# Robotic Mapping & Localization

**Kaveh Fathian**
Assistant Professor
Computer Science Department
Colorado School of Mines

## Lec08: Camera Calibration

# Our Simple SLAM System


robotic-mapping / code / visual-odometry / vo_epipolar.cpp

KavehFathian first commit

Code Blame 97 lines (88 loc) · 3.52 KB

```cpp
#include <iostream>
#include <vector>
#include <string>
#include "opencv2/opencv.hpp"

int main()
{
    const char* video_file = "../data/KITTI07/image_0/%06d.png";
    double f = 707.0912;
    cv::Point2d c(601.8873, 183.1104);
    bool use_5pt = true;
    int min_inlier_num = 100;
    double min_inlier_ratio = 0.2;
    const char* traj_file = "vo_epipolar.xyz";
```

# Our Simple SLAM System


```cpp
#include <iostream>
#include <vector>
#include <string>
#include "opencv2/opencv.hpp"

int main()
{
    const char* video_file = "../data/KITTI07/image_0/%06d.png";
    double f = 707.0912;
    cv::Point2d c(601.8873, 183.1104);
    bool use_5pt = true;
    int min_inlier_num = 100;
    double min_inlier_ratio = 0.2;
    const char* traj_file = "vo_epipolar.xyz";

    // Open a video and get an initial image
    cv::VideoCapture video;
    if (!video.open(video_file)) return -1;

    cv::Mat gray_prev;
    video >> gray_prev;
    if (gray_prev.empty())
    {
        video.release();
        return -1;
    }
    if (gray_prev.channels() > 1) cv::cvtColor(gray_prev, gray_prev,
    cv::COLOR_RGB2GRAY);

    // Run the monocular visual odometry
    cv::Mat K = (cv::Mat_<double>(3, 3) << f, 0, c.x, 0, f, c.y, 0, 0, 1);
    cv::Mat camera_pose = cv::Mat::eye(4, 4, CV_64F);
    FILE* camera_traj = fopen(traj_file, "wt");
    if (camera_traj == NULL) return -1;
```

OpenCV library

Camera parameters

SLAM system parameters

Convert RGB images to gray

Camera calibration matrix

Pose (position & orientation) estimates

# Our Simple SLAM System

```cpp
while (true)
{
    // Grab an image from the video
    cv::Mat img, gray;
    video >> img;
    if (img.empty()) break;
    if (img.channels() > 1) cv::cvtColor(img, gray, cv::COLOR_RGB2GRAY);
    else                    gray = img.clone();

    // Extract optical flow
    std::vector<cv::Point2f> pts_prev, pts;
    cv::goodFeaturesToTrack(gray_prev, pts_prev, 2000, 0.01, 10);
    std::vector<uchar> status;
    cv::Mat err;
    cv::calcOpticalFlowPyrLK(gray_prev, gray, pts_prev, pts, status, err);
    gray_prev = gray;

    // Calculate relative pose
    cv::Mat E, inlier_mask;
    if (use_5pt)
    {
        E = cv::findEssentialMat(pts_prev, pts, f, c, cv::RANSAC, 0.999, 1,
        inlier_mask);
    }
    else
    {
        cv::Mat F = cv::findFundamentalMat(pts_prev, pts, cv::FM_RANSAC, 1, 0.
        99, inlier_mask);
        E = K.t() * F * K;
    }
    cv::Mat R, t;
    int inlier_num = cv::recoverPose(E, pts_prev, pts, R, t, f, c, inlier_mask);
    double inlier_ratio = static_cast<double>(inlier_num) / static_cast<double>
    (pts.size());

    // Accumulate relative pose if result is reliable
    cv::Vec3b info_color(0, 255, 0);
    if ((inlier_num > min_inlier_num) && (inlier_ratio > min_inlier_ratio))
    {
        cv::Mat T = cv::Mat::eye(4, 4, R.type());
        T(cv::Rect(0, 0, 3, 3)) = R * 1.0;
        T.col(3).rowRange(0, 3) = t * 1.0;
        camera_pose = camera_pose * T.inv();
        info_color = cv::Vec3b(0, 0, 255);
    }
```


Iterate over images

Feature detection & tracking using optical flow

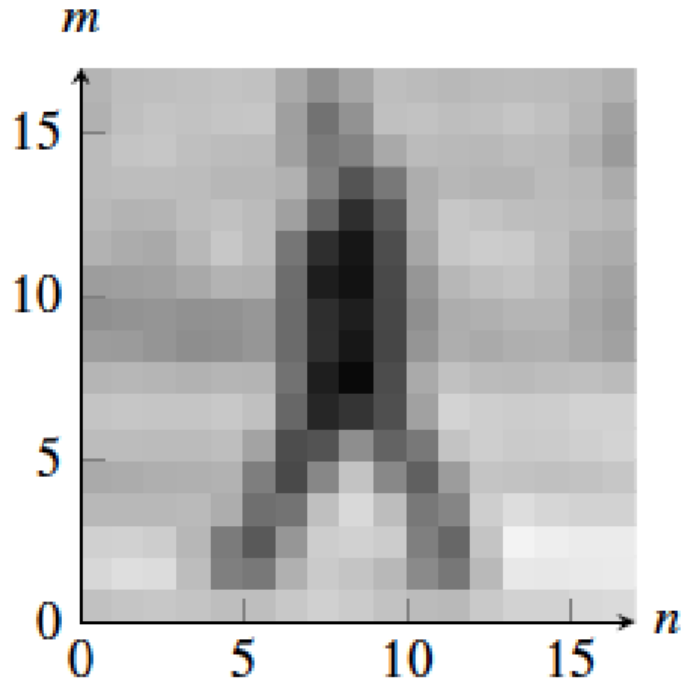
Estimate camera pose (in the form of essential matrix) from consecutive images

RANSAC outlier rejection

Extract camera pose from essential matrix

Camera pose (and trajectory) over time

# Image as a 2D discrete signal

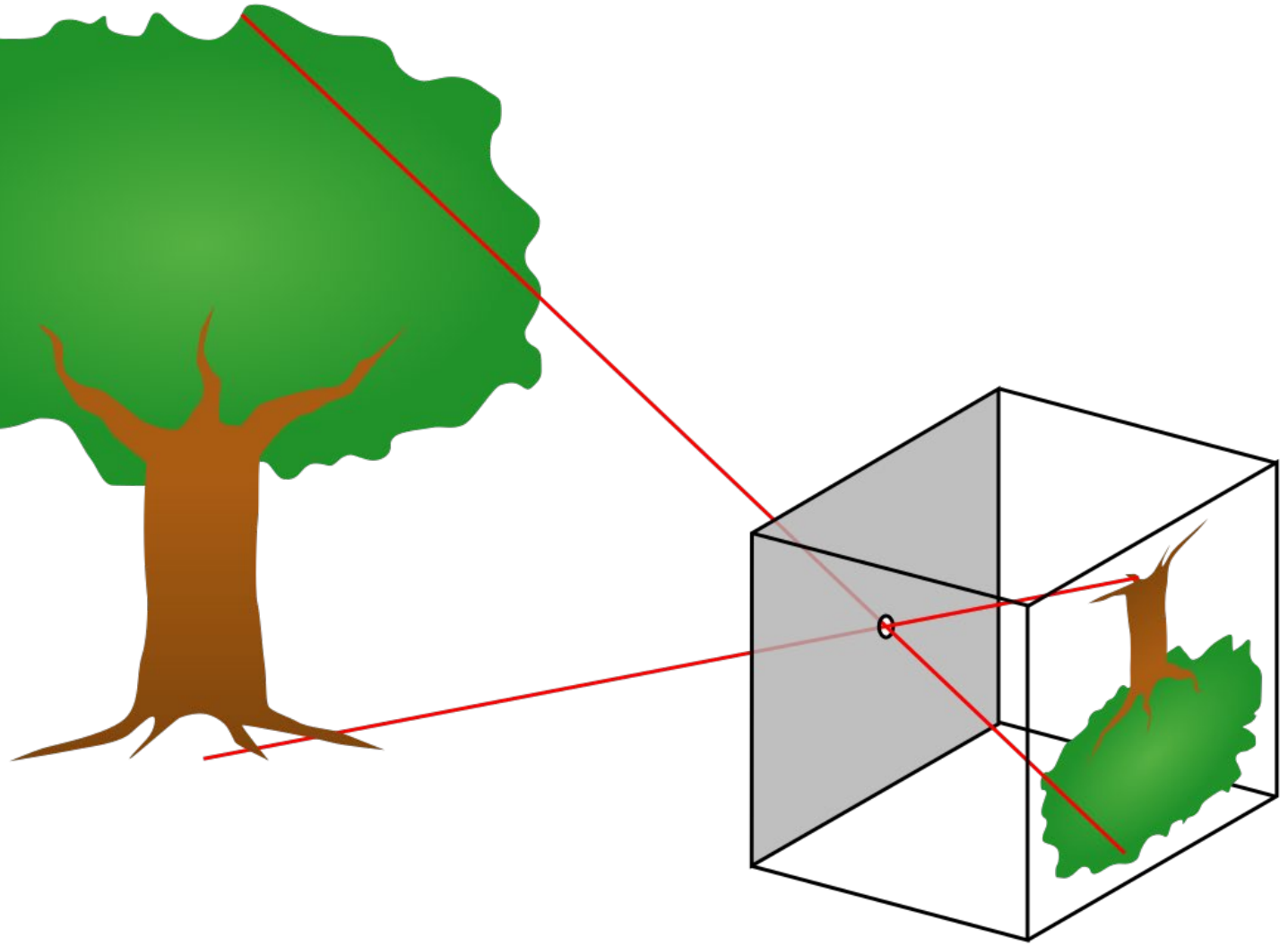




A tiny person of 18 x 18 pixels

$$\mathbf{I} = \begin{bmatrix}
160 & 175 & 171 & 168 & 168 & 172 & 164 & 158 & 167 & 173 & 167 & 163 & 162 & 164 & 160 & 159 & 163 & 162 \\
149 & 164 & 172 & 175 & 178 & 179 & 176 & 118 & 97 & 168 & 175 & 171 & 169 & 175 & 176 & 177 & 165 & 152 \\
161 & 166 & 182 & 171 & 170 & 177 & 175 & 116 & 109 & 169 & 177 & 173 & 168 & 175 & 175 & 159 & 153 & 123 \\
171 & 174 & 177 & 175 & 167 & 161 & 157 & 138 & 103 & 112 & 157 & 164 & 159 & 160 & 165 & 169 & 148 & 144 \\
163 & 163 & 162 & 165 & 167 & 164 & 178 & 167 & 77 & 55 & 134 & 170 & 167 & 162 & 164 & 175 & 168 & 160 \\
173 & 164 & 158 & 165 & 180 & 180 & 150 & 89 & 61 & 34 & 137 & 186 & 186 & 182 & 175 & 165 & 160 & 164 \\
152 & 155 & 146 & 147 & 169 & 180 & 163 & 51 & 24 & 32 & 119 & 163 & 175 & 182 & 181 & 162 & 148 & 153 \\
134 & 135 & 147 & 149 & 150 & 147 & 148 & 62 & 36 & 46 & 114 & 157 & 163 & 167 & 169 & 163 & 146 & 147 \\
135 & 132 & 131 & 125 & 115 & 129 & 132 & 74 & 54 & 41 & 104 & 156 & 152 & 156 & 164 & 156 & 141 & 144 \\
151 & 155 & 151 & 145 & 144 & 149 & 143 & 71 & 31 & 29 & 129 & 164 & 157 & 155 & 159 & 158 & 156 & 148 \\
172 & 174 & 178 & 177 & 177 & 181 & 174 & 54 & 21 & 29 & 136 & 190 & 180 & 179 & 176 & 184 & 187 & 182 \\
177 & 178 & 176 & 173 & 174 & 180 & 150 & 27 & 101 & 94 & 74 & 189 & 188 & 186 & 183 & 186 & 188 & 187 \\
160 & 160 & 163 & 163 & 161 & 167 & 100 & 45 & 169 & 166 & 59 & 136 & 184 & 176 & 175 & 177 & 185 & 186 \\
147 & 150 & 153 & 155 & 160 & 155 & 56 & 111 & 182 & 180 & 104 & 84 & 168 & 172 & 171 & 164 & 168 & 167 \\
184 & 182 & 178 & 175 & 179 & 133 & 86 & 191 & 201 & 204 & 191 & 79 & 172 & 220 & 217 & 205 & 209 & 200 \\
184 & 187 & 192 & 182 & 124 & 32 & 109 & 168 & 171 & 167 & 163 & 51 & 105 & 203 & 209 & 203 & 210 & 205 \\
191 & 198 & 203 & 197 & 175 & 149 & 169 & 189 & 190 & 173 & 160 & 145 & 156 & 202 & 199 & 201 & 205 & 202 \\
153 & 149 & 153 & 155 & 173 & 182 & 179 & 177 & 182 & 177 & 182 & 185 & 179 & 177 & 167 & 176 & 182 & 180
\end{bmatrix}$$

# Pinhole imaging


real-world object

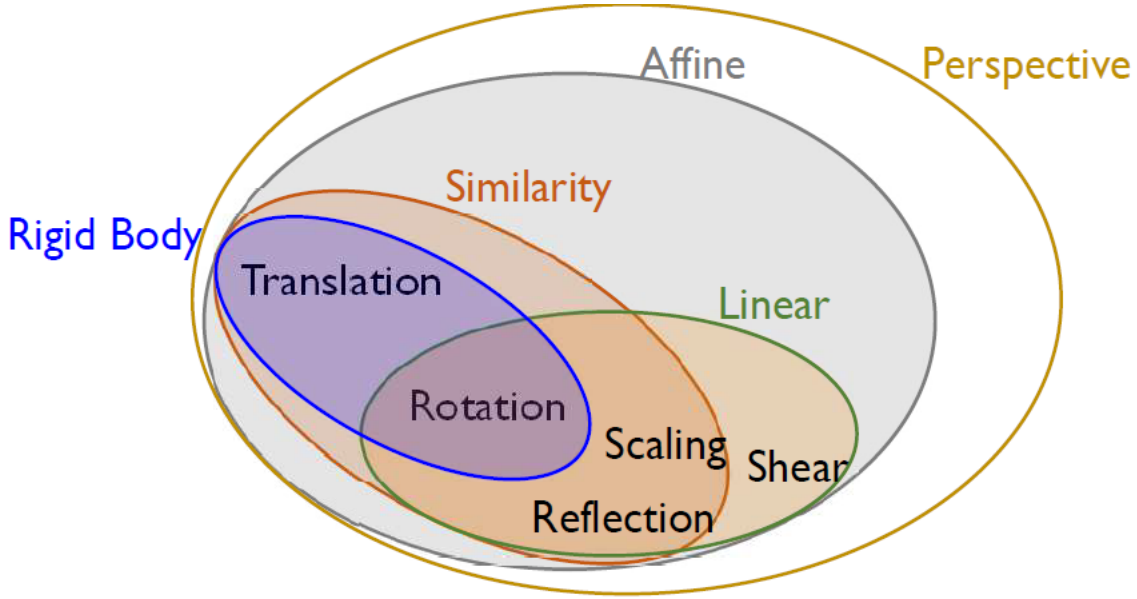


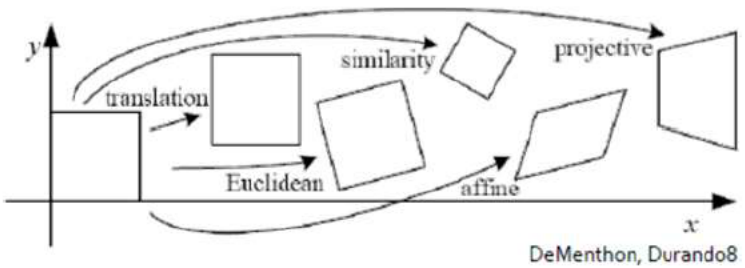
copy of real-world object (inverted & scaled)

# 2D Transformations: Properties Preserved

Transformations can be classified based on the properties they preserve:


- Rigid Body
  - angles, lengths, areas
- Similarity
  - angles, length ratios
- Linear
  - linear combination
- Affine
  - parallel lines, length ratios, area ratios
- Perspective
  - collinearity, cross-ratio
- …




Sugih Jamin

# Cameras are Projective Transformation Machines

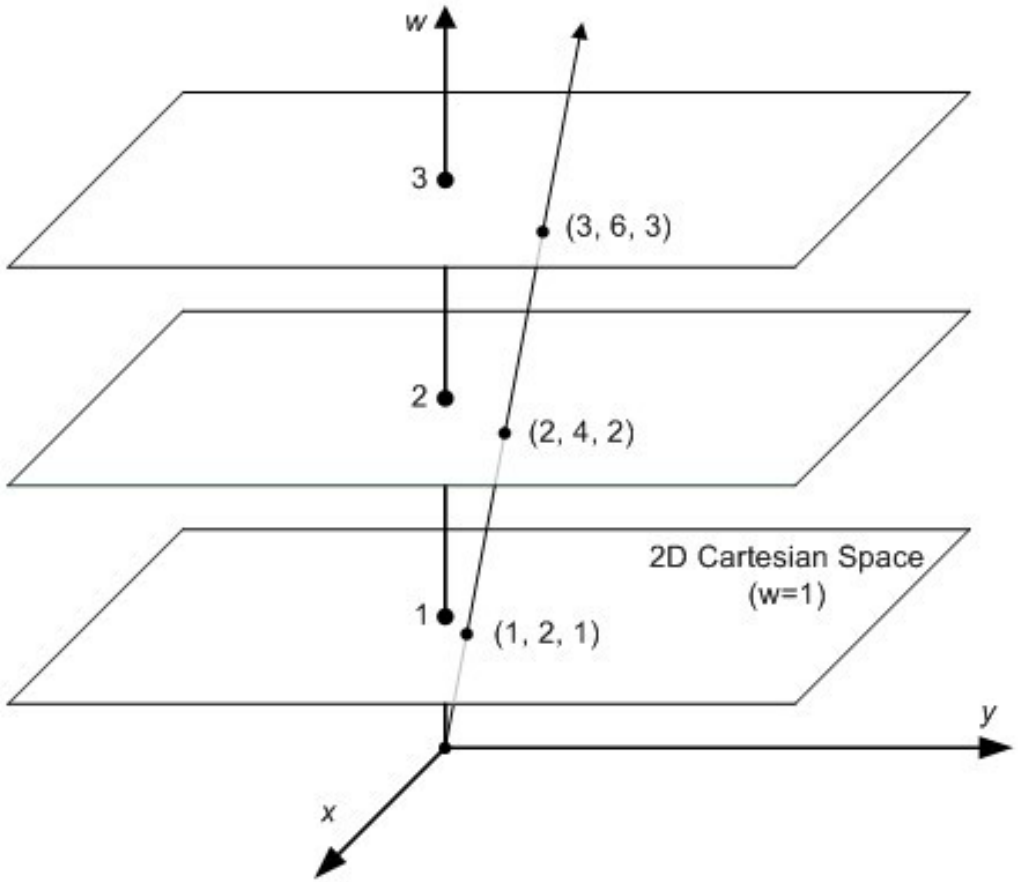

Projective Transformation

3D World

2D Image


# Homogeneous Coordinates

- Projective
- Point becomes a line




Song Ho Ahn

To homogeneous

$$(x, y) \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

From homogeneous

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} \Rightarrow (x/w, y/w)$$

# The camera as a coordinate transformation


homogeneous coordinates

$$\mathbf{x} = \mathbf{P}\mathbf{X}$$

2D image point

camera matrix

3D world point

A camera is a mapping from the 3D world to a 2D image

# The camera as a coordinate transformation

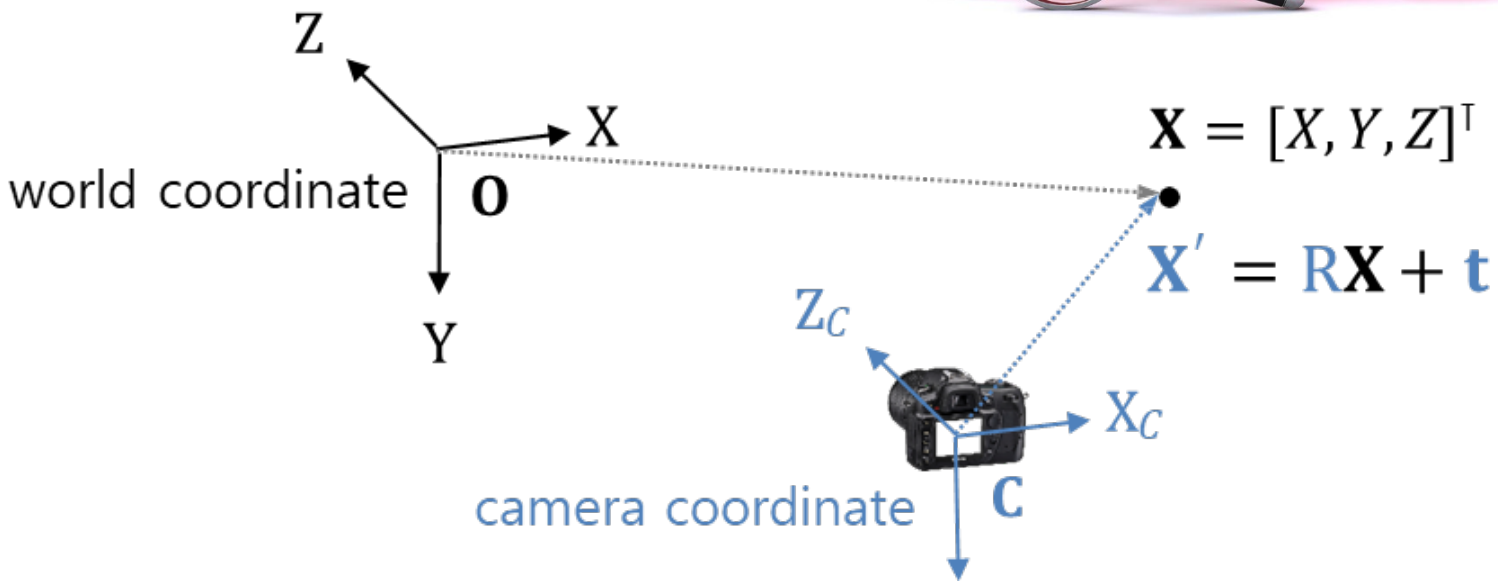

$$\mathbf{x} = \mathbf{P}\mathbf{X}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

| homogeneous image coordinates | camera matrix | homogeneous world coordinates |
|---|---|---|
| 3 x 1 | 3 x 4 | 4 x 1 |

# General pinhole camera matrix


$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$$




$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{array}\right]$$

intrinsic parameters

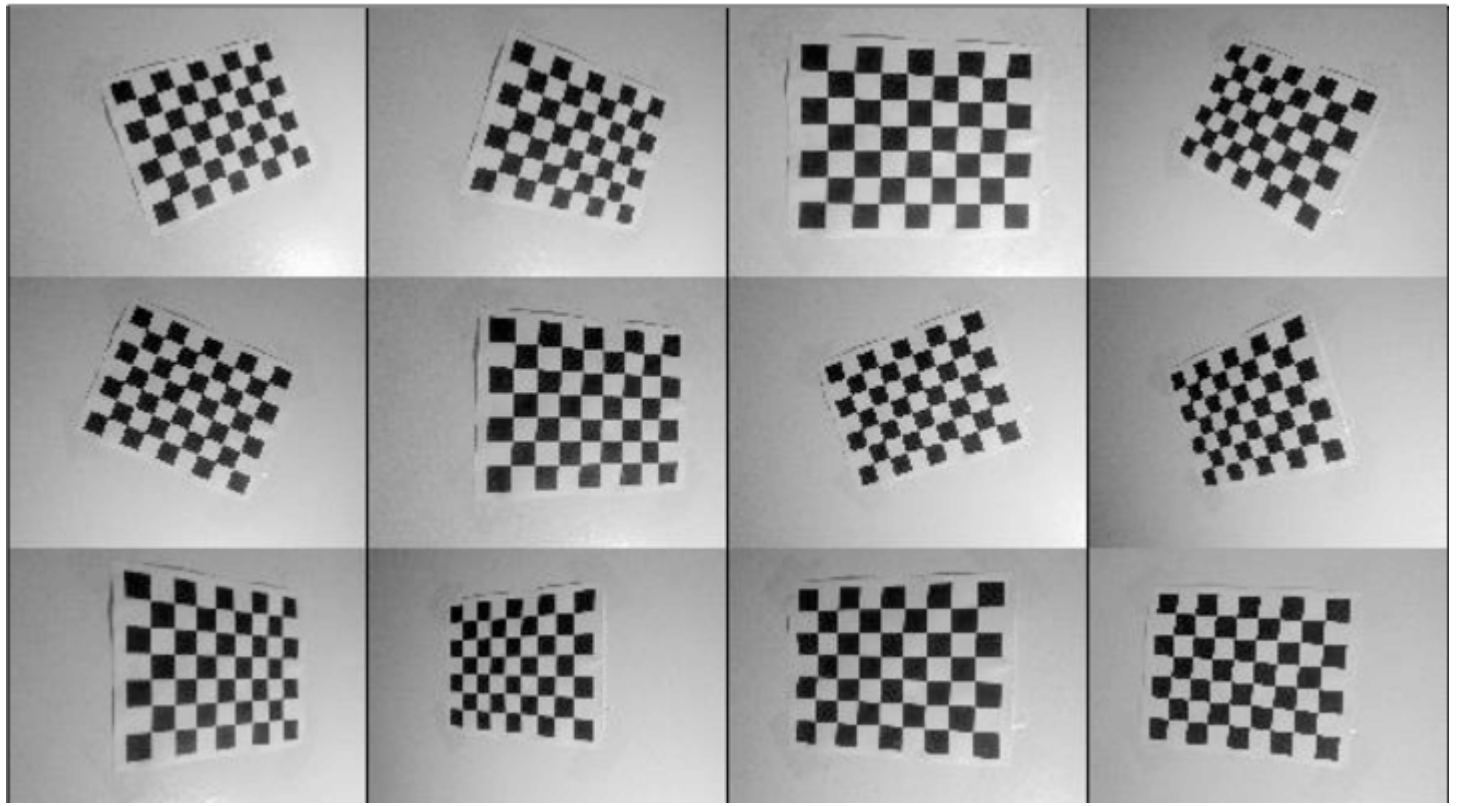
extrinsic parameters

$$\mathbf{R} = \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix} \quad \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

3D rotation

3D translation

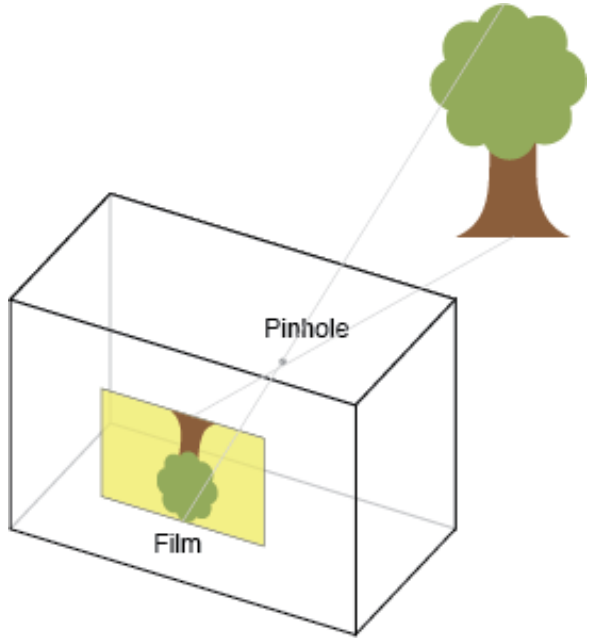
# Geometric Camera Calibration



# Camera Calibration

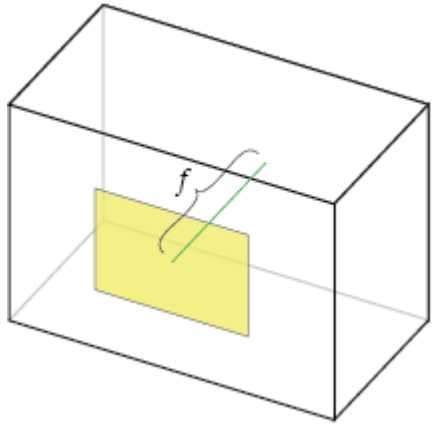
$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{bmatrix}$$

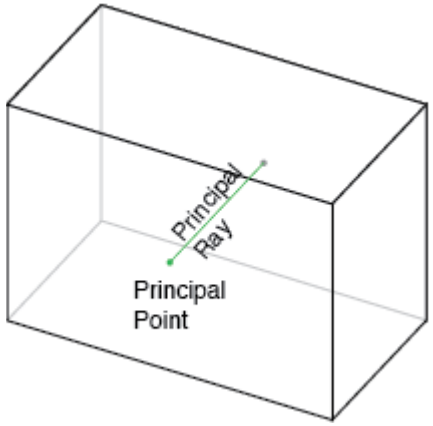
intrinsic parameters  extrinsic parameters

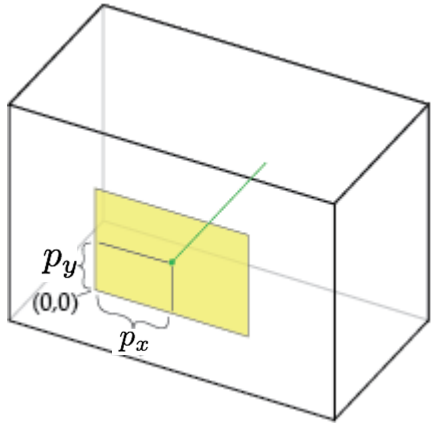


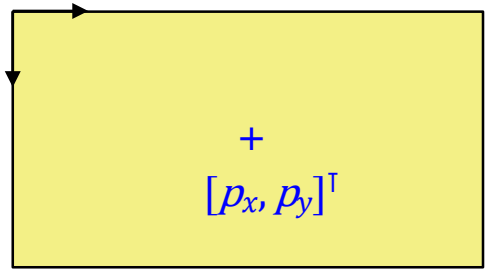

Camera calibration estimates the intrinsic parameters:

- $f$ is the focal length of the camera (e.g., in millimeters or pixels)
- $p_x$ and $p_y$ are locations of the principal/optical point (in millimeters or pixels)



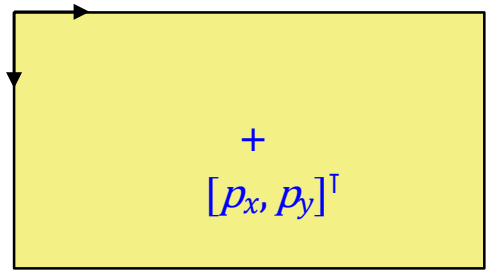




**NOTE**: You can convert pixel units to world units (e.g. mm) if you know sensor dimensions in world units. For example, if you know the image sensor has a width $W$ in millimeters, and the image width in pixels is $w$, you can convert the focal length $f$ to world units $F$ as $F = f\frac{W}{w}$

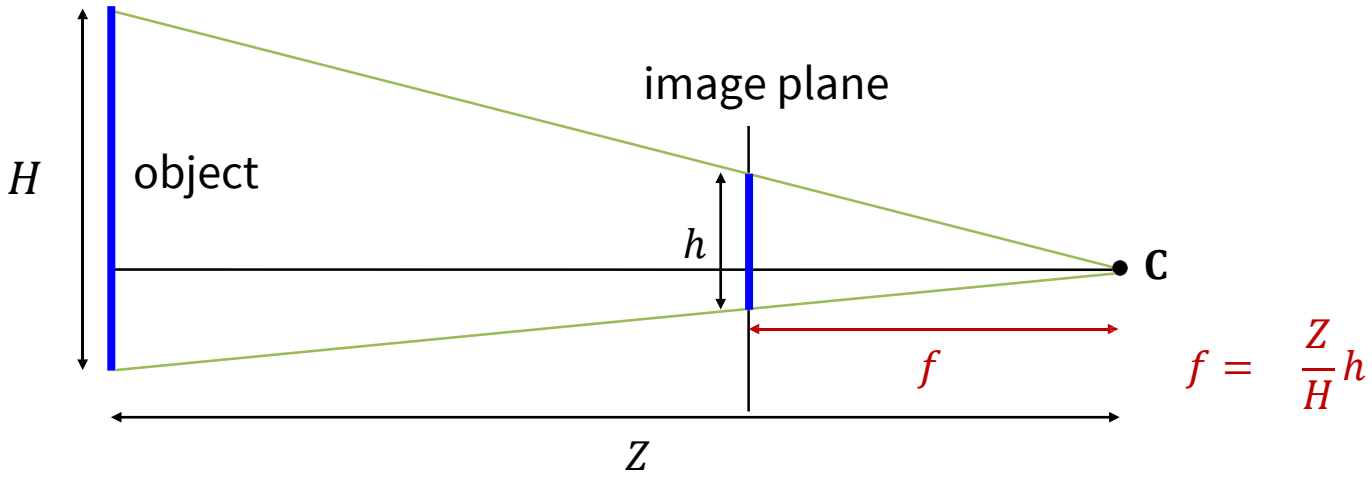
# Simple Camera Calibration

$$\begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Example: Simple camera calibration
  - Unknowns:
    - Focal length ($f$) of the camera (unit: [pixel])
    - Principal point $(p_x, p_y)$ of the camera (unit: [pixel])
  - Given: The observed object height ($h$) on the image plane (unit: [pixel])
  - Assumptions
    - The object height ($H$) and distance ($Z$) from the camera are known.
    - The object is aligned with the image plane.

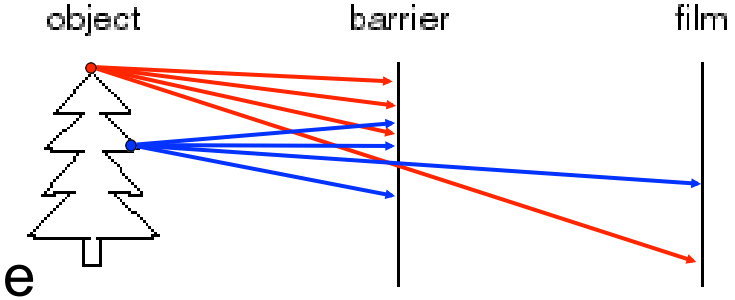



Principal point$(p_x, p_y)$ can be taken as the image center




$$f = \frac{Z}{H} h$$

# Pinhole Camera Model




Assumptions made in the pinhole camera model

- All rays from the object point intersect in a single point
- All image points lie on a plane
- The ray from the object point to the image point is a straight line

**These assumption often do not hold for real cameras/lenses**

**➔ Calibration fixes image distortions, resulting in a good pinhole model**

telephoto

normal

wide-angle

fisheye

# What does it mean to “calibrate a camera”?

Many different ways to calibrate a camera:

- Radiometric calibration
- Color calibration
- Geometric calibration
- Noise calibration
- Lens (or aberration) calibration

before

after

# Geometric Distortion Models

**Q: How to represent geometric distortions?**

- **Geometric distortion models**
  - A camera lens generates geometric distortion, which can be approximated (modeled) as a nonlinear function $f_d$.
  - Geometric distortion models $f_d$ are mostly defined on the normalized image plane.
  - Camera projection with geometric distortion: $\mathbf{x} = \text{proj}(\mathbf{X}; \mathrm{K}, \mathrm{R}, \mathbf{t}, d)$ where $d$ is a set of distortion coefficients.

Note: $\mathbf{x} = \mathrm{K}(\mathrm{R}\mathbf{X} + \mathbf{t})$ without distortion and normalization

$$\mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \longrightarrow \mathbf{X}' = \mathrm{R}\mathbf{X} + \mathbf{t} \longrightarrow \hat{\mathbf{x}} = \begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} = \begin{bmatrix} X'/Z' \\ Y'/Z' \end{bmatrix} \longrightarrow \hat{\mathbf{x}}_d = f_d(\hat{\mathbf{x}}) \longrightarrow \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f_x \hat{x}_d + c_x \\ f_y \hat{y}_d + c_y \end{bmatrix}$$

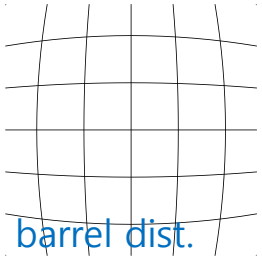
3D point (the world coordinate) | 3D point (the camera coordinate) | (the normalized image plane) | **geometric distortion** | pinhole camera projection

# Geometric Distortion Models

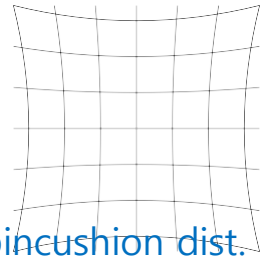
- **Geometric distortion models**
  - **Polynomial distortion model** (a.k.a. Brown-Conrady model; 1919)

$$\begin{bmatrix}\hat{x}_d \\ \hat{y}_d\end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + \cdots)\begin{bmatrix}\hat{x} \\ \hat{y}\end{bmatrix} + (1 + p_3 r^2 + p_4 r^4 + \cdots)\begin{bmatrix}2p_1\hat{x}\hat{y} + p_2(r^2 + 2\hat{x}^2) \\ 2p_2\hat{x}\hat{y} + p_1(r^2 + 2y^2)\end{bmatrix} \quad \text{where} \quad r^2 = \hat{x}^2 + \hat{y}^2$$

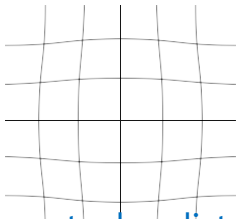
    - OpenCV: `cv.projectPoints()` ↔ `cv.undistortPoints()`



barrel dist.

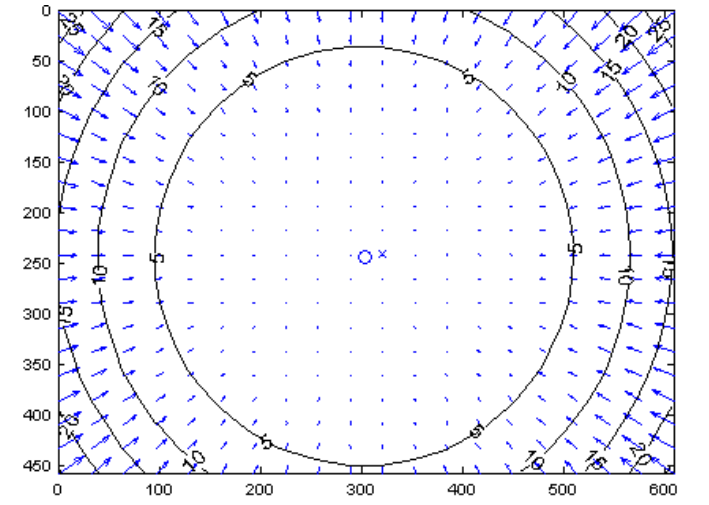


pincushion dist.



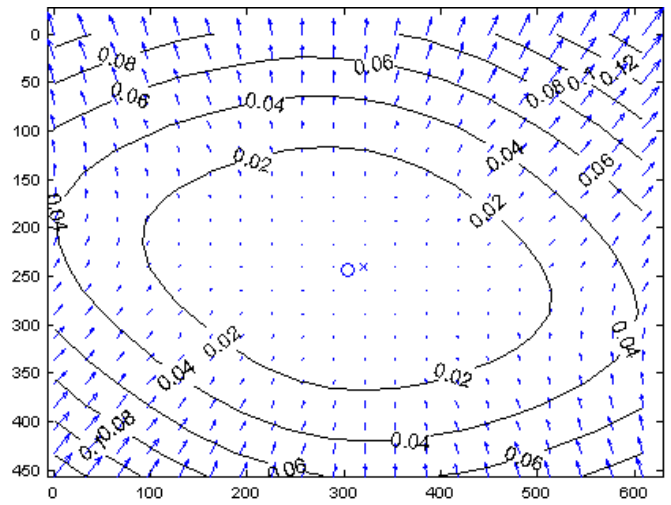
mustache dist.

**Radial distortion**

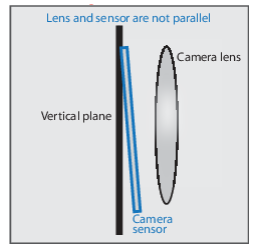


Pixel error = [0.1174, 0.1159]
Focal Length = (657.303, 657.744) +/- [0.2849, 0.2894]
Principal Point = (302.717, 242.334) +/- [0.5912, 0.5571]
Skew = 0.0004198 +/- 0.0001905
Radial coefficients = (-0.2535, 0.1187, 0) +/- [0.00231, 0.009418, 0]
Tangential coefficients = (-0.0002789, 5.174e-005) +/- [0.0001217, 0.0001208]

**Tangential distortion**



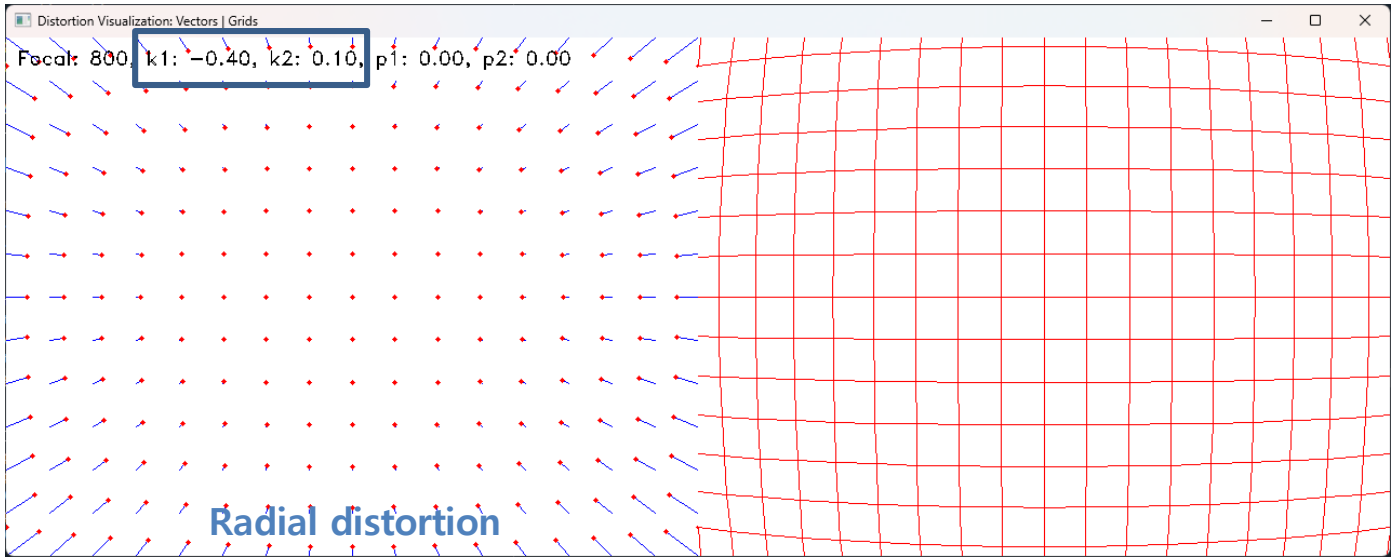
Pixel error = [0.1174, 0.1159]
Focal Length = (657.303, 657.744) +/- [0.2849, 0.2894]
Principal Point = (302.717, 242.334) +/- [0.5912, 0.5571]
Skew = 0.0004198 +/- 0.0001905
Radial coefficients = (-0.2535, 0.1187, 0) +/- [0.00231, 0.009418, 0]
Tangential coefficients = (-0.0002789, 5.174e-005) +/- [0.0001217, 0.0001208]

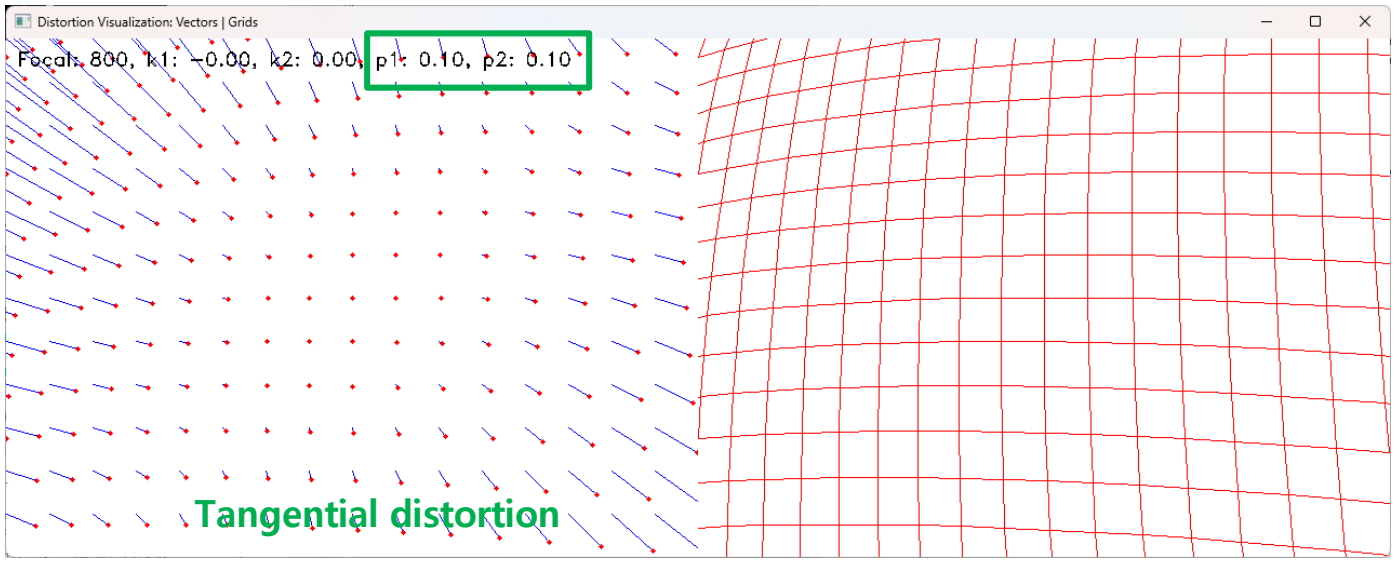



Why?
Lens and sensor are not parallel.
(usually negligible)

# Geometric Distortion Models

- Example: **Geometric distortion visualization**



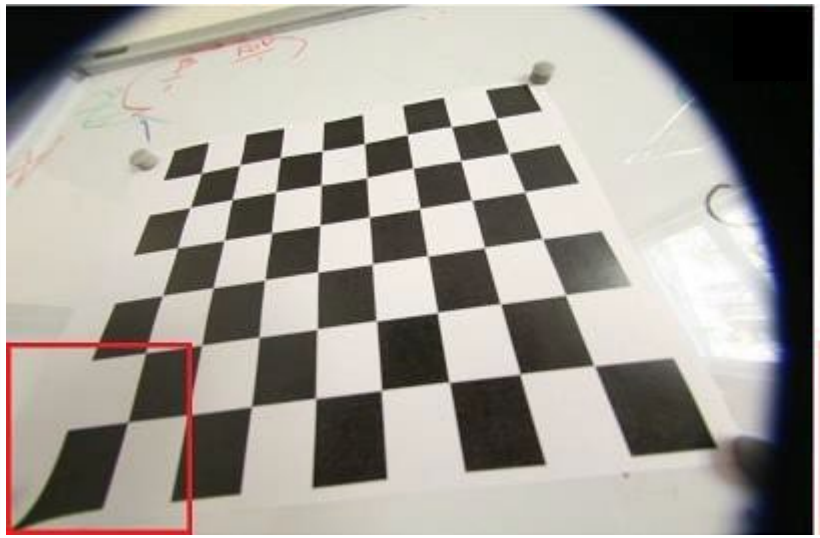
# Camera Projection Model

- **Geometric distortion models**
  - **Fisheye lens model** (a.k.a. Kannala-Brandt model; T-PAMI 2006)

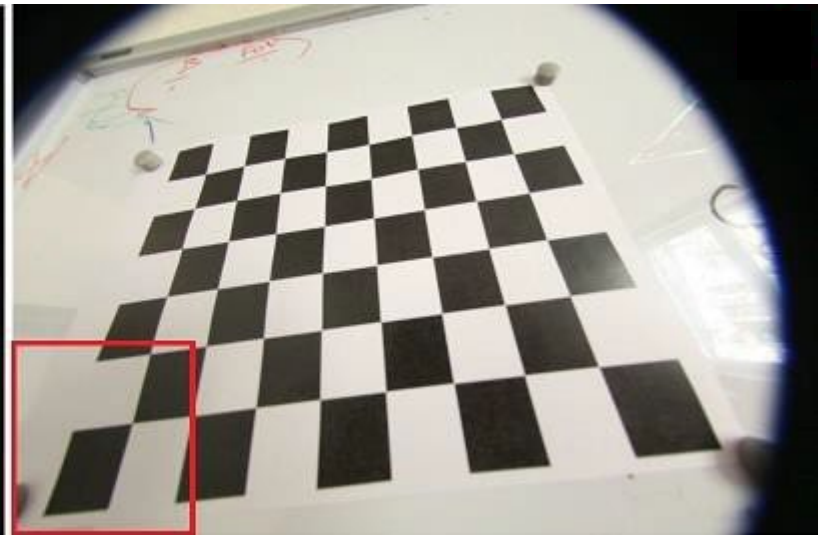
$$\begin{bmatrix}\hat{x}_d \\ \hat{y}_d\end{bmatrix} = (1 + k_1\theta^2 + k_2\theta^4 + \cdots)\frac{\theta}{r}\begin{bmatrix}\hat{x} \\ \hat{y}\end{bmatrix} \quad \text{where} \quad r^2 = \hat{x}^2 + \hat{y}^2 \quad \text{and} \quad \theta = \tan^{-1} r$$

  - The fisheye lens model can describe strong barrel distortion especially around image boundary.
  - OpenCV: `cv.fisheye.projectPoints()` ↔ `cv.fisheye.undistortPoints()`

Original fisheye image



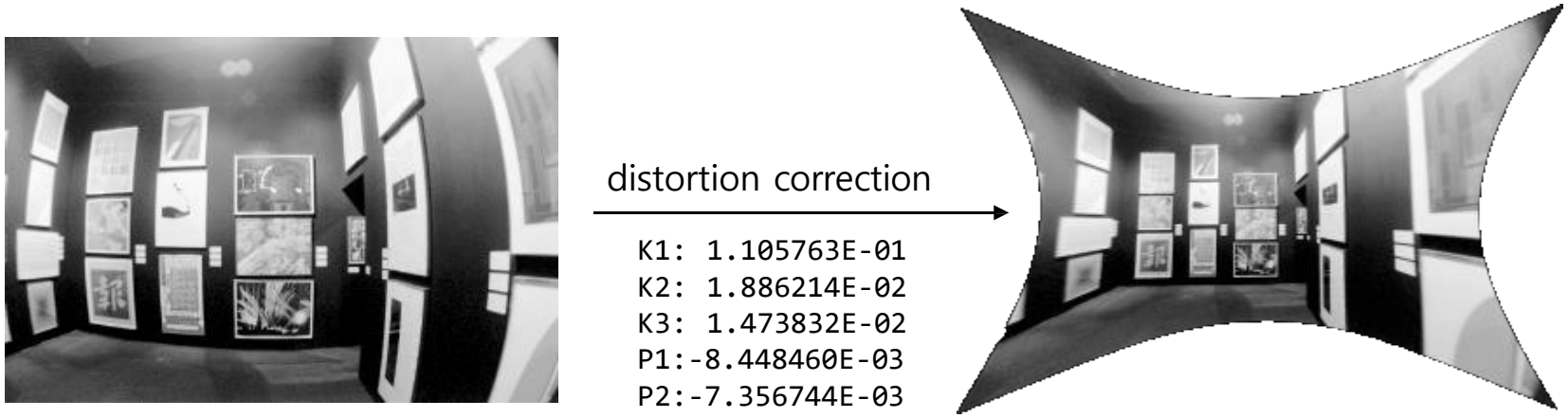
Polynomial distortion model



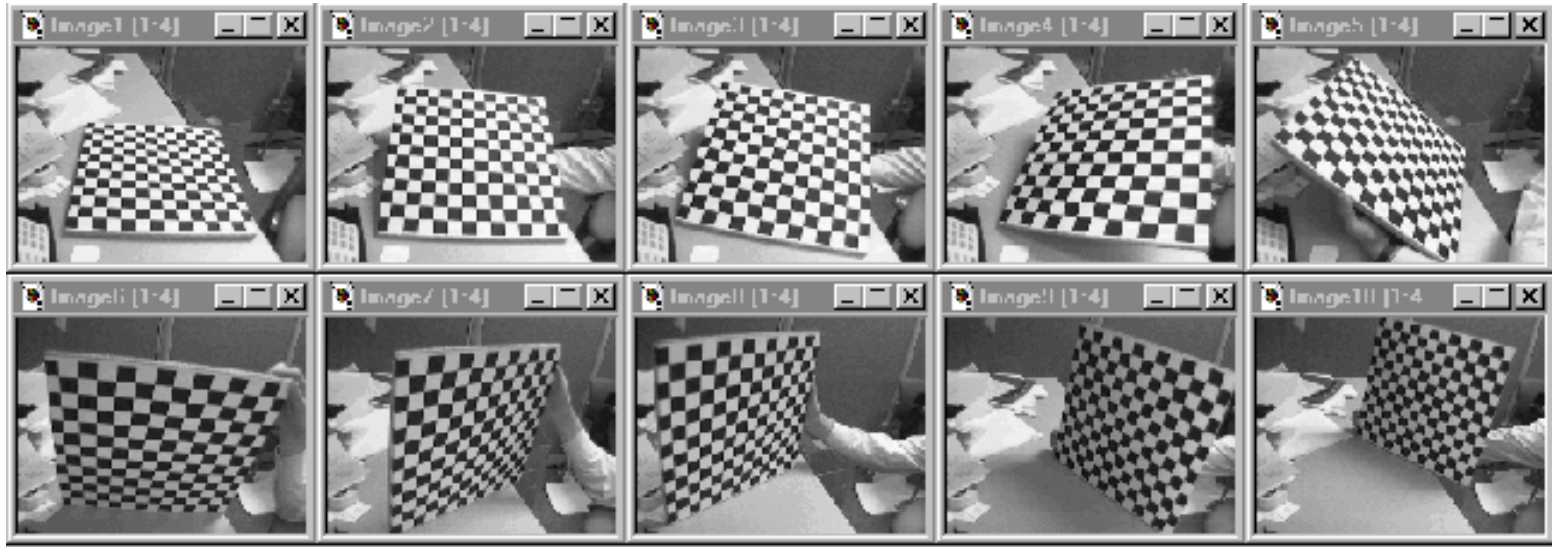
Fisheye lens model

# Geometric Distortion Models

- **Geometric distortion correction**
  - Input: The original image
  - Output: Its rectified image (without geometric distortion)
  - Given: Its camera matrix and distortion coefficient
  - Solutions for the polynomial distortion model
    - OpenCV `cv.undistort()` and `cv.undistortPoints()` (Note: included in `imgproc` module)
      ↔ `cv.projectPoints()` (Note: included in `calib3d` module)



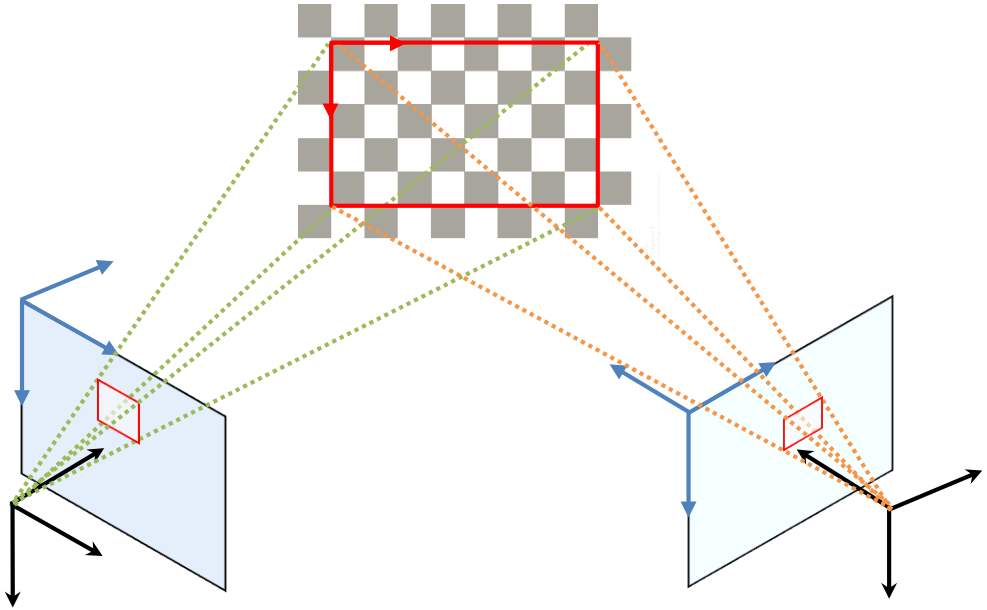
# Multi-Plane Calibration



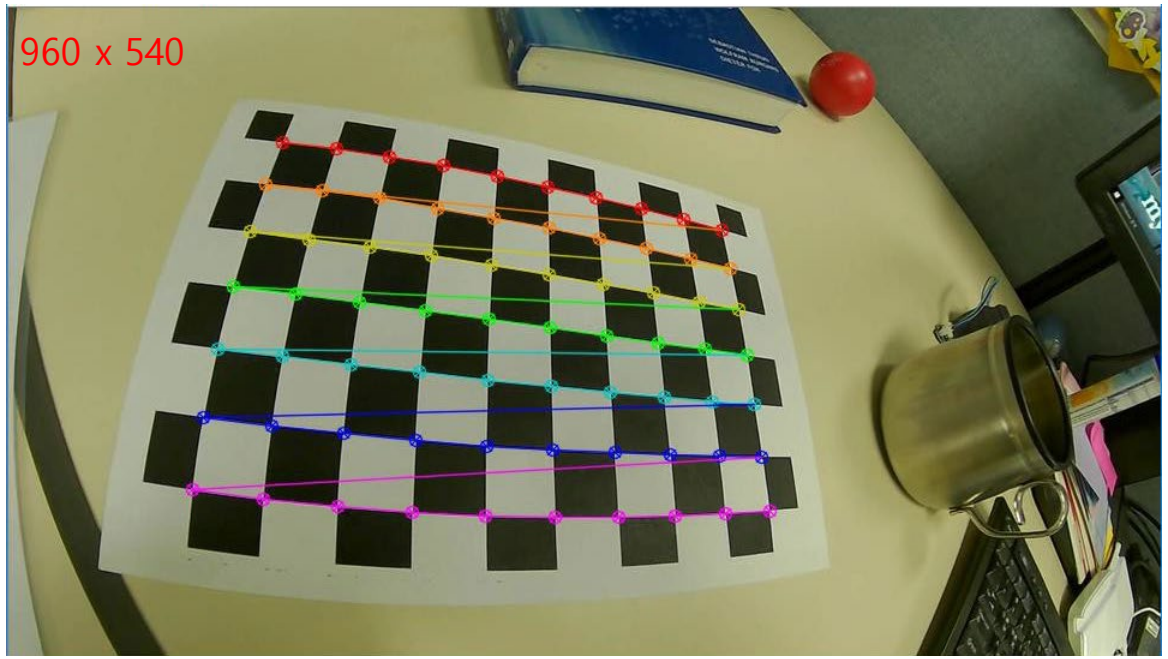
- Only requires a planar checkerboard
- Don’t have to know positions/orientations
- Great code available online!
  - MATLAB: http://www.vision.caltech.edu/bouguetj/calib_doc/index.html
  - OpenCV: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

# Camera Calibration

- **Camera calibration**
  - Unknown: Intrinsic + $m$ x extrinsic parameters (5* + $m$ x 6 DOF)
    - Note: The number of intrinsic parameters* can be varied according to user configuration
  - Solutions [Tools]
    - OpenCV: `cv.calibrateCamera()` and `cv.initCameraMatrix2D()`
    - Camera Calibration Toolbox for MATLAB, Jean-Yves Bouguet
    - Camera Calibration using OpenCV
  - How to get calibration boards:
    - Print out the pattern and stick it on a hard board
    - Calibration Checkerboard Collection, Mark Jones
    - Pattern Generator, calib.io



# Camera Calibration Example

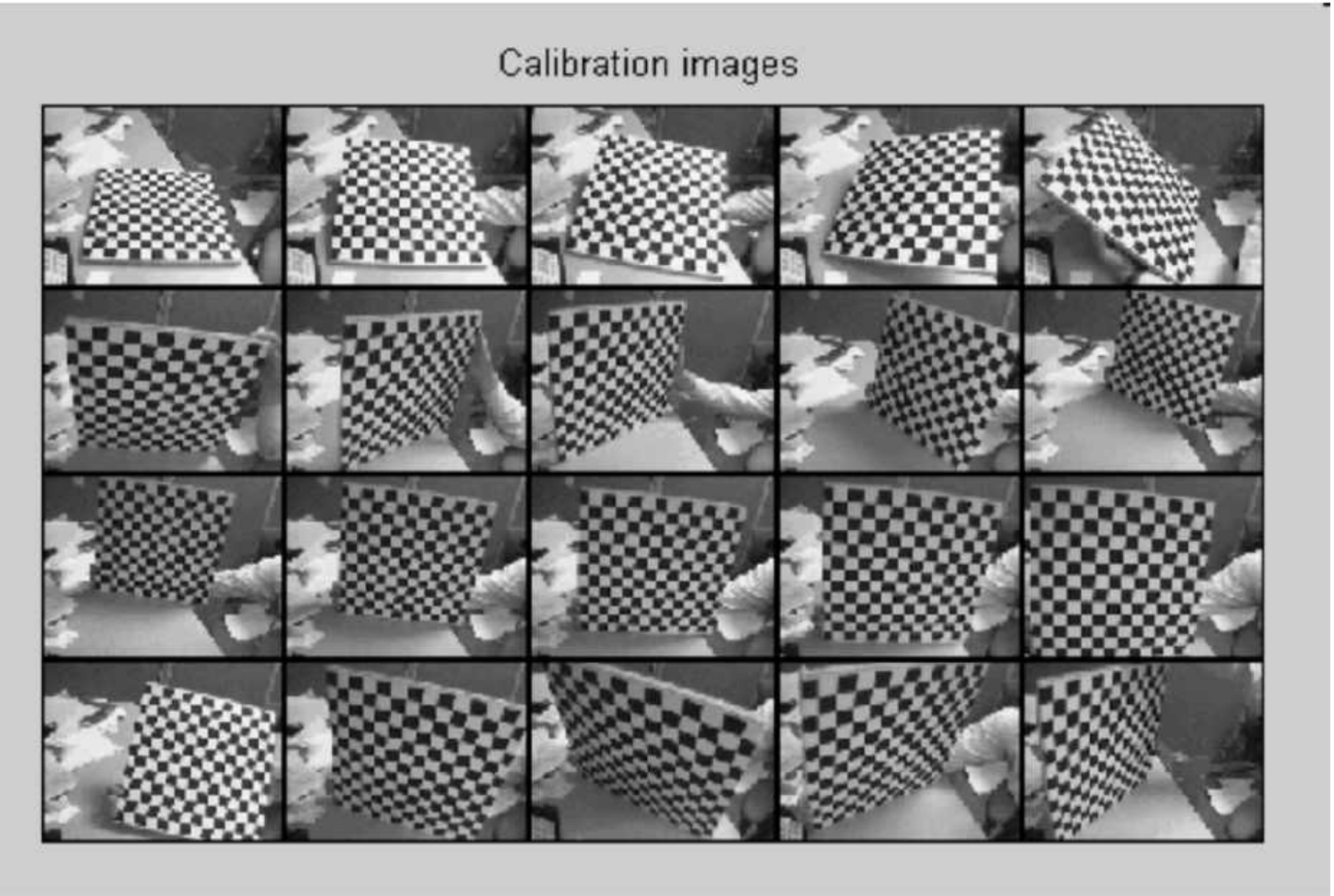



```
## Camera Calibration Results
* The number of applied images = 22
* RMS error = 0.473353
* Camera matrix (K) =
  [432.7390364738057, 0, 476.0614994349778]
  [0, 431.2395555913084, 288.7602152621297]
  [0, 0, 1]
* Distortion coefficient (k1, k2, p1, p2, k3, ...) =
  [-0.2852754904152874, 0.1016466459919075, -0.0004420196146339175, 0.0001149909868437517, -0.01803978785585194]
```

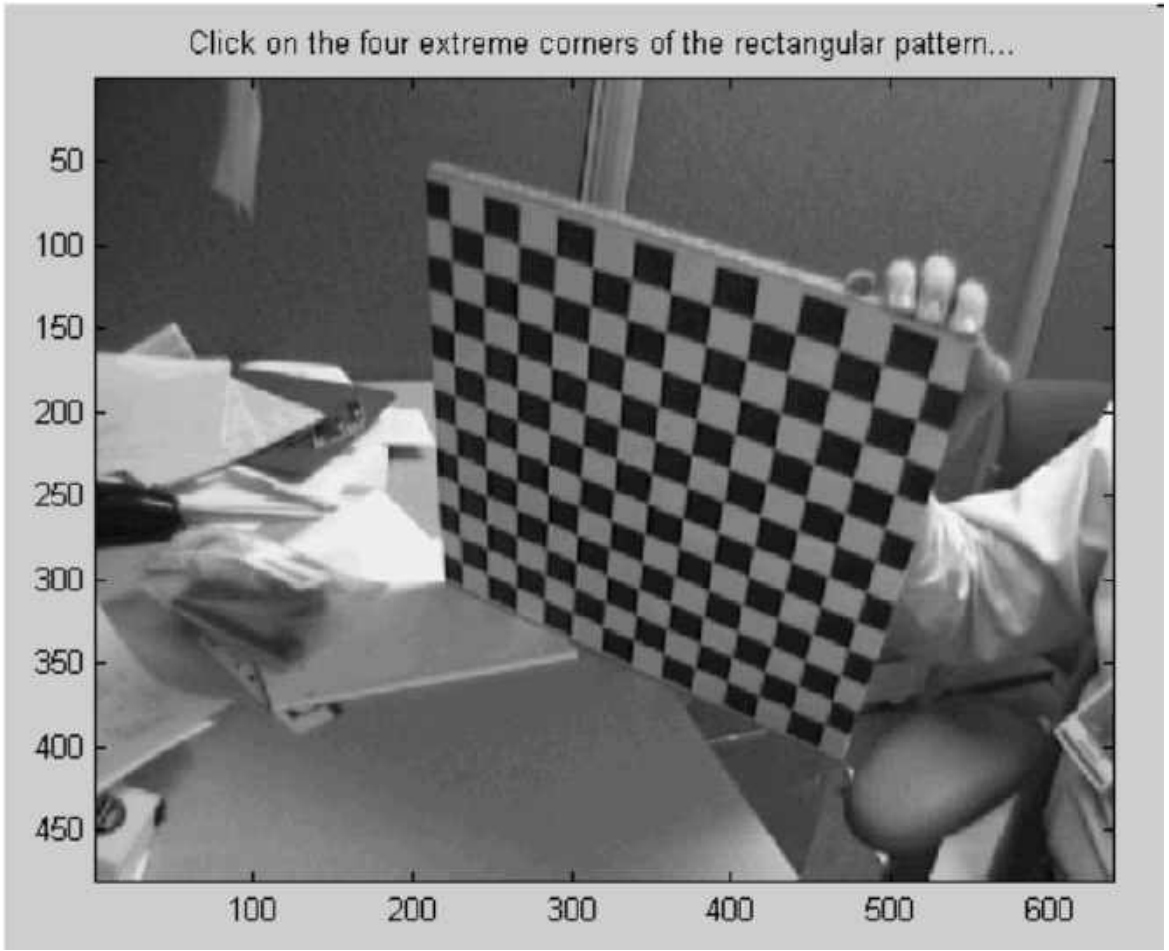
Note) Close to the center of the image, (480, 270)

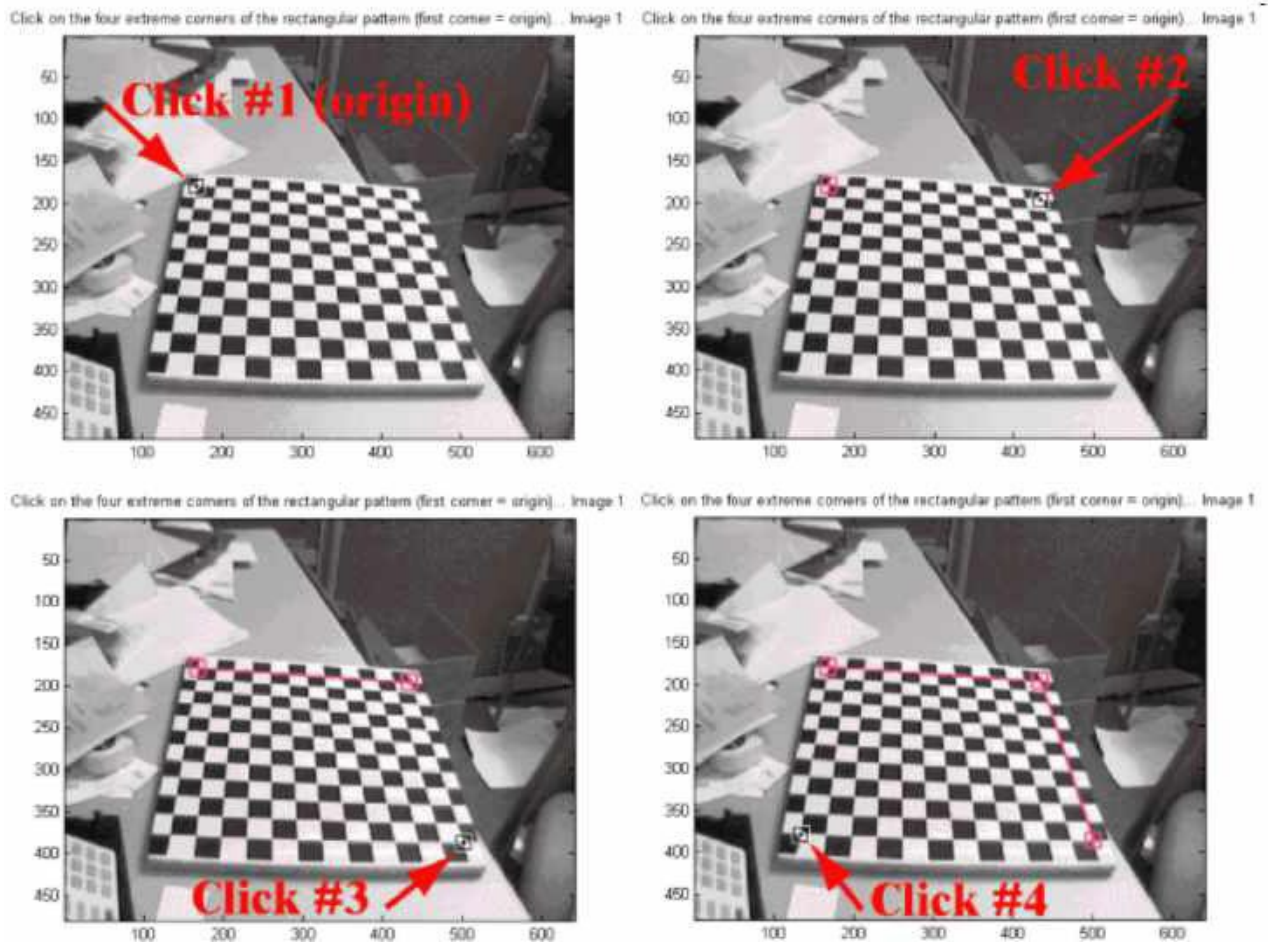
Note: Close to zero (usually negligible)

# Step-by-step demonstration

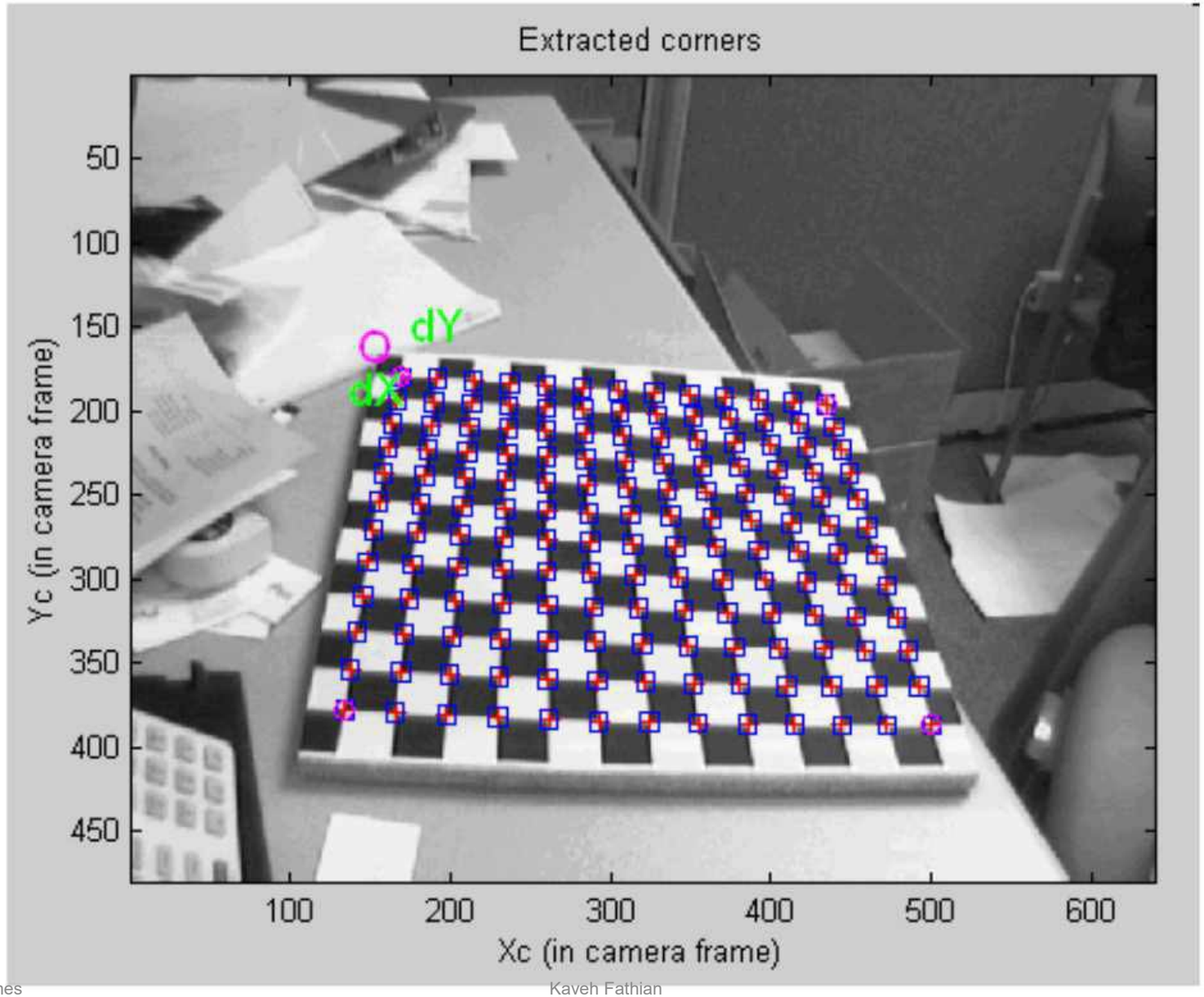


# Step-by-step demonstration

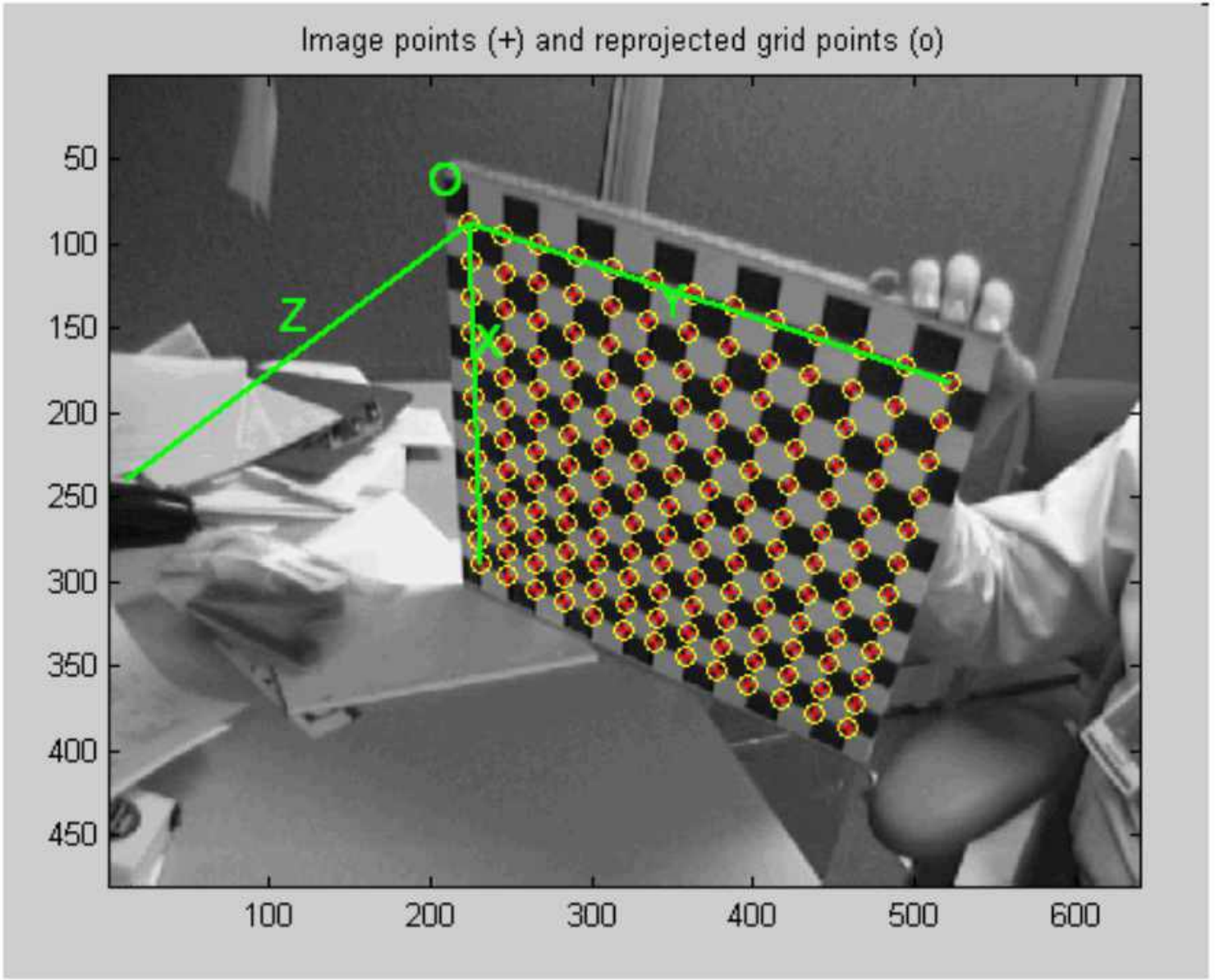


# Step-by-step demonstration



# Step-by-step demonstration



# Step-by-step demonstration