Introduction to Computer Vision : HW 1

Jeong Min Lee

March 28, 2024

Notation

1. $O(n)$: Orthogonal group with dimension n.
2. $r = \min(m, n)$: From $\mathbf{A} = \mathbf{U\Sigma V^T}$, $\mathbf{U} \in O(m)$, while $\mathbf{V} \in O(n)$
3. Σ : diagonal matrix whose entries are σ_i

1 Camera Calibration

a

From the result of SVD,

$$\mathbf{AV} = \mathbf{U\Sigma} \tag{1}$$

Since $\mathbf{V} \in O(n)$, the column vectors of $\mathbf{V}$, denoting $\mathbf{v_i}$, can form the basis of $\mathbb{R}^n$. That is, $\forall \mathbf{p} \in \mathbb{R}^n, \mathbf{p} = \sum_i a_i \mathbf{v_i}$, for $a_i \in \mathbb{R}$. This results in $\mathbf{Ap} = \sum_i a_i \mathbf{Av_i} = \sum_i a_i \sigma_i \mathbf{u_i}$. Thus,

$$\|\mathbf{Av}\|^2 = \sum_i a_i^2 \sigma_i^2 \tag{2}$$

It is a convention to make the diagonal entries of $\mathbf{\Sigma}$, σ_i, be ordered, that is, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$. Thus, to minimize equation 2, $a_1 = a_2 = \cdots = a_{r-1} = 0$ and $a_r = 1$, since $\mathbf{p}$ is normalized. This implies $\mathbf{p} = \mathbf{v_n}$.(note that $r = n$ since we assumed over-determined system.)

b

The following code is the implmentation to calculate the camera matrix for given corresponding points. I truncated the data load code in the following code for readability. Please refer to my jupyter note submission to get the whole implementation.

```
x = ... #(2D points & truncated due to redundancy)

X = ... #(3D points & truncated due to redundancy)

A = []
for i in range(len(X)):
    xx = X[i]
    u,v = x[i]
    A.append(
        [xx[0], xx[1], xx[2],1,0,0,0,0,-u*xx[0], -u*xx[1], -u*xx[2], -u]
    )
    A.append(
        [0,0,0,0,xx[0], xx[1], xx[2],1,-v*xx[0], -v*xx[1], -v*xx[2], -v ]
    )
import numpy as np
A = np.array(A)
[U,S,V] = np.linalg.svd(A)
p = V[-1,:]
P_1 = p.reshape((3,4))
print(P_1)
```

The result of the code above is

```
[[ 3.09963996e-03  1.46204548e-04 -4.48497465e-04 -9.78930678e-01]
 [ 3.07018252e-04  6.37193664e-04 -2.77356178e-03 -2.04144405e-01]
 [ 1.67933533e-06  2.74767684e-06 -6.83964827e-07 -1.32882928e-03]]
```

c

Before determining $\mathbf{P}$, I proved that $\mathbf{p} = \mathbf{A}^\dagger \mathbf{b}$. Consider SVD of $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathbf{T} \in \mathbb{R}^{m\times n}$, where $\mathbf{U} \in O(m), \boldsymbol{\Sigma} \in \mathbb{R}^{m\times n}, \mathbf{V} \in O(n)$.

$$\|\mathbf{Ap} - \mathbf{b}\|^2 = \|\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\mathbf{T}\mathbf{p} - \mathbf{b}\|^2 = \|\boldsymbol{\Sigma}\mathbf{V}^\mathbf{T}\mathbf{p} - \mathbf{U}^\mathbf{T}\mathbf{b}\| \quad (\because \mathbf{U} \in O(m)) \tag{3}$$

Let $\mathbf{V}^\mathbf{T}\mathbf{p} = \mathbf{q}, \mathbf{U}^\mathbf{T}\mathbf{b} = \mathbf{r}$ and rewrite the problem as follow.

$$\text{Find } \arg\min_{\mathbf{q}} \|\boldsymbol{\Sigma}\mathbf{q} - \mathbf{r}\|^2 \tag{4}$$

This problem can be solved as follow.

$$\|\boldsymbol{\Sigma}\mathbf{q} - \mathbf{r}\|^2 = \sum_{i=1}^{r}(\sigma_i q_i - r_i)^2 \text{ where } r = \min\{m, n\} \tag{5}$$

We can uniquely select the $q_i = r_i/\sigma_i$, for $i = 1, \cdots, r$, or $\mathbf{q} = \boldsymbol{\Sigma}^{-1}\mathbf{r}$, to minimize this expression. This implies $\mathbf{p} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^\mathbf{T}\mathbf{b}$. Thus, proof is done, noting that $\mathbf{A}^\dagger = (\mathbf{A}^\mathbf{T}\mathbf{A})^{-1}\mathbf{A}^\mathbf{T} = (\mathbf{V}\boldsymbol{\Sigma}^\mathbf{T}\boldsymbol{\Sigma}\mathbf{V}^\mathbf{T})^{-1}\boldsymbol{\Sigma}^\mathbf{T}\mathbf{U}^\mathbf{T} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^\mathbf{T}$

I used following code to calculate the camera matrix.

```
def pseudo_inverse(A):
    return np.linalg.pinv(A)

A = []
b = []
for i in range(len(X)):
    xx = X[i]
    u,v = x[i]
    A.append(
        [xx[0], xx[1], xx[2],1,0,0,0,0,-u*xx[0], -u*xx[1], -u*xx[2]]
    )
    A.append(
        [0,0,0,0,xx[0], xx[1], xx[2],1,-v*xx[0], -v*xx[1], -v*xx[2]]
    )
    b.append(u)
    b.append(v)

A = np.array(A)
b = np.array(b)
p = np.matmul(pseudo_inverse(A),b)
p = np.append(p,1)
P_2 = p.reshape((3,4))
print(P_2)
```

The result of the code above is following.

```
[[-2.33259098e+00 -1.09993113e-01  3.37413916e-01  7.36673920e+02]
 [-2.31050254e-01 -4.79506029e-01  2.08717636e+00  1.53627756e+02]
 [-1.26379606e-03 -2.06770917e-03  5.14635233e-04  1.00000000e+00]]
```

I used following code to check whether the result of problem 1-(b) and problem 1-(c) agree by setting $m_{34} = 1$ in the camera matrix calculated on problem 1-(b). Since the result of following code is similar to camera matrix above, I can verify that both methodology are equivalent.

```
print(P_2[-1,-1]/P_1[-1,-1]*P_1)
```

The reason they are not exactly same is the floating point number error.

```
print(P_2[-1,-1]/P_1[-1,-1]*P_1 - P_2)
```

The code above returns the following matrix.

```
[[-1.86427717e-05 -3.19670417e-05  9.93167477e-05  1.26469280e-02]
 [ 6.08843664e-06 -9.04052567e-06  4.56969897e-05 -4.92271847e-04]
 [ 2.54860563e-08 -3.33770907e-08  7.71079873e-08 -1.11022302e-16]]
```

As the (3,4) element of the matrix above is almost the accuracy limit of floating number system, there is some deviation when scaling of camera matrix in problem 1-(b).[1] This is common phenomenon in computer science. Eventhough such deviation, the relative error is extremely small. I calcuated the relative error of each element

[1] The assertion that a 64-bit computer system is restricted to a maximum precision of 16 decimal digits is commonly based on the standard double-precision floating-point format, adhering to the IEEE 754 standard.

in second camera matrix by applying following formular, elementwisely.[2] The result is described on following matrix.

$$(\text{Scaled camera matrix in } P\_1 - P\_2)/P\_2 \tag{6}$$

```
[[ 7.99230207e-06  2.90627666e-04  2.94346922e-04  1.71676066e-05]
 [-2.63511359e-05  1.88538311e-05  2.18941679e-05 -3.20431581e-06]
 [-2.01662730e-05  1.61420625e-05  1.49830370e-04 -1.11022302e-16]]
```

[2]Please refer the jupyter notebook to see my implementation