HOMEWORK 3: 3D RECONSTRUCTION

Lifan Yu (lifany)

16-820 Advanced Computer Vision (Fall 2023)
https://16820advancedcv.github.io/
OUT: Ocotber 4th, 2023
DUE: October 25th, 2023
Instructor: David Held
TAs: Vineet Tambe, Bart Duisterhof, Ronit Hire, Dishani Lahiri

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.

amsmath

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.

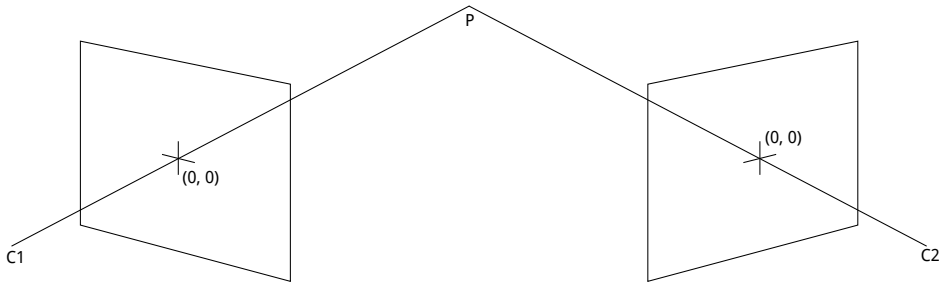


Figure 1: Figure for Q1.1. $C1$ and $C2$ are the optical centers. The principal axes intersect at point $\mathbf{w}$ (P in the figure).

Q1.1

A 3D point projected into a camera coordinate system can be represented as $\lambda x = KX$ as follows

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_{sx} & f_{s\theta} & O_x \\ 0 & f_{sy} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

With normalisation: $\lambda K^{-1}x = X$, normalised $x' = K^{-1}x$, we have $\lambda x' = X$
Let X_1 denote a 3D point in camera 1's coordinates, and X_2 denote the same point in camera 2's coordinates. We have $X_2 = RX_1 + T$.
Plugging in the normalised coordinates: $\lambda_1 x'_1 = X_1, \lambda_2 x'_2 = X_2$
We have:

Q1.1

$$\lambda_2 x_2' = R(\lambda_1 x_1') + T$$

Take cross product

$$\lambda_2 \hat{T} x_2' = \lambda_1 \hat{T} R x_1' + \hat{T} T$$

Take dot product with x_2

$$\lambda_2 x_2'^T \hat{T} x_2' = \lambda_1 x_2'^T \hat{T} R x_1'$$

Because the product is a vector's dot product with a cross product, it represents the volume of the parallel piped formed by the 3 vectors. On the left side of the equation, because 2 of the vectors are the same, the volume equals zero. We have

$$x_2'^T \hat{T} R x_1' = 0$$

Here with normalised coordinates, $E = F = \hat{T}R$.

$$x_2'^T F x_1' = 0$$

The principal point has coordinates $x_2' = x_1' = [0, 0, 1]$ in both camera 1 and camera 2.

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

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

$$F_{33} = 0$$

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

Q1.2

From previous derivations we have the following between normalised coordinates

$$x_2' \hat{T} R x_1' = 0$$

Plug in $x_1' = K_1^{-1}\mathbf{x_1}$, $x_2' = K_2^{-1}\mathbf{x_2}$, with $\mathbf{x_1}, \mathbf{x_2}$ unormalised, we have

$$\mathbf{x_2}^T K_2^{-T} \hat{T} R^T K_1^{-1} \mathbf{x_1} = 0$$

The fundamental matrix $F = K_2^{-T} \hat{T} R^T K_1^{-1}$, we have

$$\mathbf{x_2} F \mathbf{x_1} = 0$$

With **pure translation**, we have
(1) No rotation: $R = I$
(2) Translating along x axis, let $T = \begin{bmatrix} t \\ 0 \\ 0 \end{bmatrix}$ with $t \neq 0$. The cross product matrix $\hat{T} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t \\ 0 & t & 0 \end{bmatrix}$

Additionally, we assume cameras 1 and 2 have the same intrinsics, so

$$K_1 = K_2 = K$$

The fundamental matrix F then becomes

$$F = K^{-T} \hat{T} K^{-1}$$

Because $K = \begin{bmatrix} f_{sx} & f_{s\theta} & O_x \\ 0 & f_{sy} & O_y \\ 0 & 0 & 1 \end{bmatrix}$ is an upper triangular matrix, its inverse is also upper triangular, with the (3, 3) element = 1. We can write $K^{-1} = \begin{bmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & 1 \end{bmatrix}$, with $a, b, c, d, e \in R$

Substituting into $F = K^{-T} \hat{T} K^{-1}$, we have the expression below: (next page)

Q1.2

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

Let $t' = dt$. Because $f_{sy} \neq 0$, we have $d \neq 0$. The translation amount $t \neq 0$
Thus

$$\mathbf{t'} \neq \mathbf{0}$$

Substitute $F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t' \\ 0 & t' & 0 \end{bmatrix}$ into $\mathbf{x_2} F \mathbf{x_1} = 0$ gives

$$\begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t' \\ 0 & t' & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = 0$$

$$\begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -t' \\ t'y_1 \end{bmatrix} = 0$$

$$-t'y_2 + t'y_1 = 0$$

$$t'y_1 = t'y_2$$

Because $t' \neq 0$, we must have $\boxed{y_2 = y_1}$.

All the points that gets mapped from (x_1, y_1) in camera 1 to (x_2, y_2) in camera 2 lie on the the $\boxed{\text{epipolar line } y = y_2}$. Vice versa for camera 1, it has the $\boxed{\text{epipolar line } y = y_1}$.

Both epipolar lines are parallel to the x axis.

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

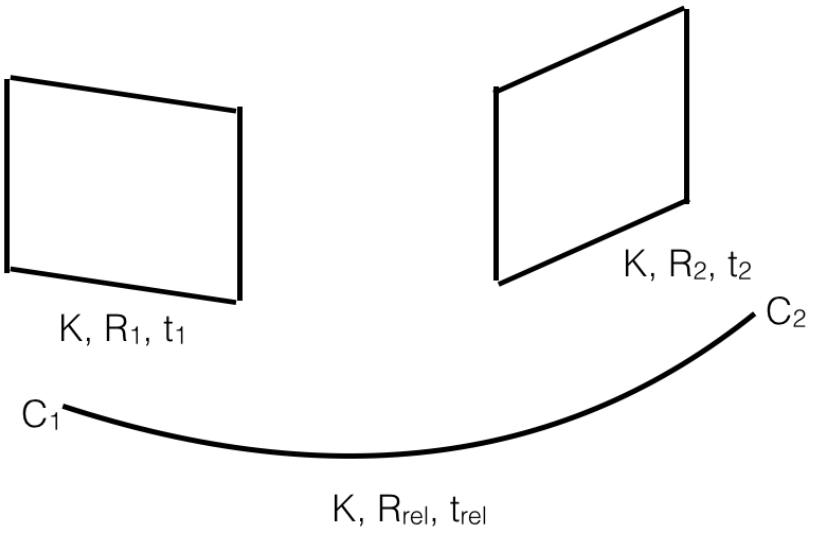


Figure 2: Figure for Q1.3. $C1$ and $C2$ are the optical centers. The rotation and the translation is obtained using inertial sensors. $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$ are the relative rotation and translation between two frames.

Q1.3

The R_i, t_i measured are relevant to the initial X_0.
At timestep $i = 1, X_1 = R_1 X_0 + t_1$
At timestep $i = 2, X_2 = R_2 X_0 + t_2$
...
At timestep $i, X_i = R_i X_0 + t_i$
At timestep $j, X_j = R_j X_0 + t_j$

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

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

The relative rotation and translation are

$$\boxed{R_{rel} = R_j R_i^{-1}}$$
$$\boxed{t_{rel} = t_j - R_j R_i^{-1} t_i}$$

Q1.3

A 3D point projection into camera coordinates are expressed as $\lambda_i x_i = KX_i, \lambda_j X_j = KX_j$
We have:
(1) Unormalised: $\lambda_i K^{-1}\mathbf{x_j} = X_i$ and $\lambda_j K^{-1}\mathbf{x_i} = X_j$
(2) Normalised: $\lambda_i x_i' = X_i$ and $\lambda_j K^{-1} x_j' = X_j$, with $x_i' = K^{-1}\mathbf{x_i}, x_j' = K^{-1}\mathbf{x_j}$
Substituting into the above expression gives:

$$\lambda_j x_j' = R_{rel}(\lambda_i x_i') + t_{rel}$$

We get

$$x_j'^T \hat{t}_{rel} R_{rel} x_i' = 0$$

Essential matrix $\boxed{E = \hat{t}_{rel} R_{rel}}$

$$\mathbf{x_j}^T K^{-T} \hat{t}_{rel} R_{rel} K^{-1} \mathbf{x_i} = 0$$

Fundamental matrix $\boxed{F = K^{-T} \hat{t}_{rel} R_{rel} K^{-1}}$

Where $t_{rel} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}, \hat{t}_{rel} = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}$

Part II

Practice

1 Overview

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

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.

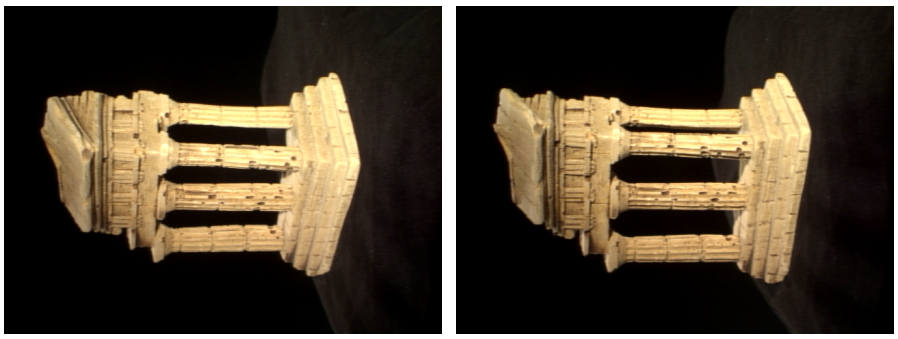


Figure 3: `Temple` images for this assignment

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/

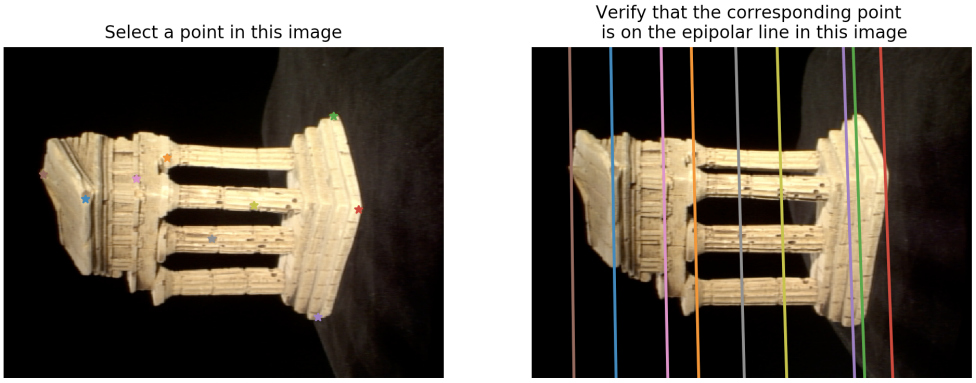


Figure 4: `displayEpipolarF` in `helper.py` creates a GUI for visualizing epipolar lines

- 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:

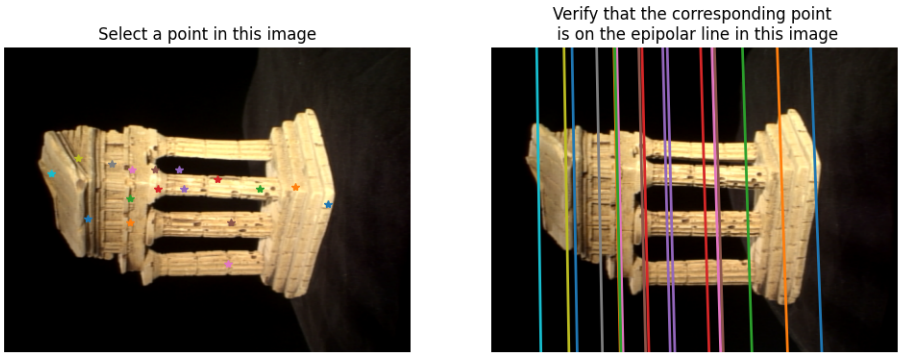
- Write your recovered $\mathbf{F}$
- Include an image of some example output of `displayEpipolarF`
- Include the code snippet of `eightpoint` function

Q2.1

Recovered F: (in 8 iterations)

```
[[-2.19299582e-07  2.95926445e-05 -2.51886343e-01]
 [ 1.28064547e-05 -6.64493709e-07  2.63771740e-03]
 [ 2.42229086e-01 -6.82585550e-03  1.00000000e+00]]
```

Output image:



Q2.1

Code of eightpoint():

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

    # (1) Normalize the input pts1 and pts2 using the matrix T.
    N = pts1.shape[0]
    pts11 = np.append(pts1, np.ones((N, 1)), axis=1)
    pts21 = np.append(pts2, np.ones((N, 1)), axis=1)
    T = np.array([[1/M, 0, 0],
                  [0, 1/M, 0],
                  [0, 0, 1]])
    npts1, npts2 = np.matmul(pts11, T), np.matmul(pts21, T)

    # (2) Setup the eight point algorithm's equation, which is AF(:) = 0
    A = np.zeros(shape=(N, 9))
    for i in range(N):
      row = np.matmul(npts2[i].reshape((3, 1)), npts1[i].reshape((1, 3))).
                                                  reshape((9, ))
      A[i] = row

    # (3) Solve for the least square solution using SVD.
    U, Sigma, VT = np.linalg.svd(A)
    # Take last row of VT (last column of VT.T) in special case SVD Ax=0
    # Math reference: www.cse.unr.edu/~bebis/CS791E/Notes/SVD.pdf
    F = VT[-1, :].reshape((3, 3))

    # (4) Use the function `_singularize` (provided) to enforce the
                                            singularity condition.
    F = _singularize(F)

    # (5) Use the function `refineF` (provided) to refine the computed
                                            fundamental matrix.
    F = refineF(F, npts1[:, :-1], npts2[:, :-1])

    # (6) Unscale the fundamental matrix
    F = np.matmul(T, np.matmul(F, T))
    F = F/F[-1][-1]
    print("F", F)

    return F
```

2.2 The Seven Point Algorithm (Extra Credit)

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

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

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

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

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

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

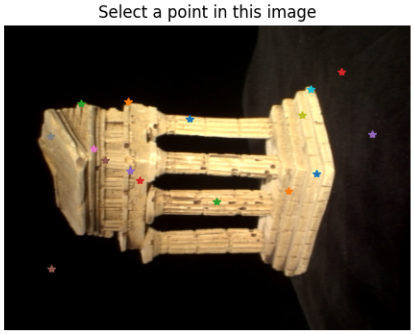
- Write your recovered **F**
- Include an image of some example output of `displayEpipolarF`
- Include the code snippet of `sevenpoint` function

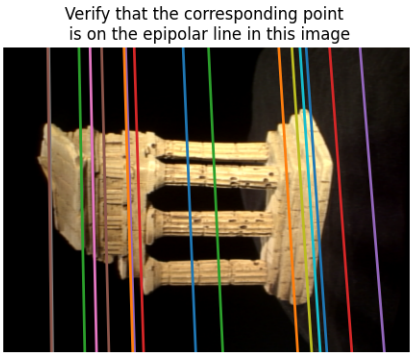
Q2.2

Recovered F: (final F, after 500 iterations)

```
[[ 8.10457567e-07  8.90919506e-06 -2.01028424e-01]
 [ 2.63329748e-05 -6.00542594e-07  6.97429503e-04]
 [ 1.92182049e-01 -4.20123580e-03  1.00000000e+00]]
```

Final output image after 500 iterations:





Q2.2

```
def det_func(a, f1, f2):
    return np.linalg.det(a*f1 + (1-a)*f2)

def sevenpoint(pts1, pts2, M):
    Farray = []
    # ----- TODO -----
    # YOUR CODE HERE
    # (1) Normalize the input pts1 and pts2 scale paramter M.
    N = pts1.shape[0]
    pts11 = np.append(pts1, np.ones((N, 1)), axis=1)
    pts21 = np.append(pts2, np.ones((N, 1)), axis=1)
    T = np.array([[1/M, 0, 0],
                  [0, 1/M, 0],
                  [0, 0, 1]])
    npts1, npts2 = np.matmul(pts11, T), np.matmul(pts21, T)

    # (2) Setup the seven point algorithm's equation.
    A = np.zeros(shape=(N, 9))
    for i in range(N):
      row = np.matmul(npts2[i].reshape((3, 1)), npts1[i].reshape((1, 3))).
                                                  reshape((9, ))
      A[i] = row

    # (3) Solve for the least square solution using SVD.
    U, Sigma, VT = np.linalg.svd(A)

    # (4) Pick the last two colum vector of vT.T (the two null space solution
                                                  f1 and f2)
    f1 = VT[-2, :].reshape((3, 3))
    f2 = VT[-1, :].reshape((3, 3))

    # solve [c3, c2, c1, c0]] using:
    # [a_0**3, a_0**2, a_0, 1]  [c3   [ f(a0)
    # [a_1**3, a_1**2, a_1, 1] * c2 =  f(a1)
    # [a_2**3, a_2**2, a_2, 1]   c1    f(a2)
    # [a_3**3, a_3**2, a_3, 1]   c0]   f(a3) ]
    avals = [0, 1/4, 1/2, 3/4]
    amat = np.zeros((4, 4))
    for i in range(4):
        amat[i] = np.array([avals[i]**3, avals[i]**2, avals[i], 1])
    bmat = np.array([det_func(avals[i], f1, f2) for i in range(4)]).reshape((4
                                                  , 1))
    res = np.ravel(np.linalg.solve(amat, bmat))
    c3, c2, c1, c0 = res[0], res[1], res[2], res[3]
    roots = np.polynomial.polynomial.polyroots([c0, c1, c2, c3])
    idx, = np.where(np.iscomplex(roots) == False)
    roots = roots[list(idx)]

    # (6) Unscale the fundamental matrixes and return as Farray
    Farray = []
    for r in roots:
        F = r*f1 + (1-r)*f2
        F = _singularize(F)
        F = np.matmul(T, np.matmul(F, T))
        F = F/F[-1][-1]
        Farray.append(F)
    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

Estimated E:

```
[[-3.37160578e+00  4.56615841e+02 -2.47389468e+03]
 [ 1.97604174e+02 -1.02902585e+01  6.43966014e+01]
 [ 2.48074270e+03  1.98563818e+01  1.00000000e+00]]
```

Code of essentialMatrix():

```
def essentialMatrix(F, K1, K2):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE
    E = np.matmul(K2.T, np.matmul(F, K1))
    E = E/E[-1][-1]
    print("E", E)

    return E
```

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

Note: The matrices M1 and M2 here are of the form: $\mathbf{M}_1 = \begin{bmatrix}\mathbf{I}|0\end{bmatrix}$ 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

For each point $\mathbf{x_i}$ in the camera image, we have

$$\mathbf{x_i} = PX_i$$

With P as the projection matrix and X as the corresponding 3D point. Expanding with specific values:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \alpha \begin{bmatrix} p1 & p2 & p3 & p4 \\ p5 & p6 & p7 & p8 \\ p9 & p10 & p11 & p12 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

We can denote matrix $\mathbf{P}$ using its row vectors as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \alpha \begin{bmatrix} \mathbf{p1^T} \\ \mathbf{p2^T} \\ \mathbf{p3^T} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Taking cross product of the left and right gives:

$$\mathbf{x} \times \mathbf{PX} = 0$$

Removing the scalar to let $z = 1$, the cross product is then:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \times \begin{bmatrix} \mathbf{p1^T} \\ \mathbf{p2^T} \\ \mathbf{p3^T} \end{bmatrix} \mathbf{X} = 0$$

$$\begin{bmatrix} y\mathbf{p3^T} - \mathbf{p2^T} \\ \mathbf{p1^T} - x\mathbf{p3^T} \end{bmatrix} \mathbf{X} = 0$$

To solve for $\mathbf{X}$ we need (at least) 2 corresponding points $\mathbf{x} = [x, y]^T, \mathbf{x'} = [x', y']^T$:

$$\begin{bmatrix} y\mathbf{p3^T} - \mathbf{p2^T} \\ \mathbf{p1^T} - x\mathbf{p3^T} \\ y'\mathbf{p3^T} - \mathbf{p2^T} \\ \mathbf{p1^T} - x'\mathbf{p3^T} \end{bmatrix} \mathbf{X} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

We use SVD to solve for X. Using SVD's special case where b = 0, we take the last row of V.T matrix (the last column of V matrix)
$\mathbf{P}$ matrix here is the C matrix in the Python program. The specific terms of the A matrix is as follows:

$$A = \begin{bmatrix} y\mathbf{p3^T} - \mathbf{p2^T} \\ \mathbf{p1^T} - x\mathbf{p3^T} \\ y'\mathbf{p3^T} - \mathbf{p2^T} \\ \mathbf{p1^T} - x'\mathbf{p3^T} \end{bmatrix} = \begin{bmatrix} y\mathbf{C1}row3 - \mathbf{C1}row2 \\ \mathbf{C1}row1 - x\mathbf{C1}row3 \\ y'\mathbf{C2}rpw3 - \mathbf{C2}row2 \\ \mathbf{C2}row1 - x'\mathbf{C2}row3 \end{bmatrix}$$

$$A = \begin{bmatrix} y_1 * C1_{31} - C1_{21} & y_1 * C1_{32} - C1_{22} & y1 * C1_{33} - C1_{23} & y1 * C1_{34} - C1_{24} \\ C1_{11} - x_1 * C1_{31} & C1_{12} - x_1 * C1_{32} & C1_{13} - x_1 * C1_{33} & C1_{14} * x_1 * C1_{34} \\ y_2 * C2_{31} - C2_{21} & y_2 * C2_{32} - C2_{22} & y2 * C2_{33} - C2_{23} & y2 * C2_{34} - C2_{24} \\ C2_{11} - x_2 * C2_{31} & C2_{12} - x_2 * C2_{32} & C2_{13} - x_2 * C2_{33} & C2_{14} * x_2 * C2_{34} \end{bmatrix}$$

Q3.2

```
def triangulate(C1, pts1, C2, pts2):
    # Replace pass by your implementation
    # ----- TODO -----
    # YOUR CODE HERE
    P = np.zeros((pts1.shape[0], 3))
    err = 0

    # (1) For every input point, form A using the corresponding points from
                                            pts1 & pts2 and C1 & C2
    # Math reference: https://www.cs.cmu.edu/~16385/s17/Slides/11.
                                            4_Triangulation.pdf
    p11, p21, p31 = C1[0, :], C1[1, :], C1[2, :]
    p12, p22, p32 = C2[0, :], C2[1, :], C2[2, :]
    for i in range(pts1.shape[0]):
        xi1, yi1, xi2, yi2 = pts1[i][0], pts1[i][1], pts2[i][0], pts2[i][1]
        Ai = np.array([
            yi2 * p32 - p22,
            p12 - xi2 * p32,
            yi1 * p31 - p21,
            p11 - xi1 * p31
        ])

        # (2) Solve for the least square solution using np.linalg.svd
        U, Sigma, VT = np.linalg.svd(Ai, 0)
        Xi = VT[-1, :]
        Xi = Xi/Xi[-1]

        # (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)
        proji1, proji2 = np.matmul(C1, Xi.reshape((4, 1))), np.matmul(C2, Xi.
                                            reshape(4, 1))
        proji1, proji2 = (proji1/proji1[-1][0])[:-1], (proji2/proji2[-1][0])[:
                                            -1]

        # (4) Keep track of the 3D points and projection error, and continue
                                            to next point
        P[i] = Xi[:-1]
        xiyi1, xiyi2 = pts1[i].reshape(2, 1), pts2[i].reshape(2, 1)
        err1, err2 = np.linalg.norm(xiyi1 - proji1), np.linalg.norm(xiyi2 -
                                            proji2)
        err += err1**2 + err2**2

    return P, err
```

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

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

Q3.3

```
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'.
https://www.overleaf.com/project/652c738442ecf5d88af5ab4a
    """
    # ----- TODO -----
    # YOUR CODE HERE
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    E = essentialMatrix(F, K1, K2)
    M2s = camera2(E)
    M1 = np.hstack((np.identity(3), np.zeros(3)[:, np.newaxis]))

    minErr = np.Inf

    for i in range(M2s.shape[-1]):
        thisM2 = M2s[:, :, i]
        C1, C2 = np.matmul(K1, M1), np.matmul(K2, thisM2)
        thisP, thisErr = triangulate(C1, pts1, C2, pts2)
        if thisErr < minErr:
            minErr = thisErr
            P = thisP
            M2 = thisM2

    C2 = np.matmul(K2, M2)
    return M2, C2, 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.

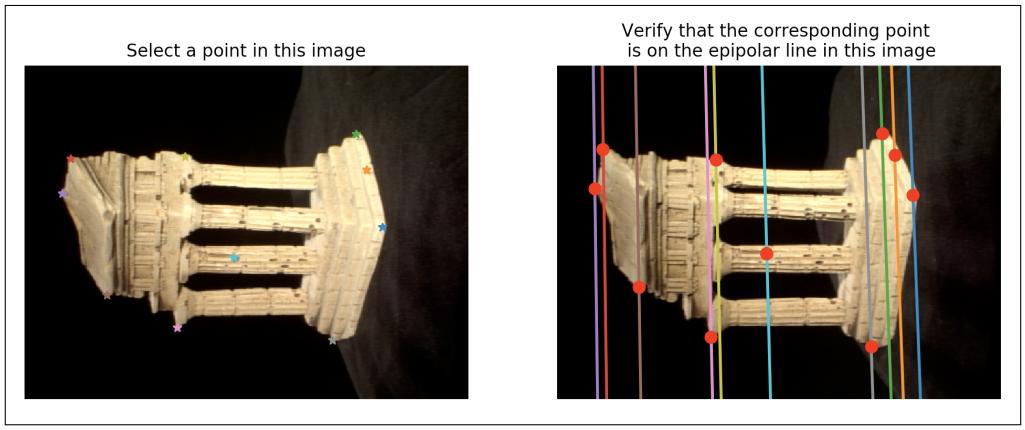


Figure 5: `epipolarMatchGUI` shows the corresponding point found by calling `epipolarCorrespondence`

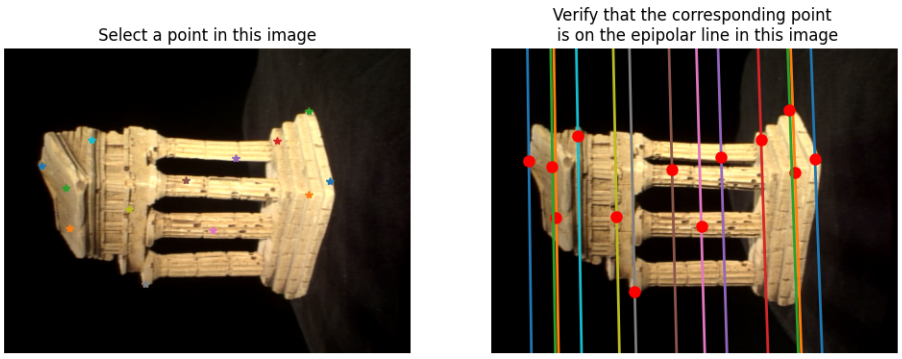
It's not necessary for your matcher to get *every* possible point right, but it should get easy points (such as those with distinctive, corner-like windows). It should also be good enough to render an intelligible representation in the next question.

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

```
from scipy.ndimage import
def epipolarCorrespondence(im1, im2, F, x1, y1):
    imW, imH = im1.shape[1], im1.shape[0]
    W = 7 # for convenience, define 0.5*window size in number of pixels as W
    R = 5*W # search range
    # (1) Given input [x1, x2], use the fundamental matrix to recover the
                                            corresponding epipolar line on
                                            image2
    point1 = np.array([x1, y1, 1])
    epi_coords = np.matmul(F, point1)
    a, b, c = epi_coords[0], epi_coords[1], epi_coords[2]

    # (2) Search along this line to check nearby pixel intensity (you can
                                            define a search window) to  find
                                            the best matches
    x2, y2, minErr = 0, 0, np.Inf
    xstart1, xend1, ystart1, yend1 = x1 - W, x1 + W, y1 - W, y1 + W # check
                                            window in x1, y1's surroundings
    if xstart1 <= W or xend1 >= imW-W or ystart1 <= W or yend1 >= imH-W:
        return x1, y1
    window1 = im1[ystart1:yend1, xstart1:xend1] # original window in image 1
    # slide along the correct axis to avoid sliding out if the image and
                                            getting zero intensity values
    slideX = False
    if b != 0:
        if np.absolute(a/b) < imH/imW:
            slideX = True
    if slideX:
        # slide along x axis only if the absolute slope is relatively small
        minX, maxX = max(W+1, int(xstart1-(imW/imH)*R)), min(int(xend1+(imW/
                                                imH)*R), imW-W)
        for myX in range(minX, maxX):
            myY = int((-a/b)*myX - c/b)
            if myY > W and myY < imH - W:
                xstart2, xend2, ystart2, yend2 = myX - W, myX + W, myY - W,
                                                        myY + W
                window2 = im2[ystart2:yend2, xstart2:xend2]
                # (3) Use guassian weighting to weight the pixel simlairty
                err = gaussian_filter(np.absolute(window1 - window2), sigma=1)
                                                        .mean()
                if err < minErr:
                    x2, y2, minErr = myX, myY, err
    else:
        # or we slide along y axis
        minY, maxY = max(W+1, int(ystart1-(imH/imW)*R)), min(int(yend1+(imH/
                                                imW)*R), imH-W)
        for myY in range(minY, maxY):
            myX = int((-b/a)*myY - c/a)
            if myX > W and myX < imW - W:
                xstart2, xend2, ystart2, yend2 = myX - W, myX + W, myY - W,
                                                        myY + W
                window2 = im2[ystart2:yend2, xstart2:xend2]
                # (3) Use guassian weighting to weight the pixel simlairty
                err = gaussian_filter(np.absolute(window1 - window2), sigma=1)
                                                        .mean()
                if err < minErr:
                    x2, y2, minErr = myX, myY, err
    return x2, y2
```

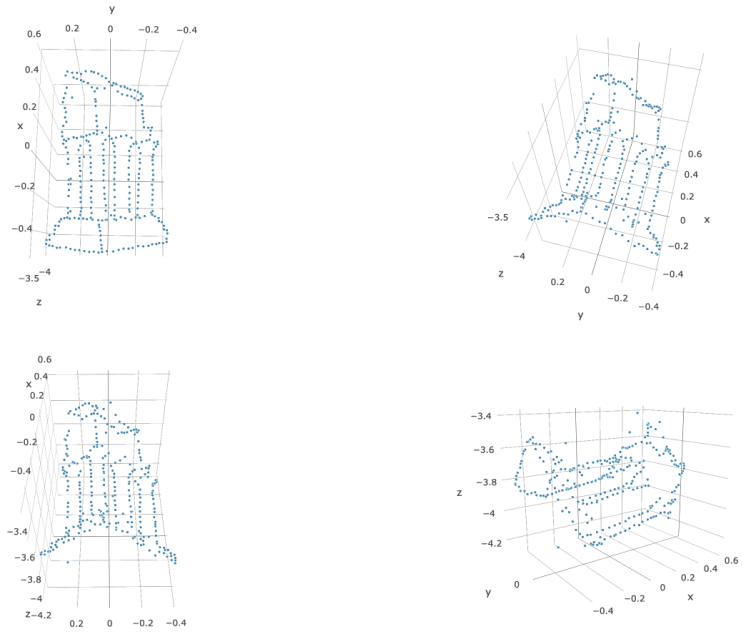


Figure 6: An example point cloud

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

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

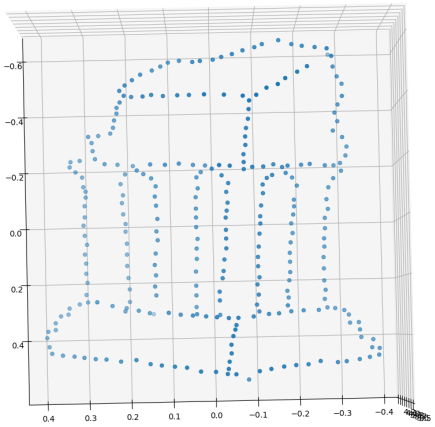
Output: Again, save the matrix $\mathbf{F}$, matrices $\mathbf{M1}, \mathbf{M2}, \mathbf{C1}, \mathbf{C2}$ which you used to generate the screenshots to the file `q4_2.npz`.

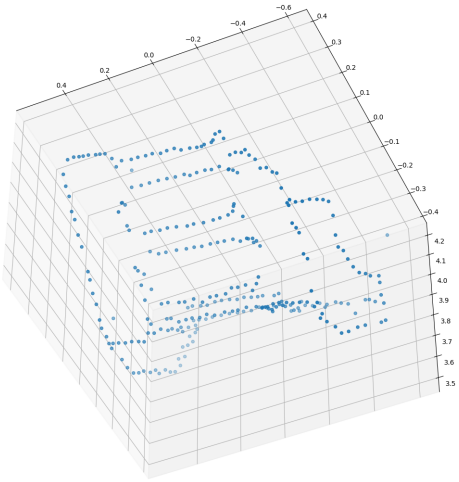
In your write-up:

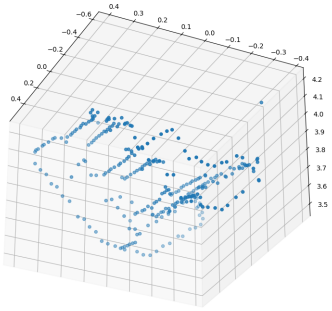
- Take a few screenshots of the 3D visualization so that the outline of the temple is clearly visible, and include them in your writeup.
- Include the code snippet of `compute3D_pts` function in your write-up.

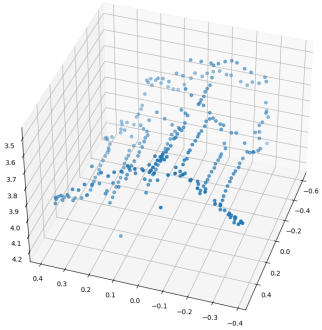
Q4.2

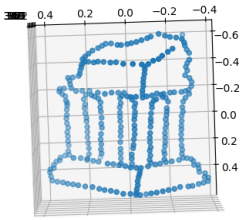
findM2 minimum error 517.6303994067101

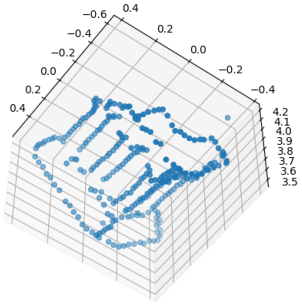


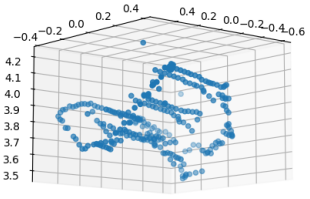


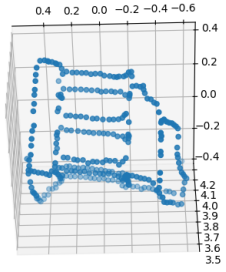












Q4.2

```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    # ----- TODO -----
    # YOUR CODE HERE
    # (1) Use epipolarCorrespondence to find the corresponding point for [x1
                                                y1] (find [x2, y2])
    N = temple_pts1.shape[0]
    x1s, y1s = temple_pts1[:, 0], temple_pts1[:, 1]
    temple_pts2 = np.zeros((N, 2))
    for i in range(N):
        temple_pts2[i] = epipolarCorrespondence(im1, im2, F, x1s[i], y1s[i])
    print(temple_pts2)
    # (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)
    M2, C2, P = findM2(F, temple_pts1, temple_pts2, intrinsics)
    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

RANSAC compared with eightpoint (on noisy data)
Error of all points from eightpoint: 7335.446557409245
Error of inliers from eightpoint: 474.4501682191251
Error of all points from RANSAC: 14126.68435567927 (**higher than eightpoint**)
Error of inliers from RANSAC: 0.5778256160166599 (**much lower than eightpoints**)
Maximum number of inliers from RANSAC: 110
From the error results we can see that RANSAC does a better job calculating the F that fits inliers, while eightpoints simply tries to calculate on all data and was heavily influenced by outliers by having lower overall error but significantly higher error on inliers.
Therefore, RANSAC is able to compute a good F even with noisy data.

Error metrics I used:
Sum of squared errors.
It is used for for deciding which points are inliers, and for an overall noisy extent of the correspondence.

How I decide which points are inliers:
I used the helper function

```
calc_epi_error(pts1_homo, pts2_homo, F)
```

to calcualte the sum of squared distance between the corresponding points and the estimated epipolar lines. If a point's error distance is smaller than "tol" then the point is an inlier.

Effect of nIters:
Increasing nIter increases the chance of getting a better F. This is because each time we randomly sample 8 pairs of points and these 8 points might be really noisy points or not so noisy ones. They might or might not be a good representation for the entire corresponding point pair. However, choosing too large nIter values can be really time inefficient.

Effect of tol:
Tolerence is the error distance threshold value to consider a point as an outlier. If this value is too big, we will allow noisy points to be considered as inliers from the epipolar lines. This way the best F calculated might not be accurate enough. If this value is too small, it does not take into account the noise of the correspondence, and the number of inliers can be too small, and can lead to not finding an F or finding a bad F. Generally, this value should be small.

Q5.1

```
def ransacF(pts1, pts2, M, nIters=1000, tol=5):
    # TODO: Replace pass by your implementation
    N = pts1.shape[0]
    x = 8
    it, maxnumInliers, minErr = 0, 0, np.Inf
    while it < nIters:
        idx = random.sample(range(0, N), x)
        sampled_points1, sampled_points2 = pts1[idx], pts2[idx]
        myF = eightpoint(sampled_points1, sampled_points2, M)
        pts1_homo, pts2_homo = np.hstack((pts1, np.ones((N, 1)))), np.hstack((
                                                    pts2, np.ones((N, 1))))
        err = calc_epi_error(pts1_homo, pts2_homo, myF)
        in_idx = np.where(err < tol)[0]
        numInliers = len(in_idx)
        if numInliers > maxnumInliers:
            F, inliers = myF, np.where(err < tol, True, False).reshape((N, 1))
            minErr, maxnumInliers = err[in_idx].mean(), numInliers,
        it += 1

    return F, inliers
```

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

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

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

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

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

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

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

Reference: Rodrigues formula and this pdf.

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

Q5.2

```
def rodrigues(r):
    # TODO: Replace pass by your implementation
    theta = np.linalg.norm(r)
    r = r.reshape((3, 1))
    u = r/theta
    I = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
    ux = np.array([[0, -u[2][0], u[1][0]], [u[2][0], 0, -u[0][0]], [-u[1][0],
                                                u[0][0], 0]])
    R = I * np.cos(theta) + (1 - np.cos(theta)) * np.matmul(u, u.T) + ux * np.
                                                sin(theta)
    return R


def invRodrigues(R):
    # TODO: Replace pass by your implementation
    A = 0.5 * (R - R.T)
    rho = np.array([A[2][1], A[0][2], A[1][0]])
    s = np.linalg.norm(rho)
    c = 0.5 * (R[0][0] + R[1][1] + R[2][2] - 1)

    if s == 0 and c == 1:
        r = np.array([0, 0, 0])
    elif s == 0 and c == -1:
        RI = R + np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
        normL = [np.linalg.norm(RI[:, 0]), np.linalg.norm(RI[:, 1]), np.linalg
                                                    .norm(RI[:, 2])]
        idx = normL.index(max(normL))
        v = RI[:, idx]
        u = v/np.linalg.norm(v)
        u = u.reshape((0, 3))
        r = -u * np.pi
    else:
        u = rho / s
        theta = np.arctan2(s, c)
        r = u * theta

    return r
```

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.

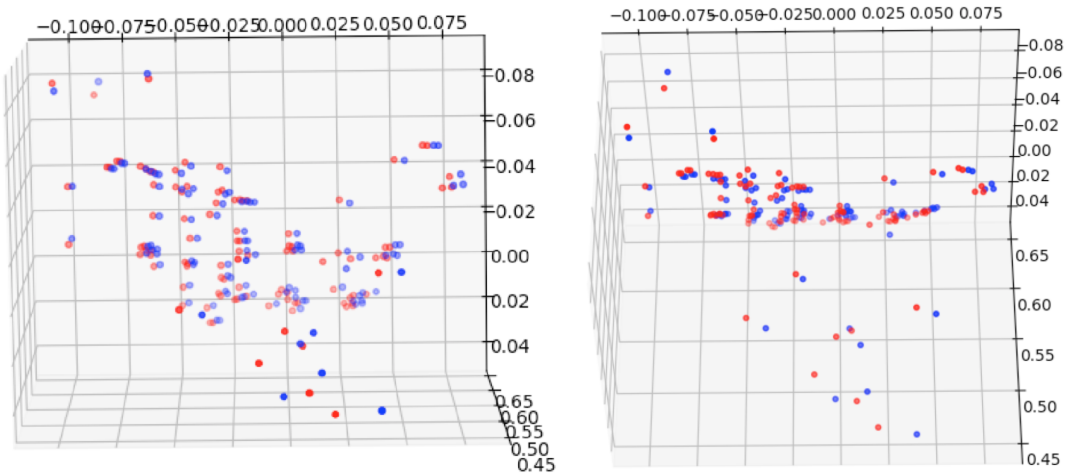
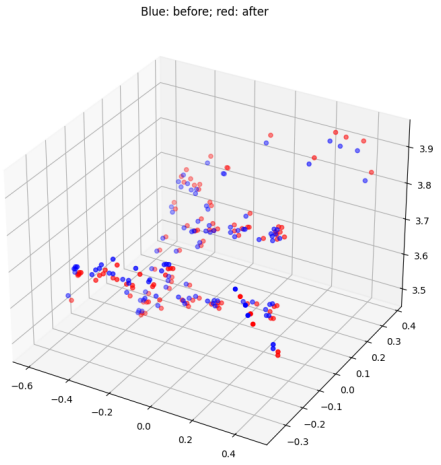
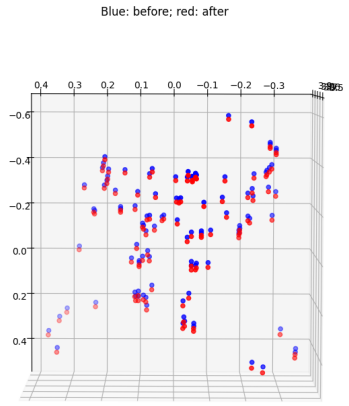


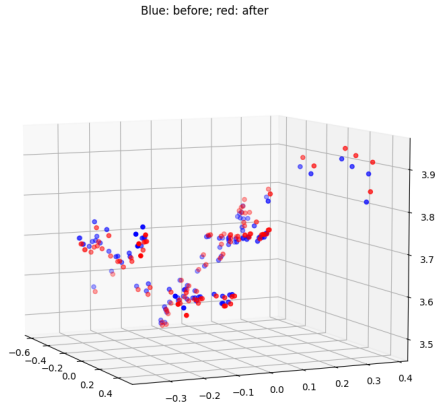
Figure 7: Visualization of 3D points for noisy correspondences before and after using bundle adjustment

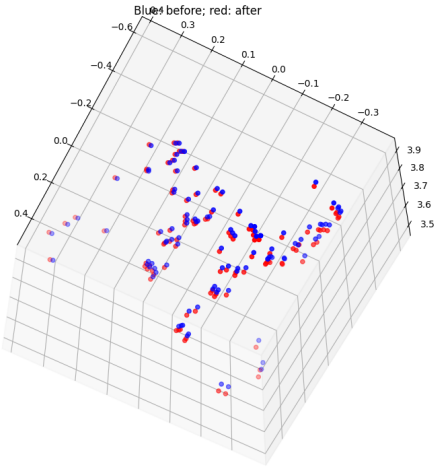
Q5.3

```
reprojection error start 12628.82513600413

reprojection error end (optimized) 9.486043765967437

M2 start
[[ 0.99940762  0.03059833  0.0157528  -0.01910485]
 [-0.03368792  0.96339274  0.2659691  -1.        ]
 [-0.00703792 -0.26634222  0.96385284  0.06156856]]

M2 end (optimized)
[[ 9.99431183e-01  3.26994045e-02  8.24979894e-03 -2.95267457e-02]
 [-3.37156837e-02  9.63383634e-01  2.65998546e-01 -1.00012517e+00]
 [ 7.50272745e-04 -2.66125389e-01  9.63938128e-01  1.24401877e-01]]
```









Q5.3

```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    # TODO: Replace pass by your implementation
    # (1) recover M2 from (R2, t2) in x
    t2, r2, P = x[-3:], x[-6:-3], x[:-6]
    R2 = rodrigues(r2)
    P = P.reshape((int(len(P)/3), 3))
    M2 = np.hstack((R2, t2.reshape((3, 1))))

    # (2) calculate projected points in both images
    C1 = np.matmul(K1, M1)
    C2 = np.matmul(K2, M2)
    P_homo = np.hstack((P, np.ones((P.shape[0], 1)))) # N x 4
    proj1, proj2 = np.matmul(C1, P_homo.T), np.matmul(C2, P_homo.T) # 3 x 4 by
                                                    4 x N = 3 x N
    proj1, proj2 = (proj1/proj1[-1])[:-1].T, (proj2/proj2[-1])[:-1].T #
                                                   Normalise and reshape into N x 2

    # (3) calculate residuals
    residual = np.append((p1 - proj1).flatten(), (p2 - proj2).flatten())
    return residual

def myFunc(x, K1, M1, p1, K2, p2):
    # objective function, the reprojection error
    return np.linalg.norm(rodriguesResidual(K1, M1, p1, K2, p2, x))**2

def bundleAdjustment(K1, M1, p1, K2, M2_init, p2, P_init):
    obj_start = obj_end = 0
    # ----- TODO -----
    # YOUR CODE HERE
    # (1) Calculate initial residual
    R2_init, t2_init = M2_init[:, :-1], M2_init[:, -1]
    r2_init= invRodrigues(R2_init)
    x_init = np.append(P_init.flatten(), r2_init)
    x_init = np.append(x_init, t2_init)
    obj_start = rodriguesResidual(K1, M1, p1, K2, p2, x_init)

    # (2) Minimize objective function using scipy.optimize.minimize the
                                                    rodrigues residual
    x_op = scipy.optimize.minimize(myFunc, x_init, args=(K1, M1, p1, K2, p2),
                                                   method="Powell").x

    # (3) Collect the optimized results
    obj_end = rodriguesResidual(K1, M1, p1, K2, p2, x_op)
    t2, r2, P = x_op[-3:], x_op[-6:-3], x_op[:-6]
    R2 = rodrigues(r2)
    P = P.reshape((int(len(P)/3), 3))
    M2 = np.hstack((R2, t2.reshape((3, 1))))

    return M2, P, obj_start, obj_end
```

Q5.3

The following modifications are made to findM2()
We select the M2 that satisfies: the 3D points P are in front of both camera 1 an camera 2, which means, P's mean value of Z coordinates should be greater than 0 for both cameras.

```
def findM2(F, pts1, pts2, intrinsics, filename="q3_3.npz"):

    # ----- TODO -----
    # YOUR CODE HERE
    K1, K2 = intrinsics["K1"], intrinsics["K2"]
    E = essentialMatrix(F, K1, K2)
    M2s = camera2(E)
    M1 = np.hstack((np.identity(3), np.zeros(3)[:, np.newaxis]))

    minErr = np.Inf

    for i in range(M2s.shape[-1]):
        # thisM2 = M2s[:, :, 0] # for q4_2
        thisM2 = M2s[:, :, i]
        C1, C2 = np.matmul(K1, M1), np.matmul(K2, thisM2)
        thisP, thisErr = triangulate(C1, pts1, C2, pts2)
        # -- for q5
        thisP2 = np.matmul(thisM2, np.hstack((thisP, np.ones((thisP.shape[0],
                                                    1))))).T).T
        if np.mean(thisP[:, -1]) > 0 and np.mean(thisP2[:, -1]) > 0:
            minErr = thisErr
            P = thisP
            M2 = thisM2
        # -- (end) for q5
        # if thisErr < minErr:
        #     minErr = thisErr
        #     P = thisP
        #     M2 = thisM2
    C2 = np.matmul(K2, M2)
    print("FindM2 error", minErr)

    return M2, C2, P
```

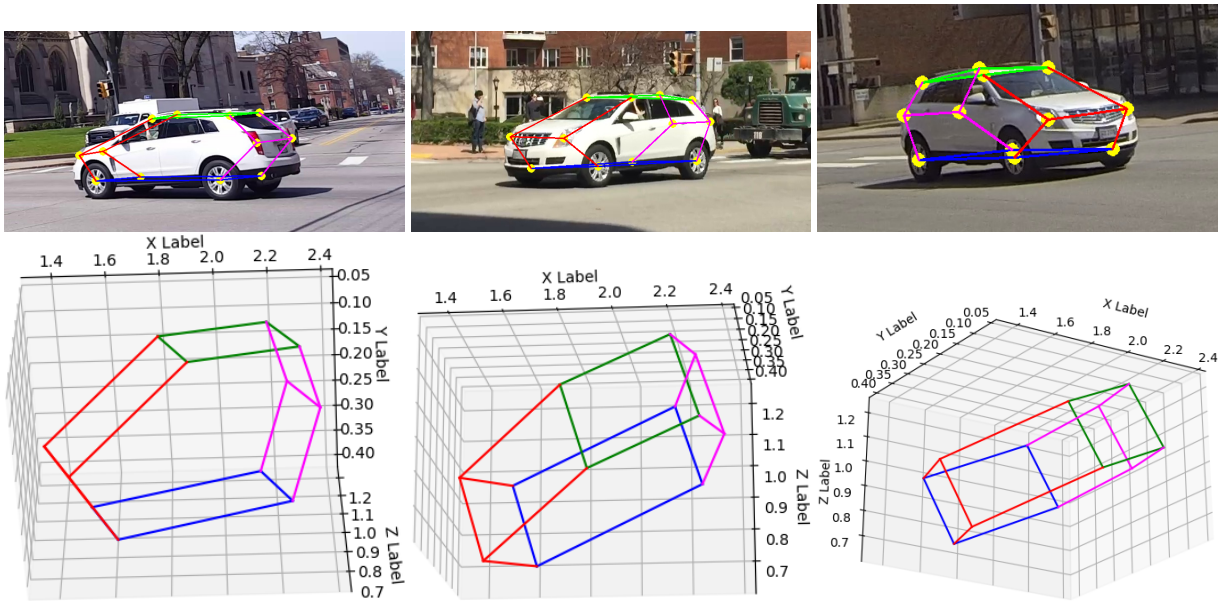


Figure 8: An example detections on the top and the reconstructions from multiple views

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

Q6.1 [Extra Credit - 15 points] Write a function to compute the 3D keypoint locations `P` given the 2D part detections `pts1`, `pts2` and `pts3` and the camera projection matrices `C1`, `C2`, `C3`. The camera matrices are given in the numpy files.

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

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

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

[2] Code Used For Detection and Reconstruction

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

Method I used to compute 3D locations

(1) Find the point indices where at least 2 points have confidence level greater than the threshold

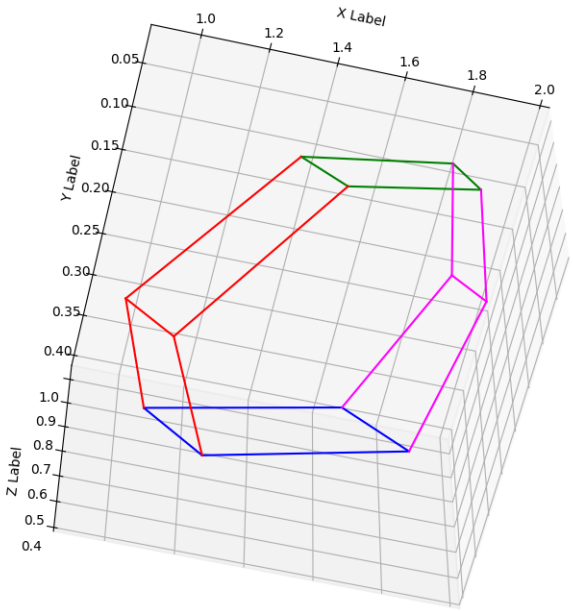
(2) Loop through these selected indices. We construct the A matrix and solve for the 3D point using SVD, Similar to question 3.

- If there are all 3 points have confidence level greater than the threshold, then we construct A matrix with the cross product of All C1, C2, C3 with the corresponding x's and y's to compute their corresponding 3D point. The A matrix has 6 rows in this case.

- If there are only 2 points with confidence level greater than the threshold, then we construct A matrix with the cross product of the 2 C matrices and the 2 pairs of x's and y's. The A matrix has only 4 rows in this case.

(3) For the reprojection error, we take the mean of the sum of squared norm. First add up the squared error norm. Then divide by the number of points points with confidence value greater than the threshold (either 2 or 3). The result is sufficiently accurate with or without bundle adjustment.

Reprojection error: 756.88



Q6.1

```
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres=300):
    # TODO: Replace pass by your implementation
    N = pts1.shape[0]
    P = np.zeros((N, 3))
    err = 0
    mask1, mask2, mask3 = np.where(pts1[:, -1]<Thres, 0, 1), np.where(pts2[:,
                                                -1]<Thres, 0, 1), np.where(pts3[:
                                                , -1]<Thres, 0, 1)
    in_idx = np.where(mask1 + mask2 + mask3 > 1)[0]
    p11, p21, p31 = C1[0, :], C1[1, :], C1[2, :]
    p12, p22, p32 = C2[0, :], C2[1, :], C2[2, :]
    p13, p23, p33 = C3[0, :], C3[1, :], C3[2, :]

    for i in in_idx:
        # (1) For every input point, form A using the corresponding points
                                                from pts1 pts2 & pts3 and C1
                                                C2 & C3
        # Math reference: https://www.cs.cmu.edu/~16385/s17/Slides/11.
                                                4_Triangulation.pdf
        Ai = np.array([])
        if mask1[i]:
            Ai = np.append(Ai, np.array([pts1[i][1] * p31 - p21, p11 - pts1[i]
                                                [0] * p31])).reshape(-1,
                                                4)
        if mask2[i]:
            Ai = np.append(Ai, np.array([pts2[i][1] * p32 - p22, p12 - pts2[i]
                                                [0] * p32])).reshape(-1,
                                                4)
        if mask3[i]:
            Ai = np.append(Ai, np.array([pts3[i][1] * p33 - p23, p13 - pts3[i]
                                                [0] * p33])).reshape(-1,
                                                4)

        # (2) Solve for the least square solution using np.linalg.svd
        U, Sigma, VT = np.linalg.svd(Ai, 0)
        Xi = VT[-1, :]
        Xi = Xi/Xi[-1]
```

CONTINUED ON THE NEXT PAGE

Q6.1

```
            # (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)
            XiT = Xi.reshape(4, 1)
            P[i] = Xi[:-1]
            erri = 0
            if mask1[i]:
                proji1 = np.matmul(C1, XiT)
                proji1 = (proji1/proji1[-1][0])[:-1]
                xiyi1 = pts1[i][:-1].reshape(2, 1)
                err1 = np.linalg.norm(xiyi1 - proji1)
                erri += err1**2
            if mask2[i]:
                proji2 = np.matmul(C2, XiT)
                proji2 = (proji2/proji2[-1][0])[:-1]
                xiyi2 = pts2[i][:-1].reshape(2, 1)
                err2 = np.linalg.norm(xiyi2 - proji2)
                erri += err2**2
            if mask3[i]:
                proji3 = np.matmul(C3, XiT)
                proji3 = (proji3/proji3[-1][0])[:-1]
                xiyi3 = pts3[i][:-1].reshape(2, 1)
                err3 = np.linalg.norm(xiyi3 - proji3)
                erri += err3**2

            # (4) Keep track of the 3D points and projection error, and continue
                                                    to next point
            erri = erri / (0.5*Ai.shape[0])
            err += erri
            print("erri", i, erri)

        print("err", err)
        return P, err
```

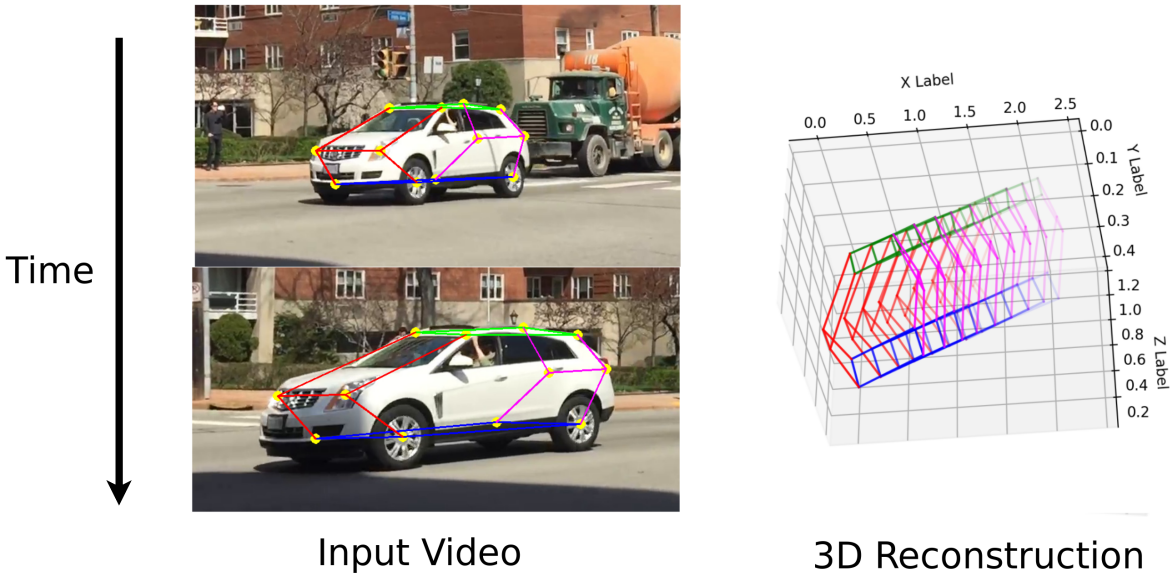


Figure 9: Spatiotemporal reconstruction of the car (right) with the projections at two different time instances in a single view(left)

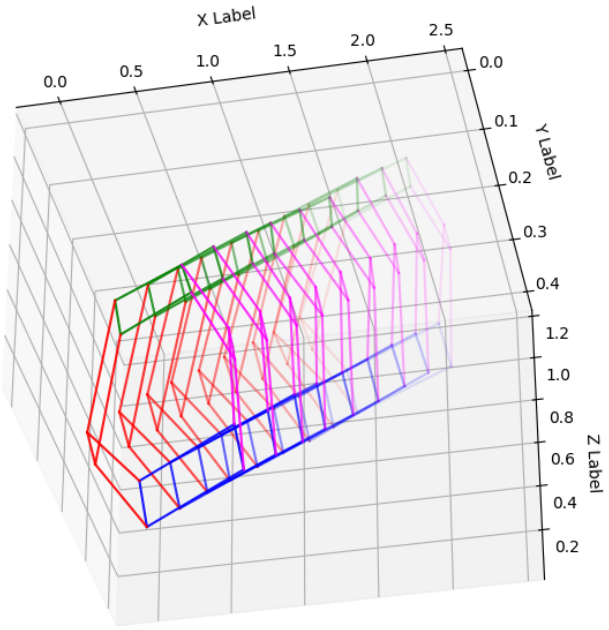
Q6.2 [Extra Credit - 15 points]

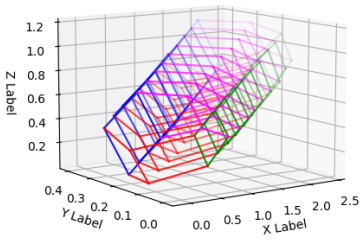
From the previous question you have done a 3D reconstruction at a time instance. Now you are going to iteratively repeat the process over time and compute a spatio temporal reconstruction of the car. The images in the `data/q6` folder shows the motion of the car at an intersection captured from multiple views. The images are given as `(cam1_time0.jpg, ..., cam1_time9.jpg)` for camera 1 and `(cam2_time0.jpg, ..., cam2_time9.jpg)` for camera2 and `(cam3_time0.jpg, ..., cam3_time9.jpg)` for camera3. The corresponding detections and camera matrices are given in `(time0.npz, ..., time9.npz)`. Use the above details and compute the spatio temporal reconstruction of the car for all 10 time instances and plot them by completing the `plot_3d_keypoint_video` function. A sample plot with the first and last time instance reconstuction of the car with the reprojections shown in the Figure 9.

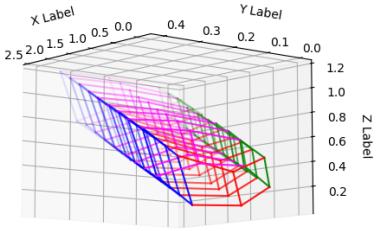
In your write-up:

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

Q6.2


X Label
0.0
0.5
1.0
1.5
2.0
2.5
Y Label
0.0
0.1
0.2
0.3
0.4
Z Label
1.2
1.0
0.8
0.6
0.4
0.2


Z Label
1.2
1.0
0.8
0.6
0.4
0.2
Y Label
0.4
0.3
0.2
0.1
0.0
X Label
0.0
0.5
1.0
1.5
2.0
2.5


X Label
2.5
2.0
1.5
1.0
0.5
0.0
Y Label
0.4
0.3
0.2
0.1
0.0
Z Label
1.2
1.0
0.8
0.6
0.4
0.2

Q6.2

```
plot_3d_keypoint_video(pts_3d_video):
    # TODO: Replace pass by your implementation
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")

    N = len(pts_3d_video)
    for i in range(N):
        pts_3d = pts_3d_video[i]
        num_points = pts_3d.shape[0]
        for j in range(len(connections_3d)):
            index0, index1 = connections_3d[j]
            xline = [pts_3d[index0, 0], pts_3d[index1, 0]]
            yline = [pts_3d[index0, 1], pts_3d[index1, 1]]
            zline = [pts_3d[index0, 2], pts_3d[index1, 2]]
            ax.plot(xline, yline, zline, color=colors[j], alpha = i/N)

    np.set_printoptions(threshold=1e6, suppress=True)
    ax.set_xlabel("X Label")
    ax.set_ylabel("Y Label")
    ax.set_zlabel("Z Label")
    plt.show()
```

7 Deliverables

The assignment (code and write-up) should be submitted to Gradescope. The write-up should be named `<AndrewId>_hw3.pdf` and the code should be a zip named `<AndrewId>_hw3.zip`. ***Please make sure that you assign the location of answers to each question on Gradescope.*** The zip should have the following files in the structure defined below. (Note: Neglecting to follow the submission structure will incur a huge score penalty!). You can run the included `checkA4Submission.py` script to ensure that your zip folder structure is correct.

- `<AndrewId>_hw3.pdf`: your write-up.
- `q2_1_eightpoint.py`: script for Q2.1.
- `q2_2_sevenpoint.py`: script for Q2.2.
- `q3_1_essential_matrix.py`: script for Q3.1.
- `q3_2_triangulate.py`: script for Q3.2.
- `q4_1_epipolar_correspondence.py`: script for Q4.1.
- `q4_2_visualize.py`: script for Q4.2.
- `q5_bundle_adjustment.py`: script for Q5.
- `q6_ec_multiview_reconstruction.py`: script for (extra-credit) Q6.
- `helper.py`: helper functions.
- `q2_1.npz`: file with output of Q2.1.
- `q2_2.npz`: file with output of Q2.2.
- `q3_1.npz`: file with output of Q3.1.
- `q3_3.npz`: file with output of Q3.3.
- `q4_1.npz`: file with output of Q4.1.
- `q4_2.npz`: file with output of Q4.2.
- `q6_1.npz`: (extra-credit) file with the output of Q6.1.

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

8 FAQs

Credits: Paul Nadan

Q2.1: Does it matter if we unscale **F** before or after calling refineF?

The relationship between $\mathbf{F}$ 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.