CS6476 - PS3: Geometry

Martin Saint-Jalmes (msaintjalmes3)

In [1]:

```
# Importing Numpy, OpenCV and Matplotlib
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#Importing python utilities
import math
from timeit import default_timer as timer
from itertools import izip

#imports for interactive jupyter
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

1. Calibration

1.1. Least Squares solving

In [2]:

```
def points(pathTo2D = "./pts2d-norm-pic_a.txt", pathTo3D = "./pts3d-norm.txt"):
    file2D = open(pathTo2D)
    points2D = np.array([map(float,s.strip().split()) for s in file2D.readlines()])
    file2D.close()

    file3D = open(pathTo3D)
    points3D = np.array([map(float,s.strip().split()) for s in file3D.readlines()])
    file3D.close()

    if points2D.shape[0] != points3D.shape[0]:
        raise Exception('Different list sizes for 2D and 3D points, can\'t match.')

    if points2D.shape[0] < 6:
        raise Exception('Needs 6 points or more.')

    return points2D, points3D
```

In [3]:

```
def findM(pathTo2D = "./pts2d-norm-pic_a.txt", pathTo3D = "./pts3d-norm.txt"):
    points2D, points3D = points(pathTo2D = pathTo2D, pathTo3D = pathTo3D)

    A = np.array([[[X, Y, Z, 1, 0, 0, 0, 0, -u*X, -u*Y, -u*Z, -u],\
                   [0, 0, 0, 0, X, Y, Z, 1, -v*X, -v*Y, -v*Z, -v]] \
                  for (u, v), (X, Y, Z) in izip(points2D, points3D)]).reshape(-1,12)

    eigvals, normed_eigvects = np.linalg.eig(np.matmul(A.T, A))

    return normed_eigvects[:, np.argmin(eigvals)].reshape(3,4)
```

In [4]:

```
M = findM()
M
```

Out[4]:

```
array([[-0.45827554,  0.29474237,  0.01395746, -0.0040258 ],
       [ 0.05085589,  0.0545847 ,  0.54105993,  0.05237592],
       [-0.10900958, -0.17834548,  0.04426782, -0.5968205 ]])
```

We will now test this solution by multiplying M with each of the 3D points.

In [5]:

```
def residuals(M, pathTo2D = "./pts2d-norm-pic_a.txt", pathTo3D = "./pts3d-norm.txt", la
stpointOnly = False):
    points2D, points3D = points(pathTo2D = "./pts2d-norm-pic_a.txt", pathTo3D = "./pts3
d-norm.txt")

    estd2D = np.matmul(M, np.pad(points3D.T, ((0,1),(0,0)), 'constant', constant_values
 = 1))
    estd2D = estd2D/estd2D[2] #normalization
    if lastpointOnly:
        return np.sqrt(np.sum((points2D.T - estd2D[:-1])**2, axis = 0))[-1], estd2D.T[-
1]
    else:
        return np.sum(np.sqrt(np.sum((points2D.T - estd2D[:-1])**2, axis = 0))), estd2D
.T
```

In [6]:

```
TotalResidual, _ = residuals(M)
TotalResidual
```

Out[6]:

```
0.044548941765583314
```

In [7]:

```
LastPointResidual, LastPointEstimation = residuals(M, lastpointOnly=True)
```

In [8]:

```
LastPointResidual
```

Out[8]:

```
0.0015621360462174003
```

In [9]:

```
LastPointEstimation
```

Out[9]:

```
array([ 0.14190608, -0.45184301,  1.        ])
```

The estimated projection of the last point using the M matrix is (0.14190608, -0.45184301), having renormalized the point. The residual between that point and its true coordinates is 0.00156

1.2. Solving M with different point sets

In [10]:

```
def best_eval(k_values = [8,12,16], iters = 10, test_size = 4, pathTo2D = "./pts2d-pic_
b.txt", pathTo3D = "./pts3d.txt"):
    points2D, points3D = points(pathTo2D = pathTo2D, pathTo3D = pathTo3D)

    AvgRes = np.zeros((len(k_values), iters))

    for i, k in enumerate(k_values):
        bestAvgRes = np.inf
        for it in range(iters):
            k_sample = np.random.choice(np.arange(points2D.shape[0]), k, replace=False)
            points2D_sample = points2D[k_sample]
            points3D_sample = points3D[k_sample]

            A = np.array([[[X, Y, Z, 1, 0, 0, 0, 0, -u*X, -u*Y, -u*Z, -u],\
               [0, 0, 0, 0, X, Y, Z, 1, -v*X, -v*Y, -v*Z, -v]] \
              for (u, v), (X, Y, Z) in izip(points2D_sample, points3D_sample)]).reshape
(-1,12)

            eigvals, normed_eigvects = np.linalg.eig(np.matmul(A.T, A))

            M = normed_eigvects[:, np.argmin(eigvals)].reshape(3,4)

            test_idx = np.random.choice(np.setdiff1d(np.arange(points2D.shape[0]), k_sa
mple), test_size, replace=False)
            points3D_test = points3D[test_idx]
            points2D_test = points2D[test_idx]

            estd2D = np.matmul(M, np.pad(points3D_test.T, ((0,1),(0,0)), 'constant', co
nstant_values = 1))
            estd2D = estd2D/estd2D[2] #normalization
            res = np.mean(np.sqrt(np.sum((points2D_test.T - estd2D[:-1])**2, axis = 0
)))
            AvgRes[i, it]  = res

            if res < bestAvgRes:
                bestAvgRes = res
                bestM = M

    return pd.DataFrame(data=AvgRes, index=k_values), bestM
```

In [11]:

```
AvgRes, M = best_eval()
```

In [12]:

```
AvgRes["Avg over iterations"] = AvgRes.mean(axis = 1)
AvgRes
```

Out[12]:

	0	1	2	3	4	5	6	7	8	9	Avg over iterations
8	6.938399	1.764136	1.922663	2.156405	2.472430	1.134872	1.414657	3.933766	2.357821	2.003203	2.609835
12	1.425206	1.456054	1.590898	1.196853	2.396635	1.131780	1.044034	0.639276	1.448352	1.052777	1.338186
16	0.948399	1.247235	1.333880	1.350000	1.233938	1.045115	1.524194	1.177098	0.908633	0.786859	1.155535

In [13]:

```
M
```

Out[13]:

```
array([[-6.94188974e-03,  4.02522167e-03,  1.33771895e-03,
         8.26905955e-01],
       [-1.54709871e-03, -1.02461684e-03,  7.27837980e-03,
         5.62220991e-01],
       [-7.61514645e-06, -3.71312542e-06,  1.89374098e-06,
         3.39109698e-03]])
```

Overall, we can observe that the more points we use to solve M, the lower the average residual we get. Having full results for 10 distinct runs allows us to get more insight regarding possible variability in our estimations.
At the 2nd iteration, using 12 points, we get a very high average residual that is 5 times the average residual value of other runs. This outlier (possibly due to the choice of a bad point in those used for the estimation of M, or a point that was very off from its regular location in the test set, or a bad/unstable eigenvalue estimation using SVD) raises the average residual over iterations for k=12 points. However the general observation that overconstraining the system decreases errors/residuals still holds in general.

1.3. Camera center

In [14]:

```
def findCenter(M):
    return -1 * np.matmul(np.linalg.inv(M[:,:-1]), M[:,-1])
```

In [15]:

```
findCenter(M)
```

Out[15]:

```
array([303.09646555, 307.17745696,  30.42412736])
```

2. Fundamental Matrix Evaluation

2.1. Least squares estimate of $\tilde{F}$

In [16]:

```
def findFtilde(pathTo2DPicA = "./pts2d-pic_a.txt", pathTo2DPicB = "./pts2d-pic_b.txt"):
    pointsA, pointsB = points(pathTo2DPicA, pathTo2DPicB)

    F = np.array([[u_b*u, u_b*v, u_b, v_b*u, v_b*v, v_b, u, v, 1] \
                  for (u, v), (u_b, v_b) in izip(pointsA, pointsB)]).reshape(-1,9)

    eigvals, normed_eigvects = np.linalg.eig(np.matmul(F.T, F))

    return normed_eigvects[:, np.argmin(eigvals)].reshape(3,3)
```

In [17]:

```
Ftilde = findFtilde()
Ftilde
```

Out[17]:

```
array([[-6.60698417e-07,  7.91031620e-06, -1.88600197e-03],
       [ 8.82396296e-06,  1.21382934e-06,  1.72332901e-02],
       [-9.07382303e-04, -2.64234650e-02,  9.99500092e-01]])
```

2.2. Estimation of F (rank 2)

In [18]:

```
def reduceFrank(Ftilde):
    U, Sigma, V = np.linalg.svd(Ftilde)

    Sigma[np.argmin(Sigma)] = 0
    return np.matmul(U * Sigma, V)
```

In [19]:

```
F = reduceFrank(Ftilde)
F
```

Out[19]:

```
array([[-5.36264197e-07,  7.90364770e-06, -1.88600204e-03],
       [ 8.83539184e-06,  1.21321686e-06,  1.72332901e-02],
       [-9.07382265e-04, -2.64234650e-02,  9.99500092e-01]])
```

2.3. Epipolar line estimation

In [20]:

```
pointsA, pointsB = points("./pts2d-pic_a.txt", "./pts2d-pic_b.txt")
l_b = np.matmul(F, np.pad(pointsA.T, ((0,1),(0,0)), 'constant', constant_values = 1))
l_a = np.matmul(F.T, np.pad(pointsB.T, ((0,1),(0,0)), 'constant', constant_values = 1))
```

In [21]:

```
pic_a = cv2.imread("./pic_a.jpg", cv2.IMREAD_COLOR)
(max_row, max_col, _) = pic_a.shape

p_UL = [0, 0, 1]
p_UR = [0, max_col-1, 1]
p_BL = [max_row-1, 0, 1]
p_BR = [max_row-1, max_col-1, 1]

l_L = np.cross(p_UL, p_BL)
l_R = np.cross(p_UR, p_BR)
```

In [22]:

```
P_L_b = np.array([np.cross(li, l_L) for li in l_b.T])
P_R_b = np.array([np.cross(li, l_R) for li in l_b.T])

P_L_a = np.array([np.cross(li, l_L) for li in l_a.T])
P_R_a = np.array([np.cross(li, l_R) for li in l_a.T])
```

In [23]:

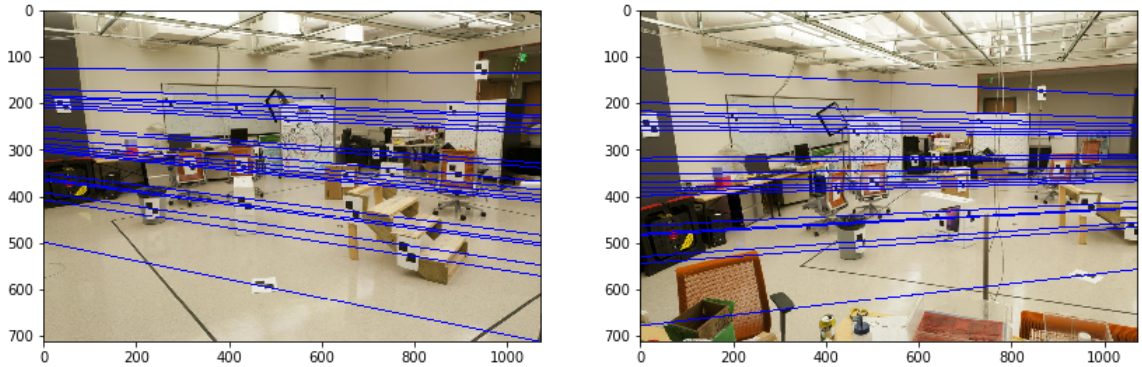
```
P_L_b = (P_L_b.T/P_L_b[:,2]).astype(int).T
P_R_b = (P_R_b.T/P_R_b[:,2]).astype(int).T
P_L_a = (P_L_a.T/P_L_a[:,2]).astype(int).T
P_R_a = (P_R_a.T/P_R_a[:,2]).astype(int).T
```

In [24]:

```
pic_a = cv2.imread("./pic_a.jpg", cv2.IMREAD_COLOR)
pic_b = cv2.imread("./pic_b.jpg", cv2.IMREAD_COLOR)

plt.figure(figsize=(14,9))

plt.subplot(1,2,1)
for (row1, col1, _), (row2, col2, _) in zip(P_L_a, P_R_a):
    cv2.line(pic_a, (row1, col1), (row2, col2), (255, 0, 0), 2)
plt.imshow(cv2.cvtColor(pic_a, cv2.COLOR_BGR2RGB))

plt.subplot(1,2,2)
for (row1, col1, _), (row2, col2, _) in zip(P_L_b, P_R_b):
    cv2.line(pic_b, (row1, col1), (row2, col2), (255, 0, 0), 2)
plt.imshow(cv2.cvtColor(pic_b, cv2.COLOR_BGR2RGB))
```

Out[24]:

```
<matplotlib.image.AxesImage at 0x18c95b38>
```



2.4x Transform matrices for normalization

In [25]:

```
pointsA, pointsB = points("./pts2d-pic_a.txt", "./pts2d-pic_b.txt")
(mu_u_a, mu_v_a) = np.mean(pointsA, axis = 0)
(mu_u_b, mu_v_b) = np.mean(pointsB, axis = 0)

pointsA_Normalized = pointsA - (mu_u_a, mu_v_a)
pointsB_Normalized = pointsB - (mu_u_b, mu_v_b)

s_a = 1/np.std(pointsA_Normalized)
s_b = 1/np.std(pointsB_Normalized)
```

In [26]:

```
scale_mat_a = np.diag([s_a, s_a, 1])
center_mat_a = np.eye(3); center_mat_a[0:2,2] = [-mu_u_a, -mu_v_a]
print(scale_mat_a); print(center_mat_a)
```

```
[[0.00536182 0.         0.        ]
 [0.         0.00536182 0.        ]
 [0.         0.         1.        ]]
[[   1.      0.   -558.95]
 [   0.      1.   -325.6 ]
 [   0.      0.      1.  ]]
```

In [27]:

```
scale_mat_b = np.diag([s_b, s_b, 1])
center_mat_b = np.eye(3); center_mat_b[0:2,2] = [-mu_u_b, -mu_v_b]
print(scale_mat_b); print(center_mat_b)
```

```
[[0.00506058 0.         0.        ]
 [0.         0.00506058 0.        ]
 [0.         0.         1.        ]]
[[   1.     0.  -616.7]
 [   0.     1.  -346.9]
 [   0.     0.     1. ]]
```

In [28]:

```
T_a = np.matmul(scale_mat_a, center_mat_a)
T_a
```

Out[28]:

```
array([[ 0.00536182,  0.        , -2.99699089],
       [ 0.        ,  0.00536182, -1.74580953],
       [ 0.        ,  0.        ,  1.        ]])
```

In [29]:

```
T_b = np.matmul(scale_mat_b, center_mat_b)
T_b
```

Out[29]:

```
array([[ 0.00506058,  0.        , -3.1208599 ],
       [ 0.        ,  0.00506058, -1.75551532],
       [ 0.        ,  0.        ,  1.        ]])
```

In [30]:

```
pointsA_Normalized = np.matmul(T_a, np.pad(pointsA.T, ((0,1),(0,0)), 'constant', consta
nt_values = 1))
pointsB_Normalized = np.matmul(T_b, np.pad(pointsB.T, ((0,1),(0,0)), 'constant', consta
nt_values = 1))
```

In [31]:

```
def estimate_F_hat(pointsA, pointsB):
    F = np.array([[u_b*u, u_b*v, u_b, v_b*u, v_b*v, v_b, u, v, 1] \
                for (u, v, _), (u_b, v_b, _) in izip(pointsA.T, pointsB.T)]).reshape(-1,9
)

    eigvals, normed_eigvects = np.linalg.eig(np.matmul(F.T, F))

    Ftilde = normed_eigvects[:, np.argmin(eigvals)].reshape(3,3)

    U, Sigma, V = np.linalg.svd(Ftilde)
    Sigma[np.argmin(Sigma)] = 0

    return np.matmul(U * Sigma, V)
```

In [32]:

```
F_hat = estimate_F_hat(pointsA_Normalized, pointsB_Normalized)
F_hat
```

Out[32]:

```
array([[ 0.00432111, -0.05927067, -0.01109126],
       [-0.04098664,  0.01007756, -0.74414192],
       [-0.05405815,  0.66154769,  0.01577379]])
```

2.5x Redraw epipolar lines with a better estimation of F

In [33]:

```
F = np.matmul(np.matmul(T_b.T, F_hat), T_a)
F
```

Out[33]:

```
array([[ 1.17248591e-07, -1.60824663e-06,  4.01980786e-04],
       [-1.11212887e-06,  2.73443755e-07, -3.23319884e-03],
       [ 2.36400817e-05,  4.44404958e-03, -1.03455561e-01]])
```

In [34]:

```
l_b = np.matmul(F, np.pad(pointsA.T, ((0,1),(0,0)), 'constant', constant_values = 1))
l_a = np.matmul(F.T, np.pad(pointsB.T, ((0,1),(0,0)), 'constant', constant_values = 1))

P_L_b = np.array([np.cross(li, l_L) for li in l_b.T])
P_R_b = np.array([np.cross(li, l_R) for li in l_b.T])

P_L_a = np.array([np.cross(li, l_L) for li in l_a.T])
P_R_a = np.array([np.cross(li, l_R) for li in l_a.T])

P_L_b = (P_L_b.T/P_L_b[:,2]).astype(int).T
P_R_b = (P_R_b.T/P_R_b[:,2]).astype(int).T
P_L_a = (P_L_a.T/P_L_a[:,2]).astype(int).T
P_R_a = (P_R_a.T/P_R_a[:,2]).astype(int).T

pic_a = cv2.imread("./pic_a.jpg", cv2.IMREAD_COLOR)
pic_b = cv2.imread("./pic_b.jpg", cv2.IMREAD_COLOR)

plt.figure(figsize=(14,9))

plt.subplot(1,2,1)
for (row1, col1, _), (row2, col2, _) in zip(P_L_a, P_R_a):
    cv2.line(pic_a, (row1, col1), (row2, col2), (255, 0, 0), 2)
plt.imshow(cv2.cvtColor(pic_a, cv2.COLOR_BGR2RGB))

plt.subplot(1,2,2)
for (row1, col1, _), (row2, col2, _) in zip(P_L_b, P_R_b):
    cv2.line(pic_b, (row1, col1), (row2, col2), (255, 0, 0), 2)
plt.imshow(cv2.cvtColor(pic_b, cv2.COLOR_BGR2RGB))
```

Out[34]:

```
<matplotlib.image.AxesImage at 0x1932b128>
```