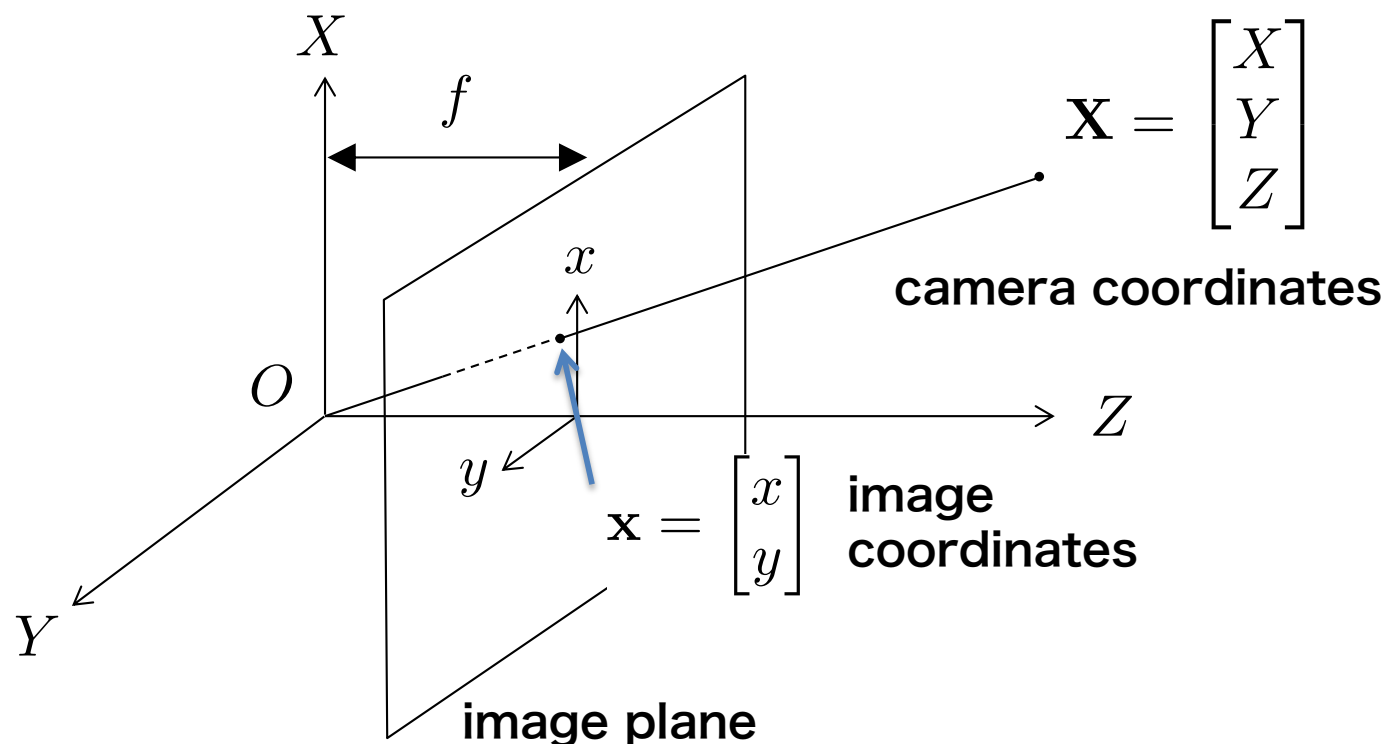


# コンピュータビジョン カメラモデル

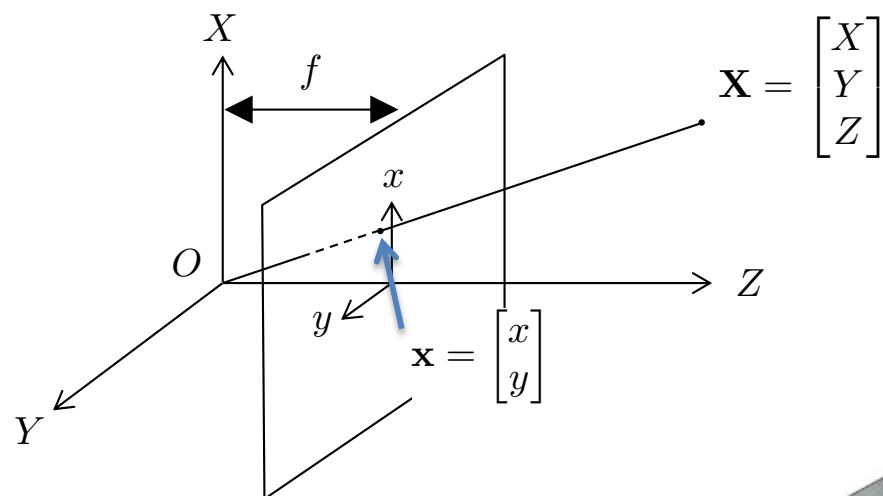
# カメラの幾何学的モデル (1/3)

- Pinhole camera model (or central projection)

$$x = f \frac{X}{Z} \quad y = f \frac{Y}{Z} \quad \Rightarrow \quad \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

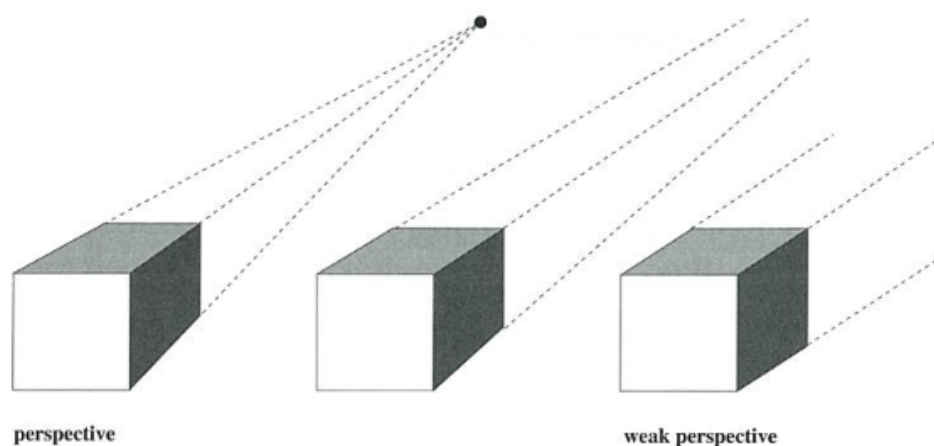


# 焦点距離と画角の関係



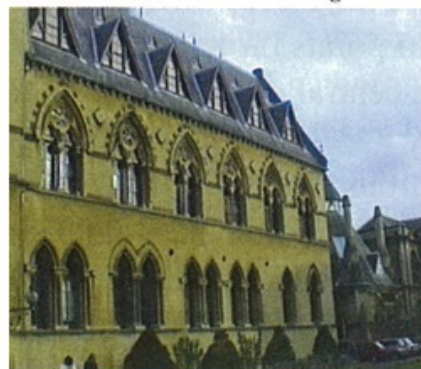
$f$  : focal length(焦点距離)

注：画像面と投影中心間の距離



| 仕様表               |  |
|-------------------|--|
| イメージセンサー          |  |
| センサータイプ           | 35mmフルサイズ (35.8mm x 23.9mm) "Exmor" CMOS<br>センサー アスペクト比3:2   |
| 総画素数              | 約2470万画素   |
| 有効画素数             | 約2430万画素   |
| レンズ               |  |
| レンズタイプ            | Carl Zeiss "Sonnar-T*" レンズ<br>(レンズ構成: 7群8枚(AAレンズ含む非球面レンズ3枚)) |
| F値(開放)            | F2   |
| 虹彩枚数              | 9枚   |
| 焦点距離              | f=35mm   |
| 35mm換算値(静止画3:2時)  | f=35mm   |
| 35mm換算値(静止画16:9時) | f=37mm   |
| 35mm換算値(動画16:9時)  | f=44mm(手ブレ補正オン) f=37mm(手ブレ補正オフ)                              |

<http://www.sony.jp/cyber-shot/products/DSC-RX1/spec.html>

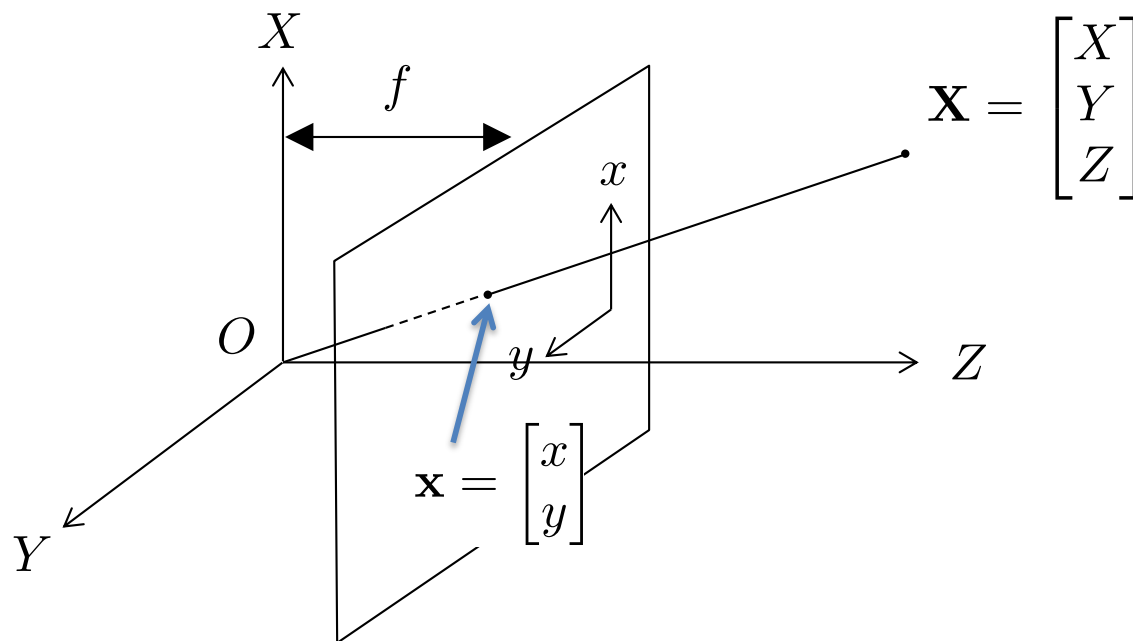


[Hartley-Zisserman03]

## カメラの幾何学的モデル (2/3)

- Offset of projection center

$$x = f \frac{X}{Z} + x_0 \quad y = f \frac{Y}{Z} + y_0 \quad \Rightarrow \quad \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \begin{bmatrix} f & 0 & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$



$$\mathbf{K} = \begin{bmatrix} f & 0 & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix of  
internal parameters



# カメラの幾何学的モデル (3/3)

- 最も一般化したもの
  - Non-square pixels → aspect ratio
  - Non-perpendicular image plane

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \begin{bmatrix} f & sf & x_0 \\ 0 & \alpha f & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

$s$  : skew  
 $\alpha$  : aspect ratio

- Matrix of internal parameters
  - 5 parameters = 5DoFs
  - Upper triangular

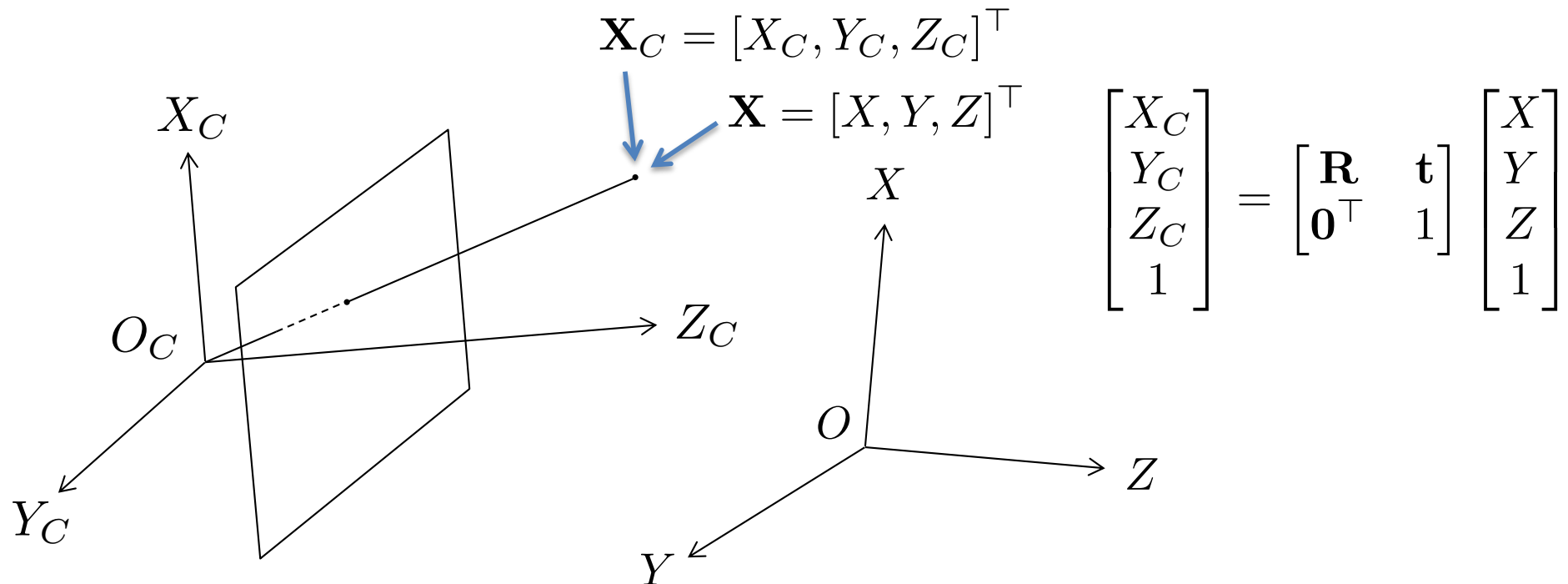
$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \mathbf{K} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} f & sf & x_0 \\ 0 & \alpha f & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

# カメラの姿勢

- 空間に固定された**世界(world)座標系**とカメラに固定された**カメラ座標系**の間の座標変換

$$\mathbf{X}_C = \mathbf{R}\mathbf{X} + \mathbf{t}$$

$\mathbf{R}$  :  $3 \times 3$  回転行列     $\mathbf{t}$  : 並進ベクトル



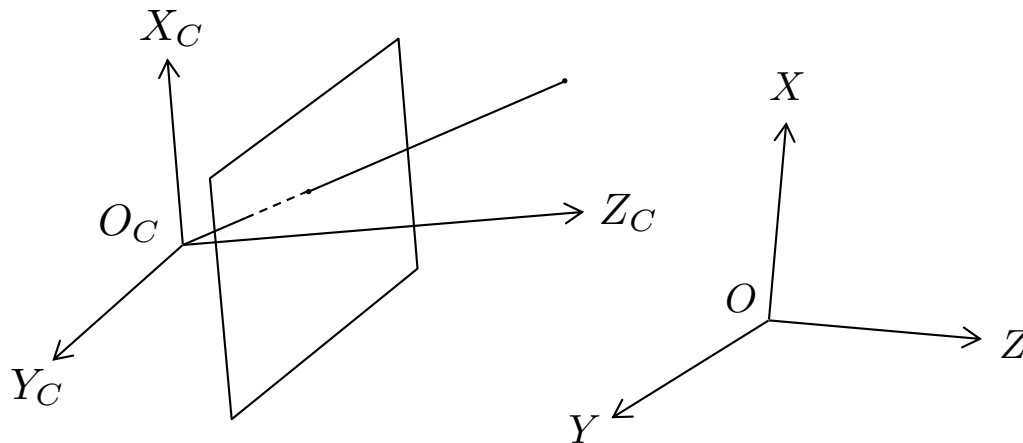
# Camera matrix

- 世界座標 → カメラ座標 × カメラ座標 → 画像座標

$$\mathbf{P} = \mathbf{K}[\mathbf{R} \mid \mathbf{t}] : \text{camera matrix (3x4)}$$

$\underbrace{\hspace{1cm}}$   $\underbrace{\hspace{1cm}}$   
 internal    external parameters

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \mathbf{P} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad \Longleftarrow \quad \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \underbrace{\mathbf{K}[\mathbf{R} \mid \mathbf{t}]}_{\substack{\text{internal} \quad \text{external} \\ \text{parameters} \quad \text{parameters}}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$



$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \mathbf{K} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \quad \mathbf{X}_C = \mathbf{R}\mathbf{X} + \mathbf{t}$$

# Rotation matrix

- 回転行列の表現方法は複数ある
  - 回転軸角度, Euler角, quaternion
- 回転軸角度表現
  - 向きが回転軸方向, 長さが回転角度の3成分ベクトルで表現
  - ベクトルから  $3 \times 3$  回転行列を計算 (Rodrigues formula)
  - `R, jac = cv2.Rodrigues(v)` (逆変換も可能)

$$\mathbf{R} = \mathbf{I} + \sin \theta [\mathbf{v}]_{\times} + (1 - \cos \theta)(\mathbf{v}\mathbf{v}^{\top} - \mathbf{I})$$

$$[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$$

# カメラモデルの確認

- 空間の立方体を画像面に投影し描画する関数 `draw_cube()`

```
import numpy, cv2 cam_model.py

pt = numpy.array([[0,0,0], [0,0,-1], [0,1,0], [1,0,0],¥
                  [0,1,-1],[1,0,-1], [1,1,0], [1,1,-1]],float)
ll = [[0,1],[0,2],[0,3],[1,4],[1,5],[2,4],¥
      [2,6],[3,5],[3,6],[4,7],[5,7],[6,7]]

def draw_cube(image, K, base, length, Rmat, tvec):
    #print base.shape, cubept[0:1,:].T.shape, tvec.shape
    for l in ll:
        ipt1 = K.dot(length*Rmat.dot(pt[l[0]:l[0]+1,:].T+base)¥
                + tvec)
        ipt2 = K.dot(length*Rmat.dot(pt[l[1]:l[1]+1,:].T+base)¥
                + tvec)
        cv2.line(image, (ipt1[0]/ipt1[2], ipt1[1]/ipt1[2]),¥
                  (ipt2[0]/ipt2[2], ipt2[1]/ipt2[2]), (0,0,255), 3)

    ...
```

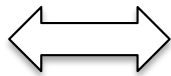
# カメラモデルの確認

```
if __name__ == "__main__":cam_model.py  
  
    focal, z0 = 1000.0, 500.0 # z0 must be greater than 100  
    cube_orig = numpy.array([[ -0.5, -0.5, 0.5]], float).T  
  
    ang = 0.0  
    while 1:  
        rot1, jac1 = cv2.Rodrigues(numpy.array([[0, ang, 0]], float))  
        rot2, jac2 = cv2.Rodrigues(numpy.array([[0.4, 0, 0]], float))  
        ang = ang + numpy.pi/100  
        rot = rot2.dot(rot1)  
        trans = numpy.array([[0, 0, z0]], float).T  
  
        K = numpy.array([[focal, 0.0, 320.0],  
                        [0.0, focal, 240.0],  
                        [0.0, 0.0, 1.0]], float)  
  
        image = numpy.zeros((480, 640, 3), dtype=numpy.uint8)  
        draw_cube(image, K, cube_orig, 100, rot, trans)  
  
        cv2.imshow('Image', image)  
  
    ...
```

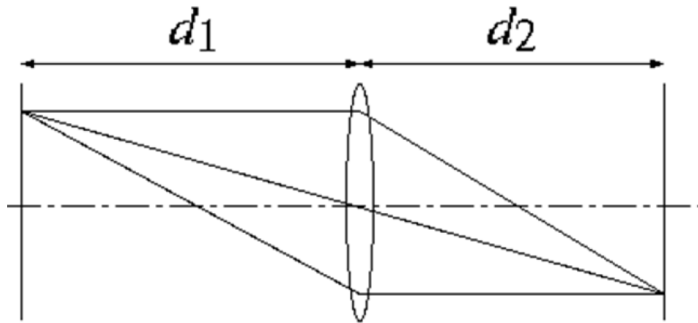
# レンズ

- 空間の1点から発した光線の集合を画像面（イメージセンサ）上の1点に集める働き

理想レンズ

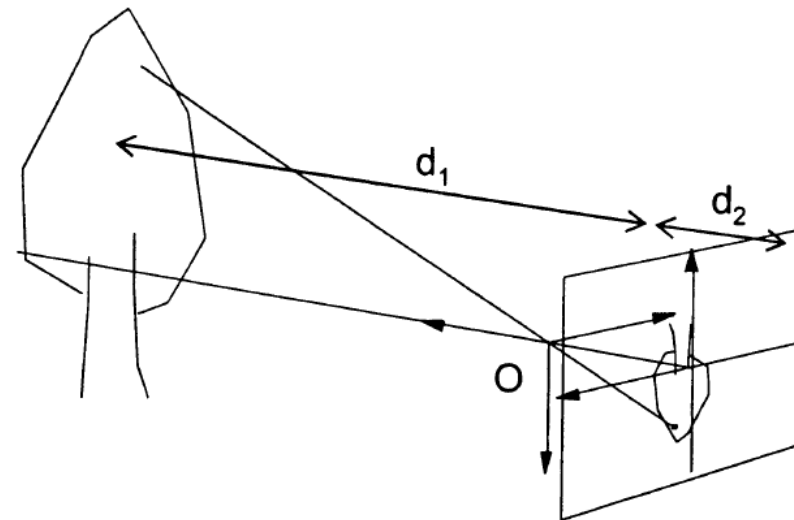


中心投影

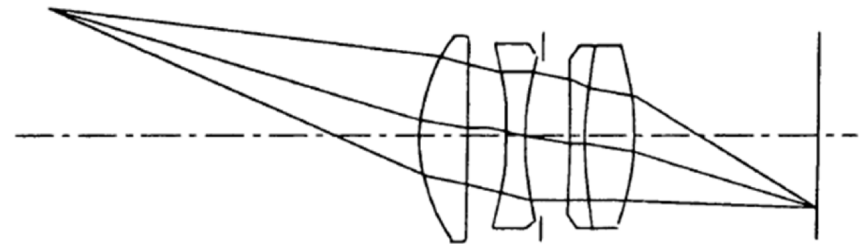


$$\frac{1}{f} = \frac{1}{d_1} + \frac{1}{d_2}$$

Gaussの結像公式



現実のレンズの例



# レンズ歪み

$$\begin{aligned} X' &= X/Z \\ Y' &= Y/Z \end{aligned} \Rightarrow \begin{aligned} X'' &= (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) X' + 2p_1 X' Y' + p_2 (r^2 + 2X'^2) \\ Y'' &= (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) Y' + p_1 (r^2 + 2Y'^2) + 2p_2 X' Y' \end{aligned}$$



$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \begin{bmatrix} f & sf & x_0 \\ 0 & \alpha f & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X'' \\ Y'' \\ 1 \end{bmatrix}$$

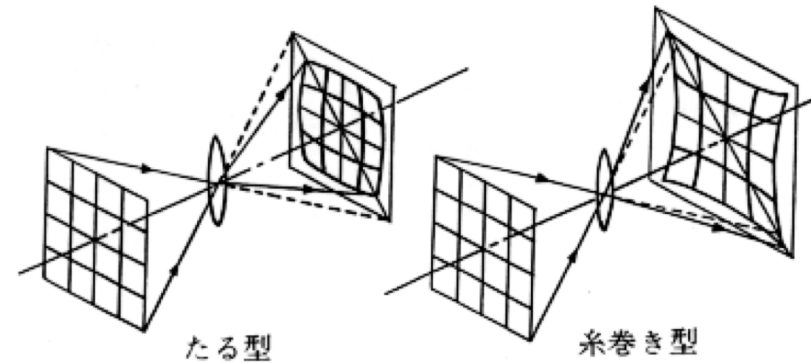


図3.2-1 歪曲

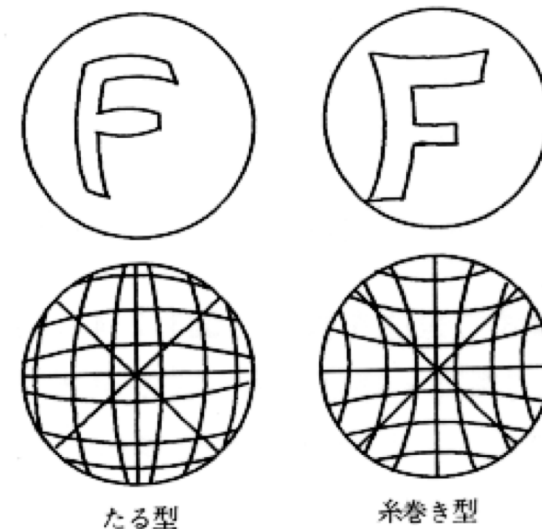
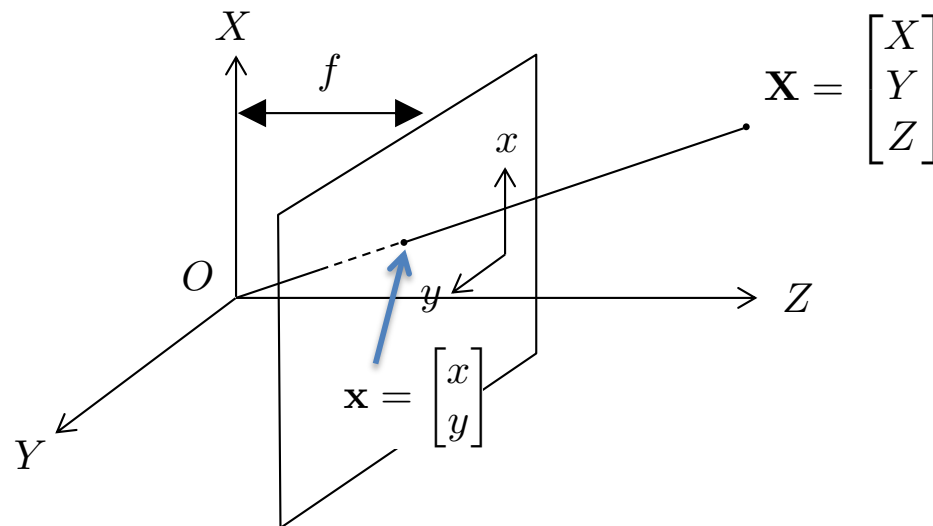


図3.2-2 歪曲があっても放射方向は曲がらない



# 平面像と2次元射影変換

- 平面とその像の関係は2次元射影変換で与えられる

general 3D points

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \mathbf{K}[\mathbf{R} \mid \mathbf{t}] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

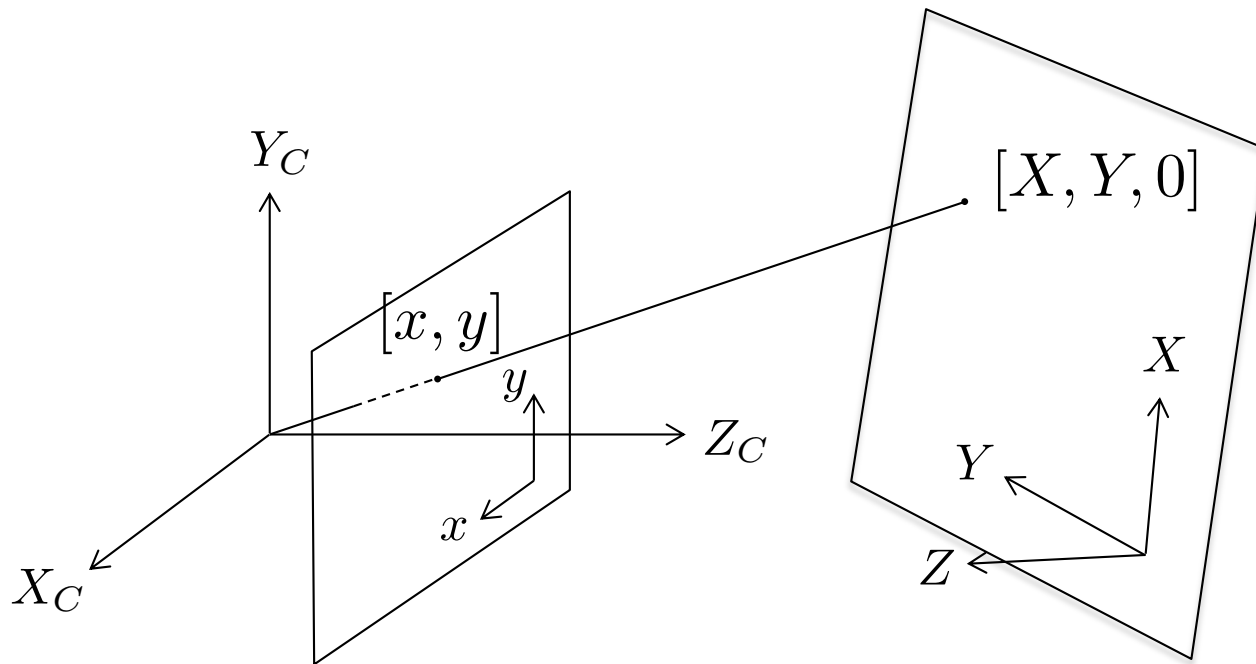
$\Rightarrow$   
 $Z = 0$

planar points

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \mathbf{K}[\mathbf{r}_1 \ \mathbf{r}_2 \ \mathbf{t}] \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$\mathbf{H} = \mathbf{K}[\mathbf{r}_1 \ \mathbf{r}_2 \ \mathbf{t}]$$

3×3 matrix  
2D projective trans.  
(Planar homography)

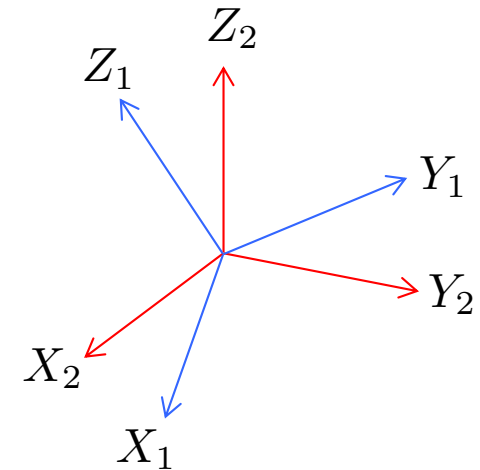


# カメラの回転と2次元射影変換

- カメラがその場回転して得られる像どうし（あるいは無限遠方の像）は2次元射影変換で変換される

$$\begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} = \mathbf{R}_{12} \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix} + \mathbf{t}_{12}$$

~~その場回転~~

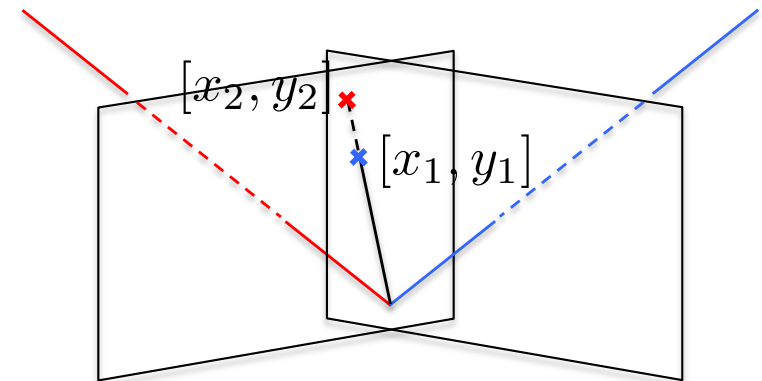


$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \propto \mathbf{K}_2 \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} \quad \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \propto \mathbf{K}_1 \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix}$$

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \propto \mathbf{K}_2 \mathbf{R}_{12} \mathbf{K}_1^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

回転後画像の点

回転前



# Camera calibration (カメラの校正)

```
...
while 1:
    stat, image = cap.read(0)

    ret, centers = cv2.findCirclesGrid(image, (patw, path))
    cv2.drawChessboardCorners(image, (patw, path), centers, ret)

    cv2.imshow('Camera', image)
    key = cv2.waitKey(10)

    if key == 0x1b: # ESC
        break
    elif key == 0x20 and ret == True:
        print 'Saved!'
        objp_list.append(objp.astype(numpy.float32))
        imgp_list.append(centers)

if len(objp_list) >= 3:
    ...
    cv2.calibrateCamera(objp_list, imgp_list, (image.shape[1],
mage.shape[0]), K, dist)
...
```

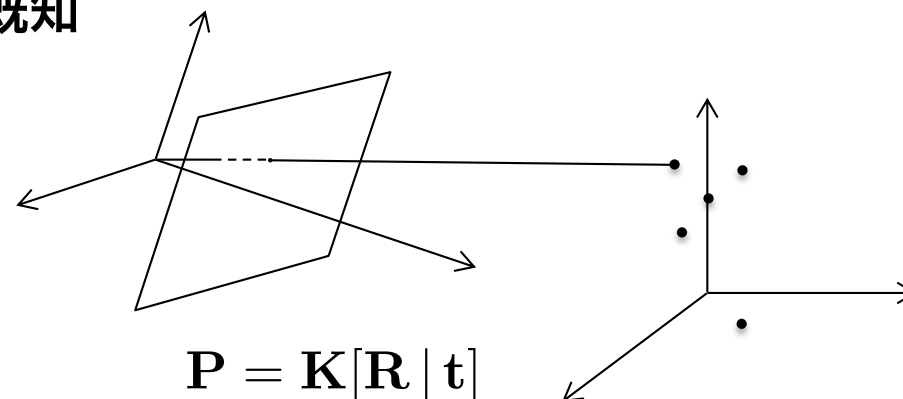
calib.py

"Zhang's calibration method"

# PnP: Perspective-N-Point

- 3次元座標が既知の点群( $\geq 3$ )の像からカメラの姿勢を計算
  - カメラの内部パラメータ ( $K$ ) 既知

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \propto \mathbf{P} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}, \quad (i = 1, \dots, n)$$



```
while 1:
    stat, image = cap.read(0)

    retval, centers = cv2.findCirclesGrid(image, (patw, path))

    if retval == True:
        ret, rvec, tvec = cv2.solvePnP(objp, centers, K, dist)
        Rmat, jac = cv2.Rodrigues(rvec)
        cam_model.draw_cube(image, K, ¥
                               numpy.array([[0.0,0.0,0.0]], float).T,¥
                               3.0, Rmat, tvec)
```

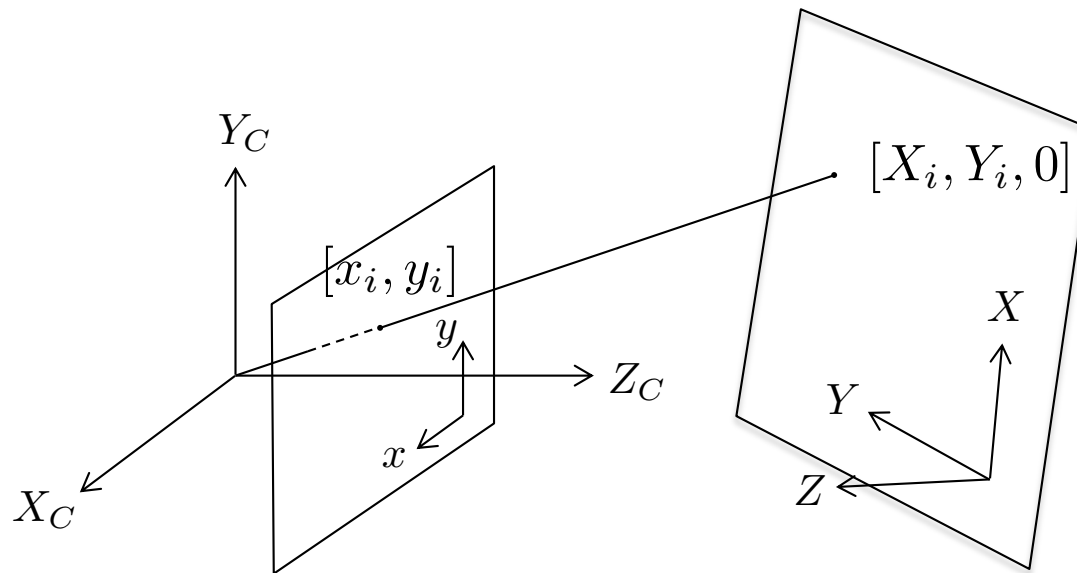
pnp.py

# DLTによる2次元射影変換の計算

※ DLT=Direct Linear Transformation

- 平面上の点  $X_i$  とその像  $x_i$  のペアが  $n$  点与えられたとき、平面と画像間の射影変換を求める
  - $n \geq 4$

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \propto \mathbf{H} \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix} \quad (i = 1, \dots, n)$$



# DLTによる2次元射影変換の計算

$$\begin{aligned}
 x_i &= \frac{h_{00}X_i + h_{01}Y_i + h_{02}}{h_{20}X_i + h_{21}Y_i + h_{22}} \\
 y_i &= \frac{h_{10}X_i + h_{11}Y_i + h_{12}}{h_{20}X_i + h_{21}Y_i + h_{22}}
 \end{aligned}
 \iff
 \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \propto \mathbf{H} \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix}$$

$\Downarrow$

$$h_{00}X_i + h_{01}Y_i + h_{02} - (h_{20}X_i + h_{21}Y_i + h_{22})x_i = 0$$

$$h_{10}X_i + h_{11}Y_i + h_{12} - (h_{20}X_i + h_{21}Y_i + h_{22})y_i = 0$$

or

$$\begin{bmatrix} X_i & Y_i & 1 & 0 & 0 & 0 & -x_iX_i & -x_iY_i & -x_i \\ 0 & 0 & 0 & X_i & Y_i & 1 & -y_iX_i & -y_iY_i & -y_i \end{bmatrix}
 \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \mathbf{0}$$

(Hの成分に関する線形方程式)

# DLTによる2次元射影変換の計算

$$2n \begin{bmatrix} X_1 & Y_1 & 1 & 0 & 0 & 0 & -x_1 X_1 & -x_1 Y_1 & -x_1 \\ 0 & 0 & 0 & X_1 & Y_1 & 1 & -y_1 X_1 & -y_1 Y_1 & -y_1 \\ X_2 & Y_2 & 1 & 0 & 0 & 0 & -x_2 X_2 & -x_2 Y_2 & -x_2 \\ 0 & 0 & 0 & X_2 & Y_2 & 1 & -y_2 X_2 & -y_2 Y_2 & -y_2 \\ & & & & \vdots & & & & \\ X_n & Y_n & 1 & 0 & 0 & 0 & -x_n X_n & -x_n Y_n & -x_n \\ 0 & 0 & 0 & X_n & Y_n & 1 & -y_n X_n & -y_n Y_n & -y_n \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \mathbf{0}$$

$$\mathbf{A}\mathbf{h} = \mathbf{0}$$

### (Hの成分に関する線形方程式)

解（ $n=4$ の場合は厳密解・ $n>4$ の場合は最小二乗解）は  
特異値分解の最小特異値に対応するベクトル： $h = v_8$

$$\mathbf{A} = \mathbf{U}\mathbf{W}\mathbf{V}^\top \quad \mathbf{V} = [\mathbf{v}_0, \dots, \mathbf{v}_8]$$

# DLTによる2次元射影変換の計算

対応点から2次元射影変換を求める部分：

```
def calcHomography(objp, imgp):  
    numPts = objp.shape[0]  
    A = numpy.zeros((numPts*2, 9), float)  
    imgp = imgp.reshape(numPts,2)  
    for i in range(numPts):  
        A[i*2+0,0:2] = objp[i,0:2]  
        A[i*2+0,2] = 1.0  
        A[i*2+0,6:9] = -imgp[i,0]*A[i*2+0,0:3]  
        A[i*2+1,3:5] = objp[i,0:2]  
        A[i*2+1,5] = 1.0  
        A[i*2+1,6:9] = -imgp[i,1]*A[i*2+1,3:6]  
    U, w, Vt = numpy.linalg.svd(A)  
    return Vt[8,:].reshape(3,3)
```

homography.py

$$\mathbf{A} = \mathbf{U}\mathbf{W}\mathbf{V}^\top$$

$$\mathbf{V} = [\mathbf{v}_0, \dots, \mathbf{v}_8]$$

$$\mathbf{h} = \mathbf{v}_8$$

$$\begin{bmatrix} X_1 & Y_1 & 1 & 0 & 0 & 0 & -x_1X_1 & -x_1Y_1 & -x_1 \\ 0 & 0 & 0 & X_1 & Y_1 & 1 & -y_1X_1 & -y_1Y_1 & -y_1 \\ X_2 & Y_2 & 1 & 0 & 0 & 0 & -x_2X_2 & -x_2Y_2 & -x_2 \\ 0 & 0 & 0 & X_2 & Y_2 & 1 & -y_2X_2 & -y_2Y_2 & -y_2 \\ & & & & & \vdots & & & \\ X_n & Y_n & 1 & 0 & 0 & 0 & -x_nX_n & -x_nY_n & -x_n \\ 0 & 0 & 0 & X_n & Y_n & 1 & -y_nX_n & -y_nY_n & -y_n \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \mathbf{0}$$



# DLTによる2次元射影変換の計算

求めた2次元射影変換を分解してカメラ姿勢を求める：

$$\mathbf{H} \propto \mathbf{K} [\mathbf{r}_0 \ \mathbf{r}_1 \ \mathbf{t}] \xRightarrow{K^{-1} \text{をかける}} \mathbf{Q} \equiv \mathbf{K}^{-1} \mathbf{H} \propto [\mathbf{r}_0 \ \mathbf{r}_1 \ \mathbf{t}]$$

$$\Rightarrow \mathbf{Q} / |\mathbf{q}_0| = [\mathbf{r}_0 \ \mathbf{r}_1 \ \mathbf{t}] \quad \mathbf{R} = [\mathbf{r}_0 \ \mathbf{r}_1 \ (\mathbf{r}_0 \times \mathbf{r}_1)]$$

列ベクトルの長さを1に

3列目は1, 2列目と直交

```
homography.py
retval, centers = cv2.findCirclesGrid(image, (patw, path))

if retval == True:
    H = calcHomography(objp, centers) # H = K[r0,r1,t]
    Q = Q*numpy.sign(numpy.linalg.det(Q))
    Q = numpy.linalg.inv(K).dot(H) # Q=K^{-1}H
    R = Q / (numpy.linalg.norm(Q[:,0])) # Normalize
    t = R[:,2].reshape(3,1).copy() # Deep copy
    R[:,2] = numpy.cross(R[:,0],R[:,1]) # r2=r0xr1
    cam_model.draw_cube(image, K,¥
                        numpy.array([[0.0,0.0,0.0]], float).T,¥
                        3.0, R, t)
```

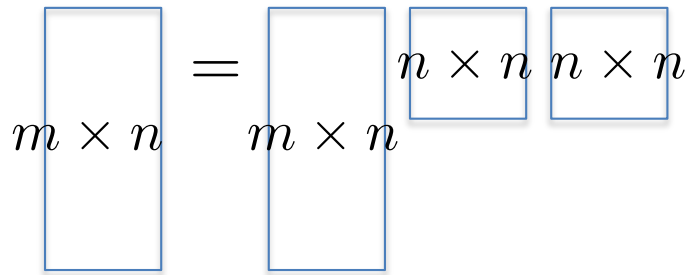
# 特異値分解

(singular value decomposition)

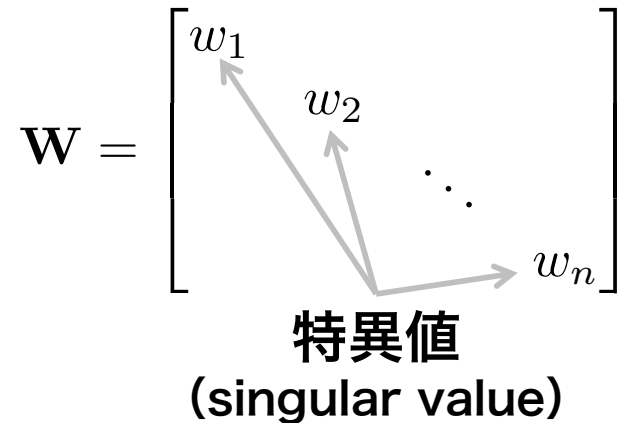
- 任意の  $m \times n$  行列  $A$  は次のように分解可能

$$A = U W V^T$$

直交 対角 直交行列


$$\begin{matrix} m \times n \\ \boxed{\phantom{000000}} \end{matrix} = \begin{matrix} m \times n \\ \boxed{\phantom{000000}} \end{matrix} \begin{matrix} n \times n \\ \boxed{\phantom{000000}} \end{matrix} \begin{matrix} n \times n \\ \boxed{\phantom{000000}} \end{matrix}$$

$$U^T U = I \quad V^T V = I$$


$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

特異値  
(singular value)

※ 特異値を降順にすれば分解は一意

※ 非ゼロ特異値の個数 = 行列のrank

※ 一般化逆行列:  $A^\dagger = V W^{-1} U^T$

$$\begin{aligned} A^\dagger A &= [(A^T A)^{-1} A^T] A \\ &= V W^{-1} U^T U W V^T = I \end{aligned}$$

# 線形最小二乗

## 1) 線形方程式 $Ax = b$ ( $A : m \times n$ )

- $m = n$  で  $A$  が full rank  $\rightarrow$  解あり :  $x = (A^T A)^{-1} A^T b$
- $m > n$  で  $A$  が full rank  $\rightarrow$  厳密解なし・最小二乗解は

$$x = \underset{x}{\operatorname{argmin}} \|Ax - b\|^2 = (A^T A)^{-1} A^T b$$

- $m < n$  の場合一般には解は不定 ( $A$  の列空間内に  $b$  がなければ解なし)

$$\text{※ 一般化逆行列 : } A^\dagger = (A^T A)^{-1} A^T$$

## 2) 線形方程式 $Ax = 0$

- 無意味な解  $x = 0$  は考えない・ $\|x\|^2 = 1$  なる解を考える
- $A$  が full rank でない  $\rightarrow A$  のゼロ特異値に対応するベクトルが解
- $A$  が full rank  $\rightarrow A$  の最小特異値に対応するベクトルが最小二乗解

$$x = \underset{x}{\operatorname{argmin}} \|Ax\|^2$$

# 線形最小二乗

1)  $J(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|^2 \rightarrow \min.$

停留点 :  $\frac{dJ}{d\mathbf{x}} = 2\mathbf{A}^\top (\mathbf{Ax} - \mathbf{b}) = 0 \Rightarrow \mathbf{x} = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{b}$

2)  $J(\mathbf{x}) = \|\mathbf{Ax}\|^2 \rightarrow \min.$  subject to  $\|\mathbf{x}\|^2 = 1$

**Lagrange multiplier:**

$$J(\mathbf{x}) = \|\mathbf{Ax}\|^2 + \lambda(\|\mathbf{x}\|^2 - 1) \rightarrow \min.$$

停留点 :  $\frac{dJ}{d\mathbf{x}} = 2\mathbf{A}^\top \mathbf{Ax} - 2\lambda\mathbf{x} = \mathbf{0} \Rightarrow \mathbf{A}^\top \mathbf{Ax} = \lambda\mathbf{x}$

最小値はこの固有値問題の解 :  $\mathbf{x} = \mathbf{e}_k$  ( $\mathbf{A}^\top \mathbf{A}\mathbf{e}_k = \lambda_k \mathbf{e}_k$ )

最小固有値に対応するベクトルが解 :

$$\|\mathbf{Ax}\|^2 = \mathbf{e}_k^\top \mathbf{A}^\top \mathbf{A}\mathbf{e}_k = \lambda_k \|\mathbf{e}_k\|^2 = \lambda_k$$

注)  $\mathbf{A}^\top \mathbf{A}$ の固有値= $\mathbf{A}$ の特異値の二乗

# レポート 1

- カメラをその場回転させて撮影した2枚の画像は、説明した通り2次元射影変換で関係付けられる（=片方を2次元射影変換するともう一方とぴったり重なる）
  - これはパノラマ画像の生成原理でもある
  - `cv2.warpPerspective()`
- この原理を使って、同様の画像複数枚を使って「超広角レンズで撮影したような画像」を合成せよ
  - 4点以上の画像間対応が必要
    - 対応を手動で求める → 最大A評価
    - 自動で求める → AA
  - 重複部分はblendingすること
    - blending手法の美しさ → 点数に反映
- pythonのコードと実行結果例
  - コードの内容と結果の見せ方 → 点数に

