After your finish the assignment, remember to run all cells and save the note book to your local machine as a PDF for gradescope submission by pressing Ctrl-P or Cmd-P. Make sure images are not split between pages; insert Text blocks to make sure this is the case before printing to PDF!

List your collaborators here:

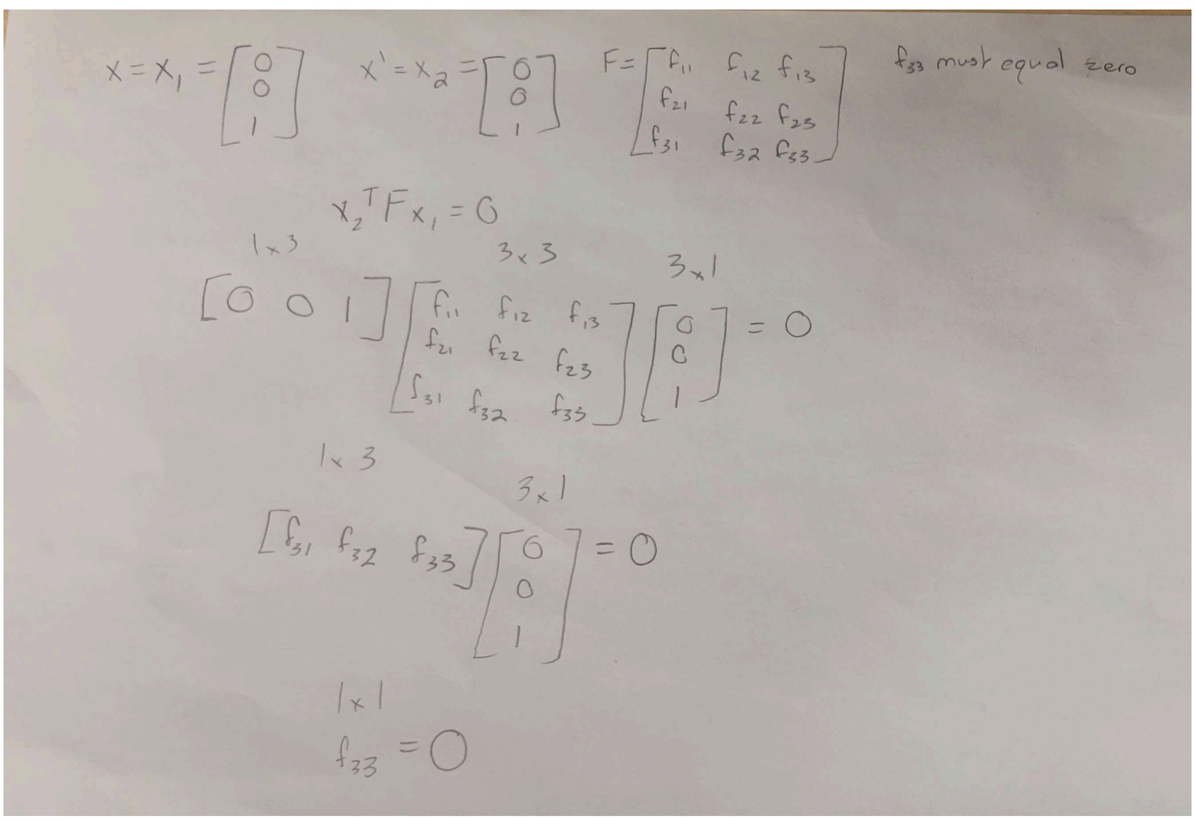
16720 HW 4: 3D Reconstruction

Problem 1: Theory

1.1

See pdf for the question.

===== your answer here for 1.1! =====



$x = x_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ $x' = x_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ $F = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}$ f_{33} must equal zero

$$x_2^T F x_1 = 0$$

$$\underset{1\times 3}{\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}} \underset{3\times 3}{\begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}} \underset{3\times 1}{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}} = 0$$

$$\underset{1\times 3}{\begin{bmatrix} f_{31} & f_{32} & f_{33} \end{bmatrix}} \underset{3\times 1}{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}} = 0$$

$$\underset{1\times 1}{f_{33}} = 0$$

===== end of your answer for 1.1 =====

1.2

See pdf for the question.

===== your answer here for 1.2! =====

$$R_{rel} = R_{i+1} R_i^T \qquad t_{rel} = t_{i+1} - t_i$$

$$E = \hat{T} R_{rel}$$

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

===== end of your answer for 1.2 =====

Coding

Initialization

Run the following code, which imports the modules you'll need and defines helper functions you may need to use later in your implementations.

In [ ]:
```
import os
import numpy as np
import scipy
import scipy.optimize
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
#from cv2 import cv2_imshow
import cv2


connections_3d = [[0,1], [1,3], [2,3], [2,0], [4,5], [6,7], [8,9], [9,11], [10,11], [1
```

```
                  [1,5], [5,9], [2,6], [6,10], [3,7], [7,11]]
color_links = [(255,0,0),(255,0,0),(255,0,0),(255,0,0),(0,0,255),(255,0,255),(0,255,0)
colors = ['blue','blue','blue','blue','red','magenta','green','green','green','green',


def visualize_keypoints(image, pts, Threshold=100):
    '''
    This function visualizes the 2d keypoint pairs in connections_3d
    (as define above) whose match score lies above a given Threshold
    in an OpenCV GUI frame, against an image background.

    :param image: image as a numpy array, of shape (height, width, 3) where 3 is the n
    :param pts: np.array of shape (num_points, 3)
    '''
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    for i in range(12):
        cx, cy = pts[i][0:2]
        if pts[i][2]>Threshold:
            cv2.circle(image,(int(cx),int(cy)),5,(0,255,255),5)

    for i in range(len(connections_3d)):
        idx0, idx1 = connections_3d[i]
        if pts[idx0][2]>Threshold and pts[idx1][2]>Threshold:
            x0, y0 = pts[idx0][0:2]
            x1, y1 = pts[idx1][0:2]
            cv2.line(image, (int(x0), int(y0)), (int(x1), int(y1)), color_links[i], 2)

    cv2_imshow(image)

    return image


def plot_3d_keypoint(pts_3d):
    '''
    this function visualizes 3d keypoints on a matplotlib 3d axes

    :param pts_3d: np.array of shape (num_points, 3)
    '''
    fig = plt.figure()
    num_points = pts_3d.shape[0]
    ax = fig.add_subplot(111, projection='3d')
    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])
    np.set_printoptions(threshold=1e6, suppress=True)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()


def calc_epi_error(pts1_homo, pts2_homo, F):
    '''
    Helper function to calcualte the sum of squared distance between the
    corresponding points and the estimated epipolar lines.

    pts1_homo \dot F.T \dot pts2_homo = 0
```

```
    :param pts1_homo: of shape (num_points, 3); in homogeneous coordinates, not normal
    :param pts2_homo: same specification as to pts1_homo.
    :param F: Fundamental matrix
    '''

    line1s = pts1_homo.dot(F.T)
    dist1 = np.square(np.divide(np.sum(np.multiply(
        line1s, pts2_homo), axis=1), np.linalg.norm(line1s[:, :2], axis=1)))

    line2s = pts2_homo.dot(F)
    dist2 = np.square(np.divide(np.sum(np.multiply(
        line2s, pts1_homo), axis=1), np.linalg.norm(line2s[:, :2], axis=1)))

    ress = (dist1 + dist2).flatten()
    return ress


def toHomogenous(pts):
    """
    Adds a stack of ones at the end, to turn a set of points into a set of
    homogeneous points.

    :params pts: in shape (num_points, 2).
    """
    return np.vstack([pts[:,0],pts[:,1],np.ones(pts.shape[0])]).T.copy()


def _epipoles(E):
    """
    gets the epipoles from the Essential Matrix.

    :params E: Essential matrix.
    """
    U, S, V = np.linalg.svd(E)
    e1 = V[-1, :]
    U, S, V = np.linalg.svd(E.T)
    e2 = V[-1, :]
    return e1, e2


def displayEpipolarF(I1, I2, F, points):
    """
    GUI interface you may use to help you verify your calculated fundamental
    matrix F. Select a point I1 in one view, and it should correctly correspond
    to the displayed point in the second view.
    """
    e1, e2 = _epipoles(F)

    sy, sx, _ = I2.shape

    f, [ax1, ax2] = plt.subplots(1, 2, figsize=(12, 9))
    ax1.imshow(I1)
    ax1.set_title('The point you selected:')
    ax2.imshow(I2)
    ax2.set_title('Verify that the corresponding point \n is on the epipolar line in t

    plt.sca(ax1)

    colors = ['r','g','b','y','m','k']
```

```
    for i, out in enumerate(points):
      x, y = out #[0]

      xc = x
      yc = y
      v = np.array([xc, yc, 1])
      l = F.dot(v)
      s = np.sqrt(l[0]**2+l[1]**2)

      if s==0:
          print('Zero line vector in displayEpipolar')

      l = l/s

      if l[0] != 0:
          ye = sy-1
          ys = 0
          xe = -(l[1] * ye + l[2])/l[0]
          xs = -(l[1] * ys + l[2])/l[0]
      else:
          xe = sx-1
          xs = 0
          ye = -(l[0] * xe + l[2])/l[1]
          ys = -(l[0] * xs + l[2])/l[1]

      # plt.plot(x,y, '*', 'MarkerSize', 6, 'LineWidth', 2);
      ax1.plot(x, y, '*', markersize=6, linewidth=2, color=colors[i%len(colors)])
      ax2.plot([xs, xe], [ys, ye], linewidth=2, color=colors[i%len(colors)])
    plt.draw()


def _singularize(F):
    U, S, V = np.linalg.svd(F)
    S[-1] = 0
    F = U.dot(np.diag(S).dot(V))
    return F

def _objective_F(f, pts1, pts2):
    F = _singularize(f.reshape([3, 3]))
    num_points = pts1.shape[0]
    hpts1 = np.concatenate([pts1, np.ones([num_points, 1])], axis=1)
    hpts2 = np.concatenate([pts2, np.ones([num_points, 1])], axis=1)
    Fp1 = F.dot(hpts1.T)
    FTp2 = F.T.dot(hpts2.T)

    r = 0
    for fp1, fp2, hp2 in zip(Fp1.T, FTp2.T, hpts2):
        r += (hp2.dot(fp1))**2 * (1/(fp1[0]**2 + fp1[1]**2) + 1/(fp2[0]**2 + fp2[1]**2
    return r

def refineF(F, pts1, pts2):
    f = scipy.optimize.fmin_powell(
        lambda x: _objective_F(x, pts1, pts2), F.reshape([-1]),
        maxiter=100000,
        maxfun=10000,
        disp=False
    )
    return _singularize(f.reshape([3, 3]))
```

```
# Used in 4.2 Epipolar Correspondence
def epipolarMatchGUI(I1, I2, F, points, epipolarCorrespondence):
    e1, e2 = _epipoles(F)

    sy, sx, _ = I2.shape

    f, [ax1, ax2] = plt.subplots(1, 2, figsize=(12, 9))
    ax1.imshow(I1)
    ax1.set_title('The point you selected:')
    ax2.imshow(I2)
    ax2.set_title('Verify that the corresponding point \n is on the epipolar line in t

    plt.sca(ax1)

    colors = ['r','g','b','y','m','k']

    for i, out in enumerate(points):
      x, y = out

      xc = int(x)
      yc = int(y)
      v = np.array([xc, yc, 1])
      l = F.dot(v)
      s = np.sqrt(l[0]**2+l[1]**2)

      if s==0:
          print('Zero line vector in displayEpipolar')

      l = l/s

      if l[0] != 0:
          ye = sy-1
          ys = 0
          xe = -(l[1] * ye + l[2])/l[0]
          xs = -(l[1] * ys + l[2])/l[0]
      else:
          xe = sx-1
          xs = 0
          ye = -(l[0] * xe + l[2])/l[1]
          ys = -(l[0] * xs + l[2])/l[1]

      ax1.plot(x, y, '*', markersize=6, linewidth=2, color=colors[i%len(colors)])
      ax2.plot([xs, xe], [ys, ye], linewidth=2, color=colors[i%len(colors)])

      # draw points
      x2, y2 = epipolarCorrespondence(I1, I2, F, xc, yc)
      ax2.plot(x2, y2, 'ro', markersize=8, linewidth=2)
      plt.draw()
```

Set up data

In this section, we will download the test case image views, camera intrinsics, and point correnspondences, which you will use for testing your implementations.

In [ ]:
```
# if not os.path.exists('data'):
#   !wget https://www.andrew.cmu.edu/user/eweng/data.zip -O data.zip
```

```
#   !unzip -qq "data.zip"
#   print("downloaded and unzipped data")
```

Problem 2: Estimating the Fundamental Matrix with the Eight-point Algorithm

In this part, implement the 8-point algorithm you learned in class, which estimates the fundamental matrix from corresponding points in two images.

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

  HINTS:
  (1) Normalize the input pts1 and pts2 using the matrix T.
  (2) Setup the eight point algorithm's equation.
  (3) Solve for the least square solution using SVD.
  (4) Use the function `_singularize` (provided in the helper functions above) to enfo
  (5) Use the function `refineF` (provided in the helper functions above) to refine th
      (Remember to use the normalized points instead of the original points)
  (6) Unscale the fundamental matrix by the lower right corner element
  '''



  F = None
  N = pts1.shape[0]

  # ===== your code here! =====
  # Normalize points
  # pts1 = pts1
  # pts2 = pts2
  pts1 = pts1/M
  pts2 = pts2/M
  rows_A = 9
  t_mat = np.eye(3)
  t_mat[0,0] = 1/M
  t_mat[1,1] = 1/M

  # Set up algorithims equation
  A = np.zeros((N,rows_A))
  x1 = pts2[:,0]
  y1 = pts2[:,1]
  x2 = pts1[:,0]
  y2 = pts1[:,1]
  A[:,0] = x1*x2
  A[:,1] = x1*y2
  A[:,2] = x1
  A[:,3] = y1*x2
  A[:,4] = y1*y2
  A[:,5] = y1
  A[:,6] = x2
  A[:,7] = y2
```

```
    A[:,8] = 1.0 #(np.zeros_like(x1) + 1.0)

    # b = np.zeros((N,1))
    AtA = A.T @ A

    # Solve for solution using SVD
    u, s, vt = np.linalg.svd(A)
    eigVal, eigVec = np.linalg.eig(AtA)
    zero_singular_value_indices = np.argmin(eigVal)
    x = vt[-1]
    F = np.reshape(x,(3,3))

    # Singularize F
    F = _singularize(F)

    # Refine F
    F = refineF(F,pts1,pts2)

    # Unscale F
    F = t_mat.T@F@t_mat
    F = F / F[2,2]

    # ==== end of code ====

    return F
```

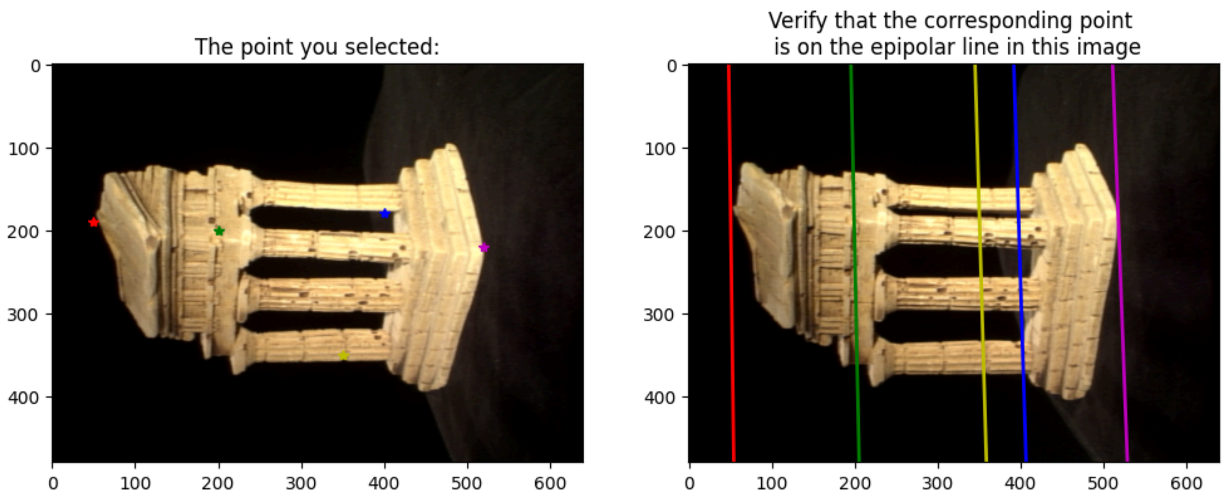
Run this code to test your implementation of the 8-point algorithm. Your code should pass all the assert statements at the end.

In [ ]:
```
DATA_PARENT_FOLDER_DIR = 'data\\data'
DATA_PARENT_DIR = os.getcwd()
DATA_PARENT_DIR = os.path.join(DATA_PARENT_DIR,DATA_PARENT_FOLDER_DIR)
HW4_SUBDIR = ''
DATA_DIR = os.path.join(DATA_PARENT_DIR, HW4_SUBDIR)

correspondence = np.load(os.path.join(DATA_DIR,'some_corresp.npz')) # Loading correspo
intrinsics = np.load(os.path.join(DATA_DIR,'intrinsics.npz')) # Loading the intrinscis
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread(os.path.join(DATA_DIR,'im1.png'))
im2 = plt.imread(os.path.join(DATA_DIR,'im2.png'))

Mval = np.max([*im1.shape, *im2.shape])
F = eightpoint(pts1, pts2, M=Mval)
print(f'recovered F:\n{F.round(4)}')

# Simple Tests to verify your implementation:
pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)
# pts1_homogenous[:,0:2] = pts1_homogenous[:,0:2]/Mval
# pts2_homogenous[:,0:2] = pts2_homogenous[:,0:2]/Mval

assert F.shape == (3, 3), "F is wrong shape"
assert F[2, 2] == 1, "F_33 != 1"
assert np.linalg.matrix_rank(F) == 2, "F should have rank 2"
assert np.mean(calc_epi_error(pts1_homogenous, pts2_homogenous, F)) < 1, "F error is t
```

```
recovered F:
[[-0.      0.     -0.2519]
 [ 0.     -0.      0.0026]
 [ 0.2422 -0.0068  1.    ]]
```

The following tool may help you debug. You may specify a point in im1, and view the corresponding epipolar line in im2 based on the F you found. In your submission, make sure you include the debug picture below, with at least five epipolar point-line correspondences taht show that your calculation of F is correct.

In [ ]:
```
# the points in im1, whose correnponding epipolar line in im2 you'd like to verify
point = [(50,190),(200, 200), (400,180), (350,350), (520, 220)]
# feel free to change these point, to verify different point correspondences
displayEpipolarF(im1, im2, F, point)
```



Problem 3: Metric Reconstruction

3.1 Essential Matrix

In [ ]:
```
def essentialMatrix(F, K1, K2):
  '''
  Q3.1: Compute the essential matrix E.
    Input:  F, fundamental matrix
            K1, internal camera calibration matrix of camera 1
            K2, internal camera calibration matrix of camera 2
    Output: E, the essential matrix
  '''

  # ----- TODO -----
  ### BEGIN SOLUTION
  E = K2.T @ F @ K1
  E = E / E[2,2]
  ### END SOLUTION
  return E
```

Run the following code to check your implementation.

In [ ]:
```
correspondence = np.load(os.path.join(DATA_DIR,'some_corresp.npz')) # Loading correspo
intrinsics = np.load(os.path.join(DATA_DIR,'intrinsics.npz')) # Loading the intrinscis
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread(os.path.join(DATA_DIR,'im1.png'))
im2 = plt.imread(os.path.join(DATA_DIR,'im2.png'))

F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
E = essentialMatrix(F, K1, K2)
print(f'recovered E:\n{E.round(4)}')

# Simple Tests to verify your implementation:
assert(E[2, 2] == 1)
assert(np.linalg.matrix_rank(E) == 2)
```

```
recovered E:
[[-3.3716000e+00  4.5661580e+02 -2.4738947e+03]
 [ 1.9760420e+02 -1.0290300e+01  6.4396600e+01]
 [ 2.4807427e+03  1.9856400e+01  1.0000000e+00]]
```

3.2 Triangulation

In [ ]:
```
def triangulate(C1, pts1, C2, pts2):
    '''
    Q3.2: Triangulate a set of 2D coordinates in the image to a set of 3D points.
    Input:  C1, the 3x4 camera matrix
            pts1, the Nx2 matrix with the 2D image coordinates per row
            C2, the 3x4 camera matrix
            pts2, the Nx2 matrix with the 2D image coordinates per row
    Output: P, the Nx3 matrix with the corresponding 3D points per row
            err, the reprojection error.

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

    # ----- TODO -----
    ### BEGIN SOLUTION
    N = pts1.shape[0]

    p1_t = C1[0,:].reshape((1,4))
    p2_t = C1[1,:].reshape((1,4))
    p3_t = C1[2,:].reshape((1,4))
    p1p_t = C2[0,:].reshape((1,4))
    p2p_t = C2[1,:].reshape((1,4))
    p3p_t = C2[2,:].reshape((1,4))

    x = pts1[:,0].reshape((N,1))
    y = pts1[:,1].reshape((N,1))
    xp = pts2[:,0].reshape((N,1))
    yp = pts2[:,1].reshape((N,1))
```

```
  A = np.zeros((4,4))
  P_homog = np.zeros((N,4))
  P = np.zeros((N,3))

  # Solve for 3D homogenous points
  for i in range(N):
    A[0,:] = y[i,0]*p3_t-p2_t
    A[1,:] = p1_t-x[i,0]*p3_t
    A[2,:] = yp[i,0]*p3p_t-p2p_t
    A[3,:] = p1p_t-xp[i,0]*p3p_t
    AtA = A.T @ A
    u, s, vt = np.linalg.svd(A)
    eigVal, eigVec = np.linalg.eig(AtA)
    zero_singular_value_indices = np.argmin(eigVal)
    sol = vt[-1]
    P_homog[i,:] = np.reshape(sol,(1,4))

  # Find the equivalent 3D non homgeneous points through w' scaling
  P[:,0] = P_homog[:,0] / P_homog[:,3]
  P[:,1] = P_homog[:,1] / P_homog[:,3]
  P[:,2] = P_homog[:,2] / P_homog[:,3]

  proj1_homog = np.dot(C1,P_homog.T).T
  proj2_homog = np.dot(C2,P_homog.T).T

  proj1 = np.zeros((N,2))
  proj2 = np.zeros((N,2))
  proj1[:,0] = proj1_homog[:,0] / proj1_homog[:,2]
  proj1[:,1] = proj1_homog[:,1] / proj1_homog[:,2]
  proj2[:,0] = proj2_homog[:,0] / proj2_homog[:,2]
  proj2[:,1] = proj2_homog[:,1] / proj2_homog[:,2]

  err1 = np.square(np.linalg.norm((proj1-pts1),ord=2,axis=1,keepdims=True))
  err2 = np.square(np.linalg.norm((proj2-pts2),ord=2,axis=1,keepdims=True))
  err = err1 + err2
  err = np.sum(err,axis=0)
  ### END SOLUTION

  return P, err
```

3.3 Find M2

In [ ]:

```
def camera2(E):
  """helper function to find the 4 possibile M2 matrices"""
  U,S,V = np.linalg.svd(E)
  m = S[:2].mean()
  E = U.dot(np.array([[m,0,0], [0,m,0], [0,0,0]])).dot(V)
  U,S,V = np.linalg.svd(E)
  W = np.array([[0,-1,0], [1,0,0], [0,0,1]])

  if np.linalg.det(U.dot(W).dot(V))<0:
      W = -W

  M2s = np.zeros([3,4,4])
  M2s[:,:,0] = np.concatenate([U.dot(W).dot(V), U[:,2].reshape([-1, 1])/abs(U[:,2]).ma
  M2s[:,:,1] = np.concatenate([U.dot(W).dot(V), -U[:,2].reshape([-1, 1])/abs(U[:,2]).m
  M2s[:,:,2] = np.concatenate([U.dot(W.T).dot(V), U[:,2].reshape([-1, 1])/abs(U[:,2]).
  M2s[:,:,3] = np.concatenate([U.dot(W.T).dot(V), -U[:,2].reshape([-1, 1])/abs(U[:,2])
```

```
  return M2s


def findM2(F, pts1, pts2, intrinsics):
    '''
    Q3.3: 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) K

    ***
    Hints:
    (1) Loop through the 'M2s' and use triangulate to calculate the 3D points and proj
        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 mat

    '''


    K1, K2 = intrinsics['K1'], intrinsics['K2']

    # ----- TODO -----
    ### BEGIN SOLUTION
    N = pts1.shape[0]
    E = essentialMatrix(F, K1, K2)
    M2s = camera2(E)

    M1 = np.hstack((np.identity(3), np.zeros(3)[:,np.newaxis]))
    C1 = K1.dot(M1)

    pts1_homog = 0

    for i in range(4):
      # Check if points in front of camera
      M2_now = M2s[:,:,i]
      C2_now = K2 @ M2_now

      P_now, err_now = triangulate(C1,pts1,C2_now,pts2)
      condition = (P_now[:,2] > 0) # Z coordinates are all positive
      if (condition.all()):
        M2 = M2_now
        C2 = C2_now
        P = P_now

    ### END SOLUTION

    return M2, C2, P
```

Run the following code to check your implementation of triangulation and findM2.

In [ ]:

```
correspondence = np.load(os.path.join(DATA_DIR,'some_corresp.npz')) # Loading correspo
intrinsics = np.load(os.path.join(DATA_DIR,'intrinsics.npz')) # Loading the intrinscis
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread(os.path.join(DATA_DIR,'im1.png'))
im2 = plt.imread(os.path.join(DATA_DIR,'im2.png'))
```

```
F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))

M2, C2, P = findM2(F, pts1, pts2, intrinsics)

# Simple Tests to verify your implementation:
M1 = np.hstack((np.identity(3), np.zeros(3)[:,np.newaxis]))
C1 = K1.dot(M1)
P_test, err = triangulate(C1, pts1, C2, pts2)
print(err)
assert(err < 500)
```

```
[351.89796623]
```

Problem 4: 3D Visualization

In [ ]:

```
def epipolarCorrespondence(im1, im2, F, x1, y1):
  '''
  Q4.1: 3D visualization of the temple images.
  Input:  im1, the first image
          im2, the second image
          F, the fundamental matrix
          x1, x-coordinates of a pixel on im1
          y1, y-coordinates of a pixel on im1
  Output: x2, x-coordinates of the pixel on im2
          y2, y-coordinates of the pixel on im2

  Hints:
  (1) Given input [x1, x2], use the fundamental matrix to recover the corresponding ep
  (2) Search along this line to check nearby pixel intensity (you can define a search
      find the best matches
  (3) Use gaussian weighting to weight the pixel similarity

  '''
  # ----- TODO -----
  # YOUR CODE HERE
  p1_homo2d = np.hstack([x1,y1,1.0])
  line = F @ p1_homo2d
  # print(line)
  a = line[0]
  b = line[1]
  c = line[2] # ax + by + c = 0
  # print(im2.shape)
  xmax = im2.shape[1] # 640
  ymax = im2.shape[0] # 480
  bestX = -1
  bestY = -1
  window_x = 10
  window_y = 10
  pm_x = int(window_x/2)
  pm_y = int(window_y/2)
  patch1 = im1[y1-pm_y:y1+pm_y,x1-pm_x:x1+pm_x,:]
  firstTime = 1
  lowestNorm = -1

  for i in range(pm_y,ymax-pm_y):
    pixelX = int(-(c + b*i) /a)
    pixelY = int(i)
    if (pixelX > 0) and (pixelX < xmax) and (pixelY > 0) and (pixelY < ymax):
```

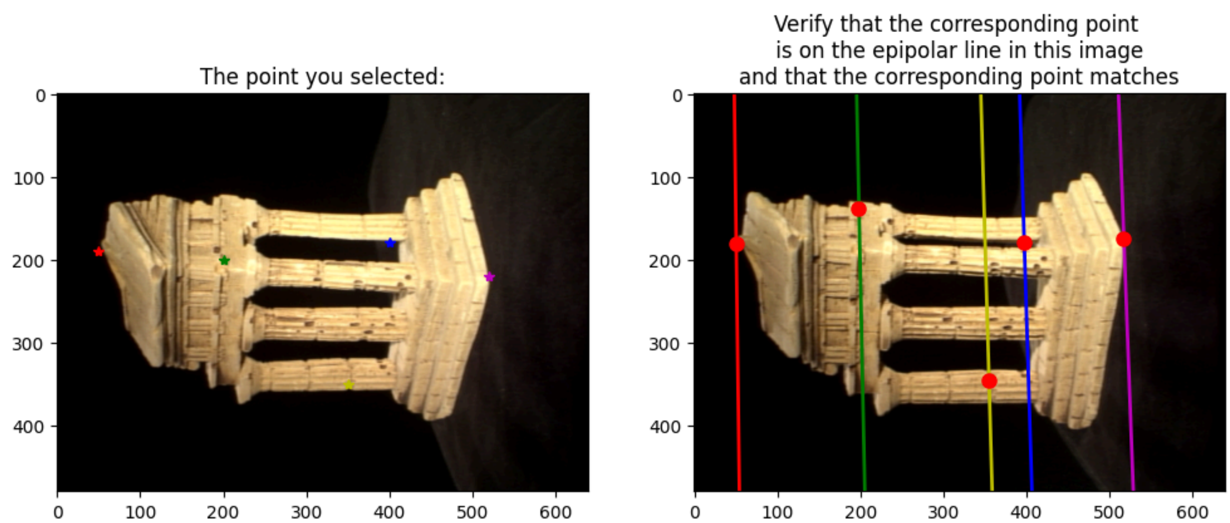
```
            patch2 = im2[pixelY-pm_y:pixelY+pm_y,pixelX-pm_x:pixelX+pm_x,:]
            norm = np.linalg.norm(patch1 - patch2).sum()/(window_x*window_y)
            if (firstTime == 1) or (norm < lowestNorm):
              bestX = pixelX
              bestY = pixelY
              lowestNorm = norm
              if (firstTime == 1):
                firstTime = 0

    x2 = bestX
    y2 = bestY
    # END YOUR CODE
    return x2, y2
```

Run the following code to check your implementation.

In [ ]:
```
correspondence = np.load(os.path.join(DATA_DIR,'some_corresp.npz')) # Loading correspo
intrinsics = np.load(os.path.join(DATA_DIR,'intrinsics.npz')) # Loading the intrinscis
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread(os.path.join(DATA_DIR,'im1.png'))
im2 = plt.imread(os.path.join(DATA_DIR,'im2.png'))

F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))

# Simple Tests to verify your implementation:
x2, y2 = epipolarCorrespondence(im1, im2, F, 119, 217)
assert(np.linalg.norm(np.array([x2, y2]) - np.array([118, 181])) < 10)
```

Use the below tool to debug your code.

In [ ]:
```
# the points in im1 whose correnponding epipolar line in im2 you'd like to verify
points = [(50,190), (200, 200), (400,180), (350,350), (520, 220)]
# feel free to change these points to verify different point correspondences
epipolarMatchGUI(im1, im2, F, points, epipolarCorrespondence)
```



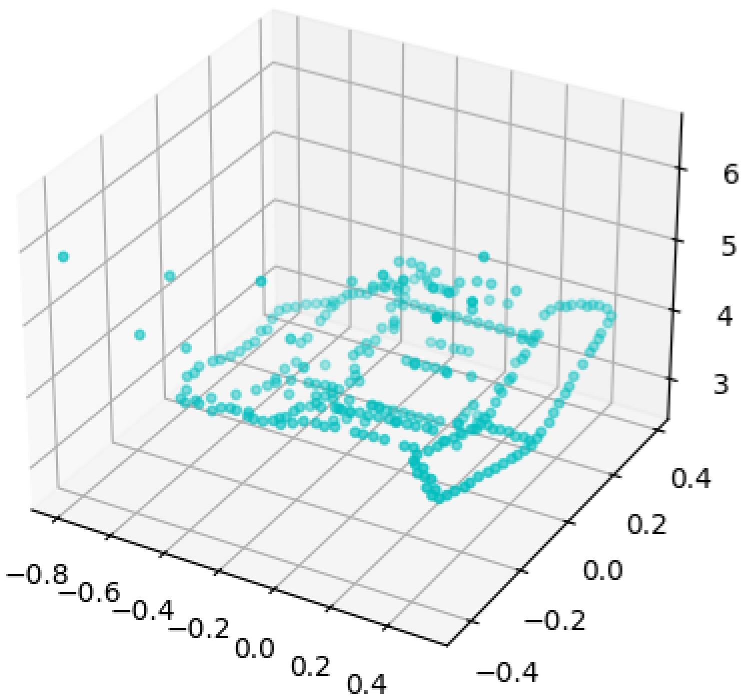
4.2 Temple Visualization

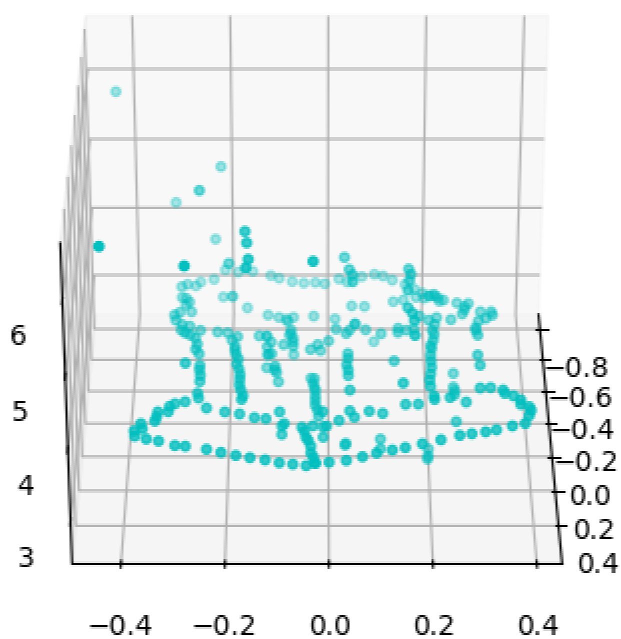
In [ ]:
```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
  '''
  Q4.2: Finding the 3D position of given points based on epipolar correspondence and t
  Input:  temple_pts1, chosen points from im1
          intrinsics, the intrinsics dictionary for calling epipolarCorrespondence
          F, the fundamental matrix
          im1, the first image
          im2, the second image
  Output: P (Nx3) the recovered 3D points

  Hints:
  (1) Use epipolarCorrespondence to find the corresponding point for [x1 y1] (find [x2
  (2) Now you have a set of corresponding points [x1, y1] and [x2, y2], you can comput
      matrix and use triangulate to find the 3D points.
  (3) Use the function findM2 to find the 3D points P (do not recalculate fundamental
  (4) As a reference, our solution's best error is around ~2200 on the 3D points.
  '''
  # ----- TODO -----
  # YOUR CODE HERE
  temple_points2 = np.zeros_like(temple_pts1)
  for i, (x1, y1) in enumerate(temple_pts1):
      xval, yval = epipolarCorrespondence(im1, im2, F, x1, y1)
      temple_points2[i] = [xval, yval]
  M2, C2, P = findM2(F, temple_pts1, temple_points2, intrinsics)

  return P
  # END YOUR CODE
```

Below, integrate everything together. The provided starter code loads in the temple data found at `data/templeCoords.npz`, which contains 288 hand-selected points from im1 saved in the variables x1 and y1. Then, get the 3d points from the 2d point point correspondences by calling the function you just implemented, as well as other necessary function. Finally, visualize the 3D reconstruction using matplotlib or plotly 3d scatter plot.

In [ ]:
```
temple_coords = np.load(os.path.join(DATA_DIR,'templeCoords.npz')) # Loading temple co
correspondence = np.load(os.path.join(DATA_DIR,'some_corresp.npz')) # Loading correspo
intrinsics = np.load(os.path.join(DATA_DIR,'intrinsics.npz')) # Loading the intrinscis
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread(os.path.join(DATA_DIR,'im1.png'))
im2 = plt.imread(os.path.join(DATA_DIR,'im2.png'))

# ----- TODO -----
# Call eightpoint to get the F matrix
# Call compute3D_pts to get the 3D points and visualize using matplotlib scatter
# hint: you can change the viewpoint of a matplotlib 3d axes using
# `ax.view_init(azim, elev)` where azim is the rotation around the vertical z
# axis, and elev is the angle of elevation from the x-y plane

temple_pts1 = np.hstack([temple_coords['x1'], temple_coords['y1']])

# YOUR CODE HERE
F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
P = compute3D_pts(temple_pts1, intrinsics, F, im1, im2)
# END YOUR CODE

fig = plt.figure()
```

```
ax = fig.add_subplot(111, projection='3d')
ax.scatter(P[:, 0], P[:, 1], P[:, 2], s=10, c='c', depthshade=True)
plt.draw()

# also show a different viewpoint
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(P[:, 0], P[:, 1], P[:, 2], s=10, c='c', depthshade=True)
ax.view_init(30, 0)
plt.draw()
```





Problem 5: Bundle Adjustment

Below is the implementation of RANSAC for Fundamental Matrix Recovery.

In [ ]:
```
def ransacF(pts1, pts2, M, nIters=100, tol=10):
  '''
  Input:  pts1, Nx2 Matrix
          pts2, Nx2 Matrix
          M, a scaler parameter
          nIters, Number of iterations of the Ransac
          tol, tolerence for inliers
  Output: F, the fundamental matrix
          inliers, Nx1 bool vector set to true for inliers

  '''
  N = pts1.shape[0]
  pts1_homo, pts2_homo = toHomogenous(pts1), toHomogenous(pts2)
  best_inlier = 0
  inlier_curr = None

  for i in range(nIters):
      choice = np.random.choice(range(pts1.shape[0]), 8)
      pts1_choice = pts1[choice, :]
      pts2_choice = pts2[choice, :]
      F = eightpoint(pts1_choice, pts2_choice, M)
      ress = calc_epi_error(pts1_homo, pts2_homo, F)
      curr_num_inliner = np.sum(ress < tol)
      if curr_num_inliner > best_inlier:
          F_curr = F
          inlier_curr = (ress < tol)
          best_inlier = curr_num_inliner
  inlier_curr = inlier_curr.reshape(inlier_curr.shape[0], 1)
  indixing_array = inlier_curr.flatten()
  pts1_inlier = pts1[indixing_array]
  pts2_inlier = pts2[indixing_array]
  F = eightpoint(pts1_inlier, pts2_inlier, M)
  return F, inlier_curr
```

Below is the implementation of Rodrigues and Inverse Rodrigues Formulas. See the pdf for the detailed explanation of the functions.

In [ ]:
```
def rodrigues(r):
  '''
      Input:  r, a 3x1 vector
      Output: R, a rotation matrix
  '''

  r = np.array(r).flatten()
  I = np.eye(3)
  theta = np.linalg.norm(r)
  if theta == 0:
      return I
  else:
      U = (r/theta)[:, np.newaxis]
      Ux, Uy, Uz = r/theta
```

```
        K = np.array([[0, -Uz, Uy], [Uz, 0, -Ux], [-Uy, Ux, 0]])
        R = I * np.cos(theta) + np.sin(theta) * K + \
            (1 - np.cos(theta)) * np.matmul(U, U.T)
    return R


def invRodrigues(R):
    '''
    Input:  R, a rotation matrix
    Output: r, a 3x1 vector
    '''

    def s_half(r):
        r1, r2, r3 = r
        if np.linalg.norm(r) == np.pi and (r1 == r2 and r1 == 0 and r2 == 0 and r3 < 0)
            return -r
        else:
            return r

    A = (R - R.T)/2
    ro = [A[2, 1], A[0, 2], A[1, 0]]
    s = np.linalg.norm(ro)
    c = (np.sum(np.matrix(R).diagonal()) - 1)/2
    if s == 0 and c == 1:
        r = np.zeros(3)
    elif s == 0 and c == -1:
        col = np.eye(3) + R
        col_idx = np.nonzero(
            np.array(np.sum(col != 0, axis=0)).flatten())[0][0]
        v = col[:, col_idx]
        u = v/np.linalg.norm(v)
        r = s_half(u * np.pi)
    else:
        u = ro/s
        theta = np.arctan2(s, c)
        r = u * theta

    return r
```

Rodrigues Residual objective function

In [ ]:
```
def rodriguesResidual(K1, M1, p1, K2, p2, x):
    '''
    Q5.1: Rodrigues residual.
      Input:  K1, the intrinsics of camera 1
              M1, the extrinsics of camera 1
              p1, the 2D coordinates of points in image 1
              K2, the intrinsics of camera 2
              p2, the 2D coordinates of points in image 2
              x, the flattened concatenationg of P, r2, and t2.
      Output: residuals, 4N x 1 vector, the difference between original and estimated pr
    '''
    N = p1.shape[0]
    # ----- TODO -----
    ### BEGIN SOLUTION
    # Set up P, r2, and t2
    P = x[:3*N].reshape([N, 3])
    r2 = x[3*N:3*N+3]
```

```
    t2 = x[3*N+3:]

    Ph = np.hstack([P, np.ones(N)[:, np.newaxis]])

    # Calculate camera matrices
    C1 = K1 @ M1
    C2 = K2 @ np.hstack([rodrigues(r2), t2[:, np.newaxis]])

    # Calculated new p values
    p1_new = Ph @ C1.T
    p2_new = Ph @ C2.T
    p1_new = p1_new[:, :2] / p1_new[:, 2:]
    p2_new = p2_new[:, :2] / p2_new[:, 2:]
    residuals = np.concatenate([(p1-p1_new).reshape([-1]), (p2-p2_new).reshape([-1])])
    ### END SOLUTION
    return residuals
```

Bundle Adjustment

In [ ]:

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

    Hints:
    (1) Use the scipy.optimize.minimize function to minimize the objective function, rod
        You can try different (method='..') in scipy.optimize.minimize for best results.
    '''
    obj_start = obj_end = 0
    # ----- TODO -----
    ### BEGIN SOLUTION
    # Find the x0 initial values
    x0 = np.concatenate([P_init.reshape([-1]), invRodrigues(M2_init[:, :3]), M2_init[:,

    # Setup the objective function
    f = lambda x: np.sum((rodriguesResidual(K1, M1, p1, K2, p2, x))**2)
    obj_start = f(x0)

    # Find the residuals
    res = scipy.optimize.minimize(f, x0, method='Powell')

    # Find optimized parameters
    P = res.x[:3*P_init.shape[0]].reshape(P_init.shape[0], 3)
    r2 = res.x[3*P_init.shape[0]:3*P_init.shape[0]+3]
    t2 = res.x[3*P_init.shape[0]+3:]
    M2 = np.hstack([rodrigues(r2), t2.reshape([-1, 1])])

    # End the optimization while storing the result
```

```
    obj_end = f(res.x)
    ### END SOLUTION
    return M2, P, obj_start, obj_end
```

Put it all together

1. Call the ransacF function to find the fundamental matrix
2. Call the findM2 function to find the extrinsics of the second camera
3. Call the bundleAdjustment function to optimize the extrinsics and 3D points
4. Plot the 3D points before and after bundle adjustment using the plot_3D_dual function

On the given temple data, bundle adjustment can take up to 2 min to run.

In [ ]:

```
# Visualization:
np.random.seed(1)
correspondence = np.load(os.path.join(DATA_DIR,'some_corresp_noisy.npz')) # Loading no
intrinsics = np.load(os.path.join(DATA_DIR,'intrinsics.npz')) # Loading the intrinscis
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread(os.path.join(DATA_DIR,'im1.png'))
im2 = plt.imread(os.path.join(DATA_DIR,'im2.png'))
M=np.max([*im1.shape, *im2.shape])

# YOUR CODE HERE
'''
Call the ransacF function to find the fundamental matrix
Call the findM2 function to find the extrinsics of the second camera
Call the bundleAdjustment function to optimize the extrinsics and 3D points
'''
# Call ransac
F, inliers = ransacF(pts1, pts2, M)

# Get inliers
pts1_inliers = pts1[inliers.flatten()]
pts2_inliers = pts2[inliers.flatten()]

# Find the P matrix
M2, C2, P = findM2(F, pts1_inliers, pts2_inliers, intrinsics)

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

# Do the bundle adjustment
M2_opt, P_opt, obj_start, obj_end = bundleAdjustment(K1, M1, pts1_inliers, K2, M2, pts
# END YOUR CODE
print(f"Before reprojection error: {obj_start}, After: {obj_end}")
```

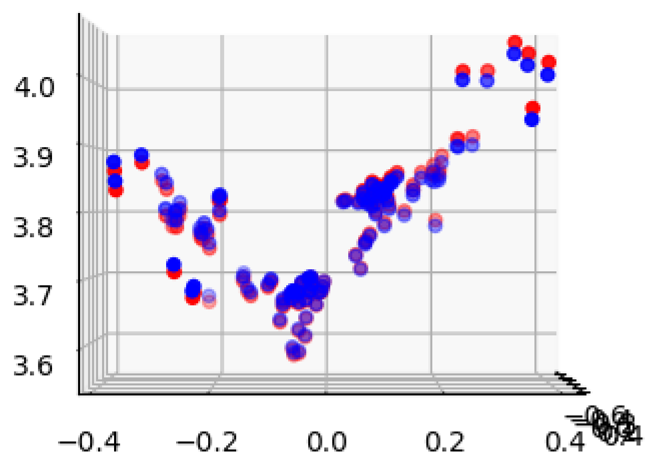
```
Before reprojection error: 352.84188180020834, After: 10.905073237845963
```

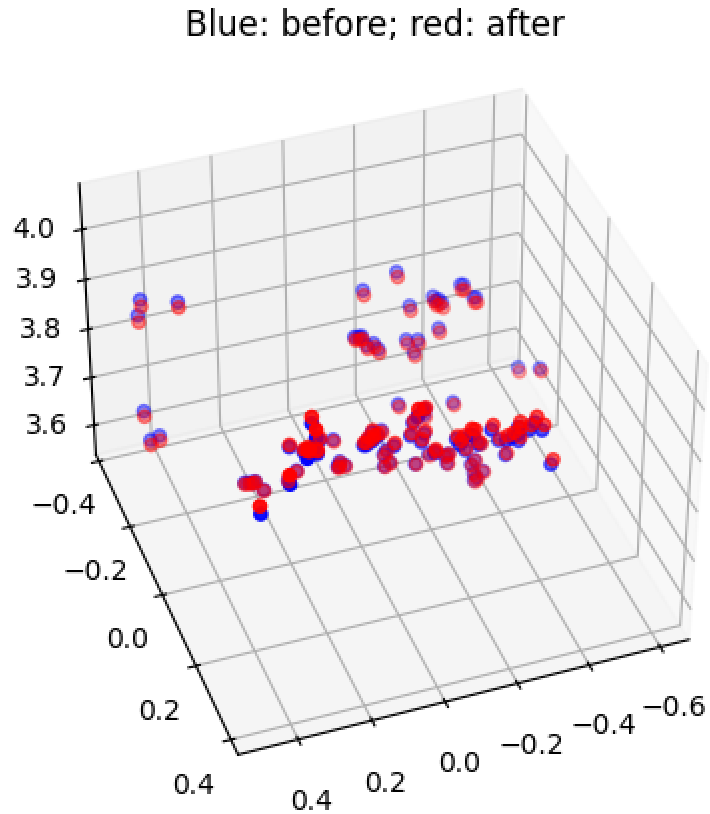
In [ ]:

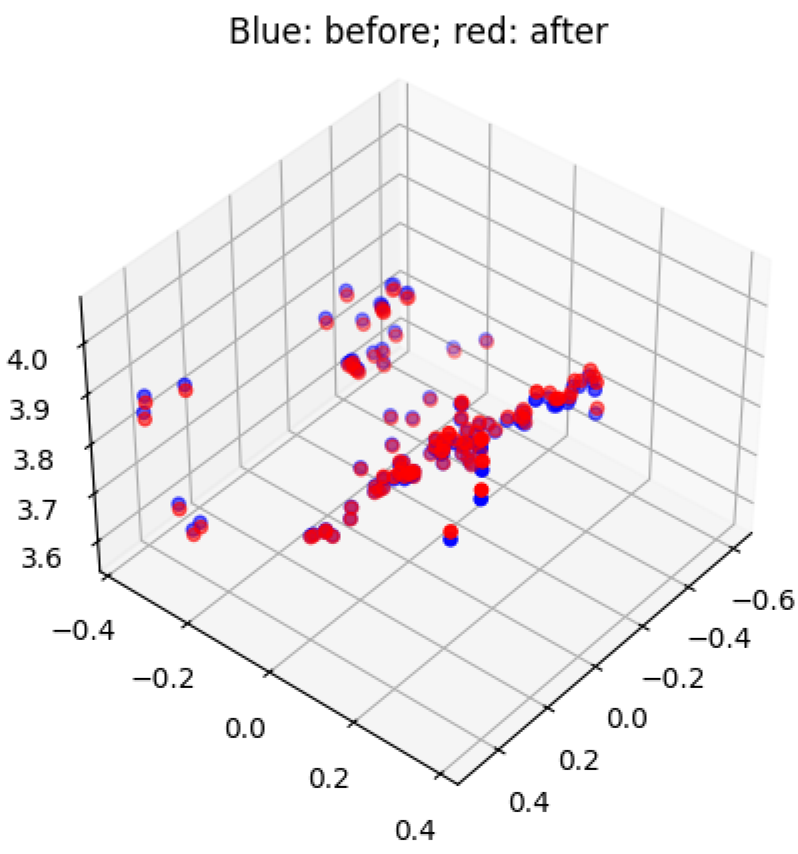
```
# helper function for visualization
def plot_3D_dual(P_before, P_after, azim=70, elev=45):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title("Blue: before; red: after")
    ax.scatter(P_before[:,0], P_before[:,1], P_before[:,2], c = 'blue')
    ax.scatter(P_after[:,0], P_after[:,1], P_after[:,2], c='red')
    ax.view_init(azim=azim, elev=elev)
    plt.draw()
```

```
P_start = P
P_end = P_opt

# plots the 3d points before and after BA from different viewpoints
plot_3D_dual(P_start, P_end, azim=0, elev=0)
plot_3D_dual(P_start, P_end, azim=70, elev=40)
plot_3D_dual(P_start, P_end, azim=40, elev=40)
```

Blue: before; red: after







(Extra Credit) Problem 6: Multiview Keypoint Reconstruction

6 Multi-View Reconstruction of keypoints

In [ ]:
```
def MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres = 100):
    '''
    Q6.1 Multi-View Reconstruction of keypoints.
        Input:  C1, the 3x4 camera matrix
                pts1, the Nx3 matrix with the 2D image coordinates and confidence per ro
                C2, the 3x4 camera matrix
                pts2, the Nx3 matrix with the 2D image coordinates and confidence per ro
                C3, the 3x4 camera matrix
                pts3, the Nx3 matrix with the 2D image coordinates and confidence per ro
        Output: P, the Nx3 matrix with the corresponding 3D points for each keypoint per
                err, the reprojection error.
    '''

    # Replace pass with your implementation
    # ------- TODO -------
    # YOUR CODE HERE

    return P, err
    # END YOUR CODE
```

Plot Spatio-temporal (3D) keypoints

In [ ]:
```
def plot_3d_keypoint_video(pts_3d_video):
    '''
    Plot Spatio-temporal (3D) keypoints
        :param car_points: np.array points * 3
    '''

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    for pts_3d in pts_3d_video:
        num_points = pts_3d.shape[1]
        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])
    np.set_printoptions(threshold=1e6, suppress=True)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()
```

Put it all together for all 10 timesteps.

In [ ]:
```
# pts_3d_video = []
# for loop in range(10):
#   print(f"processing time frame - {loop}")

#   data_path = os.path.join('data/q6/','time'+str(loop)+'.npz')
#   image1_path = os.path.join('data/q6/','cam1_time'+str(loop)+'.jpg')
#   image2_path = os.path.join('data/q6/','cam2_time'+str(loop)+'.jpg')
```

```
#   image3_path = os.path.join('data/q6/','cam3_time'+str(loop)+'.jpg')

#   im1 = plt.imread(image1_path)
#   im2 = plt.imread(image2_path)
#   im3 = plt.imread(image3_path)

#   data = np.load(data_path)
#   pts1 = data['pts1']
#   pts2 = data['pts2']
#   pts3 = data['pts3']

#   K1 = data['K1']
#   K2 = data['K2']
#   K3 = data['K3']

#   M1 = data['M1']
#   M2 = data['M2']
#   M3 = data['M3']

#   if loop == 0 or loop==9:  # feel free to modify to visualize keypoints at other lo
#     img = visualize_keypoints(im2, pts2)

#   # YOUR CODE HERE

#   # END YOUR CODE

#   if loop == 0:
#     plot_3d_keypoint(pts_3d)

# plot_3d_keypoint_video(pts_3d_video)
```

In [ ]:

```
```