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.

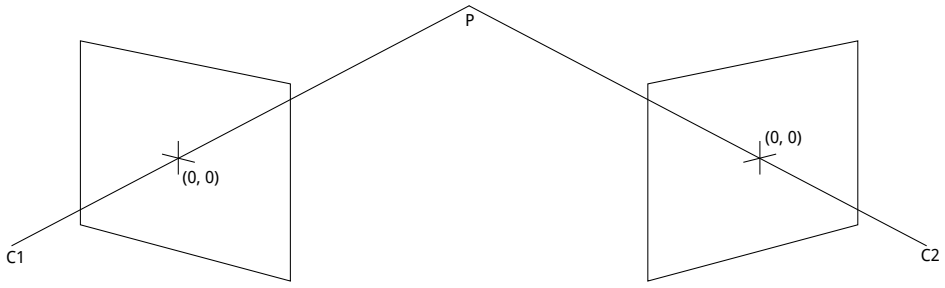


Figure 1: Figure for Q1.1. $C1$ and $C2$ are the optical centers. The principal axes intersect at point $\mathbf{w}$ (P in the figure).

Q1.1

Suppose $\mathbf{x}_1, \mathbf{x}_2$ are the 2D points (origin) on $C1$ and $C2$, respectively, and F is the fundamental matrix between the two cameras. We have the following relationship.

$$\mathbf{x}_2^T F \mathbf{x}_1 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0$$

We have $F_{33} = 0$ from the equation.

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

Let the translation vector between the two cameras be:

$$\mathbf{t} = \begin{bmatrix} t_x \\ 0 \\ 0 \end{bmatrix}, \quad \hat{T} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

Since the rotation matrix is effective an identity matrix, we have the essential matrix:

$$E = \hat{T}R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

Suppose $\mathbf{x}_1 = [x_1, y_1, 1]$ and $\mathbf{x}_2 = [x_2, y_2, 1]$. We have:

$$\mathbf{x}_2 K^{-T} E K^{-1} \mathbf{x}_1 = [x_2, y_2, 1] \begin{bmatrix} 0 \\ a \\ b \end{bmatrix} = a y_2 + b = 0$$

which describes a horizontal line. Similarly, we can derive a equation describing a line:

$$\mathbf{x}_2 K^{-T} E K^{-1} \mathbf{x}_1 = [0, a', b'] \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = a' y_1 + b' = 0$$

Both lines are horizontal and 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 $\mathbf{t}_{rel}$.

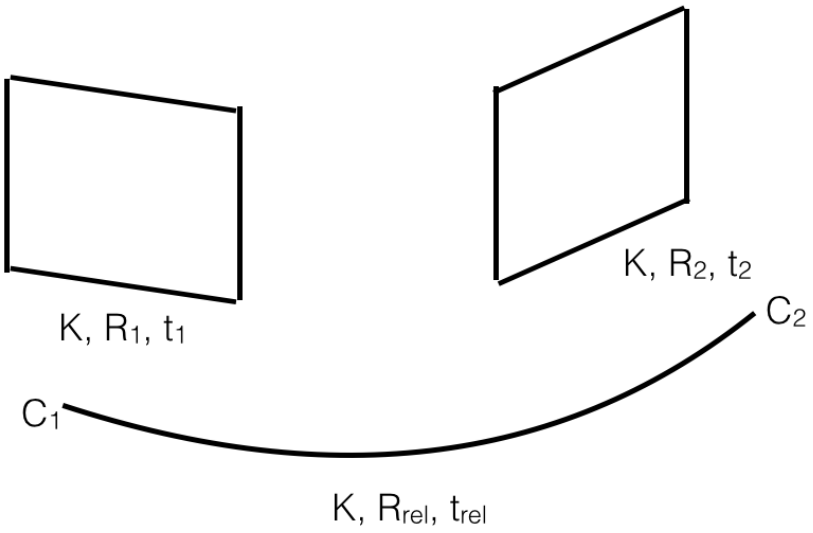


Figure 2: Figure for Q1.3. $C1$ and $C2$ are the optical centers. The rotation and the translation is obtained using inertial sensors. $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$ are the relative rotation and translation between two frames.

Q1.3

Let $\mathbf{x}_1$ and $\mathbf{x}_2$ be the corresponding points between camera 1 and 2, and P be their 3D point.

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \mathbf{x}_1 = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

We have this equation:

$$\mathbf{x}_1 = K\left(R_1 P + t_1\right) \Rightarrow P = R_1^{-1}K^{-1}\mathbf{x}_1 - R_1^{-1}t_1$$

Similarly for camera 2:

$$P = R_2^{-1}K^{-1}\mathbf{x}_2 - R_2^{-1}t_2$$

Combine the two equation for both cameras, we have:

$$\mathbf{x}_2 = KR_2\left(R_1^{-1}K^{-1}\mathbf{x}_1 - R_1^{-1}t_1 + R_2^{-1}t_2\right) = KR_2R_1^{-1}K^{-1}\mathbf{x}_1 - KR_2R_1^{-1}t_1 + Kt_2$$

Finally, we have our final results.

$$\begin{cases} R_{\text{rel}} = KR_2R_1^{-1}K^{-1} \\ t_{\text{rel}} = -KR_2R_1^{-1}t_1 + Kt_2 \\ E = \hat{T}_{\text{rel}}R_{\text{rel}} \\ F = K^{-T}\hat{T}_{\text{rel}}R_{\text{rel}}K^{-1} \end{cases}$$

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.

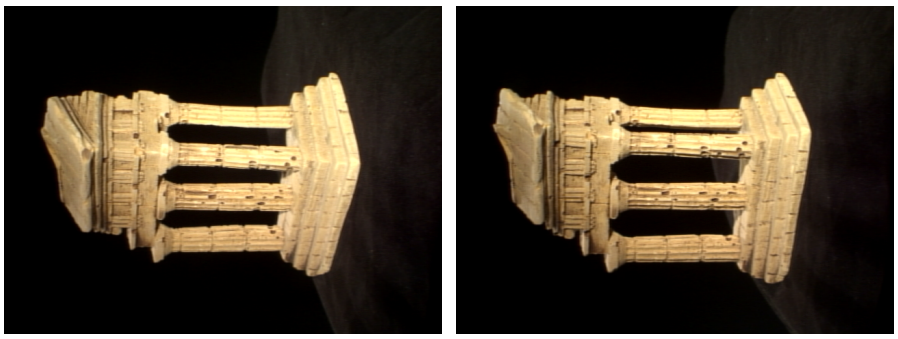


Figure 3: `Temple` images for this assignment

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/

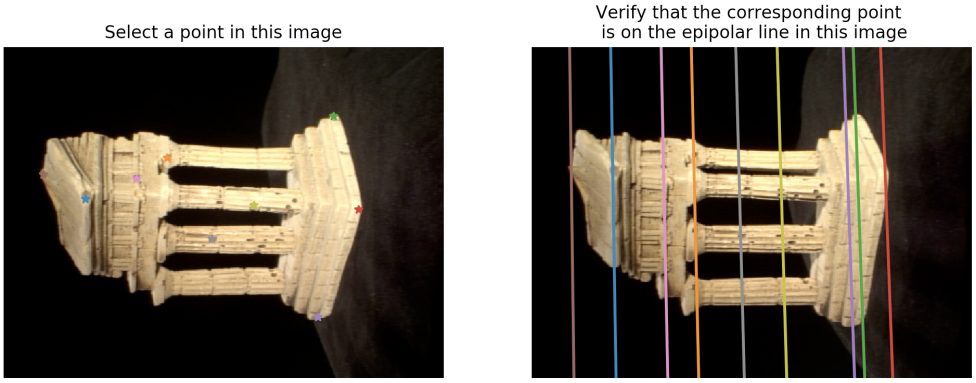


Figure 4: `displayEpipolarF` in `helper.py` creates a GUI for visualizing epipolar lines

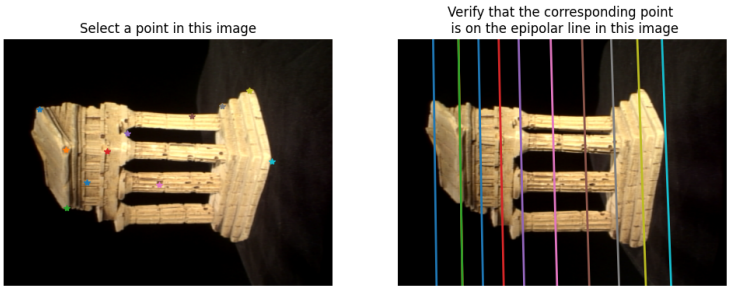
- You should scale the data as was discussed in class, by dividing each coordinate by M (the maximum of the image's width and height). After computing $\mathbf{F}$, you will have to "unscale" the fundamental matrix.
  Hint: If $\mathbf{x}_{normalized} = \mathbf{T}\mathbf{x}$, then $\mathbf{F}_{unnormalized} = \mathbf{T}^T\mathbf{F}\mathbf{T}$.
  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

$$F = \begin{bmatrix} -2.19299583 \times 10^{-7} & 2.95926445 \times 10^{-5} & -2.51886343 \times 10^{-1} \\ 1.28064547 \times 10^{-5} & -6.64493709 \times 10^{-7} & 2.63771739 \times 10^{-3} \\ 2.42229086 \times 10^{-1} & -6.82585550 \times 10^{-3} & 1.00000000 \end{bmatrix}$$




```python
def eightpoint(pts1, pts2, M):
    T = np.array([[1 / M, 0, 0], [0, 1 / M, 0], [0, 0, 1]])
    pts1, pts2 = pts1 / M, pts2 / M # Normalizing the points
    x1, y1 = pts1[:, 0], pts1[:, 1]
    x2, y2 = pts2[:, 0], pts2[:, 1]

    A = np.array([x2 * x1, x2 * y1, x2, y2 * x1, y2 * y1, y2, x1, y1, np.
        ones_like(x1)]).T # (N, 9)

    _, _, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)

    F = _singularize(F) # Enforcing the singularity condition
    F = refineF(F, pts1, pts2)

    F = T.T @ F @ T # Unscaling the fundamental matrix
    return F / F[2, 2]
```

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.

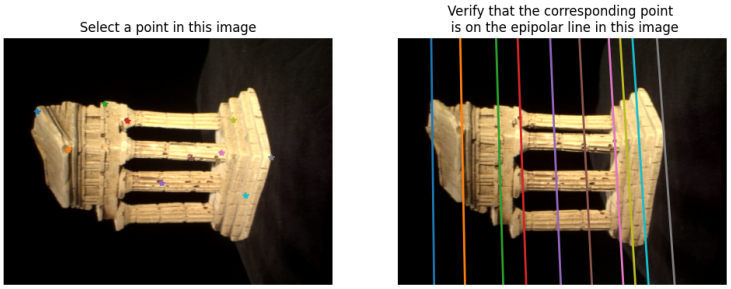
Manually select 7 points from the provided point in `data/some_corresp.npz`, and use these points to recover a fundamental matrix **F**. Use `calc_epi_error` in `helper.py` to calculate the error to pick the best one, and use `displayEpipolarF` to visualize and verify the solution.

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

$$F = \begin{bmatrix} 8.10457567 \times 10^{-7} & 8.90919506 \times 10^{-6} & -2.01028424 \times 10^{-1} \\ 2.63329748 \times 10^{-5} & -6.00542594 \times 10^{-7} & 6.97429503 \times 10^{-4} \\ 1.92182049 \times 10^{-1} & -4.20123580 \times 10^{-3} & 1.00000000 \end{bmatrix}$$



Q2.2

```python
def sevenpoint(pts1, pts2, M):
    T = np.array([[1 / M, 0, 0], [0, 1 / M, 0], [0, 0, 1]])
    pts1, pts2 = pts1 / M, pts2 / M # Normalizing the points
    x1, y1 = pts1[:, 0], pts1[:, 1]
    x2, y2 = pts2[:, 0], pts2[:, 1]

    A = np.array([x2 * x1, x2 * y1, x2, y2 * x1, y2 * y1, y2, x1, y1, np.
        ones_like(x1)]).T # (N, 9)

    _, _, V = np.linalg.svd(A)
    F1 = V[-1].reshape(3, 3)
    F2 = V[-2].reshape(3, 3)

    # Construct the polynomial for solving alpha
    f = lambda alpha: np.linalg.det(alpha * F1 + (1 - alpha) * F2)
    coeff = [0] * 4
    coeff[0] = (f(2) - f(-2) - 2 * (f(1) - f(-1))) / 12
    coeff[1] = (f(1) + f(-1)) / 2 - f(0)
    coeff[2] = (f(1) - f(-1) - 2 * coeff[0]) / 2
    coeff[3] = f(0)

    # Might have 1 or 3 real roots
    roots = np.roots(coeff)
    roots = [root for root in roots if np.isreal(root)]

    # Construct back to F
    Farray = [a * F1 + (1 - a) * F2 for a in roots]
    # Singularize and refine
    Farray = [_singularize(F) for F in Farray]
    Farray = [refineF(F, pts1, pts2) for F in Farray]
    # Unscale, and force F[2, 2] = 1
    Farray = [T.T @ F @ T for F in Farray]
    Farray = [F / F[2, 2] for F in Farray]

    return Farray
```

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

$$\mathbf{E} = \begin{bmatrix} -5.06936459 \times 10^{-1} & 6.86542948 \times 10^{1} & -3.71961460 \times 10^{2} \\ 2.97106977 \times 10^{1} & -1.54718776 & 9.68232563 \\ 3.72991091 \times 10^{2} & 2.98549846 & 1.50354606 \times 10^{-1} \end{bmatrix}$$

```python
def essentialMatrix(F, K1, K2):
    E = K2.T @ F @ K1
    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 $\mathbf{w}_i = \big[x_i, y_i, z_i\big]^T$, 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

$$A_i = \begin{bmatrix}\mathbf{x}1i \times C_1 \\ \mathbf{x}2i \times C_2\end{bmatrix}$$

Note that since the third row of the matrix $\mathbf{x}1i \times C_1$ is linearly dependent on the first and second row, I drop the last row for constructing A. That gives us each camera two equations:

$$\mathbf{x}_{[\times]} \cdot \mathbf{C} = \begin{bmatrix}0 & -1 & y \\ 1 & 0 & -x\end{bmatrix}\begin{bmatrix}c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34}\end{bmatrix}$$

$$= \begin{bmatrix}-yc_{31} + c_{21} & -yc_{32} + c_{22} & -yc_{33} + c_{23} & -yc_{34} + c_{24} \\ xc_{31} - c_{11} & xc_{32} - c_{12} & xc_{33} - c_{13} & xc_{34} - c_{14}\end{bmatrix}$$

Combining both projection for camera 1 and camera 2 gives us a 4x4 matrix A.

Q3.2

```python
def triangulate(C1, pts1, C2, pts2):
    x1, y1 = pts1[:, 0], pts1[:, 1]
    x2, y2 = pts2[:, 0], pts2[:, 1]
    N = x1.shape[0] # Number of points

    # Construct skew-symmetric matrices for all points
    skew_symmetric1 = np.array([
        [np.zeros(N), -1.0 * np.ones(N), y1],
        [np.ones(N), np.zeros(N), -1.0 * x1],
        [-1.0 * y1, x1, np.zeros(N)]
    ]).transpose(2, 0, 1) # Shape (N, 3, 3)

    skew_symmetric2 = np.array([
        [np.zeros(N), -1.0 * np.ones(N), y2],
        [np.ones(N), np.zeros(N), -1.0 * x2],
        [-1.0 * y2, x2, np.zeros(N)]
    ]).transpose(2, 0, 1) # Shape (N, 3, 3)

    # Only consider the first two rows because the third row is linearly
        dependent
    eq1 = (skew_symmetric1 @ C1)[:, :2, :]  # Shape (N, 2, 4)
    eq2 = (skew_symmetric2 @ C2)[:, :2, :]  # Shape (N, 2, 4)

    A = np.concatenate([eq1, eq2], axis=1)  # Shape (N, 4, 4)

    # Initialize the 3D points matrix (in homogeneous coordinates)
    P = np.zeros((N, 4))
    for i in range(N):
        _, _, V = np.linalg.svd(A[i]) # V has shape (4, 4)
        omega = V[-1]
        P[i, :] = omega / omega[-1] # Normalize the homogeneous coordinates

    # Project the 3D points back to 2D
    proj1 = C1 @ P.T  # Shape (3, N)
    proj2 = C2 @ P.T  # Shape (3, N)

    # Normalize, back to non-homogeneous coordinates
    proj1 = proj1[:2, :] / proj1[2, :]  # Shape (2, N)
    proj2 = proj2[:2, :] / proj2[2, :]  # Shape (2, N)

    # Compute the reprojection error
    err = np.sum(np.linalg.norm(proj1.T - pts1, axis=1) + np.linalg.norm(proj2
        .T - pts2, axis=1))

    return P[:, :3], 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

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

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

    # C1 = K1 @ M1 = K1 @ [I|0]
    M1 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
    C1 = K1 @ M1

    best_M2, best_C2, best_P = None, None, None
    most_pos = 0
    for i in range(4):
        # C2 = K2 @ M2 = K2 @ [R|t]
        M2 = M2s[:,:,i]
        C2 = K2 @ M2
        P, _ = triangulate(C1, pts1, C2, pts2)

        # Check if the 3D points are in front of the camera
        if np.sum(P[:, -1] > 0) > most_pos:
            best_M2, best_C2, best_P = M2, C2, P
            most_pos = np.sum(P[:, -1] > 0)

    if filename is not None:
        np.savez(filename, M2=best_M2, C2=best_C2, P=best_P)

    return best_M2, best_C2, best_P
```

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.

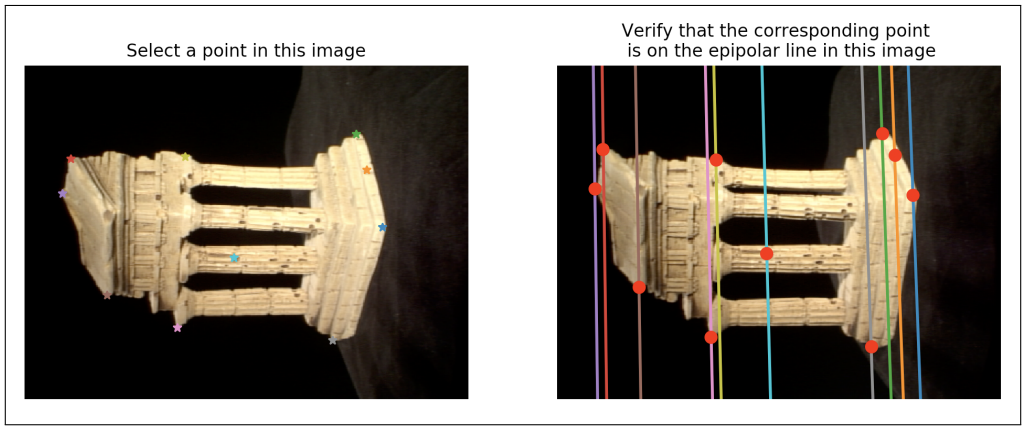


Figure 5: `epipolarMatchGUI` shows the corresponding point found by calling `epipolarCorrespondence`

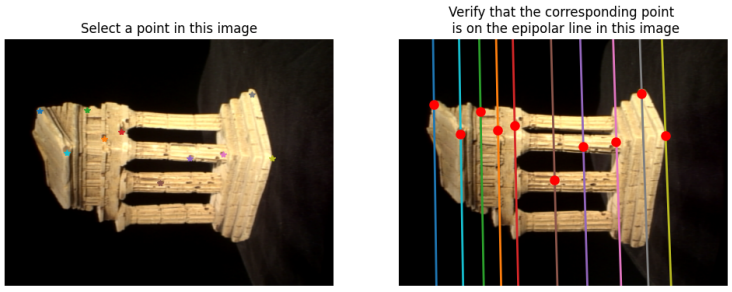
It's not necessary for your matcher to get *every* possible point right, but it should get easy points (such as those with distinctive, corner-like windows). It should also be good enough to render an intelligible representation in the next question.

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



Q4.1

```python
def epipolarCorrespondence(im1, im2, F, x1: int, y1: int):
    window_size = 20
    # Assume two images only differ by a small amount, i.e. d(p1, p2) is small
    search_range = 50
    half_window = window_size // 2

    # Template patch around the point (x1, y1) in im1
    window1 = im1[y1 - half_window:y1 + half_window + 1, x1 - half_window:x1 +
        half_window + 1]

    # Gaussian filter helps us focus more on the center point
    window1 = gaussian_filter(window1, sigma=1)

    # Compute the epipolar line in im2 using fundamental matrix
    epipolar_line = F @ np.array([x1, y1, 1]).T
    epipolar_line = epipolar_line / np.linalg.norm(epipolar_line[:2])

    height, width = im2.shape[:2]
    min_y, max_y = max(0, y1 - search_range), min(height, y1 + search_range)

    best_match = None
    min_error = float('inf')
    # Iterate along the epipolar line within the range to find the best match
    for y2 in range(min_y, max_y):
        # x * l[0] + y * l[1] + l[2] = 0 ==> x = -(l[1] * y + l[2]) / l[0]
        x2 = int(-(epipolar_line[1] * y2 + epipolar_line[2]) / epipolar_line
            [0])
        if x2 - half_window < 0 or x2 + half_window >= width or y2 -
            half_window < 0 or y2 + half_window >= height:
            continue

        window2 = im2[y2 - half_window:y2 + half_window + 1, x2 - half_window:
            x2 + half_window + 1]
        # Same filter for the im2 window
        window2 = gaussian_filter(window2, sigma=1)

        error = np.sum((window1 - window2) ** 2)
        if error < min_error:
            min_error = error
            best_match = (x2, y2)

    return best_match
```

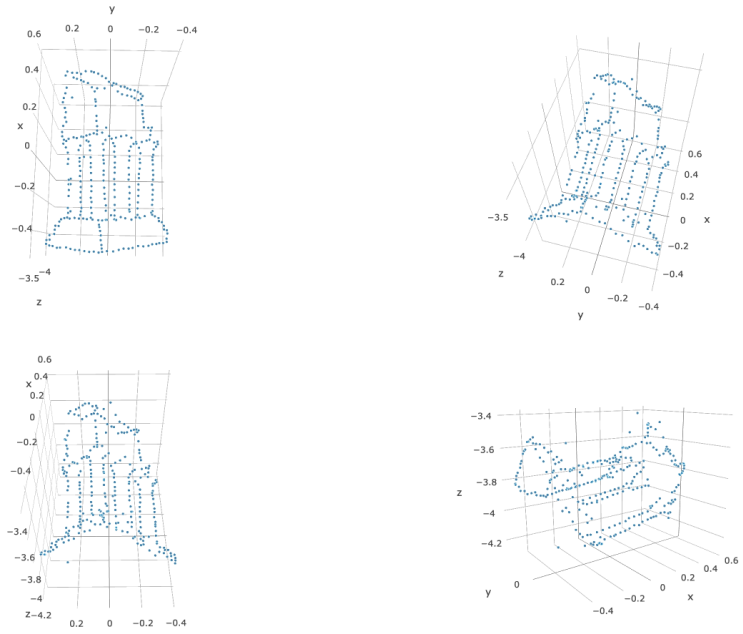


Figure 6: An example point cloud

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`.

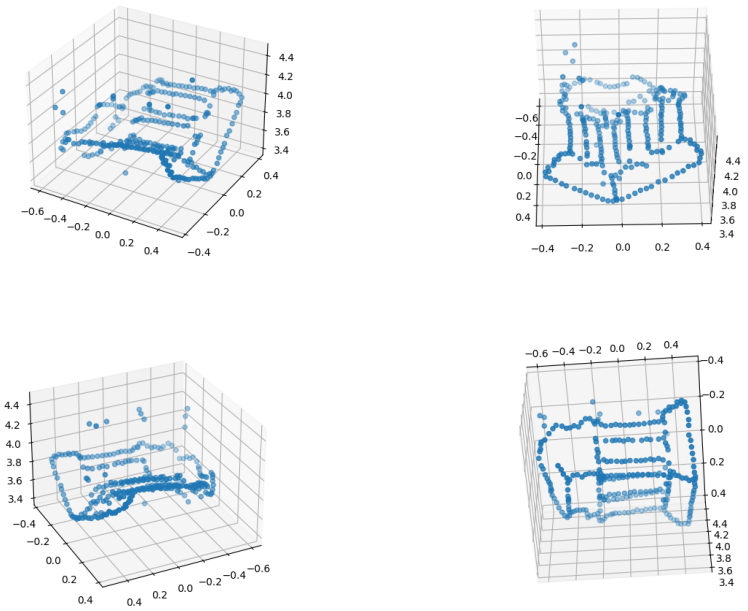
Now, we can determine the 3D location of these point correspondences using the `triangulate` function. These 3D point locations can then be plotted using the Matplotlib or plotly package. Complete the `compute3D_pts` function in `q4_2_visualize.py`, which loads the necessary files from `../data/` to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



```python
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    pts2 = np.array([epipolarCorrespondence(im1, im2, F, x1, y1) for x1, y1 in
        temple_pts1])
    M2, C2, P = findM2(F, temple_pts1, pts2, intrinsics, filename=None)
    return 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

I used `np.sum(calc_epi_error())` to calculate my sum of squared distance re-projection error. I count a point as an inlier if the SSD error is less than the tolerance. Note that I calculated the reprojection error only using the inliers returned by `ransacF`. The comparison:

Error for RANSAC: 36.74359044188147
Error for 8 point: 67875.97238975769

Clearly RANSAC helps eliminating the outliers and gives a better F.

The effect on different parameters for running RANSAC:
`nIters`: If it is larger, the RANSAC runs more iterations. The algorithm would have higher chance to find a "good" fundamental matrix, but the execution time would also be longer.
`tolerance`: If it is larger, the algorithm would more tolerable on identifying inliers, and vice versa. It should be tuned to get a better result. I used `tolerance`=3 in my experiments.

```python
def ransacF(pts1, pts2, M, nIters=1000, tol=10):
    MODE = "EIGHT" # Change to "SEVEN" for seven point algorithm
    N = pts1.shape[0] # Number of points

    pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)
    bestF = np.zeros((3, 3))
    best_inliers = np.zeros(N, dtype=bool)

    for _ in tqdm(range(nIters)):
        if MODE == "EIGHT":
            idx = np.random.choice(N, 8) # Should be replace=False, but
                somehow it doesn't work
            F = eightpoint(pts1[idx], pts2[idx], M)
        else: # use seven point algorithm
            idx = np.random.choice(N, 7)
            F = sevenpoint(pts1[idx], pts2[idx], M)

        inliers = calc_epi_error(pts1_homogenous, pts2_homogenous, F) < tol

        if np.sum(inliers) > np.sum(best_inliers):
            best_inliers = inliers
            bestF = F

    bestF = eightpoint(pts1[best_inliers], pts2[best_inliers], M)
    return bestF, best_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

```python
def invRodrigues(R):
    # Make sure R is a valid rotation matrix
    assert np.allclose(R.T @ R, np.identity(3), atol=1e-6), "R is not a valid rotation matrix, R^T * R != I"
    assert np.isclose(np.linalg.det(R), 1.0, atol=1e-6), "R is not a valid rotation matrix, det(R) != 1"

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

    if np.isclose(s, 0) and np.isclose(c, 1):
        return np.zeros(3)
    if np.isclose(s, 0) and np.isclose(c, -1):
        # Let v = a nonzero column of R + I
        M = (R + np.eye(3)) / 2
        for i in range(3):
            if np.linalg.norm(M[:, i]) > 1e-10:
                v = M[:, i]
                break

        u = v / np.linalg.norm(v)
        r = np.pi * u
        # Construct S_{1/2}(u * pi)
        if np.isclose(np.linalg.norm(r), np.pi) and ((r[0] == r[1] == 0 and r[2] < 0)
            or (r[0] == 0 and r[1] < 0) or r[0] < 0):
            r = -r

    u = rho / s
    theta = np.arctan2(s, c)
    r = theta * u
    return r
```

Q5.2

```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    N = p1.shape[0]
    P = x[:3 * N].reshape(N, 3)  # 3D points, shape (N, 3)
    r2 = x[3 * N:3 * N + 3]      # Rotation vector, shape (3,)
    t2 = x[3 * N + 3:]           # Translation vector, shape (3,)

    R2 = rodrigues(r2)
    # M2 = [R2 | t2]
    M2 = np.hstack((R2, t2.reshape(-1, 1)))

    C1 = K1 @ M1 # Camera 1 projection matrix
    C2 = K2 @ M2 # Camera 2 projection matrix

    P_homo = np.hstack((P, np.ones((N, 1)))) # Convert 3D points to
        homogeneous coordinates, shape (N, 4)
    p1_hat_homo = (C1 @ P_homo.T).T # 2D point projected to image 1, shape (N,
         3)
    p2_hat_homo = (C2 @ P_homo.T).T # 2D point projected to image 2, shape (N,
         3)

    p1_hat = p1_hat_homo[:, :2] / p1_hat_homo[:, 2:3] # shape (N, 2)
    p2_hat = p2_hat_homo[:, :2] / p2_hat_homo[:, 2:3] # shape (N, 2)

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

Q5.3 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

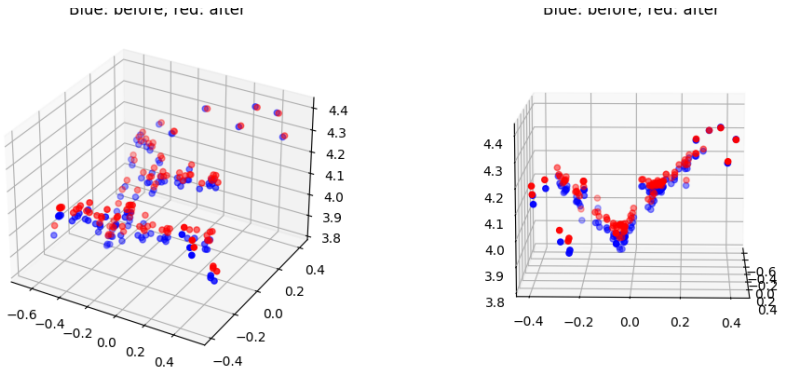


Figure 7: Visualization of original 3D points and the optimized points

Reprojection error before optimization: 6999.104550
Reprojection error after optimization: 9.102508

Q5.3

```python
def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    N = p1.shape[0] # Number of points

    # Extract the rotation and translation from the initial extrinsics of
        camera 2
    R2_init = M2_init[:, :3]
    t2_init = M2_init[:, 3]
    # Convert initial rotation matrix to Rodrigues vector
    r2_init = invRodrigues(R2_init)

    # Concatenate 3D points, rotation vector, and translation for
        rodriguesResidual
    x0 = np.concatenate([P_init.flatten(), r2_init, t2_init]) # (3N + 3 + 3,)

    # Use sum of squared residuals as the objective function
    def objective(x):
        residuals = rodriguesResidual(K1, M1, p1, K2, p2, x)
        return np.sum(residuals**2)

    # Run optimization using Scipy's minimize, use default BFGS algorithm
    result = minimize(objective, x0, method='BFGS', options={'maxiter': 1000,
        'disp': False})

    # Extract optimized parameters
    x_opt = result.x
    obj_start = objective(x0) # Initial objective function value
    obj_end = result.fun      # Optimized objective function value

    # Extract optimized 3D points, rotation vector, and translation
    P_opt = x_opt[:3 * N].reshape(N, 3)
    r2_opt = x_opt[3 * N:3 * N + 3]
    t2_opt = x_opt[3 * N + 3:]

    R2_opt = rodrigues(r2_opt) # Back to rotation matrix
    M2 = np.hstack((R2_opt, t2_opt.reshape(-1, 1)))

    return M2, P_opt, obj_start, obj_end
```

Q5.3

```python
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    N = p1.shape[0]
    P = x[:3 * N].reshape(N, 3)  # 3D points, shape (N, 3)
    r2 = x[3 * N:3 * N + 3]      # Rotation vector, shape (3,)
    t2 = x[3 * N + 3:]           # Translation vector, shape (3,)

    R2 = rodrigues(r2)
    # M2 = [R2 | t2]
    M2 = np.hstack((R2, t2.reshape(-1, 1)))

    C1 = K1 @ M1 # Camera 1 projection matrix
    C2 = K2 @ M2 # Camera 2 projection matrix

    P_homo = np.hstack((P, np.ones((N, 1)))) # Convert 3D points to
        homogeneous coordinates, shape (N, 4)
    p1_hat_homo = (C1 @ P_homo.T).T # 2D point projected to image 1, shape (N,
         3)
    p2_hat_homo = (C2 @ P_homo.T).T # 2D point projected to image 2, shape (N,
         3)

    p1_hat = p1_hat_homo[:, :2] / p1_hat_homo[:, 2:3] # shape (N, 2)
    p2_hat = p2_hat_homo[:, :2] / p2_hat_homo[:, 2:3] # shape (N, 2)

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

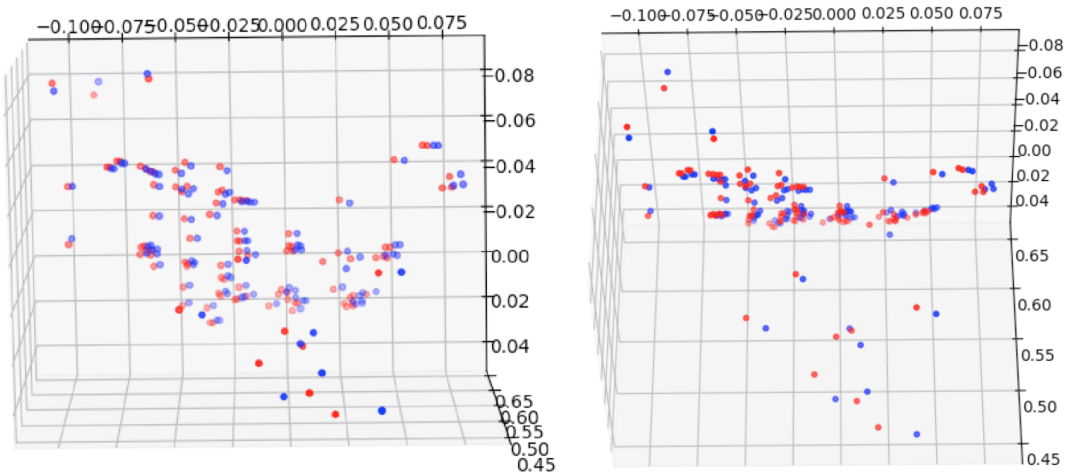


Figure 8: Visualization of 3D points for noisy correspondences before and after using bundle adjustment

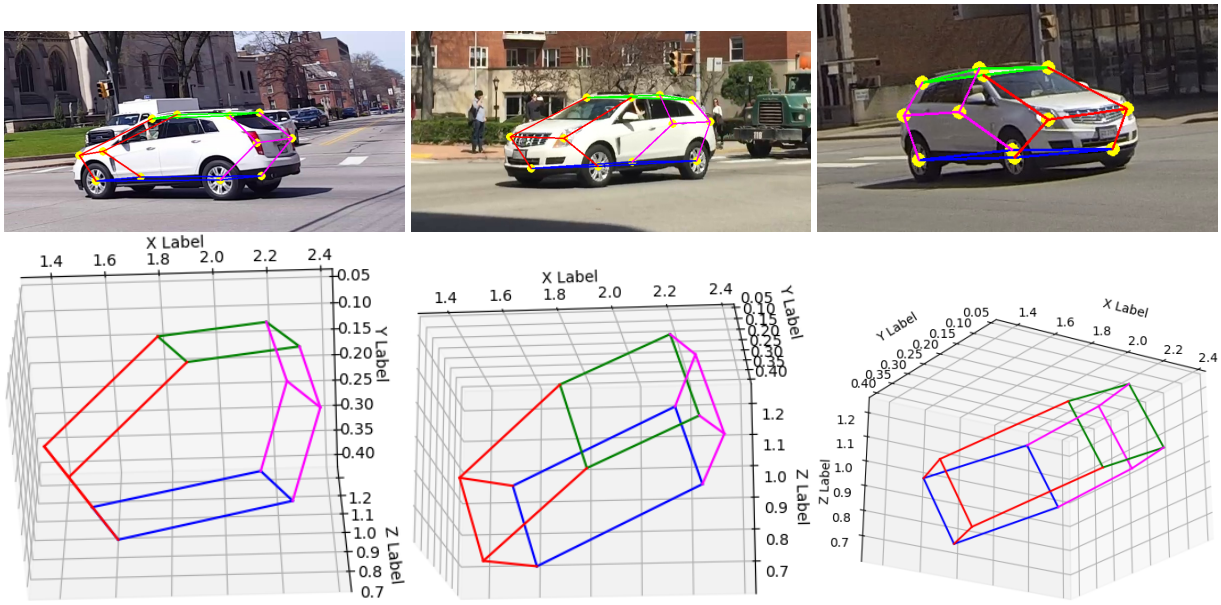


Figure 9: An example detections on the top and the reconstructions from multiple views

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 9 Top) and reconstruct vehicle keypoints and render from multiple views (Figure 9 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

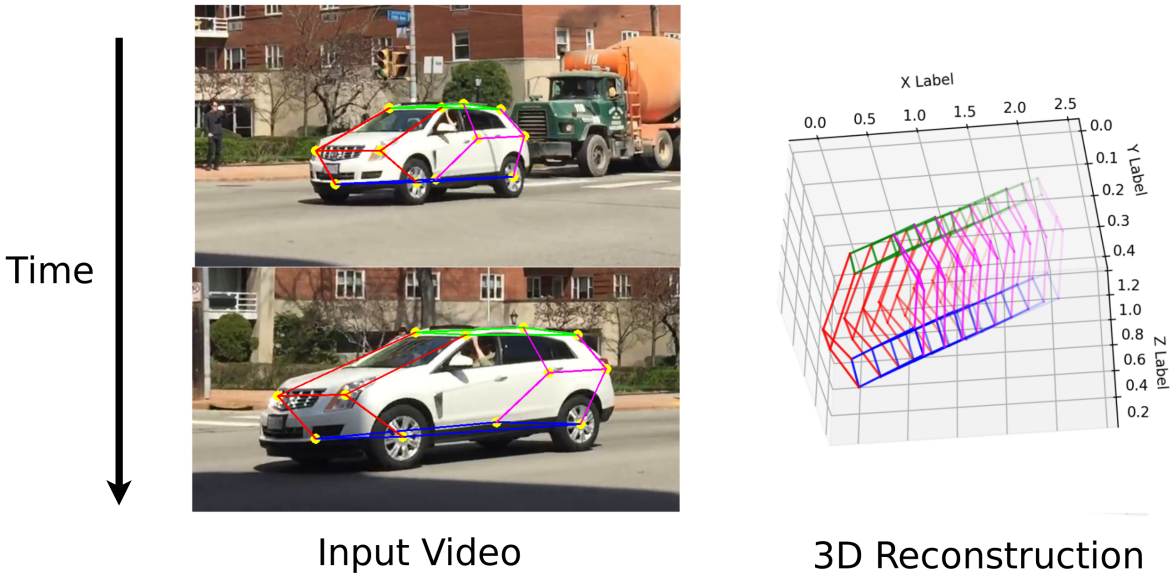


Figure 10: Spatiotemporal reconstruction of the car (right) with the projections at two different time instances in a single view(left)

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 10.

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 **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.