1 : (Matrix Cross Product - to remove α)

$$x \times C_1 X = 0 \tag{15}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \times \begin{bmatrix} C_1(Row1) \times X \\ C_1(Row2) \times X \\ C_1(Row3) \times X \end{bmatrix} = \begin{bmatrix} y * C_1(Row3) - C_1(Row2) \\ C_1(Row1) - x * C_1(Row3) \\ x * C_1(Row2) - y * C_1(Row1) \end{bmatrix} X = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{16}$$

The last equation in (3) is just linear combination of the first 2 equations. Thus, from camera 1 we get 2 equations. Similarly, from camera 2 we get 2 equation - which we concatenate to find A_i, i.e., A for the corresponding point.

Thus A_i is calculated to be :-

$$A_i = \begin{bmatrix} y * C_1(Row3) - C_1(Row2) \\ C_1(Row1) - x * C_1(Row3) \\ y' * C_2(Row3) - C_2(Row2) \\ C_2(Row1) - x' * C_2(Row3) \end{bmatrix} \tag{17}$$

Making the dimensions of A_i 4-by-4.

Question 3.3

Figure 9: Code for triangulate in **q3_2_triangulate.py**

```
def triangulate(C1, pts1, C2, pts2):
    w , err = 0,0
    N = pts1.shape[0]
    A = np.zeros((4,4))
    e = 0 #error
    P = np.zeros((N,3))

    for i in range(N):
        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,:]
        u,s,vh = np.linalg.svd(A)
        X = vh[-1,:]
        X = X/X[3]
        x = C1@X.T
        x = x/x[2]
        x_ = C2@X.T
        x_ = x_/x_[2]
        e = e + (np.linalg.norm(pts1[i]-x[0:2]) + np.linalg.norm(pts2[i]-x_[0:2]))
        P[i,:] = X[0:3]

    return P, e
```

The correct **M2**, and corresponding **C2** and P have been saved to **q3_3.npz** (submitted). (https://www.cs.cmu.edu/ 16385/s17/Slides/11.4_Triangulation.pdf)

Same error as expected from the FAQs. (next page)

Figure 10: Code for **findM2** in **q3_2_triangulate.py**

```
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)
    M1 = np.hstack((np.eye(3),np.zeros((3,1))))
    #print(M1)
    C1 = K1@M1
    M2s = camera2(E)
    e = 10000
    M2_best = None
    C2_best = None
    P_best = None

    for i in range(M2s.shape[-1]): #essentially 4 times
        M2 = M2s[:,:,i]
        C2 = K2@M2
        P_trial, err = triangulate(C1, pts1, C2, pts2)
        if (P_trial[:,-1].all() > 0 and err<e):
            e = err
            M2_best = M2
            C2_best = C2
            P_best = P_trial
    print("Best Error : ",e)
    return M2_best, C2_best, P_best
```

Figure 11: Results for **q3_2_triangulate.py**

```
Optimization terminated successfully.
         Current function value: 0.000107
         Iterations: 8
         Function evaluations: 893
Best Error :  352.2302219560274
```

4 : 3D Visualisation

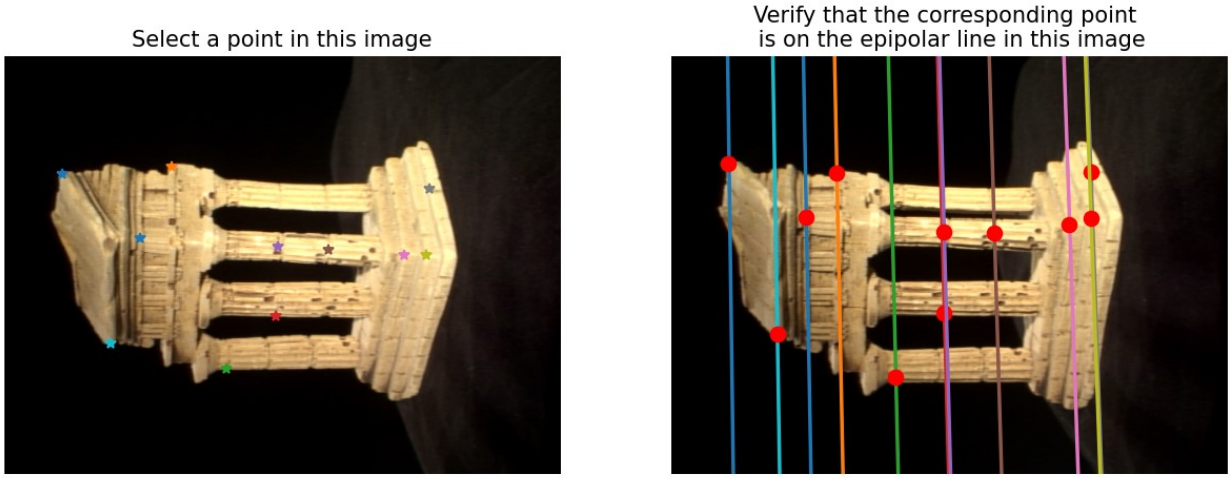
Question 4.1

Figure 12: Code for **q4_1_epipolar_correspondence.py**

```
def epipolarCorrespondence(im1, im2, F, x1, y1):
    x = np.asarray([int(x1),int(y1),1])
    line = F@x.T #search along this line
    window = 50 # varied #for 4.2 changed from 10
    c = window//2 #centre se dist

    patch = im1[y1-c:y1+c+1,x1-c:x1+c+1] #patch in image1
    #print(patch.shape)
    # gaussian - gfg
    ss = 2
    xx,yy = np.meshgrid(np.linspace(-ss,ss,window+1), np.linspace(-ss,ss,window+1)) #(-1,1) can be changed
    dist = (xx**2 + yy**2)**0.5         #print(dist.shape)
    sigma = 3
    gauss = np.exp(-((dist)**2 / (2.0*(sigma**2))))
    gauss = gauss/np.sum(gauss)
    #gauss = np.eye(gauss.shape[0])
    #print(gauss.shape)

    if len(patch.shape)>2 : #image has channels
        for i in range(patch.shape[-1]):
            img_result = gauss*patch[:,:,i]
    else:
        img_result = gauss*patch

    min_dist = 10000
    ## print(im2.shape[0]-window)
    for i in range(im2.shape[0]-window): #image goes columnwise
        y2 = i+c
        x2 = int(-(line[1]*y2+line[2])/line[0])
        patch2 = im2[y2-c:y2+c+1,x2-c:x2+c+1]
        ## print(patch2.shape, y2-c,y2+c+1,y2,c,i,i+c)
        if len(patch2.shape)>2 : #image2 has channels
            for j in range(patch2.shape[-1]):
                img2_result = gauss*patch2[:,:,j]
        else:
            img2_result = gauss*patch2

        distpoint = np.linalg.norm(np.asarray([x1-x2,y1-y2]))
        diff = np.linalg.norm(img2_result-img_result)
        if min_dist>diff and distpoint<50:
            best_x2 = x2
            best_y2 = y2
            min_dist = diff

    return best_x2,best_y2
```

The matrix **F** and points **pts1** & **pts2** have been saved to **q4_1.npz** (and submitted in the zip).

Figure 13: Results for epipolarMatchGUI in **q4_1_epipolar_correspondence.py**



Question 4.2

Figure 14: Code for **q4_2_visualize.py** - compute3d_pts and main

```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    N = temple_pts1.shape[0]
    K1, K2 = intrinsics['K1'], intrinsics['K2']
    temple_pts2 = np.zeros((N,2))

    for i in range(N):
        # corresponding points found
        temple_pts2[i,0], temple_pts2[i,1] = epipolarCorrespondence(im1,im2,F,temple_pts1[i,0],temple_pts1[i,1])

    M2,C2,P = findM2(F,temple_pts1,temple_pts2,intrinsics)
    #print(M2)
    return M2,C2,P
```

```
if __name__ == "__main__":

    temple_coords_path = np.load('data/templeCoords.npz')
    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]))
    #print(F)

    pts1_x = temple_coords_path["x1"]
    pts1_y = temple_coords_path["y1"]
    pts1 = np.hstack([pts1_x,pts1_y])
    M2,C2,P = compute3D_pts(pts1,intrinsics,F,im1,im2)
    # print(P.shape) - getting 288,3
    M1 = np.hstack((np.eye(3),np.zeros((3,1))))
    #print(M1)
    C1 = K1@M1

    np.savez("submission/q4_2.npz",F=F,M1=M1,C1=C1,M2=M2,C2=C2)

    # 3D plot - gfg
    fig = plt.figure(figsize = (10, 7))
    ax = plt.axes(projection ="3d")
    ticks_x = np.arange(-0.7, 0.7, 0.2)
    ax.set_xticks(ticks_x)
    ticks_y = np.arange(-0.45, 0.45, 0.2)
    ax.set_yticks(ticks_y)
    ticks_z = np.arange(3, 4.5, 0.2)
    ax.set_zticks(ticks_z)
    ax.set_xlim3d(-0.7,0.7)
    ax.set_ylim3d(-0.45,0.45)
    ax.set_zlim3d(3,4.5)

    ax.scatter3D(P[:,0], P[:,1], P[:,2], color = "blue")
    plt.title("q4_2_visualize")
    ax.view_init(90, 0) # yshows problem
    plt.show()
```

Figure 15: Results from Visualization

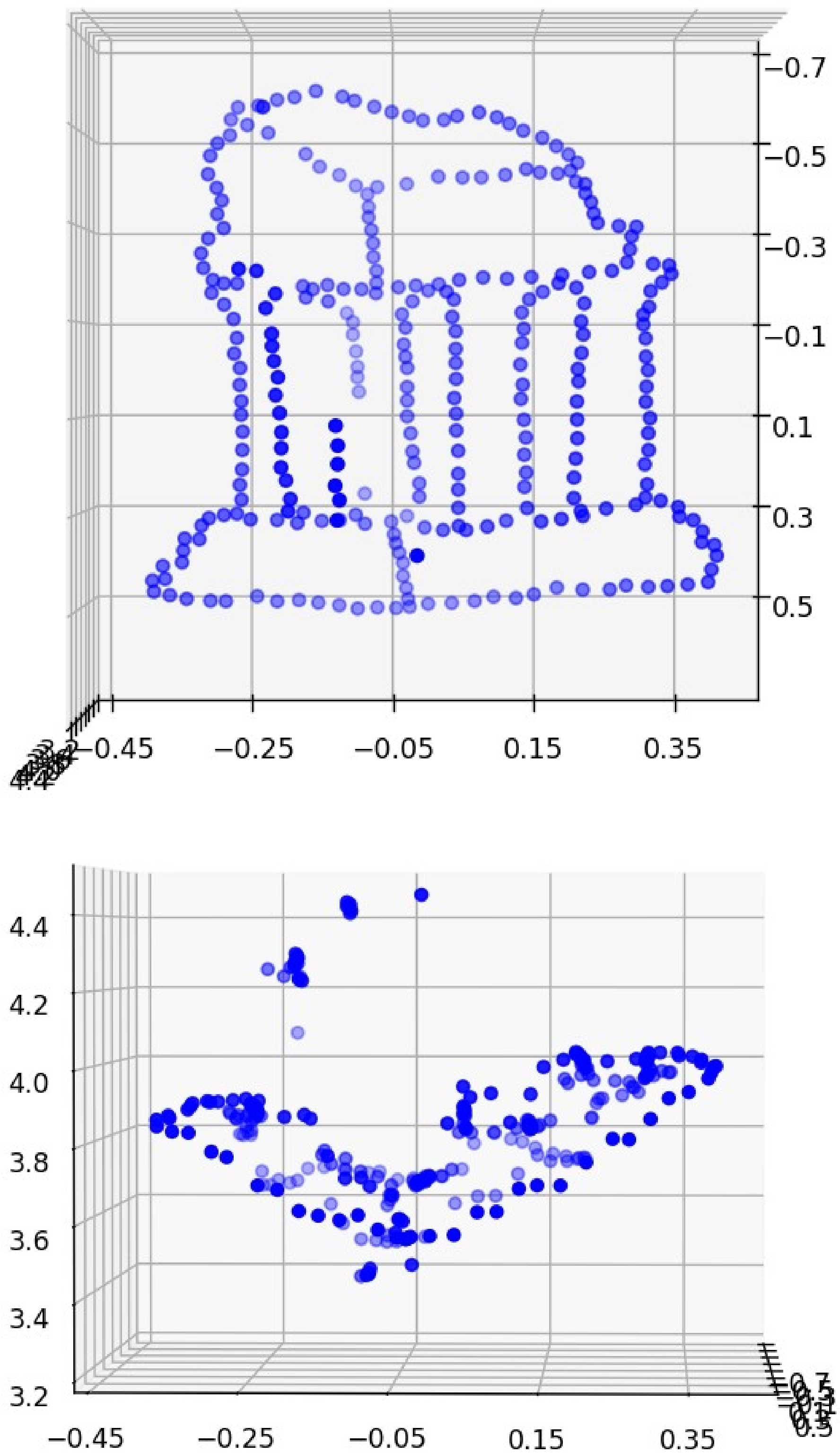
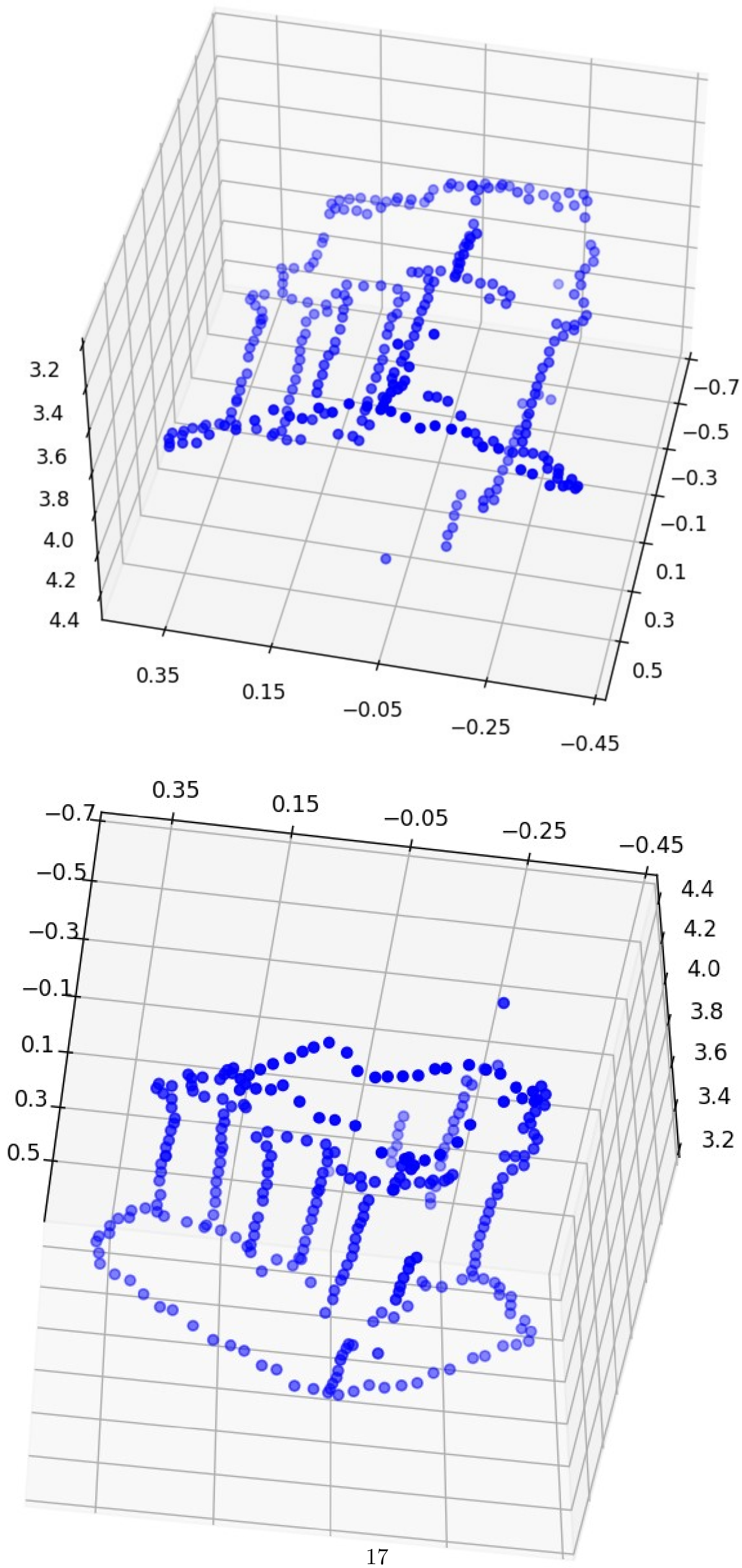


Figure 16: Results from Visualization



1 Question 5: Bundle Adjustment

Question 5.1: RANSAC

Figure 17: Code for **RANSAC**

```
def ransacF(pts1, pts2, M, nIters=1000, tol=10):
    N = pts1.shape[0]
    best_inliers = 0
    prev_inliers = 0
    best_F = None
    nIters = 100

    for i in range(nIters):
        temp_pts = np.random.choice(N, size=8,replace=False)
        pts1_ = pts1[temp_pts, :]
        pts2_ = pts2[temp_pts, :]
        F = eightpoint(pts1_, pts2_, M)
        #print(F.shape)
        dist = calc_epi_error(toHomogenous(pts1),toHomogenous(pts2), F)
        inliers_bool = (dist < tol).astype(bool)
        no_inlin = np.count_nonzero(inliers_bool)
        if(no_inlin > prev_inliers):
            best_inliers = inliers_bool
            best_F = F
            prev_inliers = no_inlin

    print((prev_inliers/N)*100) #to be 75%s
    return best_F, best_inliers
```

After iterations= 100 and tolerance= 10, **78%** of points are inliers. F is re-estimated with inliers for a better value. If the points were within the tolerance level, they were considered as inliers and saved.

Used gradient descent algorithm to minimise a scalar value. This is the sum of the euclidean distance between given points and projections.

Figure 18: F with noisy points (eightpoint) and re-estimated F

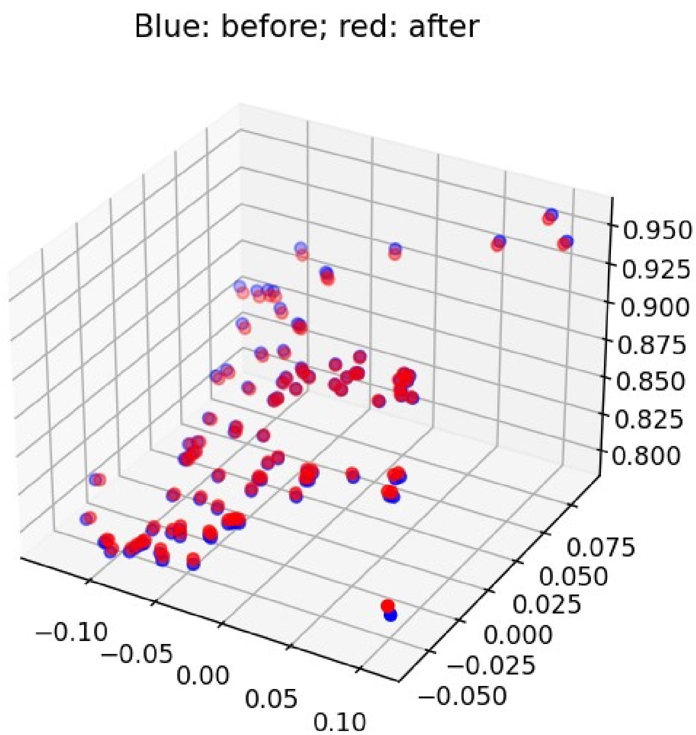
```
PS C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw4\hw4> python .\code\q5_bundle_adjustment.py
Optimization terminated successfully.
         Current function value: 2.507170
         Iterations: 22
         Function evaluations: 2764
Eight Point (noisy points
[[-1.62317569e-04  1.02911308e-03 -1.68105004e-01]
 [-1.14153928e-03  3.60765469e-04  2.41397505e-01]
 [ 2.77533519e-01 -3.76031803e-01  1.00000000e+00]]
```

Figure 19: F with noisy points (eightpoint) and re-estimated F

```
F reestimated from RANSAC
[[ 4.25607861e-07  5.26432462e-05 -2.03307590e-01]
 [-1.58347725e-05 -1.08682661e-06  7.57501495e-03]
 [ 1.94144352e-01 -1.09929774e-02  1.00000000e+00]]
```

The tolerance is reduced to - Hence, we get lesser percentage of inliers - 67%. Results below :

Figure 20: Result for tolerance = 2




```
F reestimated from RANSAC
[[ 2.15503110e-04 -7.18700048e-03 -8.25220174e+00]
 [ 7.73919574e-03  1.29018840e-04 -1.42373907e+00]
 [ 8.05787595e+00  1.43310780e+00  1.00000000e+00]]
Best Error :  1133.19385081236
Before 1133.1938508143126, After 45.78809549554861
```

When the tolerance is increased to 25 or 50 doesn't change the results from tolerance = 10. However, changing it to 100, crashes the program suggesting that a lesser error than 10000 wasn't found (as per my code). 96% of the points become inlier at this tolerance and the error is high.

Figure 21: Result for tolerance = 100

```
96.3963963963964
F reestimated from RANSAC
[[ 2.83009343e-08  1.15724383e-04 -2.53267247e-01]
 [-7.29714651e-05 -2.80606690e-06  2.11156832e-02]
 [ 2.41505750e-01 -2.31026797e-02  1.00000000e+00]]
Best Error :  10000
```

Reducing Iterations to 200 doesn't have any significant effect.

Question 5.2: Rodrigues Rotation

Figure 22: Code for **rodrigues**

```
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
```

Figure 23: Code for **inv_rodrigues**

```
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
```

Question 5.3: Bundle Adjustment

Figure 24: Code for **rodriguesResidual**

```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    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
```

Figure 25: Code for **Bundle Adjustment**

```
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    obj_start = obj_end = 0
    R2_init = M2_init[:, 0:3]
    t2_init = M2_init[:, 3]
    x_init = np.concatenate((P_init.flatten(), invRodrigues(R2_init).flatten(), t2_init.flatten()))

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

    obj_start = func(x_init)
    x_update = scipy.optimize.minimize(func,x_init,method = "CG").x #other method gave issues with size
    obj_end = func(x_update)
    N = p1.shape[0]
    P = x_update[:-6].reshape((N,3))
    R2 = rodrigues(x_update[-6:-3].reshape((3,)))
    t2 = x_update[-3:].reshape((3,1))
    M2 = np.hstack((R2, t2))

    return M2, P, obj_start, obj_end
```

Figure 26: **Plot_3D**

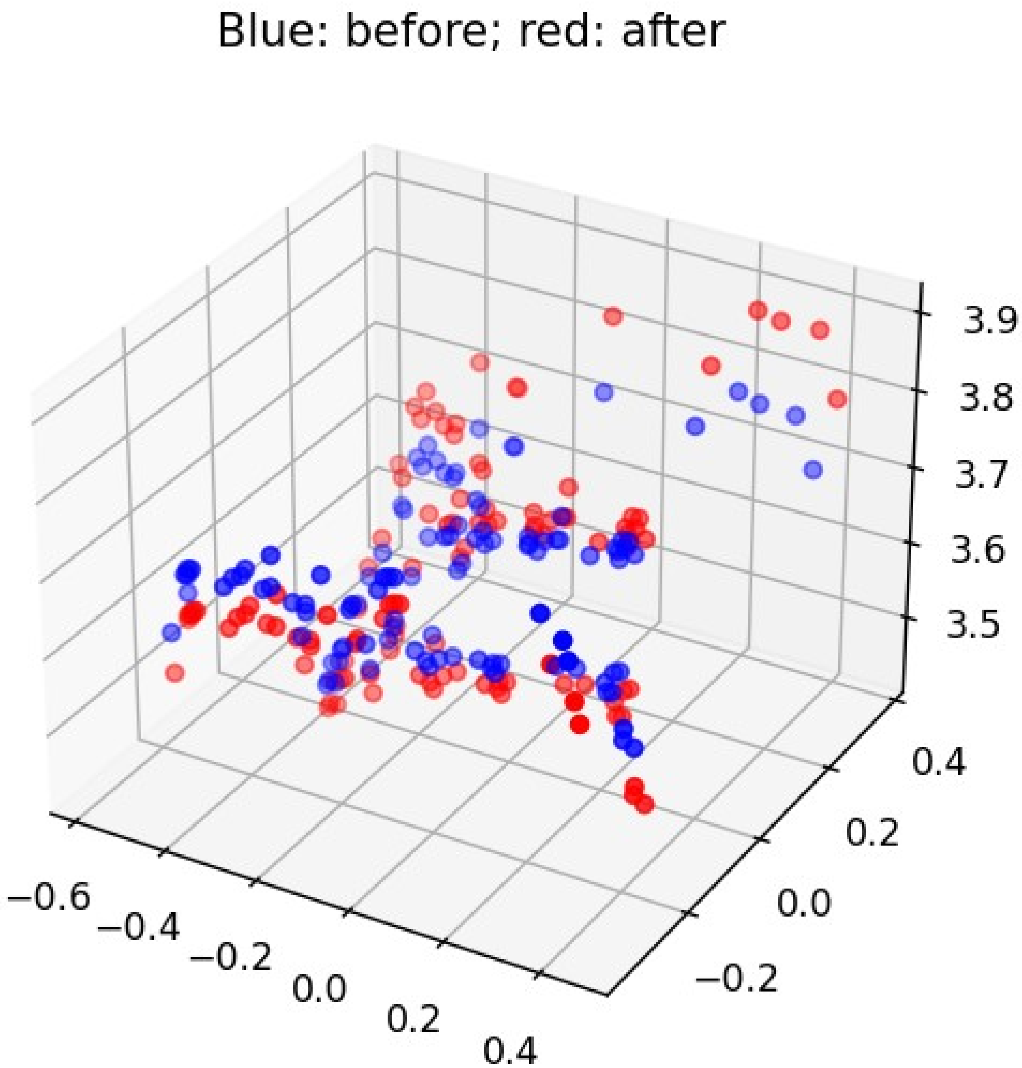


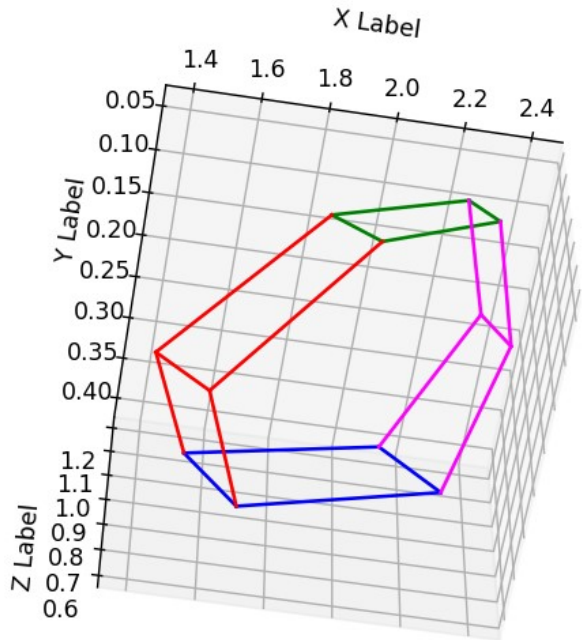
Figure 27: Error - Before and After

Before 6385.791297807882, After 10.91260579882069

The reprojection error, both before and after, shown above.

6.1: Multiview 3D Reconstruction

Figure 28: Result for 1 frame - Multiview Reconstruction



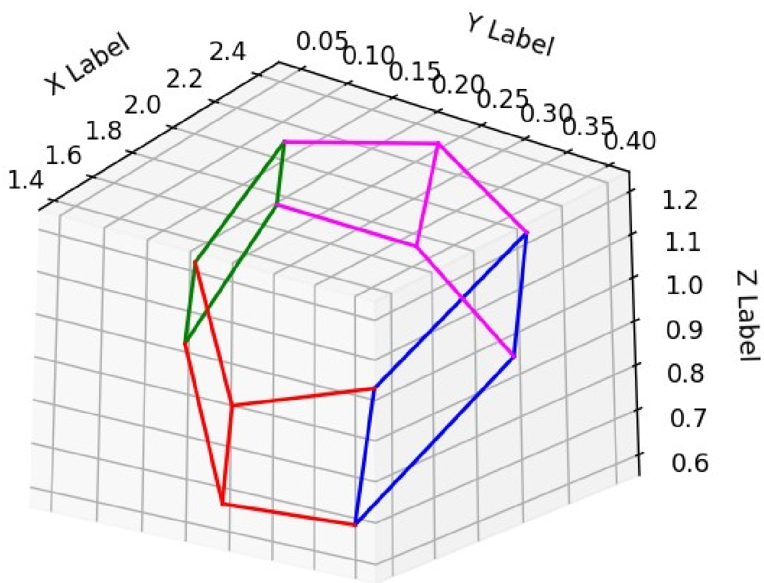


Figure 29: Result for 1 frame - Multiview Reconstruction

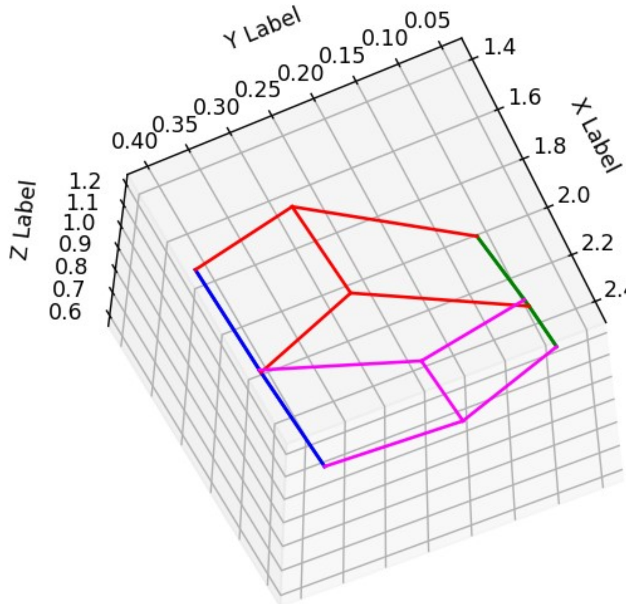


Figure 30: Code for - 6.1: MultiviewReconstruction

```python
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres = 100):
    N = pts1.shape[0]
    P = np.zeros((N,3))
    e = np.zeros(N)

    for i in range(N):

        if(pts1[i,2]<=pts2[i,2] and pts1[i,2]<=pts3[i,2]): #least confidence in camera 1
            Ptemp, etemp = triangulate(C2, np.expand_dims(pts2[i,:2],axis=0), C3, np.expand_dims(pts3[i,:2],axis=0))

        elif(pts2[i,2]<pts3[i,2] and pts2[i,2]<pts1[i,2]):
            Ptemp, etemp = triangulate(C3, np.expand_dims(pts3[i,:2],axis=0), C1, np.expand_dims(pts1[i,:2],axis=0))

        else:
            Ptemp, etemp = triangulate(C1, np.expand_dims(pts1[i,:2],axis=0), C2, np.expand_dims(pts2[i,:2],axis=0))

        #### print(Ptemp.shape)
        P[i,:] = Ptemp
        e[i] = etemp

    #print(P.shape)
    return P,e
```

Out of the 12 points given in 3 different images - I use the points from any two images such that the confidence of the 2 points individually is more than the third's. This increases the probability for us to get good estimates of the 3D construction.

6.2: Video

Figure 31: Result for Video - Multiview Reconstruction

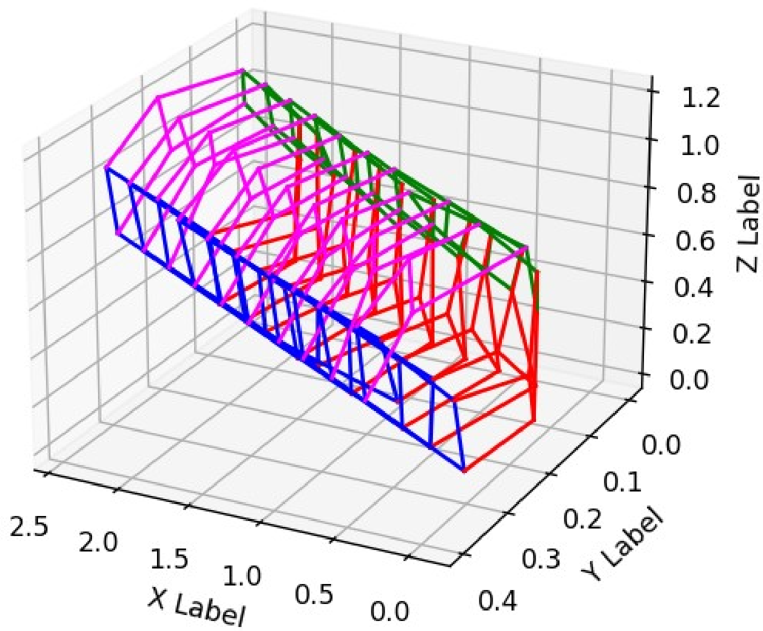


Figure 32: Code for Video - Multiview Reconstruction

```
def plot_3d_keypoint_video(pts_3d_video):
    fig = plt.figure()
    num_points = pts_3d_video.shape[1]
    ax = plt.axes(projection='3d')

    for i in range(pts_3d_video.shape[0]):
        for j in range(len(connections_3d)):
            index0, index1 = connections_3d[j]
            xline = [pts_3d_video[i,index0,0], pts_3d_video[i,index1,0]]
            yline = [pts_3d_video[i,index0,1], pts_3d_video[i,index1,1]]
            zline = [pts_3d_video[i,index0,2], pts_3d_video[i,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()
```