HOMEWORK 3: 3D RECONSTRUCTION

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

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

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.

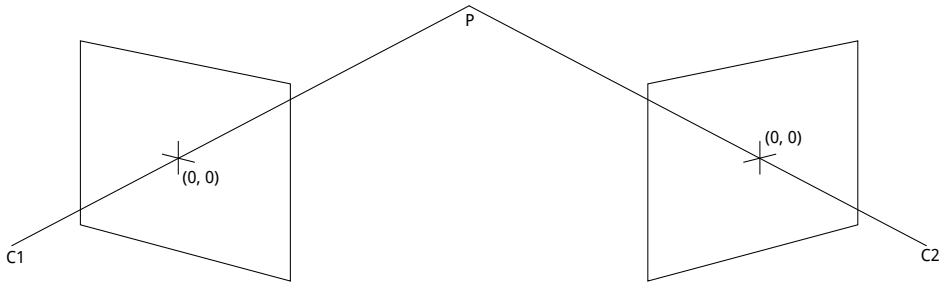

Figure 1: Figure for Q1.1. $C1$ and $C2$ are the optical centers. The principal axes intersect at point $\mathbf{w}$ (P in the figure).

Q1.1

Under normalized image coordinates, both the principal axes of camera C1 and C2 intersect at a 3D point **w**, plus C1 intercepting at image plane at coordinate $x_1 = \begin{bmatrix}0\\0\\1\end{bmatrix}$ (in homogeneous coordinate) and C2 intercepting at image plane at coordinate $x_2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$ (in homogeneous coordinate), then x_1 and x_2 satisify the following equation: $x_2^T F x_1 = 0$, where F = $\begin{bmatrix}F_{11} & F_{12} & F_{13}\\F_{21} & F_{22} & F_{23}\\F_{31} & F_{32} & F_{33}\end{bmatrix}$. When we plug in x_1 and x_2, it shows that $\begin{bmatrix}0 & 0 & 1\end{bmatrix}\begin{bmatrix}F_{11} & F_{12} & F_{13}\\F_{21} & F_{22} & F_{23}\\F_{31} & F_{32} & F_{33}\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix} = 0$, and it results in F33 = 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. Back up your argument with relevant equations. You may assume both cameras have the same intrinsics.

Q1.2

Since the two cameras only differ by a pure translation, the rotation matrix R = I (Identity matrix), and the translation matrix $T = [T_x, 0, 0]^T$. Then we have the following equation: $\begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} K^{-\top} E K^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = 0$, where E = $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -T_x \\ 0 & T_x & 0 \end{bmatrix}$. We can view $K^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix}$ in normalized camer coordinates, then the above equation would be $\begin{bmatrix} x_{n2} & y_{n2} & 1 \end{bmatrix} E \begin{bmatrix} x_{n1} \\ y_{n1} \\ 1 \end{bmatrix} = 0$. We now have epipolar line in C2 as $l_2 = E \begin{bmatrix} x_{n1} \\ y_{n1} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -T_x \\ T_x y_{n1} \end{bmatrix} \rightarrow x_{n2}^T \begin{bmatrix} 0 \\ -T_x \\ T_x y_{n1} \end{bmatrix} = -y_{n2} T_x + T_x y_{n1} = 0$, and epipolar line in C1 as $l_1 = E^T \begin{bmatrix} x_{n2} \\ y_{n2} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ T_x \\ -T_x y_{n2} \end{bmatrix} \rightarrow x_{n1}^T \begin{bmatrix} 0 \\ T_x \\ -T_x y_{n2} \end{bmatrix} = y_{n1} T_x - T_x y_{n2} = 0$. We can see that l_1 and l_2 are both horizontal lines (parallel to the x-axis), and they are parallel to each other.

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 $\mathbf{t}_{rel}$.

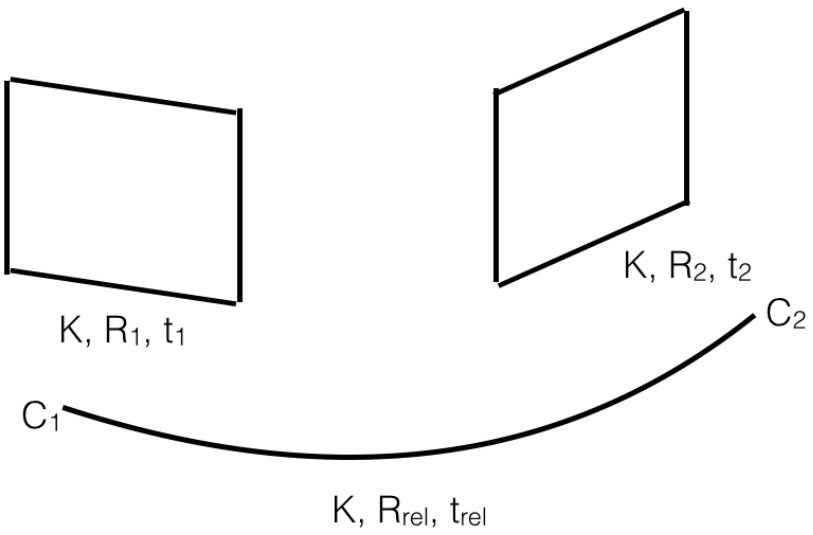

Figure 2: Figure for Q1.3. $C1$ and $C2$ are the optical centers. The rotation and the translation is obtained using inertial sensors. $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$ are the relative rotation and translation between two frames.

Q1.3

Assume P = $\begin{bmatrix} X_P \\ Y_P \\ Z_P \end{bmatrix}$ is the 3D interception point of camera at time 1 and time 2, that is, camera C1 and C2. Then we have image point in C1 and C2 as x_1 and x_2 as the following equations:

$$\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = K(R_1 \begin{bmatrix} X_P \\ Y_P \\ Z_P \end{bmatrix}) + t_1 \tag{1}$$

$$\lambda_2 \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = K(R_2 \begin{bmatrix} X_P \\ Y_P \\ Z_P \end{bmatrix}) + t_2 \tag{2}$$

The λ_1 and λ_2 are the scalar for homogeneous coordinates. When we substitute $\begin{bmatrix} X_P \\ Y_P \\ Z_P \end{bmatrix}$ with $\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$ in equation (2), then we have the following equation:

$$\begin{aligned} \lambda_2 \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} &= K(R_2 R_1^{-1}(K^{-1}\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - t_1) + t_2) \\ &= KR_2R_1^{-1}K^{-1}\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - KR_2R_1^{-1}t_1 + Kt_2 \\ &= KR_{rel}K^{-1}\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} + Kt_{rel} \end{aligned} \tag{3}$$

Accordingly, $R_{rel} = R_2R_1^{-1}$, and $t_{rel} = t_2 - R_2R_1^{-1}t_1$. Also, the essential matrix E is the cross product of t_{rel} and R_{rel}: $E = t_{rel} \times R_{rel}$, and the fundamental matrix F is: $F = K^{-T}(t_{rel} \times R_{rel})K^{-1}$.

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.

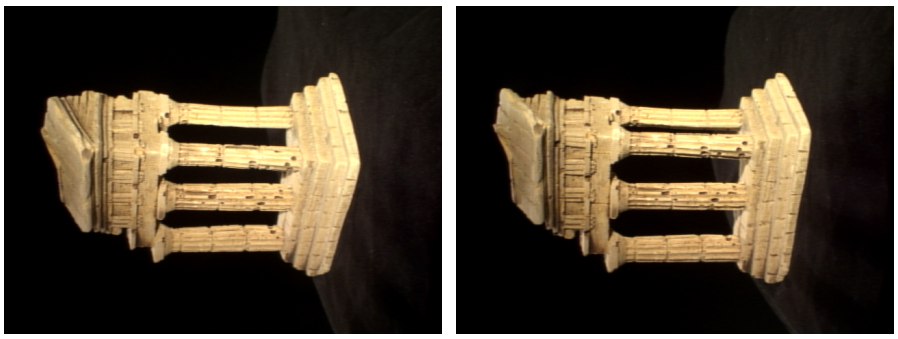

Figure 3: `Temple` images for this assignment

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/

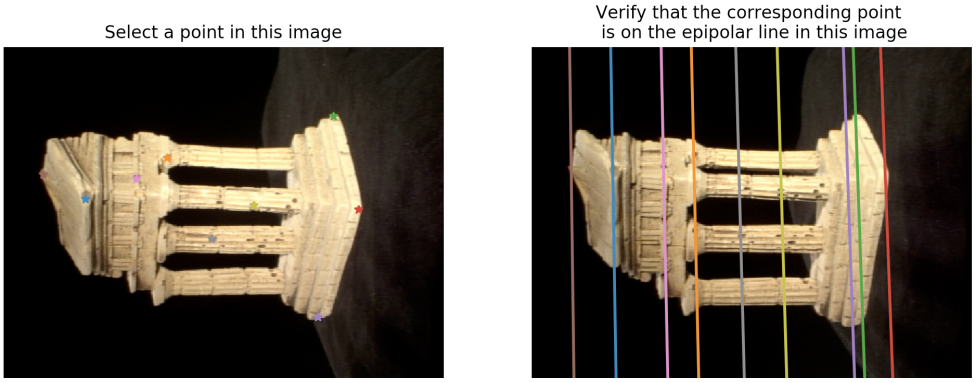

Figure 4: `displayEpipolarF` in `helper.py` creates a GUI for visualizing epipolar lines

- 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

As in the output: $./q2\_1.npz$, the recovered F is:

$$\begin{pmatrix} -2.19293792e^{-07} & 2.95926413e^{-05} & -2.51886251e^{-01} \\ 1.28064423e^{-05} & -6.64493522e^{-07} & 2.63771761e^{-03} \\ 2.42228993e^{-01} & -6.82585388e^{-03} & 1.00000000e^{+00} \end{pmatrix}$$

Besides, M = 640.

The following Figure 5 shows the result of the Eight Point Algorithm:

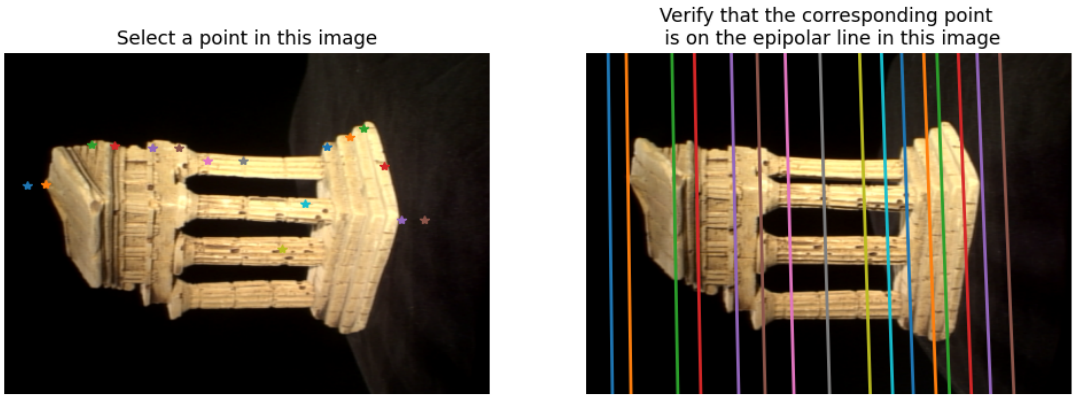

Figure 5: Result of the Eight Point Algorithm

The following Figure 6 and Figure 7 show the code snippet of the Eight Point Algorithm in q2_1_eightpoint.py:

```
"""
Q2.1: Eight Point Algorithm
    Input:  pts1, Nx2 Matrix
            pts2, Nx2 Matrix
            M, a scalar parameter computed as max (imwidth, imheight)
    Output: F, the fundamental matrix

    HINTS:
    (1) Normalize the input pts1 and pts2 using the matrix T.
    (2) Setup the eight point algorithm's equation.
    (3) Solve for the least square solution using SVD.
    (4) Use the function `_singularize` (provided) to enforce the singularity condition.
    (5) Use the function `refineF` (provided) to refine the computed fundamental matrix.
        (Remember to use the normalized points instead of the original points)
    (6) Unscale the fundamental matrix
"""

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

    # Normalizing to [-1, 1] x [-1, 1]
    T_matrix, pts1_norm, pts2_norm = normalizeImagePts(pts1, pts2, M)

    # Setup equations and solve least square solution using SVD
    F_matrix = computeF(pts1_norm, pts2_norm)

    # Enforce singularity condition.
    F_sing = _singularize(F_matrix)

    # Refined the F
    F_refine = refineF(F_sing, pts1_norm, pts2_norm)
    #F_refine = F_sing

    # Unscale the F
    F_unscaled = T_matrix.T @ F_refine @ T_matrix

    # Make sure F is unique avoiding infinite scaling possibilities
    F_ret = F_unscaled/F_unscaled[2, 2]

    return F_ret
```

Figure 6: First Code Snippet

```
def computeF(x1, x2):
    # Construct A matrix
    A = np.zeros((x1.shape[0], 9))
    for i in range(x1.shape[0]):
        xi1 = x1[i][0]
        yi1 = x1[i][1]
        xi2 = x2[i][0]
        yi2 = x2[i][1]
        A[i] = [xi1*xi2, xi2*yi1, xi2, yi2*xi1, yi1*yi2, yi2, xi1, yi1, 1]

    # Solve by directly call SVD
    try:
        u, sigma, v = np.linalg.svd(A)
    except np.linalg.LinAlgError as e:
        eps = 1e-10  # Small regularization constant
        perturbation = eps * np.random.rand(A.shape[0], A.shape[1])
        A_regularized = A + perturbation
        u, sigma, v = np.linalg.svd(A_regularized)

    # Choose the eigenvector w/ smallest eigenvalues
    v1_smallest_eigenvec = v[-1, :]
    F2to1_v1_smallest = np.reshape(v1_smallest_eigenvec, (3, 3))
    F2to1_v1 = np.array(F2to1_v1_smallest, dtype=np.float32)

    return F2to1_v1

def normalizeImagePts(pts1, pts2, M):
    T_matrix = np.zeros((3, 3), dtype=np.float32)
    T_matrix[0][0] = 1/M
    T_matrix[0][2] = 0
    T_matrix[1][1] = 1/M
    T_matrix[1][2] = 0
    T_matrix[2][2] = 1

    '''
    T_matrix = np.zeros((3, 3), dtype=np.float32)
    T_matrix[0][0] = 2/M
    T_matrix[0][2] = -1
    T_matrix[1][1] = 2/M
    T_matrix[1][2] = -1
    T_matrix[2][2] = 1
    '''

    x1_homo = np.hstack([pts1, np.ones((pts1.shape[0], 1))]) #Nx3
    x1_norm = T_matrix@x1_homo.T #3xN
    x1_norm = x1_norm.T #Nx3
    x1_col1 = x1_norm[:, 0]/x1_norm[:, 2]
    x1_col2 = x1_norm[:, 1]/x1_norm[:, 2]
    x1_norm = np.column_stack((x1_col1, x1_col2)) #Nx2

    x2_homo = np.hstack([pts2, np.ones((pts2.shape[0], 1))]) #Nx3
    x2_norm = T_matrix@x2_homo.T #3xN
    x2_norm = x2_norm.T #Nx3
    x2_col1 = x2_norm[:, 0]/x2_norm[:, 2]
    x2_col2 = x2_norm[:, 1]/x2_norm[:, 2]
    x2_norm = np.column_stack((x2_col1, x2_col2)) #Nx2

    return T_matrix, x1_norm, x2_norm
```

Figure 7: Second Code Snippet

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

As in the output: $./q2\_2.npz$, the recovered F is:

$$\begin{pmatrix} 8.10457899e^{-07} & 8.90918106e^{-06} & -2.01028454e^{-01} \\ 2.63329911e^{-05} & -6.00542406e^{-07} & 6.97427365e^{-04} \\ 1.92182078e^{-01} & -4.20123348e^{-03} & 1.00000000e^{+00} \end{pmatrix}$$

Besides, M = 640.

The following Figure 8 shows the result of the Eight Point Algorithm:

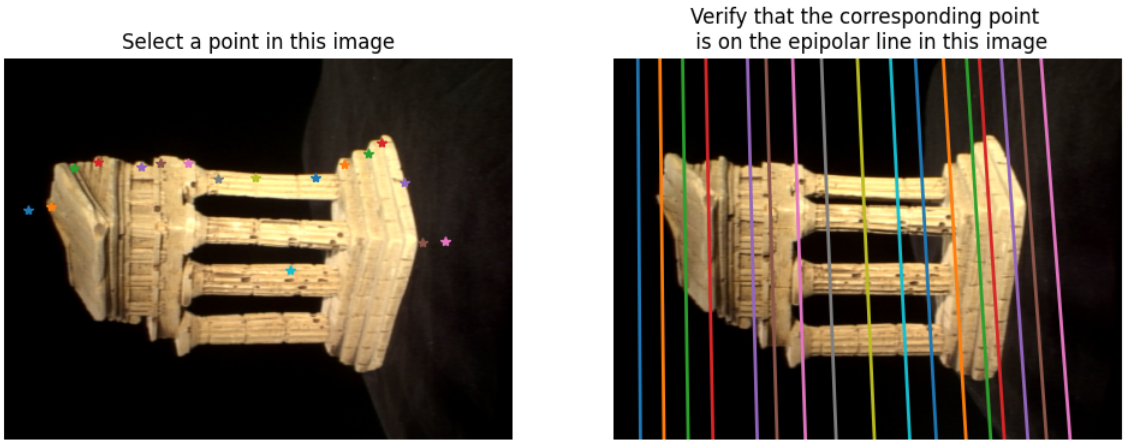

Figure 8: Result of the Seven Point Algorithm

The following Figure 9, Figure 10, Figure 11 and Figure 12 show the code snippet of the Seven Point Algorithm in q2_2_sevenpoint.py:

```
"""
Q2.2: Seven Point Algorithm for calculating the fundamental matrix
    Input:  pts1, 7x2 Matrix containing the corresponding points from image1
            pts2, 7x2 Matrix containing the corresponding points from image2
            M, a scalar parameter computed as max (imwidth, imheight)
    Output: Farray, a list of estimated 3x3 fundamental matrixes.

    HINTS:
    (1) Normalize the input pts1 and pts2 scale paramter M.
    (2) Setup the seven point algorithm's equation.
    (3) Solve for the least square solution using SVD.
    (4) Pick the last two colum vector of vT.T (the two null space solution f1 and f2)
    (5) Use the singularity constraint to solve for the cubic polynomial equation of  F = a*f1 + (1-a)*f2 that leads to
        det(F) = 0. Solving this polynomial will give you one or three real solutions of the fundamental matrix.
        Use np.polynomial.polynomial.polyroots to solve for the roots
    (6) Unscale the fundamental matrixes and return as Farray
"""


def sevenpoint(pts1, pts2, M):
    Farray = []
    # ----- TODO -----
    # YOUR CODE HERE

    # Normalizing to [0, 1] x [0, 1]
    T_matrix, pts1_norm, pts2_norm = normalizeImagePts(pts1, pts2, M)

    # Setup matrix and solve SVD and pick the last two columns
    F_v1_matrix, F_v2_matrix = computeF(pts1_norm, pts2_norm)

    # Calculate c4-c0
    coefficients = calCoeff(F_v1_matrix, F_v2_matrix)

    # Solve for the roots
    roots = np.polynomial.polynomial.polyroots(list(coefficients))

    # Setup equations and solve least square solution using SVD
    F_alpha_array = [Falpha(alpha, F_v1_matrix, F_v2_matrix) for alpha in roots]

    # Enforce singularity condition.
    F_sing_array = []
    for F_matrix in F_alpha_array:
        try:
            F_sing = _singularize(F_matrix)
        except np.linalg.LinAlgError as e:
            F_sing = F_matrix.copy()
        F_sing_array.append(F_sing)
    #F_sing_array = [_singularize(F_matrix) for F_matrix in F_alpha_array]
    #F_sing_array = [F_matrix for F_matrix in F_alpha_array]

    # Refined the F
    F_refine_array = []
    for F_sing in F_sing_array:
        try:
            F_refine = refineF(F_sing, pts1_norm, pts2_norm)
        except np.linalg.LinAlgError as e:
            F_refine = F_sing.copy()
        F_refine_array.append(F_refine)
    #F_refine_array = [refineF(F_sing, pts1_norm, pts2_norm) for F_sing in F_sing_array]
    #F_refine_array = [F_sing for F_sing in F_sing_array]

    # Unscale the F
    F_unscaled_array = [T_matrix.T @ F_refine @ T_matrix for F_refine in F_refine_array]

    # Make sure F is unique avoiding infinite scaling possibilities
    Farray = [F_unscaled/F_unscaled[2, 2] for F_unscaled in F_unscaled_array]

    return Farray
```

Figure 9: First Code Snippet

```
def calCoeff(F_v1_smallest: np.ndarray, F_v2_smallest: np.ndarray) -> np.ndarray :
    # Create det of F(alpha)
    F_p0      = FalphaDet(0 , F_v1_smallest, F_v2_smallest)
    F_p1      = FalphaDet(1 , F_v1_smallest, F_v2_smallest)
    F_n1      = FalphaDet(-1, F_v1_smallest, F_v2_smallest)
    F_p2      = FalphaDet(2 , F_v1_smallest, F_v2_smallest)
    F_n2      = FalphaDet(-2 , F_v1_smallest, F_v2_smallest)


    # Calculate the coefficients for the polynomial equation
    '''
    coefficients = np.zeros(4)
    coefficients[0] = F_p0
    coefficients[2] = (F_p1+F_n1)/2.0 - coefficients[0]
    coefficients[3] = (F_p3 - 9*coefficients[2] - coefficients[0])/15 - (F_p2 - 4*coefficients[2] - coefficients[0])/10
    coefficients[1] = F_p1 - (coefficients[0]+coefficients[2]+coefficients[3])
    '''
    coefficients = np.zeros(4)
    coefficients[0] = F_p0
    coefficients[2] = (F_p1+F_n1)/2.0 - coefficients[0]
    coefficients[3] = (F_p2-F_n2)/12.0 - (F_p1-F_n1)/6.0
    coefficients[1] = (F_p1-F_n1)/2.0 - coefficients[3]

    return coefficients
```

Figure 10: Second Code Snippet

Q2.2 continued

```python
def computeF(x1, x2):
    # Construct A matrix
    A = np.zeros((x1.shape[0], 9))
    for i in range(x1.shape[0]):
        xi1 = x1[i][0]
        yi1 = x1[i][1]
        xi2 = x2[i][0]
        yi2 = x2[i][1]
        A[i] = [xi1*xi2, xi2*yi1, xi2, yi2*xi1, yi1*yi2, yi2, xi1, yi1, 1]

    # Solve by directly call SVD
    try:
        u, sigma, v = np.linalg.svd(A)
    except np.linalg.LinAlgError as e:
        eps = 1e-10  # Small regularization constant
        perturbation = eps * np.random.rand(A.shape[0], A.shape[1])
        A_regularized = A + perturbation
        u, sigma, v = np.linalg.svd(A_regularized)

    v1_smallest_eigenvec = v[-1, :]
    v2_smallest_eigenvec = v[-2, :]

    F2to1_v1_smallest = np.reshape(v1_smallest_eigenvec, (3, 3))
    F2to1_v2_smallest = np.reshape(v2_smallest_eigenvec, (3, 3))
    F2to1_v1 = np.array(F2to1_v1_smallest, dtype=np.float32)
    F2to1_v2 = np.array(F2to1_v2_smallest, dtype=np.float32)

    return F2to1_v1, F2to1_v2
```

Figure 11: Third Code Snippet

```python
def normalizeImagePts(pts1, pts2, M):
    T_matrix = np.zeros((3, 3), dtype=np.float32)
    T_matrix[0][0] = 1/M
    T_matrix[0][2] = 0
    T_matrix[1][1] = 1/M
    T_matrix[1][2] = 0
    T_matrix[2][2] = 1

    '''
    T_matrix = np.zeros((3, 3), dtype=np.float32)
    T_matrix[0][0] = 2/M
    T_matrix[0][2] = -1
    T_matrix[1][1] = 2/M
    T_matrix[1][2] = -1
    T_matrix[2][2] = 1
    '''

    x1_homo = np.hstack([pts1, np.ones((pts1.shape[0], 1))]) #Nx3
    x1_norm = T_matrix@x1_homo.T #3xN
    x1_norm = x1_norm.T #Nx3
    x1_col1 = x1_norm[:, 0]/x1_norm[:, 2]
    x1_col2 = x1_norm[:, 1]/x1_norm[:, 2]
    x1_norm = np.column_stack((x1_col1, x1_col2)) #Nx2

    x2_homo = np.hstack([pts2, np.ones((pts2.shape[0], 1))]) #Nx3
    x2_norm = T_matrix@x2_homo.T #3xN
    x2_norm = x2_norm.T #Nx3
    x2_col1 = x2_norm[:, 0]/x2_norm[:, 2]
    x2_col2 = x2_norm[:, 1]/x2_norm[:, 2]
    x2_norm = np.column_stack((x2_col1, x2_col2)) #Nx2

    return T_matrix, x1_norm, x2_norm
```

Figure 12: Fourth Code Snippet

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

As in the output: $./q3\_1.npz$, the recovered E is:

$$\begin{pmatrix} -5.06923074e^{-01} & 6.86542875e^{+01} & -3.71961318e^{+02} \\ 2.97106690e^{+01} & -1.54718732e^{+00} & 9.68232032e^{+00} \\ 3.72990948e^{+02} & 2.98549953e^{+00} & 1.50354471e^{-01} \end{pmatrix}$$

The following Figure 13 shows the code snippet of the q3_1_essential_matrix.py:

```python
import numpy as np
import matplotlib.pyplot as plt

from q2_1_eightpoint import eightpoint

# Insert your package here


"""
Q3.1: Compute the essential matrix E.
    Input:  F, fundamental matrix
            K1, internal camera calibration matrix of camera 1
            K2, internal camera calibration matrix of camera 2
    Output: E, the essential matrix
"""


def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE
    return K2.T @ F @ K1


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("q3_1.npz", E, F)

    # Simple Tests to verify your implementation:
    assert np.linalg.matrix_rank(E) == 2

```

Figure 13: Code Snippet

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: $\mathbf{M}_1 = \begin{bmatrix}\mathbf{I}|0\end{bmatrix}$ and $\mathbf{M}_2 = \begin{bmatrix}\mathbf{R}|\mathbf{t}\end{bmatrix}$.

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

Assume that $\mathbf{w}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$ is the 3D coordinates at point i, and $\tilde{\mathbf{w}}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}$ is its homogeneous coordinate. Besides, $u_1 = \lambda_1 \begin{bmatrix} u_{xi_1} \\ u_{yi_1} \\ 1 \end{bmatrix}$ is the 2D projected homogeneous coordinates on camera1 and $u_2 = \lambda_2 \begin{bmatrix} u_{xi_2} \\ u_{yi_2} \\ 1 \end{bmatrix}$ is the 2D projected homogeneous coordinates on camera2. Let us define the C1 and C2 camera matrix as the following:

$$C1 = \begin{bmatrix} c_{1_{11}} & c_{1_{12}} & c_{1_{13}} & c_{1_{14}} \\ c_{1_{21}} & c_{1_{22}} & c_{1_{23}} & c_{1_{24}} \\ c_{1_{31}} & c_{1_{32}} & c_{1_{33}} & c_{1_{34}} \end{bmatrix} \tag{4}$$

$$C2 = \begin{bmatrix} c_{2_{11}} & c_{2_{12}} & c_{2_{13}} & c_{2_{14}} \\ c_{2_{21}} & c_{2_{22}} & c_{2_{23}} & c_{2_{24}} \\ c_{2_{31}} & c_{2_{32}} & c_{2_{33}} & c_{2_{34}} \end{bmatrix} \tag{5}$$

Then, we have the following two equations:

$$\lambda_1 \begin{bmatrix} u_{xi_1} \\ u_{yi_1} \\ 1 \end{bmatrix} = \begin{bmatrix} c_{1_{11}} & c_{1_{12}} & c_{1_{13}} & c_{1_{14}} \\ c_{1_{21}} & c_{1_{22}} & c_{1_{23}} & c_{1_{24}} \\ c_{1_{31}} & c_{1_{32}} & c_{1_{33}} & c_{1_{34}} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \tag{6}$$

$$\lambda_2 \begin{bmatrix} u_{xi_2} \\ u_{yi_2} \\ 1 \end{bmatrix} = \begin{bmatrix} c_{2_{11}} & c_{2_{12}} & c_{2_{13}} & c_{2_{14}} \\ c_{2_{21}} & c_{2_{22}} & c_{2_{23}} & c_{2_{24}} \\ c_{2_{31}} & c_{2_{32}} & c_{2_{33}} & c_{2_{34}} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \tag{7}$$

After multiplying both equation (6) and (7) out, dividing x and y coordinates by the z coordinate, and rearranging the equations, we get the following equation:

$$\begin{bmatrix} c_{1_{11}} - u_{xi_1}c_{1_{31}} & c_{1_{12}} - u_{xi_1}c_{1_{32}} & c_{1_{13}} - u_{xi_1}c_{1_{33}} & c_{1_{14}} - u_{xi_1}c_{1_{34}} \\ c_{1_{21}} - u_{yi_1}c_{1_{31}} & c_{1_{22}} - u_{yi_1}c_{1_{32}} & c_{1_{23}} - u_{yi_1}c_{1_{33}} & c_{1_{24}} - u_{yi_1}c_{1_{34}} \\ c_{2_{11}} - u_{xi_2}c_{2_{31}} & c_{2_{12}} - u_{xi_2}c_{2_{32}} & c_{2_{13}} - u_{xi_2}c_{2_{33}} & c_{2_{14}} - u_{xi_2}c_{2_{34}} \\ c_{2_{21}} - u_{yi_2}c_{2_{31}} & c_{2_{22}} - u_{yi_2}c_{2_{32}} & c_{2_{23}} - u_{yi_2}c_{2_{33}} & c_{2_{24}} - u_{yi_2}c_{2_{34}} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} = 0 \tag{8}$$

Accordingly, we have A matrix:

$$A = \begin{bmatrix} c_{1_{11}} - u_{xi_1}c_{1_{31}} & c_{1_{12}} - u_{xi_1}c_{1_{32}} & c_{1_{13}} - u_{xi_1}c_{1_{33}} & c_{1_{14}} - u_{xi_1}c_{1_{34}} \\ c_{1_{21}} - u_{yi_1}c_{1_{31}} & c_{1_{22}} - u_{yi_1}c_{1_{32}} & c_{1_{23}} - u_{yi_1}c_{1_{33}} & c_{1_{24}} - u_{yi_1}c_{1_{34}} \\ c_{2_{11}} - u_{xi_2}c_{2_{31}} & c_{2_{12}} - u_{xi_2}c_{2_{32}} & c_{2_{13}} - u_{xi_2}c_{2_{33}} & c_{2_{14}} - u_{xi_2}c_{2_{34}} \\ c_{2_{21}} - u_{yi_2}c_{2_{31}} & c_{2_{22}} - u_{yi_2}c_{2_{32}} & c_{2_{23}} - u_{yi_2}c_{2_{33}} & c_{2_{24}} - u_{yi_2}c_{2_{34}} \end{bmatrix} \tag{9}$$

Q3.2 continued

The following Figure 14 shows the code snippet of triangulate() in the q3_2triangulate.py:

```python
"""
Q3.2: Triangulate a set of 2D coordinates in the image to a set of 3D points.
    Input:  C1, the 3x4 camera matrix
            pts1, the Nx2 matrix with the 2D image coordinates per row
            C2, the 3x4 camera matrix
            pts2, the Nx2 matrix with the 2D image coordinates per row
    Output: P, the Nx3 matrix with the corresponding 3D points per row
            err, the reprojection error.

    Hints:
    (1) For every input point, form A using the corresponding points from pts1 & pts2 and C1 & C2
    (2) Solve for the least square solution using np.linalg.svd
    (3) Calculate the reprojection error using the calculated 3D points and C1 & C2 (do not forget to convert from
        homogeneous coordinates to non-homogeneous ones)
    (4) Keep track of the 3D points and projection error, and continue to next point
    (5) You do not need to follow the exact procedure above.
"""


def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE
    # Form Matrix A
    P = np.zeros((pts1.shape[0], 3), dtype=np.float32) # the reconstructed Nx3 3D points
    err = 0.0
    for i in range(pts1.shape[0]):
        A = np.zeros((4, 4), dtype=np.float32)
        uxi1 = pts1[i, 0]
        uyi1 = pts1[i, 1]
        uxi2 = pts2[i, 0]
        uyi2 = pts2[i, 1]

        A[0] = [C1[0][0]-uxi1*C1[2][0], C1[0][1]-uxi1*C1[2][1], C1[0][2]-uxi1*C1[2][2], C1[0][3]-uxi1*C1[2][3]]
        A[1] = [C1[1][0]-uyi1*C1[2][0], C1[1][1]-uyi1*C1[2][1], C1[1][2]-uyi1*C1[2][2], C1[1][3]-uyi1*C1[2][3]]
        A[2] = [C2[0][0]-uxi2*C2[2][0], C2[0][1]-uxi2*C2[2][1], C2[0][2]-uxi2*C2[2][2], C2[0][3]-uxi2*C2[2][3]]
        A[3] = [C2[1][0]-uyi2*C2[2][0], C2[1][1]-uyi2*C2[2][1], C2[1][2]-uyi2*C2[2][2], C2[1][3]-uyi2*C2[2][3]]

        # Solve by directly call SVD
        try:
            u, sigma, v = np.linalg.svd(A)
        except np.linalg.LinAlgError as e:
            eps = 1e-10  # Small regularization constant
            perturbation = eps * np.random.rand(A.shape[0], A.shape[1])
            A_regularized = A + perturbation
            u, sigma, v = np.linalg.svd(A_regularized)

        v1_smallest_eigenvec = v[-1, :].reshape(1, -1) # the solution of wi
        P[i] = v1_smallest_eigenvec[:, 0:3] / v1_smallest_eigenvec[0, -1] #back from homogeneous coordinates to normal coordinates


    # Calculat the error
    w_homo = np.hstack([P, np.ones((P.shape[0], 1))]) #Nx4
    w1_proj = C1@w_homo.T #3x4*4xN
    w1_proj = w1_proj.T #Nx3
    x1_col1 = w1_proj[:, 0]/w1_proj[:, 2]
    x1_col2 = w1_proj[:, 1]/w1_proj[:, 2]
    x1  = np.column_stack((x1_col1, x1_col2)) #Nx2
    w2_proj = C2@w_homo.T #3x4*4xN
    w2_proj = w2_proj.T #Nx3
    x2_col1 = w2_proj[:, 0]/w2_proj[:, 2]
    x2_col2 = w2_proj[:, 1]/w2_proj[:, 2]
    x2  = np.column_stack((x2_col1, x2_col2)) #Nx2
    err = np.sum(np.linalg.norm(pts1-x1, axis=1)**2) + np.sum(np.linalg.norm(pts2-x2, axis=1)**2)

    return P, err
```

Figure 14: Code Snippet

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 following Figure 15 shows the code snippet of findM2() in the q3_2triangulate.py:

```python
def findM2(F, pts1, pts2, intrinsics, filename="q3_3.npz"):
    """
    Q2.2: Function to find 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'.

    """
    # ----- TODO -----
    # YOUR CODE HERE
    # Calculate E from F and K1/K2
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    E = essentialMatrix(F, K1, K2)

    # Calcualte M1 & M2
    M1 = np.concatenate([np.eye(3), np.zeros((3, 1))], axis=1)  #[I|0]
    M2s = camera2(E)
    C1 = K1 @ M1

    # Select the M2 that has most positive depth points
    max_posdepth_num = -1
    #min_error = np.inf
    M2_ret = M2s[:, :, 0]
    P_ret  = np.zeros((pts1.shape[0], 3))
    for index in range(M2s.shape[-1]):
        M2 = M2s[:, :, index]
        C2 = K2 @ M2
        P, error = triangulate(C1, pts1, C2, pts2)

        '''
        if error < min_error:
            min_error = error
            M2_ret = M2.copy()
            P_ret = P.copy()
        '''
        posdepth_num = np.sum((P[:, 2] > 0))
        if posdepth_num > max_posdepth_num:
            max_posdepth_num = posdepth_num
            M2_ret = M2.copy()
            P_ret = P.copy()

    C2_ret = K2 @ M2_ret

    # Write file to q3_3.npz
    np.savez(filename, M2_ret, C2_ret, P_ret)
    return M2_ret, C2_ret, P_ret
```

Figure 15: Code Snippet

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 16.

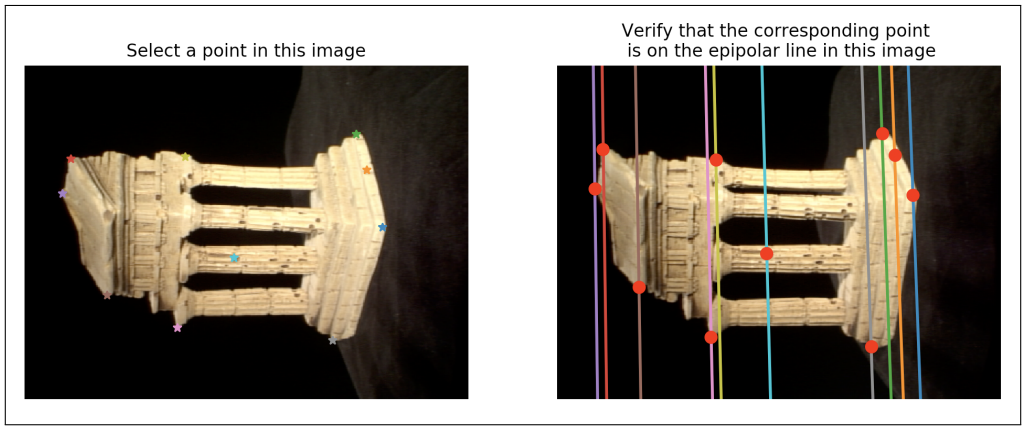

Figure 16: `epipolarMatchGUI` shows the corresponding point found by calling `epipolarCorrespondence`

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 following Figure 17 shows the screenshot of epipolarMatchGUI() with some detected correspondences using the epipolarCorrespondence() function:

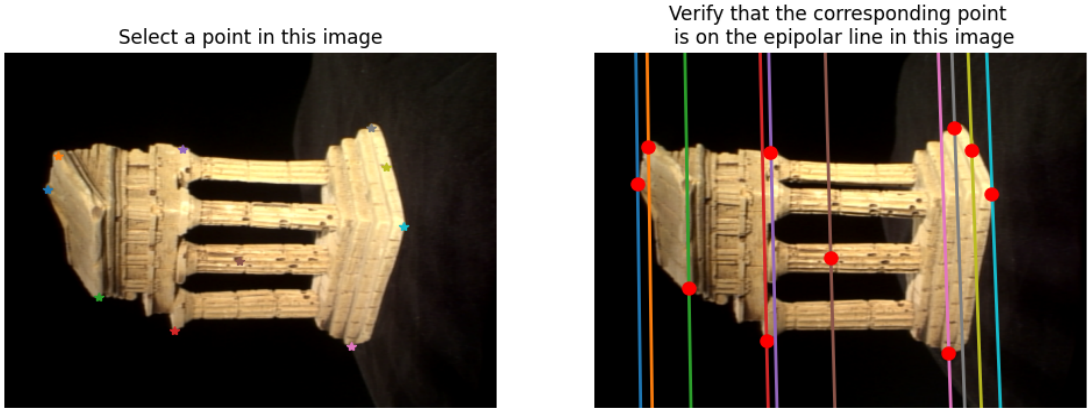

Figure 17: Result of the epipolarCorrespondence() Function

Q4.1 continued

The following Figure 18 shows the code snippet of the epipolarCorrespondence() function in q4_1_epipolar_correspondence.py:

```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE
    # Construct l2 line equation
    x1_pt = np.array([x1, y1, 1]).reshape(-1, 1)
    l2_line = F @ x1_pt

    # Extract points in im2 along the l2 line
    x2_pts = extractPointsAlongLine(l2_line, im_width=im2.shape[1], im_height=im2.shape[0])

    # Extract x1_roi
    ksize = 71
    #print(f"-------------------------------------------(x1, y1) = {(x1, y1)}")
    x1_roi = extractROI(image=im1, x=x1, y=y1, ksize=ksize)

    # Get the best pt im im2
    best_err  = np.inf
    best_cand = (None, None)
    for x2_pt_candi in x2_pts:
        #print(f"==========")
        # Extract x2_roi
        x2_roi = extractROI(image=im2, x=x2_pt_candi[0], y=x2_pt_candi[1], ksize=ksize)

        # Apply Gaussian weighting
        gaussien2DKernel = getGaussianWeight(ksize=ksize, sigma=5.0, print_verbose=False)
        #x1_gauss = applyGaussianKernel(x1_roi, gaussian2DKernel=gaussien2DKernel)
        #x2_gauss = applyGaussianKernel(x2_roi, gaussian2DKernel=gaussien2DKernel)

        # Calculate the similarity
        err = calculateError(x1_roi=x1_roi, x2_roi=x2_roi, kernel=gaussien2DKernel)
        #print(f"x2_pt_candi, err = {x2_pt_candi}, {err}")

        # Get the best
        if err < best_err:
            best_err = err
            best_cand = x2_pt_candi

    #x2, y2 = best_cand
    #error = np.linalg.norm(np.array([x2, y2]) - np.array([x1, y1]))
    #print(f"error = {error}")
    #print(f"best_cand = {best_cand}")
    return best_cand
```

Figure 18: Code Snippet of the epipolarCorrespondence() Function

Q4.2 [10 points] Included in this homework is a file `data/templeCoords.npz` which contains 288

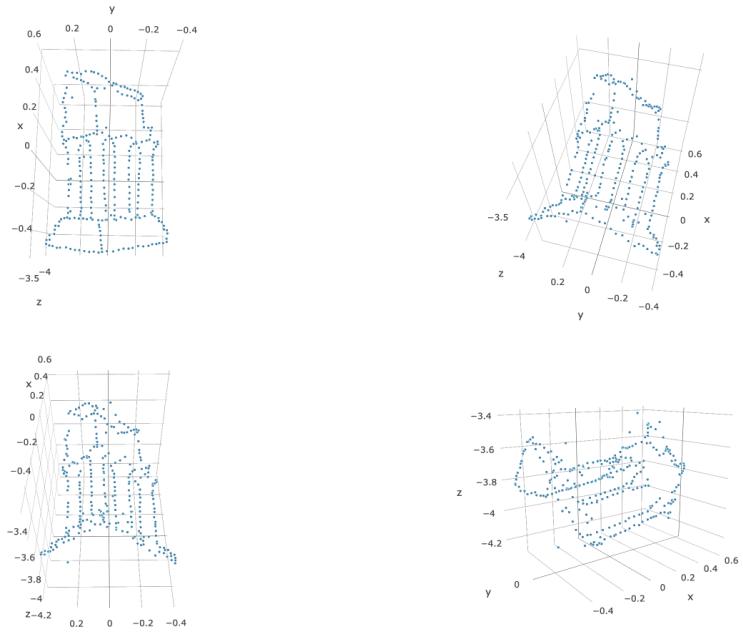

Figure 19: An example point cloud

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 19.

Output: Again, save the matrix $\mathbf{F}$, matrices $\mathbf{M}1, \mathbf{M}2, \mathbf{C}1, \mathbf{C}2$ 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 following Figure 20, Figure 21, Figure 22 and Figure 23 show the results of the 3D visualization results of the temple:

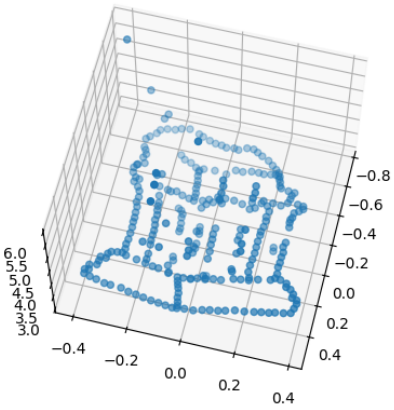

Figure 20: First Screenshot

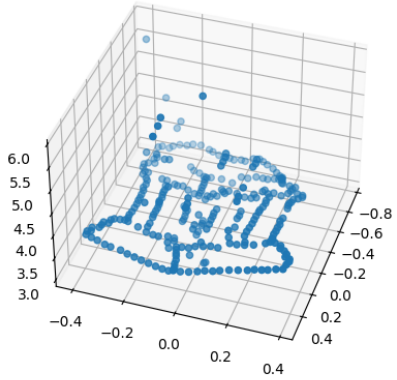

Figure 21: Second Screenshot

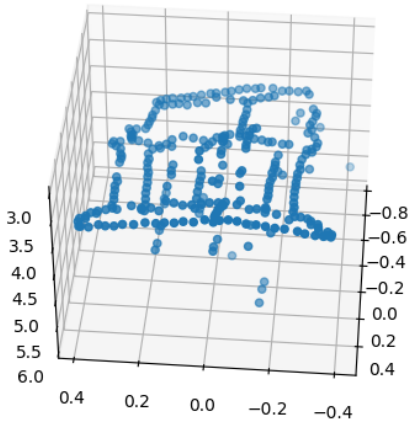

Figure 22: Third Screenshot

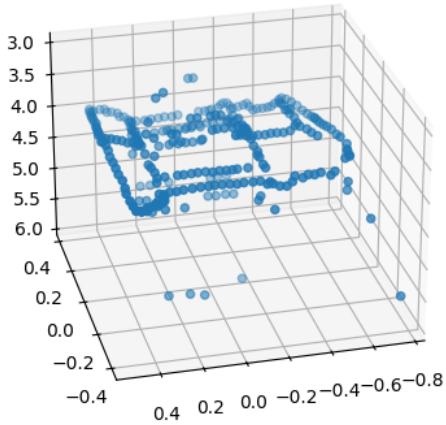

Figure 23: Fourth Screenshot

Q4.2 continued

The following Figure 24 shows the code snippet of the compute3D_pts() function in q4_2_visualize.py:

```python
"""
Q4.2: Finding the 3D position of given points based on epipolar correspondence and triangulation
    Input:  temple_pts1, chosen points from im1
            intrinsics, the intrinsics dictionary for calling epipolarCorrespondence
            F, the fundamental matrix
            im1, the first image
            im2, the second image
    Output: P (Nx3) the recovered 3D points

    Hints:
    (1) Use epipolarCorrespondence to find the corresponding point for [x1 y1] (find [x2, y2])
    (2) Now you have a set of corresponding points [x1, y1] and [x2, y2], you can compute the M2
        matrix and use triangulate to find the 3D points.
    (3) Use the function findM2 to find the 3D points P (do not recalculate fundamental matrices)
    (4) As a reference, our solution's best error is around ~2200 on the 3D points.

    Modified by Vineet Tambe, 2023.
"""


def compute3D_pts(temple_pts1, intrinsics, F, im1, im2, filename='q4_2.npz'):
    # ----- TODO -----
    # YOUR CODE HERE
    # Calcualte M1 & C1
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    M1 = np.concatenate([np.eye(3), np.zeros((3, 1))], axis=1)  #[I|0]
    C1 = K1 @ M1
    # Get corresponding point [x2, y2]
    temple_pts2 = np.zeros(temple_pts1.shape, dtype=int)
    for index, (x1, y1) in enumerate(temple_pts1):
        temple_pts2[index] = epipolarCorrespondence(im1, im2, F, x1, y1)

    # Computer M2 matrix
    M2, C2, P = findM2(F, temple_pts1, temple_pts2, intrinsics, filename="q4_2_3_3.npz")

    # Write file to filename
    np.savez(filename, F, M1, M2, C1, C2)
    return P
```

Figure 24: Code Snippet of the compute3D_pts() Function

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 following Figure 25 shows the screenshot of RANSAC Seven Point algorithm result:

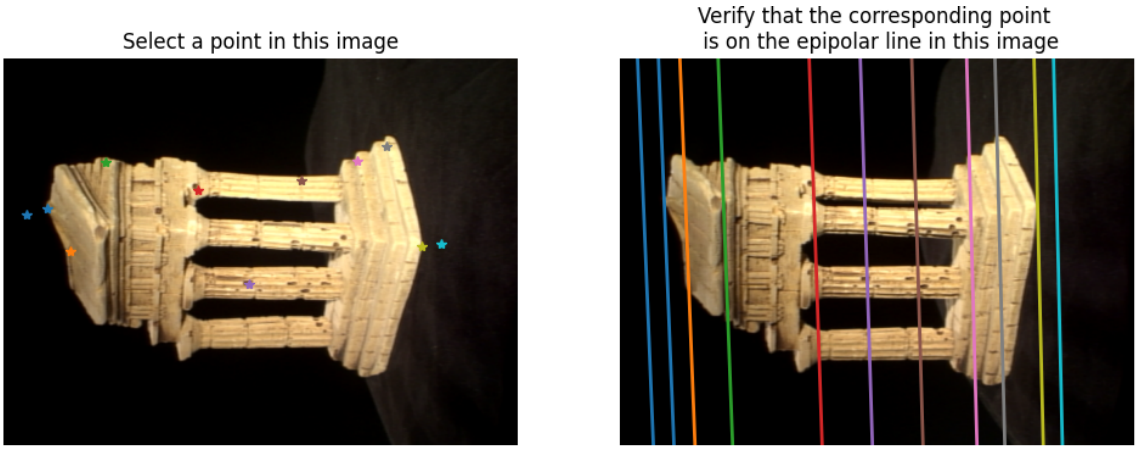

Figure 25: Result of the RANSAC Seven Point Algorithm

The following Figure 26 shows the screenshot of Eight Point algorithm result:

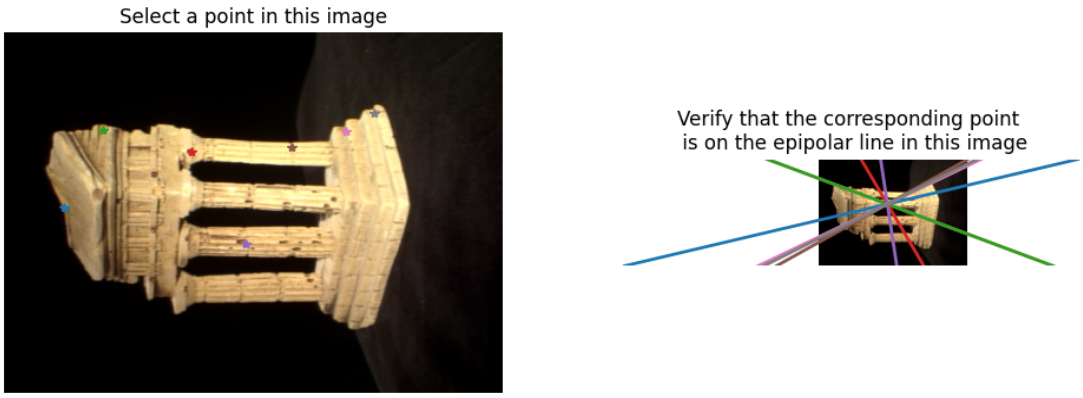

Figure 26: Result of the Eight Point Algorithm

In RANSAC Seven Point algorithm, I use nIters=1000, tol=2 to get above result, and the inlier number is 106, which is around 75% with total number of points is 140. The reason that the RANSAC Seven Point algorithm gets better result is that the points are noisy, and it is stated in the hw3.pdf that around 75% of these noisy points are inliers. While RANSAC Seven Point algorithm has the chance to reject outliers, the Eight Point algorithm can only use all the points, including outliers, to calculate the Fundamental Matrix, F. Accordingly, RANSAC Seven Point algorithm performs better than Eight Point algorithm under noisy data.

Q5.1 continued

The following shows the results of changing the value of the 'tol' parameter.

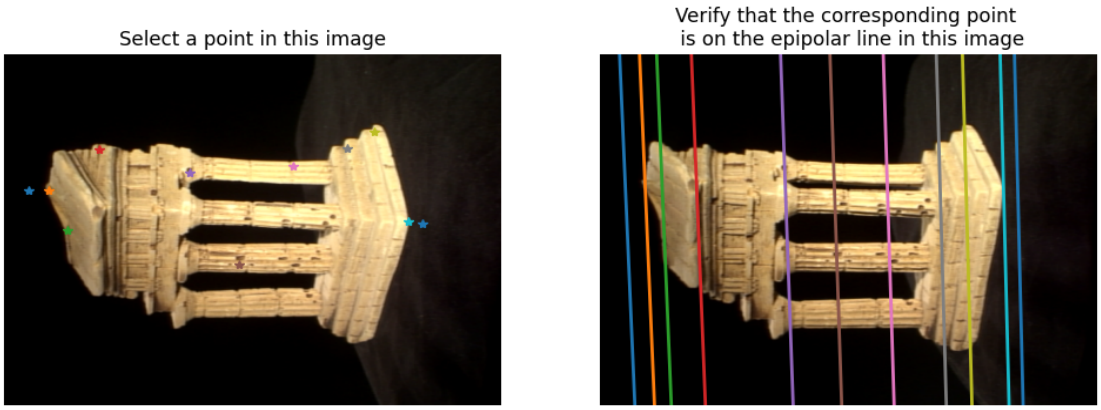

Figure 27: Result of the RANSAC Seven Point Algorithm w/ tol=0.5, inlier rate=52.86%

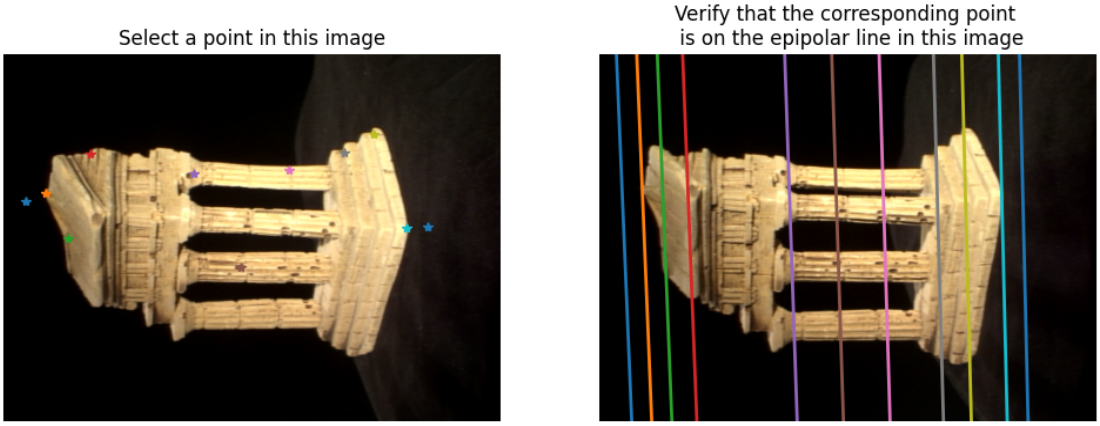

Figure 28: Result of the RANSAC Seven Point Algorithm w/ tol=1, inlier rate=65.71%

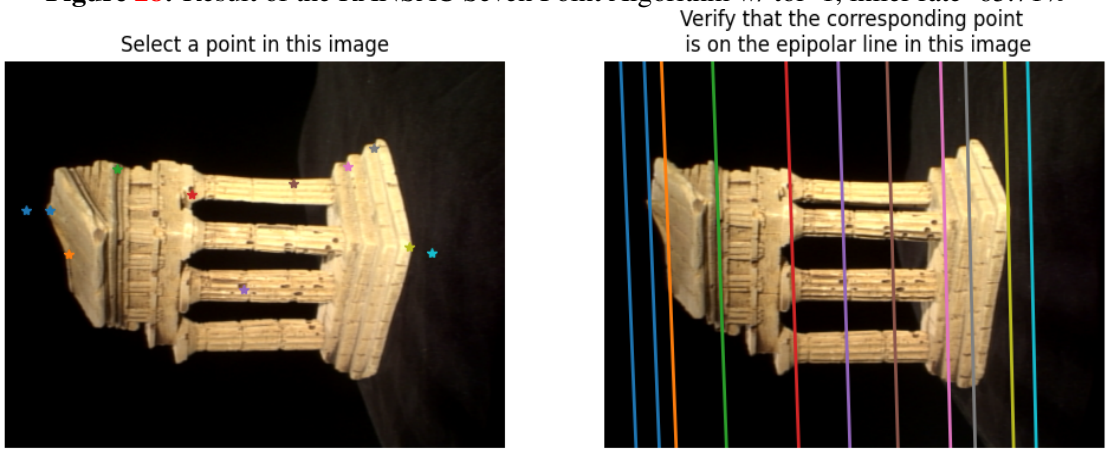

Figure 29: Result of the RANSAC Seven Point Algorithm w/ tol=4, inlier rate=76.43%

Q5.1 continued

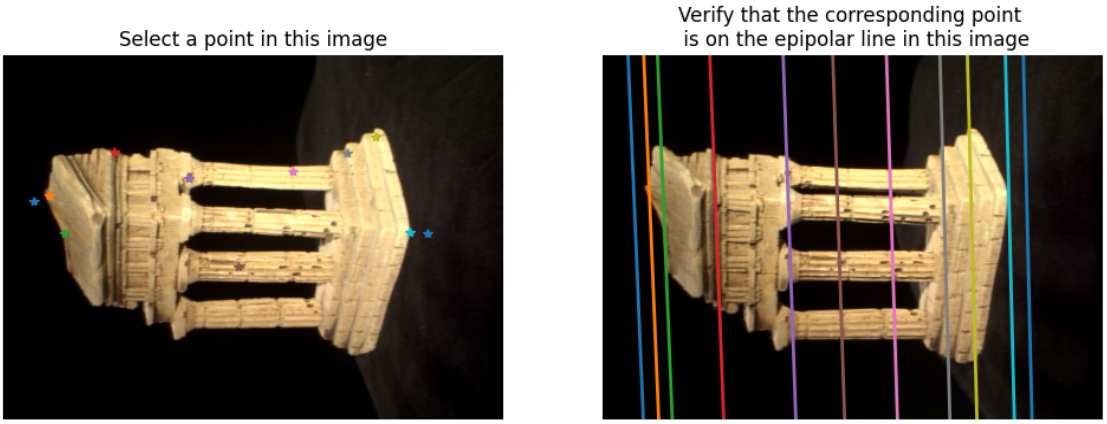

Figure 30: Result of the RANSAC Seven Point Algorithm w/ tol=8, inlier rate=78.57%

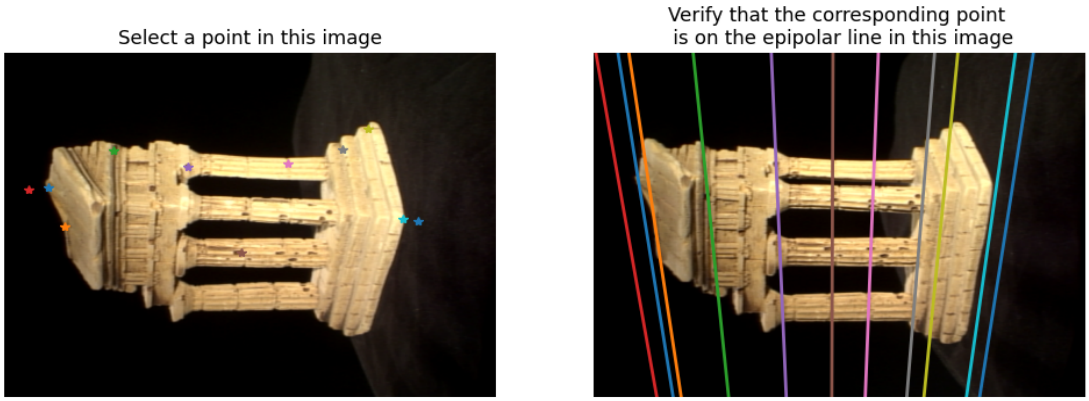

Figure 31: Result of the RANSAC Seven Point Algorithm w/ tol=16, inlier rate=79.29%

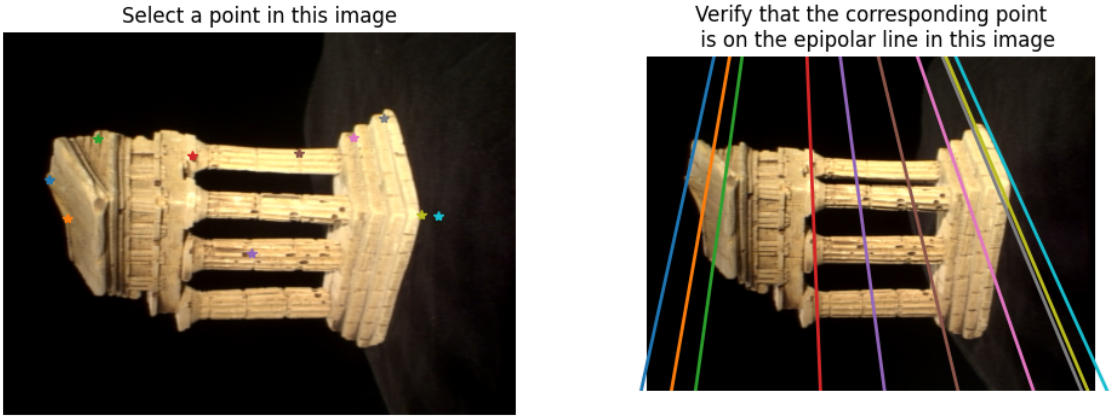

Figure 32: Result of the RANSAC Seven Point Algorithm w/ tol=128, inlier rate=80%

Q5.1 continued

As we can see from the results above that when tol is getting larger, specifically larger than 8, the result is getting worse. It is because the criteria to count a point as an inlier is getting laxer, and some of the outlier points are wrongly categorized as inliers. The resulted F is generated by some outliers and thus becomes worse. As for the necessary iteration number of RANSAC to return a good model, the value is calculated by k = $\frac{\log(1-p)}{\log(1-w^n)}$, where p = 0.999, w is about 0.75 (from hw3.pdf), and n=7, then k = 48.21, so the nIters=1000, as I set in the program for above results, should be sufficient to have RANSAC Sevent Point algorithm return a good model.

The following Figure 33 shows the code snippet of the ransacF() function in q5_bundle_adjustment.py:

```
"""
Q5.1: RANSAC method.
    Input:  pts1, Nx2 Matrix
            pts2, Nx2 Matrix
            M, a scaler parameter
            nIters, Number of iterations of the Ransac
            tol, tolerence for inliers
    Output: F, the fundamental matrix
            inliers, Nx1 bool vector set to true for inliers

    Hints:
    (1) You can use the calc_epi_error from q1 with threshold to calcualte inliers. Tune the threshold based on
        the results/expected number of inliners. You can also define your own metric.
    (2) Use the seven point alogrithm to estimate the fundamental matrix as done in q1
    (3) Choose the resulting F that has the most number of inliers
    (4) You can increase the nIters to bigger/smaller values

"""
def getFSevenPoint(pts1_choice, pts2_choice, pts1_homo, pts2_homo, M, tol, file="debug.log"):
    best_inlier = np.zeros((pts1_homo.shape[0], 1), dtype=bool)
    best_F = None
    best_inlier_cnt = -1

    Fs = sevenpoint(pts1_choice, pts2_choice, M)
    for F in Fs:
        inlier = np.zeros((pts1_homo.shape[0], 1), dtype=bool)
        error_all = np.abs(calc_epi_error(pts1_homo, pts2_homo, F))
        inlier[error_all < tol] = True
        inlier_cnt = np.sum(inlier)

        if inlier_cnt > best_inlier_cnt:
            best_inlier_cnt = inlier_cnt
            best_F = F.copy()
            best_inlier = inlier.copy()

    return best_F, best_inlier, best_inlier_cnt

def ransacF(pts1, pts2, M, nIters=1000, tol=10):
    # TODO: Replace pass by your implementation
    pts1_homo = np.hstack((pts1, np.ones((pts1.shape[0], 1))))
    pts2_homo = np.hstack((pts2, np.ones((pts2.shape[0], 1))))
    best_inlier = np.zeros((pts1_homo.shape[0], 1), dtype=bool)
    best_F = None
    best_inlier_cnt = -1

    file = open("debug.log", "w")
    for i in range(nIters):
        inlier = np.zeros((pts1.shape[0], 1), dtype=bool)
        choice = np.random.choice(range(pts1.shape[0]), 7)
        pts1_choice = pts1[choice, :]
        pts2_choice = pts2[choice, :]
        cur_F, cur_inlier, cur_inlier_cnt = getFSevenPoint(pts1_choice, pts2_choice, pts1_homo, pts2_homo, M, tol, file=file)

        if cur_inlier_cnt > best_inlier_cnt:
            best_inlier_cnt = cur_inlier_cnt
            best_F = cur_F.copy()
            best_inlier = cur_inlier.copy()

    return best_F, best_inlier
```

Figure 33: Code Snippet of the ransacF() Function

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 following Figure 34, Figure 35 show the code snippet of rodrigues() and invRodrigues() functions respectively:

```python
"""
Q5.2: Rodrigues formula.
    Input:  r, a 3x1 vector
    Output: R, a rotation matrix
"""


def rodrigues(r, epsilon=1e-20):
    # TODO: Replace pass by your implementation
    r_reshape = r.reshape((-1, 1))
    theta = np.linalg.norm(r_reshape) #floating point value
    if theta < epsilon: # == 0
        return np.eye(3)

    u = r_reshape/theta
    u_cross = np.array([[ 0      , -u[2, 0],  u[1, 0]],
                        [ u[2, 0],  0      , -u[0, 0]],
                        [-u[1, 0],  u[0, 0],  0      ]])
    R = np.eye(3)*np.cos(theta) + (1-np.cos(theta))*(u@u.T) + u_cross*np.sin(theta)
    return R

```

Figure 34: Code Snippet of rodrigues().

Figure 35: Code Snippet of invRodrigues().

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

The following Figure 36, Figure 37, Figure 38, and Figure 39 show the results of the optimization.

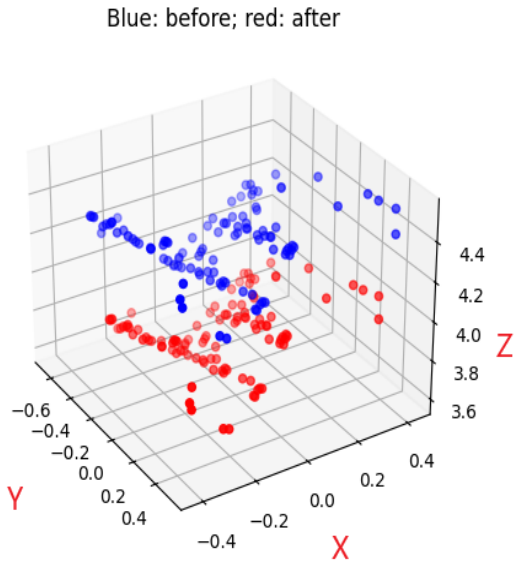

Figure 36: Perspective 1 of the Result of Optimization

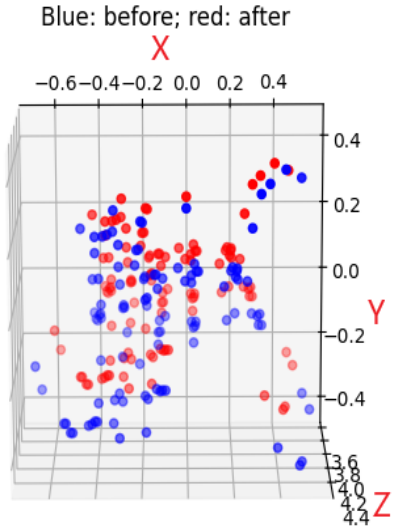

Figure 37: Perspective 2 of the Result of Optimization

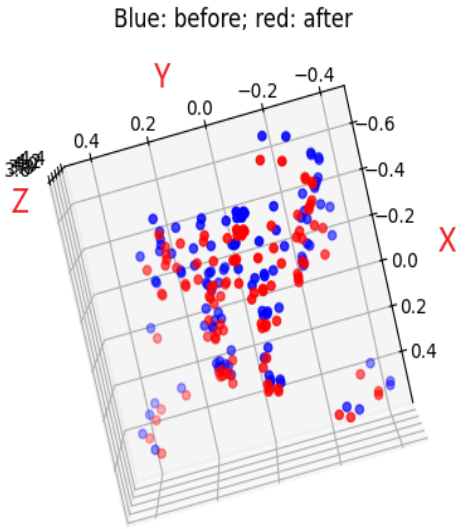

Figure 38: Perspective 3 of the Result of Optimization

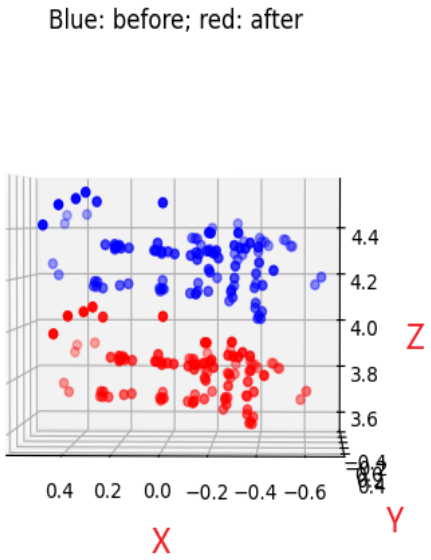

Figure 39: Perspective 4 of the Result of Optimization

Q5.3 continued

The reprojection error with initial M2 and w is 874.0899779851011.
The reprojection error with optimized M2 and w is 8.08757938434745.
The following Figure 40, Figure 41, and Figure 42 show the code snippets.

```python
# Visualization:
np.random.seed(1)
correspondence = np.load(
    "data/some_corresp_noisy.npz"
)  # Loading noisy 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")
M = np.max([*im1.shape, *im2.shape])

# TODO: YOUR CODE HERE
"""
Call the ransacF function to find the fundamental matrix
Call the findM2 function to find the extrinsics of the second camera
Call the bundleAdjustment function to optimize the extrinsics and 3D points
Plot the 3D points before and after bundle adjustment using the plot_3D_dual function
"""
#Step1: Call the ransacF function
F, inliers = ransacF(pts1, pts2, M=np.max([*im1.shape, *im2.shape]), nIters=1000, tol=2)
print(f"number of inliers = {np.sum(inliers)}\ntotal number of points = {pts1.shape[0]}\ninlier rate = {np.sum(inliers)/pts1.shape[0]*100}%")

#displayEpipolarF(im1, im2, F)
np.savez('q5_3.npz', F, inliers)
#F = np.load('q5_3.npz')['arr_0']
#inliers = np.load('q5_3.npz')['arr_1']
#displayEpipolarF(im1, im2, F)
#print(f"F = {F}")

#Step2: Call the findM2 function to find the M (extrinsics) of the second camera
p1, p2 = pts1[inliers.flatten()], pts2[inliers.flatten()]
M2_init, C2_init, P_init = findM2(F, p1, p2, intrinsics, filename="q5_3_findM2.npz")
M1 = np.concatenate([np.eye(3), np.zeros((3, 1))], axis=1)  #[I|0]

#Step3: Call the bundleAdjustment function to optimize the extrinsics and 3D points
M2_opt, P_opt, object_start, object_end = bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init)
print(f"Reprojection error with initial M2 and w = {object_start}")
print(f"Optimized Reprojection error with optimized M2 and w = {object_end}")

#Step4: Plot the 3D points before and after bundle adjustment using the plot_3D_dual function
fig = plt.figure()
ax = Axes3D(fig)
plot_3D_dual(P_init, P_opt)
```

Figure 40: Code Snippet of Main (Test) Function.

Q5.3 continued

```python
"""
Q5.3 Bundle adjustment.
    Input:  K1, the intrinsics of camera 1
            M1, the extrinsics of camera 1
            p1, the 2D coordinates of points in image 1
            K2,  the intrinsics of camera 2
            M2_init, the initial extrinsics of camera 1
            p2, the 2D coordinates of points in image 2
            P_init, the initial 3D coordinates of points
    Output: M2, the optimized extrinsics of camera 1
            P2, the optimized 3D coordinates of points
            o1, the starting objective function value with the initial input
            o2, the ending objective function value after bundle adjustment

    Hints:
    (1) Use the scipy.optimize.minimize function to minimize the objective function, rodriguesResidual.
        You can try different (method='..') in scipy.optimize.minimize for best results.
"""

def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    obj_start = obj_end = 0
    # ----- TODO -----
    # YOUR CODE HERE
    # Extract r2, t2 from M2_init
    R2 = M2_init[:, :3] # 3x3
    r2 = invRodrigues(R2, 1e-32).reshape(-1, 1) #3x1
    t2 = M2_init[:, 3].reshape(-1, 1) #3x1
    init_params = np.concatenate([P_init.flatten(), r2.flatten(), t2.flatten()])

    # Calculate the starting object loss
    obj_start = np.sum(rodriguesResidual(K1, M1, p1, K2, p2, init_params) ** 2)

    # Optimization
    object_func = lambda params: np.sum(rodriguesResidual(K1, M1, p1, K2, p2, params) ** 2)
    result = scipy.optimize.minimize(
        fun=object_func,          # Objective function (residual function)
        x0=init_params            # Initial guess for parameters
        #method='L-BFGS-B'          # Optimization method (you can try different methods)
    )
    optimized_params = result.x

    # Extract optimized r2, t2, P, and M2
    t2_optimized = optimized_params[-3:].reshape(-1, 1) #3x1
    r2_optimized = optimized_params[-6:-3].reshape(-1, 1) #3x1
    P_optimized  = optimized_params[:-6].reshape(-1, 3) #Nx3
    R2_optimized = rodrigues(r2_optimized)
    M2           = np.hstack((R2_optimized, t2_optimized))

    # Calculate the resulting object loss
    obj_end = np.sum(rodriguesResidual(K1, M1, p1, K2, p2, optimized_params) ** 2)

    return M2, P_optimized, obj_start, obj_end
```

Figure 41: Code Snippet of bundleAdjustment()

```python
"""
Q5.3: Rodrigues residual.
    Input:  K1, the intrinsics of camera 1
            M1, the extrinsics of camera 1
            p1, the 2D coordinates of points in image 1
            K2, the intrinsics of camera 2
            p2, the 2D coordinates of points in image 2
            x, the flattened concatenationg of P, r2, and t2.
    Output: residuals, 4N x 1 vector, the difference between original and estimated projections
"""


def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # TODO: Replace pass by your implementation
    P = x[:-6].reshape(-1, 3) #Nx3
    r2 = x[-6:-3].reshape(-1, 1) #3x1
    t2 = x[-3:].reshape(-1, 1) #3x1

    # Reconstruct C1
    C1 = K1@M1

    # Reconstruct C2
    R2 = rodrigues(r2)
    M2 = np.hstack((R2, t2))
    C2 = K2@M2

    # Calculate projected points on C1 and C2
    P_homo = np.hstack((P, np.ones((P.shape[0], 1)))) #Nx4
    p1_proj_homo = C1@P_homo.T #3xN
    p2_proj_homo = C2@P_homo.T #3xN

    # Convert Homogeneous back to normal coordinates
    p1_hat_x = p1_proj_homo.T[:, 0]/p1_proj_homo.T[:, 2]
    p1_hat_y = p1_proj_homo.T[:, 1]/p1_proj_homo.T[:, 2]
    p1_hat   = np.column_stack((p1_hat_x, p1_hat_y)) #Nx2
    p2_hat_x = p2_proj_homo.T[:, 0]/p2_proj_homo.T[:, 2]
    p2_hat_y = p2_proj_homo.T[:, 1]/p2_proj_homo.T[:, 2]
    p2_hat   = np.column_stack((p2_hat_x, p2_hat_y)) #Nx2

    # Calculate the residual 4Nx1 vector
    residual = np.concatenate([(p1-p1_hat).reshape(-1), (p2-p2_hat).reshape(-1)])

    return residual
```

Figure 42: Code Snippet of rodriguesResidual()

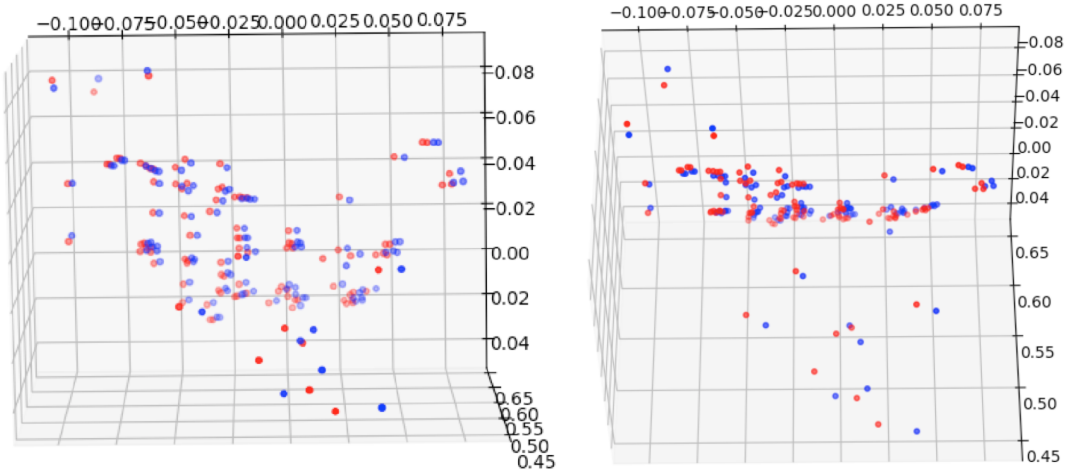

Figure 43: Visualization of 3D points for noisy correspondences before and after using bundle adjustment

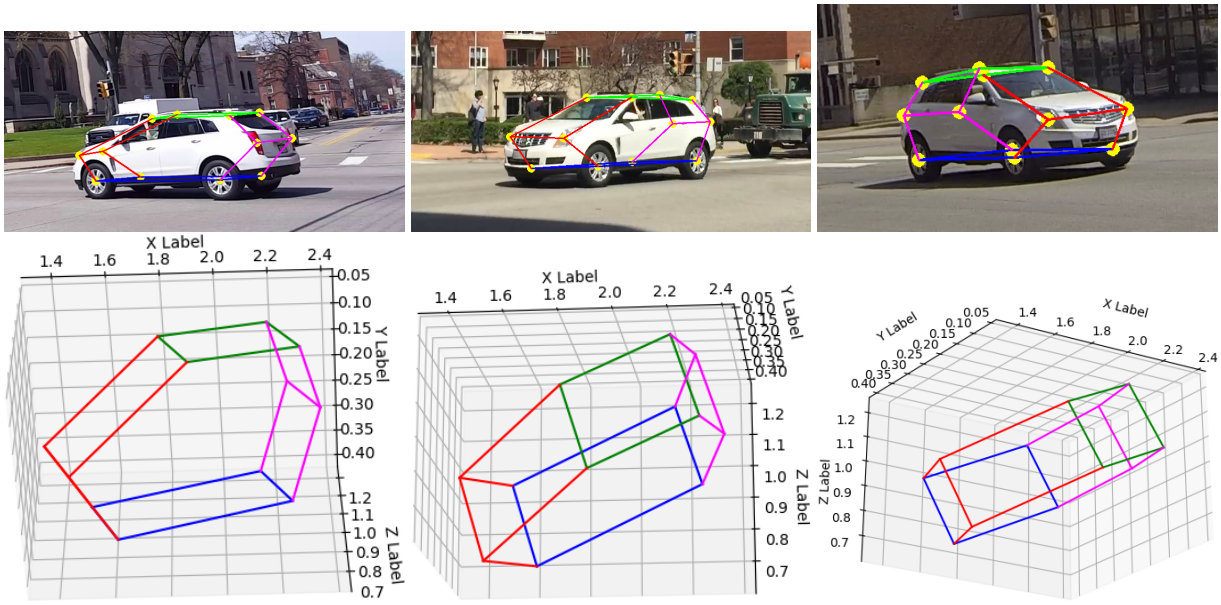

Figure 44: An example detections on the top and the reconstructions from multiple views

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 44 Top) and reconstruct vehicle keypoints and render from multiple views (Figure 44 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.

[2] Code Used For Detection and Reconstruction

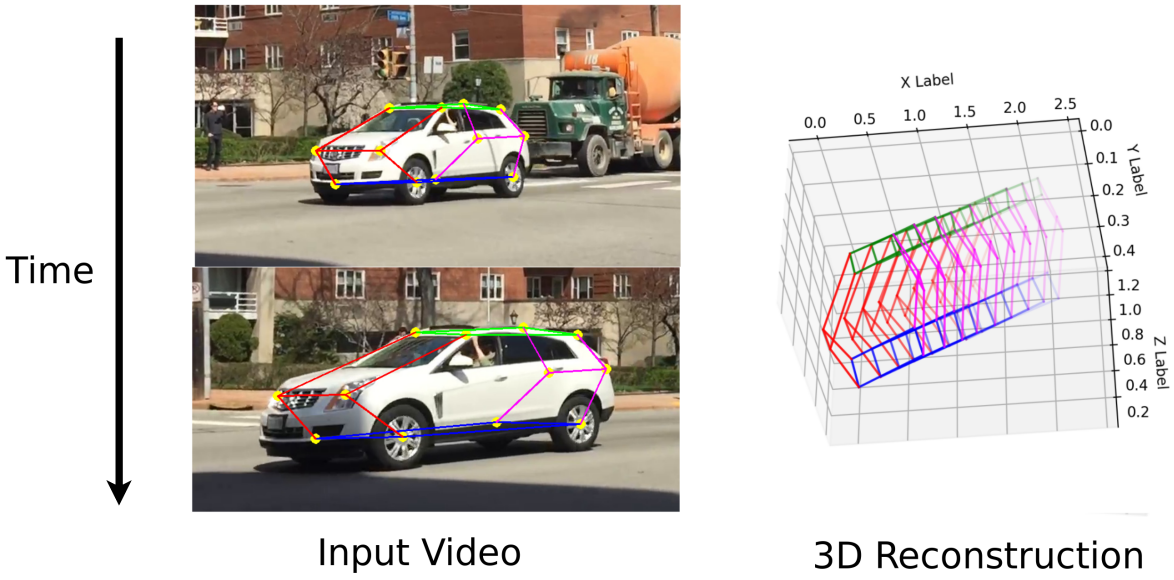

Figure 45: Spatiotemporal reconstruction of the car (right) with the projections at two different time instances in a single view(left)

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 - 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 45.

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>_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 $\mathbf{F}$ 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

[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.