SIT789 – Robotics, Computer Vision and Speech Processing

Pass Task 6.1: 3D Vision

Objectives

The objectives of this lab include:

- Practising 3D projection in pin-hole models
- Practising camera pose estimation
- Practising 3D reconstruction with the structure-from-motion (sfm) approach

---

Tasks

1. 3D projection

In this section, we will implement the camera matrix for a pin-hole camera model and then will perform 3D projection (i.e., projecting a point in 3D space onto a 2D image).

Recall that the camera matrix P of a pin-hole camera can be written as P = K [R I t] (see Eq (2) in slide 6, in Lecture 6 slides). For simplicity, we initialise K, R, and t as follows.

```
import numpy as np

#setup camera with a simple camera matrix P
f = 100
cx = 200
cy = 200
K = np.array([[f, 0, cx], [0, f, cy], [0, 0, 1]])
I = np.eye(3)
t = np.array([[0], [0], [0]])
P = np.dot(K, np.hstack((I, t)))
```

We then define a 3D projection method as,

```
def project(P, X): #X is an array of 3D points
    x = np.dot(P, X)
    for i in range(3): #convert to inhomogeneous coordinates
        x[i] /= x[2]
    return x
```

To test the camera matrix P, we use the 3D point set in house.p3d supplied in OnTrack. This dataset is from http://www.robots.ox.ac.uk/~vgg/data/mview/.

You first need to download the file house.p3d into your working directory and read data from the file using the following code.

```
#load data
points_3D = np.loadtxt('house.p3d').T #T means tranpose
points_3D = np.vstack((points_3D, np.ones(points_3D.shape[1])))
```

To visualise the 3D data in house.p3d, we can perform.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

fig = plt.figure(figsize = [15,15])

ax = fig.gca(projection = "3d")
ax.view_init(elev=None, azim=None) #you can set elevation/azimuth with different values

ax.plot(points_3D[0], points_3D[1], points_3D[2], 'o')

plt.draw()
plt.show()
```

Now, we perform 3D projection for all the 3D points in house.p3d onto an image using the camera matrix P defined above.

```
#projection
points_2D = project(P, points_3D)

#plot projection
from matplotlib import pyplot as plt
plt.plot(points_2D[0], points_2D[1], 'k.')
plt.show()
```

You can change the camera matrix P and observe the results of 3D projection accordingly.

2. Camera pose estimation

In this section, we learn how to estimate the camera matrix P of a pin-hole camera given a set of 3D points and their projected 2D locations. Suppose that we are given points_3D as a set of 3D points and points_2D as their 2D locations projected on an image. Suppose that P is unknown and we want to estimate it.

In Eq (7) in slide 14, Lecture 6 slides, we have learnt how to build a matrix A that can be used for estimation of P. We have also learnt that we need at least 6 points to build the matrix A (see slide 17, Lecture 6 slides). However, points_3D contains more than 6 points. We can check this using the following commands.

```
print(points_2D.shape)
print(points_3D.shape)
```

To build the matrix A, we simply sample points_3D by taking the first 6 points from points_3D and their corresponding 2D locations from points_2D. The sampled 3D points and corresponding 2D locations are stored in points_3D_sampled and points_2D_sampled respectively.

```
n_points = 6
points_3D_sampled = points_3D[:,:n_points]
points_2D_sampled = points_2D[:,:n_points]
```

We now build the matrix A from points_3D_sampled and points_2D_sampled as follows.

```
A = np.zeros((2*n_points, 12), np.float32)
for i in range(n_points):
    A[2*i,:4] = points_3D_sampled[:,i].T
    A[2*i,8:12] = -points_2D_sampled[0,i] * points_3D_sampled[:,i].T
    A[2*i+1,4:8] = points_3D_sampled[:,i].T
    A[2*i+1,8:12] = -points_2D_sampled[1,i] * points_3D_sampled[:,i].T
```

Given A, our problem is to find p such that Ap = 0. However, we have learnt that solving such a linear system would result in trivial solution. We can double check this simply as,

```
from scipy import linalg
p = linalg.solve(A, np.zeros((12, 1), np.float32))
print(p)
```

As shown, all the elements of p are zero, i.e., trivial solution. To address this issue, instead of directly solving Ap = 0, we minimise ||Ap|| s.t. ||p|| = 1 using SVD (singular value decomposition) technique. In particular, we first decompose A as,

```
U, S, V = linalg.svd(A)
```

Results of the above command include a matrix U, a vector S, and a matrix V satisfying U.diag(S).V^T = A. Note that S is a sorted array of singular values with decreasing order. S contains 12 values and thus the S[11] is the smallest singular value. We can verify this using the following command.

```
minS = np.min(S)
conditon = (S == minS)
minID = np.where(conditon)
print('index of the smallest singular value is: ', minID[0])
```

We finally compute P_hat (an estimate of P) by performing:

- Taking the row of V (i.e., column of V^T) that corresponds to the smallest singular value (i.e., the last row)
- Express this column in a (3x4) matrix form
- Divide all elments of the matrix by the smallest singular value

```
P_hat = V[minID[0],:].reshape(3, 4) / minS
```

Now, we print both P and P_hat.

```
print(P)
print(P_hat)
```

P_hat does not seem to be close to P! That's ok! In fact, two different camera matrices may produce the same projected image due to the conversion from homogeneous to inhomogeneous. We now perform another test, that is, to take a 3D point, project this 3D point to 2D using both P and P_hat and then compare the projection results. To do so, we simply take the first point in points_3D_sampled and project it to 2D using P_hat.

```
x_P_hat = project(P_hat, points_3D_sampled[:, 0])
print(x_P_hat)
```

Now, we print the 2D location of that point from points_2D_sampled obtained by using P.

```
x_P = points_2D_sampled[:,0]
print(x_P)
```

For a quantitative evaluation, we measure the distance between x_P and x_P_hat for all points in points_3D. The smaller the distance is, the more accurate the estimation (of P) is.

```
x_P = points_2D
x_P_hat = project(P_hat, points_3D)

dist = 0
for i in range(x_P.shape[1]):
    dist += np.linalg.norm(x_P[:,i] - x_P_hat[:,i])
dist /= x_P.shape[1]
print(dist)
```

Your task is to vary n_points in the range [10%, 20%, ..., 100%] of the number of points in points_3D and measure the corresponding dist. What is the best value for n_points?

3. 3D reconstruction

In this section, we will experiment the structure-from-motion (sfm) approach for 3D reconstruction from two views with assumption that the intrinsic calibration matrix K is given. We will use the two images temple2A.png and temple2B.png supplied in OnTrack. These images come from the paper "A comparison and evaluation of multi-view stereo reconstruction algorithms" by Seitz et al. in CVPR 2006, and can be downloaded at https://vision.middlebury.edu/mview/. The images capture a scene (a temple) at two different viewpoints.

We also use some code from this book (with minor modifications to adapt with Python 3) including homography.py, ransac.py, and sfm.py. The code is provided in OnTrack and can also be found at https://github.com/jesolem/PCV. You are also recommended to refer to https://www.opensfm.org/ for further study on the sfm approach.

You first need to download the supplied homography.py, ransac.py, and sfm.py into your working directory and then import them to your notebook. You can read a description of the RANSAC algorithm (implemented in ransac.py) at https://en.wikipedia.org/wiki/Random_sample_consensus.

```
import homography
import sfm
import ransac
```

You also need to download the two image files temple2A.png and temple2B.png into your working directory. You then read the images from those files and extract SIFT keypoints and descriptors from them. SIFT keypoints and descriptors on temple2A.png are stored in kp1 and des1, and those on temple2B.png are stored in kp2 and des2, respectively.

```
import cv2 as cv

sift = cv.xfeatures2d.SIFT_create()

img1 = cv.imread('temple2A.png')
img1_gray = cv.cvtColor(img1, cv.COLOR_BGR2GRAY)
kp1, des1 = sift.detectAndCompute(img1_gray, None)

img2 = cv.imread('temple2B.png')
img2_gray = cv.cvtColor(img2, cv.COLOR_BGR2GRAY)
kp2, des2 = sift.detectAndCompute(img2_gray, None)
```

```
img1_kp = img1.copy()
img1_kp = cv.drawKeypoints(img1, kp1, img1_kp)

print("Number of detected keypoints in img1: %d" % (len(kp1)))

img2_kp = img2.copy()
img2_kp = cv.drawKeypoints(img2, kp2, img2_kp)

print("Number of detected keypoints in img2: %d" % (len(kp2)))

img1_2_kp = np.hstack((img1_kp, img2_kp))

plt.figure(figsize = (20, 10))
plt.imshow(img1_2_kp[:,:,::-1])
plt.axis('off')
```

We match descriptors in des1 and those in des2 using cv.BFMatcher as follows.

```
bf = cv.BFMatcher(crossCheck = True) #crossCheck=True: find consistent matches
matches = bf.match(des1, des2)
matches = sorted(matches, key = lambda x:x.distance)
print("Number of consistent matches: %d" % len(matches))
```

The best 20 matches are visualised as follows.

```
img1_2_matches = cv.drawMatches(img1, kp1, img2, kp2,
                                matches[:20],
                                None,
                                flags = cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

plt.figure(figsize = (20, 10))
plt.imshow(img1_2_matches[:,:,::-1])
plt.axis('off')
```

Next, we identify the corresponding matching keypoints and store them in arrays for later use. Note that high number of matches would lead to high computational complexity for the reconstruction. Therefore, in some cases, one can define an upper bound for the number of matches, e.g., 1000.

```
n_matches = min(len(matches), 1000)

kp1_array = np.zeros((2, n_matches), np.float32)
for i in range(n_matches):
    kp1_array[0][i] = kp1[matches[i].queryIdx].pt[0]
    kp1_array[1][i] = kp1[matches[i].queryIdx].pt[1]

kp2_array = np.zeros((2, n_matches), np.float32)
for i in range(n_matches):
    kp2_array[0][i] = kp2[matches[i].trainIdx].pt[0]
    kp2_array[1][i] = kp2[matches[i].trainIdx].pt[1]
```

We convert all the keypoints into homogeneous coordinates.

```
x1 = homography.make_homog(kp1_array)
x2 = homography.make_homog(kp2_array)
```

Suppose that the intrinsic calibration matrix K is given, and we fix P1=[I|0], we need to find P2 (see slides 49-50, Lecture 6 slides). For example, we define:

```
K = np.array([[2394,0,932], [0,2398,628], [0,0,1]])
P1 = np.array([[1,0,0,0], [0,1,0,0], [0,0,1,0]])
```

We normalise x1 and x2 using K^{-1}.

```
x1n = np.dot(linalg.inv(K), x1)
x2n = np.dot(linalg.inv(K), x2)
```

We estimate the essential matrix E. **Note**: this step takes time.

```
#estimate E with RANSAC
model = sfm.RansacModel()
E, inliers = sfm.F_from_ransac(x1n, x2n, model)
```

We then compute P2.

```
#compute camera matrices (P2 will be list of four solutions)
P2_all = sfm.compute_P_from_essential(E)

#pick the solution with points in front of cameras
ind = 0
maxres = 0
for i in range(4):
    #triangulate inliers and compute depth for each camera
    X = sfm.triangulate(x1n[:, inliers], x2n[:, inliers], P1, P2_all[i])
    d1 = np.dot(P1, X)[2]
    d2 = np.dot(P2_all[i], X)[2]
    s = sum(d1 > 0) + sum(d2 > 0)
    if s > maxres:
        maxres = s
        ind = i
        infront = (d1 > 0) & (d2 > 0)
P2 = P2_all[ind]
```

Finally, we reconstruct 3D points using the triangulation algorithm (see Eq (20) in slide 52, Lecture 6 slides).

```
#triangulate inliers and remove points not in front of both cameras
X = sfm.triangulate(x1n[:, inliers], x2n[:, inliers], P1, P2)
X = X[:, infront]
```

After applying the triangulation algorithm and removing points not in front of both cameras, we obtain a set of reconstructed 3D points, named X. This set contains less than 1000 points. We can check this by performing.

```
print(len(X[0]))
```

We visualise X using the following code.

```
#3D plot
fig = plt.figure(figsize = [20,20])

ax = fig.gca(projection = "3d")
ax.view_init(elev = -80, azim= -90)

ax.plot(X[0], X[1], X[2], 'o')

plt.draw()
plt.show()
```

Your task is to visualise the original points (computed using SIFT) on img2 and the image points (computed by projecting X with camera matrix P2) on img2.

Hint: After projecting X to x by using "project" method**,** you need to convert x into image coordinates by multiply K with x.

Submission instructions

1. Perform tasks required in Section 2 and 3.
2. Complete the supplied answer sheet with required results
3. Submit the answer sheet (.pdf) and code (.py) to OnTrack.