Homework 4: 3D Reconstruction

Srividya Gandikota
sgandiko
sgandiko@andrew.cmu.edu

1 Theory

1.1 Suppose two cameras fixate on a point P (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 F_{33} element of the fundamental matrix is zero.

At the principle point P, we see that both left and right images have $p_1 = p_2 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$. Using the relationship we learned in class, we know that the fundamental matrix allows us to assume that $p_1^T \mathrm{F} p_2 = 0$. If we complete the calculation, we see that the the resulting calculation below is equivalent to 0. We notice here that the only way that the result of this calculation of 0 is possible only if F_{33} is 0 as well.

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

$$\downarrow$$

$$\begin{bmatrix} F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0$$

$$\downarrow$$

$$F_{33} = 0$$

1.2 Consider the case of two cameras viewing an object such that the second camera differs from the first by a pure translation that is parallel to the x-axis. Show that the epipolar lines in the two cameras are also parallel to the x-axis. Backup your argument with relevant equations.

Given that we want to show that there is a pure translation parallel to the x-axis, there are a few key matrices that we will need in order to prove this, essential matrix E, rotation matrix R, camera intrinsics K and translation vector t.

$$\mathrm{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathrm{t} = \begin{bmatrix} t_1 \\ 0 \\ 0 \end{bmatrix} \quad \mathrm{E} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix} \quad \mathrm{K} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

With the assumption that this system experiences pure translation, we can use these matrices to calculate the epipolar lines as such given that $x_1^T = \begin{bmatrix} a_1 & a_2 & 1 \end{bmatrix}$ and $x_2^T = \begin{bmatrix} b_1 & b_2 & 1 \end{bmatrix}$:

$$l_1^T = x_2^T E = \begin{bmatrix} b_1 & b_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & t_1 & -b_2 t_1 \end{bmatrix}$$

$$l_2^T = x_1^T E = \begin{bmatrix} a_1 & a_2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_1 \\ 0 & t_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -t_1 & a_2 t_1 \end{bmatrix}$$

We can see in these results that there is no x term present and that these two lines are parallel to the x-axis.

1.3 Suppose we have an inertial sensor which gives us the accurate positions (Ri and ti, the rotation matrix and translation vector) of the robot at time i. What will be the effective rotation (Rrel) and translation (trel) between two frames at different time stamps? Suppose the camera intrinsics (K) are known, express the essential matrix (E) and the fundamental matrix (F) in terms of K, Rrel and trel.

In order to actually determine the essential matrix E and fundamental matrix F, we need to also consider the point position within the real world w = $\begin{bmatrix} U & V & W \end{bmatrix}^T$ and its location at time i as $\begin{bmatrix} x_i & y_1 & 1 \end{bmatrix}^T$.

Using the relationship $\hat{p}_l = K_l^{-1} p_l$, we see that at time 1 $\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \text{K} \left(R_1 \begin{bmatrix} U \\ V \\ W \end{bmatrix} + t_1 \right)$.

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = R_1^{-1} \left(K^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - t_1 \right) = R_1^T K^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - R_1^T t_1$$

At time 2, $\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = K \left(R^2 \begin{bmatrix} U \\ V \\ W \end{bmatrix} + t_2 \right) = \text{K} \left(R_2 \left(R_1^T K^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - R_1^T t_1 \right) + t_2 \right)$.

This can be simplified down to such:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = K R_2 R_1^T K^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} - K R_2 R_1^T t_1 + K t_2$$

$$\text{E} = t_{rel} \text{ x } R_{rel}$$
$$t_{rel} = -K R_2 R_1^T t_1 + K t_2$$
$$R_{rel} = K R_2 R_1^T K^{-1}$$
$$F = (K^{-1})^T (t_{rel} \text{ x } R_{rel}) \; K^{-1}$$

1.4 Suppose that a camera views an object and its reflection in a plane mirror. Show that this situation is equivalent to having two images of the object which are related by a skew-symmetric fundamental matrix. You may assume that the object is flat, meaning that all points on the object are of equal distance to the mirror.

Given that we assume that the object is flat, we know that all the points on the object are of equal distance to the mirror. This means that the rotation matrix R will be the identity matrix. The calculations below depict how the fundamental matrix F is a skew symmetric matrix.

$$R_{rel} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad t_{rel} = \begin{bmatrix} t_x & t_y & t_z \end{bmatrix}$$

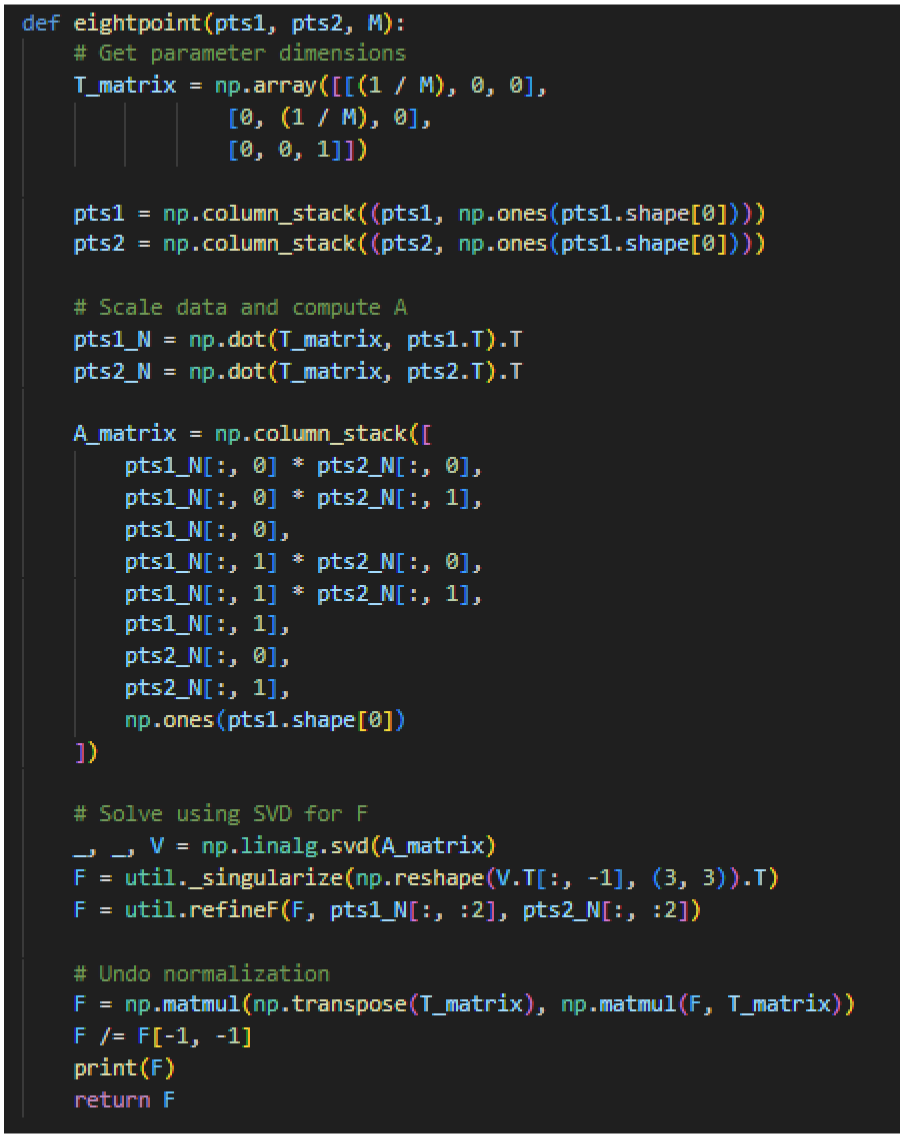
The essential matrix is still E = t x R which in this case is equal to $E = \begin{bmatrix} 0 & -t_y & -t_z \\ t_y & 0 & -t_x \\ t_z & t_x & 0 \end{bmatrix}$. From the previous section, we saw that the fundamental matrix can be found using $F = (K^{-1})^T(t_{rel} \text{ x } R_{rel})\ K^{-1}$ which with the matrix below can be seen as skew symmetric.

$$F = (K^{-1})^T \begin{bmatrix} 0 & -t_y & -t_z \\ t_y & 0 & -t_x \\ t_z & t_x & 0 \end{bmatrix} K^{-1}$$

2 Fundamental Matrix Estimation with The Eight Point Algorithm

2.1 Finish the function eightpoint in submission.py. Make sure you follow the signature for this portion of the assignment: F = eightpoint(pts1, pts2, M) where pts1 and pts2 are N × 2 matrices corresponding to the (x, y) coordinates of the N points in the first and second image respectively. M is a scale parameter.

My implementation for the function eightpoint can be found in Figure 1 below:



```python
def eightpoint(pts1, pts2, M):
    # Get parameter dimensions
    T_matrix = np.array([[(1 / M), 0, 0],
                        [0, (1 / M), 0],
                        [0, 0, 1]])

    pts1 = np.column_stack((pts1, np.ones(pts1.shape[0])))
    pts2 = np.column_stack((pts2, np.ones(pts1.shape[0])))

    # Scale data and compute A
    pts1_N = np.dot(T_matrix, pts1.T).T
    pts2_N = np.dot(T_matrix, pts2.T).T

    A_matrix = np.column_stack([
        pts1_N[:, 0] * pts2_N[:, 0],
        pts1_N[:, 0] * pts2_N[:, 1],
        pts1_N[:, 0],
        pts1_N[:, 1] * pts2_N[:, 0],
        pts1_N[:, 1] * pts2_N[:, 1],
        pts1_N[:, 1],
        pts2_N[:, 0],
        pts2_N[:, 1],
        np.ones(pts1.shape[0])
    ])

    # Solve using SVD for F
    _, _, V = np.linalg.svd(A_matrix)
    F = util._singularize(np.reshape(V.T[:, -1], (3, 3)).T)
    F = util.refineF(F, pts1_N[:, :2], pts2_N[:, :2])

    # Undo normalization
    F = np.matmul(np.transpose(T_matrix), np.matmul(F, T_matrix))
    F /= F[-1, -1]
    print(F)
    return F
```

Figure 1: eightpoint Function Implementation

The visualized epipolar lines can be found in Figure 2 below: The F matrix can be found below in Figure 3.

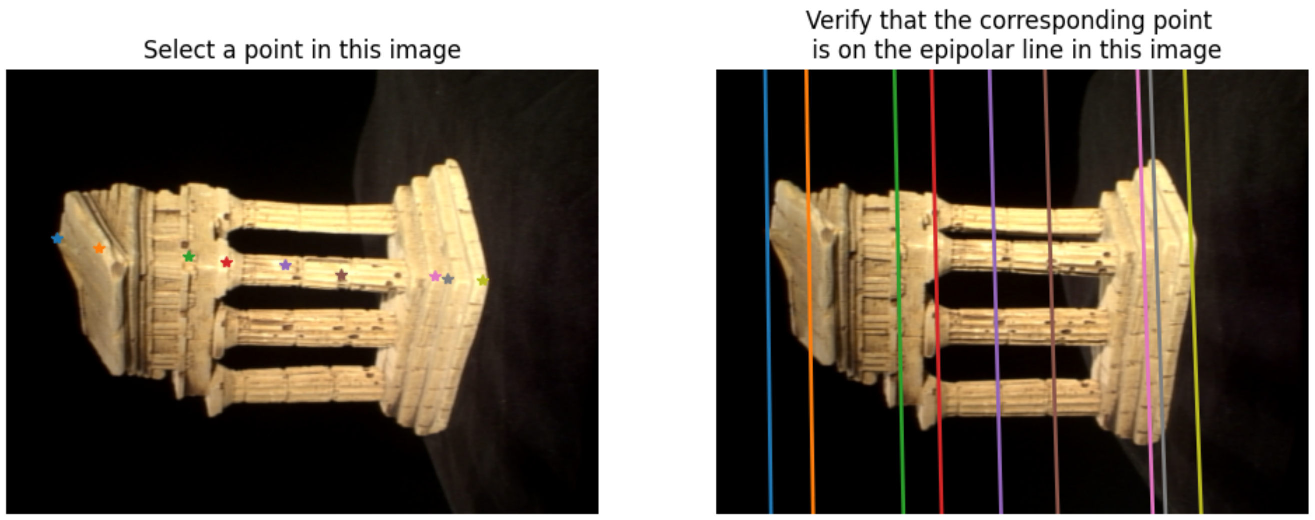
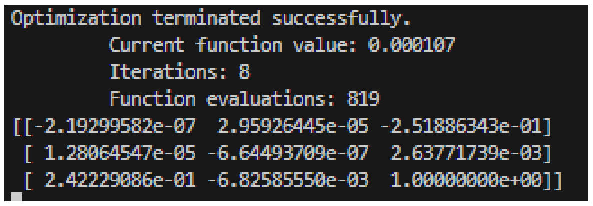

Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Figure 2: Visualized Epipolar Lines



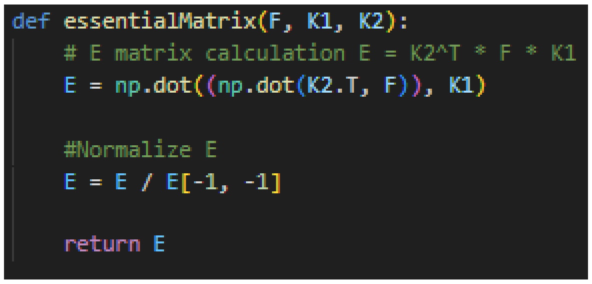
```
Optimization terminated successfully.
         Current function value: 0.000107
         Iterations: 8
         Function evaluations: 819
[[-2.19299582e-07  2.95926445e-05 -2.51886343e-01]
 [ 1.28064547e-05 -6.64493709e-07  2.63771739e-03]
 [ 2.42229086e-01 -6.82585550e-03  1.00000000e+00]]
```

Figure 3: Recovered F Matrix

3 Metric Reconstruction

3.1 Write a function to compute the essential matrix E given F, K1 and K2 with the signature: E = essentialMatrix(F, K1, K2) Output: Save your estimated E using F from the eight-point algorithm to q3 1.npz.

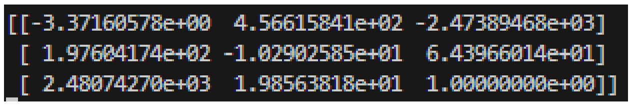
My implementation for the function essentialMatrix can be found in Figure 4 below:



```python
def essentialMatrix(F, K1, K2):
    # E matrix calculation E = K2^T * F * K1
    E = np.dot((np.dot(K2.T, F)), K1)

    #Normalize E
    E = E / E[-1, -1]

    return E
```

Figure 4: essentialMatrix Function Implementation

The results of the essential matrix can be found in Figure 5:

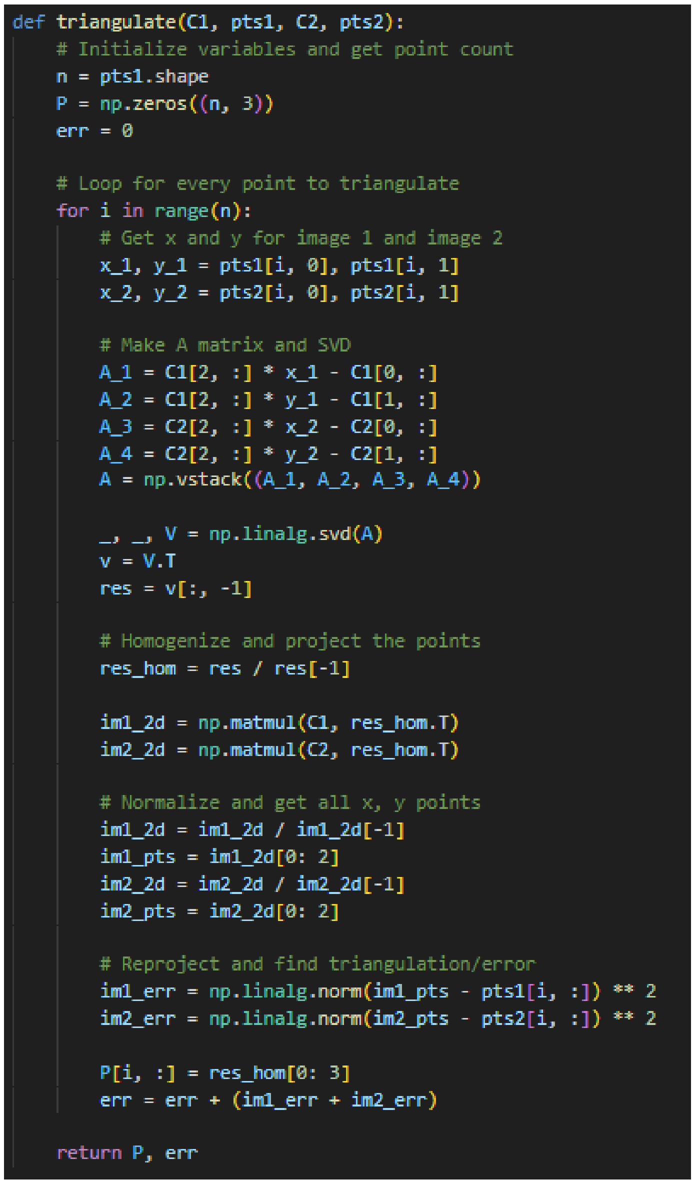


```
[[-3.37160578e+00  4.56615841e+02 -2.47389468e+03]
 [ 1.97604174e+02 -1.02902585e+01  6.43966014e+01]
 [ 2.48074270e+03  1.98563818e+01  1.00000000e+00]]
```

Figure 5: Essential Matrix

3.2 Using the above, write a function to triangulate a set of 2D coordi- nates 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 ×2 matrices with the 2D image coordinates and w is an N ×3 matrix with the corresponding 3D points per row. In your write-up: Write down the expression for the matrix Ai.

My implementation for the function triangulate can be found in Figure 6 below:



```python
def triangulate(C1, pts1, C2, pts2):
    # Initialize variables and get point count
    n = pts1.shape
    P = np.zeros((n, 3))
    err = 0

    # Loop for every point to triangulate
    for i in range(n):
        # Get x and y for image 1 and image 2
        x_1, y_1 = pts1[i, 0], pts1[i, 1]
        x_2, y_2 = pts2[i, 0], pts2[i, 1]

        # Make A matrix and SVD
        A_1 = C1[2, :] * x_1 - C1[0, :]
        A_2 = C1[2, :] * y_1 - C1[1, :]
        A_3 = C2[2, :] * x_2 - C2[0, :]
        A_4 = C2[2, :] * y_2 - C2[1, :]
        A = np.vstack((A_1, A_2, A_3, A_4))

        _, _, V = np.linalg.svd(A)
        v = V.T
        res = v[:, -1]

        # Homogenize and project the points
        res_hom = res / res[-1]

        im1_2d = np.matmul(C1, res_hom.T)
        im2_2d = np.matmul(C2, res_hom.T)

        # Normalize and get all x, y points
        im1_2d = im1_2d / im1_2d[-1]
        im1_pts = im1_2d[0: 2]
        im2_2d = im2_2d / im2_2d[-1]
        im2_pts = im2_2d[0: 2]

        # Reproject and find triangulation/error
        im1_err = np.linalg.norm(im1_pts - pts1[i, :]) ** 2
        im2_err = np.linalg.norm(im2_pts - pts2[i, :]) ** 2

        P[i, :] = res_hom[0: 3]
        err = err + (im1_err + im2_err)

    return P, err
```

Figure 6: triangulate Function Implementation

The expression for the matrix A_i is simple to calculate and can be found below.
Image 1:

$$\begin{bmatrix} C_{11} \\ C_{12} \\ C_{13} \end{bmatrix} \begin{bmatrix} u_i \\ v_i \\ w_i \\ 1 \end{bmatrix} = \begin{bmatrix} x_{1i} \\ y_{1i} \\ 1 \end{bmatrix}$$

Image 2:

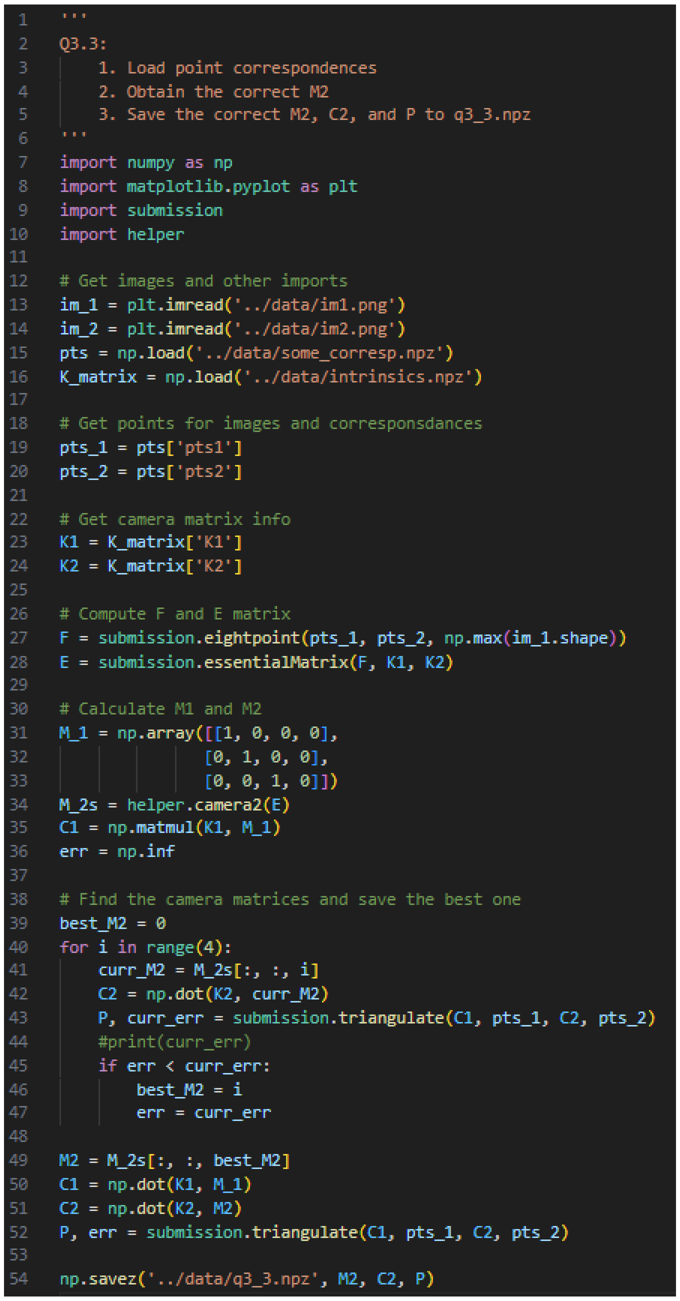
$$\begin{bmatrix} C_{21} \\ C_{22} \\ C_{23} \end{bmatrix} \begin{bmatrix} u_i \\ v_i \\ w_i \\ 1 \end{bmatrix} = \begin{bmatrix} x_{2i} \\ y_{2i} \\ 1 \end{bmatrix}$$

When multiplied to solve for A:

$$A_i = \begin{bmatrix} x_{1i}C_{13} - C_{11} \\ y_{1i}C_{13} - C_{12} \\ x_{2i}C_{23} - C_{21} \\ x_{2i}C_{23} - C_{21} \end{bmatrix}$$

3.3 Write a script findM2.py to obtain the correct M2 from M2s by testing the four solutions through triangulations. Use the correspondences from data/some corresp.npz.

My implementation for the function findM2 can be found in Figure 7 below:



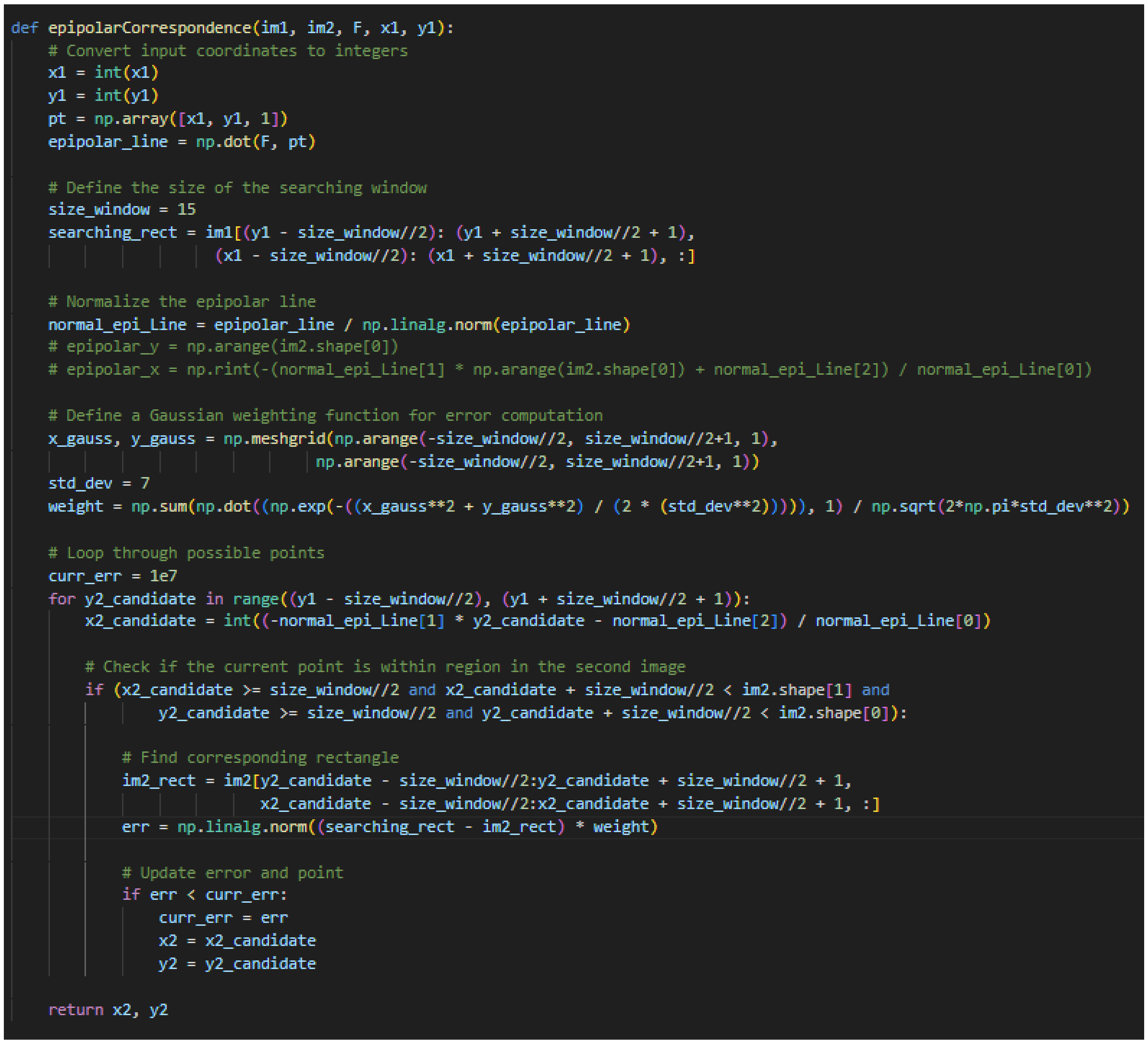
```python
'''
Q3.3:
    1. Load point correspondences
    2. Obtain the correct M2
    3. Save the correct M2, C2, and P to q3_3.npz
'''
import numpy as np
import matplotlib.pyplot as plt
import submission
import helper

# Get images and other imports
im_1 = plt.imread('../data/im1.png')
im_2 = plt.imread('../data/im2.png')
pts = np.load('../data/some_corresp.npz')
K_matrix = np.load('../data/intrinsics.npz')

# Get points for images and correspondances
pts_1 = pts['pts1']
pts_2 = pts['pts2']

# Get camera matrix info
K1 = K_matrix['K1']
K2 = K_matrix['K2']

# Compute F and E matrix
F = submission.eightpoint(pts_1, pts_2, np.max(im_1.shape))
E = submission.essentialMatrix(F, K1, K2)

# Calculate M1 and M2
M_1 = np.array([[1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0]])
M_2s = helper.camera2(E)
C1 = np.matmul(K1, M_1)
err = np.inf

# Find the camera matrices and save the best one
best_M2 = 0
for i in range(4):
    curr_M2 = M_2s[:, :, i]
    C2 = np.dot(K2, curr_M2)
    P, curr_err = submission.triangulate(C1, pts_1, C2, pts_2)
    #print(curr_err)
    if err < curr_err:
        best_M2 = i
        err = curr_err

M2 = M_2s[:, :, best_M2]
C1 = np.dot(K1, M_1)
C2 = np.dot(K2, M2)
P, err = submission.triangulate(C1, pts_1, C2, pts_2)

np.savez('../data/q3_3.npz', M2, C2, P)
```

Figure 7: findM2 Function Implementation

4 3D Visualization

4.1 Implement a function 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 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 (x1, y1) coordinates in im1 to various windows around possible matches in the im2 and returning the closest.

My implementation for the function epipolarCorrespondence can be found in Figure 8 below:



```python
def epipolarCorrespondence(im1, im2, F, x1, y1):
    # Convert input coordinates to integers
    x1 = int(x1)
    y1 = int(y1)
    pt = np.array([x1, y1, 1])
    epipolar_line = np.dot(F, pt)

    # Define the size of the searching window
    size_window = 15
    searching_rect = im1[(y1 - size_window//2): (y1 + size_window//2 + 1),
                      (x1 - size_window//2): (x1 + size_window//2 + 1), :]

    # Normalize the epipolar line
    normal_epi_Line = epipolar_line / np.linalg.norm(epipolar_line)
    # epipolar_y = np.arange(im2.shape[0])
    # epipolar_x = np.rint(-(normal_epi_Line[1] * np.arange(im2.shape[0]) + normal_epi_Line[2]) / normal_epi_Line[0])

    # Define a Gaussian weighting function for error computation
    x_gauss, y_gauss = np.meshgrid(np.arange(-size_window//2, size_window//2+1, 1),
                                 np.arange(-size_window//2, size_window//2+1, 1))
    std_dev = 7
    weight = np.sum(np.dot((np.exp(-((x_gauss**2 + y_gauss**2) / (2 * (std_dev**2))))), 1) / np.sqrt(2*np.pi*std_dev**2))

    # Loop through possible points
    curr_err = 1e7
    for y2_candidate in range((y1 - size_window//2), (y1 + size_window//2 + 1)):
        x2_candidate = int((-normal_epi_Line[1] * y2_candidate - normal_epi_Line[2]) / normal_epi_Line[0])

        # Check if the current point is within region in the second image
        if (x2_candidate >= size_window//2 and x2_candidate + size_window//2 < im2.shape[1] and
                y2_candidate >= size_window//2 and y2_candidate + size_window//2 < im2.shape[0]):

            # Find corresponding rectangle
            im2_rect = im2[y2_candidate - size_window//2:y2_candidate + size_window//2 + 1,
                           x2_candidate - size_window//2:x2_candidate + size_window//2 + 1, :]
            err = np.linalg.norm((searching_rect - im2_rect) * weight)

            # Update error and point
            if err < curr_err:
                curr_err = err
                x2 = x2_candidate
                y2 = y2_candidate

    return x2, y2
```

Figure 8: epipolarCorrespondence Function Implementation

The results from the epipolarMatchGUI correspondences can be found in Figure 9 below. One thing to note here is that most of the points correspond really well to the second image but for some corners that look different due to the camera rotation, the point does not line up exactly as seen in the case of the right most blue line. Other than that, the correspondences here can be seen to match decently well throughout.

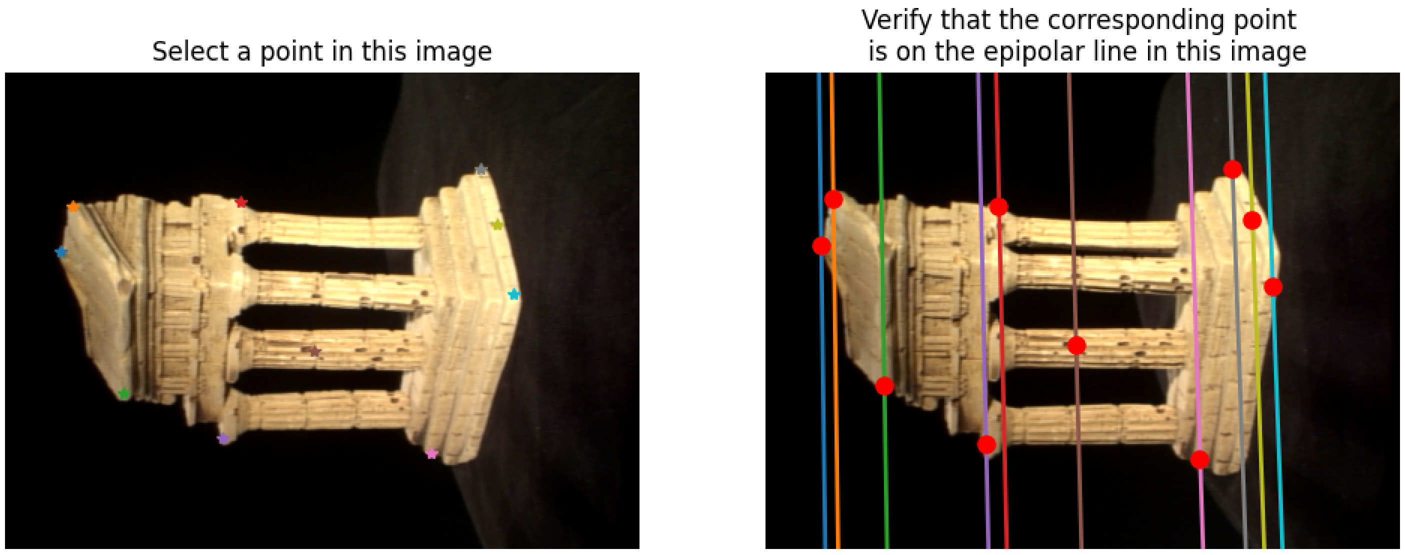
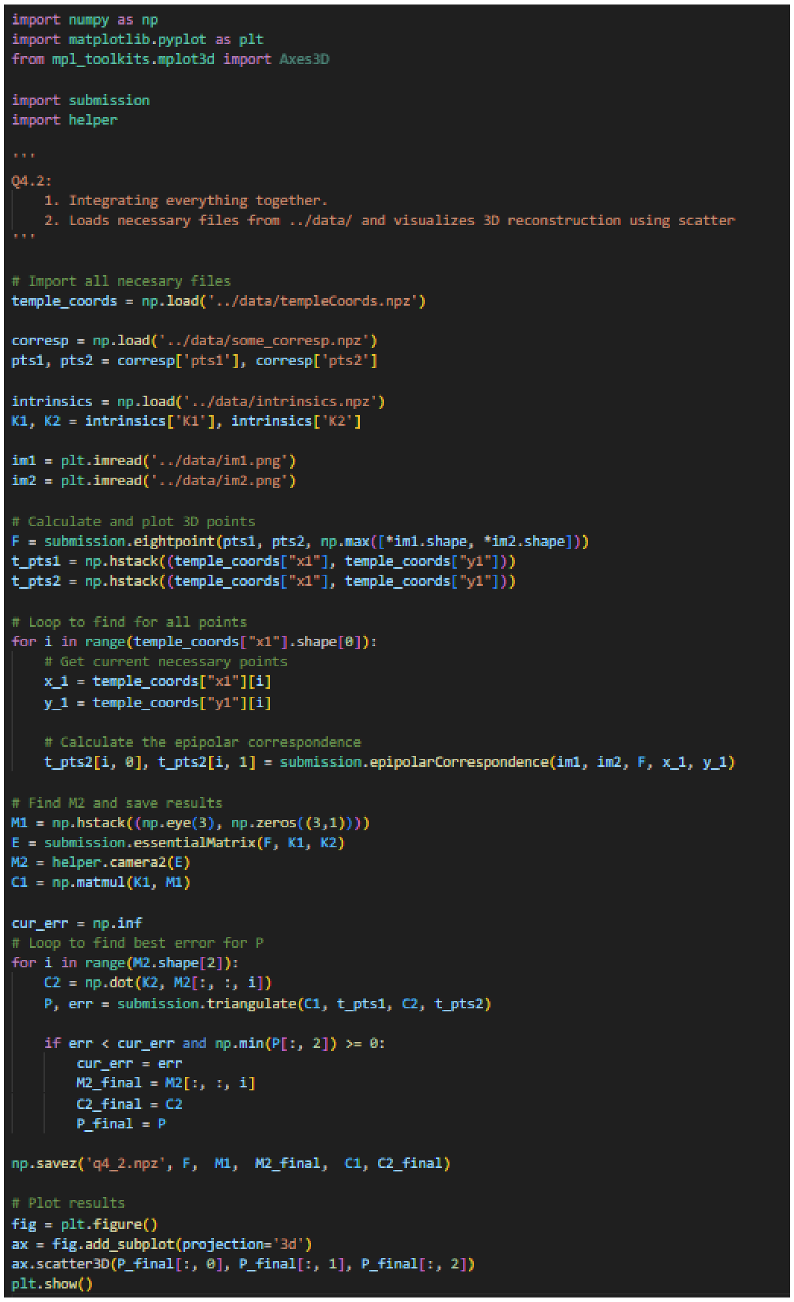

Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Figure 9: epipolarMatchGUI Results

4.2 Write a script visualize.py, which loads the necessary files from ../data/ to generate the 3D reconstruction using scatter function matplotlib. An example is shown in Figure 7. Output: Again, save the matrix F, matrices M1, M2, C1, C2 which you used to generate the screenshots to the file q4 2.npz.

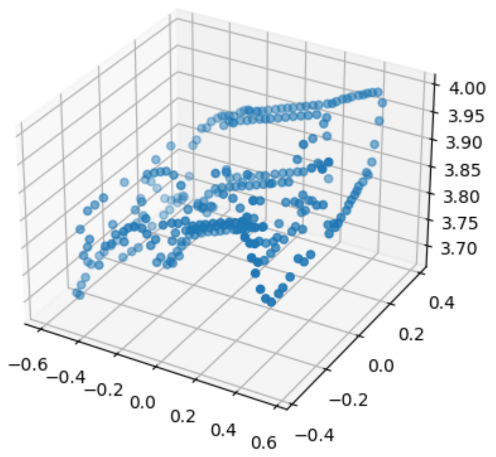
My implementation for the Python file visualize.py can be found in Figure 9 below:



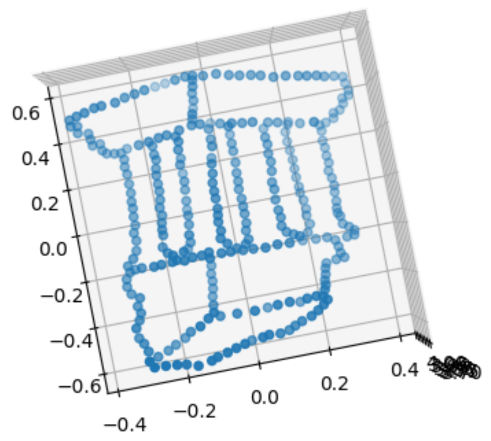
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import submission
import helper

'''
Q4.2:
    1. Integrating everything together.
    2. Loads necessary files from ../data/ and visualizes 3D reconstruction using scatter
'''

# Import all necesary files
temple_coords = np.load('../data/templeCoords.npz')

corresp = np.load('../data/some_corresp.npz')
pts1, pts2 = corresp['pts1'], corresp['pts2']

intrinsics = np.load('../data/intrinsics.npz')
K1, K2 = intrinsics['K1'], intrinsics['K2']

im1 = plt.imread('../data/im1.png')
im2 = plt.imread('../data/im2.png')

# Calculate and plot 3D points
F = submission.eightpoint(pts1, pts2, np.max([*im1.shape, *im2.shape]))
t_pts1 = np.hstack((temple_coords["x1"], temple_coords["y1"]))
t_pts2 = np.hstack((temple_coords["x1"], temple_coords["y1"]))

# Loop to find for all points
for i in range(temple_coords["x1"].shape[0]):
    # Get current necessary points
    x_1 = temple_coords["x1"][i]
    y_1 = temple_coords["y1"][i]

    # Calculate the epipolar correspondence
    t_pts2[i, 0], t_pts2[i, 1] = submission.epipolarCorrespondence(im1, im2, F, x_1, y_1)

# Find M2 and save results
M1 = np.hstack((np.eye(3), np.zeros((3,1))))
E = submission.essentialMatrix(F, K1, K2)
M2 = helper.camera2(E)
C1 = np.matmul(K1, M1)

cur_err = np.inf
# Loop to find best error for P
for i in range(M2.shape[2]):
    C2 = np.dot(K2, M2[:, :, i])
    P, err = submission.triangulate(C1, t_pts1, C2, t_pts2)

    if err < cur_err and np.min(P[:, 2]) >= 0:
        cur_err = err
        M2_final = M2[:, :, i]
        C2_final = C2
        P_final = P

np.savez('q4_2.npz', F,  M1,  M2_final,  C1, C2_final)

# Plot results
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter3D(P_final[:, 0], P_final[:, 1], P_final[:, 2])
plt.show()
```

Figure 10: Visualize File Implementation

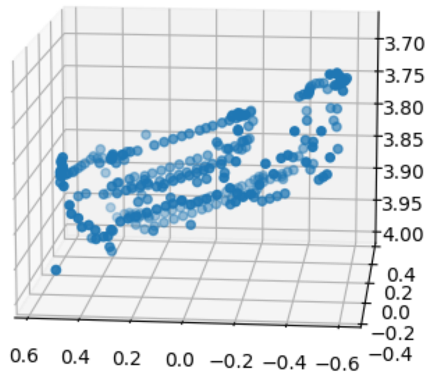
The results for my 3D visualization process cna be found in Figure 11 below.



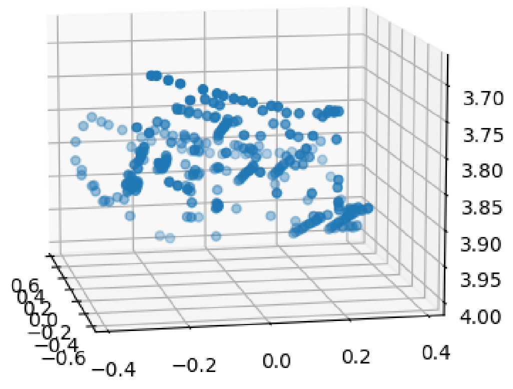
(a) Back Left View



(b) Top View



(c) Right View



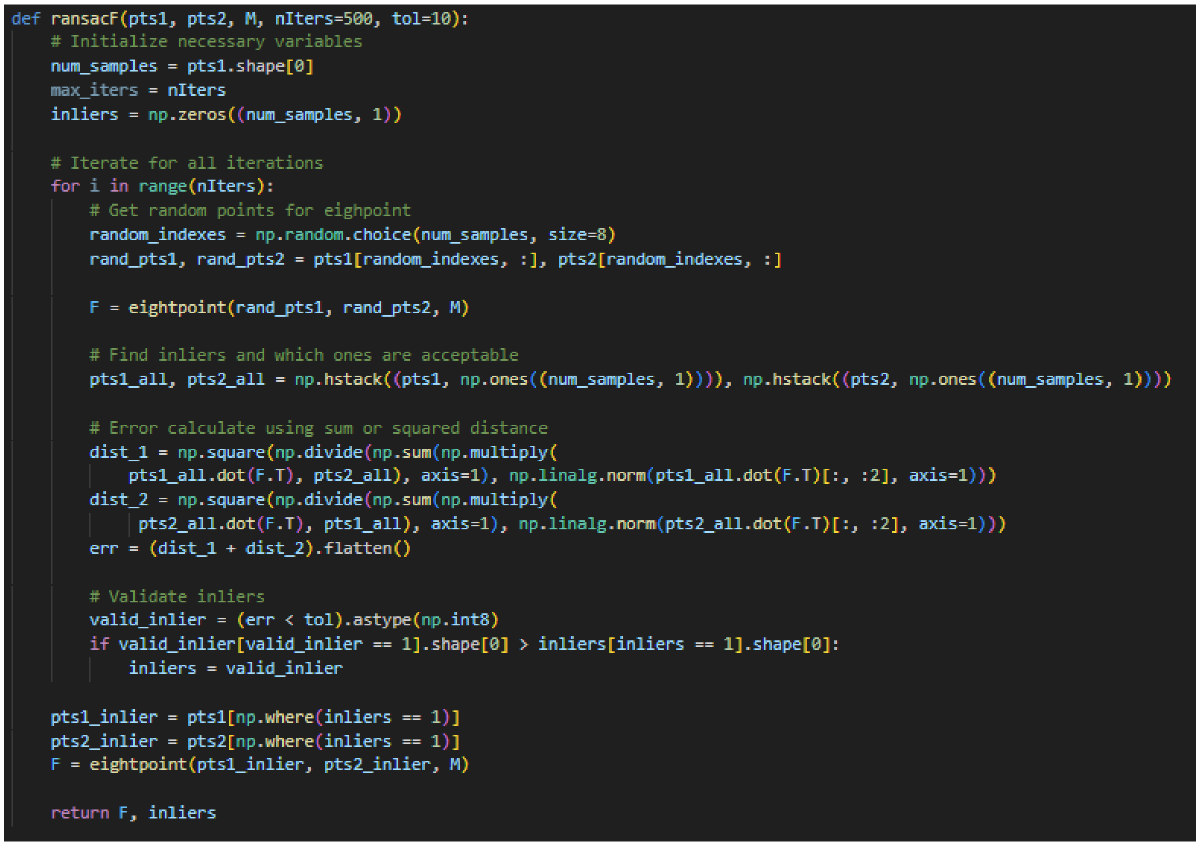
(d) Front Left View

Figure 11: Different 3D Views

5 Bundle Adjustment

5.1 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.

My implementation for the Python function ransacF can be found in Figure 12 below:



```python
def ransacF(pts1, pts2, M, nIters=500, tol=10):
    # Initialize necessary variables
    num_samples = pts1.shape[0]
    max_iters = nIters
    inliers = np.zeros((num_samples, 1))

    # Iterate for all iterations
    for i in range(nIters):
        # Get random points for eighpoint
        random_indexes = np.random.choice(num_samples, size=8)
        rand_pts1, rand_pts2 = pts1[random_indexes, :], pts2[random_indexes, :]

        F = eightpoint(rand_pts1, rand_pts2, M)

        # Find inliers and which ones are acceptable
        pts1_all, pts2_all = np.hstack((pts1, np.ones((num_samples, 1)))), np.hstack((pts2, np.ones((num_samples, 1))))

        # Error calculate using sum or squared distance
        dist_1 = np.square(np.divide(np.sum(np.multiply(
            pts1_all.dot(F.T), pts2_all), axis=1), np.linalg.norm(pts1_all.dot(F.T)[:, :2], axis=1)))
        dist_2 = np.square(np.divide(np.sum(np.multiply(
             pts2_all.dot(F.T), pts1_all), axis=1), np.linalg.norm(pts2_all.dot(F.T)[:, :2], axis=1)))
        err = (dist_1 + dist_2).flatten()

        # Validate inliers
        valid_inlier = (err < tol).astype(np.int8)
        if valid_inlier[valid_inlier == 1].shape[0] > inliers[inliers == 1].shape[0]:
            inliers = valid_inlier

    pts1_inlier = pts1[np.where(inliers == 1)]
    pts2_inlier = pts2[np.where(inliers == 1)]
    F = eightpoint(pts1_inlier, pts2_inlier, M)

    return F, inliers
```

Figure 12: RansacF Function Implementation

The error metrics that I opted to implement in this case was the sum of squared distance method. In this case, I found the distance of how far the point was relative to its epipolar line. From there, the sum of each squared distance was analyzed to find the error for the fundamental matrix.

In the ransacF function, the decision of inliers is based on the error, which quantifies the difference between observed points and their predicted locations using the fundamental matrix. This reprojection error is calculated for each pair, and points with errors below a defined threshold are considered inliers. The valid inliers array are then found as a binary mask to save which points meet the criteria for RANSAC iteration. The function repeats and updates the inliers, which contribute to a more robust estimate of the F matrix.

In terms of optimizing the ransacF function, there are only a few methods implemented to ensure that the code is optimized. The calculation of the error was simplified by using element-wise multiplication to reducing computational overhead. Also, the data types such as using dtype=np.int8 help reduce potential memory consumption.

The results of just implementing the original eightpoint can be found in Figure 13. I tested the RANSAC application twice with two different iteration counts as seen in Figures 14 anf 15. From images 13, 14, and 15, we see that adding RANSAC into our calculations made the epipolar line calculations much better than without. Along with that, we see that there is a specific threshold for the RANSAC to perform the best. With the noisy data, finding 8 points that linked to the correct number of inliers was too difficult to do under 100 iterations. The results from the 500 iteration test seem slightly better than the 100 iteration test, most likely due to the tolerance as well.

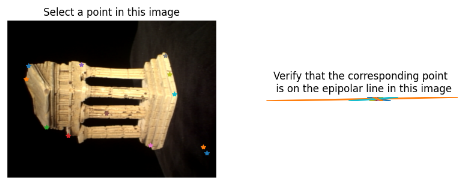

Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Figure 13: No RANSAC Results

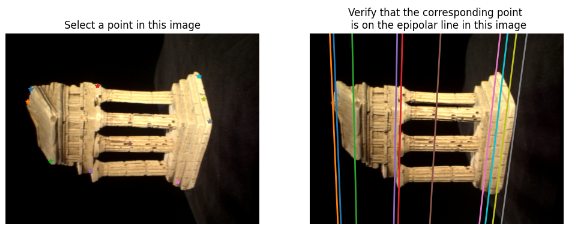

Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Figure 14: RANSAC 100 Iteration 0.8 Tolerance Results

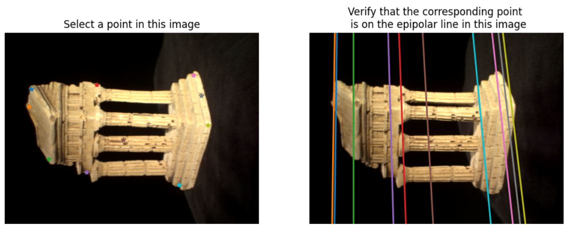
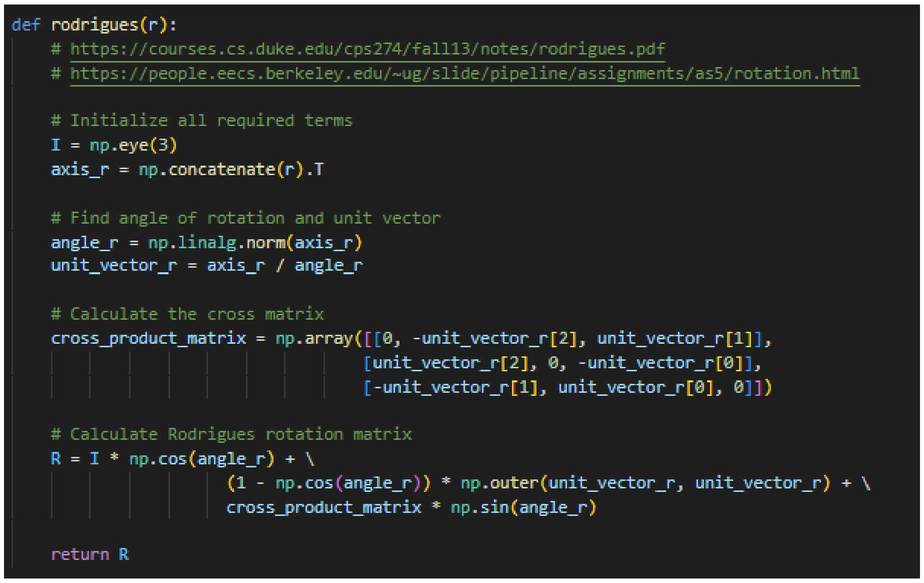

Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Figure 15: RANSAC 500 Iteration 1 Tolerance Results

5.2 Write a function that converts a Rodrigues vector r to a rotation matrix R R = rodrigues(r) as well as the inverse function that converts a rotation matrix R to a Rodrigues vector r r = invRodrigues(R)

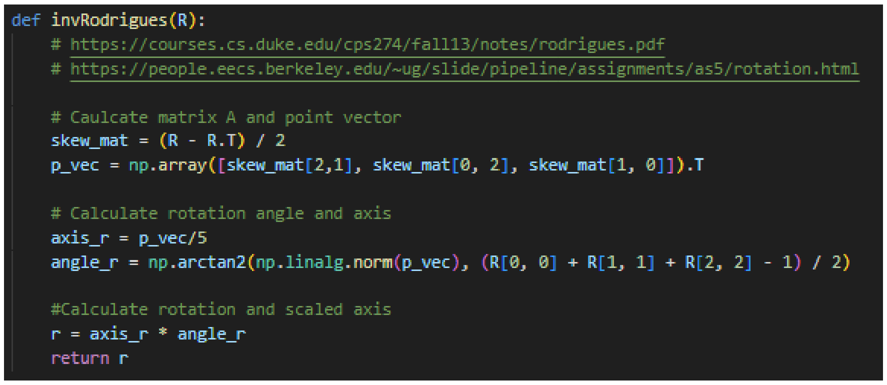
My implementation for the function rodrigues can be found in Figure 16 below:



```python
def rodrigues(r):
    # https://courses.cs.duke.edu/cps274/fall13/notes/rodrigues.pdf
    # https://people.eecs.berkeley.edu/~ug/slide/pipeline/assignments/as5/rotation.html

    # Initialize all required terms
    I = np.eye(3)
    axis_r = np.concatenate(r).T

    # Find angle of rotation and unit vector
    angle_r = np.linalg.norm(axis_r)
    unit_vector_r = axis_r / angle_r

    # Calculate the cross matrix
    cross_product_matrix = np.array([[0, -unit_vector_r[2], unit_vector_r[1]],
                                    [unit_vector_r[2], 0, -unit_vector_r[0]],
                                    [-unit_vector_r[1], unit_vector_r[0], 0]])

    # Calculate Rodrigues rotation matrix
    R = I * np.cos(angle_r) + \
                  (1 - np.cos(angle_r)) * np.outer(unit_vector_r, unit_vector_r) + \
                  cross_product_matrix * np.sin(angle_r)

    return R
```

Figure 16: Rodrigues Function Implementation

My implementation for the function inverse rodrigues can be found in Figure 17 below:



```python
def invRodrigues(R):
    # https://courses.cs.duke.edu/cps274/fall13/notes/rodrigues.pdf
    # https://people.eecs.berkeley.edu/~ug/slide/pipeline/assignments/as5/rotation.html

    # Caulcate matrix A and point vector
    skew_mat = (R - R.T) / 2
    p_vec = np.array([skew_mat[2,1], skew_mat[0, 2], skew_mat[1, 0]]).T

    # Calculate rotation angle and axis
    axis_r = p_vec/5
    angle_r = np.arctan2(np.linalg.norm(p_vec), (R[0, 0] + R[1, 1] + R[2, 2] - 1) / 2)

    #Calculate rotation and scaled axis
    r = axis_r * angle_r
    return r
```

Figure 17: Inverse Rodrigues Function Implementation