HW4_Reconstruction

March 22, 2025

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:

---

1 16720 HW 4: 3D Reconstruction

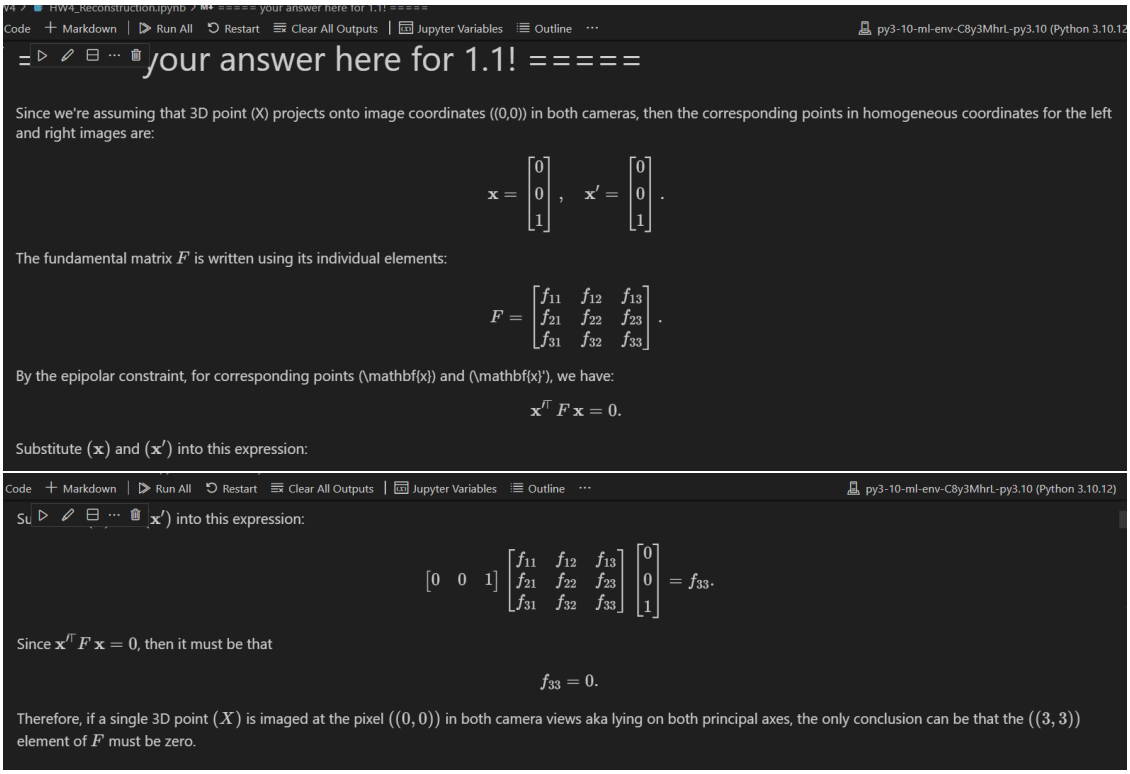
2 Problem 1: Theory

2.1 1.1

See pdf for the question.

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

Something in my LaTex kept making the pdf conversion fail so I did screen clips showing my unique python kernel to prove it was me



#

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

##1.2 See pdf for the question.

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

Let's establish that at time i, the robot's pose is described by a rotation $(\mathbf{R}_i)$ and translation $(\mathbf{t}_i)$. Likewise, at time $(i+1)$, its absolute pose is $(\mathbf{R}_{i+1})$ and $(\mathbf{t}_{i+1})$.

4.0.1 1. Relative Rotation and Translation

To find the **relative** rotation $(\mathbf{R}_{\text{rel}})$ and **relative** translation $(\mathbf{t}_{\text{rel}})$ from time (i) to $(i+1)$, we consider the transformation of a point in the (i)-th frame to the $(i+1)$-th frame.

1. The transformation from the world to the (i)-th frame is (in block form):
$$T_i = \begin{bmatrix} \mathbf{R}_i & \mathbf{t}_i \mathbf{0}^{\top} \\ & 1 \end{bmatrix}.$$
2. The inverse transform, from the (i)-th frame back to the world, is:
$$T_i^{-1} = \begin{bmatrix} \mathbf{R}_i^{\top} & -\mathbf{R}_i^{\top} \mathbf{t}_i \mathbf{0}^{\top} \\ & 1 \end{bmatrix}.$$
3. Similarly for time $(i+1)$, the transform to the world frame is (T_{i+1}). So to go from frame (i) to frame $(i+1)$, we compute (T_{i+1}, T_i^{-1}). The top-left (3×3) block of that product gives the relative rotation, while the top-right (3×1) block gives the relative translation.

Multiplying it out, we get:

$$\mathbf{R}_{\text{rel}} = \mathbf{R}_{i+1} \mathbf{R}_i^{\top}$$

$$\mathbf{t}_{\text{rel}} \;=\; \mathbf{t}_{i+1} \;-\; \mathbf{R}_{i+1}\,\mathbf{R}_i^{\top}\,\mathbf{t}_i.$$

4.0.2 2. Essential Matrix

The **Essential matrix** ($\mathbf{E}$) describing motion between two views can be written as:

$$\mathbf{E} \;=\; [\mathbf{t}_{\text{rel}}]_{\times}\,\mathbf{R}_{\text{rel}},$$

where ($[\mathbf{t}_{\text{rel}}]_{\times}$) is the ($3 \times 3$) skew-symmetric matrix that implements the cross product with ($\mathbf{t}_{\text{rel}}$). Therefore,

$$[\mathbf{t}_{\text{rel}}]_{\times} \;=\; \begin{bmatrix} 0 & -t_{\text{rel},3} & t_{\text{rel},2} \\ t_{\text{rel},3} & 0 & -t_{\text{rel},1} \\ -t_{\text{rel},2} & t_{\text{rel},1} & 0 \end{bmatrix}.$$

So then,

$$\mathbf{E} \;=\; [\mathbf{t}_{\text{rel}}]_{\times}\,\mathbf{R}_{\text{rel}}.$$

4.0.3 3. Fundamental Matrix

If the camera intrinsics are ($\mathbf{K}$), then the **Fundamental matrix** ($\mathbf{F}$) is related to ($\mathbf{E}$) by:

$$\mathbf{F} \;=\; (\mathbf{K}^{-1})^{\top}\,\mathbf{E}\,\mathbf{K}^{-1}.$$

Which is the same as:

$$\mathbf{F} \;=\; \mathbf{K}^{-\top}\,\mathbf{E}\,\mathbf{K}^{-1}.$$

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

[ ]:
```
import os
import numpy as np
import scipy
import scipy.optimize
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# from google.colab.patches import cv2_imshow running locally
import cv2


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

```
                  [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),(0,255,0),(0,255,0
colors =␣
 ↪['blue','blue','blue','blue','red','magenta','green','green','green','green','red','red','r


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 number of color channels
    :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)
    plt.figure(figsize=(8, 6))
    plt.imshow(image)
    plt.axis('off')
    plt.title("Keypoint Visualization")
    plt.show()

    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␣
 ↪normalized.
    :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 this image')

    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 this image \nand that the corresponding point matches')

    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()
```

##1.2 See pdf for the question.

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

Let's establish that at time i, the robot's pose is described by a rotation ($\mathbf{R}_i$) and translation ($\mathbf{t}_i$). Likewise, at time $(i+1)$, its absolute pose is ($\mathbf{R}_{i+1}$) and ($\mathbf{t}_{i+1}$).

6.0.1 1. Relative Rotation and Translation

To find the **relative** rotation ($\mathbf{R}_{\text{rel}}$) and **relative** translation ($\mathbf{t}_{\text{rel}}$) from time (i) to $(i+1)$, we consider the transformation of a point in the (i)-th frame to the $(i+1)$-th frame.

1. The transformation from the world to the (i)-th frame is (in block form):
$$T_i = \begin{bmatrix} \mathbf{R}_i & \mathbf{t}_i \mathbf{0}^\top \\ & 1 \end{bmatrix}.$$
2. The inverse transform, from the (i)-th frame back to the world, is:
$$T_i^{-1} = \begin{bmatrix} \mathbf{R}_i^\top & -\mathbf{R}_i^\top \mathbf{t}_i \mathbf{0}^\top \\ & 1 \end{bmatrix}.$$
3. Similarly for time $(i+1)$, the transform to the world frame is (T_{i+1}). So to go from frame (i) to frame $(i+1)$, we compute (T_{i+1}, T_i^{-1}). The top-left (3×3) block of that product gives the relative rotation, while the top-right (3×1) block gives the relative translation.

Multiplying it out, we get:

$$\begin{aligned} \mathbf{R}_{\text{rel}} &= \mathbf{R}_{i+1} \mathbf{R}_i^\top \\ \mathbf{t}_{\text{rel}} &= \mathbf{t}_{i+1} - \mathbf{R}_{i+1} \mathbf{R}_i^\top \mathbf{t}_i. \end{aligned}$$

6.0.2 2. Essential Matrix

The **Essential matrix** ($\mathbf{E}$) describing motion between two views can be written as:

$$\mathbf{E} = [\mathbf{t}_{\text{rel}}]_\times \mathbf{R}_{\text{rel}},$$

where ($[\mathbf{t}_{\text{rel}}]_\times$) is the (3×3) skew-symmetric matrix that implements the cross product with ($\mathbf{t}_{\text{rel}}$). Therefore,

$$[\mathbf{t}_{\text{rel}}]_\times = \begin{bmatrix} 0 & -t_{\text{rel},3} & t_{\text{rel},2} \\ t_{\text{rel},3} & 0 & -t_{\text{rel},1} \\ -t_{\text{rel},2} & t_{\text{rel},1} & 0 \end{bmatrix}.$$

So then,

$$\mathbf{E} = [\mathbf{t}_{\text{rel}}]_\times \mathbf{R}_{\text{rel}}.$$

6.0.3 3. Fundamental Matrix

If the camera intrinsics are ($\mathbf{K}$), then the **Fundamental matrix** ($\mathbf{F}$) is related to ($\mathbf{E}$) by:

$$\mathbf{F} = (\mathbf{K}^{-1})^\top \mathbf{E}\, \mathbf{K}^{-1}.$$

Which is the same as:

$$\mathbf{F} = \mathbf{K}^{-\top} \mathbf{E}\, \mathbf{K}^{-1}.$$

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

8 Coding

8.1 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.

[3]:
```
import os
import numpy as np
import scipy
import scipy.optimize
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# from google.colab.patches import cv2_imshow running locally
import cv2


connections_3d = [[0,1], [1,3], [2,3], [2,0], [4,5], [6,7], [8,9], [9,11],␣
 ↪[10,11], [10,8], [0,4], [4,8],
                  [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),(0,255,0),(0,255,0
colors =␣
 ↪['blue','blue','blue','blue','red','magenta','green','green','green','green','red','red','r
```

```python
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 number of color channels
    :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)
    plt.figure(figsize=(8, 6))
    plt.imshow(image)
    plt.axis('off')
    plt.title("Keypoint Visualization")
    plt.show()

    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␣
 ↪normalized.
    :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 this image')

    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 this image \nand that the corresponding point matches')

    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()
```

8.2 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.

[4]:
```
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")
```

```
--2025-03-22 00:22:33--  https://www.andrew.cmu.edu/user/eweng/data.zip
Resolving www.andrew.cmu.edu (www.andrew.cmu.edu)… 128.2.42.53
Connecting to www.andrew.cmu.edu (www.andrew.cmu.edu)|128.2.42.53|:443…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 21314971 (20M) [application/zip]
Saving to: ‘data.zip’

data.zip            100%[===================>]  20.33M  6.07MB/s    in 3.5s

2025-03-22 00:22:36 (5.77 MB/s) - ‘data.zip’ saved [21314971/21314971]

downloaded and unzipped data
```

9 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.

[7]:
```
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 enforce the singularity condition.
    (5) Use the function `refineF` (provided in the helper functions above) to␣
↪refine the computed fundamental matrix.
        (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 inputs using Matrix T
    #  Scale by 1/M for x, y; no change in homogeneous coord
    T = np.array([
        [1.0/M,  0,     0],
        [0,      1.0/M, 0],
        [0,      0,     1]
    ])

    # Convert pts1, pts2 to homogeneous and normalize
    pts1_h = np.hstack([pts1, np.ones((N, 1))])   # (N, 3)
    pts2_h = np.hstack([pts2, np.ones((N, 1))])   # (N, 3)
    pts1_norm = (T @ pts1_h.T).T                  # (N, 3)
    pts2_norm = (T @ pts2_h.T).T                  # (N, 3)

    # Construct the matrix A for A f = 0
    A = []
    for i in range(N):
        x1, y1 = pts1_norm[i, 0], pts1_norm[i, 1]
        x2, y2 = pts2_norm[i, 0], pts2_norm[i, 1]
        A.append([x2*x1, x2*y1, x2, y2*x1, y2*y1, y2, x1, y1, 1])
    A = np.array(A)  # shape (N, 9)

    # Solve for f using SVD
    U, S, Vt = np.linalg.svd(A)
    f = Vt[-1]                # last row of V^T => (9,)
    F_est = f.reshape(3, 3)

    # Enforce rank = 2
    F_est = _singularize(F_est)

    # Refine F_est by local minimization
    # Make sure to pass *normalized* points
    F_est = refineF(F_est, pts1_norm[:, :2], pts2_norm[:, :2])
    F_est = _singularize(F_est)
```

```
    # "Unscale" F back to the original coordinate system
    #    F = T^T * F_est * T
    F = T.T @ F_est @ T
    F_unscaled = T.T @ F_est @ T

    # Scale so that F[2,2] == 1
    # (only if F[2,2] != 0 -- but it shouldn't be zero in a valid solution)
    scale_factor = F_unscaled[2, 2]
    if abs(scale_factor) < 1e-12:
        # Failing gracefully if scale_factor is effectively zero:
        #   could raise an error or handle differently
        scale_factor = 1.0
    F_unscaled /= scale_factor

    F = F_unscaled

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

    return F
```

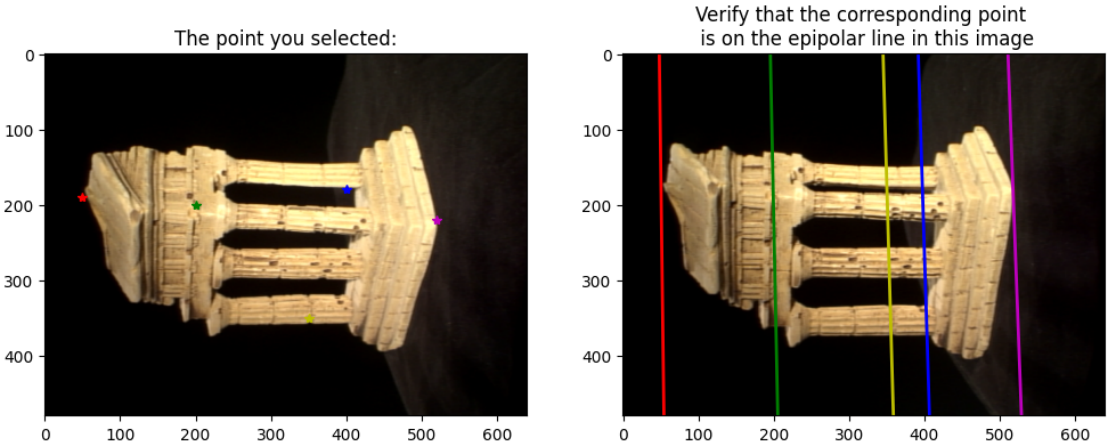
Run this code to test your implementation of the 8-point algorithm. Your code should pass all the assert statements at the end.

[8]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the␣
 ↪camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

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

# Simple Tests to verify your implementation:
pts1_homogenous, pts2_homogenous = toHomogenous(pts1), toHomogenous(pts2)

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 too high to be accurate"
```

```
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.

[9]:
```
# 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)
```



10 Problem 3: Metric Reconstruction

10.1 3.1 Essential Matrix

[12]:
```
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_est = K2.T @ F @ K1  # standard formula

  # Enforce rank(E) = 2, by setting the smallest singular value to zero
  U, S, Vt = np.linalg.svd(E_est)
  # Set the smallest singular value to 0
```

```
    S[-1] = 0
    E_est = U @ np.diag(S) @ Vt

    # Finally, scale so that E[2,2] == 1
    if abs(E_est[2, 2]) > 1e-12:
        E_est /= E_est[2, 2]
    else:
        # Fallback: If E_est[2,2] is extremely small
        E_est[2, 2] = 1
    ### END SOLUTION
    return E_est
```

Run the following code to check your implementation.

[13]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the␣
 ↪camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

F = eightpoint(pts1, pts2, M=np.max([*im1.shape, *im2.shape]))
E = essentialMatrix(F, K1, K2)
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]]
```

11 3.2 Triangulation

[14]:
```
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 and C1 & C2
        (2) Solve for the least square solution using np.linalg.svd
        (3) Calculate the reprojection error using the calculated 3D points and␣
↪C1 & C2 (do not forget to convert from
        homogeneous coordinates to non-homogeneous ones)
        (4) Keep track of the 3D points and projection error, and continue to␣
↪next point
        (5) You do not need to follow the exact procedure above.
        '''

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

        # Convert C1, C2 to row form for convenience
        #   C1 = [r1; r2; r3], each r is 1x4
        r1_C1 = C1[0, :]
        r2_C1 = C1[1, :]
        r3_C1 = C1[2, :]

        r1_C2 = C2[0, :]
        r2_C2 = C2[1, :]
        r3_C2 = C2[2, :]

        # Store 3D points here
        P = np.zeros((N, 3))

        for i in range(N):
                x1, y1 = pts1[i]
                x2, y2 = pts2[i]

                # Build the 4x4 matrix A for each correspondence
                #   from linear triangulation constraints:
                #
                #   (x1 * r3_C1 - r1_C1) * w = 0
                #   (y1 * r3_C1 - r2_C1) * w = 0
                #   (x2 * r3_C2 - r1_C2) * w = 0
                #   (y2 * r3_C2 - r2_C2) * w = 0
                A = np.array([
                x1 * r3_C1 - r1_C1,
                y1 * r3_C1 - r2_C1,
                x2 * r3_C2 - r1_C2,
                y2 * r3_C2 - r2_C2
                ])
```

```
                # Solve A w = 0 via SVD
                #  w is the homogeneous 3D point [X, Y, Z, 1]^T
                U, S, Vt = np.linalg.svd(A)
                w = Vt[-1]   # last row of V^T
                w /= w[-1]   # convert to non-homogeneous by dividing by w[3]

                # Save X, Y, Z in P
                P[i, :] = w[0:3]

        # ----- Compute reprojection error -----
        #  For each 3D point p = [X, Y, Z],
        #  project to each camera:  x_c = C p_h  (in homogeneous)
        #  then convert to 2D by dividing by the 3rd coordinate
        err = 0
        for i in range(N):
                # Original 3D point in homogeneous
                p3d_h = np.hstack([P[i], 1])

                # Project into each camera
                #   => x'_1_h, x'_2_h are 3D homogeneous image coords
                x1_h = C1 @ p3d_h
                x2_h = C2 @ p3d_h

                # Convert to non-homogeneous 2D
                x1_proj = x1_h[0:2] / x1_h[2]
                x2_proj = x2_h[0:2] / x2_h[2]

                # Compare to original points
                err += np.sum((pts1[i] - x1_proj)**2)
                err += np.sum((pts2[i] - x2_proj)**2)
        ### END SOLUTION

        return P, err
```

11.1 3.3 Find M2

[15]:
```
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]).max()], axis=1)
    M2s[:,:,1] = np.concatenate([U.dot(W).dot(V), -U[:,2].reshape([-1, 1])/abs(U[:
    ↪,2]).max()], axis=1)
    M2s[:,:,2] = np.concatenate([U.dot(W.T).dot(V), U[:,2].reshape([-1, 1])/
    ↪abs(U[:,2]).max()], axis=1)
    M2s[:,:,3] = np.concatenate([U.dot(W.T).dot(V), -U[:,2].reshape([-1, 1])/
    ↪abs(U[:,2]).max()], axis=1)
    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) 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'.

    '''

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

    # ----- TODO -----
    ### BEGIN SOLUTION
    # Compute E from F
    #  E = K2^T @ F @ K1
    #    Then get the four possible M2s from camera2(E).
    #    We'll do the standard approach here for completeness:
    E = K2.T @ F @ K1
    M2s = camera2(E)   # shape (3,4,4): four possible solutions
```

```
    # Define M1 and C1 for the first camera
    # M1 = [I | 0], so shape (3,4),
    # then C1 = K1 * M1
    M1 = np.hstack([np.eye(3), np.zeros((3,1))])  # 3x4
    C1 = K1 @ M1

    # Keep best values
    best_error = float('inf')
    best_M2 = None
    best_C2 = None
    best_P = None

    # Test each M2
    for i in range(4):
        M2_candidate = M2s[:, :, i]
        C2_candidate = K2 @ M2_candidate

        # Triangulate the 3D points
        # (Assuming you have a function 'triangulate' that returns (P, err).)
        P_candidate, err_candidate = triangulate(C1, pts1, C2_candidate, pts2)

        # Check how many points have z>0 in the first camera
        # (Often we pick the solution that yields all or most points in front.)
        num_positive_z = np.sum(P_candidate[:, 2] > 0)

        # If all points are in front of Cam1 and error is better, update best␣
↪solution
        # (Your assignment may vary: you might allow "most" points in front, or␣
↪also check Cam2.)
        if num_positive_z == P_candidate.shape[0] and err_candidate <␣
↪best_error:
            best_error = err_candidate
            best_M2 = M2_candidate
            best_C2 = C2_candidate
            best_P = P_candidate

    # 4) In case your data doesn't yield all in front,
    #    you might choose the best among the 4 anyway:
    if best_M2 is None:
        # fallback: pick the best error even if not all z>0
        fallback_error = float('inf')
        for i in range(4):
            M2_candidate = M2s[:, :, i]
            C2_candidate = K2 @ M2_candidate
            P_candidate, err_candidate = triangulate(C1, pts1, C2_candidate,␣
↪pts2)
            if err_candidate < fallback_error:
```

```
                fallback_error = err_candidate
                best_M2 = M2_candidate
                best_C2 = C2_candidate
                best_P = P_candidate
    ### END SOLUTION

    return best_M2, best_C2, best_P
```

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

[16]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the␣
 ↪camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

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

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)
C2 = K2.dot(M2)
P_test, err = triangulate(C1, pts1, C2, pts2)
assert(err < 500)
```

12 Problem 4: 3D Visualization

[17]:
```
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 epipolar line on image2
```

```
        (2) Search along this line to check nearby pixel intensity (you can␣
↪define a search window) to
        find the best matches
        (3) Use gaussian weighting to weight the pixel simlairty

        '''
        # ----- TODO -----
        # YOUR CODE HERE

        # Convert images to grayscale if needed
        if len(im1.shape) == 3:
                im1 = np.mean(im1, axis=2)
        if len(im2.shape) == 3:
                im2 = np.mean(im2, axis=2)
        im1 = im1.astype(np.float32)
        im2 = im2.astype(np.float32)

        # 2) Define a small patch size around (x1,y1).
        window_radius = 5
        window_size = 2 * window_radius + 1

        # Build a Gaussian weighting mask for the window.
        # 2D Gaussian for weighting patch comparison:
        sigma = 2.0
        # Create 1D Gaussian in [-r, r], then make 2D outer product
        coords = np.arange(-window_radius, window_radius+1)
        gauss_1d = np.exp(-(coords**2)/(2*sigma**2))
        gauss_2d = np.outer(gauss_1d, gauss_1d)
        gauss_2d /= gauss_2d.sum()  # normalize

        # Extract patch in im1 around (x1,y1), checking boundaries
        #    If patch goes out of bounds, do a safe extraction.
        h1, w1 = im1.shape
        # Ensure patch is fully inside the image
        if not (window_radius <= x1 < w1 - window_radius and window_radius <=␣
↪y1 < h1 - window_radius):
                # If out of bounds, we can clamp
                x1 = np.clip(x1, window_radius, w1 - window_radius - 1)
                y1 = np.clip(y1, window_radius, h1 - window_radius - 1)
        patch1 = im1[int(y1 - window_radius):int(y1 + window_radius + 1),
                        int(x1 - window_radius):int(x1 + window_radius + 1)]

        # Compute the epipolar line in im2 for (x1,y1).
        #    line l2 = F @ [x1,y1,1]^T => a*x + b*y + c=0
        #    In Python indexing, x ~ column, y ~ row
        x1_h = np.array([x1, y1, 1], dtype=np.float32)
        l2 = F @ x1_h
```

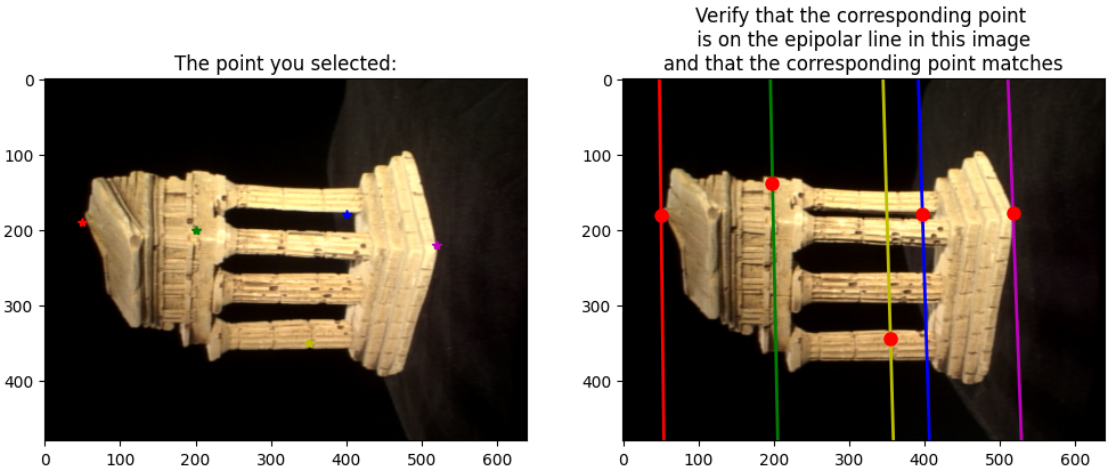
```
        a, b, c = l2

        # Search along the entire valid y-range, solve for x in im2.
        # x2 = -(b*y2 + c)/a if a != 0. Alternatively if a=0, we solve y2
        h2, w2 = im2.shape
        best_ssd = float('inf')
        best_x2, best_y2 = 0, 0

        # Define a bounding range for y2 so that the patch is valid
        for candidate_y2 in range(window_radius, h2 - window_radius):
        # Solve for candidate_x2 from line eqn a*x + b*y + c=0 => x=-(b*y+c)/a
↪if a!=0
                if abs(a) < 1e-12:
                        # If 'a' is effectively zero, fallback to a scanning
↪approach in x or skip.
                        continue
                candidate_x2 = -(b * candidate_y2 + c) / a
                candidate_x2_rounded = int(round(candidate_x2))

                # Check boundaries
                if candidate_x2_rounded < window_radius or candidate_x2_rounded
↪>= (w2 - window_radius):
                        continue

                # Extract patch in im2 around (candidate_x2_rounded,
↪candidate_y2)
                patch2 = im2[candidate_y2 - window_radius : candidate_y2 +
↪window_radius + 1,
                        candidate_x2_rounded - window_radius :
↪candidate_x2_rounded + window_radius + 1]

                # 7) Compute weighted SSD between patch1 and patch2
                if patch2.shape == patch1.shape:
                        diff = (patch1 - patch2)
                        ssd = np.sum( (diff**2) * gauss_2d )
                        # 8) Keep track of best match
                        if ssd < best_ssd:
                                best_ssd = ssd
                                best_x2 = candidate_x2_rounded
                                best_y2 = candidate_y2

        x2, y2 = best_x2, best_y2

        # END YOUR CODE
        return x2, y2
```

Run the following code to check your implementation.

[18]:
```
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the␣
 ↪camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

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

# 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.

[19]:
```
# 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

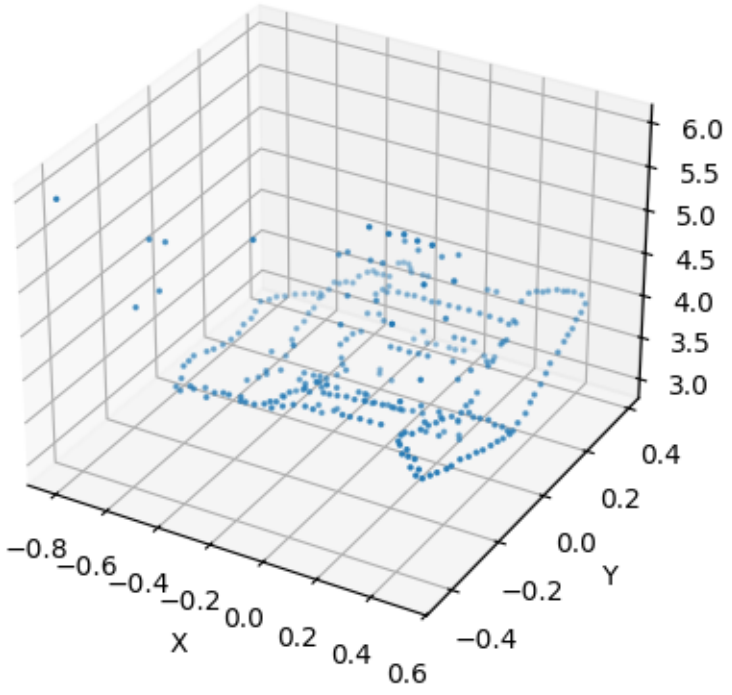
[20]:
```
def compute3D_pts(temple_pts1, intrinsics, F, im1, im2):
    '''
    Q4.2: Finding the 3D position of given points based on epipolar␣
 ↪correspondence and triangulation
    Input:  temple_pts1, chosen points from im1
            intrinsics, the intrinsics dictionary for calling␣
 ↪epipolarCorrespondence
```

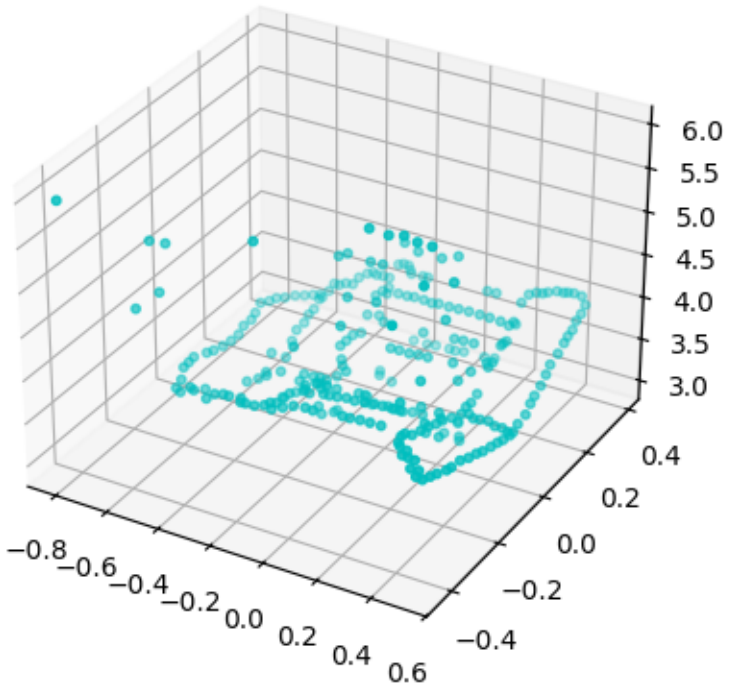
```
        F, the fundamental matrix
        im1, the first image
        im2, the second image
Output: P (Nx3) the recovered 3D points

Hints:
(1) Use epipolarCorrespondence to find the corresponding point for [x1 y1] (find [x2, y2])
(2) Now you have a set of corresponding points [x1, y1] and [x2, y2], you can compute the M2
    matrix and use triangulate to find the 3D points.
(3) Use the function findM2 to find the 3D points P (do not recalculate fundamental matrices)
(4) As a reference, our solution's best error is around ~2200 on the 3D points.
'''
# ----- TODO -----
# YOUR CODE HERE
N = temple_pts1.shape[0]
pts2 = np.zeros((N, 2), dtype=np.float32)

# 1) For each point in temple_pts1, find its match in im2
for i in range(N):
    x1, y1 = temple_pts1[i]
    x2, y2 = epipolarCorrespondence(im1, im2, F, x1, y1)
    pts2[i, 0] = x2
    pts2[i, 1] = y2

# 2) Now we have pts1 and pts2.  Call findM2 to get M2, C2, and the 3D points
M2, C2, P = findM2(F, temple_pts1, pts2, intrinsics)

# 3) Visualize the 3D points with matplotlib
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(P[:,0], P[:,1], P[:,2], s=2)
ax.set_title("3D Reconstruction of Temple Points")
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()

# Return the 3D points
return P
# END YOUR CODE
```

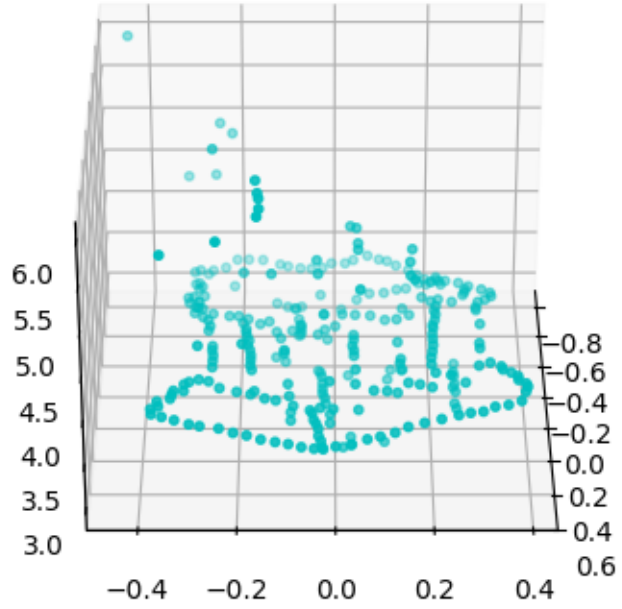
Below, integrate everything together. The provided starter code loads in the temple data found at `data/templeCoords.npz`, which contains 288 hand-selected points from im1 saved in the variables

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.

[22]:
```
temple_coords = np.load('data/templeCoords.npz') # Loading temple coordinates
correspondence = np.load('data/some_corresp.npz') # Loading correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the␣
 ↪camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')

# ----- 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
# 1) Call eightpoint to get the F matrix
M = np.max([*im1.shape, *im2.shape])  # scale parameter
F = eightpoint(pts1, pts2, M)

# 2) Call compute3D_pts to get the 3D points and visualize using matplotlib␣
 ↪scatter
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()
```


3D Reconstruction of Temple Points
6.0
5.5
5.0
4.5
4.0
3.5
3.0
0.4
0.2
0.0
−0.2
−0.4
Y
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
X


6.0
5.5
5.0
4.5
4.0
3.5
3.0
0.4
0.2
0.0
−0.2
−0.4
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6



13 Problem 5: Bundle Adjustment

Below is the implementation of RANSAC for Fundamental Matrix Recovery.

[23]:
```
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.

[24]:
```
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) or (r1 == 0 and r2 < 0) or (r1 < 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
```

13.0.1 Rodrigues Residual objective function

[25]:
```
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 projections
    '''
    N = p1.shape[0]
    # ----- TODO -----
    ### BEGIN SOLUTION
```

```
    # Parse out P, r2, t2 from x
    #     - The first 3*N entries of x correspond to the 3D points P
    #     - Then next 3 entries are r2
    #     - Then the final 3 entries are t2
    P_flat_size = 3 * N
    P_3D = x[:P_flat_size].reshape(N, 3)  # Nx3
    r2 = x[P_flat_size : P_flat_size + 3]  # 3,
    t2 = x[P_flat_size + 3 : P_flat_size + 6]  # 3,

    # Recover R2 from r2 using rodrigues
    R2 = rodrigues(r2)

    # Form M2 = [R2 | t2], shape 3x4
    M2 = np.hstack([R2, t2.reshape(3, 1)])

    # Build the full projection matrices C1 = K1 * M1 and C2 = K2 * M2
    C1 = K1 @ M1
    C2 = K2 @ M2

    # Project P_3D into each camera.
    #     Convert P_3D to homogeneous coordinates: Nx4
    P_homog = np.hstack([P_3D, np.ones((N, 1))])  # Nx4

    # Then do x_c = C @ P_homog.T  => shape (3,N)
    # Convert back to Nx2 by dividing the first two rows by the third.
    # We'll do it row-by-row or in bulk:
    p1_hat_h = (C1 @ P_homog.T).T  # Nx3
    p2_hat_h = (C2 @ P_homog.T).T  # Nx3

    # Convert to non-homogeneous (Nx2) by dividing x,y by z
    p1_hat = p1_hat_h[:, :2] / p1_hat_h[:, 2, np.newaxis]
    p2_hat = p2_hat_h[:, :2] / p2_hat_h[:, 2, np.newaxis]

    # Compute residuals = [ (p1 - p1_hat), (p2 - p2_hat) ], shape => (4N,)
    # And flatten them into a 1D array
    diff1 = p1 - p1_hat  # Nx2
    diff2 = p2 - p2_hat  # Nx2
    residuals = np.concatenate([diff1.flatten(), diff2.flatten()])
    ### END SOLUTION
    return residuals
```

13.0.2 Bundle Adjustment

[26]:
```
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, rodriguesResidual.
        You can try different (method='..') in scipy.optimize.minimize for best␣
↪results.
        '''
        obj_start = obj_end = 0
        # ----- TODO -----
        ### BEGIN SOLUTION

        # Extract R2, t2 from M2_init
        R2_init = M2_init[:, :3]   # shape (3,3)
        t2_init = M2_init[:, 3]    # shape (3,)

        # Convert R2_init to the Rodrigues vector r2 via invRodrigues
        r2_init = invRodrigues(R2_init)  # shape (3,)

        # Flatten initial 3D points P_init (size Nx3 => 3N)
        #    Then build the initial parameter vector x0
        N = P_init.shape[0]
        P_init_flat = P_init.flatten()  # shape (3N,)
        x0 = np.concatenate([P_init_flat, r2_init, t2_init])  # shape (3N + 3 +␣
↪3,)

        # Compute the initial objective: sum of squared residuals from␣
↪rodriguesResidual
        residuals_init = rodriguesResidual(K1, M1, p1, K2, p2, x0)
        obj_start = np.sum(residuals_init**2)

        # Define a lambda or function handle for the minimization
        def objective_wrapper(x_vec):
                # returns (4N,) => we want sum of squares, but minimize can␣
↪handle per-element residual
                res = rodriguesResidual(K1, M1, p1, K2, p2, x_vec)
                # We can either return res as array if we use least_squares,
                # or sum of squares if we use method='BFGS' or 'Powell'
```

```
            return np.sum(res**2)

        optim_result = scipy.optimize.minimize(objective_wrapper, x0,␣
↪method='Powell',
                                   options={'maxiter': 200})

        x_optimized = optim_result.x

        # Evaluate final objective
        residuals_opt = rodriguesResidual(K1, M1, p1, K2, p2, x_optimized)
        obj_end = np.sum(residuals_opt**2)

        # Parse out the final P, r2, t2 from x_optimized
        P_flat_size = 3 * N
        P_optimized = x_optimized[:P_flat_size].reshape((N, 3))
        r2_optimized = x_optimized[P_flat_size:P_flat_size+3]
        t2_optimized = x_optimized[P_flat_size+3:P_flat_size+6]

        # Convert r2_optimized back to R2
        R2_optimized = rodrigues(r2_optimized)

        # Form M2 = [R2 | t2]
        M2 = np.hstack([R2_optimized, t2_optimized.reshape(3, 1)])
        P = P_optimized
        ### 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.

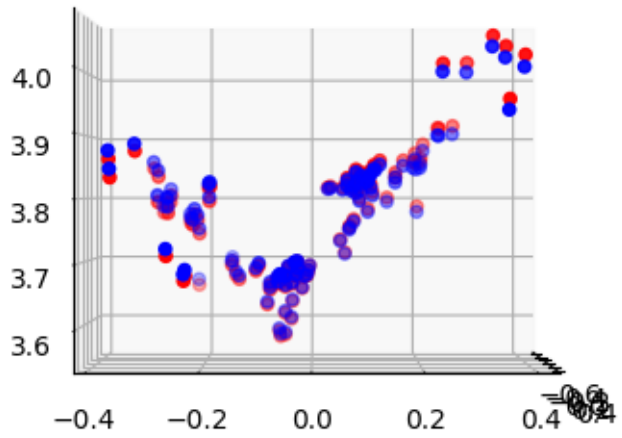
[29]:
```
# Visualization:
np.random.seed(1)
correspondence = np.load('data/some_corresp_noisy.npz') # Loading noisy␣
↪correspondences
intrinsics = np.load('data/intrinsics.npz') # Loading the intrinscis of the␣
↪camera
K1, K2 = intrinsics['K1'], intrinsics['K2']
pts1, pts2 = correspondence['pts1'], correspondence['pts2']
im1 = plt.imread('data/im1.png')
im2 = plt.imread('data/im2.png')
M=np.max([*im1.shape, *im2.shape])
```

```
# YOUR CODE HERE
# Call the ransacF function to find the fundamental matrix
F_ransac, inliers = ransacF(pts1, pts2, M, nIters=100, tol=10)
inlier_mask = inliers.flatten()
pts1_inliers = pts1[inlier_mask]
pts2_inliers = pts2[inlier_mask]

# Call the findM2 function to find the extrinsics of the second camera
M2_init, C2_init, P_init = findM2(F_ransac, pts1_inliers, pts2_inliers,␣
 ↪intrinsics)

# Call the bundleAdjustment function to optimize the extrinsics and 3D points
M2, P_final, obj_start, obj_end = bundleAdjustment(K1, M1, pts1_inliers, K2,␣
 ↪M2_init, pts2_inliers, P_init)


# END YOUR CODE
print(f"Before reprojection error: {obj_start}, After: {obj_end}")
```

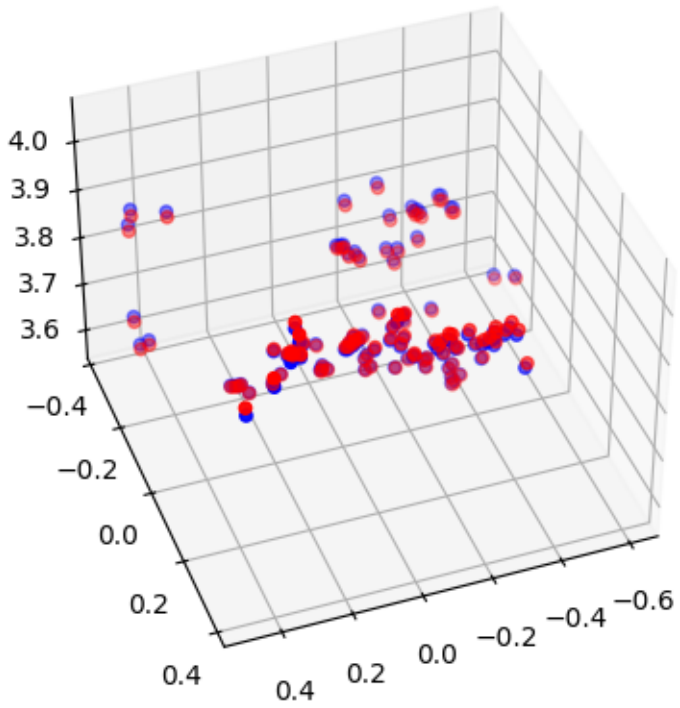
```
Before reprojection error: 352.8418811281551, After: 10.905073237212342
```

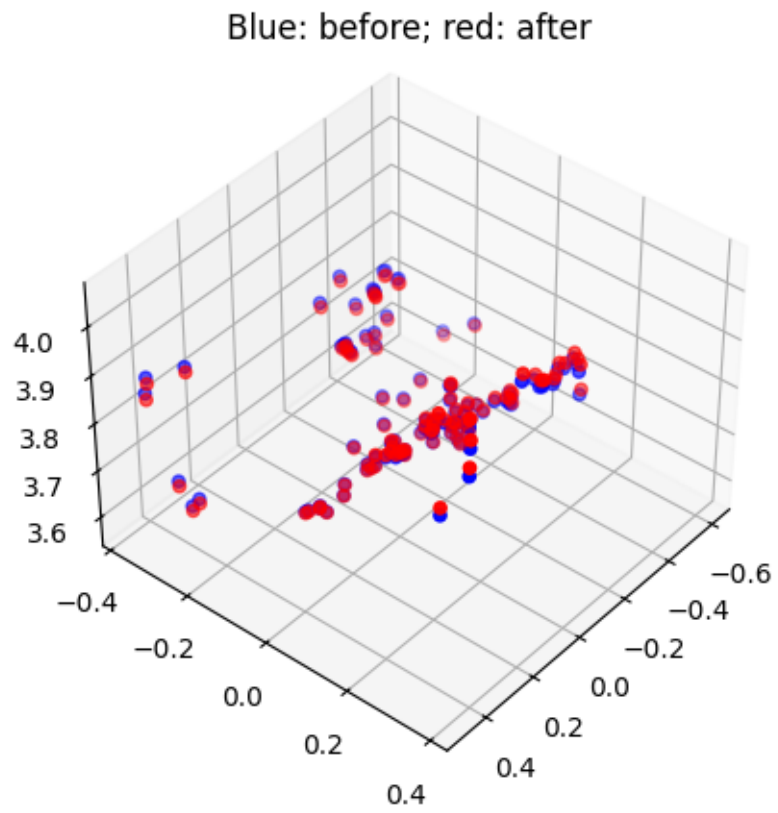
[30]:
```
# helper function for visualization
def plot_3D_dual(P_before, P_after, azim=70, elev=45):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title("Blue: before; red: after")
    ax.scatter(P_before[:,0], P_before[:,1], P_before[:,2], c = 'blue')
    ax.scatter(P_after[:,0], P_after[:,1], P_after[:,2], c='red')
    ax.view_init(azim=azim, elev=elev)
    plt.draw()

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

Blue: before; red: after



Blue: before; red: after





14 (Extra Credit) Problem 6: Multiview Keypoint Reconstruction

14.1 6 Multi-View Reconstruction of keypoints

[ ]:
```
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 row
              C2, the 3x4 camera matrix
              pts2, the Nx3 matrix with the 2D image coordinates and confidence␣
 ↪per row
              C3, the 3x4 camera matrix
              pts3, the Nx3 matrix with the 2D image coordinates and confidence␣
 ↪per row
```

```
    Output: P, the Nx3 matrix with the corresponding 3D points for each keypoint per row
            err, the reprojection error.
    '''

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

    return P, err
    # END YOUR CODE
```

14.1.1 Plot Spatio-temporal (3D) keypoints

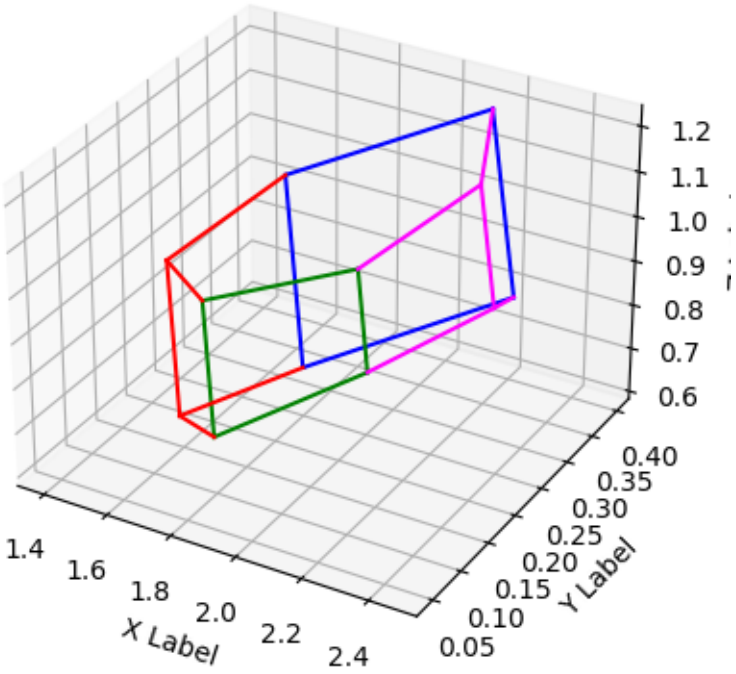
[ ]:
```
def plot_3d_keypoint_video(pts_3d_video):
    '''
    Plot Spatio-temporal (3D) keypoints
        :param car_points: np.array points * 3
    '''

    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.

[ ]:
```
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')
```

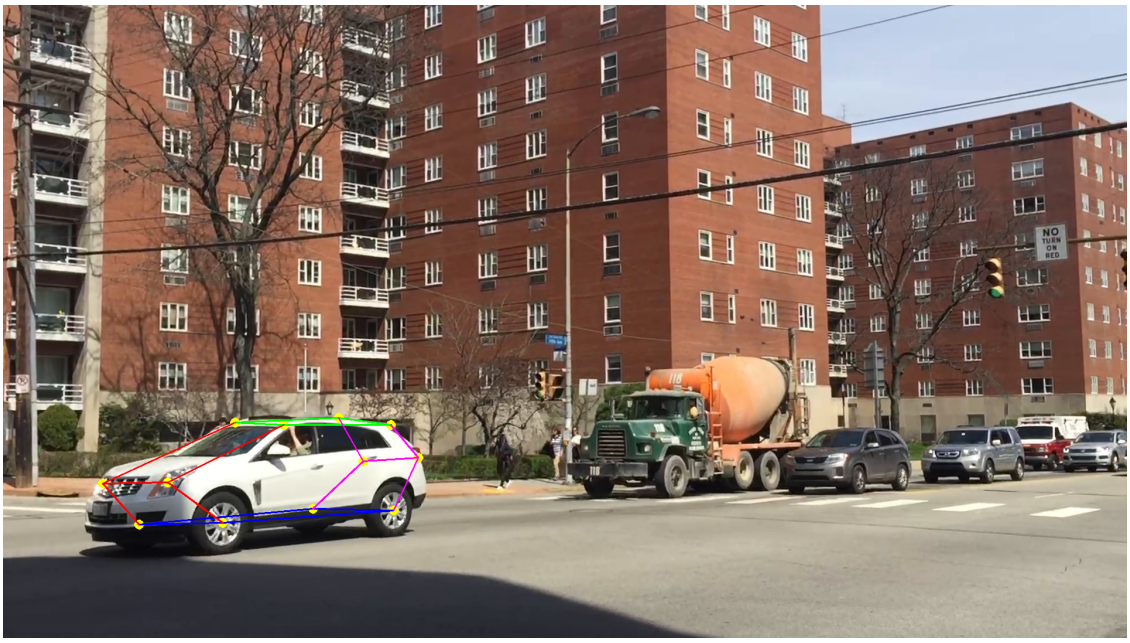
```
  im1 = plt.imread(image1_path)
  im2 = plt.imread(image2_path)
  im3 = plt.imread(image3_path)

  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 loop timesteps
    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)
```

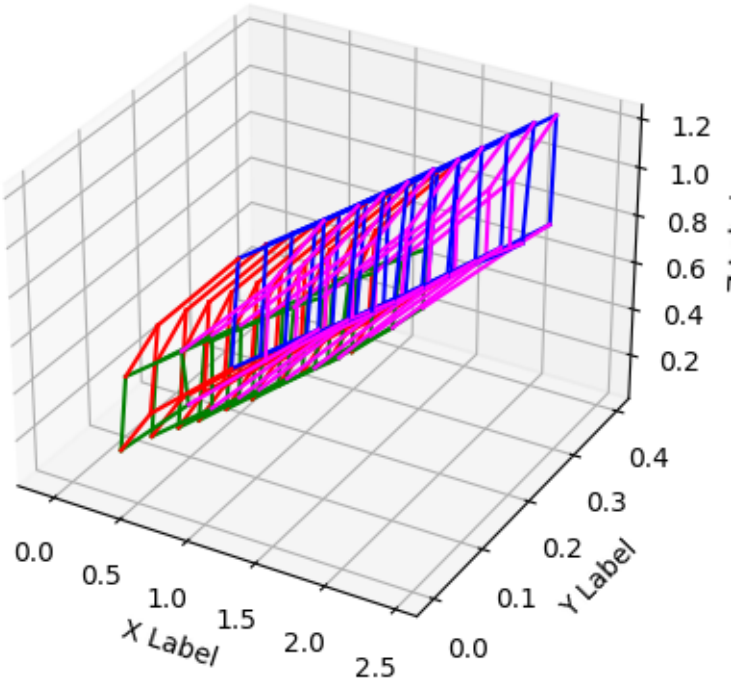
```
processing time frame - 0
```




```
processing time frame - 1
processing time frame - 2
processing time frame - 3
```

```
processing time frame - 4
processing time frame - 5
processing time frame - 6
processing time frame - 7
processing time frame - 8
processing time frame - 9
```




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

[ ]: