3d_reconstruction

July 2, 2020

1 3D Reconstruction

This exercise will guide you through two common operations we can apply to point correspondences extracted from stereo image pairs:

1. Estimation of the **fundamental matrix** (relating points in one image to lines in the other), and
2. **Triangulation** of 3D scene points given their projected image coordinates and the camera matrices.

We will use pre-computed point correspondences (loaded from file), which were estimated using a method similar to Exercise 3.

[ ]:
```
%%html
<!-- Run this cell to add heading letters per subtask (like a, b, c) -->
<style>
body {counter-reset: section;}
h2:before {counter-increment: section;
           content: counter(section, lower-alpha) ") ";}
</style>
```

[ ]:
```
%matplotlib notebook
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import imageio
import cv2
import os
from scipy import signal
from attrdict import AttrDict

def load_data(d_name):
    """ Load images and matches.

    Args:
        d_name: Should be 'house' or 'library'

```

```
    Returns:
        img0, img1: Images
        x0, x1: 3xN matrix of matching points
        P0, P1: 3X4 camera matrix
    """
    img0 = imageio.imread(f'{d_name}1.jpg')
    img1 = imageio.imread(f'{d_name}2.jpg')
    x0 = np.genfromtxt(f'{d_name}_matches_x1.csv', delimiter=',', dtype=np.
 ↪float64)
    x1 = np.genfromtxt(f'{d_name}_matches_x2.csv', delimiter=',', dtype=np.
 ↪float64)
    P0 = np.loadtxt(f'{d_name}1_camera.txt')
    P1 = np.loadtxt(f'{d_name}2_camera.txt')
    return img0, img1, x0, x1, P0, P1

def plot_multiple(images, titles=None, colormap='gray',
                  max_columns=np.inf, imwidth=4, imheight=4, share_axes=False):
    """Plot multiple images as subplots on a grid."""
    if titles is None:
        titles = [''] *len(images)
    assert len(images) == len(titles)
    n_images = len(images)
    n_cols = min(max_columns, n_images)
    n_rows = int(np.ceil(n_images / n_cols))
    fig, axes = plt.subplots(
        n_rows, n_cols, figsize=(n_cols * imwidth, n_rows * imheight),
        squeeze=False, sharex=share_axes, sharey=share_axes)

    axes = axes.flat
    # Hide subplots without content
    for ax in axes[n_images:]:
        ax.axis('off')

    if not isinstance(colormap, (list,tuple)):
        colormaps = [colormap]*n_images
    else:
        colormaps = colormap

    for ax, image, title, cmap in zip(axes, images, titles, colormaps):
        ax.imshow(image, cmap=cmap)
        ax.set_title(title)
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    fig.tight_layout()

def draw_keypoints(img, x):
```

```
    img = img.copy()
    for p, color in zip(x.T, colors):
        cv2.circle(img, (int(p[0]), int(p[1])), thickness=2, radius=1,␣
↪color=color)
    return img

def draw_point_matches(img0, img1, x0, x1, color_mask=None):
    result = np.concatenate([img0, img1], axis=1)
    img0_width = img0.shape[1]

    if color_mask is None:
        color_mask = np.ones(x0.shape[1], dtype=bool)

    for p0, p1, c_flag in zip(x0.T, x1.T, color_mask):
        p0x, p0y = int(p0[0]), int(p0[1])
        p1x, p1y = int(img0_width + p1[0]), int(p1[1])
        color = (0, 255, 0) if c_flag else (255, 0, 0)
        cv2.line(result, (p0x, p0y), (p1x, p1y),
                 color=color, thickness=1, lineType=cv2.LINE_AA)
    return result

def random_colors(n_colors):
    """Create a color map for visualizing the labels themselves,
    such that the segment boundaries become more visible, unlike
    in the visualization using the cluster peak colors.
    """
    import matplotlib.colors
    rng = np.random.RandomState(2)
    values = np.linspace(0, 1, n_colors)
    colors = plt.cm.get_cmap('hsv')(values)
    rng.shuffle(colors)
    return colors*255

colors = random_colors(1000)

def draw_line(img, l):
    if abs(l[0]) < abs(l[1]):
        # More horizontal
        slope = -l[0] / l[1]
        intercept = -l[2] / l[1]
        xs = np.array([0, img.shape[1]])
        ys = intercept + slope * xs
        cv2.line(img, (int(xs[0]), int(ys[0])), (int(xs[1]), int(ys[1])),
                 color=(0, 255, 255), thickness=1, lineType=cv2.LINE_AA)
    else:
        # More vertical
        slope = -l[1] / l[0]
```

```python
        intercept = -l[2] / l[0]
        ys = np.array([0, img.shape[0]])
        xs = intercept + slope * ys
        cv2.line(img, (int(xs[0]), int(ys[0])), (int(xs[1]), int(ys[1])),
                 color=(0, 255, 255), thickness=1, lineType=cv2.LINE_AA)

def draw_points_and_epipolar_lines(img, points, lines):
    """
    Args:
        img: First or second image
        x: 3xN matrix of points (on the same image)
        l: 3xN matrix of lines (on the same image)
    """
    points = points[:2] / points[2]  # Normalize
    img = img.copy()

    for l in lines.T:
        draw_line(img, l)

    for (x,y), color in zip(points.T, colors):
        cv2.circle(img, (int(x), int(y)), thickness=2, radius=1, color=color)

    return img
```

1.1 Fundamental Matrix Estimation

In this exercise, we will use the eight-point algorithm presented in the lecture in order to estimate the fundamental matrix between a pair of images. The overall workflow will be very similar to how we estimated the homography matrix in Exercise 3. The main difference is that a homography can only be used either when the scene is planar (with unrestricted camera transformation between the images) *or* when the camera is purely rotated but not translated (for unrestricted scene structure). If none of these conditions hold (non-planar scene with camera translation), we need to use the more general model represented by the fundamental matrix.

We will use a slightly simpler version of the algorithm here than the one presented in the lecture. Let's first assume that we are given a list of perfect correspondences $x = (u, v, 1)^T$ and $x = (u , v , 1)^T$ (in the code, we use `x0` for x' and `x1` for x), so that we don't have to deal with outliers. The fundamental matrix constraint states that each such correspondence must fulfill the equation

$$x'^T \mathrm{F} x = 0$$

We can reorder the entries of the matrix to transform this into the following equation

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

By stacking N 8 of those equations in a matrix A, we obtain the matrix equation

$$
\mathrm{Af} = 0 \tag{3}
$$

$$
\begin{bmatrix} u_1'u_1 & u_1'v_1 & u_1' & v_1'u_1 & v_1'v_1 & v_1' & u_1 & v_1 & 1 \\ u_2'u_2 & u_2'v_2 & u_2' & v_2'u_2 & v_2'v_2 & v_2' & u_2 & v_2 & 1 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ u_N'u_N & u_N'v_N & u_N' & v_N'u_N & v_N'v_N & v_N' & u_N & v_N & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ . \\ . \\ . \\ F_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ . \\ . \\ . \\ 0 \end{bmatrix} \tag{4}
$$

which can be easily solved by Singular Value Decomposition (SVD), as shown in Exercise 3. Applying SVD to A yields the decomposition A = UDV . The homogeneous least-squares solution corresponds to the least singular vector, which is given by the last column of V.

In the presence of noise, the matrix F estimated this way will, however, not satisfy the rank-2 constraint. This means that there will be no real epipoles through which all epipolar lines pass, but the intersection will be spread out over a small region. In order to enforce the rank-2 constraint, we therefore apply SVD to F and set the smallest singular value D_{33} to zero.

The reconstructed matrix will now satisfy the rank-2 constraint, and we can obtain the epipoles as

$$
\mathrm{Fe}_1 = 0
$$

$$
\mathrm{F}^{\mathrm{T}}\mathrm{e}_0 = 0
$$

by setting $\mathrm{e}_1 = \frac{[V_{13}, V_{23}, V_{33}]}{V_{33}}$ and $\mathrm{e}_0 = \frac{[U_{13}, U_{23}, U_{33}]}{U_{33}}$.

Similarly, for the points x, x , we can obtain the epipolar lines l = Fx, l = F x in the other image. Note that in projective geometry, a line is also defined by a single 3D vector. This can be easily seen by starting with the standard Euclidean formula for a line

$$
ax + by + c = 0
$$

and using the fact that the equation is unaffected by scaling to apply it to the homogeneous point x = (X, Y, W). Thus, we arrive at

$$aX + bY + cW = 0$$

$$\mathrm{l}^T x = x^T \mathrm{l} = 0$$

The parameters of the line are easily interpreted: $-a/b$ is the slope, $-c/a$ is the x-intercept, and $-c/b$ is the y-intercept.

If you are not comfortable with homogenuous coordinates or perspective geometry, have a look at following tutorials: http://www.maths.lth.se/matematiklth/personal/calle/datorseende13/notes/forelas2.pdf https://www.cse.unr.edu/~bebis/CS791E/Notes/EpipolarGeonetry.pdf http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/BEARDSLEY/node2.html

Write a function that implements the above algorithm to compute the fundamental matrix and the epipoles from a set of (at least 8) perfect correspondences given in the vectors x1 and x2.

[ ]:
```
def get_fundamental_matrix(x0, x1):
    """
    Args:
        x0, x1: 3xN arrays of N homogenous points in 2D

    Returns:
        F: The 3x3 fundamental martix such that x0.T @ F @ x1 = 0
        e0: The epipole in image 0 such that F.T  @ e0 = 0
        e1: The epipole in image 1 such that F @ e1 = 0
    """
    # YOUR CODE HERE
    raise NotImplementedError()

    return F, e0, e1
```

In order to get a quantitative estimate for the accuracy of your results, write a function `get_residual_distance` that computes the distance between points in one image and their corresponding epipolar lines (distance is positive). *Hint*: Be sure to normalize both homogeneous points (divide $(wx, wy, w)^T$ by w) and lines (divide $(a, b, d)^T$ by $\sqrt{a^2 + b^2}$) before computing the distance.

[ ]:
```
def get_residual_distance(F, x0, x1):
    # YOUR CODE HERE
    raise NotImplementedError()

    return d0, d1

def get_residual_error(F, x0, x1):
    d0, d1 = get_residual_distance(F, x0, x1)
    return 0.5 * (np.mean(d0) + np.mean(d1))
```

NOTE: throughout this exercise, you can replace 'house' with 'library' to try the algorithms on another stereo pair

[ ]:
```python
img0, img1, x0, x1, _, _ = load_data('house')
n_matches = 100
x0 = x0[:,:n_matches]
x1 = x1[:,:n_matches]

F, e0, e1 = get_fundamental_matrix(x0, x1)
residual_error = get_residual_error(F, x0, x1)

print(f'The estimated fundamental matrix F is \n{F}')

plot_multiple([draw_point_matches(img0, img1, x0, x1)],
              ['Point matches'], imwidth=8)
img0_result = draw_points_and_epipolar_lines(img0, x0, F @ x1)
img1_result = draw_points_and_epipolar_lines(img1, x1, F.T @ x0)
plot_multiple([img0_result, img1_result],
              ['Image 0 (without normalization)', 'Image 1 (without␣
 ↪normalization)'])

print(f'Epipole 0: {e0}, epipole 1: {e1}')
print(f'Fitting error: {residual_error:.1f} px')
```

1.2 Normalization

As explained in the lecture, we need to take care of normalizing the points in order to make sure the estimation problem is well conditioned. Write a function `normalize_points` that normalizes the given list of 2D points (in homogenuous coordinates) `x` by first shifting their origin to the centroid and then scaling them such that their mean distance from the origin is $\sqrt{2}$. Since the input points are homogeneous, pay attention to divide them by their last component before processing them. The function should return both the transformed points and the 3×3 transformation matrix T.

[ ]:
```python
def normalize_points(x):
    """
    Args:
        x: 3xN arrays of N homogenous points in 2D

    Return:
        x_trans: 3xN matrix of transformed points
        T: the 3x3 transformation matrix, points_trans = T * points
    """
    # YOUR CODE HERE
    raise NotImplementedError()

    return x_trans, T
```

Now write an adapted function `get_fundamental_matrix_with_normalization` that first normalizes the input points, computes the fundamental matrix based on the normalized points, and then undoes the transformation by applying $F = T_0 F T_1$ before computing the epipoles.

[ ]:
```python
def get_fundamental_matrix_with_normalization(x0, x1):
    """
    Args:
        x0, x1: 3xN arrays of N homogenous points in 2D

    Returns:
        F: The 3x3 fundamental martix such that x0'*F*x1 = 0
        e0: The epipole in image 0 such that F'*e0 = 0
        e1: The epipole in image 1 such that F*e1 = 0
    """
    # YOUR CODE HERE
    raise NotImplementedError()

    return F, e0, e1
```

[ ]:
```python
img0, img1, x0, x1, _, _ = load_data('house')

n_matches = 100
x0 = x0[:,:n_matches]
x1 = x1[:,:n_matches]

F, e0, e1 = get_fundamental_matrix_with_normalization(x0, x1)
residual_error = get_residual_error(F, x0, x1)
print(f'The estimated fundamental matrix F is \n{F}')

plot_multiple([draw_point_matches(img0, img1, x0, x1)],
              ['Point matches'], imwidth=8)
img0_result = draw_points_and_epipolar_lines(img0, x0, F @ x1)
img1_result = draw_points_and_epipolar_lines(img1, x1, F.T @ x0)
plot_multiple([img0_result, img1_result],
              ['Image 0 (with normalization)', 'Image 1 (with normalization)'])

print(f'Epipole 0: {e0}, epipole 1: {e1}')
print(f'Fitting error: {residual_error:.1f} px')
```

YOUR ANSWER HERE

[ ]:
```python
```

1.3 Selecting correspondences (optional)

Have a look at the provided script `label_matches.py`, which allows you to label your own correspondences. Test the implemented eight-point algorithm with your labeled data.

1.4 RANSAC

In practice, the correspondence set will always contain noise and outliers. We therefore apply RANSAC in order to get a robust estimate. It proceeds along the following steps:

1. Randomly select a (minimal) seed group of point correspondences on which to base the estimate.
2. Compute the fundamental matrix from this seed group.
3. Find inliers to this transformation.
4. If the number of inliers is sufficiently large (m), recompute the least-squares estimate of the fundamental matrix on all inliers.
5. Else, repeat for a maximum of k iterations.

The parameter k can be chosen automatically. Suppose w is the fraction of inlier correspondences and $n = 8$ correspondences are needed to define a hypothesis. Then the probability that a single sample of n correspondences is correct is w^n, and the probability that all samples fail is $(1-w^n)^k$. The standard strategy is thus, given an estimate for w, to choose k high enough that this value is kept below our desired failure rate.

In the following, we will implement the different steps of the RANSAC procedure and apply it for robust estimation of the fundamental matrix.

First write a function which takes as input an estimated fundamental matrix and the full set of correspondence candidates and which returns: *(1)* the ratio, and *(2)* the indices of the inliers. A point pair x, x is defined to be an inlier if the distance of x to the epipolar line $\mathrm{l} = \mathrm{F}\ \mathrm{x}$, as well as the opposite distance, are both less than some threshold .

[ ]:
```
def get_inliers(F, x0, x1, eps):
    # YOUR CODE HERE
    raise NotImplementedError()
    return indices
```

Now write a function which implements the RANSAC procedure to estimate a fundamental matrix using the normalized eight-point algorithm. For randomly sampling matches, you can use the `np.random.choice()` function. Here, we want to use a simple version of the algorithm that just runs for a fixed number of `n_iter` iterations and returns the solution with the largest inlier set.

[ ]:
```
def get_fundamental_matrix_with_ransac(x0, x1, eps=10, n_iter=1000):
    """
    Args:
        x0, x1: 3xN arrays of N homogenoous points in 2D
        eps: Inlier threshold
        n_iter: Number of iterations

    Return:
        F: The 3x3 fundamental martix such taht x2'*F*x1 = 0
        e0: The epipole in image 1 such that F'*e0 = 0
        e1: The epipole in image 2 such that F*e1 = 0
        inlier_ratio: Ratio of inlier
        inlier_indices: Indices of inlier
```

```
    """
    # YOUR CODE HERE
    raise NotImplementedError()

    return F, e0, e1, best_inlier_indices
```

Since we are working with precomputed ground-truth point correspondences loaded from file (and not estimated ones as we did in Exercise 3), we will artificially create outlier pairs using the following function. This is to simulate the effect of noisy point correspondence estimation.

```
[ ]: def inject_outliers(img, x0, x1, outlier_ratio):
         """Artifically create outliers.

         Args:
             img: Image, used only for size
             x0, x1: Input 3xN points
             outlier_ratio: Probability (ratio) of outlier

         Returns:
             x0_noisy, x1_noisy: Output points (with outlier)
             inlier_mask: 1D binary array
         """
         n_pts = x0.shape[1]
         h, w = img.shape[:2]

         outlier_mask = np.random.uniform(size=n_pts) < outlier_ratio
         n_outliers = sum(outlier_mask)
         outlier_x01 = np.random.randint(low=0, high=w, size=(2, n_outliers))
         outlier_y01 = np.random.randint(low=0, high=h, size=(2, n_outliers))

         x0_noisy, x1_noisy = x0.copy(), x1.copy()
         x0_noisy[0, outlier_mask] = np.random.randint(low=0, high=w,␣
      ↪size=n_outliers)
         x0_noisy[1, outlier_mask] = np.random.randint(low=0, high=h,␣
      ↪size=n_outliers)
         x1_noisy[0, outlier_mask] = np.random.randint(low=0, high=w,␣
      ↪size=n_outliers)
         x1_noisy[1, outlier_mask] = np.random.randint(low=0, high=h,␣
      ↪size=n_outliers)

         return x0_noisy, x1_noisy, np.invert(outlier_mask)
```

```
[ ]: img0, img1, x0, x1, _, _ = load_data('house')
     n_matches = 100
     x0 = x0[:, :n_matches]
     x1 = x1[:, :n_matches]
     x0, x1, inlier_mask = inject_outliers(img0, x0, x1, outlier_ratio=0.3)
```

```python
x0_inlier, x1_inlier = x0[:, inlier_mask], x1[:, inlier_mask]
inlier_ratio = sum(inlier_mask) / x0.shape[1]

# Without RANSAC
F_vanilla, e0_vanilla, e1_vanilla =␣
↪get_fundamental_matrix_with_normalization(x0, x1)
residual_error_vanilla = get_residual_error(F_vanilla, x0_inlier, x1_inlier)
print('-------- Without RANSAC --------')
print(f'The estimated fundamental matrix F is \n{F_vanilla}')
print(f'Epipole 0: {e0_vanilla}, epipole 1: {e1_vanilla}')
print(f'Fitting error: {residual_error_vanilla:.1f} px')

# With RANSAC
eps = 1
n_iter = 1000
F_ransac, e0_ransac, e1_ransac, inlier_indices =␣
↪get_fundamental_matrix_with_ransac(
    x0, x1, eps, n_iter)
residual_error_ransac = get_residual_error(F_ransac, x0_inlier, x1_inlier)
inlier_ratio_ransac = len(inlier_indices) / x0.shape[1]
inlier_mask_ransac = np.zeros(x0.shape[1], dtype=bool)
inlier_mask_ransac[inlier_indices] = True
print('-------- With RANSAC --------')
print(f'The estimated fundamental matrix F is \n{F_ransac}')
print(f'Epipole 0: {e0_ransac}, epipole 1: {e1_ransac}')
print(f'Fitting error: {residual_error_ransac:.1f} px')
print(f'Inlier ratio: {inlier_ratio_ransac:.0%} (groundtruth: {inlier_ratio:.
↪0%})')

# Plotting
plot_multiple([draw_point_matches(img0, img1, x0, x1, inlier_mask),
              draw_point_matches(img0, img1, x0, x1, inlier_mask_ransac)],
              ['Point matches (ground truth)',
              'Point matches (ransac)'], imwidth=8, max_columns=1)

img0_vanilla = draw_points_and_epipolar_lines(img0, x0, F_vanilla @ x1)
img1_vanilla = draw_points_and_epipolar_lines(img1, x1, F_vanilla.T @ x0)
img0_ransac = draw_points_and_epipolar_lines(img0, x0_inlier, F_ransac @␣
↪x1_inlier)
img1_ransac = draw_points_and_epipolar_lines(img1, x1_inlier, F_ransac.T @␣
↪x0_inlier)
plot_multiple([img0_vanilla, img1_vanilla, img0_ransac, img1_ransac],
              ['Image 0 (without RANSAC)', 'Image 1 (without RANSAC)',
               'Image 0 (with RANSAC)', 'Image 1 (with RANSAC)'], max_columns=2)
```

1.5 Harris points (optional)

We can use a keypoint extractor and descriptor to find matches, as in Exercise 3. You can either use code from there, or use built-in functions of OpenCV (look up how to use `cv2.BRISK_create` for example – BRISK is a non-patented alternative of SIFT).

With this, we have a full pipeline from images to the fundamental matrix.

1.6 Triangulation

As a final step, we want to reconstruct the observed points in 3D by triangulation. Note that just using two images, this is not possible without a calibration. You can therefore find camera matrices for each image provided in the archive *exercise5.zip*. They are stored as simple text files containing a single 3×4 matrix and can be read in with the `np.loadtxt` command.

For triangulation, we use the linear algebraic approach from the lecture. Given a 2D point correspondence $\mathrm{x}_1, \mathrm{x}_2$ in homogeneous coordinates, the 3D point location X is given as follows:

$$\lambda_1 \mathrm{x}_1 = \mathrm{P}_1 \mathrm{X}$$

$$\lambda_2 \mathrm{x}_2 = \mathrm{P}_2 \mathrm{X}$$

We can now build the cross-product of each point with both sides of the equation and obtain

$$\mathrm{x}_1 \times \mathrm{P}_1 \mathrm{X} = [\mathrm{x}_{1\times}] \mathrm{P}_1 \mathrm{X} = 0$$

$$\mathrm{x}_2 \times \mathrm{P}_2 \mathrm{X} = [\mathrm{x}_{2\times}] \mathrm{P}_2 \mathrm{X} = 0$$

where we used the skew-symmetrix matrices $[\mathrm{x}_{\mathrm{i}\times}]$ to replace the cross products

$$\mathrm{a} \times \mathrm{b} = [\mathrm{a}_\times]\mathrm{b} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \mathrm{b}$$

Each 2D point provides 2 independent equations for a total of 3 unknowns. We can therefore solve the overconstrained system by stacking the first two equations for each point in a matrix A and computing the least-squares solution for $\mathrm{AX} = 0$.

First write a function to find the centers of both cameras. Recall from the lecture that the camera centers are given by the null space of the camera matrices. They can thus be found by taking the SVD of the camera matrix and taking the last column of V normalized by the 4th entry.

[ ]:
```python
def camera_center_from_projection_matrix(P):
    # YOUR CODE HERE
    raise NotImplementedError()
    return center
```

```
img0, img1, x0, x1, P0, P1 = load_data('house')
C0 = camera_center_from_projection_matrix(P0)
C1 = camera_center_from_projection_matrix(P1)

print(f'P0: {P0}')
print(f'P1: {P1}')
print(f'C0: {C0}')
print(f'C1: {C1}')
```

Now write a function *triangulate* that uses linear least-square method to triangulate the position of a matching point pair in 3D, as described above. A well suited helper function is *vector_to_skew(v)* which returns a skew symmetric matrix from the vector v with 3 elements.

[ ]:
```
def vector_to_skew(vec):
    return np.array([[0,      -vec[2], vec[1]],
                     [ vec[2], 0,     -vec[0]],
                     [-vec[1], vec[0], 0]])

def triangulate(x0, x1, P0, P1):
    """ Triangulate matching points.
    Args:
        x0, x1: 3xN matrices of matching points (homogeneous)
        P0, P1: 3X4 camera matrices

    Returns:
        X: 3xN matrix of points in world space
    """
    # YOUR CODE HERE
    raise NotImplementedError()
    return X
```

Write a function to compute the reprojection errors (average distance) between the observed 2D points and the projected 3D points in the two images.

[ ]:
```
def get_reprojection_error(X, x, P):
    # YOUR CODE HERE
    raise NotImplementedError()
    return np.mean(distance)
```

[ ]:
```
def plot_3d_reconstruction(X, C0, C1):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    c = colors[:X.shape[1]]/255
    ax.scatter(X[0], X[1], X[2], c=c, alpha=1)
    ax.scatter(C0[0], C0[1], C0[2], color='red', s=100, marker='x')
    ax.scatter(C1[0], C1[1], C1[2], color='blue', s=100, marker='x')
    plt.show()
```

```python
X = triangulate(x0, x1, P0, P1)
error0 = get_reprojection_error(X, x0, P0)
error1 = get_reprojection_error(X, x1, P1)
error = 0.5 * (error0 + error1)

# Rotate the visualization to be more recognizable.
U,S,Vt = np.linalg.svd(P0[:3,:3], full_matrices=False)
rot = U @ Vt
rot = np.array([[1,0,0], [0,0,1], [0,-1,0]]) @ rot

plot_multiple([draw_keypoints(img0, x0), draw_keypoints(img1, x1)],
              ['Image 0', 'Image 1'])
plot_3d_reconstruction(rot @ X[:3]/X[3], rot @ C0, rot @ C1)
print(f'Reprojection error: {error:.1f} px')
```

1.7 Reconstruction with RANSAC (optional)

Reconstruction quality can be further improved by filtering out outliers using RANSAC based on fundamental matrix. Try to implement it.