Computer Vision HW4 Group 16

— 0856622 余家宏、309551067 吳子涵、309551122 曹芳驊

Introduction

- Structure from Motion (SfM) is a photogrammetric range imaging technique for estimating three-dimensional structures from two-dimensional image sequences.
- In this assignment, we try to reconstruct 3D points from two images. The keypoints and descriptors are extracted from images using Scale-Invariant Feature Transform (SIFT), and apply ratio distance as metrics to get correspondences of the two images. We run eight-point algorithm with RANSAC to get the fundamental matrix, and then epipolar lines and four possible essential matrices can be found. Finally, choose the best essential matrix with maximum number of points in front of camera.

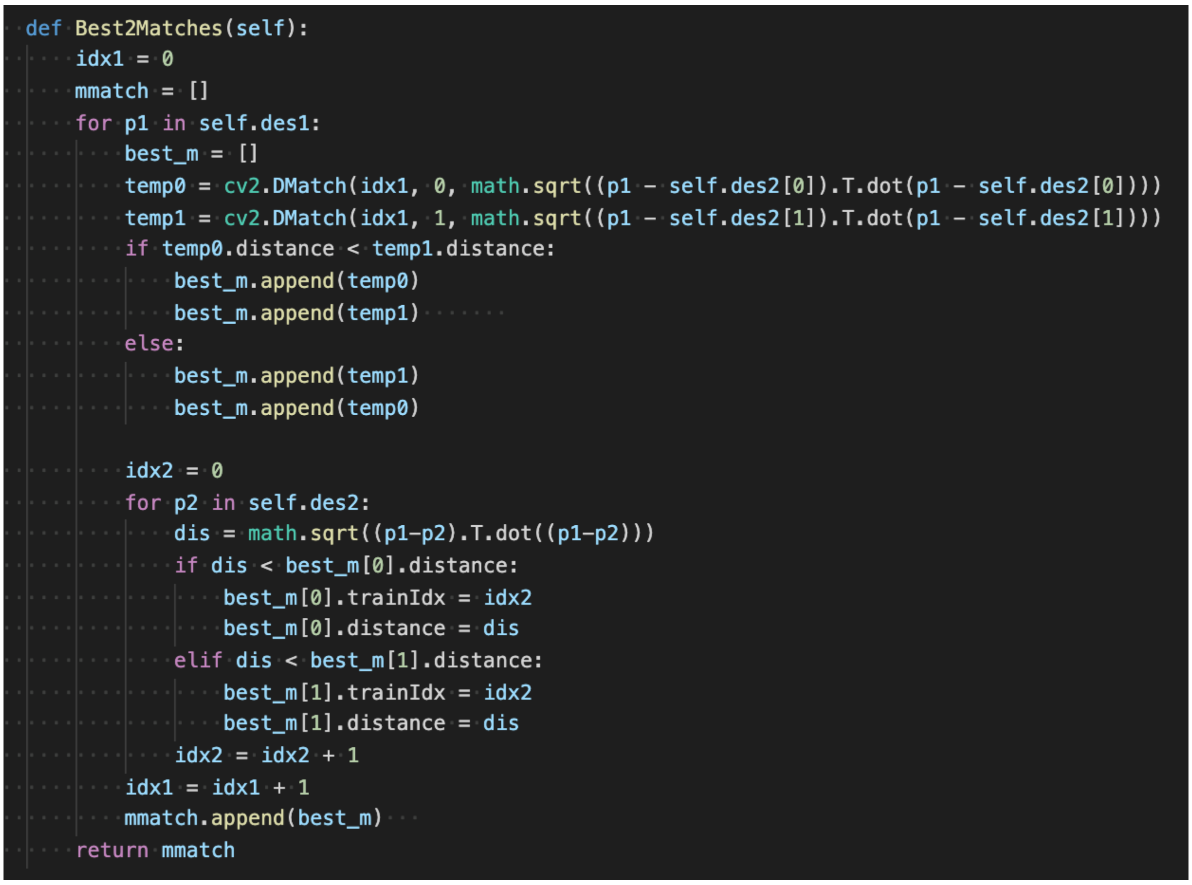
Implementation Procedure

Step 1: Find our correspondences across images

- Calculate keypoints and descriptors of an image by SIFT.

- Apply ratio distance $\frac{||f_1 - f_2||}{||f_1 - f_2'||}$ as metrics to find out the correspondences across images.

- f_1 is a feature of image 1, f_2 is the best SSD match to f_1 of image 2, and f_2' is the second best SSD match to f_1 of image 2. If the ratio distance is smaller than a preset threshold, f_2 will be considered as the match feature point of f_1.



```python
def Best2Matches(self):
    idx1 = 0
    mmatch = []
    for p1 in self.des1:
        best_m = []
        temp0 = cv2.DMatch(idx1, 0, math.sqrt((p1 - self.des2[0]).T.dot(p1 - self.des2[0])))
        temp1 = cv2.DMatch(idx1, 1, math.sqrt((p1 - self.des2[1]).T.dot(p1 - self.des2[1])))
        if temp0.distance < temp1.distance:
            best_m.append(temp0)
            best_m.append(temp1)
        else:
            best_m.append(temp1)
            best_m.append(temp0)

        idx2 = 0
        for p2 in self.des2:
            dis = math.sqrt((p1-p2).T.dot((p1-p2)))
            if dis < best_m[0].distance:
                best_m[0].trainIdx = idx2
                best_m[0].distance = dis
            elif dis < best_m[1].distance:
                best_m[1].trainIdx = idx2
                best_m[1].distance = dis
            idx2 = idx2 + 1
        idx1 = idx1 + 1
        mmatch.append(best_m)
    return mmatch
```

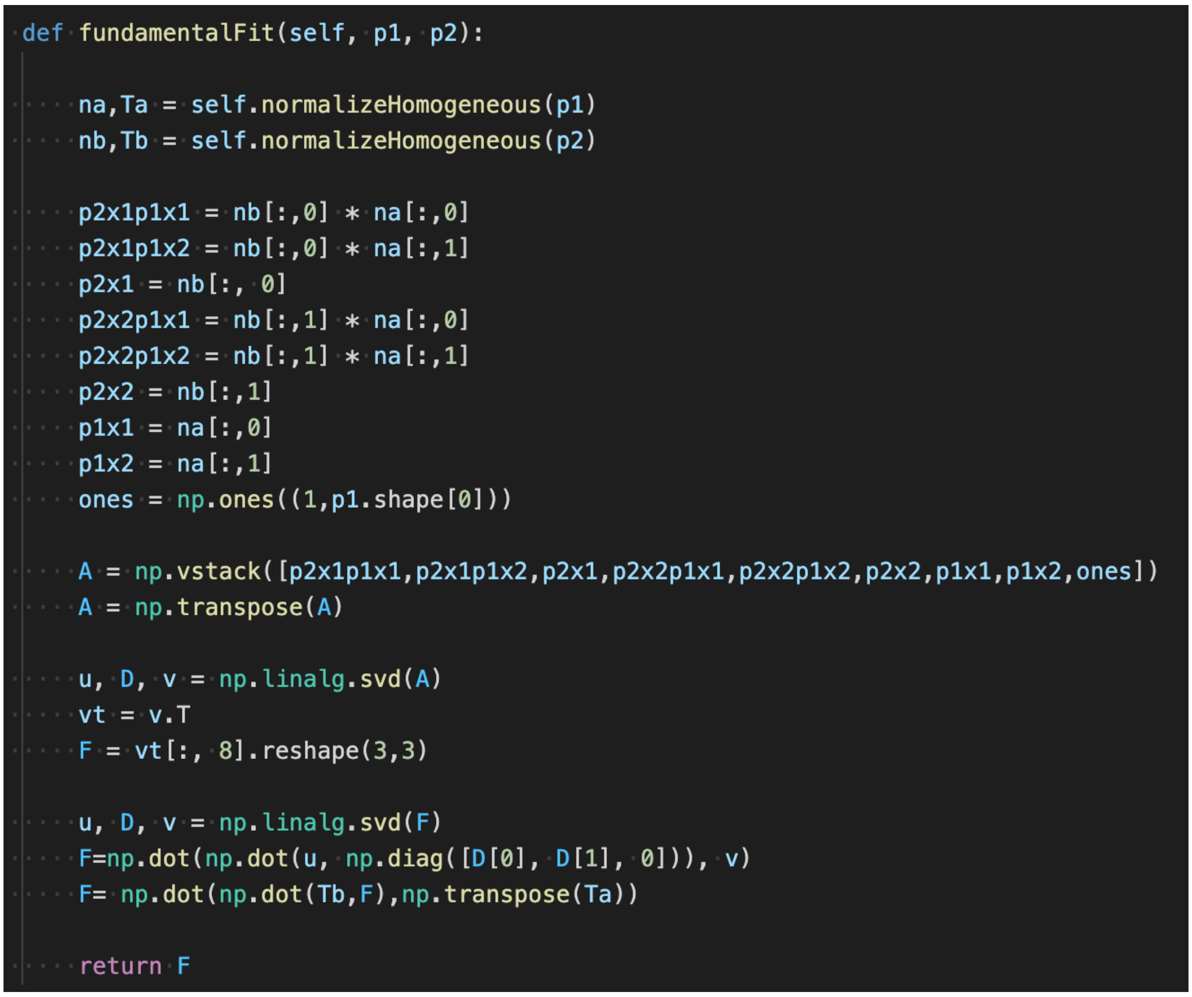
Step 2: Estimate fundamental matrix with RANSAC (normalized eight-point algorithm)

- The fundamental matrix is defined as $X'FX = 0$, where X and X′ are got from step 1. We sample eight points to obtain a fundamental matrix F.

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0$$

- The above equation turns to solve a linear system. We use Singular Value Decomposition (SVD) to achiece the fundamental matrix F.

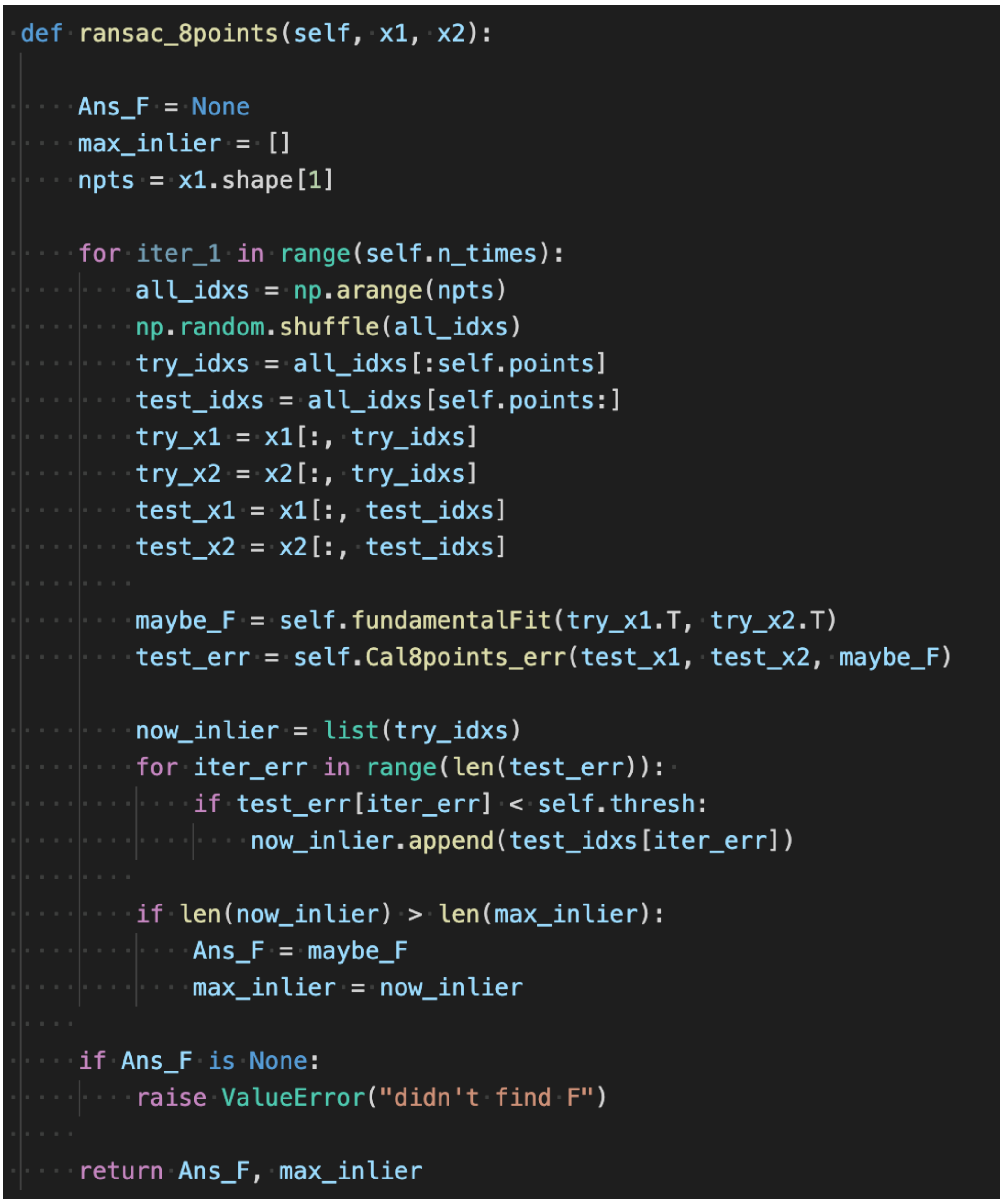
$$\mathbf{Af} = \begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = \mathbf{0}$$



```python
def fundamentalFit(self, p1, p2):

    na,Ta = self.normalizeHomogeneous(p1)
    nb,Tb = self.normalizeHomogeneous(p2)

    p2x1p1x1 = nb[:,0] * na[:,0]
    p2x1p1x2 = nb[:,0] * na[:,1]
    p2x1 = nb[:, 0]
    p2x2p1x1 = nb[:,1] * na[:,0]
    p2x2p1x2 = nb[:,1] * na[:,1]
    p2x2 = nb[:,1]
    p1x1 = na[:,0]
    p1x2 = na[:,1]
    ones = np.ones((1,p1.shape[0]))

    A = np.vstack([p2x1p1x1,p2x1p1x2,p2x1,p2x2p1x1,p2x2p1x2,p2x2,p1x1,p1x2,ones])
    A = np.transpose(A)

    u, D, v = np.linalg.svd(A)
    vt = v.T
    F = vt[:, 8].reshape(3,3)

    u, D, v = np.linalg.svd(F)
    F=np.dot(np.dot(u, np.diag([D[0], D[1], 0])), v)
    F= np.dot(np.dot(Tb,F),np.transpose(Ta))

    return F
```

- Compute Sampson distance to get test error from rest of the points to find out the best fundamental matrix, and record the inliers.

$$d(u_i, v_i) = (u_i F v_i)^2[1/((Fu^T)^2 + (Fu^T)^2) + 1/((vF)^2 + (vF)^2)]$$

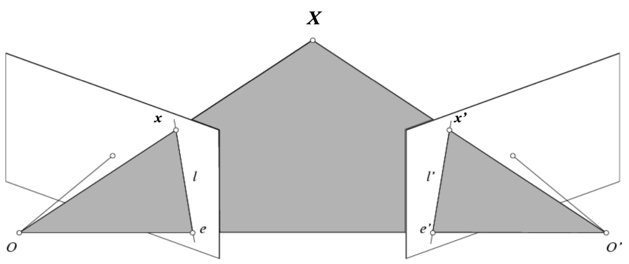


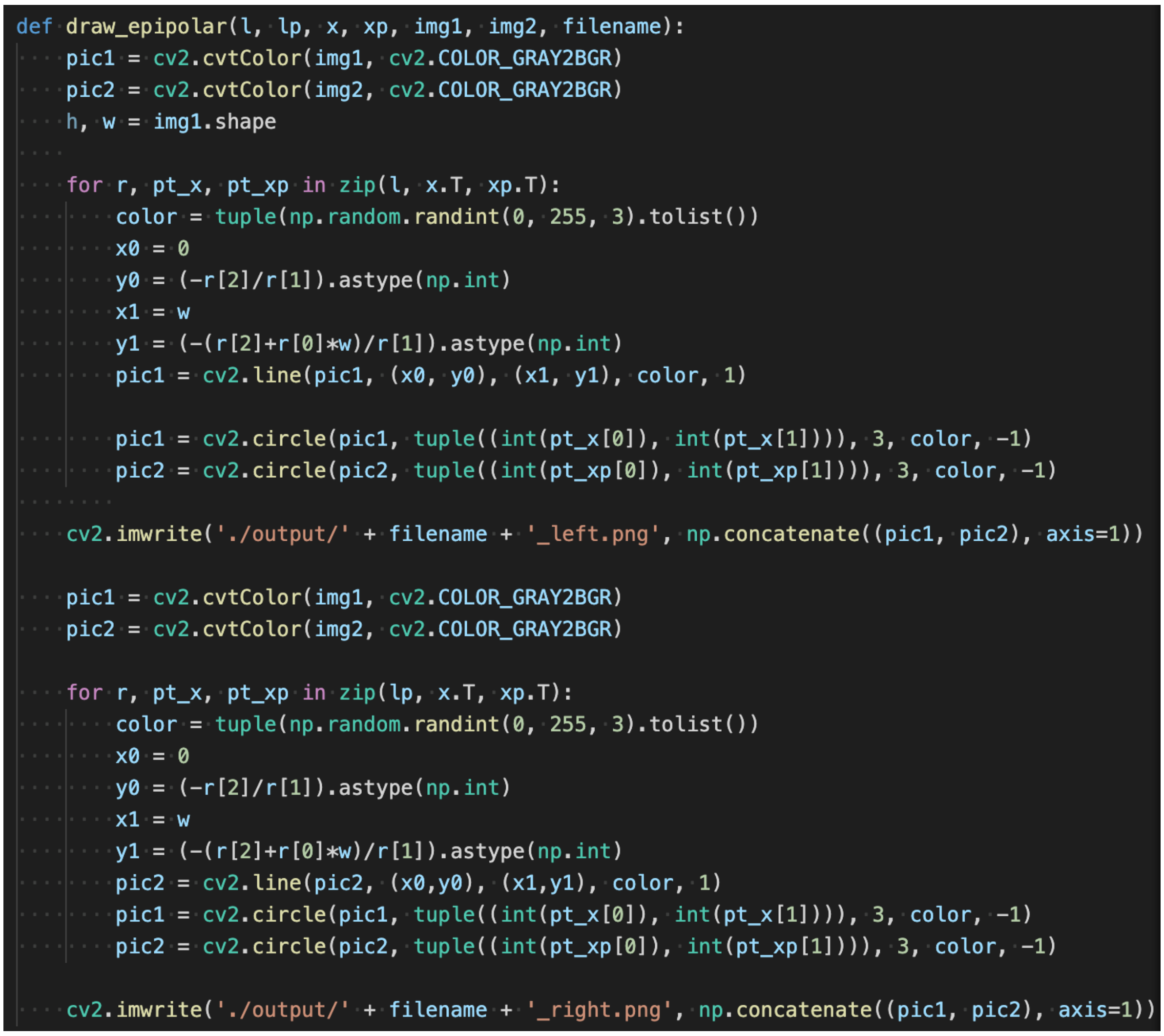
```python
def ransac_8points(self, x1, x2):

    Ans_F = None
    max_inlier = []
    npts = x1.shape[1]

    for iter_1 in range(self.n_times):
        all_idxs = np.arange(npts)
        np.random.shuffle(all_idxs)
        try_idxs = all_idxs[:self.points]
        test_idxs = all_idxs[self.points:]
        try_x1 = x1[:, try_idxs]
        try_x2 = x2[:, try_idxs]
        test_x1 = x1[:, test_idxs]
        test_x2 = x2[:, test_idxs]
        
        maybe_F = self.fundamentalFit(try_x1.T, try_x2.T)
        test_err = self.Cal8points_err(test_x1, test_x2, maybe_F)

        now_inlier = list(try_idxs)
        for iter_err in range(len(test_err)):
            if test_err[iter_err] < self.thresh:
                now_inlier.append(test_idxs[iter_err])
        
        if len(now_inlier) > len(max_inlier):
            Ans_F = maybe_F
            max_inlier = now_inlier
    
    if Ans_F is None:
        raise ValueError("didn't find F")
    
    return Ans_F, max_inlier
```

Step 3: Draw the keypoints and the epipolar lines

- Having the fundamental matrix, we can get the epipolar lines: $l = F^T x'$ and $l' = Fx$.

- There are three dimensions in l and l', which means the three variables of $ax + by + c$. We assign x0 and x1 first, and compute the corresponding y0 and y1.





```python
def draw_epipolar(l, lp, x, xp, img1, img2, filename):
    pic1 = cv2.cvtColor(img1, cv2.COLOR_GRAY2BGR)
    pic2 = cv2.cvtColor(img2, cv2.COLOR_GRAY2BGR)
    h, w = img1.shape
    
    for r, pt_x, pt_xp in zip(l, x.T, xp.T):
        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0 = 0
        y0 = (-r[2]/r[1]).astype(np.int)
        x1 = w
        y1 = (-(r[2]+r[0]*w)/r[1]).astype(np.int)
        pic1 = cv2.line(pic1, (x0, y0), (x1, y1), color, 1)

        pic1 = cv2.circle(pic1, tuple((int(pt_x[0]), int(pt_x[1]))), 3, color, -1)
        pic2 = cv2.circle(pic2, tuple((int(pt_xp[0]), int(pt_xp[1]))), 3, color, -1)
        
    cv2.imwrite('./output/' + filename + '_left.png', np.concatenate((pic1, pic2), axis=1))

    pic1 = cv2.cvtColor(img1, cv2.COLOR_GRAY2BGR)
    pic2 = cv2.cvtColor(img2, cv2.COLOR_GRAY2BGR)

    for r, pt_x, pt_xp in zip(lp, x.T, xp.T):
        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0 = 0
        y0 = (-r[2]/r[1]).astype(np.int)
        x1 = w
        y1 = (-(r[2]+r[0]*w)/r[1]).astype(np.int)
        pic2 = cv2.line(pic2, (x0,y0), (x1,y1), color, 1)
        pic1 = cv2.circle(pic1, tuple((int(pt_x[0]), int(pt_x[1]))), 3, color, -1)
        pic2 = cv2.circle(pic2, tuple((int(pt_xp[0]), int(pt_xp[1]))), 3, color, -1)

    cv2.imwrite('./output/' + filename + '_right.png', np.concatenate((pic1, pic2), axis=1))
```

Step 4: Get four possible essential matrix

- Assume the first camara matrix is $P_1 = [I|O]$.

- Use the known intrinsic matrix to compute $E = K'^T F K$, and then do SVD decomposition on E to get U, V.

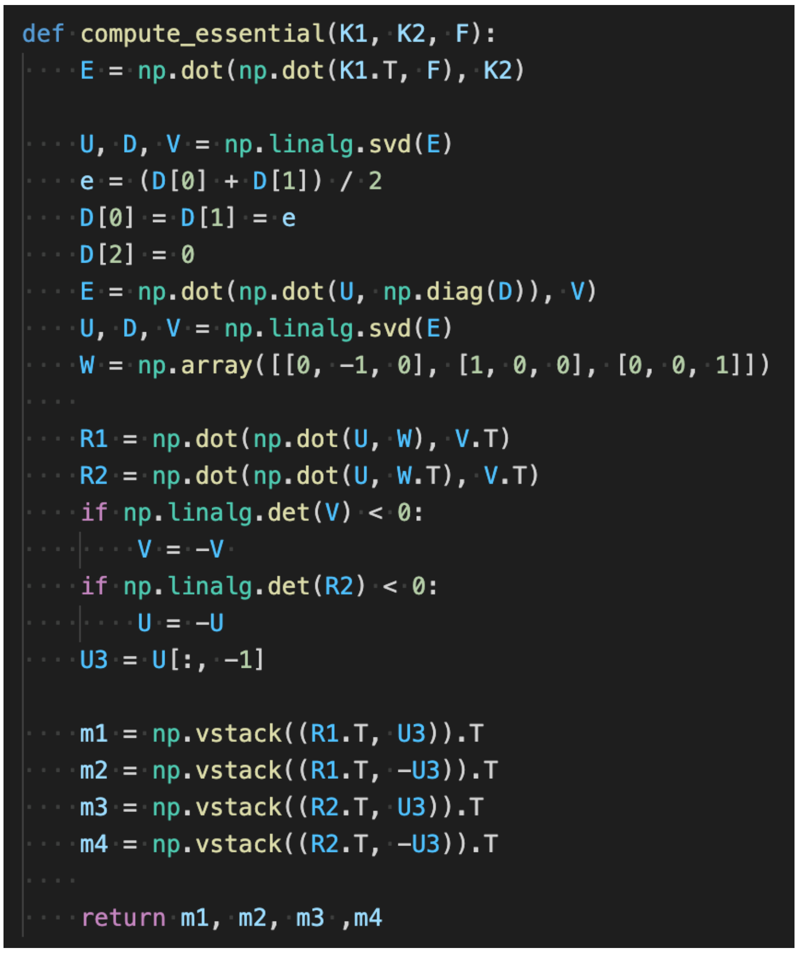
- Let essential matrix be $E = U diag(1,1,0) V^T$, and then so SVD on E again to get U and V.

- There are four possible essential matrix of the second camara matrix.

$$P_2 = [UWV^T \mid +u_3]$$
$$P_2 = [UWV^T \mid -u_3]$$
$$P_2 = [UW^TV^T \mid +u_3]$$
$$P_2 = [UW^TV^T \mid -u_3]$$

- where W is

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



```python
def compute_essential(K1, K2, F):
    E = np.dot(np.dot(K1.T, F), K2)

    U, D, V = np.linalg.svd(E)
    e = (D[0] + D[1]) / 2
    D[0] = D[1] = e
    D[2] = 0
    E = np.dot(np.dot(U, np.diag(D)), V)
    U, D, V = np.linalg.svd(E)
    W = np.array([[0, -1, 0], [1, 0, 0], [0, 0, 1]])
    
    R1 = np.dot(np.dot(U, W), V.T)
    R2 = np.dot(np.dot(U, W.T), V.T)
    if np.linalg.det(V) < 0:
        V = -V
    if np.linalg.det(R2) < 0:
        U = -U
    U3 = U[:, -1]

    m1 = np.vstack((R1.T, U3)).T
    m2 = np.vstack((R1.T, -U3)).T
    m3 = np.vstack((R2.T, U3)).T
    m4 = np.vstack((R2.T, -U3)).T
    
    return m1, m2, m3 ,m4
```

Step 5: Apply triangulation to get 3D points and find out the best essential matrix

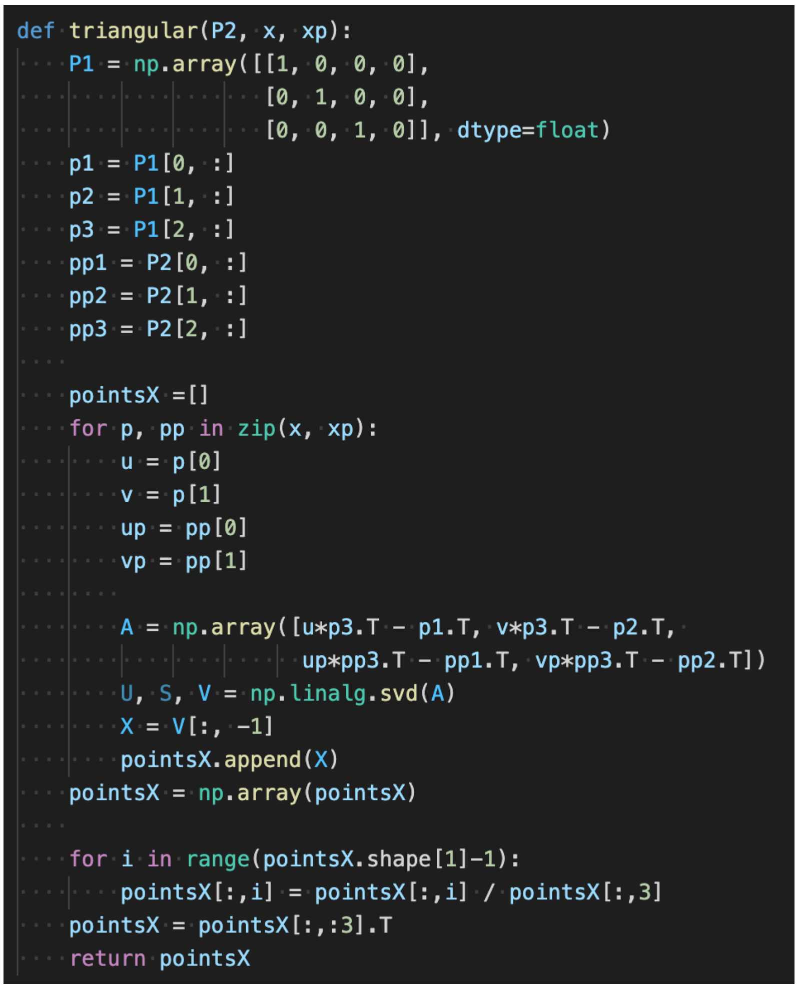
- With four possible essential matrix, we use triangulation to convert image keypoints into 3D coordinate.

$$\mathbf{A}X = 0 \quad A = \begin{bmatrix} u\mathbf{p}_3^\top - \mathbf{p}_1^\top \\ v\mathbf{p}_3^\top - \mathbf{p}_2^\top \\ u'\mathbf{p}'^\top_3 - \mathbf{p}'^\top_1 \\ v'\mathbf{p}'^\top_3 - \mathbf{p}'^\top_2 \end{bmatrix}$$

- where

$$x = w\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \qquad x^{'} = w\begin{bmatrix} u^{'} \\ v \\ 1 \end{bmatrix} \qquad P = \begin{bmatrix} p_1^T \\ p_2^T \\ p_3^T \end{bmatrix} \qquad P^{'} = \begin{bmatrix} p_1^{'T} \\ p_2^{'T} \\ p_3^{'T} \end{bmatrix}$$

$$x = PX \qquad x^{'} = P^{'}X$$



```
def triangular(P2, x, xp):
    P1 = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0],
                   [0, 0, 1, 0]], dtype=float)
    p1 = P1[0, :]
    p2 = P1[1, :]
    p3 = P1[2, :]
    pp1 = P2[0, :]
    pp2 = P2[1, :]
    pp3 = P2[2, :]

    pointsX =[]
    for p, pp in zip(x, xp):
        u = p[0]
        v = p[1]
        up = pp[0]
        vp = pp[1]

        A = np.array([u*p3.T - p1.T, v*p3.T - p2.T,
                      up*pp3.T - pp1.T, vp*pp3.T - pp2.T])
        U, S, V = np.linalg.svd(A)
        X = V[:, -1]
        pointsX.append(X)
    pointsX = np.array(pointsX)

    for i in range(pointsX.shape[1]-1):
        pointsX[:,i] = pointsX[:,i] / pointsX[:,3]
    pointsX = pointsX[:,:3].T
    return pointsX
```

- Then, calculate the number of inliers using the four essential matrix seperately, in order to get the best one with the most inliers.

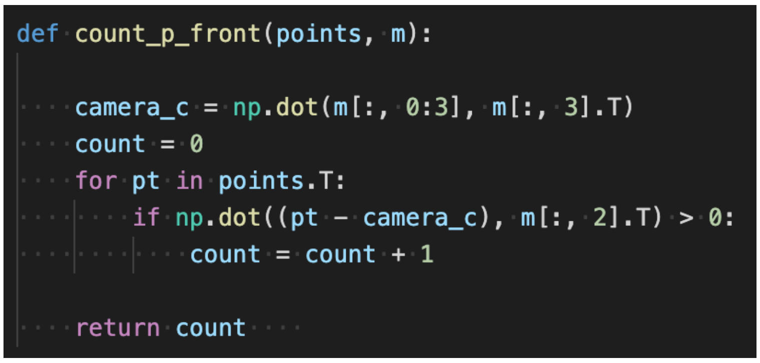
    - Compute the camara center C

$$\begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \Leftrightarrow \mathbf{C} = \begin{bmatrix} X_{world} \\ Y_{world} \\ Z_{world} \end{bmatrix} = R^{\top}\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - R^{\top}t = -R^{\top}t$$

    - Compute the angle between camera center to point and the camera's view direction

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \Leftrightarrow \left(R^{\top}\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - R^{\top}t\right) - (\mathbf{C}) = \left(R(3,:)^{\top} - R^{\top}t\right) - \left(-R^{\top}t\right) = R(3,:)^{\top}$$

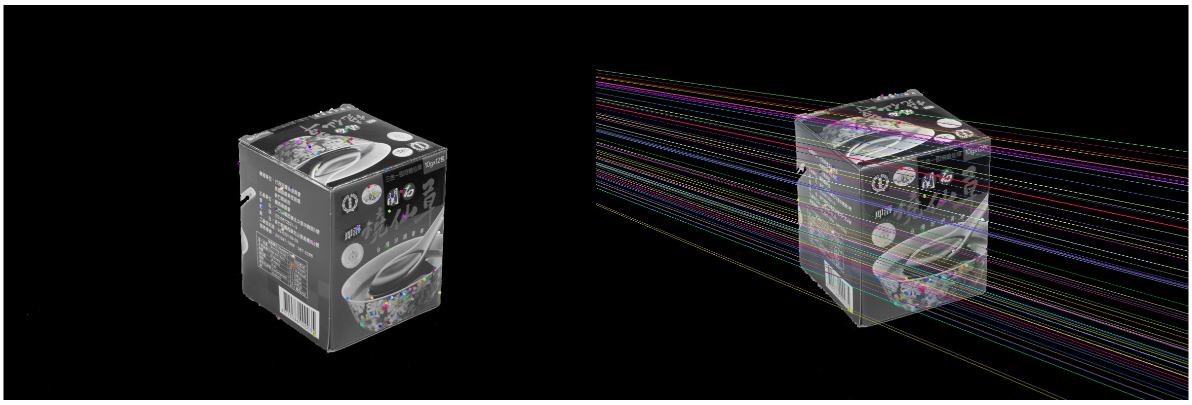
- If $(X - C) \cdot R(3,:)^T > 0$, this keypoint is in front of the camara, so it is an inlier.

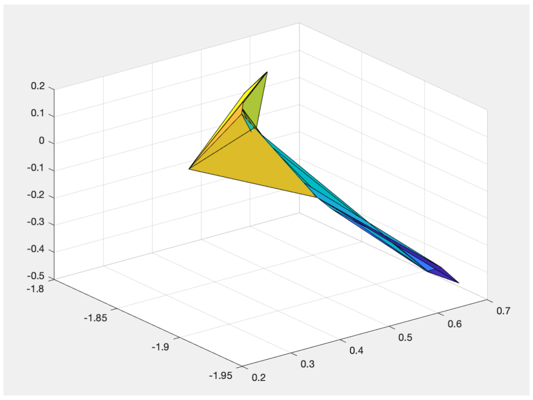


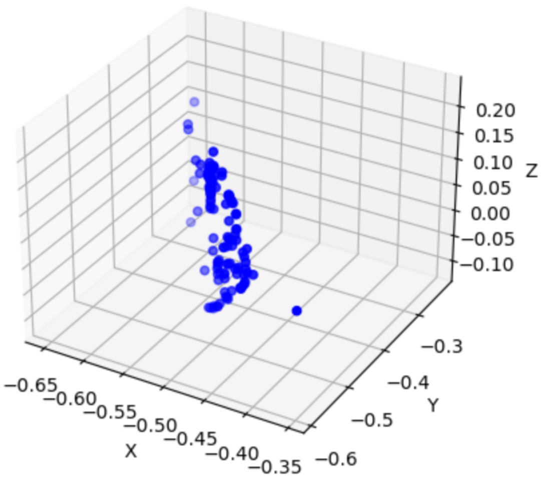
```python
def count_p_front(points, m):

    camera_c = np.dot(m[:, 0:3], m[:, 3].T)
    count = 0
    for pt in points.T:
        if np.dot((pt - camera_c), m[:, 2].T) > 0:
            count = count + 1

    return count
```

Experimental Results

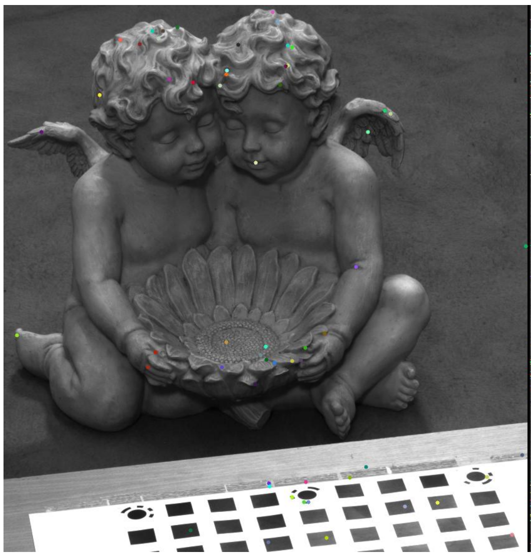
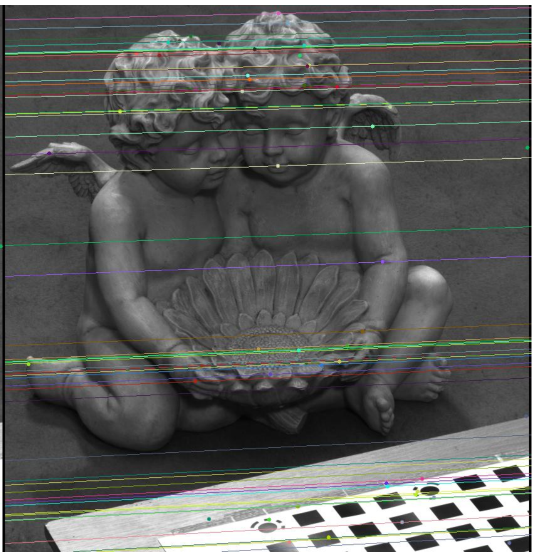
1. Mesona

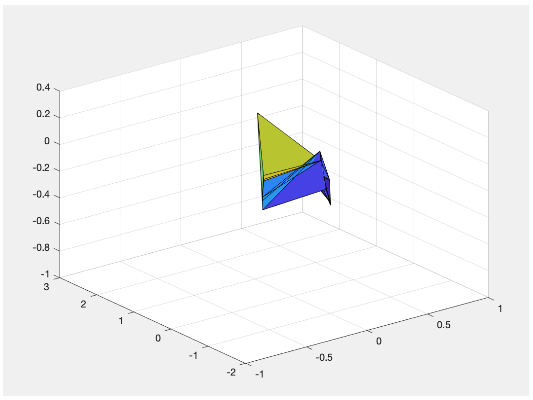


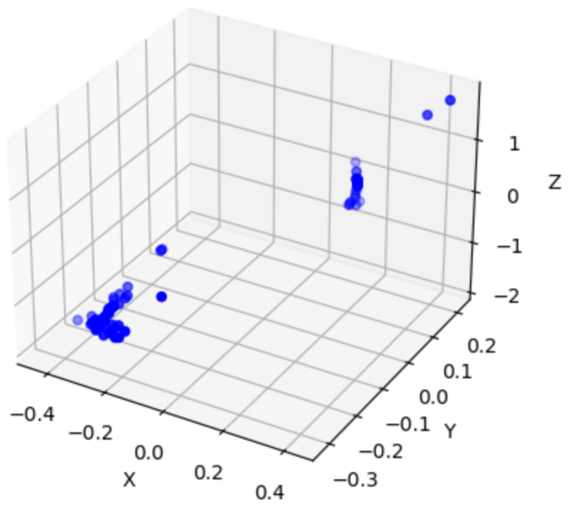




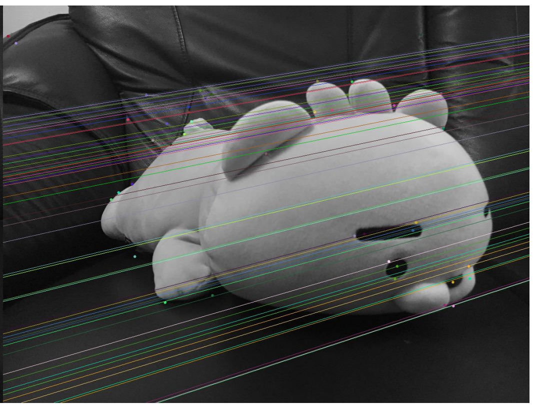
2. Statue

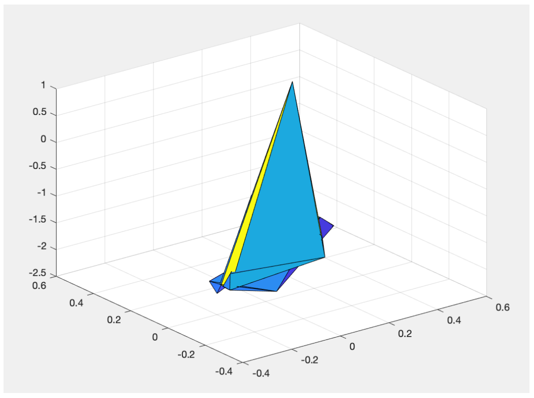



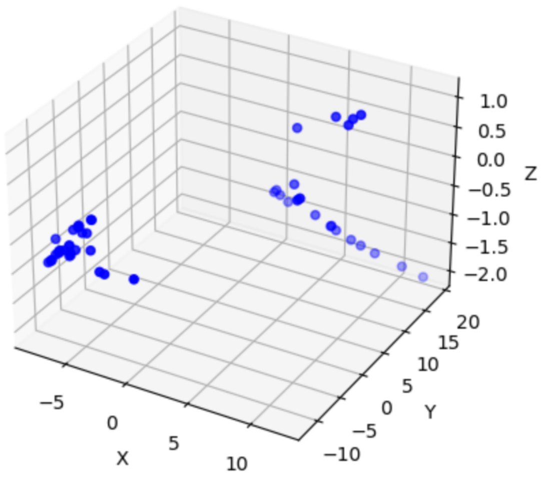




3. Our image







Discussion

- We try to increase the number of keypoints using in eight-point algorithm, and we find that it may give more robust result.
- While we drawing the 3D model using Matlab, we are confusing if the result is correct.

Conclusion

In this project, we implement Structure from Motion (SfM) to estimate 3D structure using two 2D images. We find the correspondences between images first, and then calculate the essential matrix to convert 2D keypoints into 3D coordinate. After finishing this assignment, we know more about the concept of this technique and understand the details of every step further.

Work assignment plan between team members

- Coding: 吳子涵
- Experiment: 余家宏、吳子涵、曹芳驊
- Report: 余家宏、吳子涵、曹芳驊