PA2 Kornkamol Anasart 20211182

(This assignment was written in Python 3.6 All requirements in README.md file)

Disclaimer: I tried to not use open function, but vlfeat for SIFT extraction and feature matching is not working on python, so I use cv2.ORB feature extraction and feature matching from OPENCV. And open3D to write PLY file.

Let's me introduce my code, my code separated into 2 python files including fundamental_m.py and triangulation.py. Let's look at fundamental_m.py

Overview of finding fundamental matrix:

Import image → feature extraction → feature matching → RANSAC [fivepoint_calibration]

```
39  if __name__ == '__main__':
40      # import image
41      imga = image('dataset/twoview/sfm01.jpg')
42      imgb = image('dataset/twoview/sfm02.jpg')
43      # feature extraction
44      kpa, da, kpb, db = ORB(imga, imgb, num_f=2000)
45      # matching
46      matches, pts_a, pts_b = match(da, db, kpa, kpb)
47      # img_corr = plotCorr(imga, imgb, pts_a, pts_b)
48
49      # RANSAC with five point
50      a_inlier, b_inlier, final_num_inlier, E = ransac(most_inliers=235, itr=10000,
51                                                       threshold=0.05, pts_a=pts_a, pts_b=pts_b)
52
```

Fig1: Overview code for finding fundamental matrix

Detail:

1. Import images in greyscale format into variable imga [sfm01.jpg] and imgb [sfm02.jpg] which we called image A and image B. Image A is based image for viewing point.

```
4   def image(path):
5       img = cv2.imread(path)
6       img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
7       # img = img.astype(np.float32) / 255
8       return img
```

Fig2: image function input is path of an image and output is greyscale image

2. Feature extraction part as I said I cannot use vlfeat in python3 so I switched to OpenCV ORB function. Fig3, shown my code used cv2.ORB_create to create feature extraction which compute into descriptor and keypoints from both image A and image B. ‘num_f ‘ is number of feature that we want

```
10  def ORB(imga, imgb, num_f):
11      orb = cv2.ORB_create(nfeatures=num_f)
12      kp_a, d_a = orb.detectAndCompute(imga, None)
13      kp_b, d_b = orb.detectAndCompute(imgb, None)
14      return kp_a, d_a, kp_b, d_b
```

Fig3: Feature extraction function

3. Once we got keypoints and descriptor from feature extraction function. We use cv2.BFMatcher for Brute-force matcher constructor. But not all matched point is good, so we need to set threshold for distance of matched points to find good matched following OpenCV document tutorial .(given image, I used 1.2, my images I used 1.1). And extract point coordinates (x, y) on image A and image B. The result of our possible mathcing points is 296 points.

```
28  def match(da, db, kpa, kpb):
29      bf = cv2.BFMatcher()
30      matches = bf.knnMatch(da, db, k=2)
31      good_matches = []
32      for m,n in matches:
33          if m.distance < n.distance/1.2:
34              good_matches.append([m])
35      pts_a = []
36      pts_b = []
37      for i in good_matches:
38          pts_a.append(kpa[i[0].queryIdx].pt)
39          pts_b.append(kpb[i[0].trainIdx].pt)
40      pts_a = np.array(pts_a).astype(np.int)
41      pts_b = np.array(pts_b).astype(np.int)
42      return good_matches, pts_a, pts_b
43
```

Fig4: matching fucntion

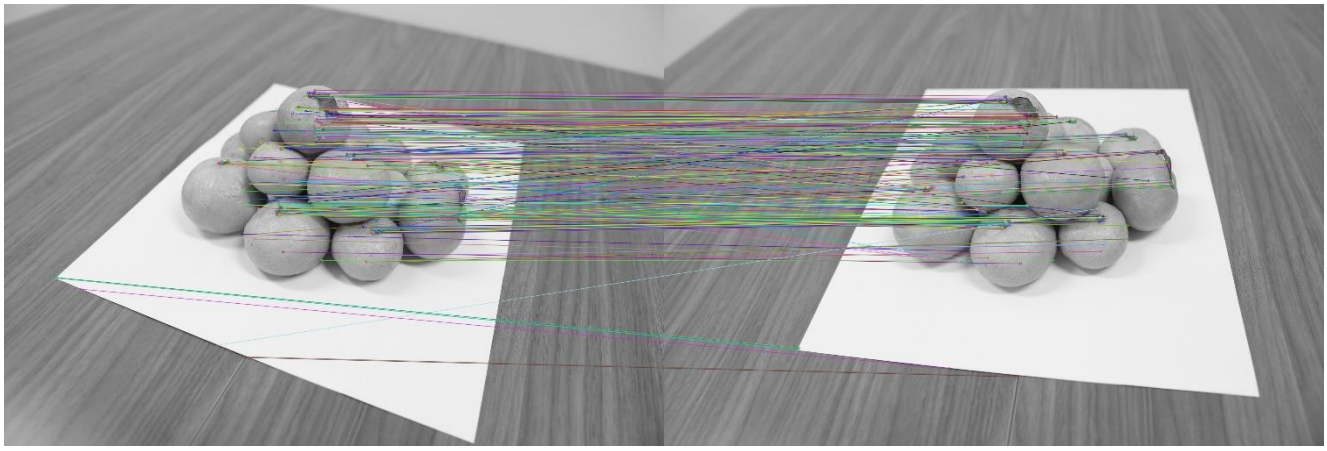

Fig5: Matching point using cv.drawKeypoints

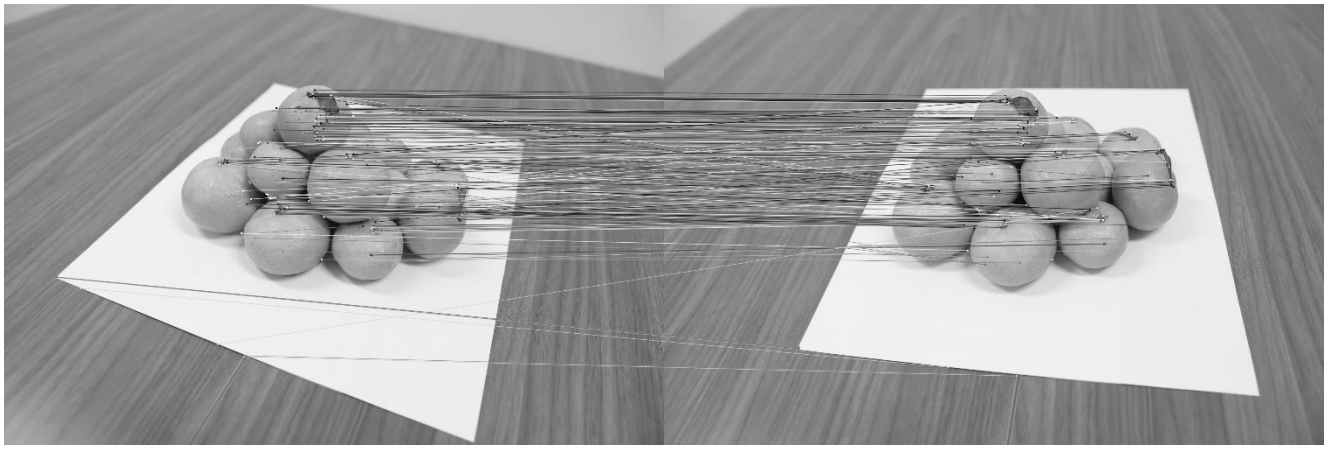

Fig6: Plot matching point coordinates (x, y) extracted from keypoints

4. Find fundamental matrix using fivepoint calibration. I implemented fivepoint calibration from MATLAB version into Python version which I had already compared result from MATLAB version and python version. So, if you investigate five-point calibration code, the function wants point coordinates in size [5, 3]. So, this is mean it needs homogeneous coordinate [x, y, 1]. So, we need to make it into homogenous coordinate first.

   The concept is we randomly choose 5 matched coordinates from image A and B. And put into fivepoint calibration function and run it multiple time in RANSAC to find best fundamental matrix which have maximum inlier. So, we need to set parameter*, 'most inliers'* for setup criteria of number of inlier points which is a greater number of inliers, you will get better fundamental matrix, *'threshold'* is number of error or epipolar distance between epipolar line and random point coordinates. *'iteration, itr'* is how many times that you want to run RANSAC. *'pts_a'* and *'pts_b'* is matched point coordinates on image A and B.

   Fivepoint calibration function give more than one fundamental matrix in one time so we add another for loop to extract fundamental matrix and change into size [3x3].

   We also set empty variable a_inlier and b_inlier to store inlier point coordinates. Another for loop for extract one by one coordinates to calculate error or distance between coordinates and epipolar line [err = $Q1^{t}FQ2$]. And we add condition if error is less than our pre define threshold that point is going to be stored in a_inlier and b_inlier.

```python
def ransac(most_inliers, itr, threshold, pts_a, pts_b):
    pts_ah = np.hstack((pts_a, np.ones((pts_a.shape[0], 1))))
    pts_bh = np.hstack((pts_b, np.ones((pts_b.shape[0], 1))))
    row, col = pts_ah.shape
    for i in range(itr):
        Evec = fivepoint(pts_ah[np.random.randint(0, row, size=5), :],
                         pts_bh[np.random.randint(0, row, size=5), :])
        for j in range(0, Evec.shape[1]):
            a_inlier = []
            b_inlier = []
            numinlier = 0
            E = Evec[:, j].reshape((3, 3))
            for p in range(row):
                Q1 = np.array(pts_ah[p, :]).reshape((3, 1))
                Q2 = np.array(pts_bh[p, :]).reshape((3, 1))
                err = np.dot(np.dot(Q1.T, E), Q2)
                err = np.abs(err)
                if err < threshold:
                    numinlier += 1
                    a_inlier.append(pts_ah[p, :])
                    b_inlier.append(pts_bh[p, :])
            if numinlier > most_inliers:
                a_inlier_final = a_inlier
                b_inlier_final = b_inlier
                num_inlier = numinlier
                E_final = E
    if a_inlier_final>0:
        return np.array(a_inlier_final), np.array(b_inlier_final), num_inlier, E_final
    else:
        return print('run RANSAC again')
```

Fig7: RANSAC [fivepoint calibration]

Note from my experiment: How to do experiment, increase most_inlier gradually such as 50 to 100, 200. You will discover that when you set large of 'most_inlier', you need to run RANSAC multiple time to find fundamental matrix that meet your criteria.

```
In[7]: print(F)
[[ 2.02391165e-07  1.04932860e-06 -6.26143758e-04]
 [ 7.96431767e-08  1.03322039e-06 -4.95849537e-04]
 [-3.41216003e-04 -1.64201984e-03  9.99998275e-01]]
```

Fig8: My best fundamental matrix with 241 inlier points from 296 possible matched points.

After we obtain best fundamental matrix. We use fundamental matrix to find 3D point coordinates.

1. Compute essential matrix: $E = K_1^t F K_2$ from : intrinsic matrix K, which K1 = K2 because it's same camera.

```
46      # essential matrix E = K1.T.*F.*K2
47      E = np.dot(np.dot(K.T, F), K)
```

Fig9: compute essential matrix

2. Decompose essential matrix into K[R|t] which is R = UWVt and t = u3 = last column of U. So we use np.linalg.svd to obtain U and Vt. (R is rotation matrix and t is translation matrix)

```
49      # P' = K[R|T]
50      # R = UWVt
51      # so we find U, Vt first: svd(E) = Udiag(1,1,0)V.T
52      U, S, Vt = np.linalg.svd(E)
```

Fig:10 SVD computation

3. There is 4 possible way of camera's position which give different rotation matrix. I assigned into case1 P' = [UWVt | +u3], case2 P' = [UWVt | -u3], case3 P' = [UWtVt | +u3], case4 P' = [UWtVt | -u3]
P, P' is camera matrix of image A and B which P = [I | 0], W = [[0, -1, 0], [1, 0, 0], [0, 0, 0]]. So now we have 4 camera matrix Pb1(P'1), Pb2 (P'2), Pb3(P'3), Pb4 (P'4).

```
58      # case 1 and 2
59      R1 = np.dot(np.dot(U, W), Vt)
60      # case 3 and 4
61      R2 = np.dot(np.dot(U, Wt), Vt)
62      # last col of U = u3 == translation [t]X
63      t = U[:, 2].reshape(3,1)
64
65      # case 1 P' = [UWVt | +u3]
66      Pb1 = np.dot(K, np.hstack((R1, t)))
67      # case 2 P' = [UWVt | -u3]
68      Pb2 = np.dot(K, np.hstack((R1, -t)))
69      # case 3 P' = [UWtVt | +u3]
70      Pb3 = np.dot(K, np.hstack((R2, t)))
71      # case 4 P' = [UWtVt | -u3]
72      Pb4 = np.dot(K, np.hstack((R2, -t)))
73
74      # P of img A  P = [I | 0]
75      Pa = np.eye(3,4)
```

Fig11: 4 possible camera's positions

4. Triangulation, finding 3D point (X) from (x, y). We need to calculate matrix A. from fig 12. And fin matrix B to solve AX = b. Dot product between [I | (x,y)] and P[:, 2] = R

$$\mathbf{A} = \begin{bmatrix} x\mathbf{p}^{3\mathrm{T}} - \mathbf{p}^{1\mathrm{T}} \\ y\mathbf{p}^{3\mathrm{T}} - \mathbf{p}^{2\mathrm{T}} \\ x'\mathbf{p}'^{3\mathrm{T}} - \mathbf{p}'^{1\mathrm{T}} \\ y'\mathbf{p}'^{3\mathrm{T}} - \mathbf{p}'^{2\mathrm{T}} \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & x \\ 0 & -1 & y \end{bmatrix} \begin{bmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & p_{22} \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & x' \\ 0 & -1 & y' \end{bmatrix} \begin{bmatrix} p'_{00} & p'_{01} & p'_{02} \\ p'_{10} & p'_{11} & p'_{12} \\ p'_{20} & p'_{21} & p'_{22} \end{bmatrix}$$

Fig12 Matrix A equation

```python
# triangulate
# AX = 0
# X = (x, y, z, 1) ** 3D point we need. So Ax = b
def triangulation(inpts_a, inpts_b, Pa, Pb):
    A = np.zeros((4, 3))
    b = np.zeros((4, 1))

    # 3D points array
    X = np.zeros((3, len(inpts_a)))
    for i in range(len(inpts_a)):
        temp1 = -np.eye(2, 3)
        temp2 = temp1
        temp1[:, 2] = inpts_a[i, :]
        temp2[:, 2] = inpts_b[i, :]
        # find matrix A
        A[0:2, :] = np.dot(temp1, Pa[0:3, 0:3])
        A[2:4, :] = np.dot(temp2, Pb[0:3, 0:3])
        # find matrix B
        b[0:2, :] = temp1.dot(Pa[0:3, 3:4])
        b[2:4, :] = temp2.dot(Pb[0:3, 3:4])
        b *= -1
        # Solve for X vector
        cv2.solve(A, b, X[:, i:i + 1], cv2.DECOMP_SVD)
    return np.hstack((X.T, np.ones((len(inpts_a), 1))))
```

Fig13: Triangulation

5. Choose camera position. (I choose it manually)

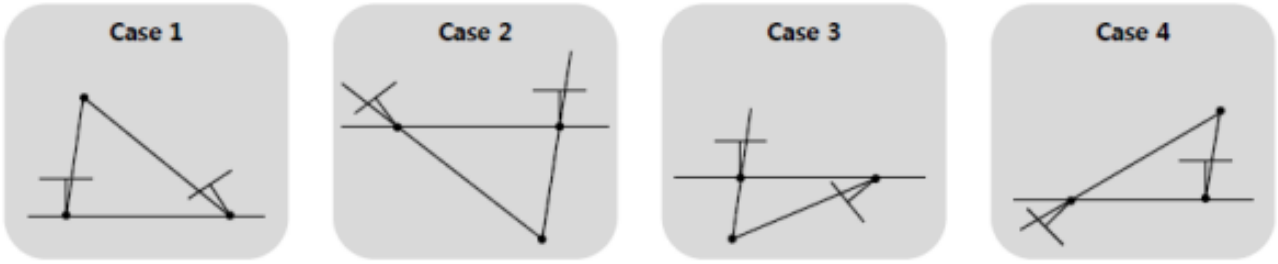

Fig 14: 4 case possible camera positions

Case1: P = [I |0], P' = Pb1, X = X1, depth of point (z coordinate) should be all positive.

Fig14: Camera's position

```
max depth of point in front of camera A: -5.084921218235732e-05
min depth of point in front of camera A: -0.00039140576594980315
max depth of point in front of camera B: -0.7418265262112684
min depth of point in front of camera B: -0.9007400027841288
```

Fig15: case 1

Case 2: P = [I|0], P' = Pb2, X = X2, depth of point should be in range positive and negative

```
max depth of point in front of camera A: 0.00039140576594980315
min depth of point in front of camera A: 5.084921218235732e-05
max depth of point in front of camera B: 0.9007400027841288
min depth of point in front of camera B: 0.7418265262112684
```

Fig16: case2

Case 3: P=[I|0], P' = Pb3, X=X3, depth of points at camera B should be negative

```
max depth of point in front of camera A: 0.0005537396988492477
min depth of point in front of camera A: 6.181902959341357e-05
max depth of point in front of camera B: -0.8367053187458333
min depth of point in front of camera B: -1.2724653414294609
```

Fig 17: case 3

Case 4: P=[I|0], P' = Pb4, X=X4, depth of points at camera B should be positive but camera A should be in range of negative and positive

```
max depth of point in front of camera A: -6.181902959341357e-05
min depth of point in front of camera A: -0.0005537396988492477
max depth of point in front of camera B: 1.2724653414294609
min depth of point in front of camera B: 0.8367053187458333
```

Fig 18: case 4

In conclusion, the camera position is case 3. How we calculate it? We calculate following this equation Fig 19.

$$\mathbf{X} = (X, Y, Z, 1)^T, P = [M|\mathbf{p}_4]$$
$$P\mathbf{X} = w(x, y, 1)^T$$
$$\text{depth}(\mathbf{X}; P) = \frac{\text{sign}(\det M)w}{T\|\mathbf{m}_3\|}$$

Fig 19: Calculate depth of point

```
PXa = np.dot(Pa, X4.T)
w = PXa/inpts_ah.T
M = Pa[:, :-1]
detm = np.linalg.det(M)
sign = np.sign(detm)
m3 = M[-1, :]
nm3 = np.linalg.norm(m3)
d = w*sign / nm3*t
maxd = np.max(d[2, :])
print('max depth of point in front of camera A:',maxd)
mind = np.min(d[2, :])
print('min depth of point in front of camera A:',mind)
```

Fig 20: Calculate depth of point

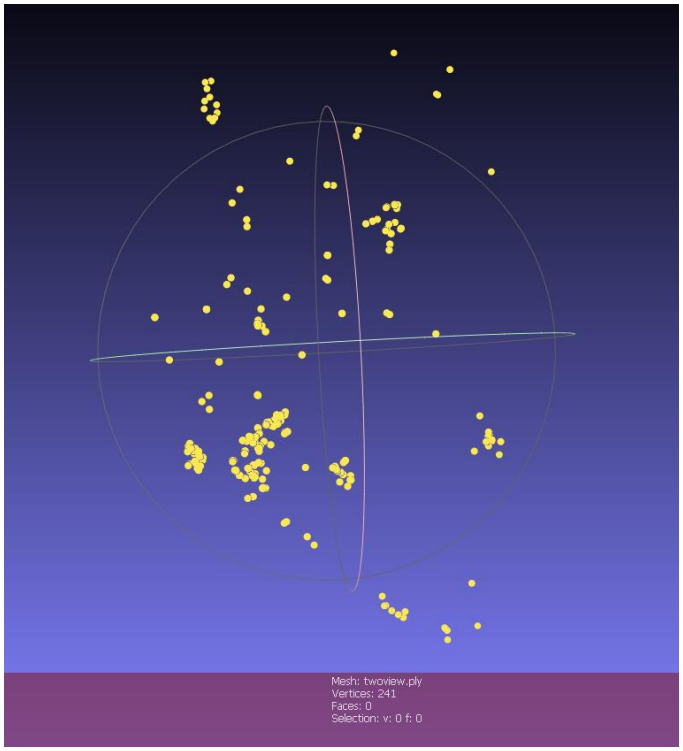

Fig 21: plot 3D points

My image dataset: (Camera position: Case1) Possible match point is 313 points and inlier points is 244.

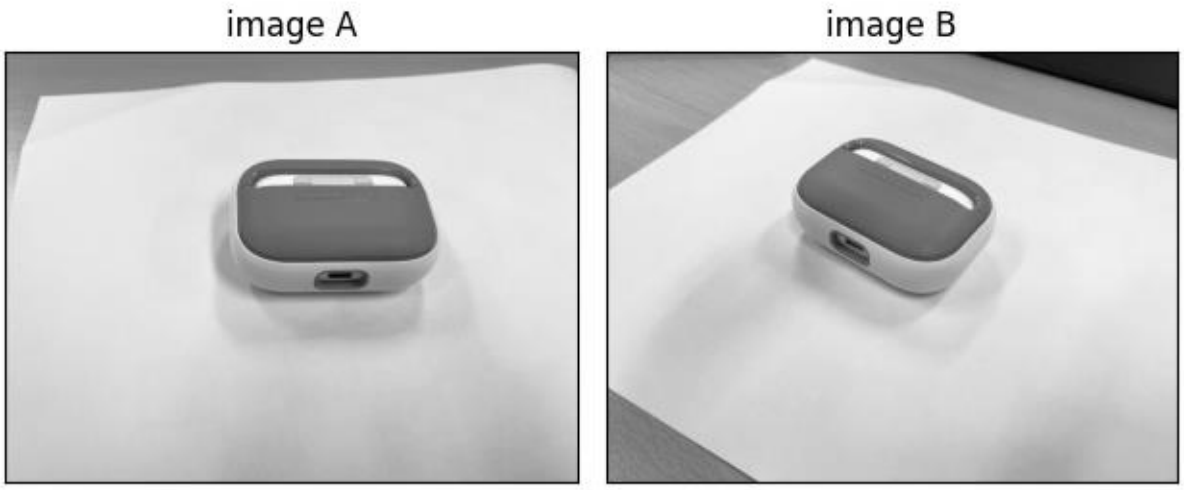

Fig 22: My image dataset for two view

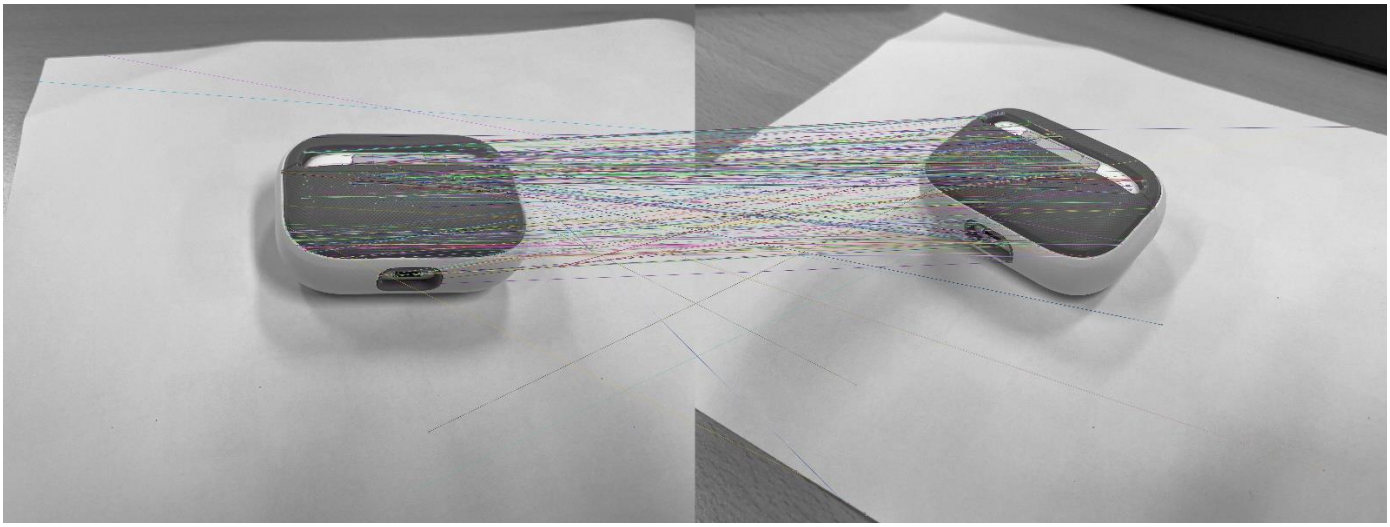

Fig 23: Matching Keypoints

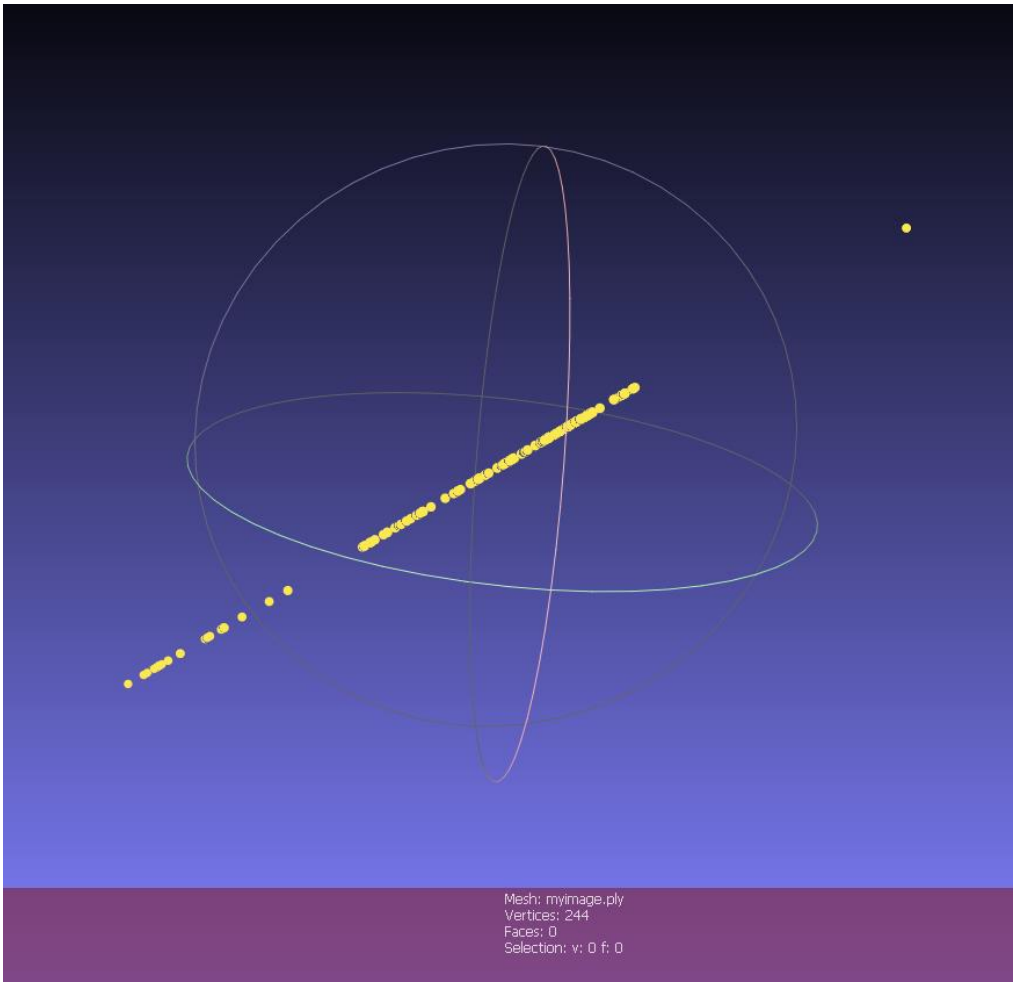

Fig 24: Plot 3D point from my images

```
[3.16984458e+03, 0.00000000e+00, 0.00000000e+00],
 [0.00000000e+00, 3.17576625e+03, 0.00000000e+00],
 [2.00258875e+03, 1.47435045e+03, 1.00000000e+00]])
```

Fig 25: My camera extrinsic matrix in file K_m.txt

```
[[ 6.87906333e-09,  1.72159752e-07, -1.69654977e-04],
 [ 2.56758022e-08,  6.42578725e-07, -6.33229764e-04],
 [-4.05473415e-05, -1.01476319e-03,  9.99999269e-01]])
```

Fig 26: My best fundamental matrix of my image dataset in file myF.npy