EX7task2

February 27, 2022

[ ]:
```
#task 2
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from scipy.io import loadmat
from utils import vgg_X_fromxP_lin, camcalibDLT

# Load the images (two views of the same scene)
im1 = np.array(Image.open('im1.jpg'))
im2 = np.array(Image.open('im2.jpg'))
# Load the camera matrices of the images and image coordinates for the corners,
# all of which can be found from the .mat-files of the working directory.
# Hint: loadmat

##--your-code-starts-here--##
P1 = loadmat("P1.mat")
P2 = loadmat("P2.mat")
P1 = P1["P1"]
P2 = P2["P2"]
coords = loadmat("cornercoordinates.mat")
##--your-code-ends-here--##

# Load image coordinates for the corners from coords-dictionary.

#--your-code-starts-here--##

x1 = coords["x1"]
x2 = coords["x2"]
y1 = coords["y1"]
y2 = coords["y2"]

#x1 = np.zeros((8, 1)) # replace me
#x2 = np.zeros((8, 1)) # replace me
#y1 = np.zeros((8, 1)) # replace me
#y2 = np.zeros((8, 1)) # replace me
#--your-code-ends-here--##
```

```
# Give labels for the corners of the shelf
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

# Define the 3D coordinates of the corners based on the known dimensions
ABCDEFGH_w = np.array([[758, 0, -295],
                       [0, 0, -295],
                       [758, 360, -295],
                       [0, 360, -295],
                       [758, 0, 0],
                       [0, 0, 0],
                       [758, 360, 0],
                       [0, 360, 0]])

# Visualize the corners in the images
plt.figure(1)
sp1 = plt.subplot(2, 1, 1)
sp2 = plt.subplot(2, 1, 2)
sp1.imshow(im1)
sp2.imshow(im2)
sp1.plot(x1, y1, 'c+', markersize=10)
sp2.plot(x2, y2, 'c+', markersize=10)
for i in range(np.size(x1)):
    sp1.annotate(labels[i], (x1[i], y1[i]), color='c', fontsize=20)
    sp2.annotate(labels[i], (x2[i], y2[i]), color='c', fontsize=20)

# Calibrate the cameras from 3D <-> 2D correspondences
P1t = camcalibDLT(np.hstack((ABCDEFGH_w, np.ones((8, 1)))),
                  np.hstack((x1, y1, np.ones((8, 1)))))
P2t = camcalibDLT(np.hstack((ABCDEFGH_w, np.ones((8, 1)))),
                  np.hstack((x2, y2, np.ones((8, 1)))))

# Visualize a 3D sketch of the shelf
edges = np.array([[0, 1], [0, 2], [2, 3], [1, 3], [0, 4], [4, 5], [1, 5], [4,␣
 ↪6], [2, 6], [3, 7], [6, 7], [5, 7]]).T
plt.figure(2)
ax = plt.axes(projection='3d')
plt.title('3D sketch of the shelf')
for i in range(np.shape(edges)[1]):
    ax.plot3D(ABCDEFGH_w[edges[:, i], 0], ABCDEFGH_w[edges[:, i], 1],␣
 ↪ABCDEFGH_w[edges[:, i], 2], 'gray')
for i in range(8):
    ax.text(ABCDEFGH_w[i][0], ABCDEFGH_w[i][1], ABCDEFGH_w[i][2], labels[i],␣
 ↪fontsize=20)

# Your task is to project the 3D corners in ABCDEFGH_w (currently␣
 ↪non-homogeneous) to
```

```
# images 1 and 2 using P1t and P2t, i.e. the calibrated camera matrices.
# Convert the projected points back to non-homogeneous form and store them into␣
 ↪the given variables below.
# Results are visualized using magenta lines.
##-your-code-starts-here-##
ABCDEFGH_w = np.hstack((ABCDEFGH_w, np.ones((8, 1))))  #horizontally column wise
x1_add = np.dot(P1t, ABCDEFGH_w.T)
x2_add = np.dot(P2t, ABCDEFGH_w.T)
cx1 = (x1_add[0, :] / x1_add[2, :])
cy1 = (x1_add[1, :] / x1_add[2, :])
cx2 = (x2_add[0, :] / x2_add[2, :])
cy2 = (x2_add[1, :] / x2_add[2, :])

#cx1 = np.zeros(np.shape(edges)[1])  # replace me
#cy1 = np.zeros(np.shape(edges)[1])  # replace me
#cx2 = np.zeros(np.shape(edges)[1])  # replace me
#cy2 = np.zeros(np.shape(edges)[1])  # replace me
##-your-code-stops-here-##

# Illustrate the edges of the shelf that connect its corners
for i in range(np.shape(edges)[1]):
    sp1.plot(cx1[edges[:, i]], cy1[edges[:, i]], 'm-')
    sp2.plot(cx2[edges[:, i]], cy2[edges[:, i]], 'm-')

# Compute a projective reconstruction of the shelf
# That is, triangulate the corner correspondences using the camera projection
# matrices which were recovered from the fundamental matrix
Xcorners = np.zeros((4, 8))
for i in range(8):
    imsize = np.array([[np.shape(im1)[1], np.shape(im2)[1]],
                       [np.shape(im1)[0], np.shape(im2)[0]]])
    u = np.array([[x1[i], x2[i]], [y1[i], y2[i]]])
    Xcorners[:, i] = vgg_X_fromxP_lin(u, [P1, P2], imsize)  # function from␣
 ↪http://www.robots.ox.ac.uk/~vgg/hzbook/code/
Xc = Xcorners[0:3, :] / Xcorners[[3, 3, 3], :]

# Visualize the projection reconstruction
# Notice that the shape is not a rectangular cuboid
# (there is a projective distortion)
plt.figure(3)
ax = plt.axes(projection='3d')
plt.title('Projection reconstruction (try rotating the shape)')
for i in range(np.shape(edges)[1]):
    ax.plot3D(Xc[0, edges[:, i]], Xc[1, edges[:, i]], Xc[2, edges[:, i]],␣
 ↪'gray')
for i in range(8):
    ax.text(Xc[0][i], Xc[1][i], Xc[2][i], labels[i], fontsize=20)
```

```
# Your next task is to project the cuboid corners 'Xc' (currently␣
 ↪non-homogeneous) to images 1 and 2.
# Use camera projection matrices P1 and P2 (recovered from the fundamental␣
 ↪matrix).
# Results are visualized using cyan lines.
# The cyan edges should be relatively close to the magenta lines from above.
# Convert the projected points back to non-homogeneous form and store into the␣
 ↪given variables below.
##-your-code-starts-here-##
Xc = np.vstack((Xc, np.ones(8)))        #vertically row wise
cx1_add = np.dot(P1, Xc)
cx2_add = np.dot(P2, Xc)
pcx1 = (cx1_add[0, :] / cx1_add[2, :])
pcy1 = (cx1_add[1, :] / cx1_add[2, :])
pcx2 = (cx2_add[0, :] / cx2_add[2, :])
pcy2 = (cx2_add[1, :] / cx2_add[2, :])

#pcx1 = np.zeros(np.shape(edges)[1])  # replace me
#pcy1 = np.zeros(np.shape(edges)[1])  # replace me
#pcx2 = np.zeros(np.shape(edges)[1])  # replace me
#pcy2 = np.zeros(np.shape(edges)[1])  # replace me
##-your-code-starts-here-##

# pcx1 and pcy1 are x and y coordinates for image 1, and similarly for image 2
plt.figure(1)
sp1.title.set_text('Cyan: projected reconstructed cuboid')
for i in range(np.shape(edges)[1]):
    sp1.plot(pcx1[edges[:, i]], pcy1[edges[:, i]], 'c-')
    sp2.plot(pcx2[edges[:, i]], pcy2[edges[:, i]], 'c-')
plt.show()
```

[ ]:
```
#a.
# The camcalibDLT performs a homogenous least squares fittin and returns a␣
 ↪projection matrix.
# The cameras are calibrated through direct linear transform
```

[ ]:
```
#b.
#The model preserves intersection and tangency. Looks distorted, but correct.
```

[ ]:
```
#c.
#pictures in the end
```

[ ]:
```
#d.
# If the calibration is correct, but the relative pose is not known, we know␣
 ↪that the angles between rays are true angles.
```

```
# Knowing this we have a similarity reconstruction.
```

Figure 2

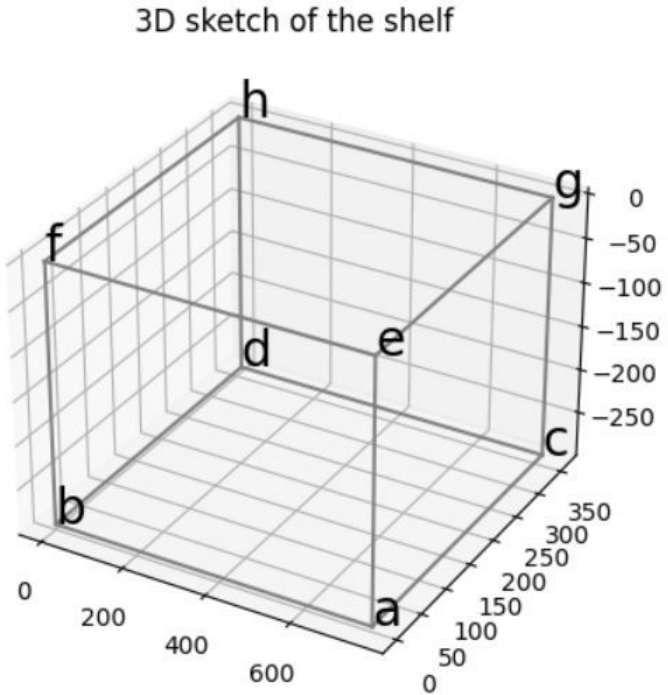
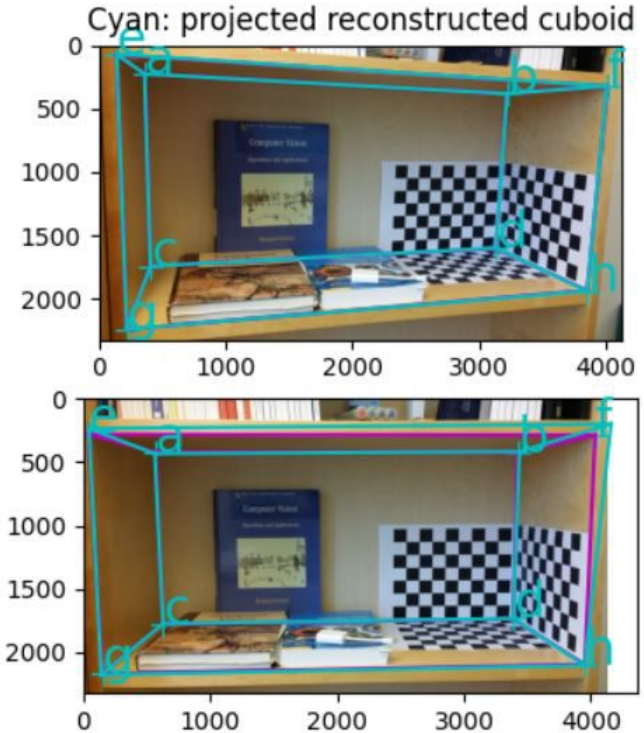


Figure 1



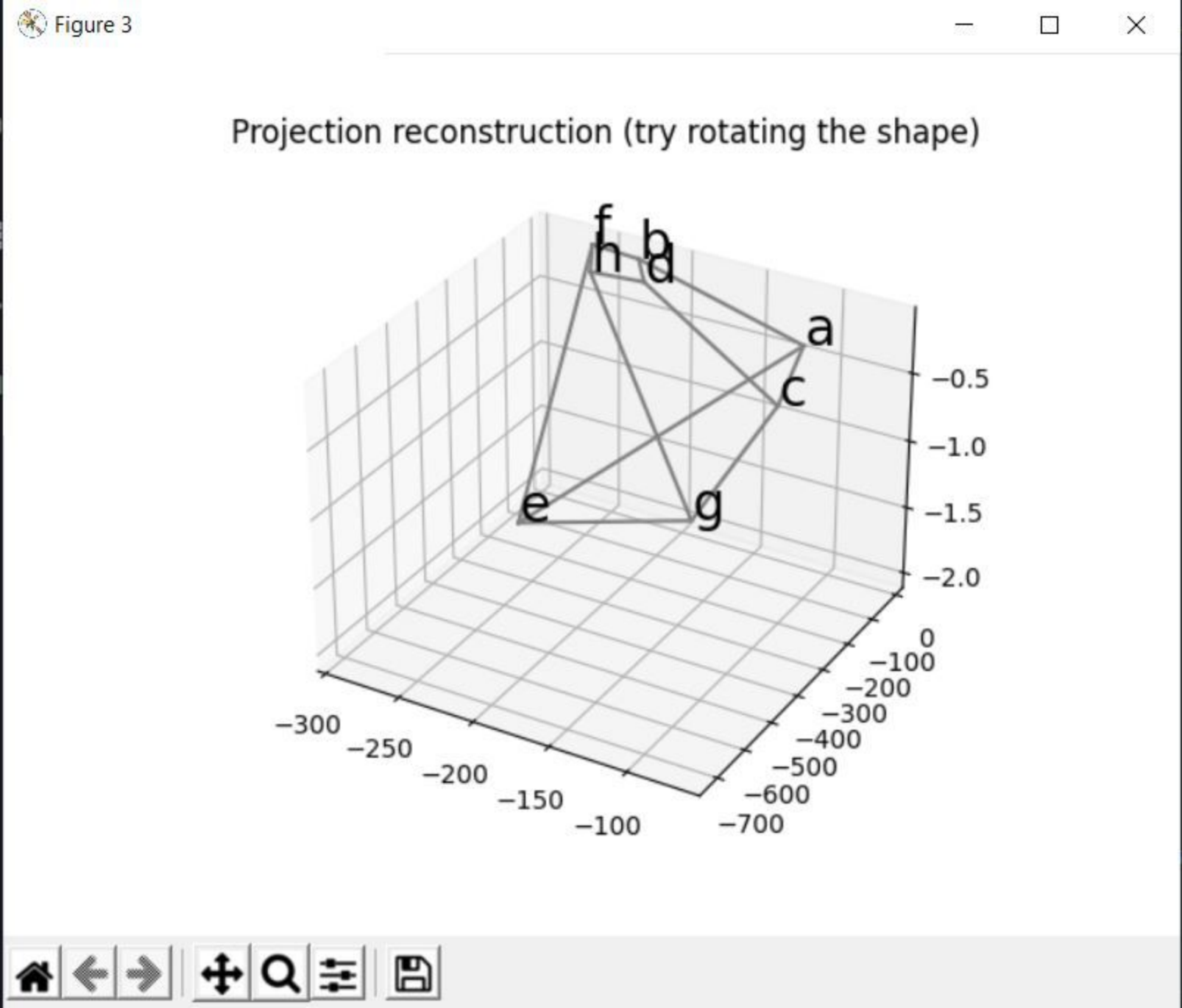


Figure 3
Projection reconstruction (try rotating the shape)
f
h
b
d
a
c
e
g
−0.5
−1.0
−1.5
−2.0
0
−100
−200
−300
−400
−500
−600
−700
−300
−250
−200
−150
−100