HOMEWORK 3: 3D RECONSTRUCTION

16-820 Advanced Computer Vision (Fall 2024)
https://16820advancedcv.github.io/

OUT: October 3rd, 2024
DUE: October 23rd, 2024
Instructor: Matthew O'Toole
TAs: Nikhil Keetha, Ayush Jain, Yuyao Shi

Abhinandan Vellanki

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 answers, 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. To use the provided template - upload the template .zip file directly to Overleaf.
  - **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 3.10.12. We recommend setting up python virtual environment (conda or venv) for the assignment.
  - 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.

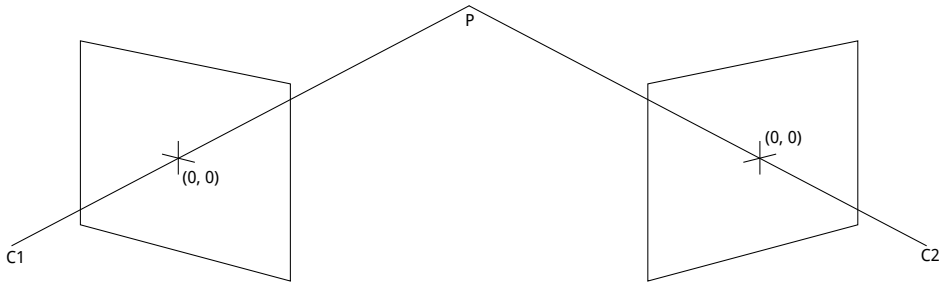


Figure 1: Figure for Q1.1. $C1$ and $C2$ are the optical centers. The principal axes intersect at point $\mathbf{w}$ (P in the figure).

Q1.1

By the definition of the coordinate origin of the image planes of C1 and C2, corresponding image points are [0, 0, 1] in both cameras.

As defined in slide 49 of Lecture 8a, the fundamental matrix F relates two corresponding points as:

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

Solution continued on next page...

Q1.1

Substituting [0, 0, 1] as the two corresponding points:

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

$$\implies \begin{bmatrix}0 & 0 & 1\end{bmatrix}\begin{bmatrix}F_{13}\\ F_{23}\\ F_{33}\end{bmatrix} = 0$$

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

Hence Proven

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. Back up your argument with relevant equations. You may assume both cameras have the same intrinsics.

Q1.2

Given that there is no rotation and pure translation between the two cameras, the rotation matrix relating their poses is $R = I$ and the translation vector is $t = \begin{bmatrix} t \\ 0 \\ 0 \end{bmatrix}$ where t is the amount of translation on the x axis. In its skew-symmetric matrix form, the translation vector is $\hat{t} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t \\ 0 & t & 0 \end{bmatrix}$.

Given the cameras have the same intrinsic matrix (K), the fundamental matrix is $F = K^{-T}\hat{t}RK^{-1}$.

$K = \begin{bmatrix} f_x & s\theta & s_x \\ 0 & f_y & s_y \\ 0 & 0 & 1 \end{bmatrix}$, is an upper triangular matrix, so its inverse is also upper-triangular.

Taking $K^{-1} = \begin{bmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & 1 \end{bmatrix}$, where a, b, c, d, e, f are real numbers and substituting $\hat{t}$,

$$F = \begin{bmatrix} a & 0 & 0 \\ b & d & 0 \\ c & e & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t \\ 0 & t & 0 \end{bmatrix} \begin{bmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -dt \\ 0 & dt & 0 \end{bmatrix}$$

The Fundamental matrix relates two corresponding points as $x_2 F x_1 = 0$ Hence,

$$\begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -dt \\ 0 & dt & 0 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = 0$$

$$\implies \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -dt \\ dt * y_2 \end{bmatrix} = 0$$

$$\implies -y_1 * dt + dt * y_2 = 0 \implies y_1 * dt = dt * y_2$$

From the definition of K, $d \neq 0$ because d corresponds to f_y which is non-zero. Hence, dt can be cancelled out, leaving $y_1 = y_2$, which means the lines have the same Y coordinate, i.e., they are parallel along the X-axis.

Q1.3 [5 points] Suppose we have an inertial sensor that 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 $\mathbf{t}_{rel}$.

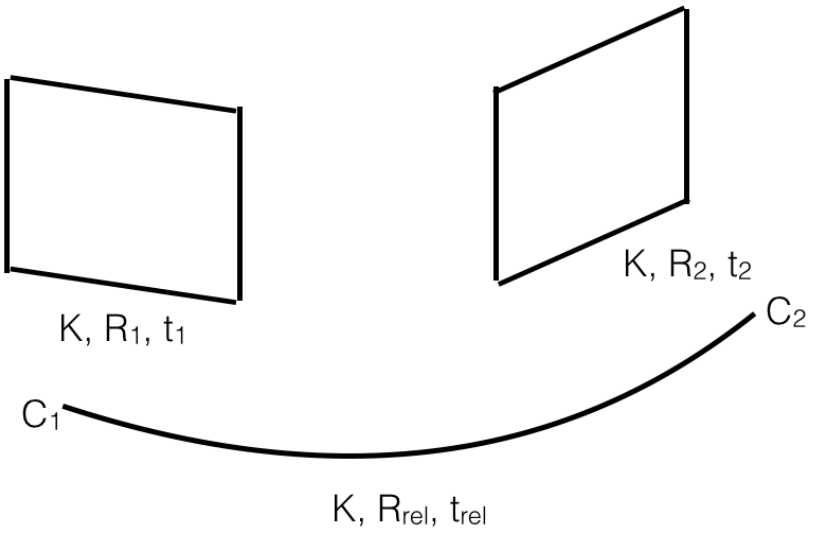


Figure 2: Figure for Q1.3. $C1$ and $C2$ are the optical centers. The rotation and the translation is obtained using inertial sensors. $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$ are the relative rotation and translation between two frames.

Q1.3

The fundamental matrix F relates two corresponding points x_2 and x_1 as $x_2 F x_1 = 0$
The inertial sensor measures rotation and translation at time i, but starts from time i=0. For time i=0, camera pose is X_0.
Hence, for time i, $X_i = R_i X_0 + t_i$ and for time j, $X_j = R_j X_0 + t_j$

$$\implies X_0 = R_i^{-1}(X_i - t_i) = R_j^{-1}(X_j - t_j)$$
$$X_j = R_j R_i^{-1} + (t_j - R_j R_i^{-1} t_i)$$

Hence, R_{rel} and t_{rel} between time i to time j:
$R_{rel} = R_j R_i^{-1}$
$t_{rel} = t_j - R_j R_i^{-1} t_i$

The relationship of corresponding points' coordinates in images taken at times i and j, is given by $x_j^T \hat{T} R x_i = 0$, where $\hat{T}$ is the skew-symmetric matrix form of the translation vector between the camera positions at time i and time j.
The essential matrix is hence given by $E = \hat{T}R$, and for this scenario, $E = \hat{t}_{rel} R_{rel}$.

The fundamental matrix F can be computed from E: $F = K^{-T} E K^{-1}$ where K is the intrinsic matrix of the camera. Hence, for this scenario, $F = K^{-T} \hat{t}_{rel} R_{rel} K^{-1}$,
where if $t_{rel} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$, $\hat{t}_{rel} = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}$

Part II

Practice

1 Overview

In this part you will begin by implementing the 8-point algorithm seen in class to estimate the fundamental matrix from corresponding points in two images (section 2). Next, given the fundamental matrix and calibrated intrinsics (which will be provided) you will compute the essential matrix and use this to compute a 3D metric reconstruction from 2D correspondences using triangulation (section 3). Then, you will implement a method to automatically match points taking advantage of epipolar constraints and make a 3D visualization of the results (section 4). Finally, you will implement RANSAC and bundle adjustment to further improve your algorithm (section 5).

2 Fundamental Matrix Estimation

In this section you will explore different methods of estimating the fundamental matrix given a pair of images. In the `data/` directory, you will find two images (see Figure 3) from the Middlebury multi-view dataset[1], which is used to evaluate the performance of modern 3D reconstruction algorithms.

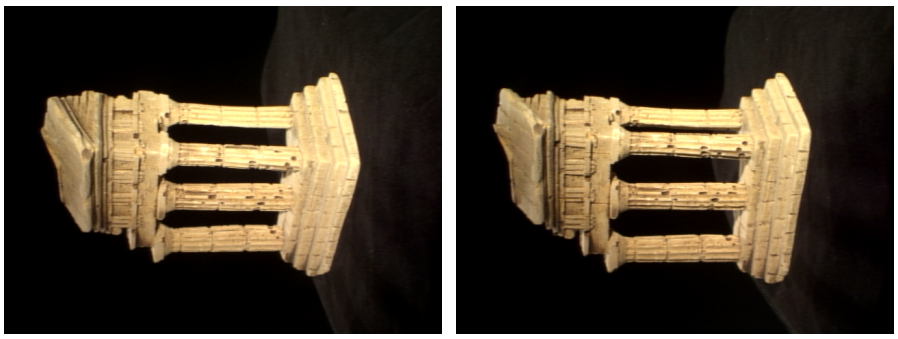


Figure 3: `Temple` images for this assignment

2.1 The Eight Point Algorithm

The 8-point algorithm (discussed in class, and outlined in Section 8.1 of [1]) is arguably the simplest method for estimating the fundamental matrix. For this section, you can use provided correspondences you can find in `data/some_corresp.npz`.

Q2.1 [10 points] Finish the function `eightpoint` in `q2_1_eightpoint.py`. Make sure you follow the signature for this portion of the assignment:

```
F = eightpoint(pts1, pts2, M)
```

where `pts1` and `pts2` are $N \times 2$ matrices corresponding to the (x, y) coordinates of the N points in the first and second image respectively. `M` is a scale parameter.

[1] http://vision.middlebury.edu/mview/data/

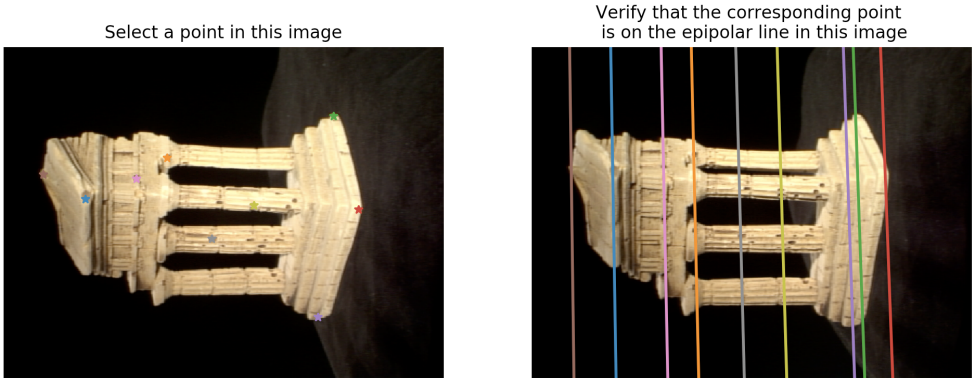


Figure 4: `displayEpipolarF` in `helper.py` creates a GUI for visualizing epipolar lines

- You should scale the data as was discussed in class, by dividing each coordinate by M (the maximum of the image's width and height). After computing $\mathbf{F}$, you will have to "unscale" the fundamental matrix.
  Hint: If $\mathbf{x}_{normalized} = \mathbf{T}\mathbf{x}$, then $\mathbf{F}_{unnormalized} = \mathbf{T}^T\mathbf{F}\mathbf{T}$.
  You must enforce the singularity condition of $\mathbf{F}$ before unscaling.
- You may find it helpful to refine the solution by using local minimization. This probably won't fix a completely broken solution, but may make a good solution better by locally minimizing a geometric cost function. For this we have provided a helper function `refineF` in `helper.py` taking in $\mathbf{F}$ and the two sets of points, which you can call from `eightpoint` before unscaling `F`.
- Remember that the x-coordinate of a point in the image is its column entry, and y-coordinate is the row entry. Also note that eight-point is just a figurative name, it just means that you need at least 8 points; your algorithm should use an over-determined system ($N > 8$ points).
- To visualize the correctness of your estimated $\mathbf{F}$, use the function `displayEpipolarF` in `helper.py`, which takes in $\mathbf{F}$, and the two images. This GUI lets you select a point in one of the images and visualize the corresponding epipolar line in the other image (Figure 4).
- In addition to visualization, we also provide a test code snippet in `q2_1_eightpoint.py` which uses helper function `calc_epi_error` to evaluate the quality of the estimated fundamental matrix. This function calculates the distance between the estimated epipolar line and the corresponding points. For the eight point algorithm, the error should on average be < 1.

Output: Save your matrix $\mathbf{F}$ and scale `M` to the file `q2_1.npz`.
In your write-up:

- Write your recovered $\mathbf{F}$
- Include an image of some example output of `displayEpipolarF`
- Include the code snippet of `eightpoint` function

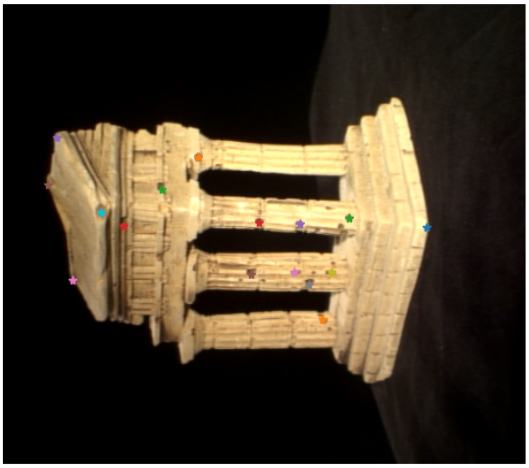
Q2.1

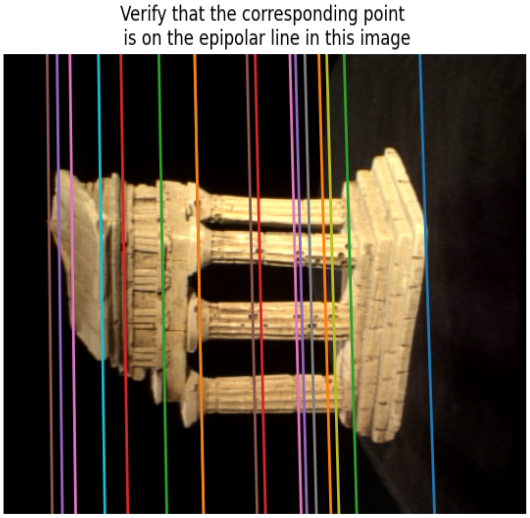
The output of the function *eightpoint(pts1, pts2, M)* was stored in a variable F. The variables F and M were saved to file **q2_1.npz**.

The **F matrix** was found to be:

```
[[-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]]
```

The output of *displayEpipolarF*:





Solution continued on next page...

Q2.1

The function *eightpoint(pts1, pts2, M)*:

```python
def eightpoint(pts1, pts2, M):
    # img = (col, row)

    # Get number of correspondences
    num_corr = pts1.shape[0]

    if num_corr < 8:
        raise ValueError("At least 8 point correspondences are required")

    # 1. Scale (normalize) the coordinates by largest image dimension
    # Normalize points to range [0, 1]
    T = np.array([[1/M, 0, 0], [0, 1/M, 0], [0, 0, 1]])

    # Normalize pts
    pts1_homo = np.hstack((pts1, np.ones((num_corr, 1))))
    pts2_homo = np.hstack((pts2, np.ones((num_corr, 1))))
    norm_pts1 = (T @ pts1_homo.T).T[:, :2]
    norm_pts2 = (T @ pts2_homo.T).T[:, :2]

    # 2. Setup 8-point algo equation
    x1, y1 = norm_pts1[:, 0], norm_pts1[:, 1]
    x2, y2 = norm_pts2[:, 0], norm_pts2[:, 1]
    A = np.vstack((x1 * x2, y1 * x2, x2, x1 * y2, y1 * y2, y2, x1, y1, np.ones(x1.shape[0]).T)).T

    # 3. Solve by svd
    U, s, V = np.linalg.svd(A)

    # Reshape into matrix
    F_norm = V[-1].reshape((3, 3))

    # 4. Singularize
    F_norm = _singularize(F_norm)

    # 5. Refine
    F_norm = refineF(F_norm, norm_pts1, norm_pts2)

    # 6. Unscale
    F = T.T @ F_norm @ T

    # make F[2][2] = 1
    F = 1/F[2][2] * F

    return F
```

2.2 The Seven Point Algorithm (Extra Credit)

Since the fundamental matrix only has seven degrees of freedom, it is possible to calculate **F** using only seven-point correspondences. This requires solving a polynomial equation. In this section, you will implement the seven-point algorithm (outlined in this post).

Q2.2 [Extra Credit - 15 points] Finish the function `sevenpoint` in `q2_2_sevenpoint.py`. Make sure you follow the signature for this portion of the assignment:

```
Farray = sevenpoint(pts1, pts2, M)
```

where pts1 and pts2 are 7×2 matrices containing the correspondences and `M` is the normalizer (use the maximum of the image's height and width), and `Farray` is a list array of length either 1 or 3 containing Fundamental matrix/matrices. Use `M` to normalize the point values between $[0, 1]$ and remember to "unnormalize" your computed **F** afterward.

Manually select 7 points from the provided point in `data/some_corresp.npz`, and use these points to recover a fundamental matrix **F**. Use `calc_epi_error` in `helper.py` to calculate the error to pick the best one, and use `displayEpipolarF` to visualize and verify the solution.

Output: Save your matrix **F** and scale `M` to the file `q2_2.npz`.
In your write-up:

- Write your recovered **F**
- Include an image of some example output of `displayEpipolarF`
- Include the code snippet of `sevenpoint` function

Q2.2

3 Metric Reconstruction

You will compute the camera matrices and triangulate the 2D points to obtain the 3D scene structure. To obtain the Euclidean scene structure, first convert the fundamental matrix $\mathbf{F}$ to an essential matrix $\mathbf{E}$. Examine the lecture notes and the textbook to find out how to do this when the internal camera calibration matrices $\mathbf{K}_1$ and $\mathbf{K}_2$ are known; these are provided in `data/intrinsics.npz`.

Q3.1 [5 points] Complete the function `essentialMatrix` in `q3_1_essential_matrix.py` to compute the essential matrix $\mathbf{E}$ given $\mathbf{F}$, $\mathbf{K}_1$ and $\mathbf{K}_2$ with the signature:

```
E = essentialMatrix(F, K1, K2)
```

Output: Save your estimated $\mathbf{E}$ using $\mathbf{F}$ from the eight-point algorithm to `q3_1.npz`.
In your write-up:

- Write your estimated $\mathbf{E}$
- Include the code snippet of `essentialMatrix` function

Q3.1

The estimated E:

```
[[-4.06342149e-01  1.53921485e+02 -2.29657484e+03]
 [ 4.40438546e+02 -4.18931134e+00  7.33993884e+01]
 [ 2.30439273e+03  3.75288180e+00  1.00000000e+00]]
```

The function *essentialmatrix(F, K1, K2)*:

```
def essentialMatrix(F, K1, K2):
    E = K2.T @ F @ K1
    E = 1/E[2][2] * E
    return E
```

Given an essential matrix, it is possible to retrieve the projective camera matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ from it. Assuming $\mathbf{M}_1$ is fixed at $[\mathbf{I}, 0]$, $\mathbf{M}_2$ can be retrieved up to a scale and four-fold rotation ambiguity. For details on recovering $\mathbf{M}_2$, see section 11.3 in Szeliski. We have provided you with the function `camera2` in `python/helper.py` to recover the four possible $\mathbf{M}_2$ matrices given $\mathbf{E}$.

Note: The matrices M1 and M2 here are of the form: $\mathbf{M}_1 = \big[\mathbf{I}|0\big]$ and $\mathbf{M}_2 = \big[\mathbf{R}|\mathbf{t}\big]$.

Q3.2 [10 points] Using the above, complete the function `triangulate` in `q3_2_triangulate.py` to triangulate a set of 2D coordinates in the image to a set of 3D points with the signature:

```
[w, err] = triangulate(C1, pts1, C2, pts2)
```

where `pts1` and `pts2` are the $N \times 2$ matrices with the 2D image coordinates and `w` is an $N \times 3$ matrix with the corresponding 3D points per row. `C1` and `C2` are the 3×4 camera matrices. Remember that you will need to multiply the given intrinsics matrices with your solution for the canonical camera matrices to obtain the final camera matrices. Various methods exist for triangulation - probably the most familiar for you is based on least squares (see [2] Chapter 7 if you want to learn about other methods).

For each point i, we want to solve for 3D coordinates $\mathbf{w}_i = \begin{bmatrix} x_i, y_i, z_i \end{bmatrix}^T$, such that when they are projected back to the two images, they are close to the original 2D points. To project the 3D coordinates back to 2D images, we first write $\mathbf{w}_i$ in homogeneous coordinates, and compute $\mathbf{C}_1\tilde{\mathbf{w}}_i$ and $\mathbf{C}_2\tilde{\mathbf{w}}_i$ to obtain the 2D homogeneous coordinates projected to camera 1 and camera 2, respectively.

For each point i, we can write this problem in the following form:

$$\mathbf{A}_i\mathbf{w}_i = 0,$$

where $\mathbf{A}_i$ is a 4×4 matrix, and $\tilde{\mathbf{w}}_i$ is a 4×1 vector of the 3D coordinates in the homogeneous form. Then, you can obtain the homogeneous least-squares solution (discussed in class) to solve for each $\mathbf{w}_i$.

Once you have implemented triangulation, check the performance by looking at the reprojection error:

$$\texttt{err} = \sum_i \|\mathbf{x}_{1i}, \widehat{\mathbf{x}_{1i}}\|^2 + \|\mathbf{x}_{2i}, \widehat{\mathbf{x}_{2i}}\|^2$$

where $\widehat{\mathbf{x}_{1i}} = Proj(\mathbf{C}_1, \mathbf{w}_i)$ and $\widehat{\mathbf{x}_{2i}} = Proj(\mathbf{C}_2, \mathbf{w}_i)$. You should see an error less than 500. Ours is around 350.

Note: `C1` and `C2` here are projection matrices of the form: $\mathbf{C}_1 = \mathbf{K}_1\mathbf{M}_1 = \mathbf{K}_1\begin{bmatrix}\mathbf{I}|0\end{bmatrix}$ and $\mathbf{C}_2 = \mathbf{K}_2\mathbf{M}_2 = \mathbf{K}_2\begin{bmatrix}\mathbf{R}|\mathbf{t}\end{bmatrix}$.

In your write-up:

- Write down the expression for the matrix $\mathbf{A}_i$ for triangulating a pair of 2D coordinates in the image to a 3D point.
- Include the code snippet of `triangulate` function.

Q3.2

For each corresponding pair of 2D points: (x_i, y_i) in pts_1 from camera 1 and (x_i', y_i') in pts_2 from camera 2, the matrix A_i can be written as:

$$A_i = \begin{bmatrix} y_i * C1_3 - C1_2 \\ C1_1 - x_i * C1_3 \\ y_i' * C2_3 - C2_2 \\ C2_1 - x_i' * C2_3 \end{bmatrix}$$

Where C1 is the camera matrix mapping 3D points to the image plane of camera 1, C2 is the camera matrix for camera 2 and the subscript on C1 and C2 is the number of the row that should be used, i.e., $C1_1$ means the first row of C1.

Q3.2

The function *triangulate(C1, pts1, C2, pts2)*:

```python
def triangulate(C1, pts1, C2, pts2):
    # get number of correspondences
    num_corr = pts1.shape[0]

    # reprojection error
    err = 0

    # 3D points
    P = []
    for i in range(num_corr):
        # get coords
        x1, y1 = pts1[i]
        x2, y2 = pts2[i]
        # setup and solve least squares problem
        A = np.array(
            [
                y1 * C1[2, :] - C1[1, :],
                C1[0, :] - x1 * C1[2, :],
                y2 * C2[2, :] - C2[1, :],
                C2[0, :] - x2 * C2[2, :],
            ]
        )
        U, s, V = np.linalg.svd(A)

        # get 3D point in homogenous coords
        X = V[-1]
        # print(X)

        # caluclate projection error
        # pts1
        proj_pts1_i_homo = C1 @ X
        # de-homogenize
        proj_pts1_i = proj_pts1_i_homo[:2] / proj_pts1_i_homo[2]
        d_1 = np.linalg.norm(pts1[i] - proj_pts1_i)**2

        # pts2
        proj_pts2_i_homo = C2 @ X
        # de-homogenize
        proj_pts2_i = proj_pts2_i_homo[:2] / proj_pts2_i_homo[2]
        d_2 = np.linalg.norm(pts2[i] - proj_pts2_i)**2

        # sum up error
        err += d_1 + d_2

        # de-homogenize world point
        X = X[:3] / X[3]

        # add to list
        P.append(X)

    return np.array(P), err
```

Q3.3 [10 points] Complete the function `findM2` in `q3_2_triangulate.py` to obtain the correct `M2` from `M2s` by testing the four solutions through triangulations.
Use the correspondences from `data/some_corresp.npz`.

Output: Save the correct `M2`, the corresponding `C2`, and 3D points `P` to `q3_3.npz`.
In your writeup: Include the code snippet of `findM2` function.

Q3.3

The estimates M2, C2 and 3D points were saved to file **q3_3.npz**.

The function *findM2(F, pts1, pts2, intrinsics, filename="q3_3.npz"):*

```
def findM2(F, pts1, pts2, intrinsics, filename="q3_3.npz"):
    # 1. Get four possible M2s
    # Get intrinsics
    K1, K2 = intrinsics["K1"], intrinsics["K2"]

    # Get Essential Matrix
    E = essentialMatrix(F=F, K1=K1, K2=K2)
    #print(E)

    # Get M2s
    M2s = camera2(E)
    #print(M2s)

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

    # 2. Check which M2 is correct by checking if all points are in front of both cameras
    for i in range(M2s.shape[2]):
        M2 = M2s[:, :, i]
        C2 = K2 @ M2
        C1 = K1 @ M1
        P, err = triangulate(C1=C1, pts1=pts1, C2=C2, pts2=pts2)
        if np.all(P[:, 2] > 0):
            np.savez(filename, M2=M2, C2=C2, P=P)
            return M2, C2, P

    return None, None, None
```

4 3D Visualization

You will now create a 3D visualization of the temple images. By treating our two images as a stereo-pair, we can triangulate corresponding points in each image, and render their 3D locations.

Q4.1 [15 points] In `q4_1_epipolar_correspondence.py` finish the function `epipolarCorrespondence` with the signature:

```
[x2, y2] = epipolarCorrespondence(im1, im2, F, x1, y1)
```

This function takes in the x and y coordinates of a pixel on `im1` and your fundamental matrix $\mathbf{F}$, and returns the coordinates of the pixel on `im2` which correspond to the input point. The match is obtained by computing the similarity of a small window around the (x_1, y_1) coordinates in `im1` to various windows around possible matches in the `im2` and returning the closest.

Instead of searching for the matching point at every possible location in `im2`, we can use $\mathbf{F}$ and simply search over the set of pixels that lie along the epipolar line (recall that the epipolar line passes through a single point in `im2` which corresponds to the point (x_1, y_1) in `im1`).

There are various possible ways to compute the window similarity. For this assignment, simple methods such as the Euclidean or Manhattan distances between the intensity of the pixels should suffice. See [2] chapter 11, on stereo matching, for a brief overview of these and other methods.

Implementation hints:

- Experiment with various window sizes.
- It may help to use a Gaussian weighting of the window, so that the center has greater influence than the periphery.
- Since the two images only differ by a small amount, it might be beneficial to consider matches for which the distance from (x_1, y_1) to (x_2, y_2) is small.

To help you test your `epipolarCorrespondence`, we have included a helper function `epipolarMatchGUI` in `q4_1_epipolar_correspondence.py`, which takes in two images and the fundamental matrix. This GUI allows you to click on a point in `im1`, and will use your function to display the corresponding point in `im2`. See Figure 5.

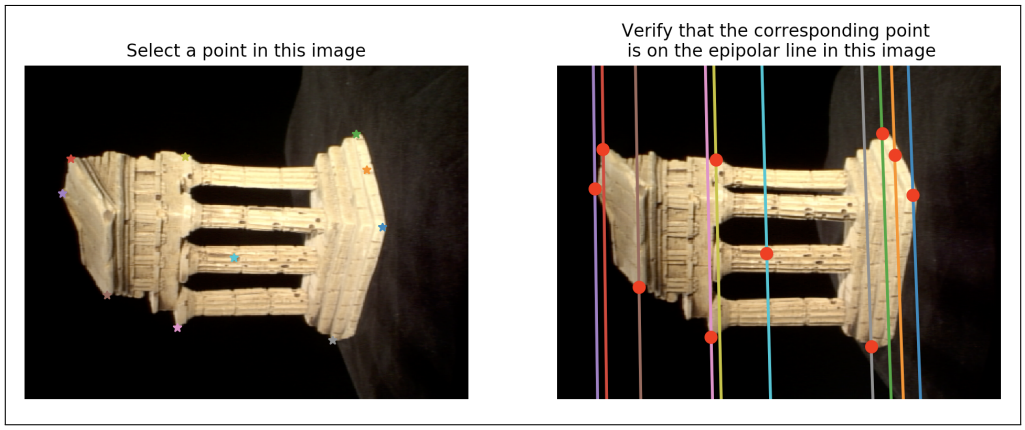


Figure 5: `epipolarMatchGUI` shows the corresponding point found by calling `epipolarCorrespondence`

It's not necessary for your matcher to get *every* possible point right, but it should get easy points (such as those with distinctive, corner-like windows). It should also be good enough to render an intelligible representation in the next question.

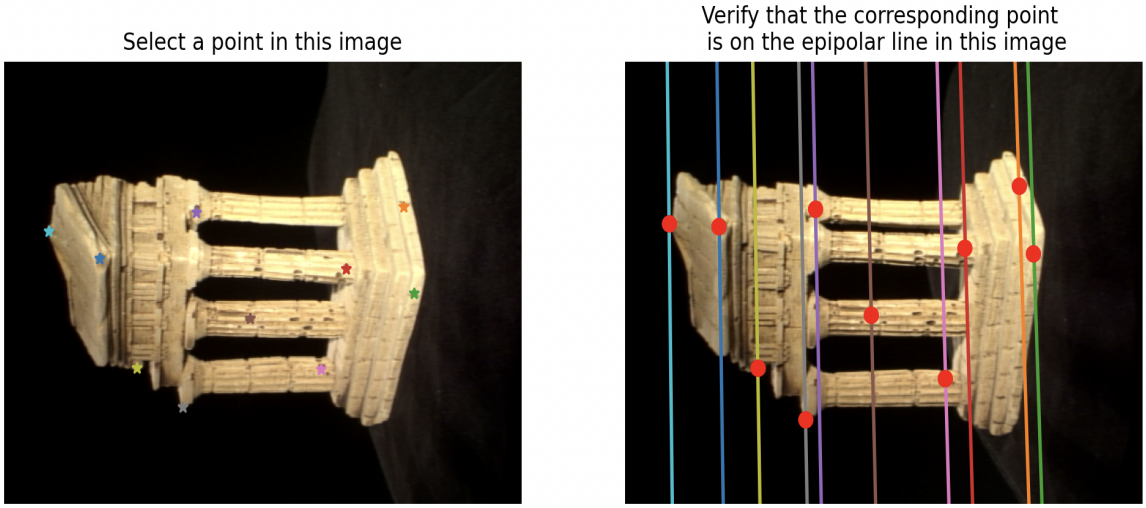
Output: Save the matrix **F**, points `pts1` and `pts2` which you used to generate the screenshot to the file `q4_1.npz`.

In your write-up:

- Include a screenshot of `epipolarMatchGUI` with some detected correspondences.
- Include the code snippet of `epipolarCorrespondence` function.

Q4.1

The output of *epipolarMatchGUI()*:



Solution continued on next page...

Q4.1

The function *epipolarCorrespondence(im1, im2, F, x1, y1)*:

```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # number of pixels surrounding the point of interest in the search window
    window_size = 2
    # distance threshold for the best match
    dist_threshold = 50

    # 1. Get the epipolar line in im2
    v = np.array([x1, y1, 1])
    l = F @ v

    # 2. Get search window from im1
    search_window = im1[
        y1 - window_size : y1 + window_size + 1, x1 - window_size : x1 + window_size + 1
    ]

    # Generate Gaussian kernel
    kernel_size= 2 * window_size + 1
    sigma = 0.5
    coords = np.linspace(-1, 1, kernel_size)
    y, x = np.meshgrid(coords, coords)
    kernel = np.exp(-((x**2 + y**2) / (2.0 * sigma**2)))
    kernel /= kernel.sum()

    # 3. Search along the epipolar line in im2 for best match
    least_error = float("inf")
    best_x2, best_y2 = 0, 0

    for y2 in range(im2.shape[0]):
        x2 = int(-1 * (l[1] * y2 + l[2]) / l[0])
        if x2 < 0 or x2 >= im2.shape[1]:
            continue

        # Get search window from im2
        search_window2 = im2[
            y2 - window_size : y2 + window_size + 1, x2 - window_size : x2 + window_size + 1
        ]

        if search_window2.shape != search_window.shape:
            continue

        dist = np.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
        if dist > dist_threshold:
            continue

        # Apply Gaussian weighting to each channel of the RGB difference image and compute the difference
        error = 0
        for channel in range(3):
            diff = np.absolute(search_window[:, :, channel] - search_window2[:, :, channel])
            weighted_diff = np.multiply(diff, kernel) # apply gaussian kernel
            error += np.linalg.norm(weighted_diff)
        error /= (2 * window_size + 1) ** 2 # normalize the error

        # Update the best match
        if error < least_error:
            least_error = error
            best_x2, best_y2 = x2, y2

    return best_x2, best_y2
```

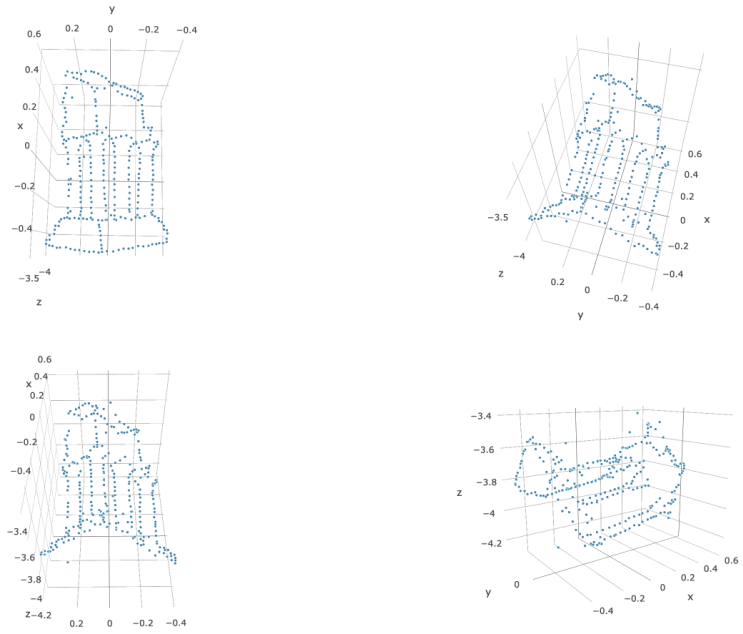


Figure 6: An example point cloud

Q4.2 [10 points] Included in this homework is a file `data/templeCoords.npz` which contains 288 hand-selected points from `im1` saved in the variables `x1` and `y1`.

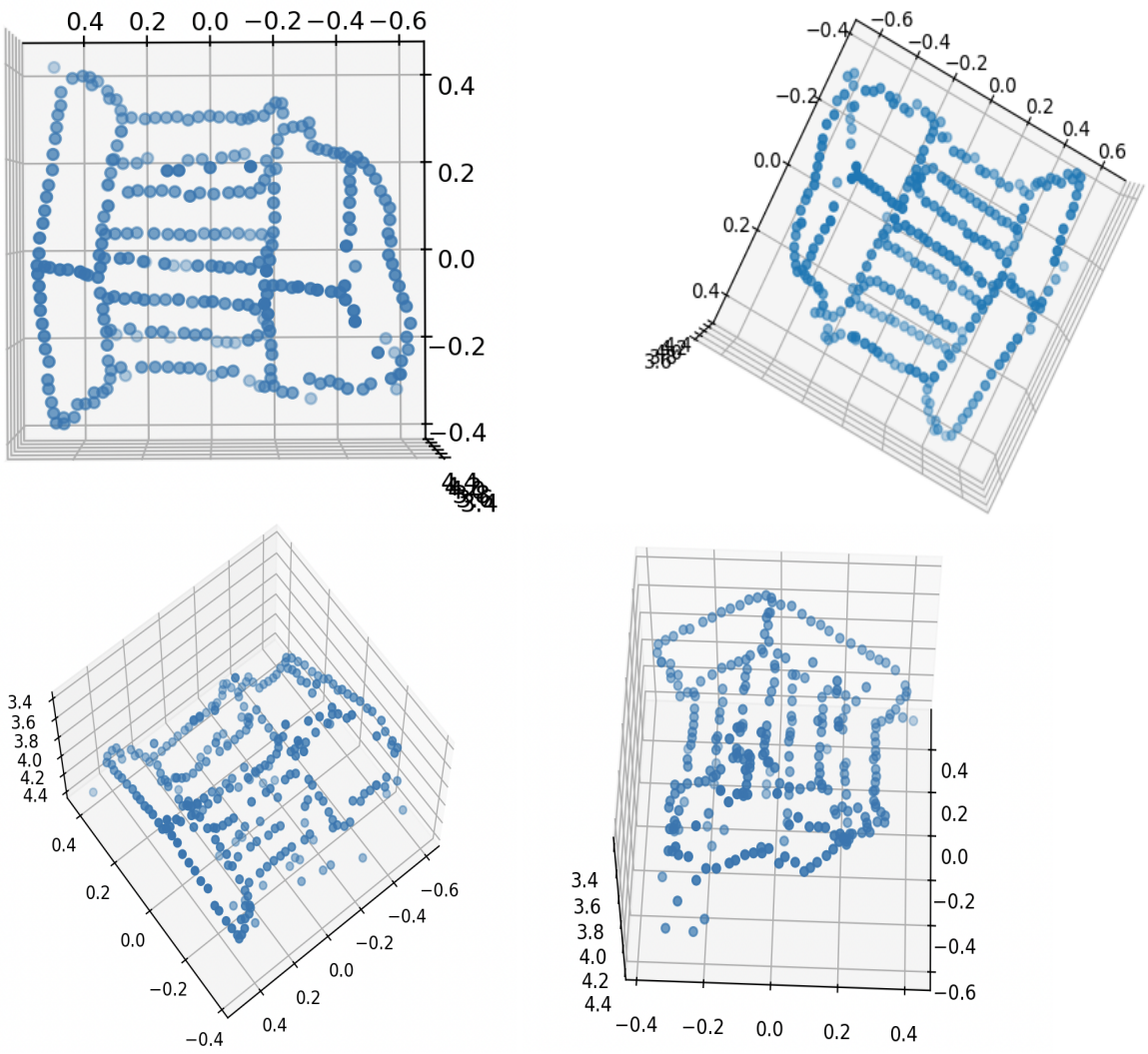
Now, we can determine the 3D location of these point correspondences using the `triangulate` function. These 3D point locations can then be plotted using the Matplotlib or plotly package. Complete the `compute3D_pts` function in `q4_2_visualize.py`, which loads the necessary files from `../data/` to generate the 3D reconstruction using `scatter` function matplotlib. An example is shown in Figure 6.

Output: Again, save the matrix $\mathbf{F}$, matrices $\mathbf{M1}, \mathbf{M2}, \mathbf{C1}, \mathbf{C2}$ which you used to generate the screenshots to the file `q4_2.npz`.

In your write-up:

- Take a few screenshots of the 3D visualization so that the outline of the temple is clearly visible, and include them in your writeup.
- Include the code snippet of `compute3D_pts` function in your write-up.

Q4.2



The function *compute3D_pts*

```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    temple_pts2 = np.zeros_like(temple_pts1)

    for i in range(temple_pts1.shape[0]):
        x1, y1 = temple_pts1[i]
        x2, y2 = epipolarCorrespondence(im1, im2, F, x1, y1)
        temple_pts2[i] = x2, y2

    # Find M2
    M2, C2, P = findM2(F=F, intrinsics=intrinsics, pts1=temple_pts1, pts2=temple_pts2)

    return M2, C2, P
```

5 Bundle Adjustment

Bundle Adjustment is commonly used as the last step of every feature-based 3D reconstruction algorithm. Given a set of images depicting a number of 3D points from different viewpoints, bundle adjustment is the process of simultaneously refining the 3D coordinates along with the camera parameters. It minimizes reprojection error, which is the squared sum of distances between image points and predicted points. In this section, you will implement bundle adjustment algorithm by yourself (make use of `q5_bundle_adjustment.py` file). Specifically,

- In Q5.1, you need to implement a RANSAC algorithm to estimate the fundamental matrix $\mathbf{F}$ and all the inliers.
- In Q5.2, you will need to write code to parameterize Rotation matrix $\mathbf{R}$ using Rodrigues formula (please check this pdf for a detailed explanation), which will enable the joint optimization process for Bundle Adjustment.
- In Q5.3, you will need to first write down the objective function in `rodriguesResidual`, and do the `bundleAdjustment`.

Q5.1 RANSAC for Fundamental Matrix Recovery [15 points] In some real world applications, manually determining correspondences is infeasible and often there will be noisy correspondences. Fortunately, the RANSAC method seen in class can be applied to the problem of fundamental matrix estimation.

Implement the above algorithm with the signature:

```
[F, inliers] = ransacF(pts1, pts2, M, nIters, tol)
```

where `M` is defined in the same way as in section 2 and inliers is a boolean vector of size equivalent to the number of points. Here inliers is set to true only for the points that satisfy the threshold defined for the given fundamental matrix $\mathbf{F}$.

We have provided some noisy correspondences in `some_corresp_noisy.npz` in which around 75% of the points are inliers.

In your write-up: Compare the result of RANSAC with the result of the eightpoint when ran on the noisy correspondences. Briefly explain the error metrics you used, how you decided which points were inliers, and any other optimizations you may have made. `nIters` is the maximum number of iterations of RANSAC and `tol` is the tolerance of the error to be considered as inliers. Discuss the effect on the Fundamental matrix by varying these values. **Please include the code snippet of the `ransacF` function in your write-up.**

- *Hints:* Use the Eight or Seven point algorithm to compute the fundamental matrix from the minimal set of points. Then compute the inliers, and refine your estimate using all the inliers.

Q5.1

The metric chosen to compare the accuracy of the fundamental matrix (F) before and after RANSAC was the **epipolar error**, calculated using *calc_epi_error(pts1, pts2, F)*.
The inliers were chosen by identifying the points for which the **epipolar error was lower than a tolerance** for a random F computed using 8 randomly selected corresponding points.
The **best F** was chosen as the one which resulted in the largest number of inliers.

Effects of number of iterations (nIters) and tolerance (tol) on the epipolar error calculated using the best F and its inliers:

1. tol: The error decreased by 56% when tolerance was halved and rose by 136% when tolerance was doubled, indicating a strong correlation to the accuracy of the fundamental matrix.
2. nIters: The error decreased by 36% when the number of iterations was increased tenfold and increased by 60% when the number of iterations was increased tenfold, also indicating a considerable correlation to the accuracy of the fundamental matrix. It was also found that the error did not decrease beyond a certain point even when number of iterations was doubled.

The function *ransacF(pts1, pts2, M, nIters, tol):*

```python
def ransacF(pts1, pts2, M, nIters=1000, tol=2):
    if pts1.shape == pts2.shape:
        N = pts1.shape[0]
    else:
        return None

    iterations = 0
    best_F= None
    max_inliers = -1
    inliers = np.zeros(N, dtype=bool)

    while iterations < nIters:
        # select four matching pairs randomly
        chosen_points_idx = np.random.choice(len(pts1), 8, replace=False)
        x1 = pts1[chosen_points_idx]
        x2 = pts2[chosen_points_idx]

        # compute F
        this_F = eightpoint(x1, x2, M)

        # get inliers
        pts1_homo, pts2_homo = toHomogenous(pts1), toHomogenous(pts2)
        err = calc_epi_error(pts1_homo, pts2_homo, this_F)
        inliers_indexes = np.where(err < tol)[0]

        # update best inliers
        if len(inliers_indexes) > max_inliers:
            best_F = this_F
            inliers = np.where(err < tol, True, False).reshape(N, 1)
            max_inliers = len(inliers_indexes)

        iterations+=1

    return best_F, inliers
```

Q5.2 Rodrigues and Invsere Rodrigues [15 points] So far we have independently solved for camera matrix, $\mathbf{M}_j$ and 3D points $\mathbf{w}_i$. In bundle adjustment, we will jointly optimize the reprojection error with respect to the points $\mathbf{w}_i$ and the camera matrix $\mathbf{C}_j$.

$$err = \sum_{ij} \|\mathbf{x}_{ij} - Proj(\mathbf{C}_j, \mathbf{w}_i)\|^2,$$

where $\mathbf{C}_j = \mathbf{K}_j\mathbf{M}_j$, same as in Q3.2.

For this homework we are going to only look at optimizing the extrinsic matrix. To do this we will be parameterizing the rotation matrix $\mathbf{R}$ using Rodrigues formula to produce vector $\mathbf{r} \in \mathbb{R}^3$. Write a function that converts a Rodrigues vector $\mathbf{r}$ to a rotation matrix $\mathbf{R}$

```
R = rodrigues(r)
```

as well as the inverse function that converts a rotation matrix $\mathbf{R}$ to a Rodrigues vector $\mathbf{r}$

```
r = invRodrigues(R)
```

Reference: Rodrigues formula and this pdf.

In your write-up: Include the code snippet of `rodrigues` and `invRodrigues` functions.

Q5.2

The function *rodrigues(r)*

```python
def rodrigues(r):
    # compute Rotation matrix corresponding to the rotation vector r
    theta = np.linalg.norm(r)
    if theta == 0:
        return np.identity(3)
    r = r / theta
    r_x = np.array([[0, -r[2], r[1]], [r[2], 0, -r[0]], [-r[1], r[0], 0]])
    R = np.cos(theta) * np.identity(3) + np.sin(theta) * r_x + (1 - np.cos(theta)) * np.outer(r, r)
    return R
```

Solution continued on next page...

Q5.2

The function *invRodrigues(r)*

```python
def invRodrigues(R):
    # compute the rotation vector r corresponding to the Rotation matrix R
    A = (R - R.T) / 2
    p = np.array([A[2, 1], A[0, 2], A[1, 0]])
    s = np.linalg.norm(p)
    c = (R.trace() - 1) / 2
    if s == 0 and c == 1:
        return np.zeros(3)
    if s == 0 and c == -1:
        v = np.zeros(3)
        for i in range(3):
            if R[i, i] + 1 > 1e-6:
                v[i] = np.sqrt((R[i, i] + 1) / 2)
                break
        return np.pi * v
    theta = np.arctan2(s, c)
    return theta * p / s
```

Q5.3 Bundle Adjustment [10 points]

Using this parameterization, write an optimization function

```
residuals = rodriguesResidual(K1, M1, p1, K2, p2, x)
```

where `x` is the flattened concatenation of $\mathbf{x}$, $\mathbf{r}_2$, and $\mathbf{t}_2$. $\mathbf{w}$ are the 3D points; $\mathbf{r}_2$ and $\mathbf{t}_2$ are the rotation (in the Rodrigues vector form) and translation vectors associated with the projection matrix $\mathbf{M}_2$. The `residuals` are the difference between original image projections and estimated projections (the square of $L2$-norm of this vector corresponds to the error we computed in Q3.2):

```
residuals = numpy.concatenate([(p1-p1_hat).reshape([-1]),
                               (p2-p2_hat).reshape([-1])])
```

Use this error function and Scipy's optimizer `minimize` to write a function that optimizes for the best extrinsic matrix and 3D points using the inlier correspondences from `some_corresp_noisy.npz` and the RANSAC estimate of the extrinsics and 3D points as an initialization.

```
[M2, w, o1, o2] = bundleAdjustment(K1, M1, p1, K2, M2_init, p2, w_init)
```

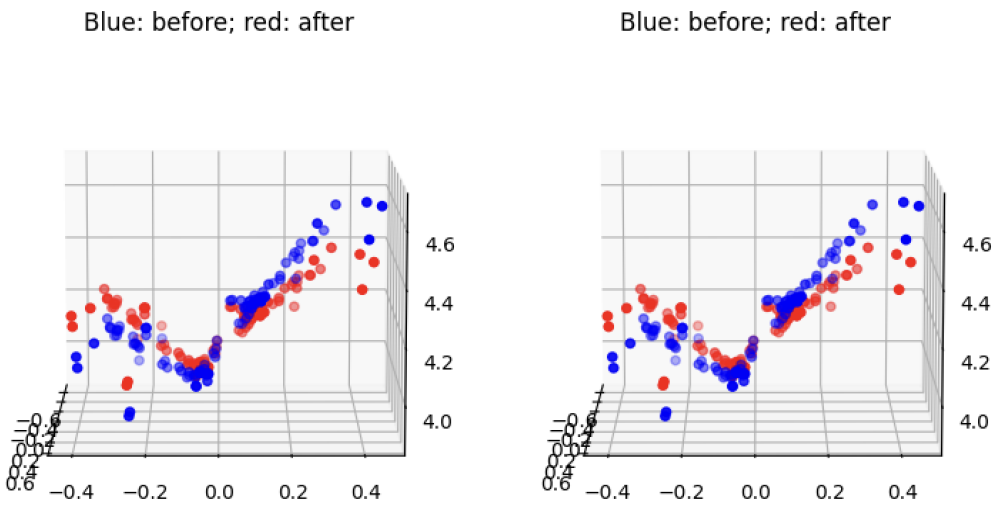
Try to extract the rotation and translation from `M2_init`, then use `invRodrigues` you implemented previously to transform the rotation, concatenate it with translation and the 3D points, then the concatenate vector are variables to be optimized. After obtaining optimized vector, decompose it back to rotation using `rodrigues` you implemented previously, translation and 3D points coordinates.

In your write-up:

- Include an image of the original 3D points and the optimized points (use the provided `plot_3D_dual` function).
- Report the reprojection error with your initial $\mathbf{M}_2$ and $\mathbf{w}$, as well as with the optimized matrices.
- Include the code snippets for `rodriguesResidual` and `bundleAdjustment` in your write-up.

Hint: For reference, our solution achieves a reprojection error around 10 after optimization. Your exact error may differ slightly.

Q5.3



Initial reprojection error with inital M_2 and w: **5026.345716096261**
Final reprojection error with optimized matrices: **9.879762511455533**

Solution continued on next page

Q5.3

The function *rodriguesResidual(K1, M1, p1, K2, p2, x)*

```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # Extract P, r2, t2 from x
    P = x[: 3 * len(p1)].reshape(-1, 3) #first 3N elements
    r2 = x[3 * len(p1) : 3 * len(p1) + 3] #next 3 elements
    t2 = x[3 * len(p1) + 3 :] #remaining elements

    R2 = rodrigues(r2) #get rotation matrix

    # compute C1, C2
    C1 = K1 @ M1
    C2 = K2 @ np.hstack((R2, t2.reshape(-1, 1)))

    # project P to get the estimated points
    p1_est_homo = C1 @ np.hstack((P, np.ones((P.shape[0], 1)))).T
    p2_est_homo = C2 @ np.hstack((P, np.ones((P.shape[0], 1)))).T
    p1_est = (p1_est_homo[:2] / p1_est_homo[2]).T  # project to 2D
    p2_est = (p2_est_homo[:2] / p2_est_homo[2]).T  # project to 2D

    residuals1 = p1 - p1_est  # Error for image 1
    residuals2 = p2 - p2_est  # Error for image 2

    # compute residuals
    residuals = np.concatenate((residuals1.flatten(), residuals2.flatten()))
    return residuals
```

The function *bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init)*

```python
def obj_fun(x, K1, M1, p1, K2, p2):
    return np.linalg.norm(rodriguesResidual(K1, M1, p1, K2, p2, x))**2


def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    # stack P, r2, t2 to create initial vector for optimization
    R2_init, t2_init = M2_init[:, :3], M2_init[:, 3]
    r2_init = invRodrigues(R2_init)
    x_init = np.hstack((P_init.flatten(), r2_init, t2_init))

    # compute initial objective function value
    obj_start = np.linalg.norm(rodriguesResidual(K1, M1, p1, K2, p2, x_init))

    # minimize the objective function
    res = scipy.optimize.minimize(
        fun=obj_fun,
        x0=x_init,
        args=(K1,M1,p1,K2,p2,),
        method="Powell",
    )

    # Get optimized vector
    x_opt = res.x

    # extract optimized P, r2, t2
    P_opt = x_opt[: 3 * len(p1)].reshape(-1, 3)
    r2_opt = x_opt[3 * len(p1) : 3 * len(p1) + 3]
    t2_opt = x_opt[3 * len(p1) + 3 :]

    # compute R2 from r2
    R2_opt = rodrigues(r2_opt)

    # compute M2
    M2_opt = np.hstack((R2_opt, t2_opt[:, np.newaxis]))

    # compute final objective function value
    obj_end = np.linalg.norm(rodriguesResidual(K1, M1, p1, K2, p2, x_opt))

    return M2_opt, P_opt, obj_start, obj_end
```

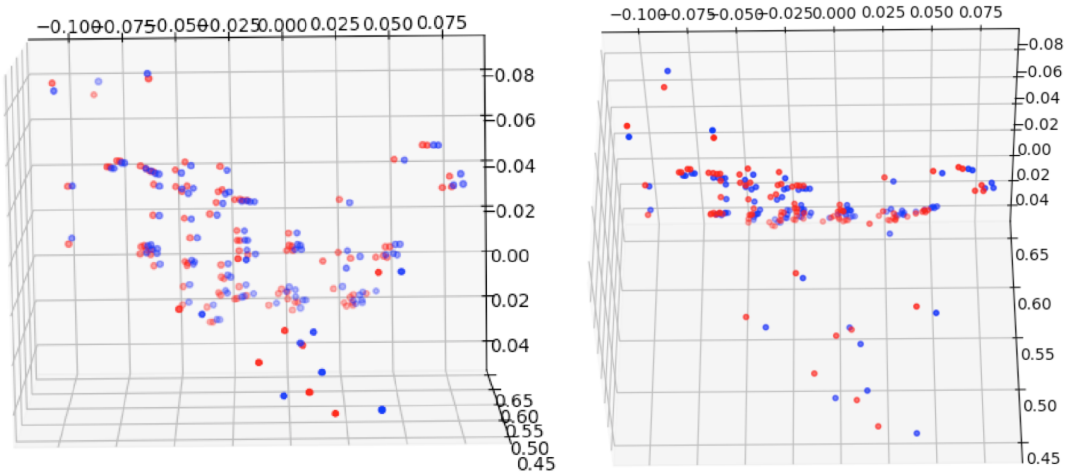


Figure 7: Visualization of 3D points for noisy correspondences before and after using bundle adjustment

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 8 Top) and reconstruct vehicle keypoints and render from multiple views (Figure 8 Bottom). Make use of `q6_ec_multiview_reconstruction.py` file and `data/q6` folder contains the images.

Q6.1 [Extra Credit - 15 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[2] 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.

In your write-up:

- Describe the method you used to compute the 3D locations.

[2] Code Used For Detection and Reconstruction

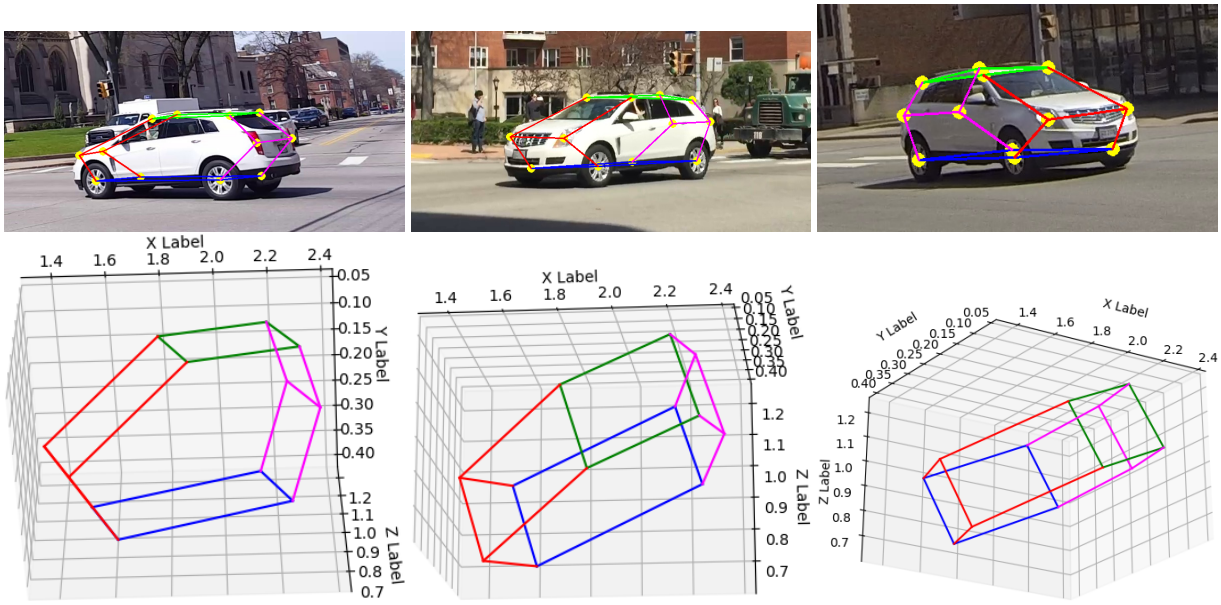


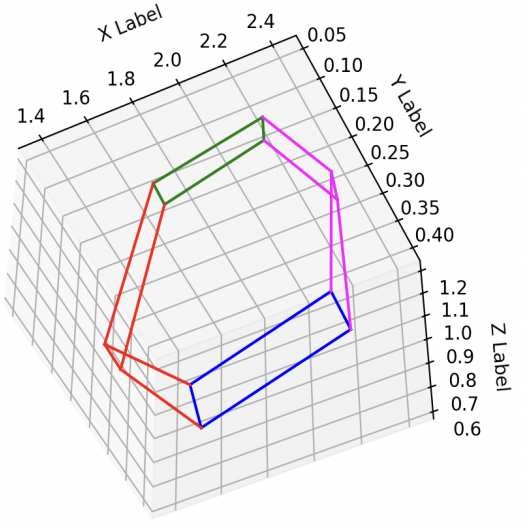
Figure 8: An example detections on the top and the reconstructions from multiple views

- 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

To compute 3D locations, the confidence of each corresponding point was compared against the threshold to select the points to use for triangulation. For 2D triangulation, the function described in Q3.2 was used. When all three points are of high enough confidence, the triangulation function was modified to extend the A_i matrix to include two more rows that account for the third point and the third camera matrix. The 3-view triangulation function is given later in this answer

The reconstructed 3D keypoints for the first frame:



Q6.1

The function *MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres=300)*

```python
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres=300):
    # pts = [x, y, confidence]

    # Extract the 2D points and confidences
    x1,y1,conf1 = pts1[:,0], pts1[:,1], pts1[:,2]
    x2,y2,conf2 = pts2[:,0], pts2[:,1], pts2[:,2]
    x3,y3,conf3 = pts3[:,0], pts3[:,1], pts3[:,2]

    # Loop through correspondences
    P = []
    err = 0
    for i in range(pts1.shape[0]):
        # Check if confidence is above threshold
        if conf1[i] > Thres and conf2[i] > Thres and conf3[i] > Thres:
            # 3view triangulation
            # Get 2D points
            pt1 = np.array([[x1[i], y1[i]]])
            pt2 = np.array([[x2[i], y2[i]]])
            pt3 = np.array([[x3[i], y3[i]]])

            # Triangulate the 3D point
            p, error = triangulate_3(C1, pt1, C2, pt2, C3, pt3)
            P.append(p)
            err += error
        elif conf1[i] > Thres and conf2[i] > Thres and conf3[i] <= Thres:
            # 2view triangulation
            # Get 2D points
            pt1 = np.array([[x1[i], y1[i]]])
            pt2 = np.array([[x2[i], y2[i]]])

            # Triangulate the 3D point
            p, error = triangulate(C1, pt1, C2, pt2)
            P.append(p)
            err += error
        elif conf1[i] > Thres and conf2[i] <= Thres and conf3[i] > Thres:
            # 2view triangulation
            # Get 2D points
            pt1 = np.array([[x1[i], y1[i]]])
            pt3 = np.array([[x3[i], y3[i]]])

            # Triangulate the 3D point
            p, error = triangulate(C1, pt1, C3, pt3)
            P.append(p)
            err += error
        elif conf1[i] <= Thres and conf2[i] > Thres and conf3[i] > Thres:
            # 2view triangulation
            # Get 2D points
            pt2 = np.array([[x2[i], y2[i]]])
            pt3 = np.array([[x3[i], y3[i]]])

            # Triangulate the 3D point
            p, error = triangulate(C2, pt2, C3, pt3)
            P.append(p)
            err += error
        else:
            print("cannot triangulate at:", i)
            continue

    P = np.array(P)
    # remove redundant dimension
    P = np.squeeze(P, axis=1)

    return P, err
```

Q6.1

The function *triangulate_3(C1, pts1, C2, pts2, C3, pts3)*

```python
def triangulate_3(C1, pts1, C2, pts2, C3, pts3):
    num_corr = pts1.shape[0]
    err = 0
    P = []

    for i in range(num_corr):
        x1, y1 = pts1[i]
        x2, y2 = pts2[i]
        x3, y3 = pts3[i]

        A = np.array(
            [
                y1 * C1[2, :] - C1[1, :],
                C1[0, :] - x1 * C1[2, :],
                y2 * C2[2, :] - C2[1, :],
                C2[0, :] - x2 * C2[2, :],
                y3 * C3[2, :] - C3[1, :],
                C3[0, :] - x3 * C3[2, :],
            ]
        )

        U, s, V = np.linalg.svd(A)
        X = V[-1]
        X /= X[3]  # De-homogenize

        # Calculate reprojection errors for all views
        proj_pts1_i_homo = C1 @ X
        proj_pts1_i = proj_pts1_i_homo[:2] / proj_pts1_i_homo[2]
        d_1 = np.linalg.norm(pts1[i] - proj_pts1_i) ** 2

        proj_pts2_i_homo = C2 @ X
        proj_pts2_i = proj_pts2_i_homo[:2] / proj_pts2_i_homo[2]
        d_2 = np.linalg.norm(pts2[i] - proj_pts2_i) ** 2

        proj_pts3_i_homo = C3 @ X
        proj_pts3_i = proj_pts3_i_homo[:2] / proj_pts3_i_homo[2]
        d_3 = np.linalg.norm(pts3[i] - proj_pts3_i) ** 2

        err += d_1 + d_2 + d_3
        P.append(X[:3])

    return np.array(P), err
```

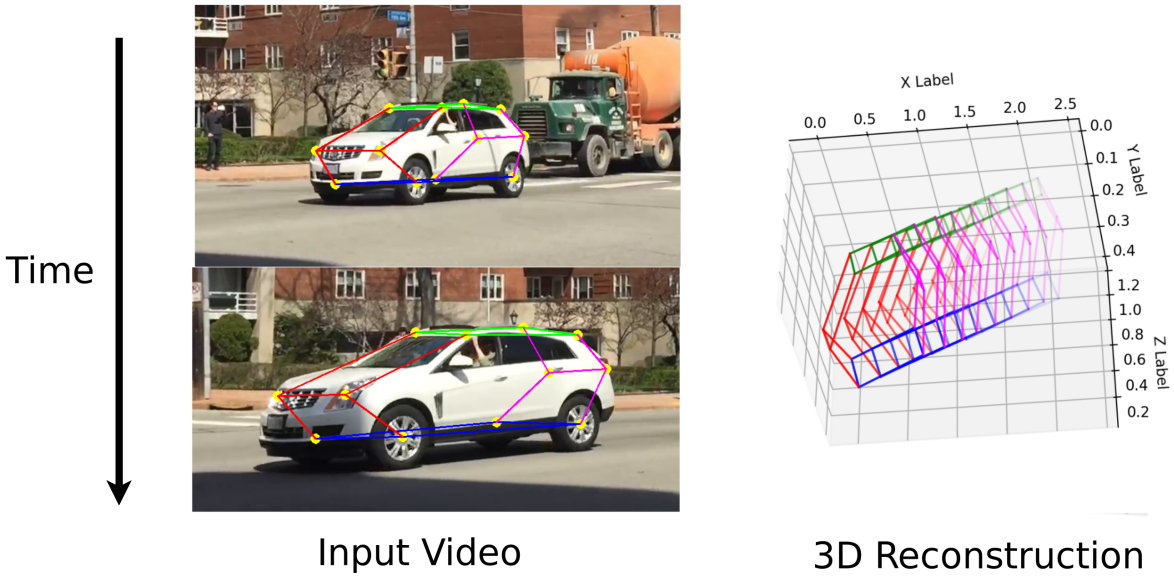


Figure 9: Spatiotemporal reconstruction of the car (right) with the projections at two different time instances in a single view(left)

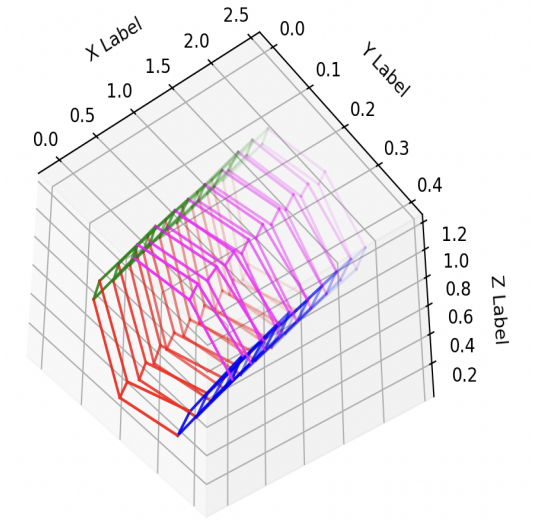
Q6.2 [Extra Credit - 15 points] From the previous question you have done a 3D reconstruction at a time instance. Now you are going to iteratively repeat the process over time and compute a spatio temporal reconstruction of the car. The images in the `data/q6` folder shows the motion of the car at an 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 9.

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

Spatio-temporal reconstruction of the car keypoints:



The function *plot_3d_keypoint_video(pts_3d_video)*

```
def plot_3d_keypoint_video(pts_3d_video):

    # Define figure
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    ax.set_xlabel("X Label")
    ax.set_ylabel("Y Label")
    ax.set_zlabel("Z Label")

    # Plot 3D points over time
    for i in range(len(pts_3d_video)):
        pt_3D = pts_3d_video[i]
        for j in range(len(connections_3d)):
            index0, index1 = connections_3d[j]
            xline = [pt_3D[index0, 0], pt_3D[index1, 0]]
            yline = [pt_3D[index0, 1], pt_3D[index1, 1]]
            zline = [pt_3D[index0, 2], pt_3D[index1, 2]]
            ax.plot(xline, yline, zline, color=colors[j], alpha=i / len(pts_3d_video))
    plt.show()
```

7 Deliverables

The assignment (code and write-up) should be submitted to Gradescope. The write-up should be named `<AndrewId>_hw3.pdf` and the code should be a zip named `<AndrewId>_hw3.zip`. ***Please make sure that you assign the location of answers to each question 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!). You can run the included `checkA4Submission.py` script to ensure that your zip folder structure is correct.

- `<AndrewId>_hw3.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 the 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 **F** before or after calling refineF?

The relationship between **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 **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 **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

[1] David A Forsyth and Jean Ponce. *Computer vision: a modern approach*. prentice hall professional technical reference, 2002.

[2] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Nature, 2022.