# Introduction to Computer Vision : HW 1

Jeong Min Lee

March 28, 2024

## 1 Camera Calibration

### a

From the result of SVD,

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

Since $\mathbf{V} \in \mathbf{O(n)}$, the column vector of $\mathbf{V}$, denoting $\mathbf{v_i}$ , can form the basis of $\mathbb{R}^n$. That is, $\forall \mathbf{x} \in \mathbf{R}^n, \mathbf{x} = \sum_i a_i \mathbf{v_i}$, 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 $\mathbf{\Sigma}$, $\sigma_i$, 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.

```
x = [[880, 214],
[43, 203],
[270, 197],
[886, 347],
[745, 302],
[943, 128],
[476, 590],
[419, 214],
[317, 335],
[783, 521],
[235, 427],
[665, 429],
[655, 362],
[427, 333],
[412, 415],
[746, 351],
[434, 415],
[525, 234],
[716,308],
[602, 187]]

X = [[312.747, 309.140, 30.086],
[305.796, 311.649, 30.356],
[307.694, 312.358, 30.418],
[310.149, 307.186, 29.298],
[311.937, 310.105, 29.216],
[311.202, 307.572, 30.682],
[307.106, 306.876, 28.660],
[309.317, 312.490, 30.230],
[307.435, 310.151, 29.318],
[308.253, 306.300, 28.881],
[306.650, 309.301, 28.905],
[308.069, 306.831, 29.189],
[309.671, 308.834, 29.029],
[308.255, 309.955, 29.267],
[307.546, 308.613, 28.963],
[311.036, 309.206, 28.913],
[307.518, 308.175, 29.069],
[309.950, 311.262, 29.990],
[312.160, 310.772, 29.080],
[311.988, 312.709, 30.514]]
```

```
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)
```

### c

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

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

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

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

This problem can be solved as follow.

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

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

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)
```

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 agrees to camera matrix above, I can verify that both methodology are equivalent.

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