

# Solving Problems In Image Formation and Geometry



Sunglok Choi, Assistant Professor, Ph.D.  
Computer Science and Engineering Department, SEOULTECH  
[sunglok@seoultech.ac.kr](mailto:sunglok@seoultech.ac.kr) | <https://mint-lab.github.io/>

# Table of Contents: Solving Problems

- Solving **Linear Equations** using Linear Algebra
- Solving **Nonlinear Equations** using Nonlinear Optimization
- Solving **Linear Equations in 3D Vision**
  - Affine transformation estimation
  - Planar homography estimation
    - Note) Image warping using homography
- Solving **Nonlinear Equations in 3D Vision**
  - Absolute camera pose estimation (PnP)
  - Camera calibration

## Getting Started with a Simple Problem

- Jane bought 3 apples and 4 oranges and paid 10,000 KRW.
- Daniel bought 5 apples and 2 orange and paid 12,000 KRW.

- **Question) How much is an apple and an orange, respectively?**

- **Answer) Calculating the price of apple and orange**

- Unknown: the price apple  $x$  and the price of orange  $y$
- Constraints:  $3x + 4y = 10000$  and  $5x + 2y = 12000$
- Solution using an inverse matrix

$$\begin{bmatrix} 3 & 4 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 10000 \\ 12000 \end{bmatrix} \rightarrow A\mathbf{x} = \mathbf{b}$$

$$\text{Solve } A\mathbf{x} = \mathbf{b} \rightarrow \mathbf{x} = A^{-1}\mathbf{b} = \begin{bmatrix} 2000 \\ 1000 \end{bmatrix}$$

Therefore, an apple is 2,000 KRW, and an orange is 1,000 KRW.

# Getting Started with Line Fitting

Q) How about finding a line from three points?

(1, 4)



(4, 2)



(7, 1)



- Line representation:  $y = ax + b$  ( $y = -\frac{2}{3}x + \frac{14}{3}$ )
- Slope  $a = \frac{2-4}{4-1} = -\frac{2}{3}$
- Y intercept  $b = 4 - a \cdot 1 = \frac{14}{3}$

# Solving Linear Equations using Linear Algebra

- **Example) Line fitting from two points, (1, 4) and (4, 2)**    Q) How about finding a line from three points?

- Unknown: Line parameters  $a$  and  $b$  (line representation:  $y = ax + b$ )
- Constraints:  $a \cdot 1 + b = 4$  and  $a \cdot 4 + b = 2$
- Solution

$$\begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} \rightarrow \mathbf{Ax} = \mathbf{b}$$
$$\therefore \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \frac{1}{3} \begin{bmatrix} -2 \\ 14 \end{bmatrix}$$

- **Example) Line fitting from two points, (1, 4) and (4, 2), using [NumPy](#)**

```
import numpy as np
```

```
A = np.array([[1., 1.], [4., 1.]])
```

```
b = np.array([[4.], [2.]])
```

```
A_inv = np.linalg.inv(A)
```

```
print(A_inv * b) # [[-1.33333333  1.33333333]
                  # [ 2.66666667 -0.66666667]] Note) broadcast
```

```
print(A_inv @ b) # [[-0.66666667]
                  # [ 4.66666667]] Note) A_inv.matmul(b)
```

# Solving Linear Equations using Linear Algebra

- **Example) Line fitting from more than two points such as (1, 4), (4, 2), and (7, 1)**

- Unknown: Line parameters  $a$  and  $b$  (line representation:  $y = ax + b$ )
- Constraints:  $a \cdot 1 + b = 4$ ,  $a \cdot 4 + b = 2$ , and  $a \cdot 7 + b = 1$
- Solution

$$\begin{bmatrix} 1 & 1 \\ 4 & 1 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} \rightarrow A\mathbf{x} = \mathbf{b}$$

$\therefore \mathbf{x} = A^\dagger \mathbf{b}$  where  $A^\dagger$  is a pseudo-inverse (Note:  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{Ax} - \mathbf{b}\|_2^2$ )

- **Note) Pseudo-inverse** (~ a generalized matrix inverse)

- **Not necessarily square** ( $A$ :  $m \times n$  matrix,  $A^\dagger$ :  $n \times m$  matrix)
- Left inverse ( $A^\dagger A = I_n$ ):  $A^\dagger = (A^\top A)^{-1} A^\top$ 
  - If  $A$  has linearly independent columns ( $\operatorname{rank}(A) = n$ )
- Right inverse ( $AA^\dagger = I_m$ ):  $A^\dagger = A^\top (AA^\top)^{-1}$ 
  - If  $A$  has linearly independent rows ( $\operatorname{rank}(A) = m$ )

# Solving Linear Equations using Linear Algebra

- **Example) Line fitting from more than two points such as (1, 4), (4, 2), and (7, 1)**

- Unknown: Line parameters  $a$  and  $b$  (line representation:  $y = ax + b$ )
- Constraints:  $a \cdot 1 + b = 4$ ,  $a \cdot 4 + b = 2$ , and  $a \cdot 7 + b = 1$
- Solution

$$\begin{bmatrix} 1 & 1 \\ 4 & 1 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} \rightarrow A\mathbf{x} = \mathbf{b}$$

$\therefore \mathbf{x} = A^{\dagger}\mathbf{b}$  where  $A^{\dagger}$  is a pseudo-inverse (Note:  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|A\mathbf{x} - \mathbf{b}\|_2^2$ )

- **Example) Line fitting from more than two points such as (1, 4), (4, 2), and (7, 1), using [NumPy](#)**

```
import numpy as np
```

```
A = np.array([[1., 1.], [4., 1.], [7., 1.]])
```

```
b = np.array([[4.], [2.], [1.]])
```

```
A_inv = np.linalg.pinv(A)
```

```
print(A_inv @ b) # [[-0.5], [ 4.33333333]]
```

# Getting Started with Line Fitting

(1, 4)

(4, 2)

- Line representation:  $y = ax + b$  ( $y = -\frac{2}{3}x + \frac{14}{3}$ )
- Slope  $a = \frac{2-4}{4-1} = -\frac{2}{3}$
- Y intercept  $b = 4 - a \cdot 1 = \frac{14}{3}$

Q) Can it represent a vertical line such as  $x = 1$ ?

- Line representation:  $ax + by + c = 0$   
( $2x + 3y - 14 = 0$ ;  $4x + 6y - 28 = 0$ )  
→ additional constraint  $a^2 + b^2 = 1$
- Its shorter form:  $\mathbf{n}^T \mathbf{x} + c = 0$   
( $\mathbf{n} = \begin{bmatrix} a \\ b \end{bmatrix}$  and  $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ )

Normal vector





# Solving Linear Equations using Linear Algebra

- **Example) Line fitting from two points, (1, 4) and (4, 2)**

- Unknown: Line parameters  $a$ ,  $b$ , and  $c$  (line representation:  $ax + by + c = 0$ )
- Constraints:  $a \cdot 1 + b \cdot 4 + c = 0$  and  $a \cdot 4 + b \cdot 2 + c = 0$
- Solution

$$\begin{bmatrix} 1 & 4 & 1 \\ 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow A\mathbf{x} = \mathbf{0} \text{ (homogeneous equation) } \quad \cancel{\mathbf{x} = A^{-1}\mathbf{b}}$$

$$\therefore \mathbf{x} = \text{null}(A) = [2, 3, -14]^T \rightarrow 2x + 3y - 14 = 0$$

- **Note) Null space** (a.k.a. [kernel](#))

- A set of vectors which map  $A$  ( $m$ -by- $n$  matrix) to the zero vector

$$\text{null}(A) = \{ \mathbf{v} \in K^n \mid A\mathbf{v} = \mathbf{0} \}$$

- [Rank-nullity theorem](#):  $\text{rank}(A) + \text{nullity}(A) = n$

# Solving Linear Equations using Linear Algebra

- **Example) Line fitting from two points, (1, 4) and (4, 2)**

- Unknown: Line parameters  $a$ ,  $b$ , and  $c$  (line representation:  $ax + by + c = 0$ )
- Constraints:  $a \cdot 1 + b \cdot 4 + c = 0$  and  $a \cdot 4 + b \cdot 2 + c = 0$
- Solution

$$\begin{bmatrix} 1 & 4 & 1 \\ 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow A\mathbf{x} = \mathbf{0} \text{ (homogeneous equation) } \mathbf{x} = A^\dagger \mathbf{b}$$

$$\therefore \mathbf{x} = \text{null}(A) = [2, 3, -14]^\top \rightarrow 2x + 3y - 14 = 0$$

- **Example) Line fitting from two points, (1, 4) and (4, 2), using [NumPy](#)**

```
import numpy as np
from scipy import linalg
```

```
A = np.array([[1., 4., 1.], [4., 2., 1.]])
x = linalg.null_space(A)
print(x / x[0]) # [[1.], [1.5], [-7.]]
```

Q) How about finding a line from three points?

A) No null space for  $A = \begin{bmatrix} 1 & 4 & 1 \\ 4 & 2 & 1 \\ 7 & 1 & 1 \end{bmatrix}$  ( $\because$  full rank)

# Solving Linear Equations using Linear Algebra

- **Example) Line fitting from more than two points such as (1, 4), (4, 2), and (7, 1)**

- Unknown: Line parameters  $a$ ,  $b$ , and  $c$  (line representation:  $ax + by + c = 0$ )
- Constraints:  $a \cdot 1 + b \cdot 4 + c = 0$ ,  $a \cdot 4 + b \cdot 2 + c = 0$ , and  $a \cdot 7 + b \cdot 1 + c = 0$
- Solution

$$A = \begin{bmatrix} 1 & 4 & 1 \\ 4 & 2 & 1 \\ 7 & 1 & 1 \end{bmatrix} = USV^T \text{ using } \text{singular value decomposition (SVD)}$$

$\therefore \mathbf{x}$  is the last row of  $V^T$ . (Note:  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{Ax}\|^2$  with  $\|\mathbf{x}\|^2 = 1$  condition)

- **Example) Line fitting from more than two points such as (1, 4), (4, 2), and (7, 1), using [NumPy](#)**

```
import numpy as np
```

```
A = np.array([[1., 4., 1.], [4., 2., 1.], [7., 1., 1.]])
```

```
U, S, Vt = np.linalg.svd(A, full_matrices=True)
```

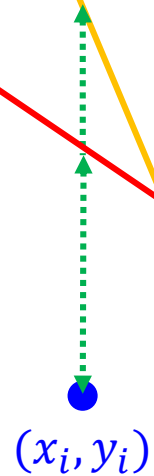
```
x = Vt[-1].T
```

```
print(x / -x[1]) # [[-0.50824973], [-1.], [4.37835787]]
```

```
# Note) [[w], [b]] = [[-0.5], [ 4.33333333]]
```

# Getting Started with Line Fitting

- Line representation:  $y = ax + b$
- Algebraic distance  $d_a = (ax_i + b) - y_i$   
(signed distance)
- Line fitting using  $\hat{\mathbf{x}} = \mathbf{A}^\dagger \mathbf{b} \rightarrow \hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$   
 $\hat{a}, \hat{b} = \underset{a,b}{\operatorname{argmin}} \sum_i (ax_i + b - y_i)^2$



Q) Which line is more closer to the point?

# Getting Started with Line Fitting

- Line representation:  $ax + by + c = 0$
- Geometric distance  $d_g = \frac{ax_i + by_i + c}{\sqrt{a^2 + b^2}} = \frac{\mathbf{n}^T \mathbf{x}_i + c}{\|\mathbf{n}\|}$   
(signed distance)
- Line fitting using  $\hat{a}, \hat{b}, \hat{c} = \operatorname{argmin}_{a,b,c} \sum_i \left( \frac{ax_i + by_i + c}{\sqrt{a^2 + b^2}} \right)^2$

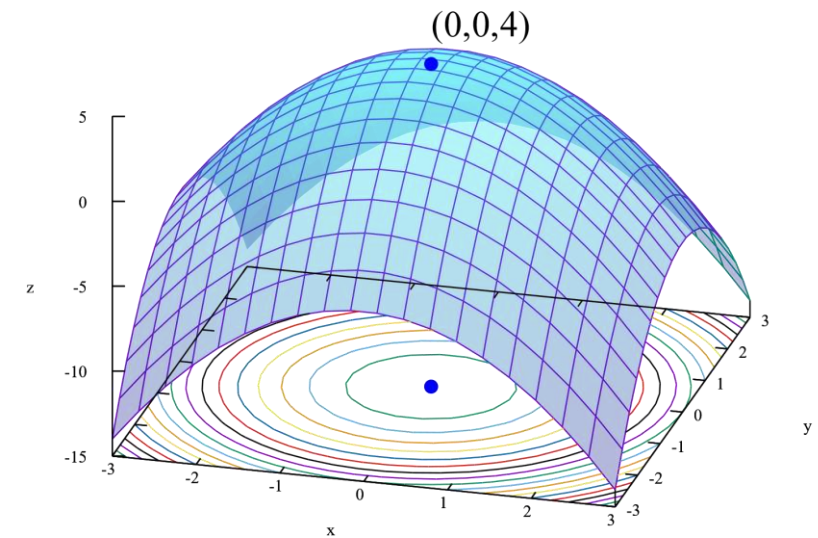


$(x_i, y_i)$

Q) Which line is more closer to the point?

# Solving Nonlinear Equation using Nonlinear Optimization

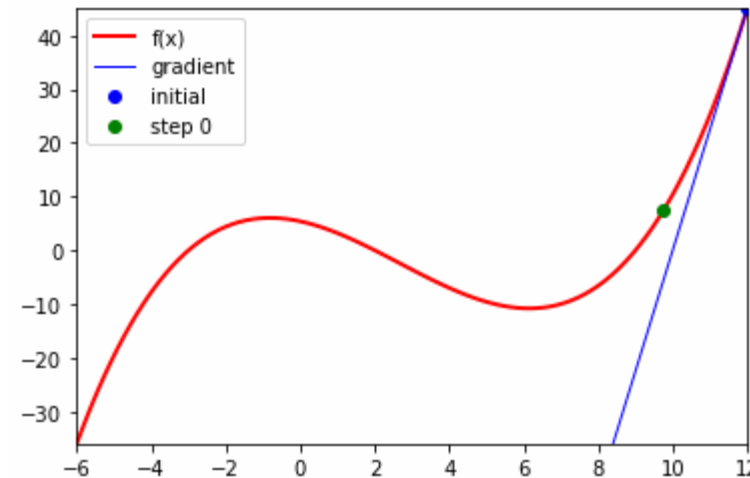
- Nonlinear optimization is the process of solving an optimization problem where some of the constraints or the objective function are **nonlinear**.
  - Alias: Nonlinear programming (NLP)
  - Mathematically,  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x})$  subject to  $g_i(\mathbf{x}) \leq 0$  for each  $i \in \{1, \dots, m\}$   
 $h_j(\mathbf{x}) = 0$  for each  $j \in \{1, \dots, p\}$   
 $\mathbf{x} \in X$  ( $X$  is a subset of  $\mathbb{R}^n$ )
    - $f(\mathbf{x})$ : The real-valued objective function
    - $g_i(\mathbf{x})$ : The  $i$ -th real-valued inequality constraint function
    - $h_j(\mathbf{x})$ : The  $j$ -th real-valued equality constraint function
  - Example) The objective function  $f(x, y) = 4 - (x^2 + y^2)$  is nonlinear.



# Nonlinear Optimization

- Gradient descent

- A **first-order iterative algorithm** for finding a local minimum of a differentiable function by pursuing to the opposite direction of the gradient of the function at the current point
- Mathematically,  $x_{t+1} = x_t - \gamma f'(x_t)$ 
  - $\gamma$ : The step size (a.k.a. learning rate)



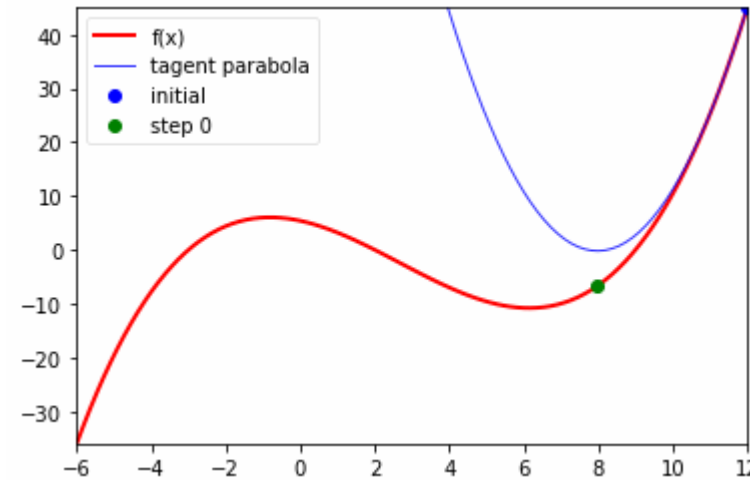
- Note) Stochastic gradient descent (SGD)

- SGD uses an approximated gradient (calculated from a randomly selected subset of the given data) instead of the actual gradient (calculated from the entire data).
- SGD variants: AdaGrad, RMSProp, Adam, ...

# Nonlinear Optimization

## ▪ Newton's method

- A **second-order iterative algorithm** for finding a local minimum of a differentiable function by pursuing the minima of the locally approximated parabola of the function at the current point
- Mathematically,  $x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$ 
  - The step size is **not** required.



## ▪ Note) Gauss-Newton method

- A special case for non-linear least squares problems
  - When the function has a form of  $f(x) = r^2(x)$ ,
  - Newton's method becomes  $x_{t+1} = x_t - \frac{r(x_t)}{r'(x_t)}$  (without the 2nd-order derivative)



# Solving Nonlinear Equation using Nonlinear Optimization

- **Example) Line fitting from more than two points such as (1, 4), (4, 2), and (7, 1), using [SciPy](#)**
  - Unknown: Line parameters  $a$ ,  $b$ , and  $c$  (line representation:  $ax + by + c = 0$ )
  - Cost function:  $f(a, b, c) = \sum_i \left( \frac{ax_i + by_i + c}{\sqrt{a^2 + b^2}} \right)^2$
  - Optimizer: [Gauss-Newton method](#) (least squares)

```
import numpy as np
from scipy.optimize import least_squares

def geometric_error(line, pts):
    a, b, c = line
    err = [(a*x + b*y + c) / np.sqrt(a*a + b*b) for (x, y) in pts]
    return err

pts = [(1, 4), (4, 2), (7, 1)]
line_init = [1, 1, 0]
result = least_squares(geometric_error, line_init, args=(pts,))
line = result['x'] / -result['x'][1] # [-0.50372575, -1., 4.34823633]
```

# Summary (Line Fitting Problems)

## ▪ Linear equations

- Inhomogeneous equations  $A\mathbf{x} = \mathbf{b} \rightarrow \mathbf{x} = A^\dagger \mathbf{b}$  where  $A^\dagger$  is a pseudo-inverse
  - Formulation)  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|A\mathbf{x} - \mathbf{b}\|_2^2$
- Homogeneous equation  $A\mathbf{x} = \mathbf{0} \rightarrow \mathbf{x}$  is the last row of  $V^\top$  where  $A = USV^\top$  (from singular value decomposition)
  - Formulation)  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|A\mathbf{x}\|^2$  with  $\|\mathbf{x}\|^2 = 1$  condition

## ▪ Nonlinear equations

- Nonlinear optimization: ~~Gradient descent, Newton's method, and Gauss-Newton method (for  $f(x) = r^2(x)$ )~~
  - Formulation)  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x})$
  - Note) Let's use magic tools such as [scipy.optimize](#) and [Ceres Solver](#) for better performance.

# Solving Linear Equations in 3D Vision

## ▪ Affine transformation estimation

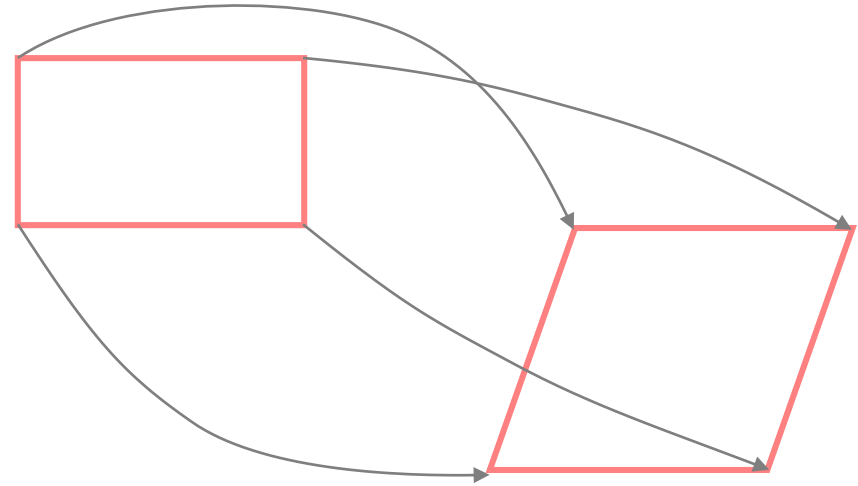
- Unknown: Affine transformation  $H$  (6 DOF)
- Given: Point correspondence  $(\mathbf{x}_1, \mathbf{x}'_1), \dots, (\mathbf{x}_n, \mathbf{x}'_n)$
- Constraints:  $n$  x affine transformation  $\mathbf{x}'_i = H\mathbf{x}_i$
- Solutions ( $n \geq 3$ )
  - OpenCV: `cv.getAffineTransform()`

### • Affine transformation estimation

– Affine transformation: 
$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix}$$

– For  $n$  pairs of points, 
$$\begin{bmatrix} x_1 & y_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1 & y_1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 0 & 0 & 1 & 0 \\ 0 & 0 & x_n & y_n & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \\ t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ \vdots \\ x'_n \\ y'_n \end{bmatrix} \rightarrow \mathbf{Ax} = \mathbf{b}$$

- Solve  $\mathbf{Ax} = \mathbf{b}$  and organize  $H$  from  $\mathbf{x}$



# Solving Linear Equations in 3D Vision

- Affine transformation estimation [affine\_estimation\_implement.py]

```
import numpy as np
import cv2 as cv
```

```
def getAffineTransform(src, dst):
    if len(src) == len(dst):
        # Solve 'Ax = b'
        A, b = [], []
        for p, q in zip(src, dst):
            A.append([p[0], p[1], 0, 0, 1, 0])
            A.append([0, 0, p[0], p[1], 0, 1])
            b.append(q[0])
            b.append(q[1])
        x = np.linalg.pinv(A) @ b

        # Reorganize `x` as a matrix
        H = np.array([[x[0], x[1], x[4]], [x[2], x[3], x[5]]])
        return H
```

```
if __name__ == '__main__':
    src = np.array([[115, 401], [776, 180], [330, 793]], dtype=np.float32)
    dst = np.array([[0, 0], [900, 0], [0, 500]], dtype=np.float32)

    my_H = getAffineTransform(src, dst)
    cv_H = cv.getAffineTransform(src, dst) # Note) It accepts only 3 pairs of points.
    ...
```

$$\begin{bmatrix} x_1 & y_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1 & y_1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 0 & 0 & 1 & 0 \\ 0 & 0 & x_n & y_n & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \\ t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ \vdots \\ x'_n \\ y'_n \end{bmatrix}$$

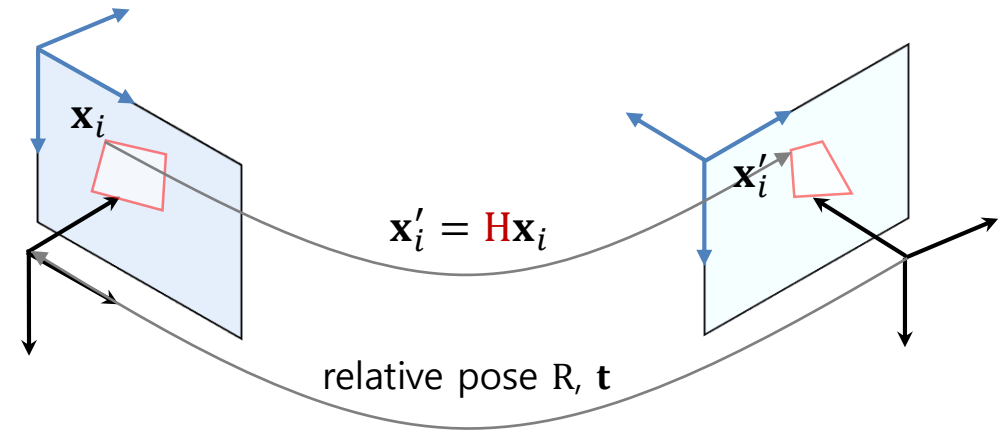
# Solving Linear Equations in 3D Vision

## Planar homography estimation

- Unknown: **Planar homography**  $H$  (8 DOF  $\leftarrow$  up to scale)
- Given: Point correspondence  $(\mathbf{x}_1, \mathbf{x}'_1), \dots, (\mathbf{x}_n, \mathbf{x}'_n)$
- ~~Constraints:  $n \times$  projective transformation  $\mathbf{x}'_i = H\mathbf{x}_i$~~

$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ v_1 & v_2 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix} \rightarrow \mathbf{x}'_i = \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} = \frac{1}{w_i} \begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix}$$

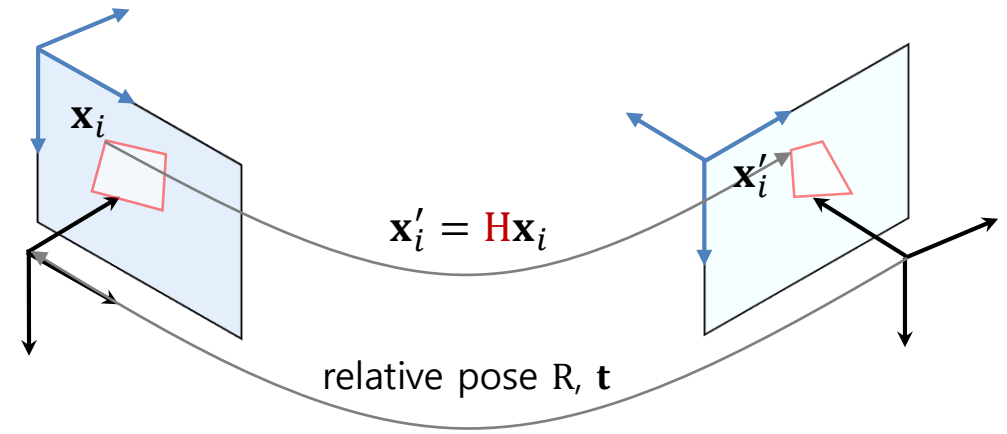
- Modified constraints:  $n \times$  transformation  $\mathbf{x}'_i \sim H\mathbf{x}_i$  (similarity) or  $\mathbf{x}'_i = \lambda_i H\mathbf{x}_i$  or  $\mathbf{x}'_i \times (H\mathbf{x}_i) = \mathbf{0}$ 
  - Note) The last element of  $\mathbf{x}_i$  and  $\mathbf{x}'_i$  in homogeneous coordinates are not necessary to be 1.
- Solutions ( $n \geq 4$ )  $\rightarrow$  4-point algorithm
  - OpenCV: `cv.getPerspectiveTransform()` and `cv.findHomography()`



# Solving Linear Equations in 3D Vision

## Planar homography estimation

- Unknown: **Planar homography**  $H$  (8 DOF ← up to scale)
- Given: Point correspondence  $(\mathbf{x}_1, \mathbf{x}'_1), \dots, (\mathbf{x}_n, \mathbf{x}'_n)$
- Constraints:  $n$  x transformation  $\mathbf{x}'_i \times (H\mathbf{x}_i) = \mathbf{0}$
- Solutions ( $n \geq 4$ ) → 4-point algorithm
  - OpenCV: `cv.getPerspectiveTransform()` and `cv.findHomography()`
  - 4-point algorithm**



$$[x'_i \quad y'_i \quad w'_i] \times \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ w_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{aligned} w'_i(x_i h_{21} + y_i h_{22} + w_i h_{23}) - y'_i(x_i h_{31} + y_i h_{32} + w_i h_{33}) &= 0 \\ w'_i(x_i h_{11} + y_i h_{12} + w_i h_{13}) - x'_i(x_i h_{31} + y_i h_{32} + w_i h_{33}) &= 0 \\ y'_i(x_i h_{11} + y_i h_{12} + w_i h_{13}) - x'_i(x_i h_{21} + y_i h_{22} + w_i h_{23}) &= 0 \end{aligned}$$

$$\text{For } n \text{ pairs, } \begin{bmatrix} 0 & 0 & 0 & w'_1 x_1 & w'_1 y_1 & w'_1 w_1 & -y'_1 x_1 & -y'_1 y_1 & -y'_1 w_1 \\ w'_1 x_1 & w'_1 y_1 & w'_1 w_1 & 0 & 0 & 0 & -x'_1 x_1 & -x'_1 y_1 & -x'_1 w_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & w'_n x_n & w'_n y_n & w'_n w_n & -y'_n x_n & -y'_n y_n & -y'_n w_n \\ w'_n x_n & w'_n y_n & w'_n w_n & 0 & 0 & 0 & -x'_n x_n & -x'_n y_n & -x'_n w_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = 0 \rightarrow A\mathbf{x} = 0$$

- Solve  $A\mathbf{x} = 0$  and reorganize  $H$  (3x3 matrix)

# Solving Linear Equations in 3D Vision

- Planar homography estimation [homography\_estimation\_implement.py]

```
import numpy as np
import cv2 as cv

def getPerspectiveTransform(src, dst):
    if len(src) == len(dst):
        # Make homogeneous coordinates if necessary
        if src.shape[1] == 2:
            src = np.hstack((src, np.ones((len(src), 1), dtype=src.dtype)))
        if dst.shape[1] == 2:
            dst = np.hstack((dst, np.ones((len(dst), 1), dtype=dst.dtype)))

        # Solve 'Ax = 0'
        A = []
        for p, q in zip(src, dst):
            A.append([0, 0, 0, q[2]*p[0], q[2]*p[1], q[2]*p[2], -q[1]*p[0], -q[1]*p[1], -q[1]*p[2]])
            A.append([q[2]*p[0], q[2]*p[1], q[2]*p[2], 0, 0, 0, -q[0]*p[0], -q[0]*p[1], -q[0]*p[2]])
        _, _, Vt = np.linalg.svd(A, full_matrices=True)
        x = Vt[-1]

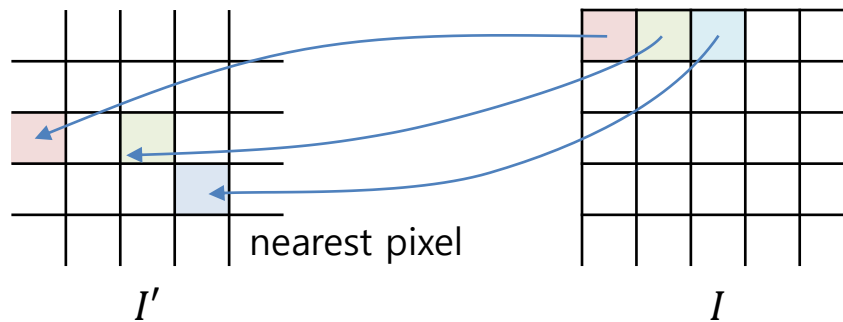
        # Reorganize `x` as a matrix
        H = x.reshape(3, -1) / x[-1] # Normalize the last element as 1
        return H

if __name__ == '__main__':
    ...
```

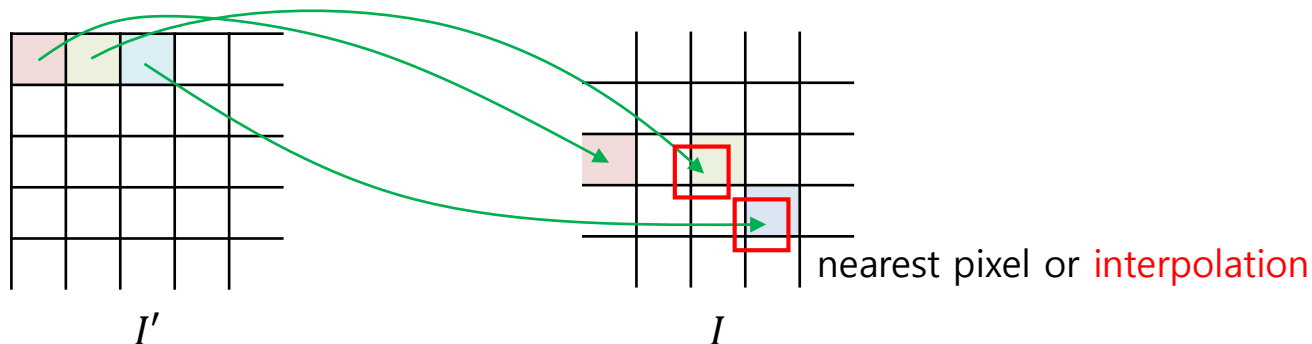
# Appendix) Image Warping

## ▪ Example) Image warping using homography

- Target: Transformed image  $I'$
- Source: Original image  $I$
- Relationship:  $I'(\mathbf{x}') = I(\mathbf{x})$  where  $\mathbf{x}' = H\mathbf{x}$
- Method #1) Select a pair of points,  $\mathbf{x} \in \{(0,0), (0,1), \dots, (1,0), (1,1), \dots\}$  and  $\mathbf{x}' = H\mathbf{x}$



- Method #2) Select a pair of points,  $\mathbf{x}' \in \{(0,0), (0,1), \dots, (1,0), (1,1), \dots\}$  and  $\mathbf{x} = H^{-1}\mathbf{x}'$





## ▪ Example) Image warping using homography [image\_warping\_implement.py]

```
import cv2 as cv
```

```
...
```

```
def warpPerspective1(src, H, dst_size):
```

```
    # Generate an empty image
```

```
    width, height = dst_size
```

```
    channel = src.shape[2] if src.ndim > 2 else 1
```

```
    dst = np.zeros((height, width, channel), dtype=src.dtype)
```

```
    # Copy a pixel from `src` to `dst`
```

```
    for py in range(img.shape[0]):
```

```
        for px in range(img.shape[1]):
```

```
            q = H @ [px, py, 1]
```

```
            qx, qy = int(q[0]/q[-1] + 0.5), int(q[1]/q[-1] + 0.5)
```

```
            if qx >= 0 and qy >= 0 and qx < width and qy < height:
```

```
                dst[qy, qx] = src[py, px]
```

```
    return dst
```

```
def warpPerspective2(src, H, dst_size):
```

```
    # Generate an empty image
```

```
    width, height = dst_size
```

```
    channel = src.shape[2] if src.ndim > 2 else 1
```

```
    dst = np.zeros((height, width, channel), dtype=src.dtype)
```

```
    # Copy a pixel from `src` to `dst`
```

```
    H_inv = np.linalg.inv(H)
```

```
    for qy in range(height):
```

```
        for qx in range(width):
```

```
            p = H_inv @ [qx, qy, 1]
```

```
            px, py = int(p[0]/p[-1] + 0.5), int(p[1]/p[-1] + 0.5)
```

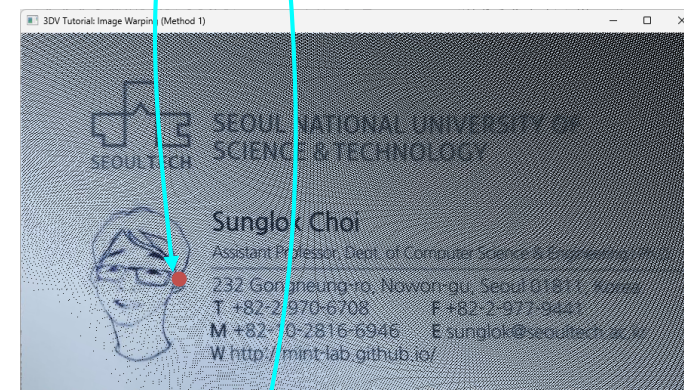
```
            if px >= 0 and py >= 0 and px < img.shape[1] and py < img.shape[0]:
```

```
                dst[qy, qx] = src[py, px]
```

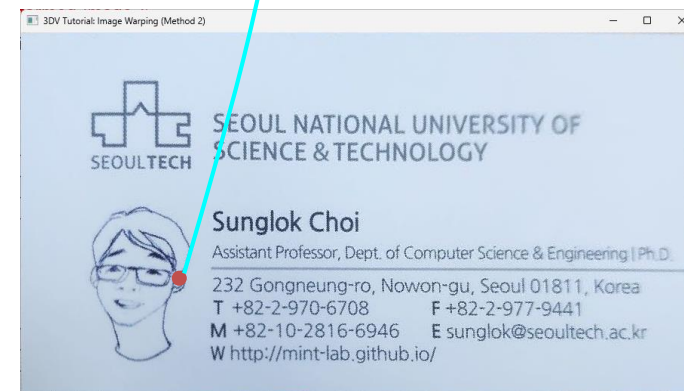
```
    return dst
```



$$\mathbf{x}' = \mathbf{H}\mathbf{x}$$



$$\mathbf{x} = \mathbf{H}^{-1}\mathbf{x}'$$



# Solving Linear Equations in 3D Vision

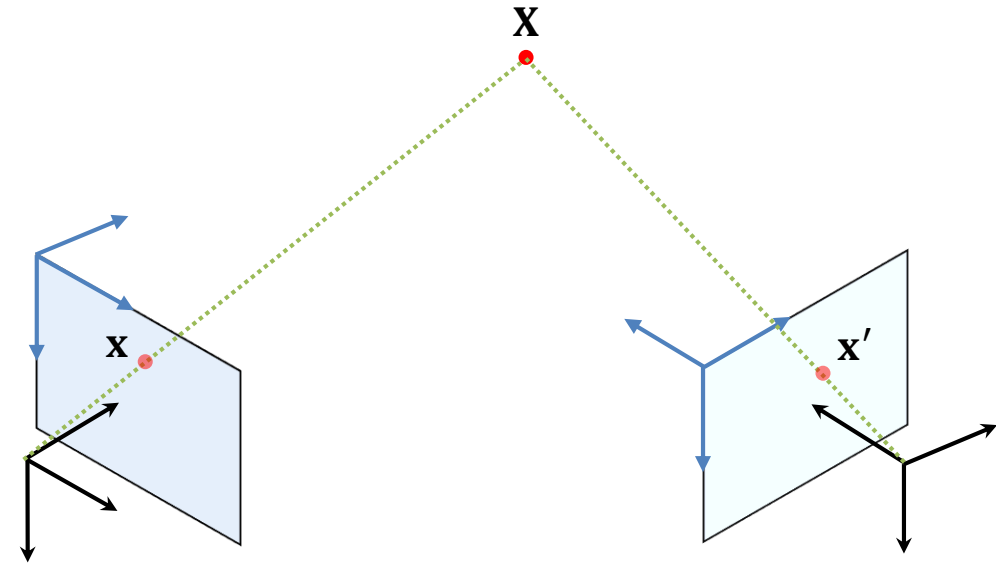
## ▪ Triangulation (point localization)

- Unknown: **Position of a 3D point  $\mathbf{X}$**  (3 DoF)
- Given: Point correspondence  $(\mathbf{x}, \mathbf{x}')$  and projection matrices  $(P, P')$
- Constraints:  $\mathbf{x} = P\mathbf{X}$ ,  $\mathbf{x}' = P'\mathbf{X}$
- Solutions
  - OpenCV: `cv.triangulatePoints()`

### • Linear triangulation

- Projection:  $\mathbf{x} = P\mathbf{X} \rightarrow \mathbf{x} \times (P\mathbf{X}) = \mathbf{0} \rightarrow$   
$$\begin{aligned} x(\mathbf{p}_3^\top \mathbf{X}) - w(\mathbf{p}_1^\top \mathbf{X}) &= 0 \\ y(\mathbf{p}_3^\top \mathbf{X}) - w(\mathbf{p}_2^\top \mathbf{X}) &= 0 \\ x(\mathbf{p}_2^\top \mathbf{X}) - y(\mathbf{p}_1^\top \mathbf{X}) &= 0 \end{aligned}$$
 where  $P = \begin{bmatrix} \mathbf{p}_1^\top \\ \mathbf{p}_2^\top \\ \mathbf{p}_3^\top \end{bmatrix}$

- Solve  $A\mathbf{X} = \mathbf{0}$  where  $A = \begin{bmatrix} x\mathbf{p}_3^\top - w\mathbf{p}_1^\top \\ y\mathbf{p}_3^\top - w\mathbf{p}_2^\top \\ x'\mathbf{p}'_3{}^\top - w'\mathbf{p}'_1{}^\top \\ y'\mathbf{p}'_3{}^\top - w'\mathbf{p}'_2{}^\top \end{bmatrix}$



- **Triangulation** (point localization) [triangulation\_implement.py]

```
import numpy as np
import cv2 as cv
```

```
def triangulatePoints(P0, P1, pts0, pts1):
    Xs = []
    for (p, q) in zip(pts0.T, pts1.T):
        # Solve 'AX = 0'
        A = np.vstack((p[0] * P0[2] - P0[0],
                        p[1] * P0[2] - P0[1],
                        q[0] * P1[2] - P1[0],
                        q[1] * P1[2] - P1[1]))
        _, _, Vt = np.linalg.svd(A, full_matrices=True)
        Xs.append(Vt[-1])
    return np.vstack(Xs).T
```

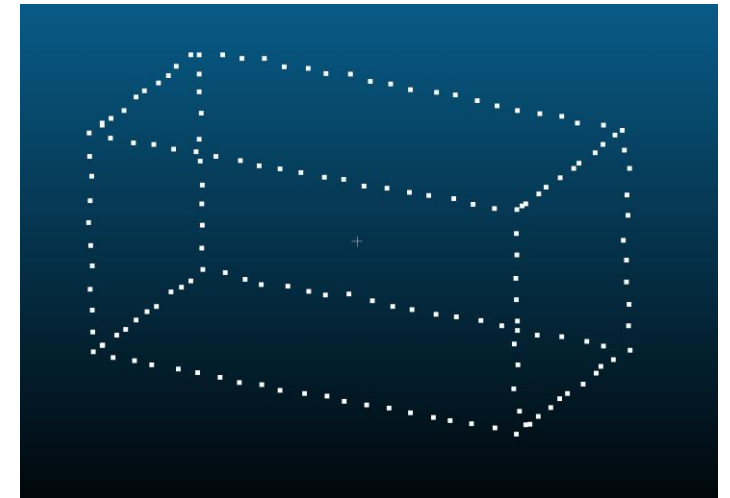
```
if __name__ == '__main__':
    f, cx, cy = 1000., 320., 240.
    pts0 = np.loadtxt('../data/image_formation0.xyz')[:, :2]
    pts1 = np.loadtxt('../data/image_formation1.xyz')[:, :2]
    output_file = 'triangulation_implement.xyz'

    # Estimate relative pose of two view
    F, _ = cv.findFundamentalMat(pts0, pts1, cv.FM_8POINT)
    K = np.array([[f, 0, cx], [0, f, cy], [0, 0, 1]])
    E = K.T @ F @ K
    _, R, t, _ = cv.recoverPose(E, pts0, pts1)
```

$$A = \begin{bmatrix} x\mathbf{p}_3^\top - w\mathbf{p}_1^\top \\ y\mathbf{p}_3^\top - w\mathbf{p}_2^\top \\ x'\mathbf{p}'_3^\top - w'\mathbf{p}'_1^\top \\ y'\mathbf{p}'_3^\top - w'\mathbf{p}'_2^\top \end{bmatrix}$$

- **Triangulation** (point localization) [triangulation\_implement.py]

```
if __name__ == '__main__':  
    f, cx, cy = 1000., 320., 240.  
    pts0 = np.loadtxt('../data/image_formation0.xyz')[:, :2]  
    pts1 = np.loadtxt('../data/image_formation1.xyz')[:, :2]  
    output_file = 'triangulation_implement.xyz'  
  
    # Estimate relative pose of two view  
    F, _ = cv.findFundamentalMat(pts0, pts1, cv.FM_8POINT)  
    K = np.array([[f, 0, cx], [0, f, cy], [0, 0, 1]])  
    E = K.T @ F @ K  
    _, R, t, _ = cv.recoverPose(E, pts0, pts1)  
  
    # Reconstruct 3D points (triangulation)  
    P0 = K @ np.eye(3, 4, dtype=np.float32)  
    Rt = np.hstack((R, t))  
    P1 = K @ Rt  
    X = triangulatePoints(P0, P1, pts0.T, pts1.T)  
    X /= X[3]  
    X = X.T  
  
    # Write the reconstructed 3D points  
    np.savetxt(output_file, X)
```

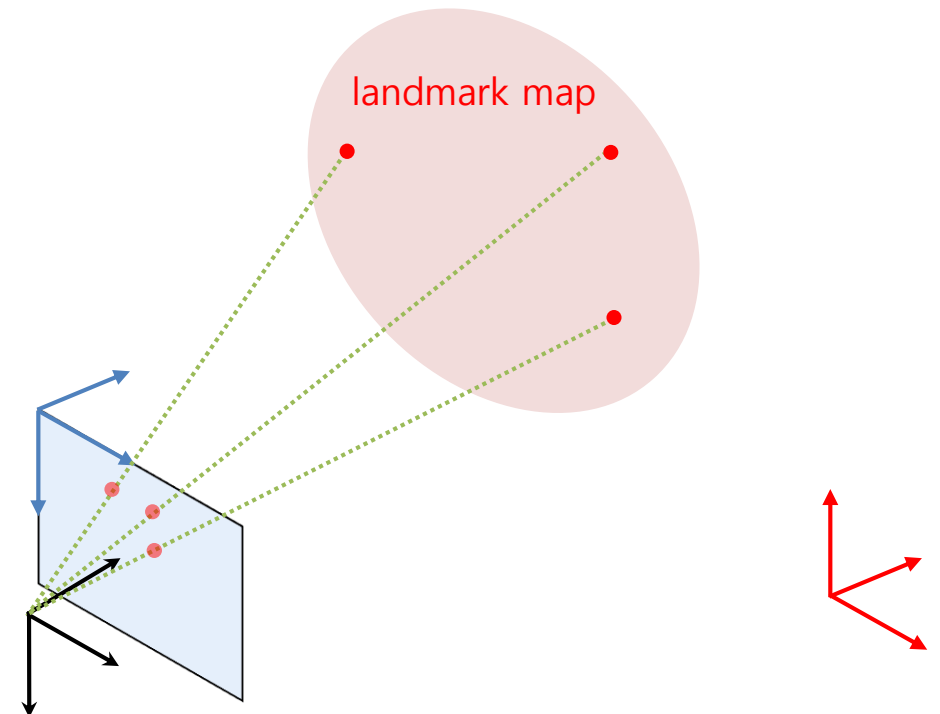


triangulation\_implement.xyz

# Solving Nonlinear Equations in 3D Vision

- **Absolute camera pose estimation** ([perspective-n-point](#); PnP)

- Unknown: Camera pose  $R$  and  $t$  (6 DOF)
- Given: 3D points  $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n$ , their projected points  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ , and camera matrix  $K$
- ~~Constraints:  $n \times$  projection  $\mathbf{x}_i = K[R|t]\mathbf{X}_i$~~
- More general constraints:  $\mathbf{x}_i = \text{proj}(\mathbf{X}_i; R, t)$  where  $\mathbf{x}_i = [x_i, y_i]^T$ 
  - Note) The projection  $\text{proj}$  generates 2D points on the image plane (not in homogeneous coordinates) considering nonlinear lens distortion.
- Solutions ( $n \geq 3$ )  $\rightarrow$  3-point algorithm
  - OpenCV: `cv.solvePnP()` and `cv.solvePnPRansac()`
  - **Iterative 3-point algorithm** using local optimization
    - Cost function
$$\hat{R}, \hat{t} = \underset{R, t}{\operatorname{argmin}} \sum_{i=1}^n \|\mathbf{x}_i - \text{proj}(\mathbf{X}_i; R, t)\|_2^2$$
    - Optimizer: Gauss-Newton method



# Solving Nonlinear Equations in 3D Vision

- **Absolute camera pose estimation** ([perspective-n-point](#); PnP) [pose\_estimation\_implement.py]

```
import numpy as np
from scipy.optimize import least_squares
from scipy.spatial.transform import Rotation
import cv2 as cv
```

```
def project_no_distort(X, rvec, t, K):
    R = Rotation.from_rotvec(rvec.flatten()).as_matrix()
    XT = X @ R.T + t # Transpose of 'X = R @ X + t'
    xT = XT @ K.T # Transpose of 'x = KX'
    xT = xT / xT[:, -1].reshape((-1, 1)) # Normalize
    return xT[:, 0:2]
```

```
def reproject_error_pnp(unknown, X, x, K):
    rvec, tvec = unknown[:3], unknown[3:]
    xp = project_no_distort(X, rvec, tvec, K)
    err = x - xp
    return err.ravel()
```

```
def solvePnP(obj_pts, img_pts, K):
    unknown_init = np.array([0, 0, 0, 0, 0, 1.]) # Sequence: rvec(3), tvec(3)
    result = least_squares(reproject_error_pnp, unknown_init, args=(obj_pts, img_pts, K))
    return result['success'], result['x'][:3], result['x'][3:]
```

# Solving Nonlinear Equations in 3D Vision

## ▪ Camera calibration

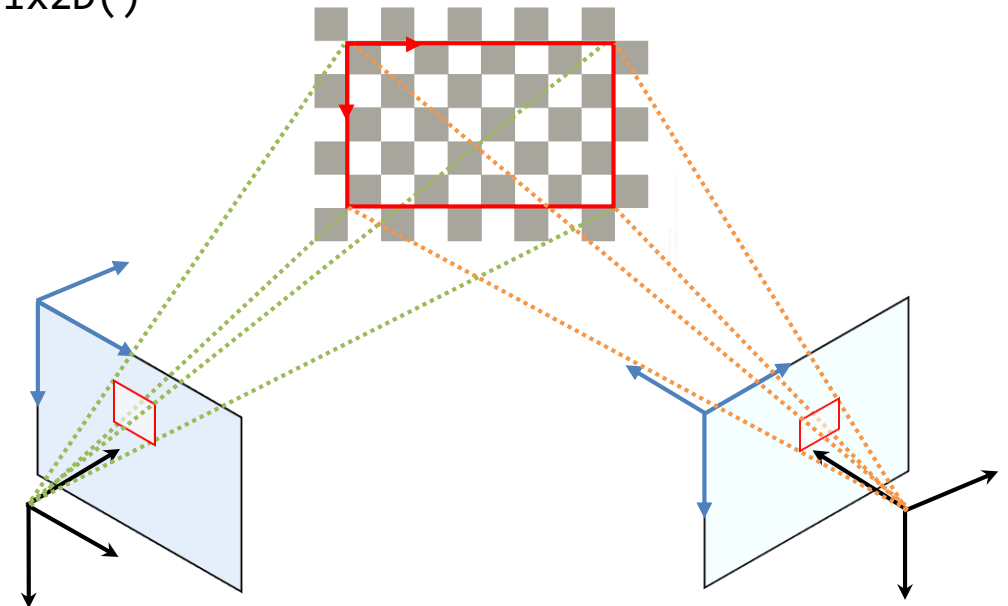
- Unknown: **Intrinsic +  $m$  x extrinsic parameters** ( $3^* + m \times 6$  DOF)
- Given: 3D points  $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n$  and their projected points,  $\mathbf{x}_i^j$ , on the  $j$ -th image
  - Note)  $m$ : the number of images,  $n$ : the number of 3D points
- ~~Constraints:  $m \times n \times$  projection  $\mathbf{x}_i^j = \mathbf{K} [\mathbf{R}_j | \mathbf{t}_j] \mathbf{X}_i$~~
- More general constraints:  $\mathbf{x}_i^j = \text{proj}(\mathbf{X}_i; \mathbf{K}_j, \mathbf{R}_j, \mathbf{t}_j)$  where  $\mathbf{x}_i = [x_i, y_i]^\top$
- Solutions [\[Tools\]](#)

- OpenCV: `cv.calibrateCamera()` and `cv.initCameraMatrix2D()`
- **Camera calibration** using local optimization

- Cost function

$$\hat{\mathbf{K}}, \hat{\mathbf{R}}_{\dots}, \hat{\mathbf{t}}_{\dots} = \underset{\mathbf{K}, \mathbf{R}_{\dots}, \mathbf{t}_{\dots}}{\operatorname{argmin}} \sum_{j=1}^m \sum_{i=1}^n \left\| \mathbf{x}_i^j - \text{proj}(\mathbf{X}_i; \mathbf{K}, \mathbf{R}_j, \mathbf{t}_j) \right\|_2^2$$

- Optimizer: Gauss-Newton method



▪ **Camera calibration** [camera\_calibration\_implement.py]

```
import numpy as np
```

```
...
```

```
def fxcycy_to_K(f, cx, cy):  
    return np.array([[f, 0, cx], [0, f, cy], [0, 0, 1]])
```

```
def reproject_error_calib(unknown, Xs, xs):  
    K = fxcycy_to_K(*unknown[0:3])  
    err = []  
    for j in range(len(xs)):  
        offset = 3 + 6 * j  
        rvec, tvec = unknown[offset:offset+3], unknown[offset+3:offset+6]  
        xp = project_no_distort(Xs[j], rvec, tvec, K)  
        err.append(xs[j] - xp)  
    return np.vstack(err).ravel()
```

```
def calibrateCamera(obj_pts, img_pts, img_size):  
    img_n = len(img_pts)  
    unknown_init = np.array([img_size[0], img_size[0]/2, img_size[1]/2] \  
                             + img_n * [0, 0, 0, 0, 0, 1.]) # Sequence: f, cx, cy, img_n * (rvec, tvec)  
    result = least_squares(reproject_error_calib, unknown_init, args=(obj_pts, img_pts))  
    K = fxcycy_to_K(*result['x'][0:3])  
    rvecs = [result['x'][(6*i+3):(6*i+6)] for i in range(img_n)]  
    tvecs = [result['x'][(6*i+6):(6*i+9)] for i in range(img_n)]  
    return result['cost'], K, np.zeros(5), rvecs, tvecs
```



# Summary (3D Vision Problems)

## ▪ Linear equations

- Inhomogeneous equations  $A\mathbf{x} = \mathbf{b} \rightarrow \mathbf{x} = A^\dagger \mathbf{b}$  where  $A^\dagger$  is a pseudo-inverse
  - Formulation)  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|A\mathbf{x} - \mathbf{b}\|_2^2$
  - Example) Affine transformation estimation
- Homogeneous equation  $A\mathbf{x} = \mathbf{0} \rightarrow \mathbf{x}$  is the last row of  $V^\top$  where  $A = USV^\top$  (from singular value decomposition)
  - Formulation)  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|A\mathbf{x}\|^2$  with  $\|\mathbf{x}\|^2 = 1$  condition
  - Example) Planar homography estimation
  - Example) Triangulation

## ▪ Nonlinear equations

- Nonlinear optimization: Magic tools such as [scipy.optimize](#) and [Ceres Solver](#)
  - Formulation)  $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x})$
  - Example) Absolute camera pose estimation (PnP)
  - Example) Camera calibration