HOMEWORK 4: 3D RECONSTRUCTION

16-720A Computer Vision (Spring 2023)
https://canvas.cmu.edu/courses/32966

OUT: March 27th, 2023
DUE: April 12th, 2023 11:59 PM
Instructor: Deva Ramanan
TAs: Kangle Deng, Vidhi Jain, Xiaofeng Guo, Chung Hee Kim, Ingrid Navarro

Instructions/Hints

- Please refer to the course logistics page for information on the **Collaboration Policy** and **Late Submission Policy**.
- **Submitting your work:** There will be two submission slots for this homework on **Gradescope**: Written and Programming.
  - **Write-up.** For written problems such as short answer, multiple choice, derivations, proofs, or plots, we will be using the written submission slot. Please use this provided template. **We don't accept handwritten submissions.** Each answer should be completed in the boxes provided below the question. You are allowed to adjust the size of these boxes, but **make sure to link your answer to each question when submitting to Gradescope**. Otherwise, your submission will not be graded.
  - **Code.** You are also required to upload your code, which you wrote to solve this homework, to the Programming submission slot. Your code may be run by TAs so please make sure it is in a workable state. The assignment must be completed using Python 3.7 or newer. We recommend setting up a conda environment, but you are free to set up your environment however you like.
  - Regrade requests can be made after the homework grades are released, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted.
- **Start early!** This homework is difficult and may take a long time to complete.
- **Verify your implementation as you proceed.** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.
- **Q&A.** If you have any questions or need clarifications, please post in Slack or visit the TAs during office hours. Additionally, we provide a **FAQ** (section 8) with questions from previous semesters. Make sure you read it prior to starting your implementations.

Overview

In this assignment you will be implementing an algorithm to reconstruct a 3D point cloud from a pair of images taken at different angles. In Part I you will answer theory questions about 3D reconstruction. In Part II you will apply the 8-point algorithm and triangulation to find and visualize 3D locations of corresponding image points.

Part I

Theory

Before implementing our own 3D reconstruction, let's take a look at some simple theory questions that may arise. The answers to the below questions should be relatively short, consisting of a few lines of math and text (maybe a diagram if it helps your understanding).

Q1.1 [5 points] Suppose two cameras fixate on a point $\mathbf{x}$ (see Figure 1) in space such that their principal axes intersect at that point. Show that if the image coordinates are normalized so that the coordinate origin $(0, 0)$ coincides with the principal point, the $\mathbf{F}_{33}$ element of the fundamental matrix is zero.

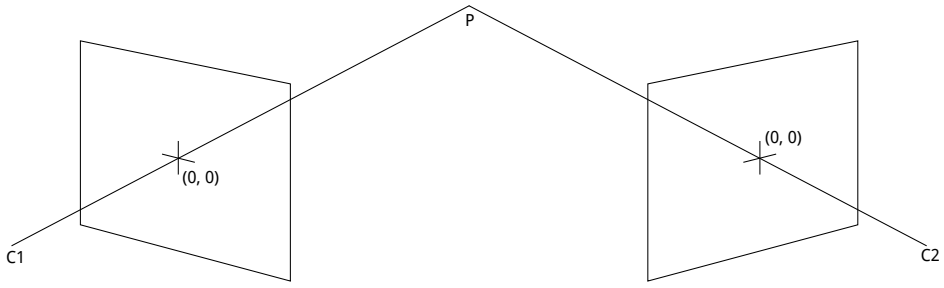

Figure 1: Figure for Q1.1. $C1$ and $C2$ are the optical centers. The principal axes intersect at point $\mathbf{w}$ (P in the figure).

Q1.1

Both principal point of two image planes coincide with coordinate origin (0, 0). Then the projected point $\vec{x}$ in the two image planes : $\vec{x}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ and $\vec{x}_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

From the property when points at origin :

$$\vec{x}_2^T F \vec{x}_1 = 0$$

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

Q1.2 [5 points] Consider the case of two cameras viewing an object such that the second camera differs from the first by a *pure translation* that is parallel to the x-axis. Show that the epipolar lines in the two cameras are also parallel to the x-axis. Backup your argument with relevant equations. You may assume both cameras have the same intrinsics.

Q1.2

For pure translation along x-axis, $R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $t = \begin{bmatrix} t_1 \\ 0 \\ 0 \end{bmatrix}$, Essential matrix $E = tR = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix}$

Let $\vec{x_1} = \begin{bmatrix} a_1 \\ a_2 \\ 1 \end{bmatrix}$, $\vec{x_2} = \begin{bmatrix} b_1 \\ b_2 \\ 1 \end{bmatrix}$, then $l_1^T = \vec{x_2}^T E$ and $l_2^T = \vec{x_1}^T E^T$

$\vec{l_1}^T = \begin{bmatrix} b_1 & b_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & t_1 & -b_2 t_1 \end{bmatrix}$; $\vec{l_2}^T = \begin{bmatrix} a_1 & a_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t_1 \\ 0 & -t_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -t_1 & a_2 t_1 \end{bmatrix}$

The epipolar line in the first camera is $t_1 y_1 - b_2 t_1 = 0$, in second camera is $-t_1 y_2 + a_2 t_1 = 0$. Since they don't contain x in both equation, the epipolar lines in the two cameras are also parallel to the x-axis.

Q1.3 [5 points] Suppose we have an inertial sensor which gives us the accurate positions ($\mathbf{R}_i$ and $\mathbf{t}_i$, the rotation matrix and translation vector) of the robot at time i. What will be the effective rotation ($\mathbf{R}_{rel}$) and translation ($\mathbf{t}_{rel}$) between two frames at different time stamps? Suppose the camera intrinsics ($\mathbf{K}$) are known, express the essential matrix ($\mathbf{E}$) and the fundamental matrix ($\mathbf{F}$) in terms of $\mathbf{K}$, $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$.

Q1.3

let 3D point be P, and let p_1 and p_2 be the corresponding point at time i and $i+1$, $p_1 = R_1 P + t_1 \Rightarrow P = R_1^{-1}(p_1 - t_1)$;

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

Thus the effective rotation and translation between two frames: $R_{rel} = R_2 R_1^{-1}$,

$t_{rel} = -R_2 R_1^{-1} t_1 + t_2 \Rightarrow E = t_{rel} \times R_{rel}$

$F = (K^{-1})^T E K^{-1} = (K^{-1})^T t_{rel} \times R_{rel} K^{-1}$

Part II

Practice

1 Overview

2 Fundamental Matrix Estimation

2.1 The Eight Point Algorithm

Q2.1 [10 points]

Q2.1

```python
def eightpoint(pts1, pts2, M):
    # scale the data by dividing each coordinate by M:
    pts1 = pts1 / M
    pts2 = pts2 / M
    x1 = pts1[:, 0]
    y1 = pts1[:, 1]
    x2 = pts2[:, 0]
    y2 = pts2[:, 1]

    # solve for least square solution
    A = np.vstack((x1*x2, x2*y1, x2, x1*y2, y1*y2, y2, x1, y1, np.ones(x1.
    shape))).T

    U, S, Vh = np.linalg.svd(A)
    f = Vh[-1, :]
    f = np.reshape(f, (3,3))

    # refine F
    F = refineF(f, pts1, pts2)

    # set the last singular val to be 0
    F = _singularize(F)

    # normalize using matrix T:
    T = np.array([[1.0 / M, 0, 0], [0, 1.0 / M, 0], [0, 0, 1]])

    # scale the data such that F_unnormalized = T.T @ F @T
    F = T.T @ F @ T
    F = F / F[2, 2]

    return F
```

Below, Figure 2 and Figure 3 show the example output and the recovered F.

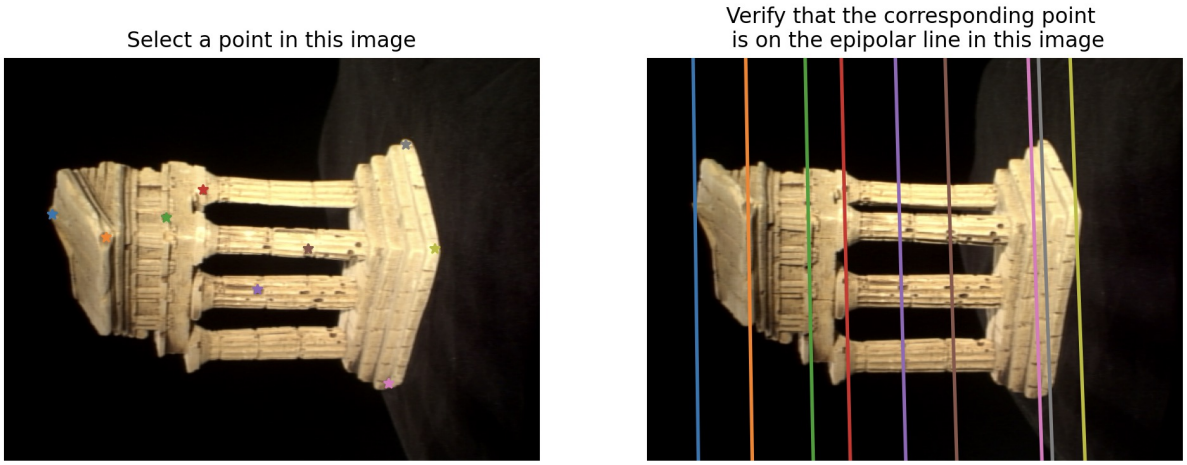

Figure 2: Example output of displayEpipolarF for Q2.1

```
Optimization terminated successfully.
         Current function value: 0.000107
         Iterations: 8
         Function evaluations: 818
[[-2.18962368e-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]]
saved
```

Figure 3: The matrix is the Recovered F

2.2 The Seven Point Algorithm

Q2.2 [15 points]

Q2.2

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

    Farray = []
    # ----- TODO -----
    # YOUR CODE HERE
    # scale the data by dividing each coordinate by M:
    pts1 = pts1 / M
    pts2 = pts2 / M
    x1 = pts1[:, 0]
    y1 = pts1[:, 1]
    x2 = pts2[:, 0]
    y2 = pts2[:, 1]

    # normalize using matrix T:
    T = np.array([[1.0 / M, 0, 0], [0, 1.0 / M, 0], [0, 0, 1]])

    A = np.vstack((x1 * x2, x2 * y1, x2, x1 * y2, y1 * y2, y2, x1, y1, np.ones
    (x1.shape))).T

    # solving for nullspace of A to get two F
    U, S, Vh = np.linalg.svd(A)
    f1 = Vh[-1, :]
    f2 = Vh[-2, :]
    f1 = np.reshape(f1, (3, 3))
    f2 = np.reshape(f2, (3, 3))

    # find F that meets the singularity constraint: det(a*F1 + (1-a)*F2) = 0
    # get the coefficient of the polynomial
    Ka = np.array([[1, 0.1, 0.1**2, 0.1**3], [1, 0.3, 0.3**2, 0.3**3],
                   [1, 0.6, 0.6**2, 0.6**3], [1, 0.9, 0.9**2, 0.9**3]])
    B = np.array([np.linalg.det(0.1 * f1 + (1 - 0.1) * f2), np.linalg.det(0.3
    * f1 + (1 - 0.3) * f2),
                  np.linalg.det(0.6 * f1 + (1 - 0.6) * f2), np.linalg.det(0.9
    * f1 + (1 - 0.9) * f2)]).T
    # k = [k0, k1, k2, k3].T, where k3a^3+k2a^2+k1a+k0
    k = np.linalg.inv(Ka) @ B
    k = k[::-1]
    alpha = np.roots(k)

    Farray = [a*f1+(1-a)*f2 for a in k]
    # refind F
    Farray = [refineF(F, pts1, pts2) for F in Farray]

    # scale the data such that F_unnormalized = T.T @ F @T
    Farray =[T.T @ F @ T for F in Farray]
    Farray = [F / F[2,2] for F in Farray]

    return Farray
```

The error I get is Error: 0.5841684840425984. Figure 4 and 5 shows the image of the example output of deisplayEpipolarF and the recovered F.

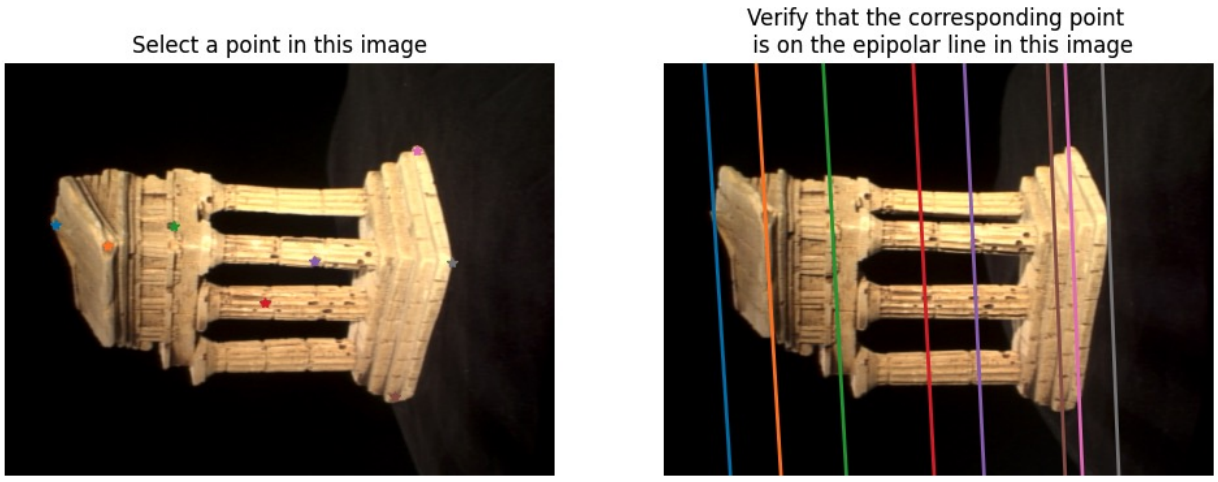

Figure 4: Example output of displayEpipolarF for Q2.1

```
F:  [[ 2.12835761e-06  4.06800050e-05 -3.14833866e-01]
 [ 5.16968957e-06 -2.05198247e-06  1.36764134e-02]
 [ 3.03048215e-01 -1.65335311e-02  1.00000000e+00]]
```

Figure 5: The matrix is the Recovered F for q2.2

3 Metric Reconstruction

Q3.1 [5 points]

Q3.1

```python
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE
    E = K2.T @ F @ K1
    E = E/ E[2,2]
    return E
```

The estimated E is shown in Figure 6.

```
E:  [[-3.36615965e+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]]
```

Figure 6: The matrix is the estimated E for q3.1

Q3.2 [10 points]

Q3.2

```python
def triangulate(C1, pts1, C2, pts2):
    n = pts1.shape[0]
    P = np.zeros((n, 3))

    # form A using the corresponding points from pts1, pts2 C1, and C2
    for i in range(n):
        x1 = pts1[i, 0]
        y1 = pts1[i, 1]
        x2 = pts2[i, 0]
        y2 = pts2[i, 1]
        A1 = y1*C1[2, :] - C1[1, :]
        A2 = C1[0, :] - x1*C1[2, :]
        A3 = y2*C2[2, :] - C2[1, :]
        A4 = C2[0, :] - x2*C2[2, :]
        A = np.vstack((A1, A2, A3, A4))
        U, S, Vh = np.linalg.svd(A)
        po = Vh[-1, :]
        P[i, :] = po[0:3] / po[3]

    # project to 2D points:
    P_homo = np.hstack((P, np.ones((n, 1))))
    err = 0

    # Keep track of the 3D points and projection error, and continue to next
    point
    for i in range(n):
        proj1 = np.dot(C1, np.transpose(P_homo[i, :]))
        proj2 = np.dot(C2, np.transpose(P_homo[i, :]))

        proj1_norm = np.transpose(proj1[:2]/proj1[2])
        proj2_norm = np.transpose(proj2[:2]/proj2[2])

        # compute error
        err += np.sum((pts1[i]-proj1_norm)**2 + (pts2[i]-proj2_norm)**2)

    return P, err
```

Figure 7 shows the expression for question 3.2.

Q3.2

$\vec{w_i} = [x_i, y_i, z_i, 1]^T$, $\vec{x_{1i}} = [x_{1i}, y_{1i}, 1]^T$, $\vec{x_{2i}} = [x_{2i}, y_{2i}, 1]^T$

$C_1 = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$, $C_2 = \begin{bmatrix} c'_{11} & c'_{12} & c'_{13} \\ c'_{21} & c'_{22} & c'_{23} \\ c'_{31} & c'_{32} & c'_{33} \end{bmatrix}$, we have $\vec{x_1} = \alpha_1 C_1 \vec{w}$

$\vec{x_1} = \alpha_1 \begin{bmatrix} - & c_{c1} & - \\ - & c_{c2} & - \\ - & c_{c3} & - \end{bmatrix} \begin{bmatrix} | \\ \bar{X} \\ | \end{bmatrix} = \alpha_1 \begin{bmatrix} c_{c1}\bar{X} \\ c_{c2}\bar{X} \\ c_{c3}\bar{X} \end{bmatrix}$ $\vec{x} \times \vec{x_1} = 0 \Rightarrow \begin{bmatrix} y_1 c_{c3} - c_{c2} \\ c_{c1} - x_1 c_{c3} \\ x_1 c_{c1} - y c_{c1} \end{bmatrix} \bar{X} = 0$

One 2D to 3D point correspond 2 equation, thus we have: $A_1 = y_1 c_{c3} - c_{c2}$, $A_2 = c_{c1} - x_1 c_{c3}$, and thus A_3 can be written as: $A_3 = y_2 c'_{c3} - c'_{c2}$, $A_4 = c'_{c1} - x_2 c'_{c3}$

$\Rightarrow A = [A_1, A_2, A_3, A_4]^T$

Figure 7: The Ai derivation and expression for q3.2

Q3.3 [10 points]

Q3.3

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

    # C1 for camera 1
    M1 = np.array([[1.0, 0, 0, 0],
                   [0, 1.0, 0, 0],
                   [0, 0, 1.0, 0]])
    C1 = K1 @ M1

    # Obtain the 4 candidates of M2s
    M2s = camera2(E)

    # Obtain the correct M2
    for i in range(M2s.shape[-1]):
        C2_tmp = K2 @ M2s[:, :, i]
        p, err = triangulate(C1, pts1, C2_tmp, pts2)

        if np.min(p[:, 2]) > 0:
            M2 = M2s[:, :, i]
            P = p
            break

    C2 = K2 @ M2
    # Save the correct M2, C2, and P to q3_3.npz
    np.savez(filename, M2=M2, C2=C2, P=P)

    return M2, C2, P
```

4 3D Visualization

Q4.1 [15 points]

Q4.1

```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # ----- TODO -----
    # YOUR CODE HERE
    # obtain the epipolar line
    P = np.array([x1, y1, 1])

    epi_line = F @ P
    epi_line = epi_line / np.linalg.norm(epi_line)
    a, b, c = epi_line

    half_window = 28
    min_err = 1e14

    x1 = int(x1)
    y1 = int(y1)
    h2, w2, _ = im2.shape

    # Get the patch around the pixel in image1
    patch1 = im1[(y1 - half_window): (y1 + half_window + 1), (x1 - half_window
    ): (x1 + half_window + 1)]

    for i in range(y1 - half_window, y1 + half_window):
        x2_tmp = int((-b * i - c) / a)

        if (x2_tmp - half_window) > 0 and (x2_tmp + half_window + 1) < w2 and
    (i - half_window) > 0 and (i + half_window + 1) < h2:
            patch2 = im2[(i - half_window): (i + half_window + 1), (x2_tmp -
    half_window): (x2_tmp + half_window + 1)]

            for j in range(0, patch2.shape[2]):
                dist = patch1[:, :, j] - patch2[:, :, j]
                error = np.linalg.norm(dist)
                # find the best matches
                if error < min_err:
                    min_err = error
                    x2 = x2_tmp
                    y2 = i

    return x2, y2
```

Figure 8 shows the screenshot of epipolarMatchGUI with some detected correspondences.

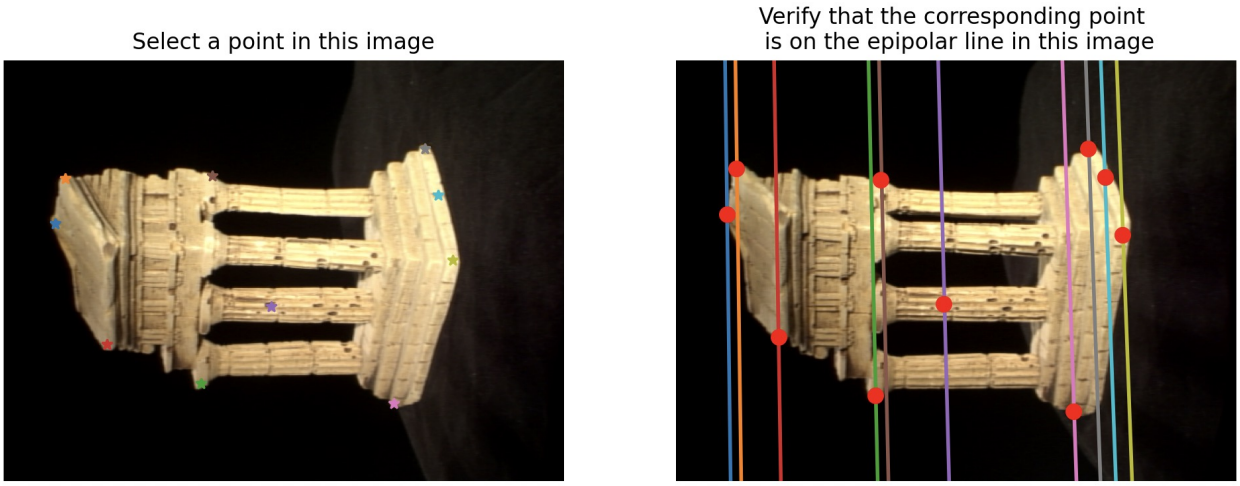

Figure 8: The screenshot of epipolarMatchGUI for q4.1

Q4.2 [10 points]

Q4.2

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

    # ----- TODO -----
    # YOUR CODE HERE
    # get the hand-selected points
    K1, K2 = intrinsics['K1'], intrinsics['K2']
    M1 = np.array([[1.0, 0, 0, 0],
                   [0, 1.0, 0, 0],
                   [0, 0, 1.0, 0]])
    C1 = K1 @ M1
    x1_t = temple_pts1[:, 0]
    y1_t = temple_pts1[:, 1]

    x1 = np.zeros((x1_t.shape[0], 1))
    y1 = np.zeros((x1_t.shape[0], 1))
    x2 = np.zeros((x1_t.shape[0], 1))
    y2 = np.zeros((y1_t.shape[0], 1))

    for i in range(x1_t.shape[0]):
        x2_t, y2_t = epipolarCorrespondence(im1, im2, F, x1_t[i], y1_t[i])
        x1[i, :] = x1_t[i]
        y1[i, :] = y1_t[i]
        x2[i, :] = x2_t
        y2[i, :] = y2_t

    p1 = np.hstack((x1, y1))
    p2 = np.hstack((x2, y2))
    M2, C2, P = findM2(F, p1, p2, intrinsics)

    np.savez('q4_2.npz', F = F, M1=M1, M2=M2, C1=C1, C2=C2)

    return P
```

Figure 9 shows the 3D visualization.

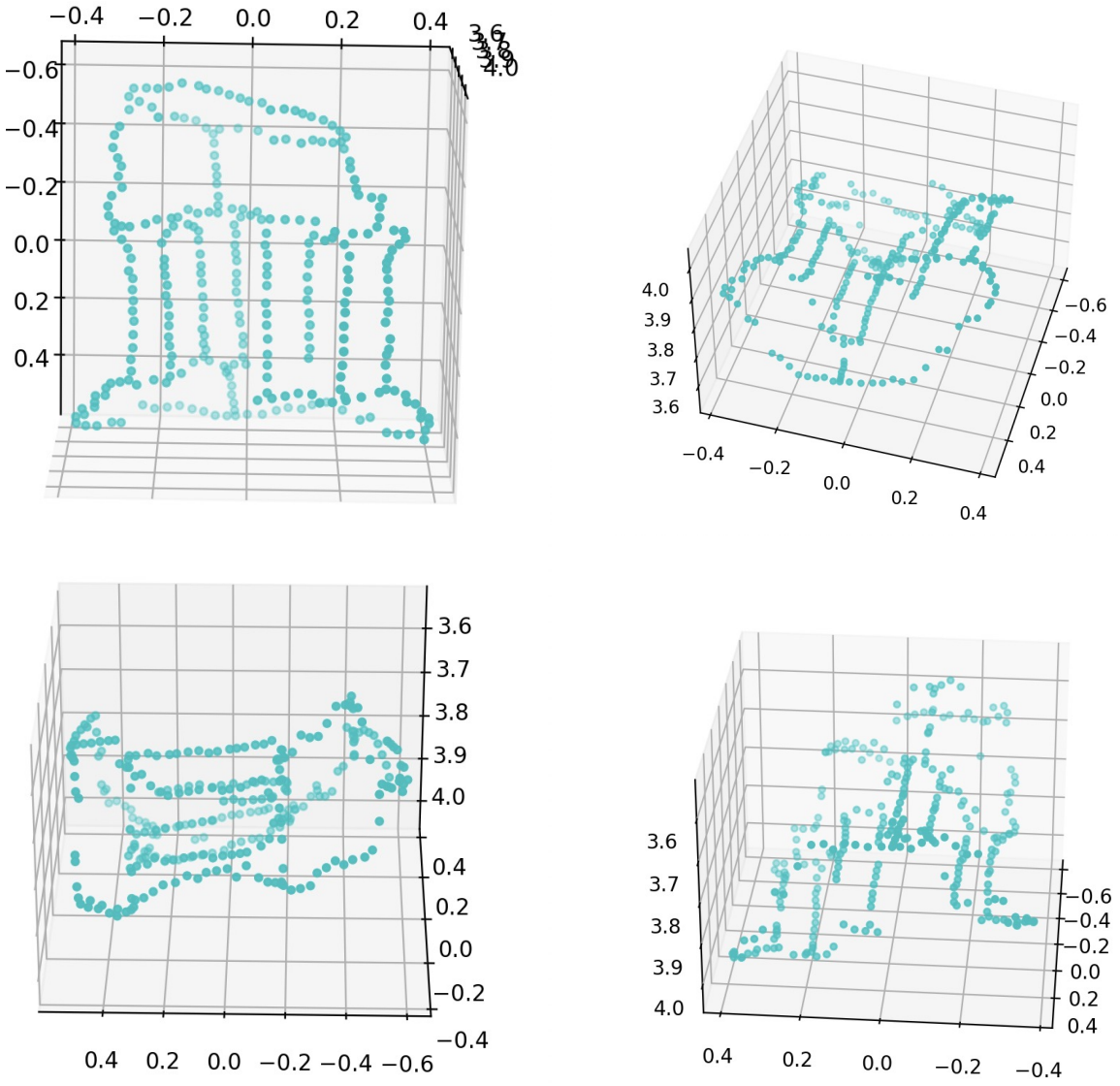

Figure 9: 3D visualization outline the temple for Q4.2

5 Bundle Adjustment

Q5.1 RANSAC for Fundamental Matrix Recovery [15 points]

Q5.1

```python
def ransacF(pts1, pts2, M, nIters=1000, tol=1):
    N = pts1.shape[0]

    pts1_homo = np.vstack((pts1.T, np.ones((1, N))))
    pts2_homo = np.vstack((pts2.T, np.ones((1, N))))

    # inliers is a vector of length N with a 1 at those matches, 0 elsewhere
    inliers = np.zeros(N)
    best_num_inliers = 0

    # Use the seven point alogrithm to estimate the fundamental matrix as done
     in q1
    for i in range(nIters):
        rand_idx = np.random.choice(pts1.shape[0], 7, replace=False)
        Farray = sevenpoint(pts1[rand_idx, :], pts2[rand_idx, :], M)

        # Choose the resulting F that has the most number of inliers
        for f in Farray:
            # Calculate epiploar line for pts1
            epi_line = f @ pts1_homo

            # calculate the distance from each point in pts2 to the
    corresponding epipolar line
            distance = abs(np.sum(epi_line * pts2_homo, axis=0)) / np.sqrt(
    epi_line[0,:] ** 2 + epi_line[1,:] ** 2)

            # Count inliers
            num_inliers = np.sum(distance < tol)

            if num_inliers > best_num_inliers:
                best_num_inliers = num_inliers
                inliers = distance < tol
                F = f

    return F, inliers
```

Figures 10 and 11 show the comparison of RANSAC with the result of the eightpoint when ran on the noisy correspondences. From the result, we can clearly see that the RANSAC is much more robust than eightpoint algorithm when having noisy correspondences.

The error metric is to calculate the distance between points in pts2 to the corresponding epipolar line and the points that count as inliers are determined as the distance is smaller than the tolerance.

The effect of the iteration "nIters" will help the model to have a much more accurate result. Increasing the number of iterations will increase the chance of finding the best model, which will result in a more accurate fundamental matrix. Varying the inlier tolerance will also affect the accuracy of the estimated fundamental matrix. Decreasing the inlier tolerance will increase the number of inliers, which will result in a more accurate fundamental matrix. Figure 12 shows the result of RANSAC when tol = 10 (where Figure 10 has a tol=1).

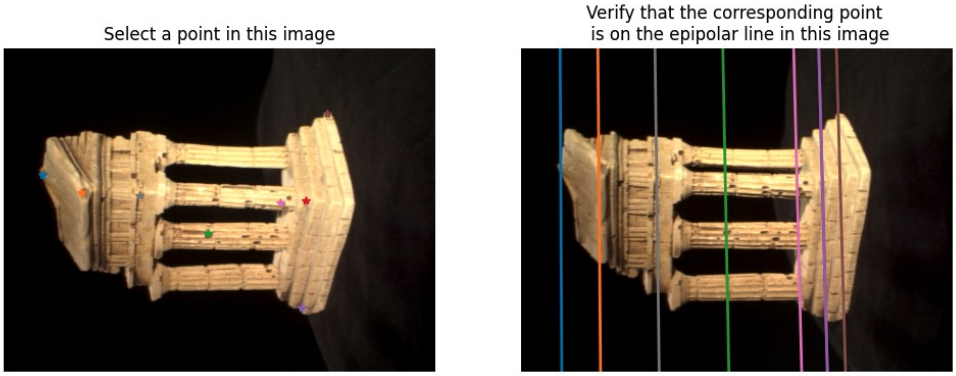

Figure 10: The result of RANSAC with tol =1, nIters = 1000 for q5.1

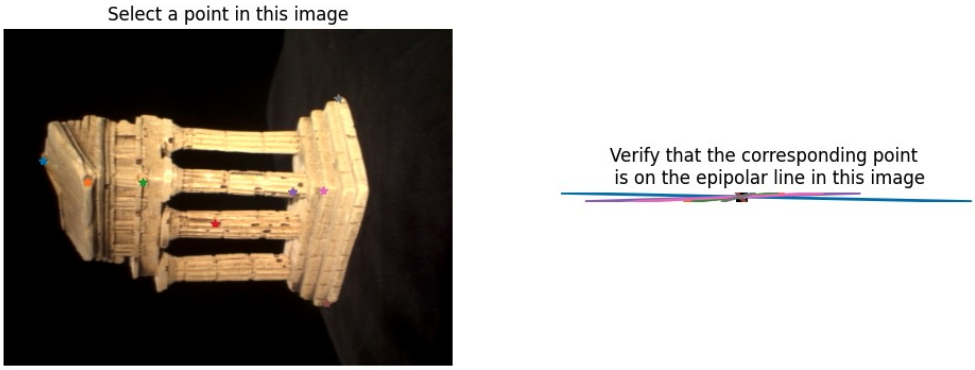

Figure 11: The result of eightpoint for q5.1

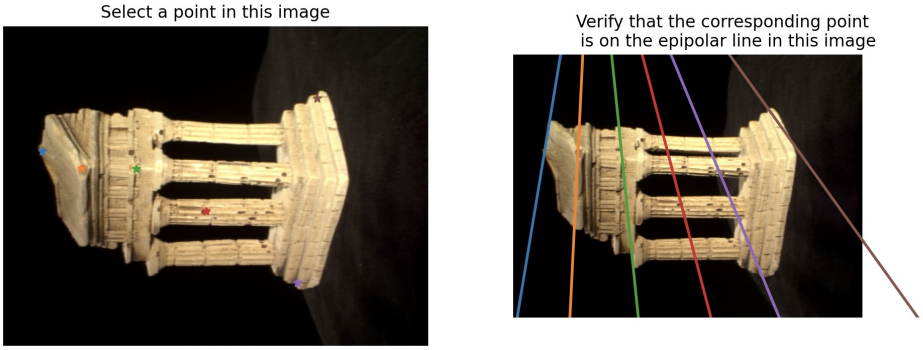

Figure 12: The result of RANSAC with tol = 10, nIters = 1000 for q5.1

Q5.2 Rodrigues and Invsere Rodrigues [15 points]

Q5.2

```python
def rodrigues(r):
    theta = np.linalg.norm(r)
    if theta == 0:
        return np.eye(3)

    u = r / theta
    uut = np.outer(u, u.T)
    ux = np.array([[0, -u[2], u[1]],
                   [u[2], 0, -u[0]],
                   [-u[1], u[0], 0]])
    # R = I cos(theta) + (1-cos(theta))uu.T + ux * sin(theta)
    R = np.eye(3) * np.cos(theta) + (1 - np.cos(theta)) * uut + ux * np.sin(
    theta)

    return R
```

Q5.2

```python
def invRodrigues(R):
    A = (R - R.T) / 2

    a32 = A[2, 1]
    a13 = A[0, 2]
    a21 = A[1, 0]

    rh = np.vstack((a32, a13, a21))
    s = np.linalg.norm(rh)
    r11 = R[0, 0]
    r22 = R[1, 1]
    r33 = R[2, 2]

    c = (r11 + r22 + r33 - 1) / 2

    if s == 0. and c == 1.:
        # vec_r = 0
        r = np.zeros((3, 1))

    elif s == 0. and c == -1.:
        # let v = a nonzero column of R+I
        v_tmp = R + np.eye(3)
        for i in range(3):
            if np.sum(v_tmp[:, i]) != 0:
                v = v_tmp[:, i]
                break

        u = v / np.linalg.norm(v)
        func = u * np.pi
        if np.linalg.norm(func) == np.pi and ((func[0, 0] == 0 and func[1, 0] == 0 and func[2, 0] < 0)
                                              or (func[0, 0] == 0 and func[1, 0] < 0)
                                              or (func[0, 0] < 0)):
            r = -func
        else:
            r = func

    else:
        u = rh / s
        theta = np.arctan2(float(s), float(c))
        r = u * theta

    return r.flatten()
```

Q5.3 Bundle Adjustment [10 points]

Q5.3

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

    N = math.floor(P.shape[0]/3)

    P = np.reshape(P, (N, 3))
    P_homo = np.vstack((P.T, np.ones((1, P.shape[0]))))

    # r2 (in the Rodrigues vector form)
    R2 = rodrigues(r2)

    # r2 and t2 are associated with the projection matrix M2: 3x4
    t2 = np.reshape(t2, (3, 1))
    M2 = np.hstack((R2, t2))

    C1 = K1 @ M1
    C2 = K2 @ M2

    x1_homo = C1 @ P_homo
    x2_homo = C2 @ P_homo

    p1_hat = (x1_homo[:2, :] / x1_homo[2, :]).T
    p2_hat = (x2_homo[:2, :] / x2_homo[2, :]).T

    # Residuals are the difference between original image projections and
    estimated projections.
    residuals = np.concatenate([(p1 - p1_hat).reshape([-1]), (p2 - p2_hat).
    reshape([-1])])

    return residuals
```

Q5.3

```python
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):

    obj_start = obj_end = 0

    # extract rotation and translation from M2_init
    R2_init = M2_init[:, :3],
    t2_init = M2_init[:, 3]

    # use invR to transform the rotation
    r2_init = invRodrigues(R2_init).reshape([-1])

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

    p = P_init.flatten()
    r2 = r2_init.flatten()
    t2 = t2_init.flatten()

    # concatenate x with translation and 3D points
    x0 = []
    x0 = np.append(x0, p)
    x0 = np.append(x0, r2)
    x0 = np.append(x0, t2)

    # optimzie for the best extrinsic matrix and 3D points
    x_opt, _ = scipy.optimize.leastsq(fun, x0)

    # decompose back to rotation, translation and 3D point
    P2 = x_opt[:-6]
    N = math.floor(P2.shape[0] / 3)
    P = np.reshape(P2, (N, 3))

    r2 = x_opt[-6:-3]
    t2 = x_opt[-3:]
    t2 = np.reshape(t2, (3, 1))

    R2 = rodrigues(r2)
    M2 = np.hstack((R2, t2))

    return M2, P, obj_start, obj_end
```

Figure 13 shows the result of the original 3D points and the optimized points, where blue is the optimized point and red is the initial point. The reprojection error with initial M2 and w is 2592.401817158513, the optimized error is 7.7443.

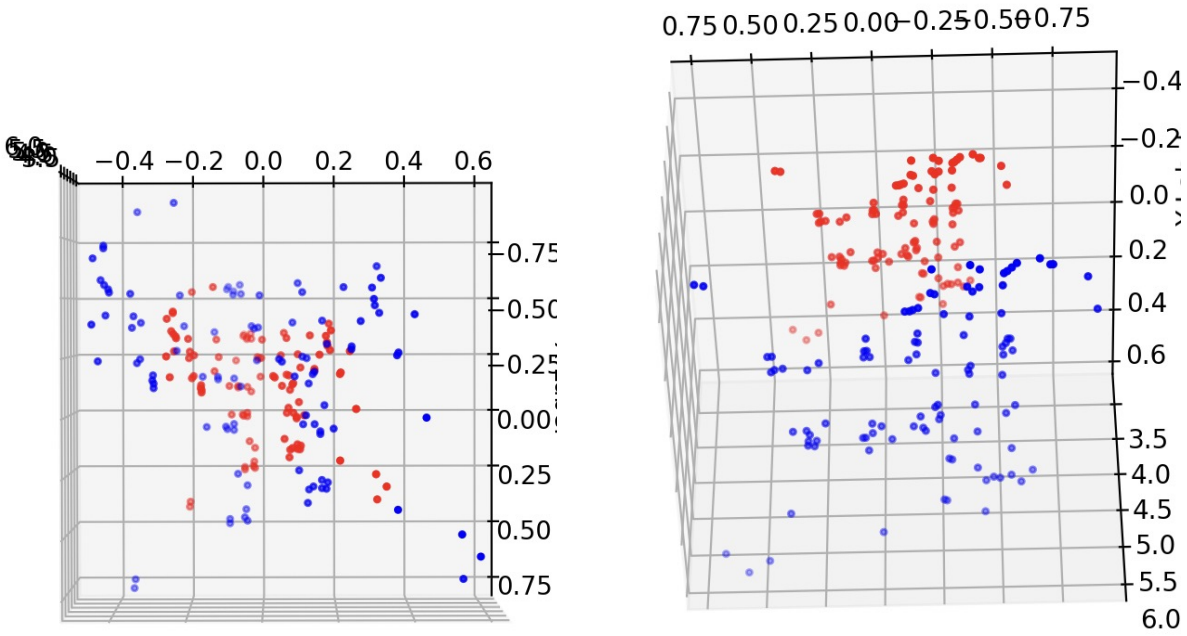

Figure 13: Visulization of 3D points for noisy correspondences before and after bundle adjustment for Q5.3

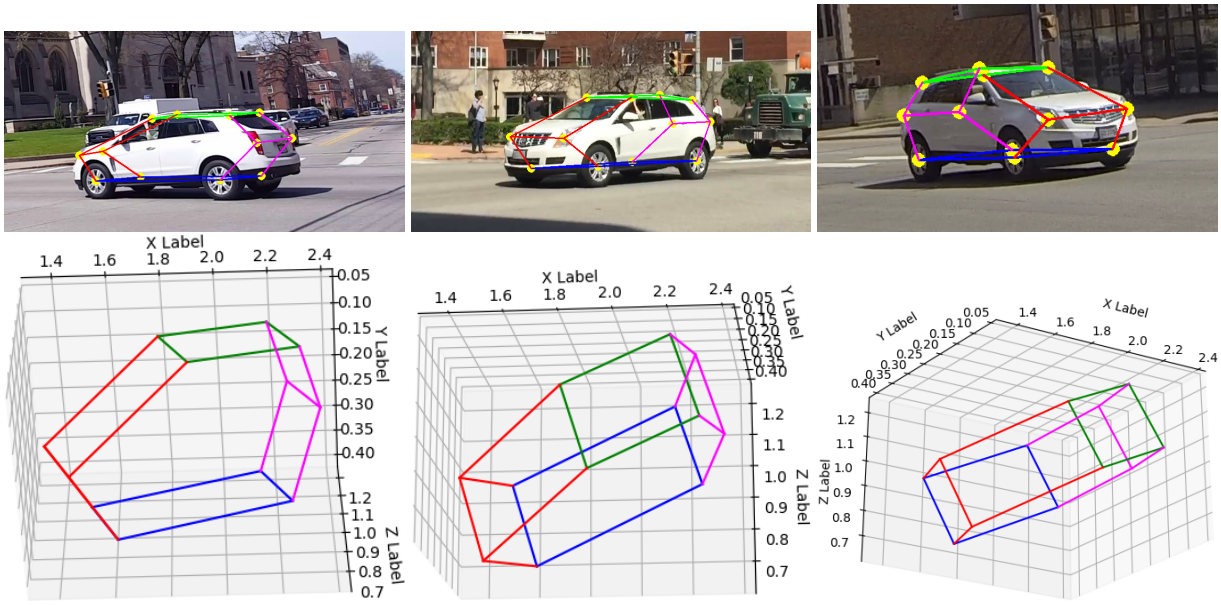

Figure 14: An example detections on the top and the reconstructions from multiple views

6 Multiview Keypoint Reconstruction (Extra Credit)

You will use multi-view capture of moving vehicles and reconstruct the motion of a car. The first part of the problem will be using a single time instance capture from three views (Figure 14 Top) and reconstruct vehicle keypoints and render from multiple views (Figure 14 Bottom). Make use of `q6_ec_multiview_reconstruction.py` file and `data/q6` folder contains the images.

Q6.1 [Extra Credit - 10 points] Write a function to compute the 3D keypoint locations `P` given the 2D part detections `pts1`, `pts2` and `pts3` and the camera projection matrices `C1`, `C2`, `C3`. The camera matrices are given in the numpy files.

```
[P, err] = MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres)
```

The 2D part detections (`pts`) are computed using a neural network[1] and correspond to different locations on a car like the wheels, headlights etc. The third column in `pts` is the confidence of localization of the keypoints. Higher confidence value represents more accurate localization of the keypoint in 2D. To visualize the 2D detections run `visualize_keypoints(image, pts, Thres)` helper function. `Thres` is defined as the confidence threshold of the 2D detected keypoints. The camera matrices (`C`) are computed by running an SFM from multiple views and are given in the numpy files with the 2D locations. By varying confidence threshold `Thres` (.i.e. considering only the points above the threshold), we get different reconstruction and accuracy. Try varying the thresholds and analyze its effects on the accuracy of the reconstruction. Save the best reconstruction (the 3D locations of the parts) from these parameters into a `q6_1.npz` file.

Hint: You can modify the triangulation function to take three views as input. After you do the threshold lets say m points lie above the threshold and n points lie below the threshold. Now your task is to use these m good points to compute the reconstruction. For each 3D location use two view or three view triangulation for intialization based on visibility after thresholding.

[1] Code Used For Detection and Reconstruction

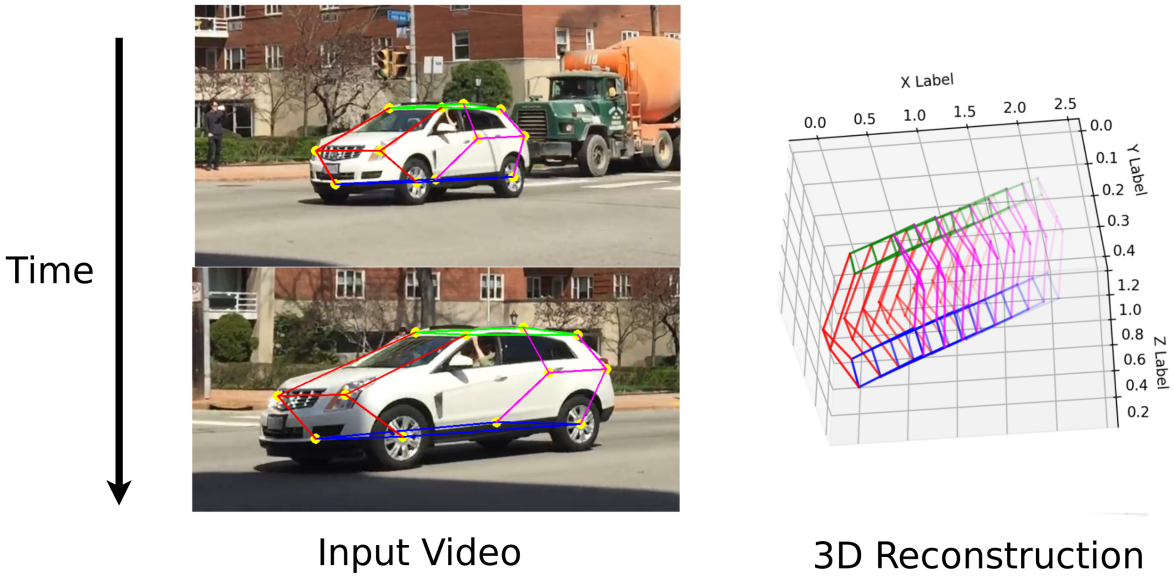

Figure 15: Spatiotemporal reconstruction of the car (right) with the projections at two different time instances in a single view(left)

In your write-up:

- Describe the method you used to compute the 3D locations.
- Include an image of the Reconstructed 3D points with the points connected using the helper function `plot_3d_keypoint(P)` with the reprojection error.
- Include the code snippets `MultiviewReconstruction` in your write-up.

Q6.1

Q6.2 [Extra Credit - 10 points] From the previous question you have done a 3D reconstruction at a time instance. Now you are going to iteratively repeat the process over time and compute a spatio temporal reconstruction of the car. The images in the `data/q6` folder shows the motion of the car at an intersection captured from multiple views. The images are given as `(cam1_time0.jpg, ..., cam1_time9.jpg)` for camera 1 and `(cam2_time0.jpg, ..., cam2_time9.jpg)` for camera2 and `(cam3_time0.jpg, ..., cam3_time9.jpg)` for camera3. The corresponding detections and camera matrices are given in `(time0.npz, ..., time9.npz)`. Use the above details and compute

the spatio temporal reconstruction of the car for all 10 time instances and plot them by completing the `plot_3d_keypoint_video` function. A sample plot with the first and last time instance reconstuction of the car with the reprojections shown in the Figure 15.

In your write-up:

- Plot the spatio-temporal reconstruction of the car for the 10 timesteps.
- Include the code snippets `plot_3d_keypoint_video` in your write-up.

Q6.2

7 Deliverables

The assignment (code and write-up) should be submitted to Gradescope. The write-up should be named `<AndrewId>_hw4.pdf` and the code should be a zip named `<AndrewId>_hw4.zip`. ***Please make sure that you assign the location of answers to each questions on Gradescope.*** The zip should have the following files in the structure defined below. (Note: Neglecting to follow the submission structure will incur a huge score penalty!).

- `<AndrewId>_hw4.pdf`: your write-up.
- `q2_1_eightpoint.py`: script for Q2.1.
- `q2_2_sevenpoint.py`: script for Q2.2.
- `q3_1_essential_matrix.py`: script for Q3.1.
- `q3_2_triangulate.py`: script for Q3.2.
- `q4_1_epipolar_correspondence.py`: script for Q4.1.
- `q4_2_visualize.py`: script for Q4.2.
- `q5_bundle_adjustment.py`: script for Q5.
- `q6_ec_multiview_reconstruction.py`: script for (extra-credit) Q6.
- `helper.py`: helper functions.
- `q2_1.npz`: file with output of Q2.1.
- `q2_2.npz`: file with output of Q2.2.

- `q3_1.npz`: file with output of Q3.1.
- `q3_3.npz`: file with output of Q3.3.
- `q4_1.npz`: file with output of Q4.1.
- `q4_2.npz`: file with output of Q4.2.
- `q6_1.npz`: (extra-credit) file with output of Q6.1.

***Do not include the data directory in your submission.**

8 FAQs

Credits: Paul Nadan

Q2.1: Does it matter if we unscale $\mathbf{F}$ before or after calling refineF?

The relationship between $\mathbf{F}$ and $\mathbf{F}_{normalized}$ is fixed and defined by a set of transformations, so we can convert at any stage before or after refinement. The nonlinear optimization in refineF may work slightly better with normalized $\mathbf{F}$, but it should be fine either way.

Q2.1: Why does the other image disappear (or become really small) when I select a point using the displayEpipolarF GUI?

This issue occurs when the corresponding epipolar line to the point you selected lies far away from the image. Something is likely wrong with your fundamental matrix.

Q2.1 Note: The GUI will provide the correct epipolar lines even if the program is using the wrong order of pts1 and pts2 in calculating the eightpoint algorithm. So one thing to check is that the optimizer should only take < 10 iterations (shown in the output) to converge if the ordering is correct.

Q3.2: How can I get started formulating the triangulation equations?

One possible method: from the first camera, $x_{1i} = P_1\omega_1 \implies x_{1i} \times P_1\omega_1 = 0 \implies A_{1i}\omega_i = 0$. This is a linear system of 3 equations, one of which is redundant (a linear combination of the other two), and 4 variables. We get a similar equation from the second camera, for a total of 4 (non-redundant) equations and 4 variables, i.e. $A_i\omega_i = 0$.

Q3.2: What is the expected value of the reprojection error?

The reprojection error for the data in `some_corresp.npz` should be around 352 (or 89 without using refineF). If you get a reprojection error of around 94 (or 1927 without using refineF) then you have somehow ended up with a transposed $\mathbf{F}$ matrix in your eightpoint function.

Q3.2: If you are getting high reprojection error but can't find any errors in your triangulate function?

one useful trick is to temporarily comment out the call to refineF in your 8-point algorithm and make sure that the epipolar lines still match up. The refineF function can sometimes find a pretty good solution even starting from a totally incorrect matrix, which results in the F matrix passing the sanity checks even if there's an error in the 8-point function. However, having a slightly incorrect F matrix can still cause the reprojection error to be really high later on even if your triangulate code is correct.

Q4.2 Note: Figure 7 in the assignment document is incorrect - if you look closely you'll notice that the z coordinates are all negative. Don't worry if your solution is different from the example as long as the 3D structure of the temple is evident.

Q5.1: How many inliers should I be getting from RANSAC?

The correct number of inliers should be around 106. This provides a good sanity check for whether the chosen tolerance value is appropriate.

References