

LECTURE 6

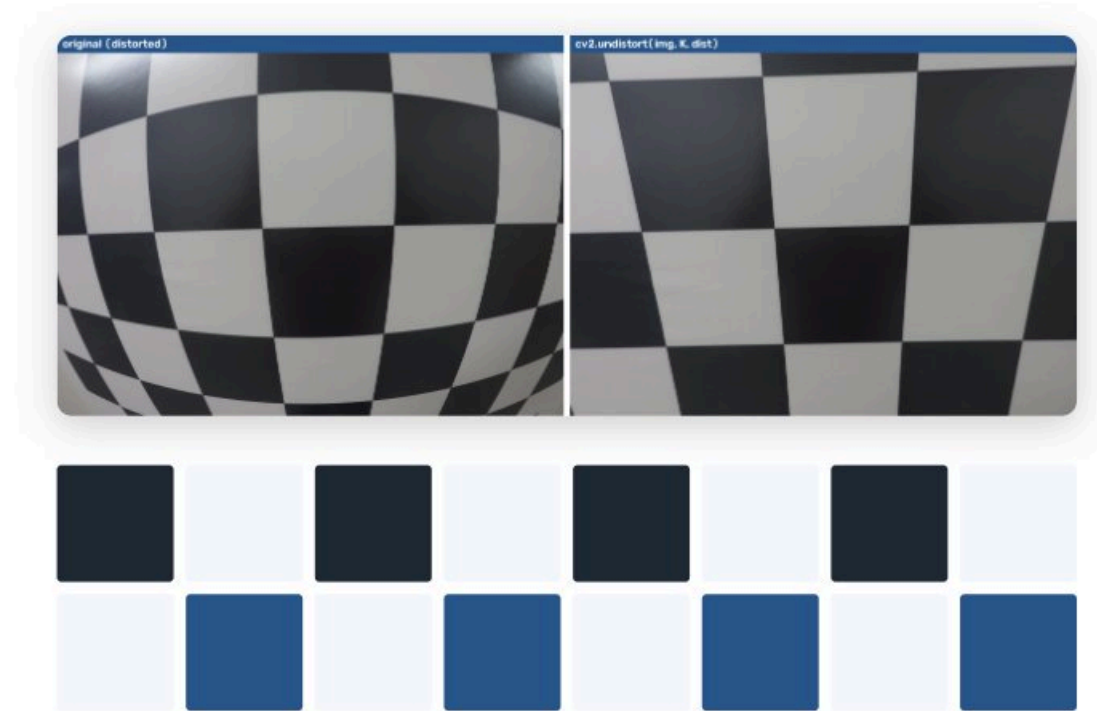
Camera Calibration

From pixels to the real world: camera models, lens distortion, calibration & perspective transforms

Pinhole, wide-angle, fisheye, stereo & Raspberry Pi cameras • hands-on with OpenCV • Lab 6

Dr. Muhammad Kazim

Department of Intelligent Systems
University of Lahore



From lane pixels to real-world measurements

Lecture 5 · you can already

- Select lane paint by **colour** and **region**
- Find edges with **Canny**, lines with **Hough**
- Draw two straight lane lines on **video**

...but everything is in **pixels**, and straight lines only.

Lecture 6 · today

- How a camera turns **3-D points into pixels** (pinhole model, $\mathbf{K}$, $[\mathbf{R} \mid \mathbf{t}]$)
- Why lenses **distort** and the 5 numbers that fix it
- **Calibrate** any camera with a chessboard (Zhang's method)
- Fisheye, ChArUco, stereo & Raspberry Pi cameras
- **Perspective transform** → bird's-eye view → lane **curvature** in metres



To steer, a car must measure how much its lane **curves**, which means looking at the road from above. That warp is only correct after we remove **lens distortion** (Udacity).

Learning outcomes

By the end of this lecture and Lab 6 you should be able to:

1. Write the **pinhole projection** $s \cdot x = K [R \mid t] X$ and explain every intrinsic and extrinsic parameter.
2. Describe **radial** and **tangential** distortion, the OpenCV coefficients `(k1, k2, p1, p2, k3)`, and when you need the **fisheye** or **omnidirectional** model instead.
3. Compare camera types (standard, wide-angle, fisheye, 360°, stereo, RGB-D, rolling vs global shutter) and calibration targets (chessboard, circles, ChArUco).
4. **Calibrate** a camera with OpenCV: corners → object/image points → `calibrateCamera` → **reprojection error**.
5. **Undistort** images and video efficiently, and judge a calibration's quality.
6. Calibrate a **stereo pair** and rectify it; calibrate your own **Raspberry Pi / USB camera**.
7. Apply a **perspective transform** to get a bird's-eye view and measure lane curvature.

How this lecture works: 8 levels, basics → advanced

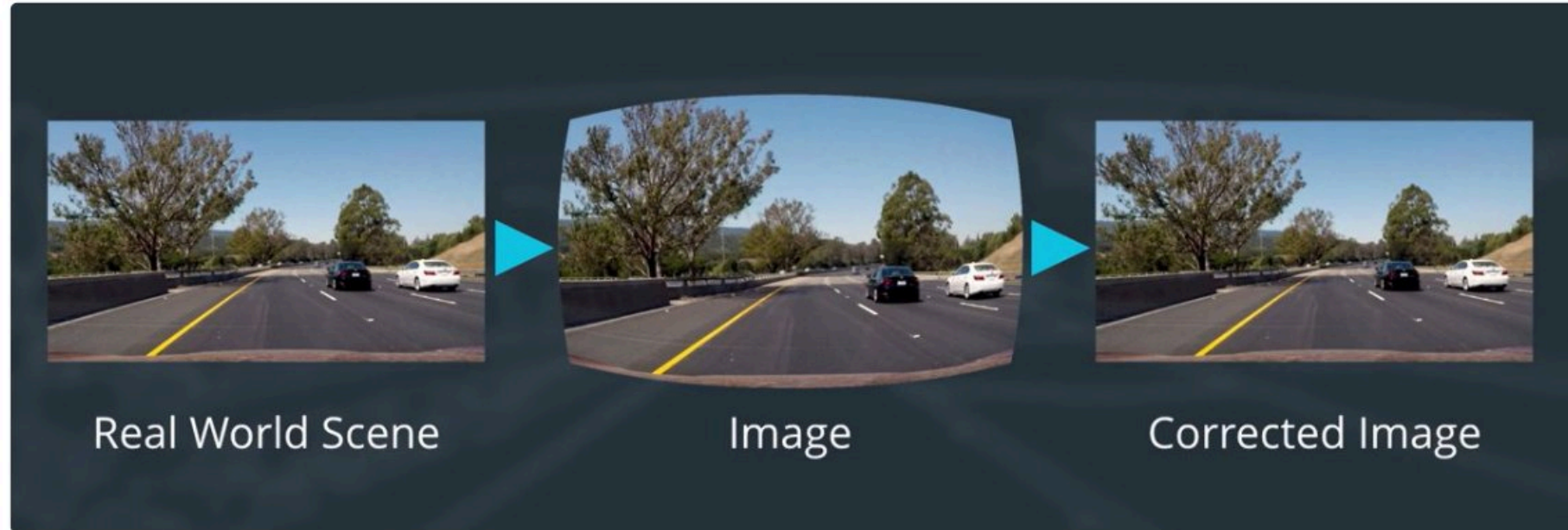
Level	Topic	Time	Hands-on
0	Pictures, pixels, lenses, coordinates	10 min	pixel explorer, phone camera
1	Why calibrate?	5 min	quiz
2	Pinhole model, K and $[R t]$	15 min	pinhole demo, K calculator · Lab 6A
3	Lens distortion	10 min	distortion explorer · Lab 6A
4	The calibration recipe in OpenCV	20 min	count corners, coverage demo · 6B, 6C
5	LIVE: calibrate the webcam	20 min	Webcam Calibration Studio · 6H
6	Use it: undistort, bird's-eye, measure	15 min	warp demo · 6D
7	Advanced: camera types, fisheye, ChArUco, stereo, Pi, Zhang	pick	6E, 6G, 6I–6M

Teacher checklist

- Print [pattern.png](#) at 100 %, glue on card, measure a square
- Laptop webcam + Chrome/Edge; open the [studio](#) once before class
- Press **S** for speaker notes (teaching tips on many slides)
- Backup: offline
[webcam_calibration_studio.py](#) or the studio's demo images

Students: every level ends with a **quiz** or **"Remember"** slide; 🖐 boxes are things to try yourself; ▶ buttons open the matching Colab notebook.

Roadmap: from the very basics to the advanced



The whole lecture in one picture: real world $\rightarrow$ (lens) $\rightarrow$ distorted image $\rightarrow$ (calibration) $\rightarrow$ corrected image (Udacity).

Camera calibration in one minute



1 • Show a known shape

Hold a printed **chessboard** in front of the camera. We know exactly where every corner is on the paper.



2 • See where it lands

Find the corners in the picture. Because of the lens and the viewpoint they are **not** where a perfect camera would put them.



3 • Solve for the numbers

Find the few numbers (**focal length, image centre, lens distortion**) that explain the difference. Do it once, use it forever.

Analogy: an eye doctor shows you a chart whose letters she knows, sees how you read it, and writes your **prescription**. The chessboard is the chart, the camera is the eye, and $K + \text{dist}$ is the camera's prescription. Afterwards we "put the glasses on": every image is corrected, and pixels can be turned into millimetres.

Key words in plain English

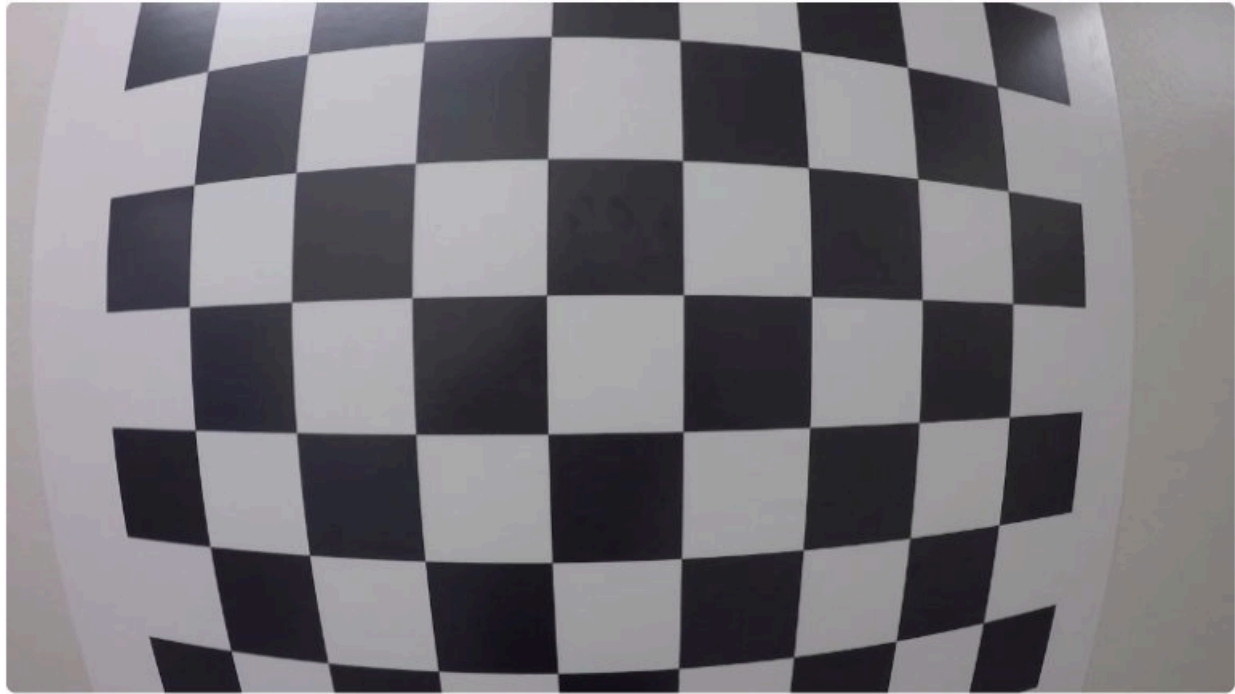
Word	Means	Everyday picture
Intrinsics, K	numbers inside the camera: focal length f_x, f_y and image centre c_x, c_y (in pixels)	the camera's ID card
Extrinsics, $[R \mid t]$	where the camera is (t) and where it looks (R)	your GPS position + which way you face
Distortion, $dist$	how much the lens bends lines near the edges: k_1, k_2, k_3 (radial), p_1, p_2 (tangential)	looking through a peephole or a glass of water
Object points	where the board corners really are (mm, $Z = 0$)	the answer sheet
Image points	where the corners appear in the photo (pixels)	the student's answers
Reprojection error	distance between where the model predicts a corner and where it was found	how far a dart lands from the bullseye
Undistort	re-draw the image as a perfect pinhole camera would have taken it	putting the glasses on
Homography / perspective transform	a 3×3 recipe that re-draws a flat thing as seen from another angle	photographing a poster straight-on

0

The very basics

Level 0 · pictures, pixels, lenses and three kinds of coordinates

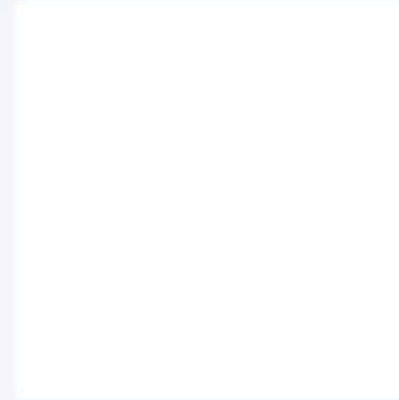
A digital picture is a grid of numbers



Move the mouse over the picture (it is `calibration_test.png`, 1280 × 720 pixels).

pixel position $(u, v) = (-, -)$

brightness = `-` (0 = black, 255 = white)



- **u** counts columns from the **left**, **v** counts rows from the **top**.
- (0, 0) is the **top-left** corner. **v** grows **downwards** (not like maths class!).
- OpenCV writes the size as (**width**, **height**); NumPy as `img.shape = (height, width)`.

Pixels are not millimetres



phone 1 m away

150 px tall

phone 2 m away

75 px tall

same 150 mm phone

A camera only gives **pixels**. How many pixels an object covers depends on

1. how **big** it is (mm),
2. how **far** it is (distance), and
3. **the camera itself**: its zoom (focal length) and lens.

Calibration measures item 3, the camera's own numbers. Then pixels can be turned into **millimetres, angles and distances**.

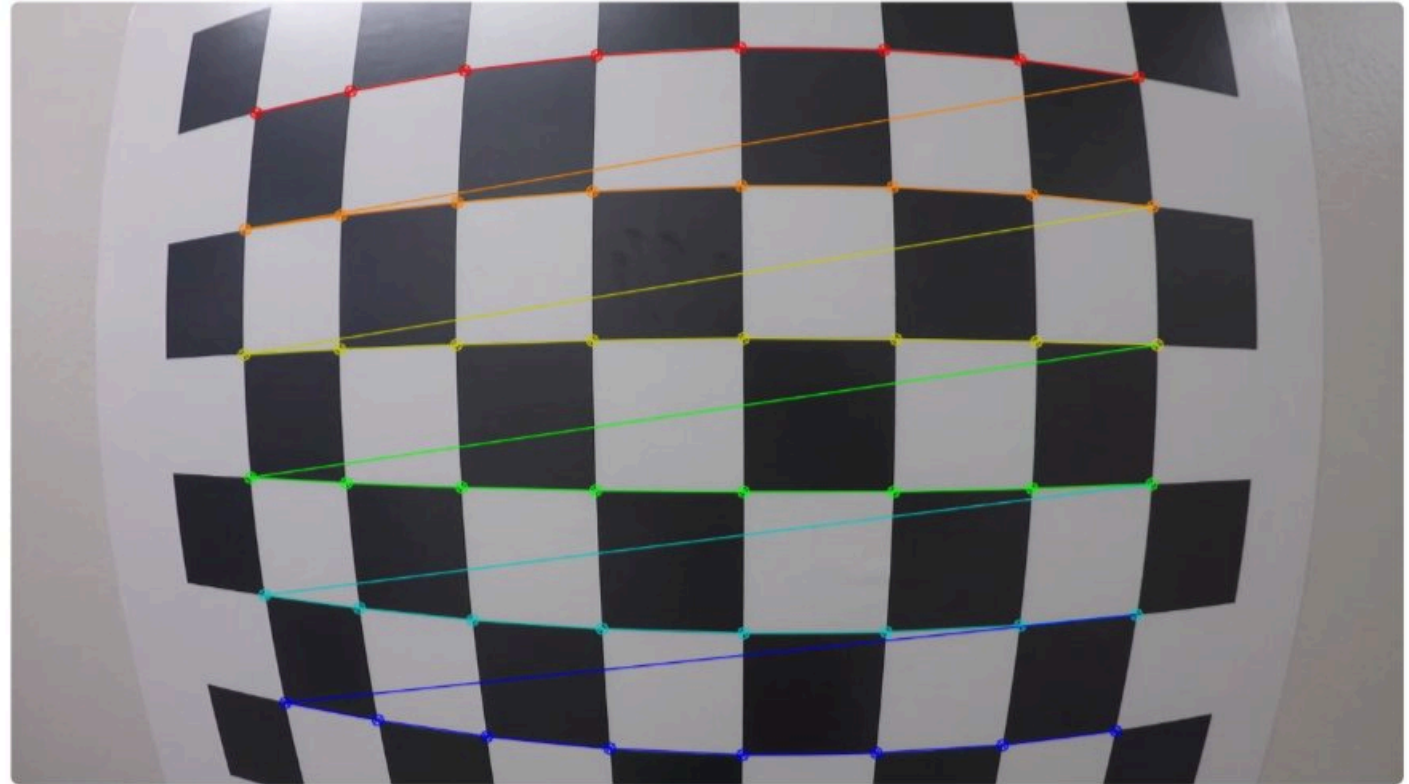
👉 **Try it:** hold your phone at arm's length, then at half the distance. How much bigger does it look on the webcam? (About $2 \times$: $\text{size} \propto 1 / \text{distance}$.)

What does a lens do?

- A lens **bends light** so that all rays from one point meet again at one point on the sensor, like a magnifying glass focusing sunlight to a spot.
- It lets in much more light than a tiny hole, so pictures are bright and sharp.
- But real lenses bend the rays near their **edges** a little too much or too little, so **straight lines look curved**, most of all near the corners of the picture.
- Wide lenses (phones' "0.5 ×", GoPro, dash-cams, many webcams) bend the most.

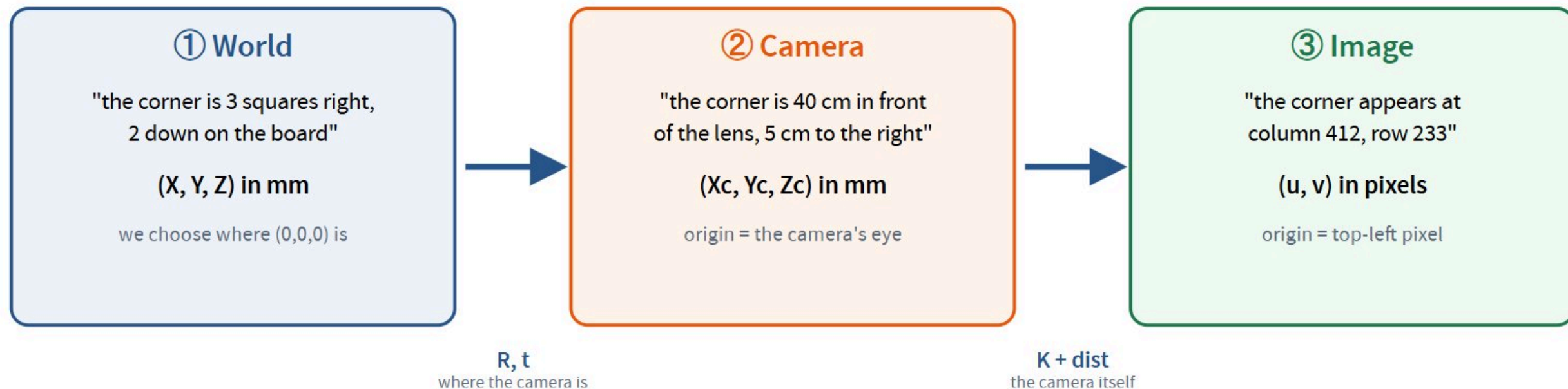


Try it: open your phone camera, switch to the ultra-wide (0.5 ×) lens and point it at a door frame. Put the frame near the edge of the picture: does it bend?



A flat chessboard through a wide-angle GoPro lens: the straight rows look curved.

A point lives in three "addresses"



In plain words: calibration finds the arrows. $K + \text{dist}$ (the camera's own numbers) are found once; R, t change every time the camera or the object moves.

Level 0 check: three quick questions

ANSWER OUT LOUD, THEN CLICK

1. An image is 640×480 . What are the (u, v) coordinates of its **centre**? Of the **bottom-right** pixel?
2. A box covers 100 px at 2 m. Roughly how many pixels at 4 m, with the same camera?
3. Which numbers never change for a camera (if you don't touch the lens or resolution): K and dist, or R and t?

1. Centre (320, 240); bottom-right (639, 479), because counting starts at 0.

2. About 50 px: twice as far $\rightarrow$ half as big.

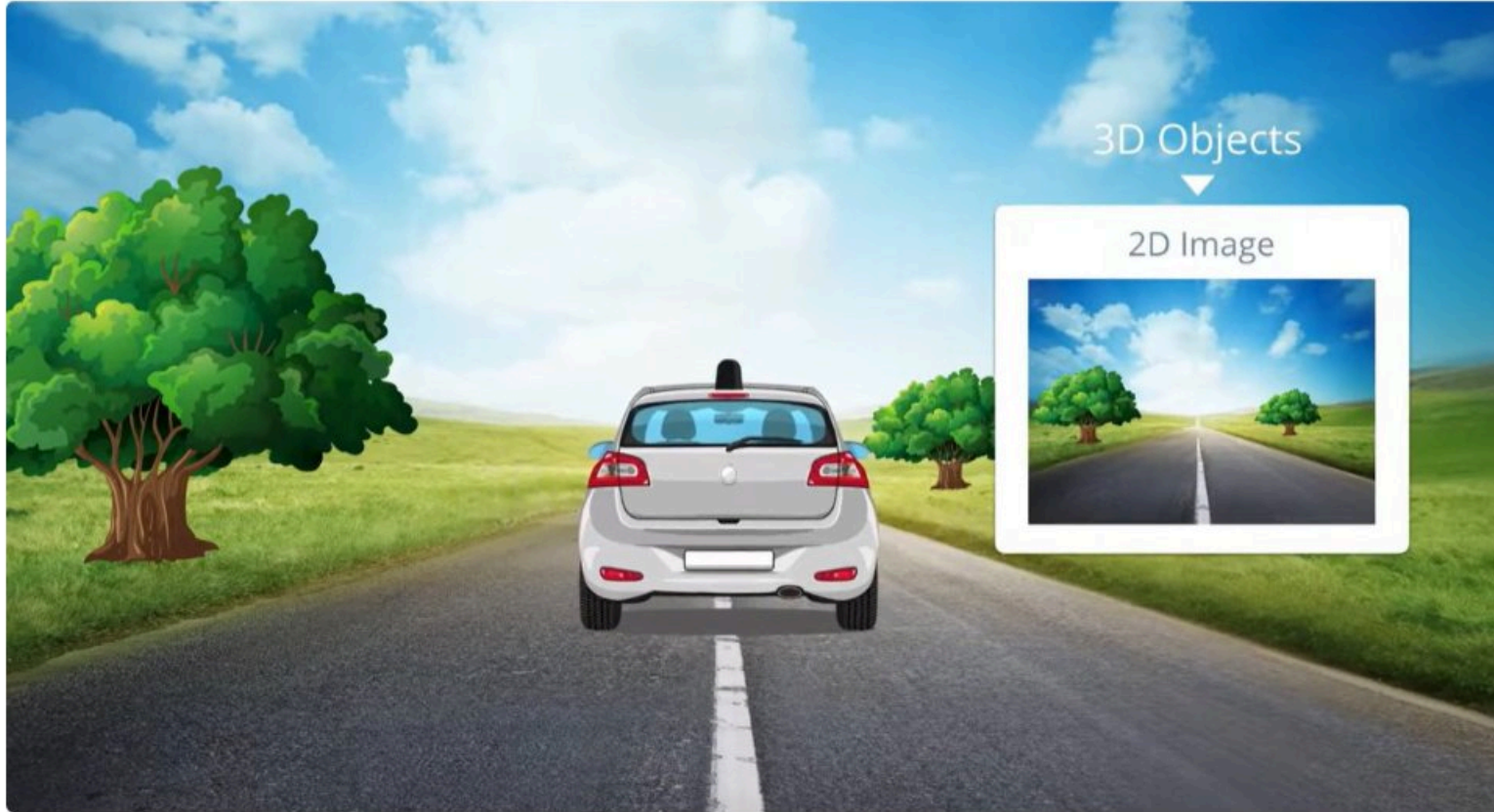
3. **K and dist** (the camera itself). R and t describe where the camera is, so they change whenever it moves.

1

Why calibrate a camera?

Level 1 · Why do cameras need calibrating?

Image distortion: 3-D world → imperfect 2-D image

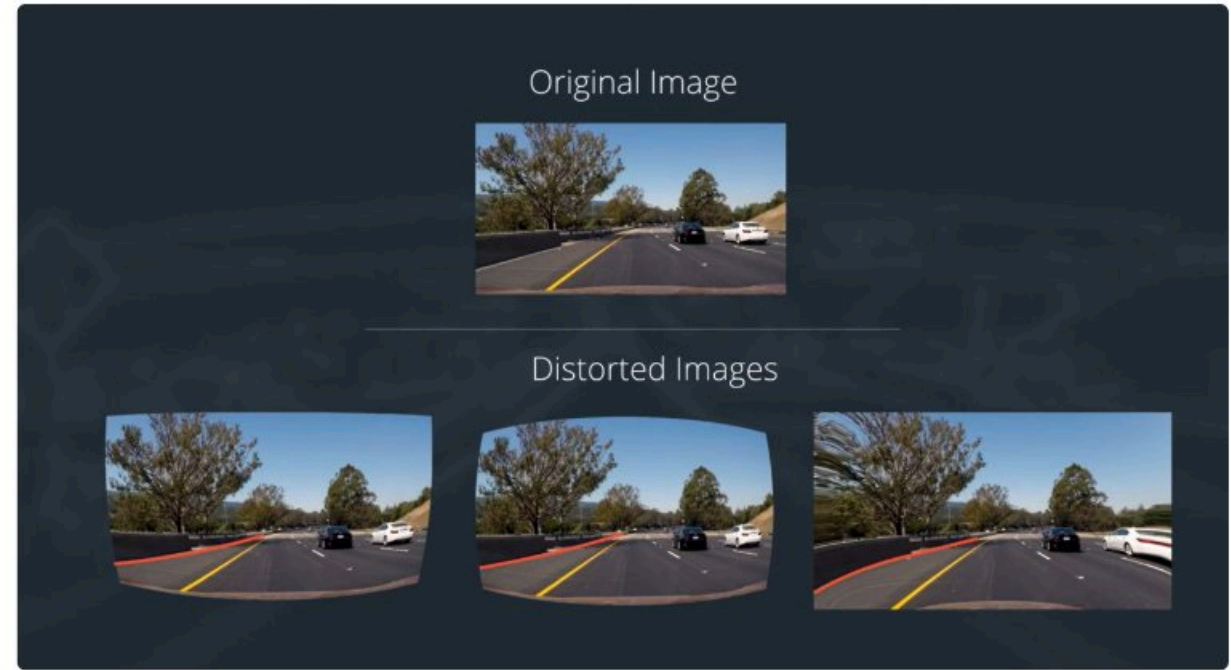


When a camera looks at 3-D objects and turns them into a 2-D image, the transformation is not perfect.

- Distortion changes the apparent **shape** and **size** of objects
- It is strongest near the **edges** of the image
- A distorted lane gives the **wrong curvature** → the wrong **steering angle**

So the **first step** of any measuring vision system is to **undo the distortion**.

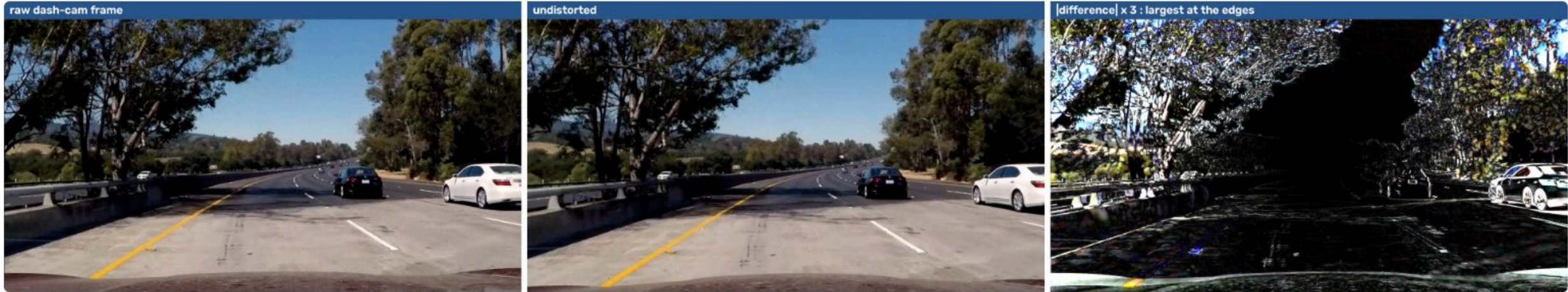
The same road through different lenses



The lane edges are **bent**, rounded or stretched outward. Distortion changes what the shape and size of objects appear to be, and a self-driving car must place itself in the world **accurately**.

Is it visible on a real dash-cam? new

► Lab 6D in Colab



A frame from the Udacity Advanced Lane Lines camera before and after undistortion (we calibrate this camera in Lab 6D). The difference image shows the correction is **largest near the image border**: exactly where the lane lines and nearby cars are.

Quiz: why is it important to correct for distortion?

QUIZ QUESTION · SELECT ALL THAT APPLY

- a. Distortion can change the apparent **size** of an object in an image.
- b. Distortion can change the apparent **shape** of an object in an image.
- c. Distortion can cause an object's appearance to change depending on **where it is in the field of view**.
- d. Distortion can make objects appear **closer or farther away** than they actually are.

All four. Each one breaks measurements: lane width, curvature, object size, distance to the car ahead.

What calibration gives you

Intrinsics K

Focal lengths f_x, f_y and principal point c_x, c_y , in pixels. Fixed for a given camera, lens, focus and resolution.

Distortion

$\text{dist} = [k1, k2, p1, p2, k3]$
(or the fisheye $[k1..k4]$). How the lens bends rays away from the ideal pinhole.

Extrinsics [R | t]

Where the camera is relative to the world (or to the board, or to a second camera). One pose per calibration image.

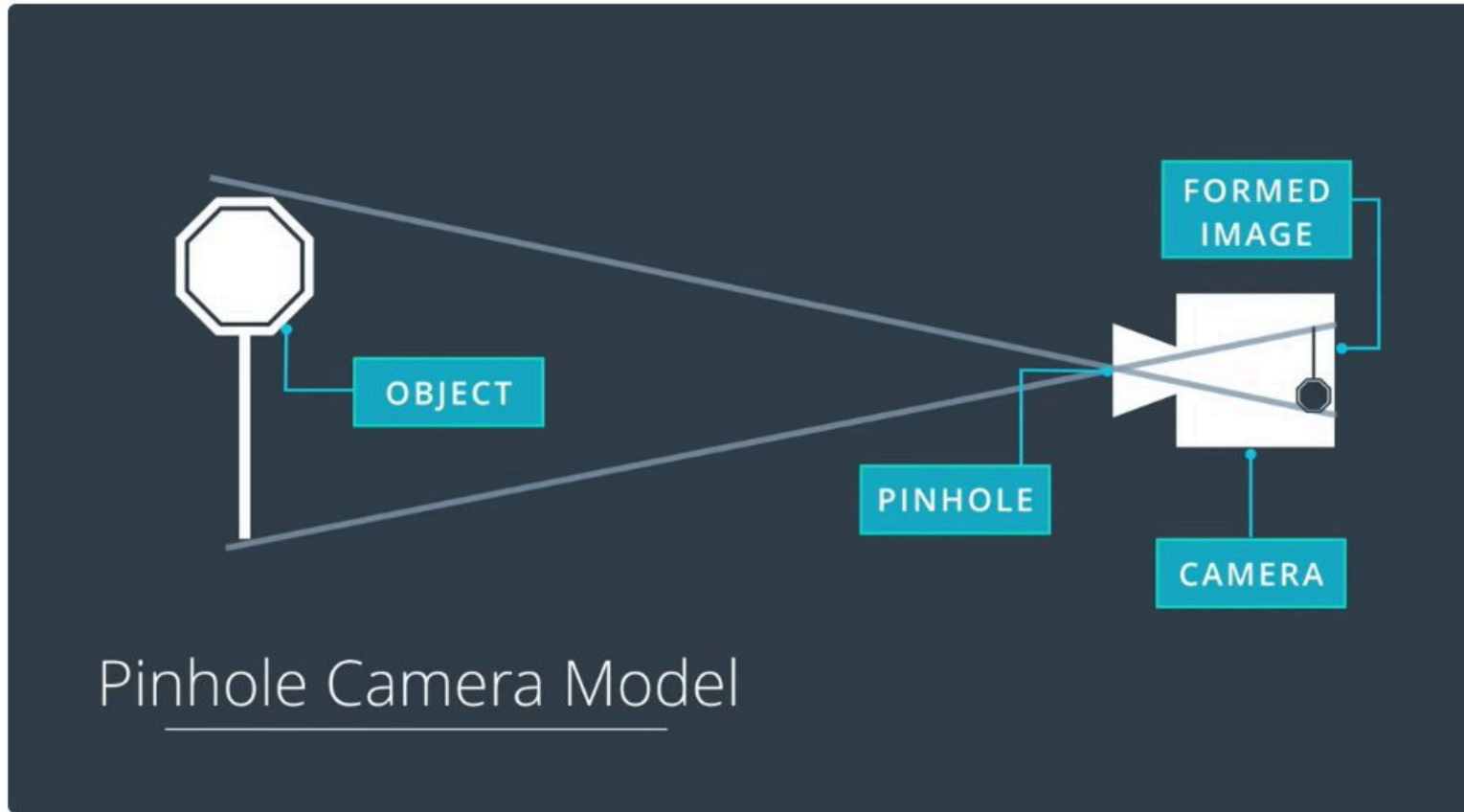
Application	Needs
Lane curvature, bird's-eye view (SDC)	K + distortion, then a perspective transform
Measuring object size in mm (inspection, Lab 9)	K + distortion + the distance or plane
Stereo depth (Lecture 10), visual SLAM, 3-D reconstruction	K, distortion and the relative pose of the cameras
AR overlays, robot pick-and-place, ArUco pose (<code>solvePnP</code>)	K + distortion → pose of objects
Multi-sensor fusion (camera + LiDAR / IMU)	intrinsics + extrinsic calibration between sensors

2

The pinhole camera model

Level 2 · How 3-D points become pixels

The pinhole camera

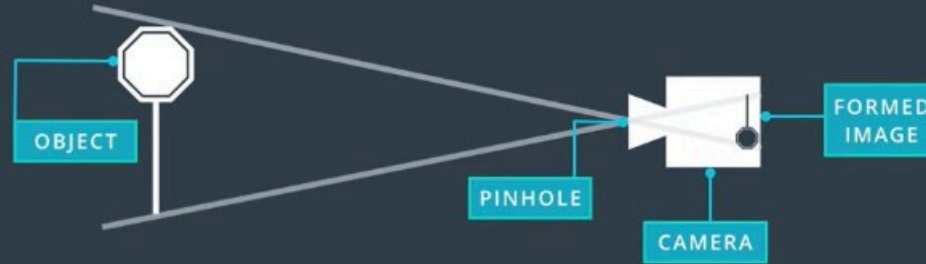


- Light reflected from an object passes through a **tiny hole** and forms an image on the back plane (film / sensor).
- The image is **upside-down and mirrored**: rays from the top of the object continue through the pinhole to the bottom of the image.
- The pinhole camera is the earliest means of projecting images: the camera obscura was described in China around 2,500 years ago.

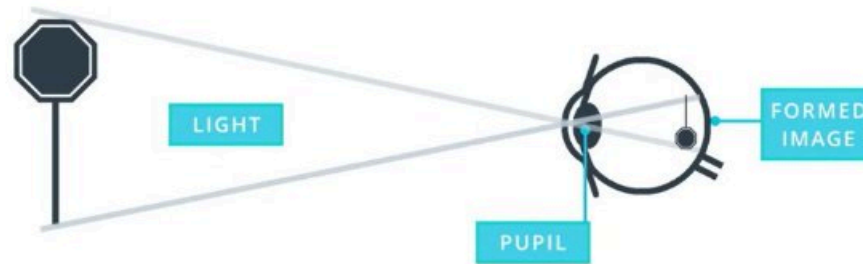
A pinhole has **no lens**, so it has **no distortion**; it just lets in very little light.

The camera and the human eye

Pinhole
Camera
Model



The
Human
Eye



The eye works the same way: the **pupil** is the pinhole and the **retina** is the image plane, and the image on it is upside-down too. The brain flips it.

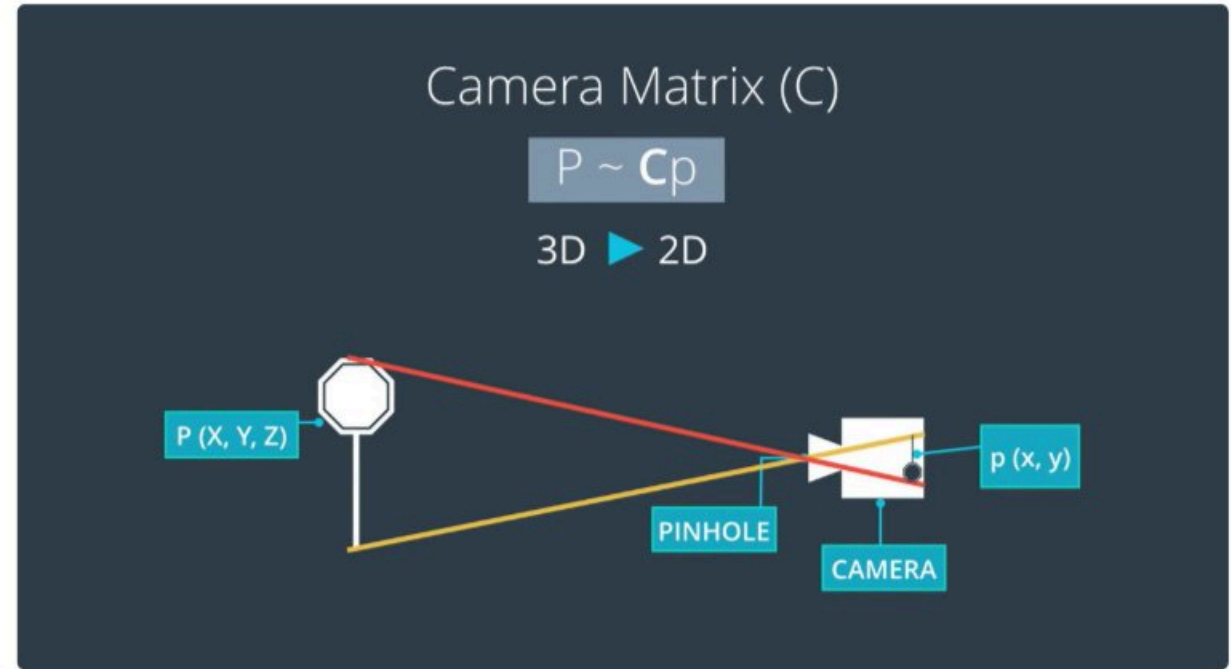
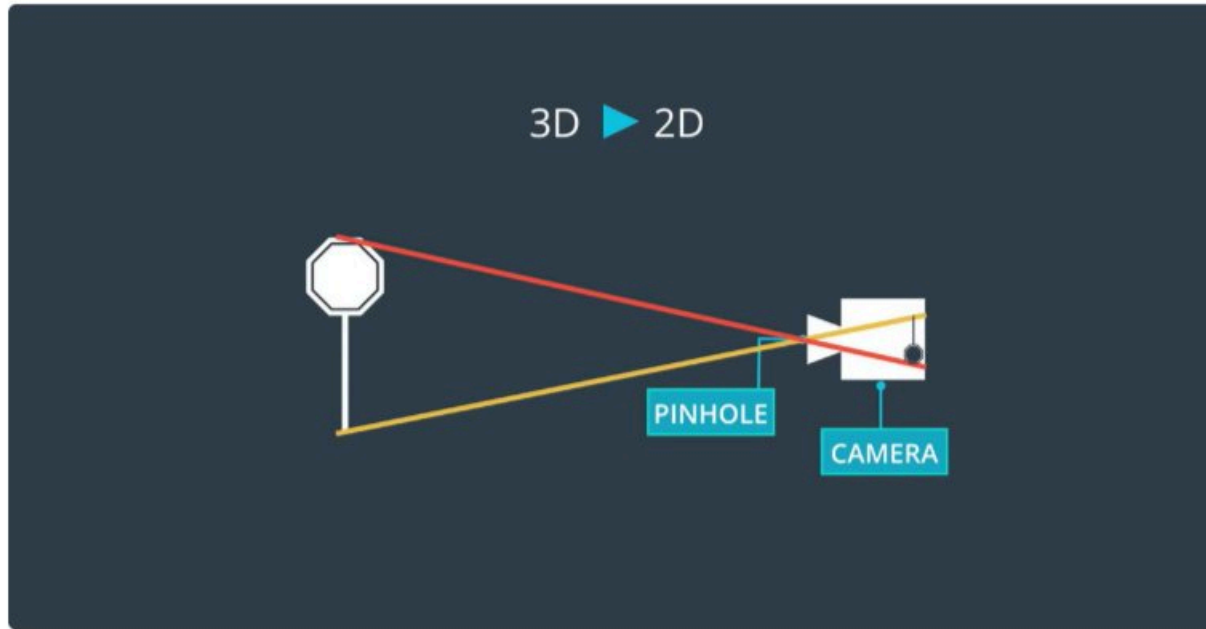
QUIZ

What is the fundamental difference between images formed with a pinhole camera and those formed using lenses?

- Pinhole images are always flipped, lens images upright
- Pinhole images are black and white
- Pinhole images are free from distortion; lenses tend to introduce distortion
- They are exactly the same

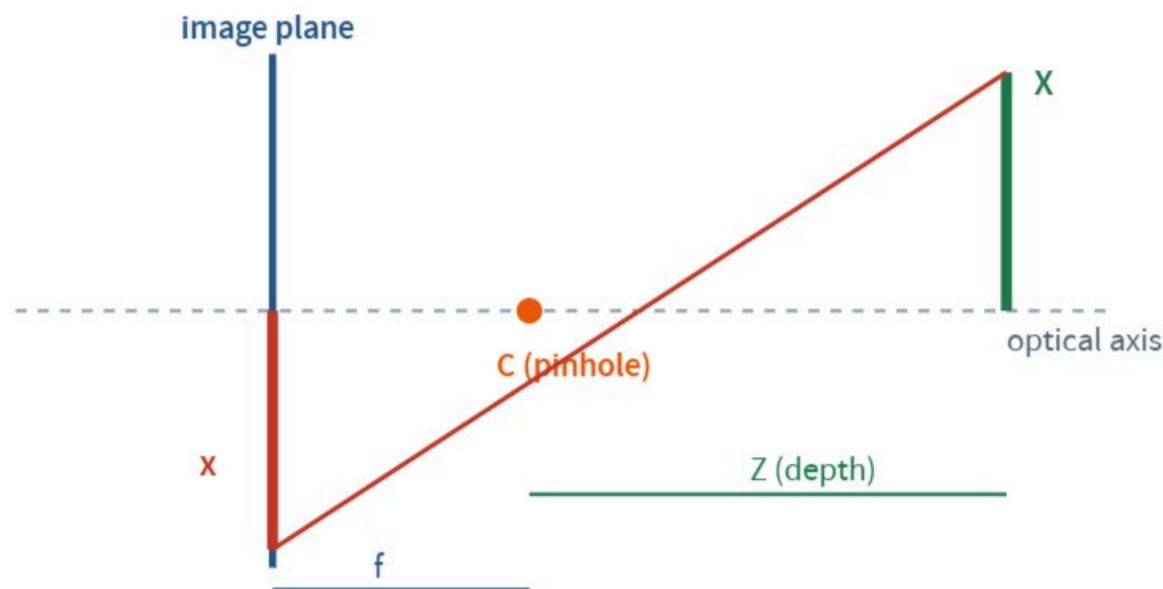
Free from distortion. Lenses gather more light but bend rays at their edges.

From 3-D to 2-D: the camera matrix



A 3-D object point $P(X, Y, Z)$ maps to a 2-D image point $p(x, y)$ through a **camera matrix** (Udacity calls it C , so $p \approx C P$). Calibration is the job of **measuring** that matrix plus the lens distortion.

Perspective projection by similar triangles



Similar triangles give, for a camera at the origin looking along Z:

$$x = f \frac{X}{Z}, \quad y = f \frac{Y}{Z}$$

- Objects **twice as far** appear **half as big**: the $1/Z$ is perspective.
- Depth is **lost**: every point on the ray through C lands on the same pixel. One camera cannot measure distance on its own.
- We put a virtual image plane **in front** of the pinhole so the image is not flipped.

In plain words: a camera is a box with a hole. Far things make small pictures (divide by Z), and a single photo cannot tell how far away something is.

From millimetres to pixels: the intrinsic matrix K

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \simeq \underbrace{\begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_K \begin{bmatrix} X/Z \\ Y/Z \\ 1 \end{bmatrix}$$

- f_x, f_y : focal length in **pixels** = f [mm] / pixel size [mm]. Equal for square pixels.
- c_x, c_y : **principal point**, where the optical axis hits the sensor ($\approx$ image centre, never exactly).
- s : skew, 0 for every modern sensor (OpenCV assumes 0).

In plain words: K turns "how far left/up the ray points" into a pixel position. f says how much the camera zooms; (c_x, c_y) is the pixel straight in front of the lens.

Worked example: Pi Camera v2

Sony IMX219: $f = 3.04$ mm, pixel = $1.12 \mu\text{m}$, 3280×2464 px.

$f_x = 3.04 / 0.00112 \approx \mathbf{2714 \text{ px}}$

$c_x \approx 3280/2 = \mathbf{1640}$, $c_y \approx \mathbf{1232}$

Horizontal FOV = $2 \cdot \text{atan}(1640 / 2714) \approx \mathbf{62^\circ}$

✓ (datasheet: 62.2°)

Datasheet values are only a starting guess. Calibration measures the real ones.

Try it: K from a datasheet interactive

Pi Cam v2

Pi Cam 3

Pi Cam 3 Wide

Global Shutter

USB webcam

focal length f (mm)

pixel size (μm)

width (px) height (px)

resize to width (px) (scaling the image scales K!)

full resolution 3280 x 2464

$f_x = f_y = f / \text{pixel} = 2714.3 \text{ px}$

$$K = \begin{bmatrix} 2714.3, & 0.0, & 1640.0, \\ 0.0, & 2714.3, & 1232.0, \\ 0.0, & 0.0, & 1.0 \end{bmatrix}$$

horizontal FOV = 62.3 deg

vertical FOV = 48.8 deg

resized to 640 x 481 (scale 0.195):

$f_x' = 529.6 \quad c_x' = 320.0 \quad c_y' = 240.4$

Sensor data from the Raspberry Pi camera documentation. Lens values of webcams vary: treat the preset as an example.

Where is the camera? Extrinsics $[R \mid \mathbf{t}]$

World points are usually not given in the camera's frame. A rotation R (3×3) and translation $\mathbf{t}$ (3×1) move them there:

$$\mathbf{X}_{cam} = R \mathbf{X}_{world} + \mathbf{t}$$

Putting it all together, the full **projection** (Lecture 10 notation):

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K [R \mid \mathbf{t}] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = P \tilde{\mathbf{X}}$$

$P = K[R \mid \mathbf{t}]$ is the 3×4 **projection matrix**: 4 intrinsic + 6 extrinsic numbers (3 rotation, 3 translation).

In OpenCV

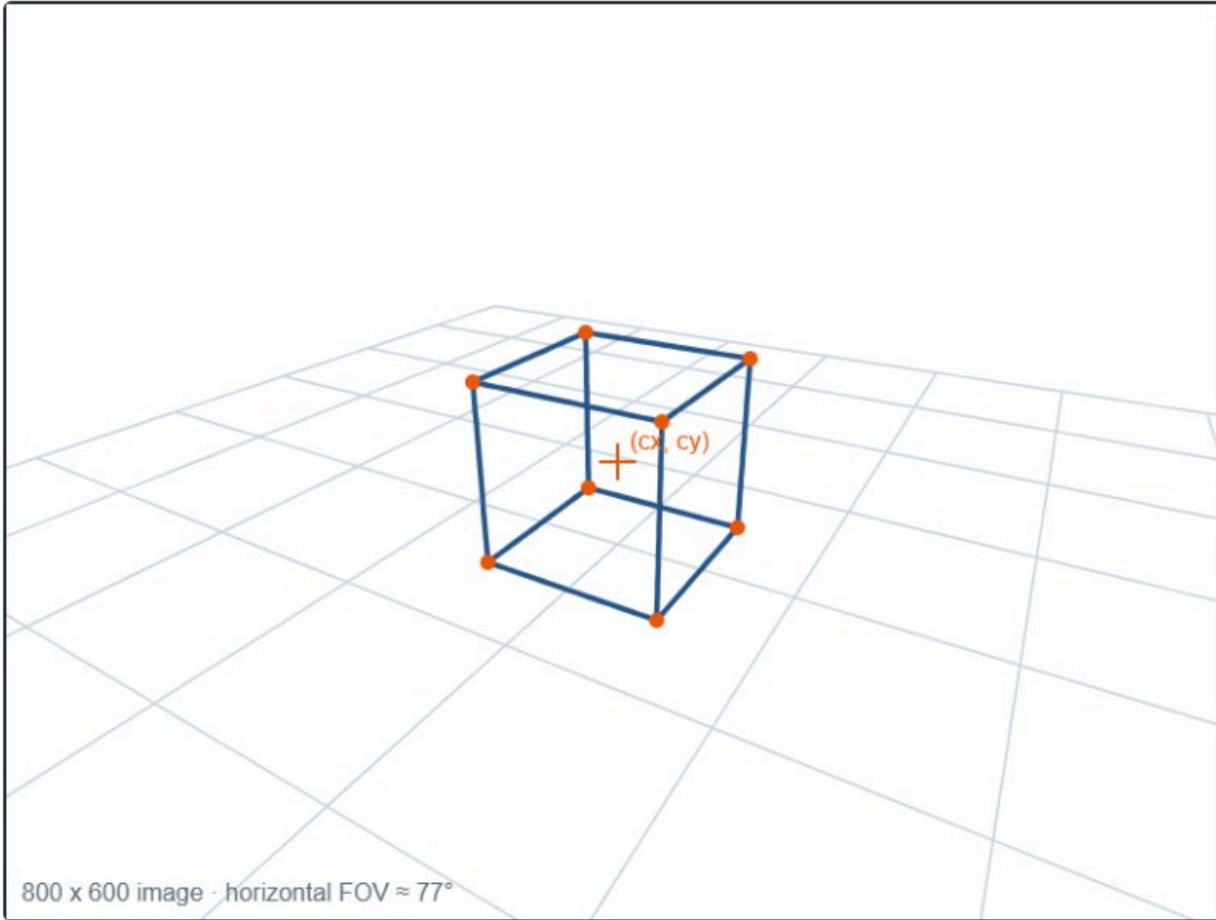
- R is stored as a 3-vector `rvec` (axis $\times$ angle, **Rodrigues**): `R, _ = cv2.Rodrigues(rvec)`
- `calibrateCamera` returns one `rvec`, `tvec` **per calibration image**: the pose of the board
- `cv2.projectPoints(X, rvec, tvec, K, dist)` applies the whole model, including distortion
- `cv2.solvePnP` does the inverse: known $K + 3\text{-D}/2\text{-D}$ matches $\rightarrow$ pose

In plain words: first move the world so the camera is at the origin looking along Z ($R, \mathbf{t}$), then let K turn it into pixels.

Live demo: the pinhole camera

interactive

► Lab 6A in Colab



focal length $f_x = f_y$ 500
principal point c_x 400
distance Z (m) 4.0
orbit around ($^\circ$) 30
camera elevation ($^\circ$) 25
barrel distortion k_1 0.00

```
K = [[ 500,    0, 400],  
      [    0, 500, 300],  
      [    0,    0,   1]]  
camera centre C = [1.81, -1.19, -3.14] m  
dist = [0.00, 0, 0, 0, 0]
```

A 1 m cube on a floor grid of 1 m squares; the camera looks at the cube. Change f : the image zooms (FOV changes). Move Z : size $\propto 1/Z$. Add $k_1 < 0$: straight edges bow outwards.

Quiz: project a point by hand

QUIZ QUESTION

A camera has $f_x = f_y = 800$ px, $c_x = 640$, $c_y = 360$, no distortion, and $R = I$, $t = 0$. Where does the point $X = (0.5, -0.25, 4)$ m land?

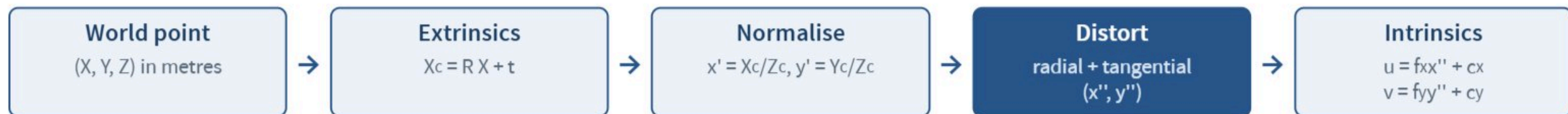
$u = 800 \cdot 0.5 / 4 + 640 = 740$, $v = 800 \cdot (-0.25) / 4 + 360 = 310$. Pixel (740, 310).

And a point twice as far, $(1, -0.5, 8)$?

The same pixel (740, 310): it lies on the same ray. Depth cannot be recovered from one image.

```
import numpy as np, cv2
K = np.array([[800, 0, 640],
              [0, 800, 360],
              [0, 0, 1]], float)
X = np.array([[0.5, -0.25, 4.0],
              [1.0, -0.50, 8.0]])
uv, _ = cv2.projectPoints(X, np.zeros(3), np.zeros(3),
                           K, None)
print(uv.reshape(-1, 2))  # [[740. 310.] [740. 310.]
```

Image formation: the full chain in OpenCV



Distortion is applied to the **normalised** coordinates (units of focal length), not to pixels. That is why the same **dist** vector works at any image size, while **K must be scaled** when you resize.

Udacity's notes write r as the distance from the **distortion centre** ($\approx$ image centre). In OpenCV it is $r^2 = x'^2 + y'^2$ in normalised coordinates.

Watch: [Camera Intrinsics and Extrinsics: 5 Minutes with Cyrill](#) (Cyrill Stachniss, Uni Bonn).

The pinhole model: the 3 things to remember

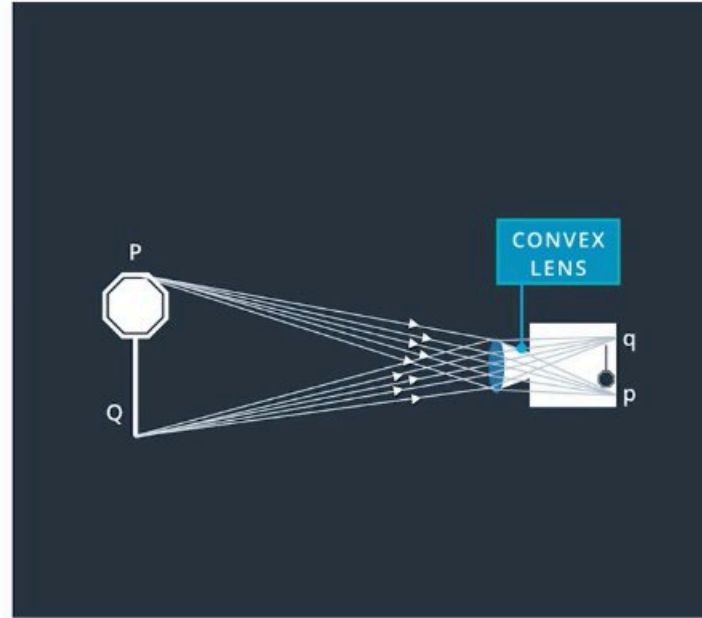
1. A camera divides by depth: $\mathbf{x} = \mathbf{f} \cdot \mathbf{X} / \mathbf{Z}$. Far things are small, and depth is lost.
2. $\mathbf{K} (f_x, f_y, c_x, c_y)$ = the camera itself. $[\mathbf{R} \mid \mathbf{t}]$ = where it stands and looks.
3. In OpenCV: `cv2.projectPoints(X, rvec, tvec, K, dist)` does the whole chain.

3

Lens distortion

Level 3 · Real cameras use lenses, and lenses bend light

Why real cameras use lenses



- A pinhole lets through **one ray** per scene point: a dark image or a very long exposure.
- A **convex lens** focuses **many rays** at once: bright images and fast shutter speeds.
- But rays bend a little **too much or too little** near the edges of a curved lens → **distortion**.

Radial distortion: the most common type



Light rays bend more (or less) the **further they are from the centre** of the lens. Lines and objects appear more or less **curved** than they are.

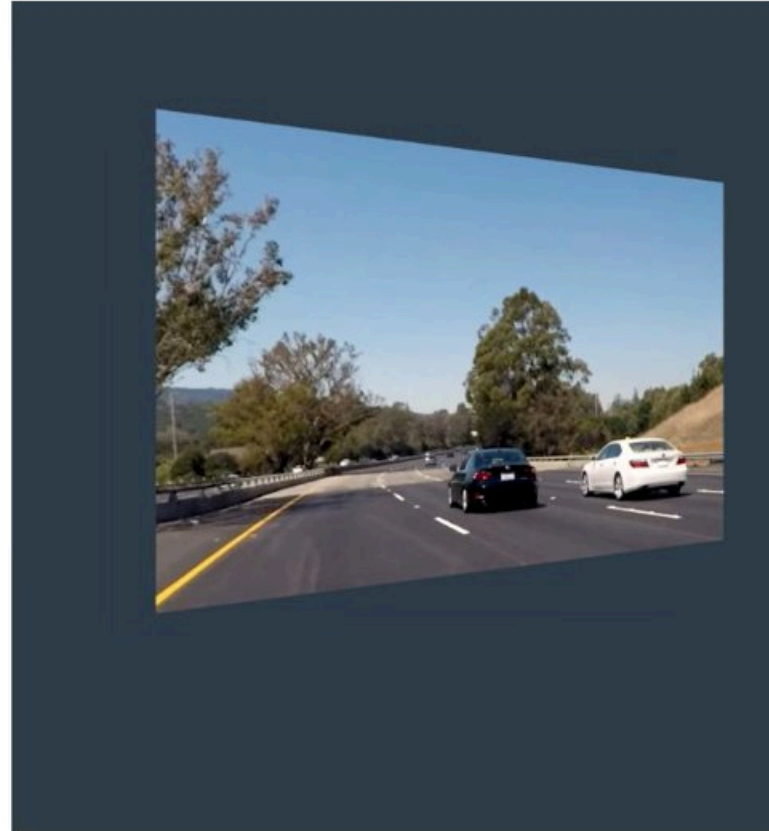
- **Barrel** ($k_1 < 0$): lines bow **outwards**: wide-angle, GoPro, dash-cams
- **Pincushion** ($k_1 > 0$): lines bow **inwards**: telephoto, some zooms
- **Mustache**: barrel in the centre, pincushion at the edge (k_1, k_2 opposite signs)

In plain words: the lens squeezes (barrel) or stretches (pincushion) the picture more and more the further you go from the centre.

Tangential distortion: a tilted lens



Tangential Distortion



Occurs when the lens is **not perfectly parallel** to the imaging plane (sensor), for example from glue or assembly tolerances in cheap modules.

The image looks **tilted**: some objects appear farther away or closer than they actually are.

Usually much smaller than radial distortion, but cheap phone, Pi and webcam modules can show it.

Some lenses distort on purpose



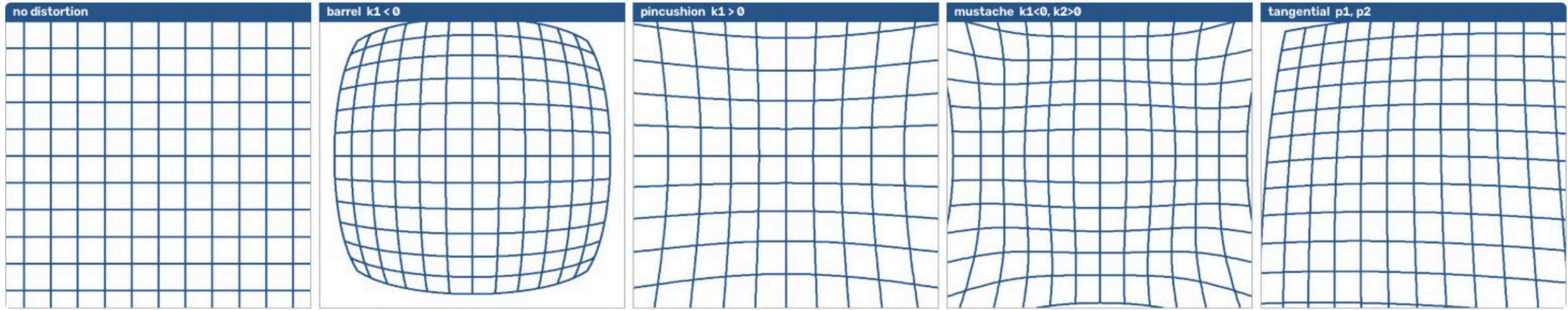
A convex car mirror: wider view, objects look farther away



Fisheye photography: 180° field of view, extreme barrel

Fine for style or a wide view, but for **measurement** (steering, sizing, depth) we need images that accurately reflect the real world.

Distortion types on a straight grid



Generated with `cv2.projectPoints` on a flat grid: same K , different `dist` vectors.

Five numbers capture the distortion

► Lab 6A in Colab

Distortion_{coefficients} = (k₁ k₂ p₁ p₂ k₃)

Radial, with $r^2 = x^2 + y^2$ (normalised coordinates):

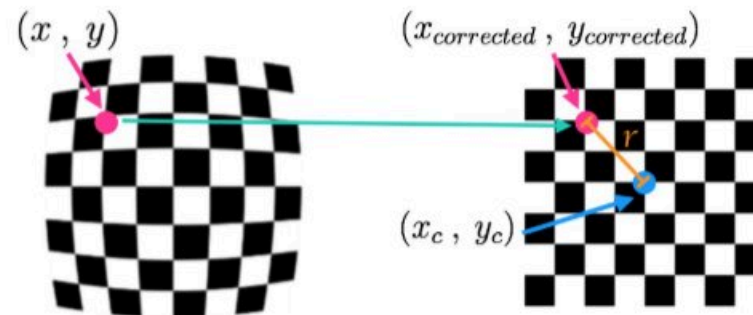
$$x_d = x (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_d = y (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Tangential:

$$x_d = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_d = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$



- OpenCV order: [k₁, k₂, p₁, p₂, k₃]. k₃ comes **last** because it is only needed for strong (wide-angle) distortion.
- Optional extras: rational k₄–k₆, thin-prism s₁–s₄, tilted sensor τ_x, τ_y, so up to 14 numbers.
- The equations go **ideal** → **distorted**. Undistortion inverts them numerically.

Distortion with real numbers (you can do this by hand)

A point near the edge

Normalised point $(x, y) = (0.8, 0.0)$, so $r^2 = 0.64$. Lens: $k_1 = -0.2$, all others 0 (a mild barrel).

factor = $1 + k_1 \cdot r^2 = 1 - 0.2 \times 0.64 = 0.872$

$x_d = 0.8 \times 0.872 = 0.698$: the point moves **13 % towards the centre**.

With $f_x = 800$ px that is $(0.8 - 0.698) \times 800 \approx 82$ pixels!

A point near the centre

$(x, y) = (0.1, 0.0)$, so $r^2 = 0.01$.

factor = $1 - 0.2 \times 0.01 = 0.998$

$x_d = 0.0998$: it moves only **0.0002**, about **0.2 pixels**.

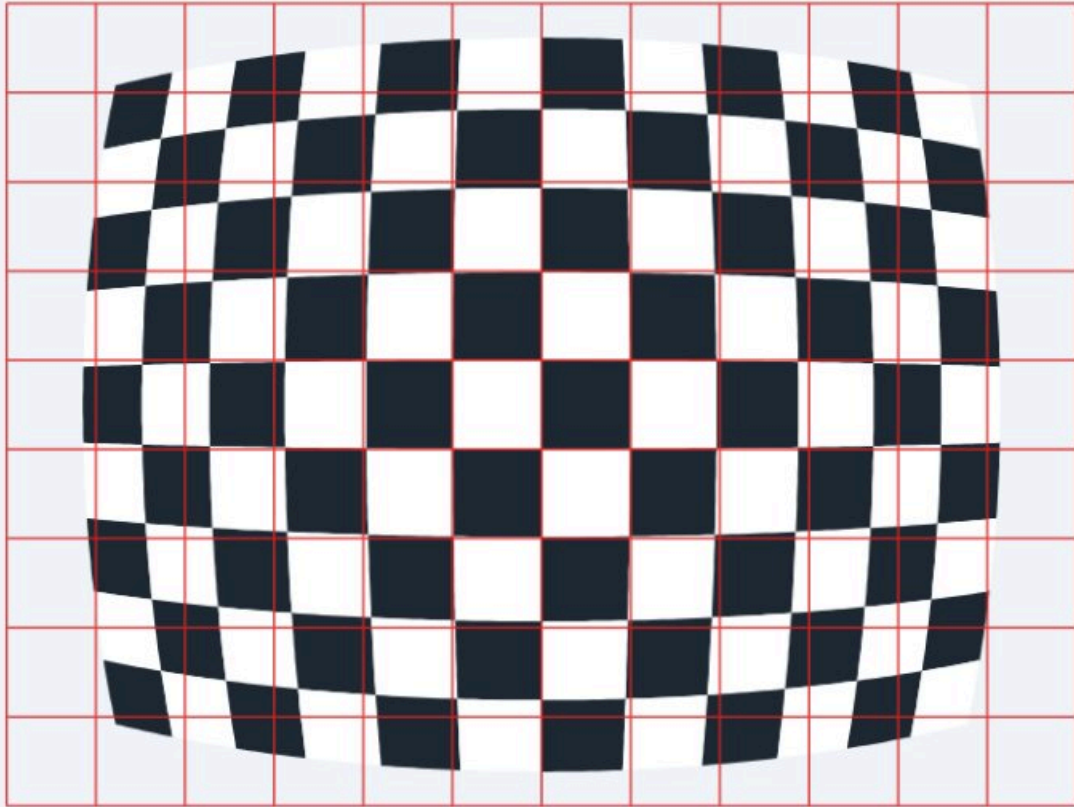
That is why the centre of a photo looks fine and the edges bend: the effect grows with r^2 .

Take-away: distortion is tiny in the middle and huge at the edges, so calibration photos **must** have the board near the edges and corners.

Live demo: lens distortion explorer

interactive

► Lab 6A in Colab



k_1 -0.25
 k_2 0.05
 k_3 0.00
 p_1 0.000
 p_2 0.000

☒ show the ideal (undistorted) board in red

barrel

pincushion

mustache

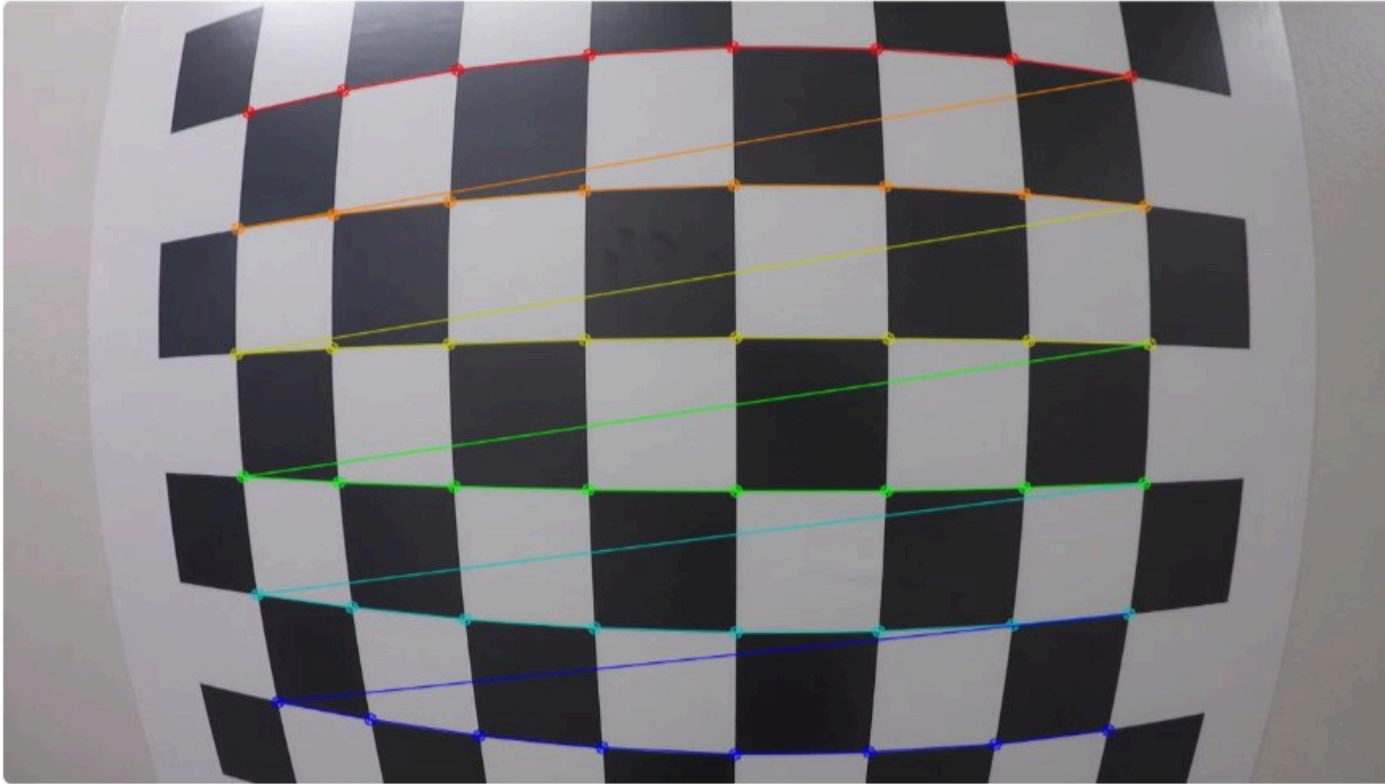
tangential

GoPro (Lab 6)

reset

```
dist = [k1, k2, p1, p2, k3]  
      = [-0.25, 0.05, 0.000, 0.000, 0.00]
```

Quiz: name that distortion



QUIZ QUESTION

This chessboard was shot with a GoPro (the camera we calibrate in Lab 6).

- What kind of distortion is it? What sign do you expect for k_1 ?
- Where in the image is the distortion strongest?
- Would a pure pinhole camera show it?

A: barrel $\rightarrow k_1 < 0$ (Lab 6 measures $k_1 \approx -0.23$). **B:** at the edges and corners (large r). **C:** No, a pinhole has no lens.

Lens distortion: the 3 things to remember

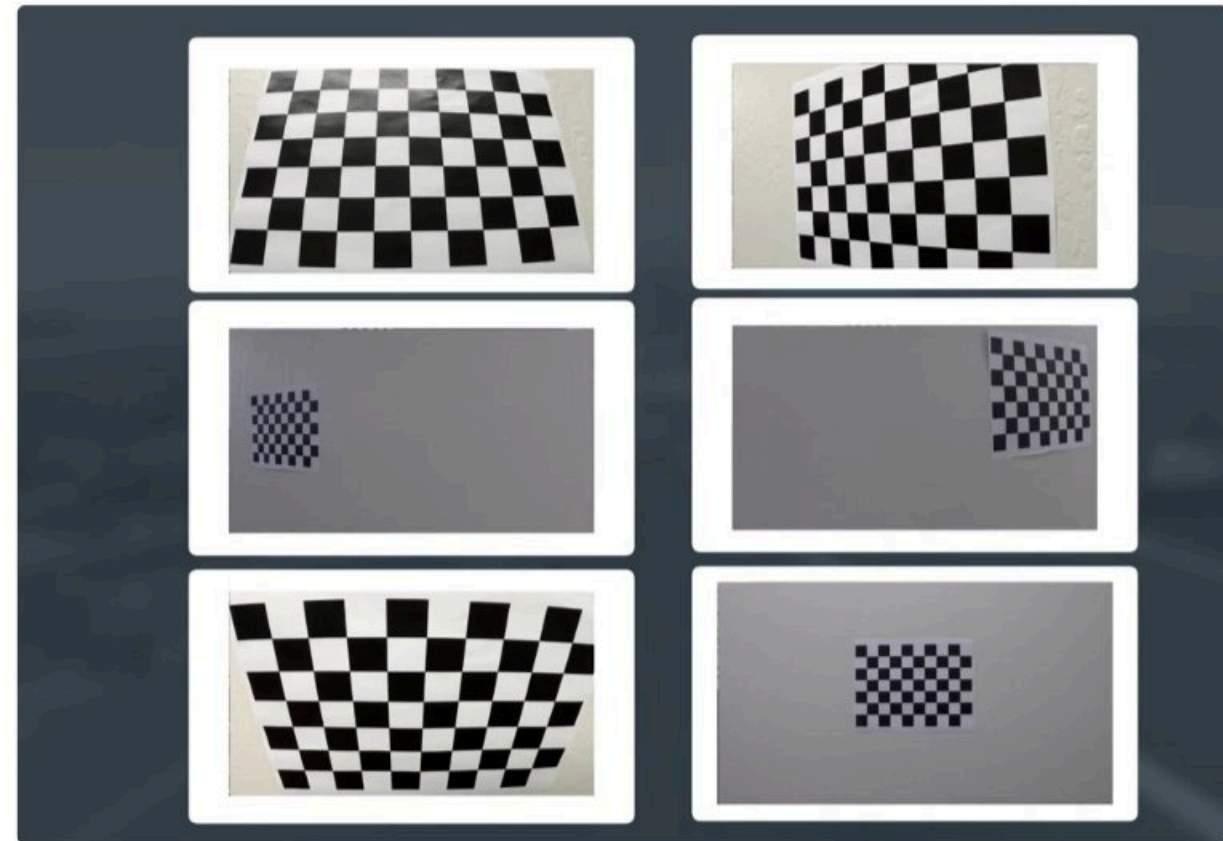
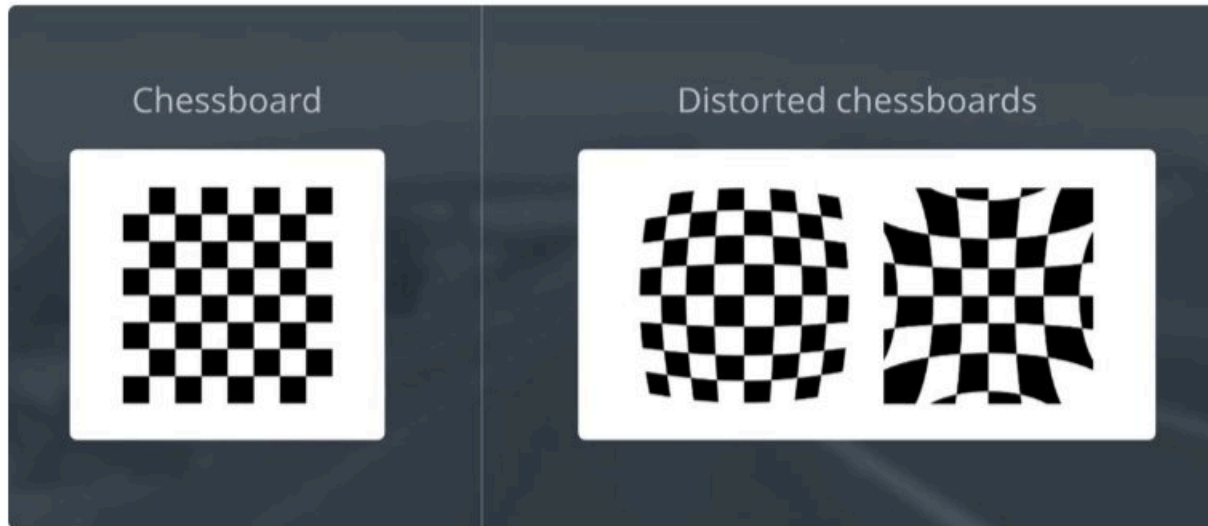
1. Real lenses bend lines, mostly **radially**: barrel ($k_1 < 0$) or pincushion ($k_1 > 0$).
2. OpenCV describes it with 5 numbers: `[k1, k2, p1, p2, k3]`.
3. It is small in the centre and large at the edges (it grows with r^2).

4

Calibrating a camera with OpenCV

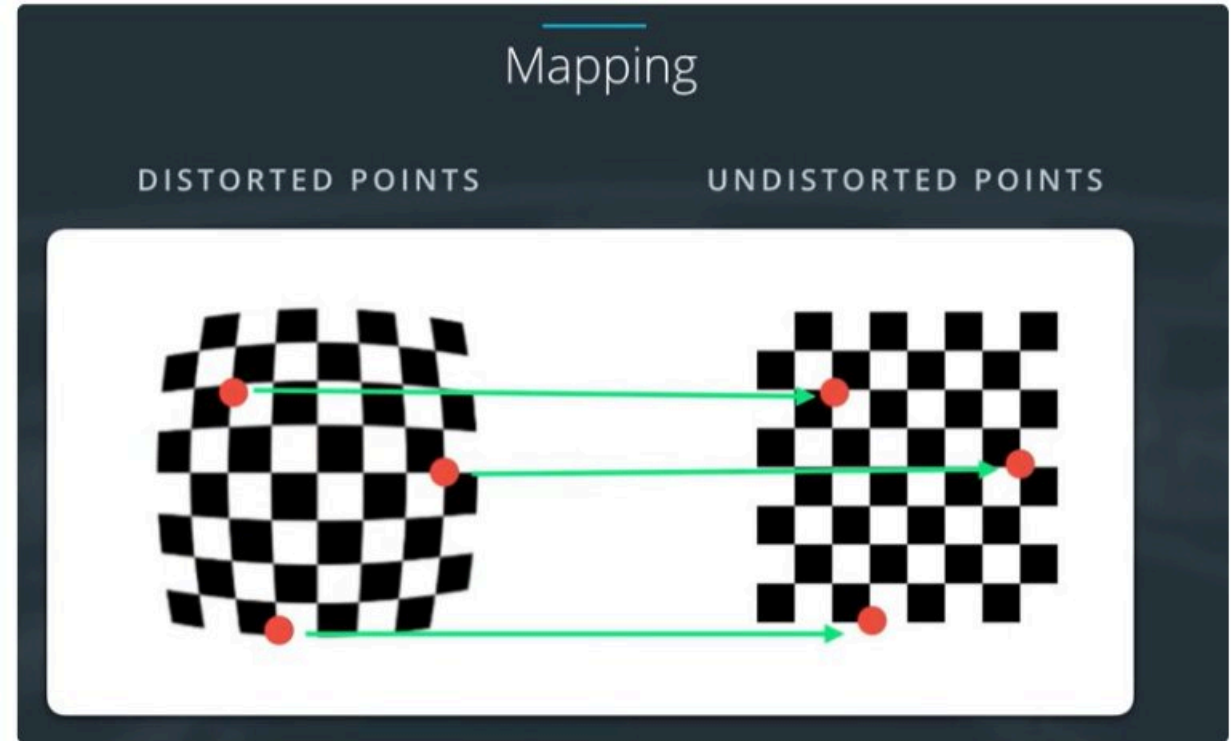
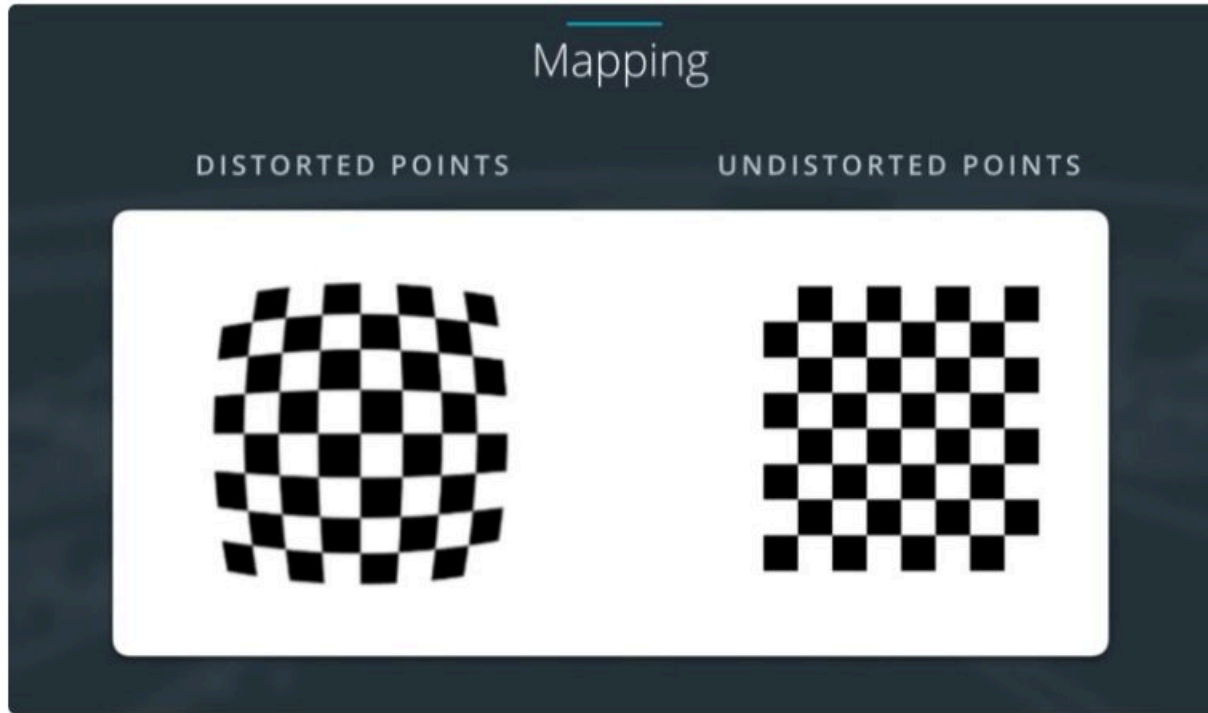
Level 4 · The recipe: chessboard → corners → K and dist

Measuring distortion with a known shape



We know exactly what a **flat chessboard** looks like, and its regular high-contrast pattern is easy to detect automatically. Take **many pictures** of it at different angles and distances; the difference between the **apparent** and the **real** square shapes reveals the distortion.

Map distorted points → undistorted points

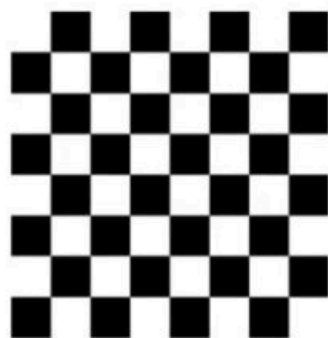


1. **Detect** the corners in each distorted image: the **image points** (2-D, pixels).
2. Write down where they **really** are on the flat board: the **object points** (3-D, $Z = 0$).
3. **Calibrate**: find K , $dist$ and each pose that best maps object points to image points.
4. **Undistort** any new image from that camera.

Quiz: how many corners? interactive

[▶ Lab 6B in Colab](#)

Map distorted points to undistorted points



QUIZ QUESTION

Count the **inside corners** in a row (**nx**) and in a column (**ny**) of the right-hand board. Only points where **two black and two white** squares meet count, not the outside corners.

nx = ny = check

nx = 8, ny = 6. The board has 9×7 squares $\rightarrow 8 \times 6$ inner corners. This is the `patternSize` you pass to OpenCV.

Finding corners: `findChessboardCorners`

[► Lab 6B in Colab](#)

calibration image (GoPro, wide-angle)

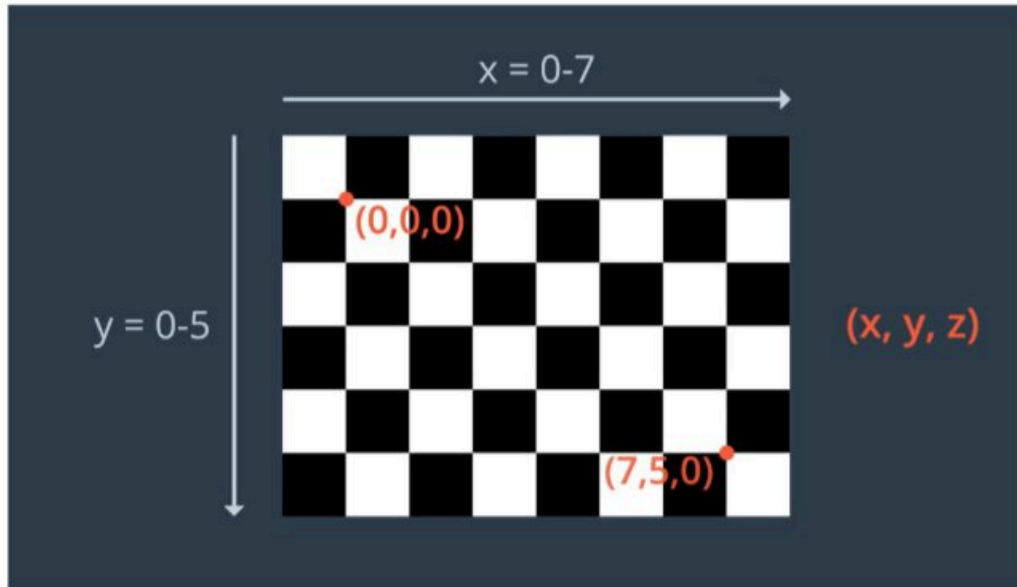


`findChessboardCorners` (8 x 6 inner corners)



```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # cv2.imread gives BGR (mpimg.imread gives RGB!)
ret, corners = cv2.findChessboardCorners(gray, (8, 6), None)
img = cv2.drawChessboardCorners(img, (8, 6), corners, ret) # coloured rows; first corner = (0,0,0)
```

Object points: where the corners really are

[► Lab 6B in Colab](#)

Top-left corner is (0, 0, 0), bottom-right is (7, 5, 0), not (8, 6, 0): counting starts at zero. $Z = 0$ for every point because the board is flat.

```
nx, ny = 8, 6
square = 25.0 # mm, measured on YOUR print

# (48, 3) array: (0,0,0), (1,0,0) ... (7,5,0)
objp = np.zeros((nx * ny, 3), np.float32)
objp[:, :2] = np.mgrid[0:nx, 0:ny].T.reshape(-1, 2)
objp *= square # now in millimetres

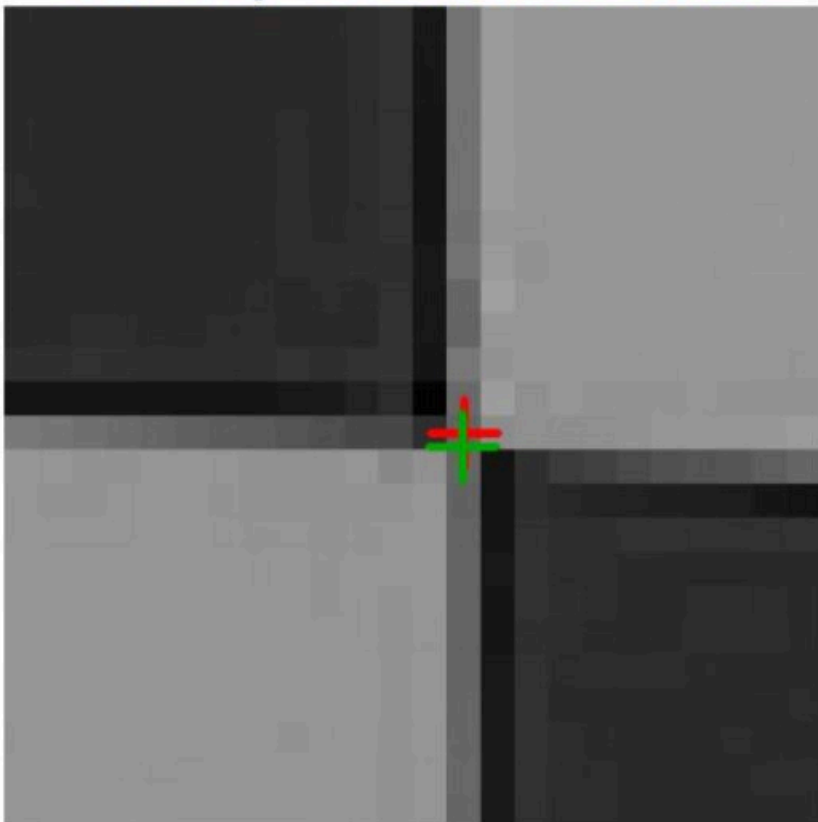
objpoints = [] # 3-D points in the board frame
imgpoints = [] # 2-D points in the image
```

Units of **objp** = units of **tvec**. In **squares** **K** and **dist** are still correct, but distances come out in squares. Multiply by the square size to get mm.

Refining to sub-pixel accuracy

► Lab 6B in Colab

24 x 24 px around one corner



+ red: whole-pixel corner + green: cornerSubPix

The saddle point between four squares is not on a pixel centre. `cornerSubPix` moves each corner to where the image gradients around it are orthogonal to the vector to the corner, typically to 0.1 px.

```
crit = (cv2.TERM_CRITERIA_EPS +  
        cv2.TERM_CRITERIA_MAX_ITER, 30, 1e-3)  
corners = cv2.cornerSubPix(gray, corners,  
                           (11, 11), (-1, -1), crit)
```

```
# OpenCV ≥ 4.3: sector-based detector, more  
# accurate and robust, sub-pixel built in  
ret, corners = cv2.findChessboardCornersSB(  
    gray, (8, 6), cv2.CALIB_CB_EXHAUSTIVE +  
    cv2.CALIB_CB_ACCURACY)
```

Window (11, 11) means a 23×23 search area: keep it smaller than one square in the image.

Calibration, step 1: collect points from every image

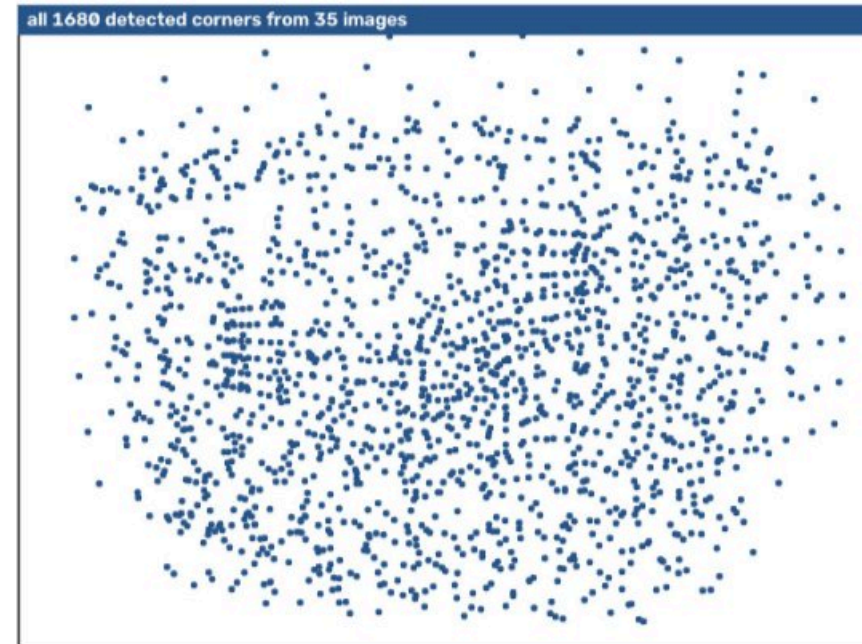
```
import glob, cv2, numpy as np

nx, ny = 8, 6
objp = np.zeros((nx * ny, 3), np.float32)
objp[:, :2] = np.mgrid[0:nx, 0:ny].T.reshape(-1, 2)

objpoints, imgpoints = [], []
images = sorted(glob.glob("calibration_wide/GOPR*.jpg"))

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)
    if ret:                                     # skip images where t
        corners = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1
        objpoints.append(objp)                  # same for every imag
        imgpoints.append(corners)

print(f"board found in {len(imgpoints)} / {len(images)} images")
```



All 1680 corners from the 35 GoPro images: they cover the **whole frame**, corners included. That is what makes a calibration reliable.

Calibration, step 2: `cv2.calibrateCamera`

[► Lab 6C in Colab](#)

```
cv2.calibrateCamera(objpoints,  
imgpoints, gray.shape[::-1], None, None)
```

Return values:
`ret, mtx, dist, rvecs, tvecs`

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(  
    objpoints, imgpoints, gray.shape[::-1], None, None)  
  
print("RMS reprojection error:", ret) # 0.82 px  
print(mtx) # [[560.0  0.0 651.1]  
            # [ 0.0 561.1 498.9]  
            # [ 0.0  0.0  1.0]]  
print(dist) # [[-0.233  0.062  0.000  0.000 -0.008]]
```

Image size: (width, height)!

`gray.shape` is (960, 1280) = (rows, cols). OpenCV wants (1280, 960) = (w, h), hence `gray.shape[::-1]`.

For a colour image use `img.shape[1::-1]`. Passing the full (960, 1280, 3) raises an error.

What comes back

- `ret` : RMS reprojection error (px)
- `mtx` : K (3×3); `dist` : (1×5)
- `rvecs`, `tvecs` : one board pose per image

What happens inside: Zhang's method (2000)

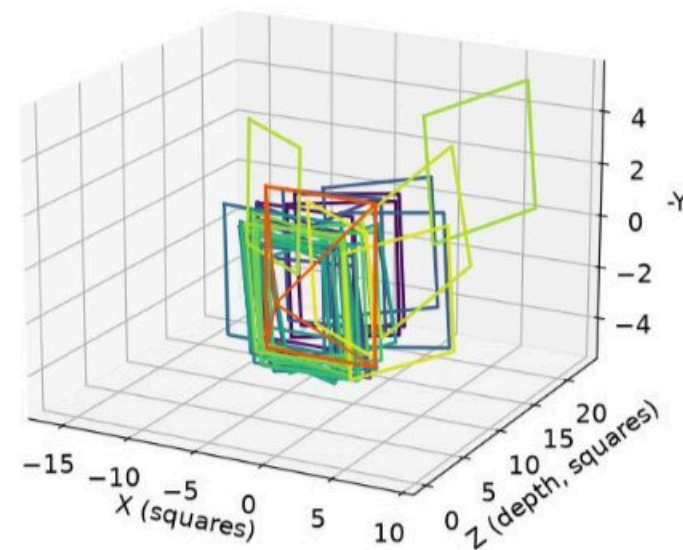
► Lab 6M in Colab

1. For each view, the flat board ($Z = 0$) maps to the image by a **homography**: $\mathbf{x} \simeq H [X \ Y \ 1]^\top$, with $H = K [\mathbf{r}_1 \ \mathbf{r}_2 \ \mathbf{t}]$.
2. Columns $\mathbf{r}_1, \mathbf{r}_2$ of a rotation are **orthonormal**, which gives 2 equations per view on $B = K^{-\top} K^{-1}$:
 $\mathbf{h}_1^\top B \mathbf{h}_2 = 0, \quad \mathbf{h}_1^\top B \mathbf{h}_1 = \mathbf{h}_2^\top B \mathbf{h}_2$
3. B has 5 unknowns (zero skew: 4) $\rightarrow \geq 3$ **views** (≥ 2 with zero skew) give a **closed-form** K ; then each view's R, t .
4. Non-linear **Levenberg–Marquardt** refinement of K , dist and all poses together, minimising the **reprojection error**.

Only a printed board, no expensive 3-D rig, so it is used everywhere: OpenCV, MATLAB, ROS.

In plain words: every tilted photo of the flat board gives two clues about K . Three photos give enough clues to solve K ; then a computer polishes all numbers together.

Extrinsics: where each board was, seen from the camera

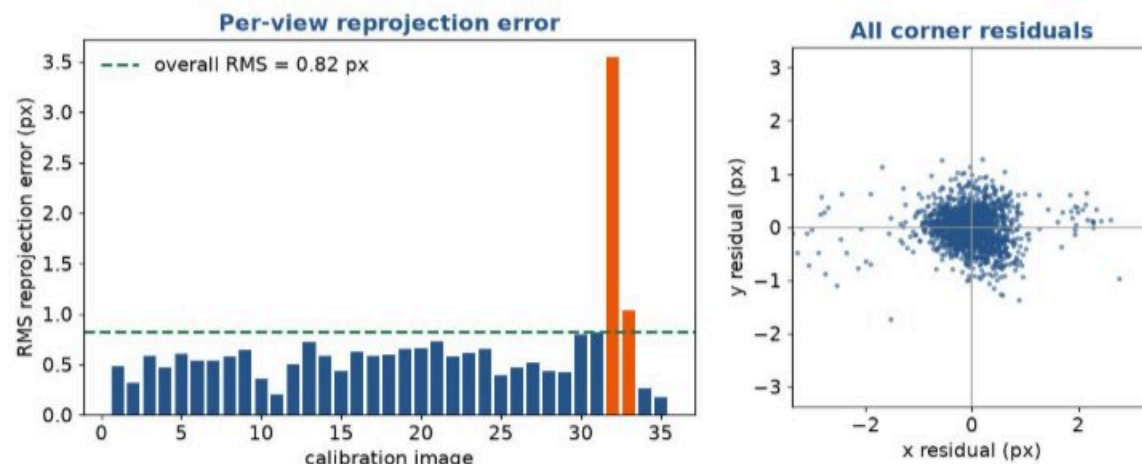


The 35 recovered board poses (rvecs, tvecs) seen from the camera (orange).

Watch: [Camera Calibration using Zhang's Method](#) and [Intrinsic Camera Calibration](#) (Cyrill Stachniss).

Is the calibration any good? Reprojection error

► Lab 6C in Colab



$$E_{RMS} = \sqrt{\frac{1}{N} \sum_i \|\mathbf{x}_i - \hat{\mathbf{x}}_i(K, \text{dist}, R_j, \mathbf{t}_j)\|^2}$$

Project the object points with the estimated model and compare with the detected corners.

- < 0.5 px: good; 0.5–1 px: acceptable; > 1 px: investigate
- One view at 3.5 px (orange): blurred or mis-detected. **Drop it and recalibrate** (Lab 6C).
- Low error $\neq$ correct: 3 nearly identical views can also fit well. Check **coverage**, too.

In plain words: put the board back into the picture using the model and measure how far the predicted corners are from the real ones. Small = the model explains the photos well.

Per-view error and saving the result

[► Lab 6C in Colab](#)

```
errs = []
for objp_i, imgp_i, r, t in zip(objpoints, imgpoints,
                                rvecs, tvecs):
    proj, _ = cv2.projectPoints(objp_i, r, t, mtx, dist)
    d = imgp_i.reshape(-1, 2) - proj.reshape(-1, 2)
    errs.append(np.sqrt((d ** 2).sum(1).mean()))

worst = np.argsort(errs)[::-1][:3]
print("worst views:", [images[i] for i in worst])

# Uncertainty of each intrinsic (std. dev.)
ret, mtx, dist, rv, tv, std_in, std_ex, per_view = \
    cv2.calibrateCameraExtended(objpoints, imgpoints,
                                size, None, None)
print("fx = %.1f ± %.1f px" % (mtx[0, 0], std_in[0, 0]))
```

```
# Save: portable YAML that C++, ROS and Python can read
fs = cv2.FileStorage("camera.yaml", cv2.FILE_STORAGE_WRITE)
fs.write("image_width", size[0]); fs.write("image_height",
fs.write("camera_matrix", mtx)
fs.write("distortion_coefficients", dist)
fs.write("rms", ret)
fs.release()

# Load it back
fs = cv2.FileStorage("camera.yaml", cv2.FILE_STORAGE_READ)
K = fs.getNode("camera_matrix").mat()
D = fs.getNode("distortion_coefficients").mat()

# or simply
np.savez("camera.npz", K=mtx, dist=dist)
# (Udacity used pickle: wide_dist_pickle.p)
```

How to take good calibration images

Coverage & variety

- **15–30 images** (Udacity: ≥ 20)
- Board in every **part of the frame**, especially the **corners**, where distortion is largest
- **Tilt** the board $\pm 30\text{--}45^\circ$ in both axes. Frontal-only views cannot separate f from distance
- Vary the **distance**: near (board fills the frame) and far

Quality

- Sharp: **still** board, good light, short exposure
- Same **resolution, focus and zoom** you will use later: K changes with all three
- Do not crop or resize the images before calibration
- Check the **per-view error** and coverage, re-shoot the gaps

The ROS calibrator turns this advice into four progress bars, **X, Y, Size and Skew**, and only enables CALIBRATE once your views cover enough of each. Try it on the next slide.

Live demo: collect a good calibration set interactive

[▶ Lab 6I in Colab](#)

board x 0
board y 0
distance 1.00
tilt left/right 0
tilt up/down 0

X 0%
Y 0%
Size 0%
Skew 0%

0 samples · **CALIBRATE (not yet)**

Bars follow ROS `camera_calibration`: the spread of the board's centre (X, Y), apparent size and skew over the accepted samples. Samples too similar to an earlier one are rejected.

The same idea in ROS 2: camera_calibration

```
# install (ROS 2 Humble / Jazzy / Rolling)
sudo apt install ros-$ROS_DISTRO-camera-calibration

# monocular: 8x6 inner corners, 25 mm squares
ros2 run camera_calibration cameracalibrator \
  --size 8x6 --square 0.025 \
  --ros-args -r image:=/camera/image_raw \
            -p camera:=/camera

# stereo pair
ros2 run camera_calibration cameracalibrator \
  --size 8x6 --square 0.025 --approximate 0.1 \
  --ros-args -r left:=/left/image_raw -r right:=/right/image_raw

# fisheye lens:  --camera-model fisheye
# ChArUco board: --pattern charuco --charuco_marker_size 0.019 --a
```

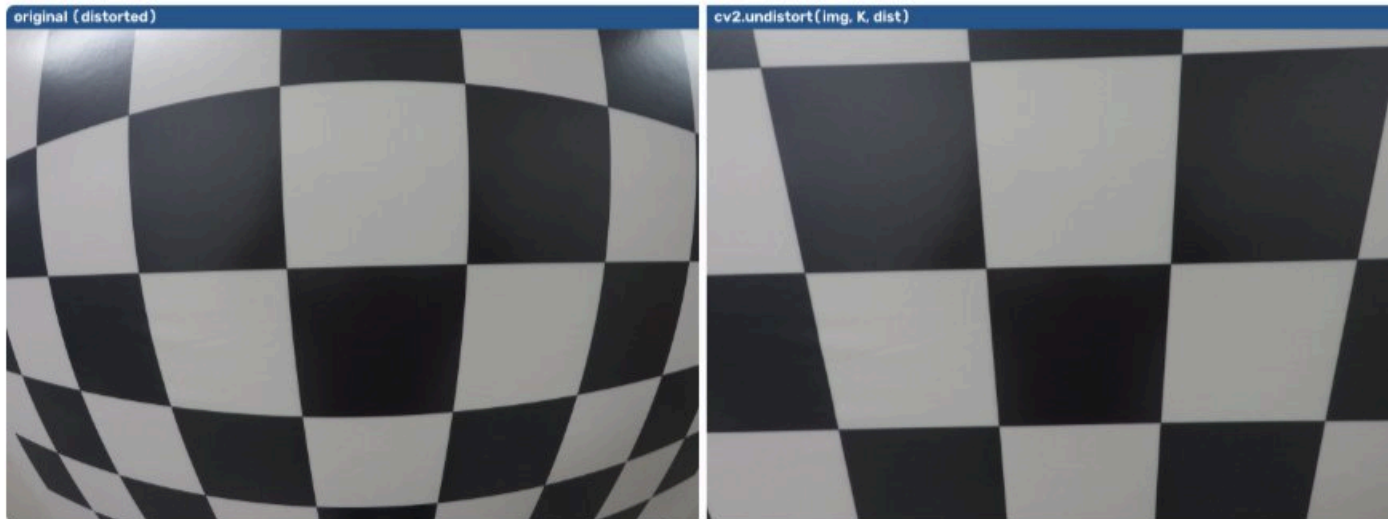
How it works

- Uses OpenCV under the hood: the Python module `camera_calibration.calibrator` (`MonoCalibrator`, `StereoCalibrator`)
- Move the board until the X / Y / Size / Skew bars go **green** → **CALIBRATE** → **SAVE / COMMIT**
- Writes a `camera_info` YAML (K, D, R, P), loaded by `camera_info_manager` and used by `image_proc` to publish `image_rect`
- Models: `pinhole` (plumb-bob) and `fisheye` (equidistant)

[docs.ros.org: camera_calibration](https://docs.ros.org/en/latest/camera_calibration.html)

Calibration, step 3: undistort

► Lab 6C in Colab



```
dst = cv2.undistort(img, mtx,  
dist, None, mtx)
```

- Takes a distorted image, K and dist; returns the corrected **destination** image
- The last argument is the **new** camera matrix; passing `mtx` keeps the same focal length
- For every output pixel OpenCV computes where it came from (the distortion model) and interpolates

Straight lines on the board are straight again: the calibration worked.

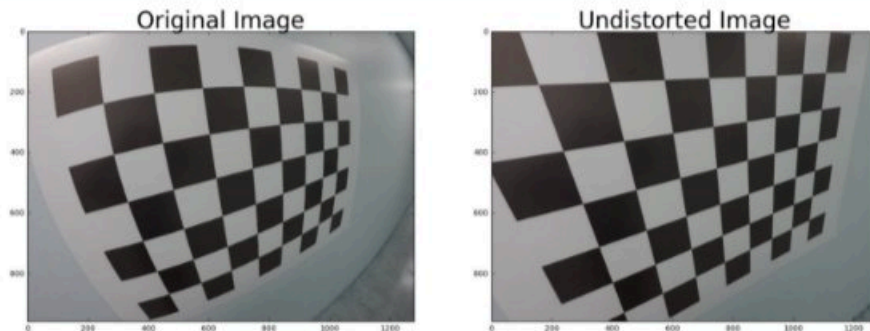
Quiz: write `cal_undistort()`

[► Lab 6C in Colab](#)

```
import pickle, cv2, numpy as np

# Saved objpoints and imgpoints (Udacity wide_dist_pickle.
dist_pickle = pickle.load(open("wide_dist_pickle.p", "rb"))
objpoints = dist_pickle["objpoints"]
imgpoints = dist_pickle["imgpoints"]
img = cv2.imread("test_image.png")

# TODO: take an image, object points and image points,
# calibrate, undistort and return the undistorted image
def cal_undistort(img, objpoints, imgpoints):
    undist = np.copy(img)          # delete this line
    return undist
```

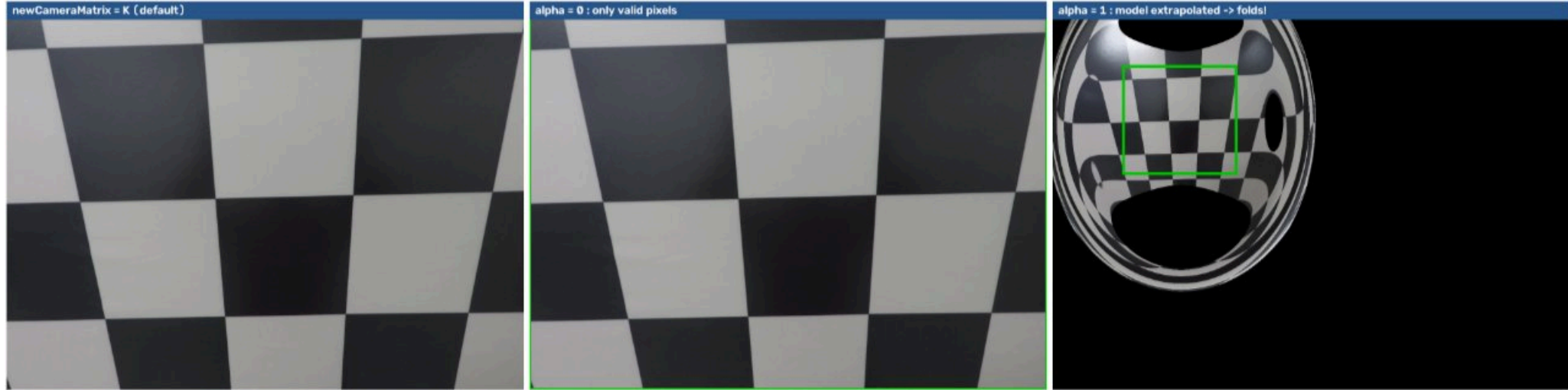


One solution

```
def cal_undistort(img, objpoints, imgpoints):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(
        objpoints, imgpoints, gray.shape[::-1], None, None)
    return cv2.undistort(img, mtx, dist, None, mtx)
```

In practice calibrate **once**, save `mtx` and `dist`, and only call `undistort` per frame. Lab 6C has an auto-check.

Keep or crop the borders? `getOptimalNewCameraMatrix`



```
h, w = img.shape[:2]
newK, roi = cv2.getOptimalNewCameraMatrix(
    mtx, dist, (w, h), alpha=0)  # 0 ... 1
und = cv2.undistort(img, mtx, dist, None, newK)
x, y, rw, rh = roi              # green box
und = und[y:y + rh, x:x + rw]  # crop black edges
```

- **alpha = 0**: zoom in until **every** output pixel is valid (loses a little FOV)
- **alpha = 1**: keep **all** source pixels (black curved borders)
- Here alpha = 1 **folds**: the 5-coefficient polynomial is only valid where there were corners. Try **CALIB_RATIONAL_MODEL** or fisheye for wide lenses (Lab 6C).

► Lab 6C in Colab

Undistorting video fast: compute the map once

[► Lab 6C in Colab](#)

```
# once, at start-up
newK, roi = cv2.getOptimalNewCameraMatrix(K, dist, (w, h), 0)
map1, map2 = cv2.initUndistortRectifyMap(
    K, dist, None, newK, (w, h), cv2.CV_16SC2)

cap = cv2.VideoCapture(0)          # or Picamera2 frames
while True:
    ok, frame = cap.read()
    if not ok: break
    und = cv2.remap(frame, map1, map2, cv2.INTER_LINEAR)
    cv2.imshow("undistorted", und)
    if cv2.waitKey(1) == 27: break

# undistort only some points (e.g. detected corners, lane pixels)
pts_ud = cv2.undistortPoints(pts, K, dist, P=newK)
```

Why it is faster

`cv2.undistort` recomputes the distortion model for every pixel on every call.

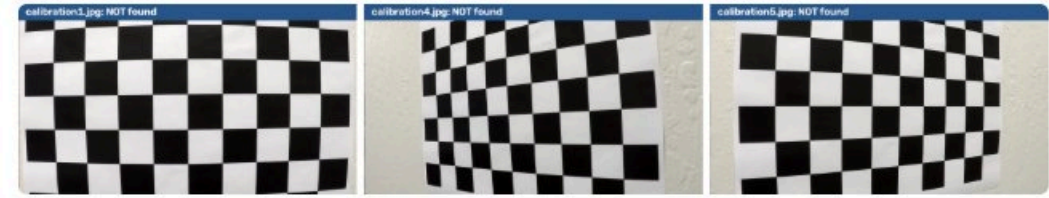
`initUndistortRectifyMap` stores the source coordinate of each output pixel **once**; `remap` is then a pure lookup: several times faster, which matters at 30 fps on a Raspberry Pi.

If you only need a few points, `undistortPoints` avoids warping the image at all.

Common calibration pitfalls

[► Lab 6F in Colab](#)

Symptom	Likely cause
<code>ret</code> is <code>False</code> for many images	wrong (nx, ny): squares instead of inner corners; board cut off by the frame; glare
<code>calibrateCamera</code> error about sizes	passed <code>img.shape</code> with 3 channels, or images of different sizes (two Udacity images are 1281×721 !)
RMS > 1 px	blur, a bent board, a mis-detected view: check per-view errors
Edges still curved / folded after undistort	no corners near the image edges; lens too wide for the model $\rightarrow$ k3, rational, or fisheye
Good RMS, wrong distances	wrong <code>square_size</code> ; calibrated at another resolution or focus
Image upside-down / mirrored axes	symmetric board (8×8): use $nx \neq ny$



3 of the 20 road-camera images fail: part of the board is **outside the frame**. Fine for a chessboard: they are skipped. A ChArUco board would still use them.

Lab 6F is a bug hunt with these pitfalls planted in the code.

Calibrating with OpenCV: the 3 things to remember

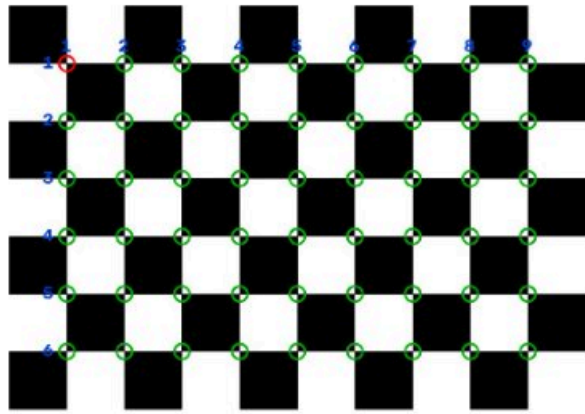
1. Corners: `findChessboardCorners` + `cornerSubPix`, count the **inner** corners.
2. `calibrateCamera(objpoints, imgpoints, (w, h))` → K, dist and one pose per photo.
3. Judge it with the **RMS reprojection error** (< 0.5 px good, > 1 px re-shoot), then `undistort` / `remap`.

5

Hands-on: calibrate the webcam, live

Level 5 · real OpenCV running in the browser, a printed chessboard, 10 minutes

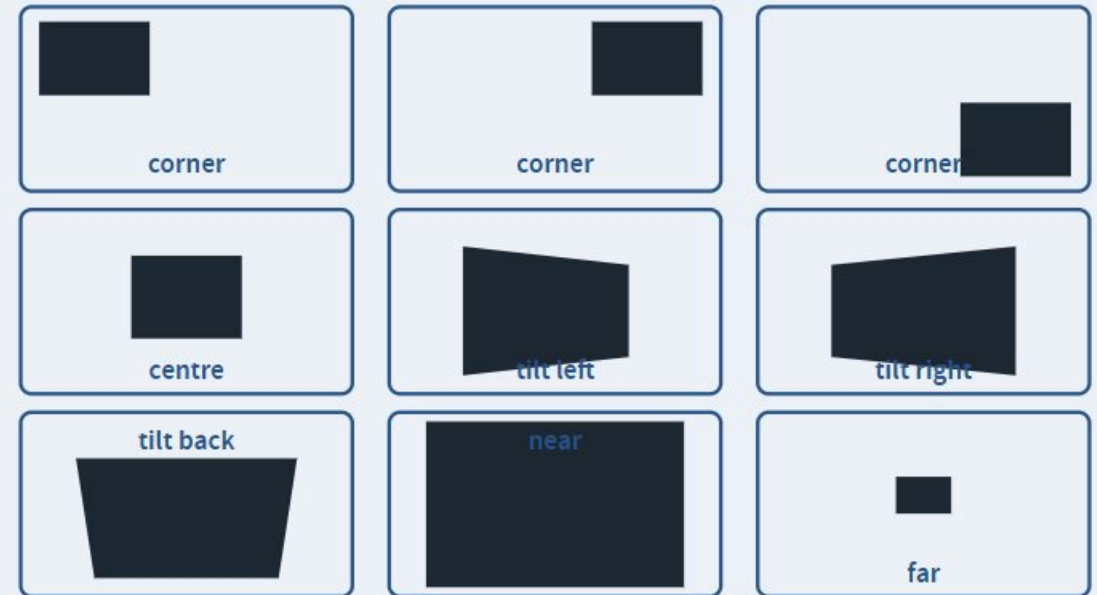
What you need (and how to hold the board)



1. **Print [pattern.png](#)** on A4 at **100 %** (no "fit to page") or **show it full-screen on a phone**
2. Glue it to stiff card. It must stay **flat**
3. **Measure** one square with a ruler (e.g. 24.5 mm)
4. A laptop webcam, Chrome or Edge, good light

It has 10×7 squares but 9×6 inner corners (red = corner number 1). We always count the inner corners.

The "calibration dance": 15–20 views



Hold it **still** for each shot. Cover the **edges and corners**, where the lens bends most.

The live demo in 7 steps

#	Teacher does	Students see	The OpenCV behind it
1	Open the studio, click Start webcam , allow the camera	the live picture	<code>VideoCapture</code> / <code>getUserMedia</code>
2	Type the board: 9 × 6 corners, measured square size	–	<code>objp = np.mgrid[...] * square</code>
3	Hold the board up	coloured corners: found!	<code>cv2.findChessboardCorners</code>
4	Turn Auto on, do the calibration dance	views appear; X/Y/Size/Skew bars fill	<code>cv2.cornerSubPix</code> , collect points
5	Click Calibrate	K, dist, RMS, the error of each view	<code>cv2.calibrateCamera</code>
6	Tab ③ Undistort , split view + grid	raw vs corrected, live	<code>initUndistortRectifyMap</code> + <code>remap</code>
7	Tab ④ Measure ; check with a tape measure	"43.2 cm away"	<code>cv2.solvePnP</code>

Tip: let two or three students take turns holding the board. Everyone watches the bars fill and learns why varied views matter.

LIVE: calibrate this webcam now live

[Open full screen](#)

 **Webcam Calibration Studio** [Lecture 6](#) [Print the pattern \(9 × 6\)](#) [Show the pattern on a phone](#)

Loading OpenCV (cv2) in your browser...

① Detect & capture

③ Undistort (live)

④ Measure distance (live)

☒ mirror view ☐ straight-line grid

camera off

1 · Start the camera

Allow the browser to use your webcam. Hold the printed 9 × 6 chessboard (or a phone showing it) in front of it.

camera default resolution 640 × 480

▶ Start webcam

No webcam? Use 20 demo images

2 · Tell the computer about your board

Inner corners (where 4 squares meet): the course pattern has 9 × 6. Measure one square with a ruler.

corners across 9 down 6 square size (mm) 25

3 · Capture 15–20 good views

Green corners = board found. **Auto** takes a picture when the board is **still** and in a **new** place. Move it to the edges and corners, tilt it, near and far.

 Capture now (space)

Auto: off

Clear all

X 0%
Y 0%
Size 0%
Skew 0%

0 views captured.

4 · Calibrate

Runs `cv2.calibrateCamera` on your views (needs at least 8; 15+ is better).

 Calibrate

☐ rational model (very wide lenses)

5 · Use it

③ **Undistort**: see the corrected live image. ④ **Measure**: hold the board up and read its

What just happened? The code you ran

```
import cv2, numpy as np
nx, ny, square = 9, 6, 24.5          # step 2: the board
objp = np.zeros((nx * ny, 3), np.float32)
objp[:, :2] = np.mgrid[0:nx, 0:ny].T.reshape(-1, 2) * square

objpoints, imgpoints = [], []
cap = cv2.VideoCapture(0)             # step 1: the webcam
while len(imgpoints) < 15:
    ok, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    found, corners = cv2.findChessboardCorners(gray, (nx, ny)) # step 3
    if found and board_is_still_and_new(corners):              # step 4 (auto)
        corners = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), crit)
        objpoints.append(objp); imgpoints.append(corners)

rms, K, dist, rvecs, tvecs = cv2.calibrateCamera(               # step 5
    objpoints, imgpoints, gray.shape[:: -1], None, None)

mapx, mapy = cv2.initUndistortRectifyMap(K, dist, None, K, (w, h), cv2.CV_32FC1)
fixed = cv2.remap(frame, mapx, mapy, cv2.INTER_LINEAR)         # step 6

ok, rvec, tvec = cv2.solvePnP(objp, corners, K, dist)           # step 7
print("distance:", np.linalg.norm(tvec) / 10, "cm")
```

The studio runs exactly these OpenCV-Python calls: real `cv2` 4.10, compiled to WebAssembly and running inside your browser. The video never leaves your laptop.

The same program as a desktop app, which works offline:

```
pip install opencv-python numpy
python webcam_calibration_studio.py --board 9x6 --mm 24.5
# SPACE capture · A auto · C calibrate · U undistort · M mea
```

[webcam_calibration_studio.py](#) in LAB/Lab_6

Reading your result: is it good?

A typical laptop webcam at 640×480

```
RMS = 0.31 px
K = [[612.4,    0, 318.9],
     [    0, 611.8, 243.6],
     [    0,    0,    1]]
dist = [0.04, -0.16, 0.001, -0.002, 0.09]
```

- **RMS 0.31 px:** on average the model is off by a third of a pixel. Excellent.
- $f_x \approx f_y \approx 612$: square pixels ✓. Field of view = $2 \cdot \text{atan}(320 / 612) \approx 55^\circ$.
- $(c_x, c_y) \approx (319, 244)$: close to the centre (320, 240) ✓.
- **k1 small:** webcams have little distortion. Wide cameras have $k1 \approx -0.2$ to -0.4 .

You see	Means	Do
RMS < 0.5 px	excellent	save camera.yaml
RMS 0.5–1 px	usable	delete the red (worst) views, recalibrate
RMS > 1 px	bad views	blur? bent board? re-shoot, hold still
f_x and f_y differ by > 2 %	too few tilted views	add tilted and near views
c_x, c_y far from the centre	edges not covered	put the board in the corners
distance wrong by a fixed %	wrong square size	measure the square again

Remember: a small RMS only says the model fits your photos. The ruler test (next slide) says whether it fits the world.

Three ways to test a calibration

① Straight-line test

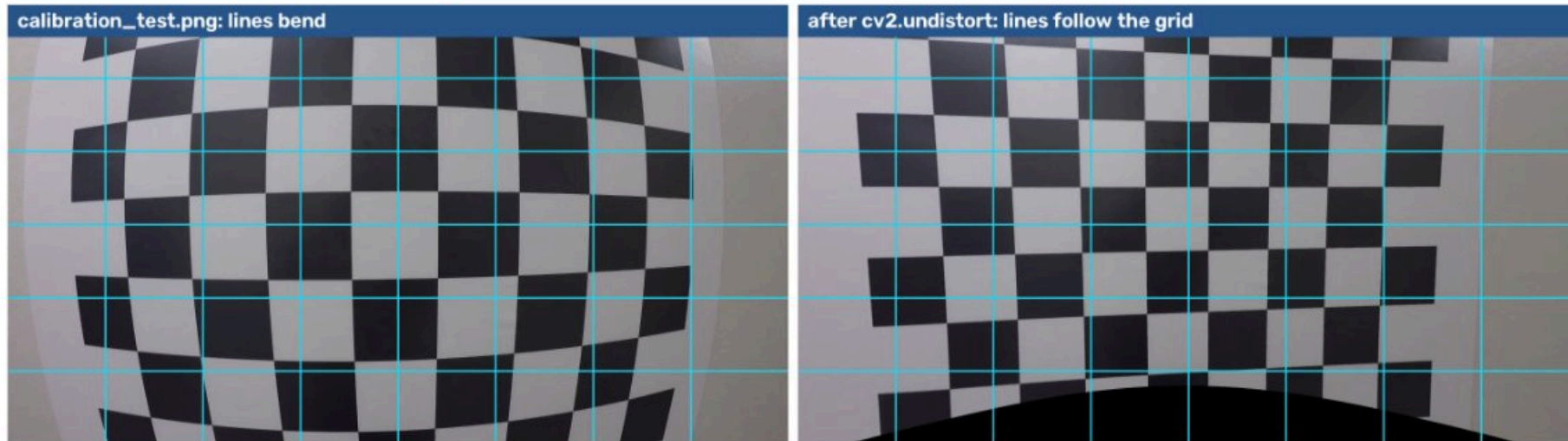
Undistort tab, **split view + grid**. Hold a ruler or door frame near the image edge: in the corrected half it must follow the grid lines.

② Ruler test

Measure tab. Put the board **40 cm** from the webcam (tape measure). The studio should say 40 ± 1 cm. Move to 60 cm, then 80 cm.

③ Test image

Click **Try the test image:** `calibration_test.png` from a GoPro, straightened with that GoPro's calibration (Lab 6C).



If the live demo goes wrong

Problem	Fix
No camera prompt / "NotAllowedError"	click the camera icon in the address bar → Allow; or Open full screen (own tab)
"NotReadableError" / black picture	close Zoom, Teams, the Windows Camera app: only one program can use the webcam
"board not found" all the time	show the whole board with a white margin; check 9×6 ; more light; avoid reflections on glossy paper or a phone
Corners found but flicker	hold still; webcams auto-expose; move closer to a window or lamp
Auto never captures	the board must be still and in a new place: move to a new corner or tilt
OpenCV does not load (no internet)	run <code>python webcam_calibration_studio.py</code> (offline), or use the 20 demo images
RMS high	delete the red views (click their bars), recalibrate

Live calibration: the 3 things to remember

1. A printed, **flat** 9×6 board + a **measured** square size + 15–20 **varied, still** views.
2. `findChessboardCorners` → `calibrateCamera` → check **RMS** and delete bad views.
3. Test it in the world: **straight lines** stay straight, the **ruler** distance comes out right.

[Open the studio](#)[Lab 6H: calibrate your own camera \(notebook\)](#)[Lab 6I: Raspberry Pi version](#)

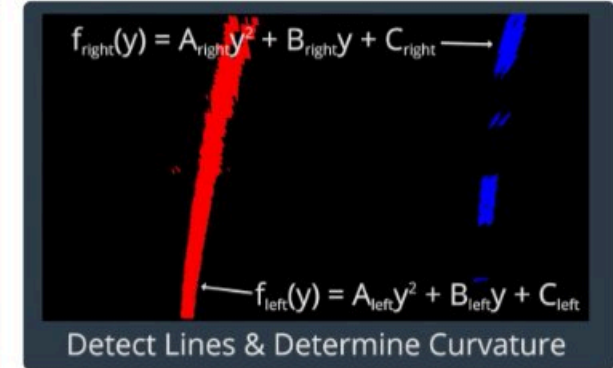
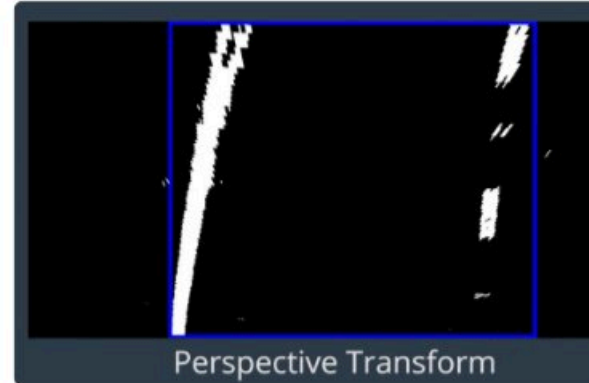
6

Perspective transform & lane curvature

Level 6 · Use your calibration: straighten, warp and measure the road

Why we need all this: lane curvature

► Lab 6D in Colab



A self-driving car needs a **steering angle**, which comes from the car's speed and dynamics and **how much the lane curves**:

1. threshold the lane pixels (Lecture 5)
2. **perspective transform** → bird's-eye view
3. fit a polynomial to each line
4. compute the curvature

For a near-vertical lane line

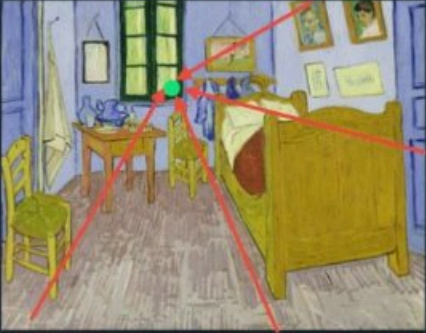
$$f(y) = Ay^2 + By + C$$

A → curvature · **B** → heading (direction) · **C** → position (offset from the left of the image)

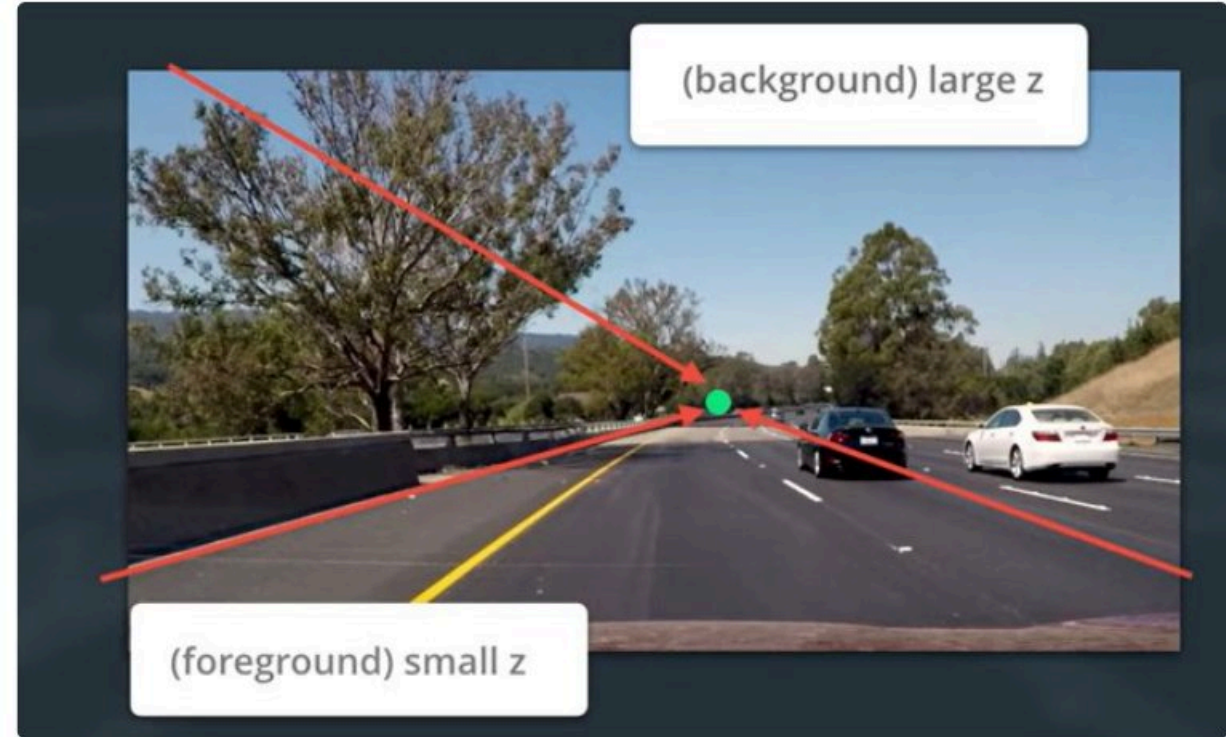
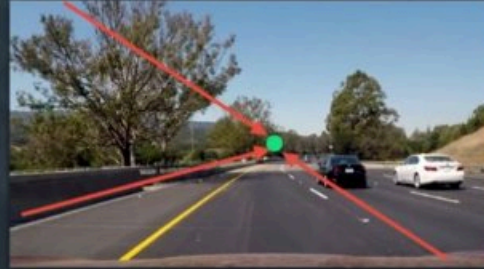
$$R_{curve} = \frac{(1 + (2Ay + B)^2)^{3/2}}{|2A|}$$

Perspective: far things look small, parallel lines meet

Van Gogh's "Bedroom in Arles"



Camera image



The greater an object's Z (distance from the camera), the smaller it appears: the $1/Z$ of the pinhole model. Artists use the same rule. Lane lines that are **parallel** on the road **converge** to a vanishing point in the image.

A perspective transform changes the viewpoint



4 points on a plane in the image ...



... mapped to 4 points of a rectangle

A perspective transform maps the points of a given image to **different, desired** image points, as if seen from a new viewpoint. It effectively drags points towards or away from the camera. **Four point pairs** are enough to define it (a 3×3 **homography** with 8 degrees of freedom).

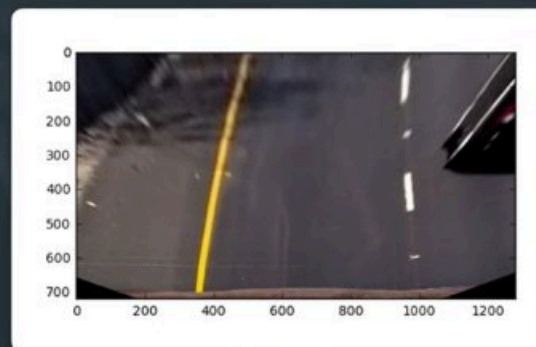
In plain words: four corners of a flat thing are enough to tell OpenCV how to "walk around" it and photograph it from the front or from above.

The bird's-eye view of a road

Perspective transform



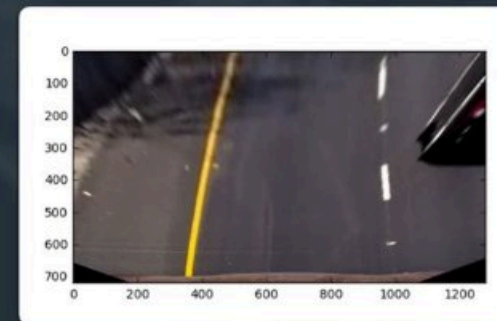
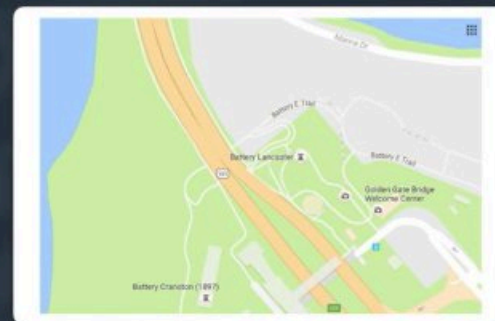
NORMAL VIEW



BIRD'S-EYE VIEW

Maps

BIRD'S-EYE VIEW EXAMPLES



- From the car's camera, the left line seems to lean and curve to the right, the right line to lean left without much curve.
- From above, both lines are **parallel**, a **consistent width** apart, and curve **by the same amount**.
- A top-down view also lines up directly with a **map**.

Quiz: why a bird's-eye view?

QUIZ QUESTION

For images from a front-facing car camera, why do we want to perform a perspective transform?

- a. Because it is more straightforward to take a gradient on a top-down view.
- b. Because ultimately we want to measure the **curvature** of the lines, and to do that we need to transform to a top-down view.
- c. Because transforming the perspective removes the effect of curvature from our measurement.
- d. Because it removes the effect of shadows and colour differences.

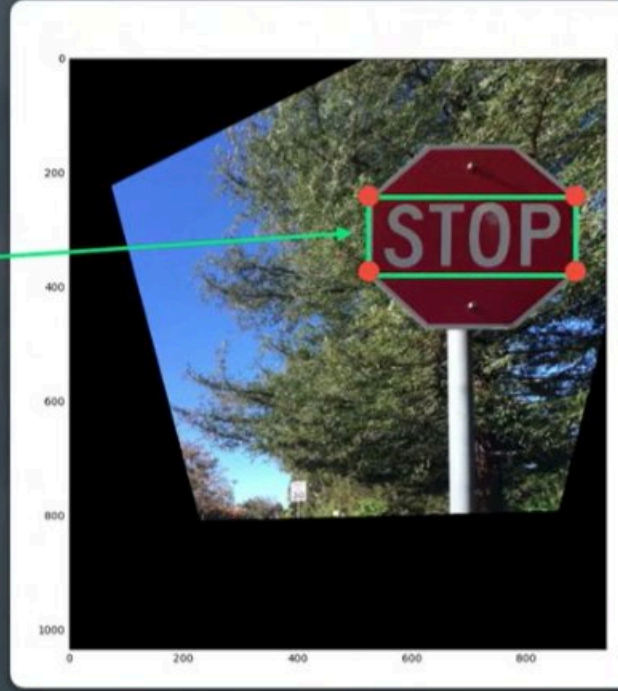
B. Curvature is a property of the road seen from above. In the camera view perspective bends and squeezes it.

Example: transform a stop sign

ORIGINAL IMAGE



"WARPED" IMAGE



1. Pick **4 source points** that define a rectangle on a **plane** in the image (the face of the sign)
2. Pick **4 destination points**: where they should appear in the **warped** image
3. `getPerspectiveTransform(src, dst) → M`
4. `warpPerspective(img, M, size) → the warped image`

A traffic sign is a good test because **text** shows clearly whether the transform is right.

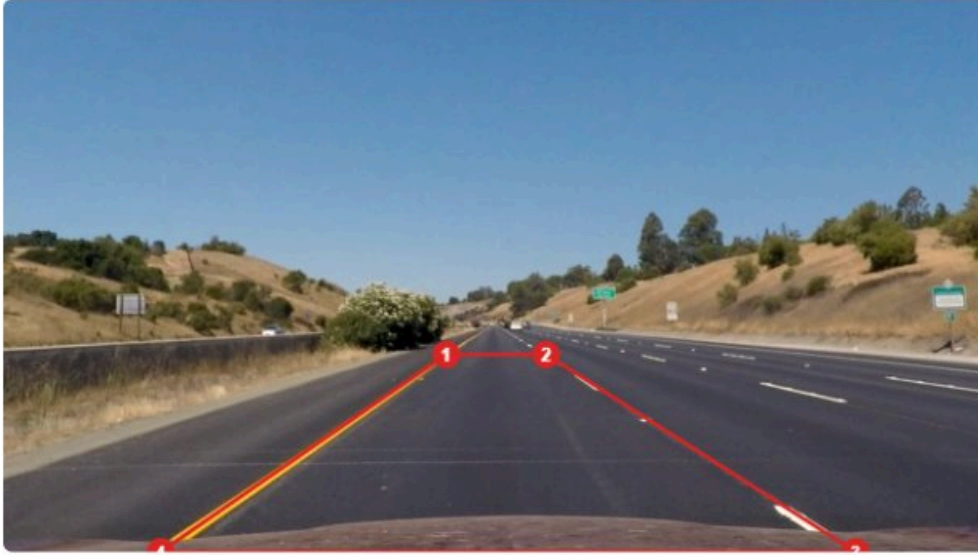
Perspective transform: the code

```
def warp(img):  
    img_size = (img.shape[1], img.shape[0])      # (w, h)  
    # four source points on the sign (read off by hovering  
    # the mouse over the image), clockwise from top-right  
    src = np.float32([[850, 320], [865, 450],  
                      [533, 350], [535, 210]])  
    # where we want them: a front-facing rectangle  
    dst = np.float32([[870, 240], [870, 370],  
                      [520, 370], [520, 240]])  
    M = cv2.getPerspectiveTransform(src, dst)  
    Minv = cv2.getPerspectiveTransform(dst, src)  # to un-warp  
    warped = cv2.warpPerspective(img, M, img_size,  
                                  flags=cv2.INTER_LINEAR)  
    return warped
```



Choosing points **by hand** is fine for a fixed car camera (do it once). Many systems pick them **automatically** from corners, edges or detected objects, as you will with chessboard corners in the next quiz.

Live demo: drag four points → bird's-eye view interactive

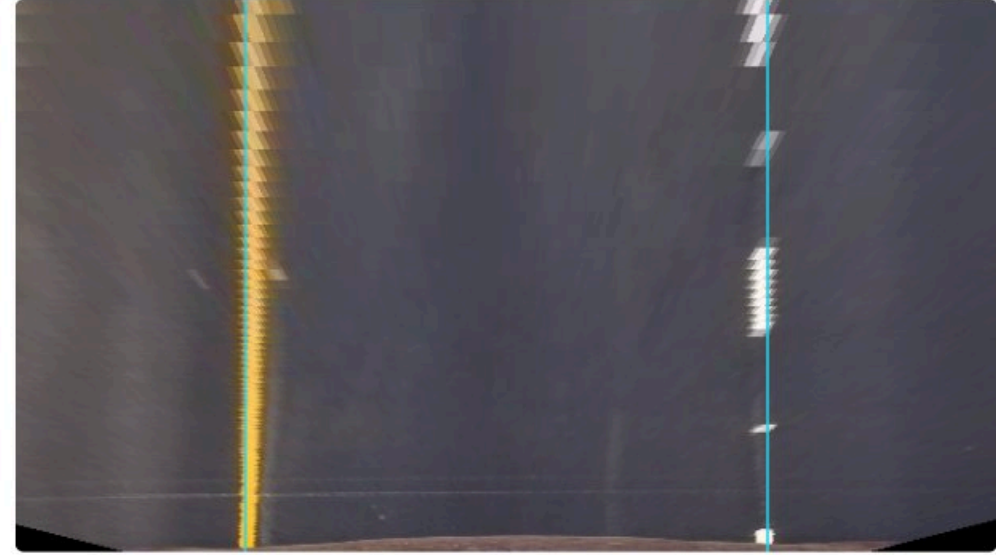


Drag the 4 red handles onto the lane lines (undistorted frame, Lab 6D camera).

recommended points

a bad choice

toggle grid



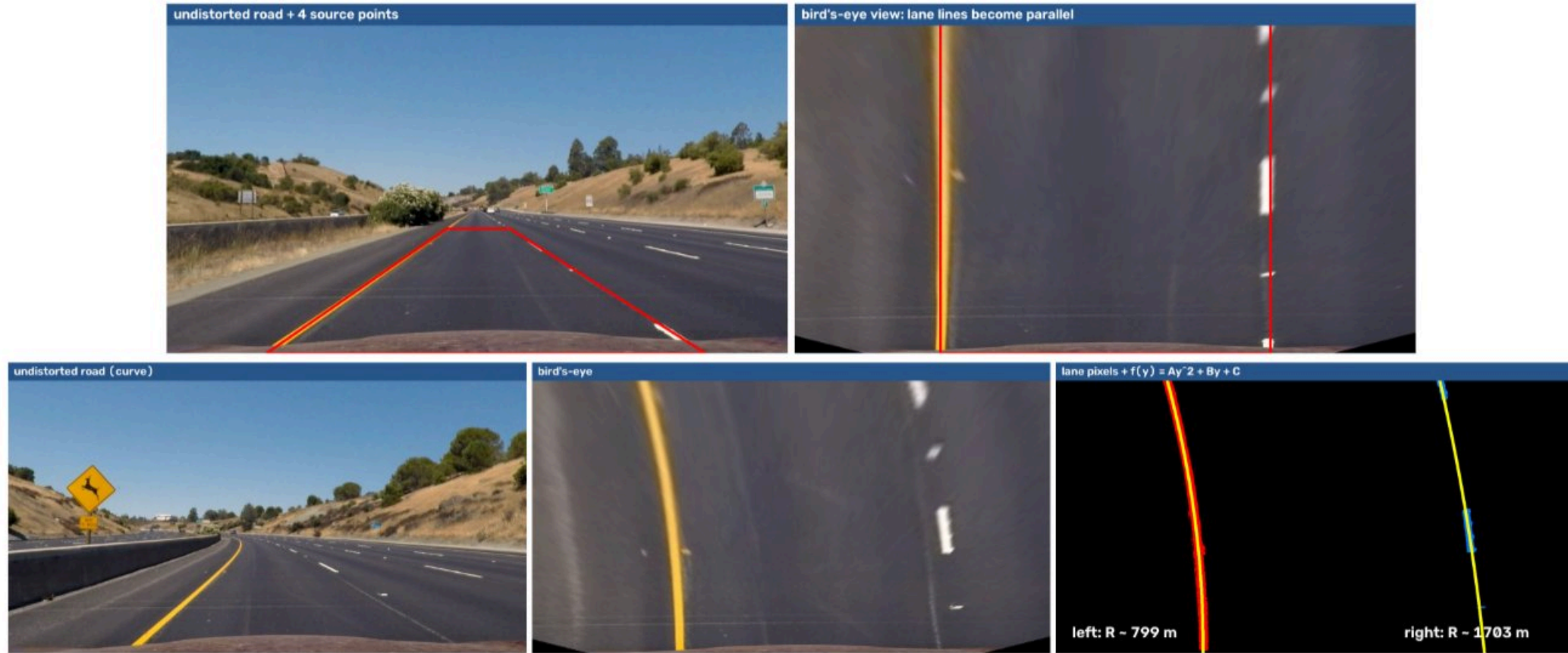
Warped: good points make the lane lines vertical and parallel.

```
src = [[577,463],[707,463],[1110,720],[205,720]]  
M = [[ -0.537,  -1.492,  969.782],  
      [ -0.000,  -2.002,  927.096],  
      [ -0.000,  -0.002,   1.000]]
```

► Lab 6D in Colab

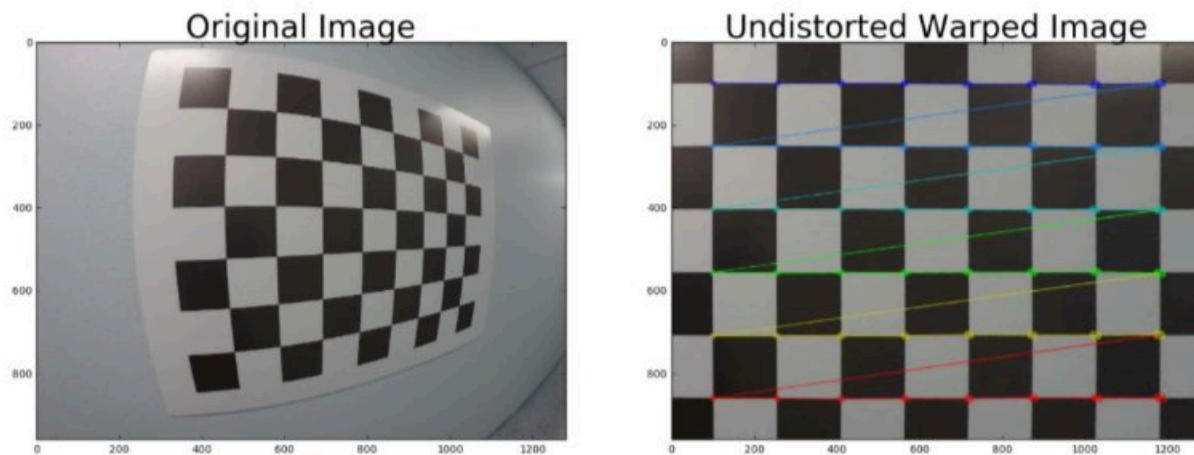
Undistort → warp → fit: on the real road camera

► Lab 6D in Colab



Top: $\text{src} = [[577, 463], [707, 463], [1110, 720], [205, 720]] \rightarrow \text{dst} = [[300, 0], [980, 0], [980, 720], [300, 720]]$. Bottom: lane pixels in the bird's-eye view, second-order fits and the radius in metres (assuming 30 m per 720 px vertically and 3.7 m per 700 px horizontally; Lab 6D).

Quiz: undistort and transform a chessboard

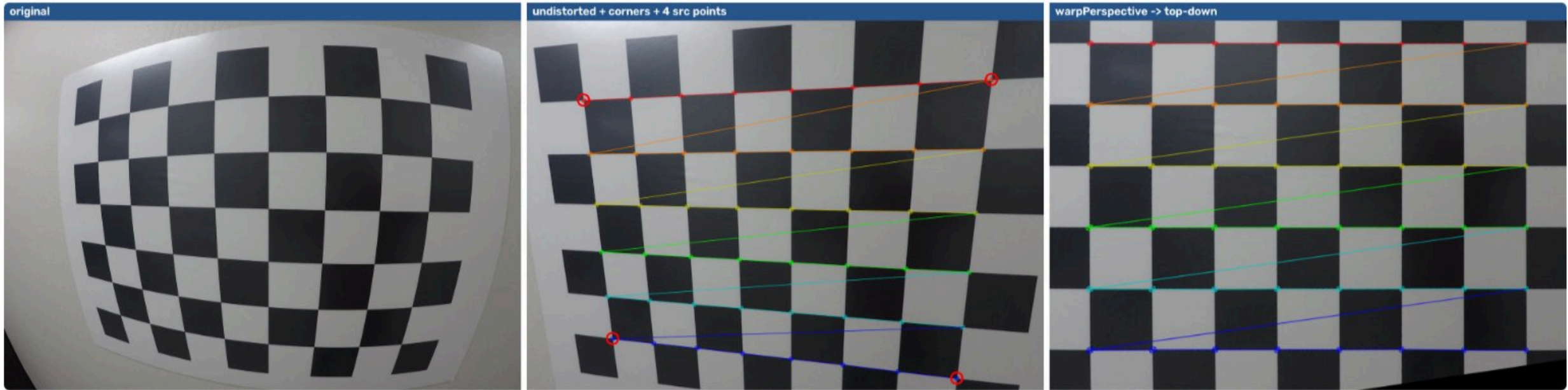


Given `mtx` and `dist`, write `corners_unwarp(img, nx, ny, mtx, dist)`:

1. Undistort with `cv2.undistort`
2. Convert to grayscale and find the chessboard corners; draw them
3. `src` = the **outer 4 detected corners**
4. `dst` = 4 points of a rectangle, in the **same order** as `src`
5. `getPerspectiveTransform` $\rightarrow$ `M`; `warpPerspective` $\rightarrow$ top-down view

```
def corners_unwarp(img, nx, ny, mtx, dist):  
    undist = cv2.undistort(img, mtx, dist, None, mtx)  
    gray = cv2.cvtColor(undist, cv2.COLOR_BGR2GRAY)  
    ret, corners = cv2.findChessboardCorners(gray, (nx, ny),  
    M, warped = None, undist  
    if ret:  
        corners = corners.reshape(-1, 2)  # (N,1,2) in 4.x,  
        cv2.drawChessboardCorners(undist, (nx, ny), corners,  
        offset = 100  
        w, h = gray.shape[1], gray.shape[0]  
        src = np.float32([corners[0], corners[nx - 1],  
                           corners[-1], corners[-nx]])  
        dst = np.float32([[offset, offset], [w - offset, off  
                           [w - offset, h - offset], [offset,  
        M = cv2.getPerspectiveTransform(src, dst)  
        warped = cv2.warpPerspective(undist, M, (w, h))  
    return warped, M
```

Undistort & transform: the result



Our solution on a GoPro image from Lab 6. Note the order: **undistort first**. A homography maps straight lines to straight lines, so it cannot remove lens curvature.

OpenCV 5 note: `findChessboardCorners` returns corners of shape $(N, 2)$; OpenCV 4.x returned $(N, 1, 2)$. Code like `corners[0][0]` breaks. Use `corners.reshape(-1, 2)` to work with both.

Quiz: match each process to its purpose

MATCHING CHALLENGE

Processes: ① Camera calibration · ② Distortion correction · ③ Perspective transform

Purposes:

- a. To transform an image so that we are effectively viewing objects from a different angle or direction.
- b. To ensure that the geometrical shape of objects is represented consistently, no matter where they appear in an image.
- c. To compute the transformation between 3-D object points in the world and 2-D image points.
- d. To measure the orientation angles of objects in an image. (distractor)

① **Camera calibration** → C: computes the mapping between 3-D world points and 2-D image points.

② **Distortion correction** → B: shapes look the same anywhere in the frame.

③ **Perspective transform** → A: view the scene from a different angle, e.g. from above.

D is not a purpose of any of the three on its own.

Perspective transform: the 3 things to remember

1. **Undistort first**, then warp: a homography keeps straight lines straight, so it cannot fix lens curves.
2. 4 point pairs → `getPerspectiveTransform` → `warpPerspective`.
3. Bird's-eye view → fit $x = Ay^2 + By + C$ → lane curvature in metres.

7

Advanced: more cameras, better targets

Level 7 · advanced · camera types · ChArUco · fisheye · stereo · Raspberry Pi · tools

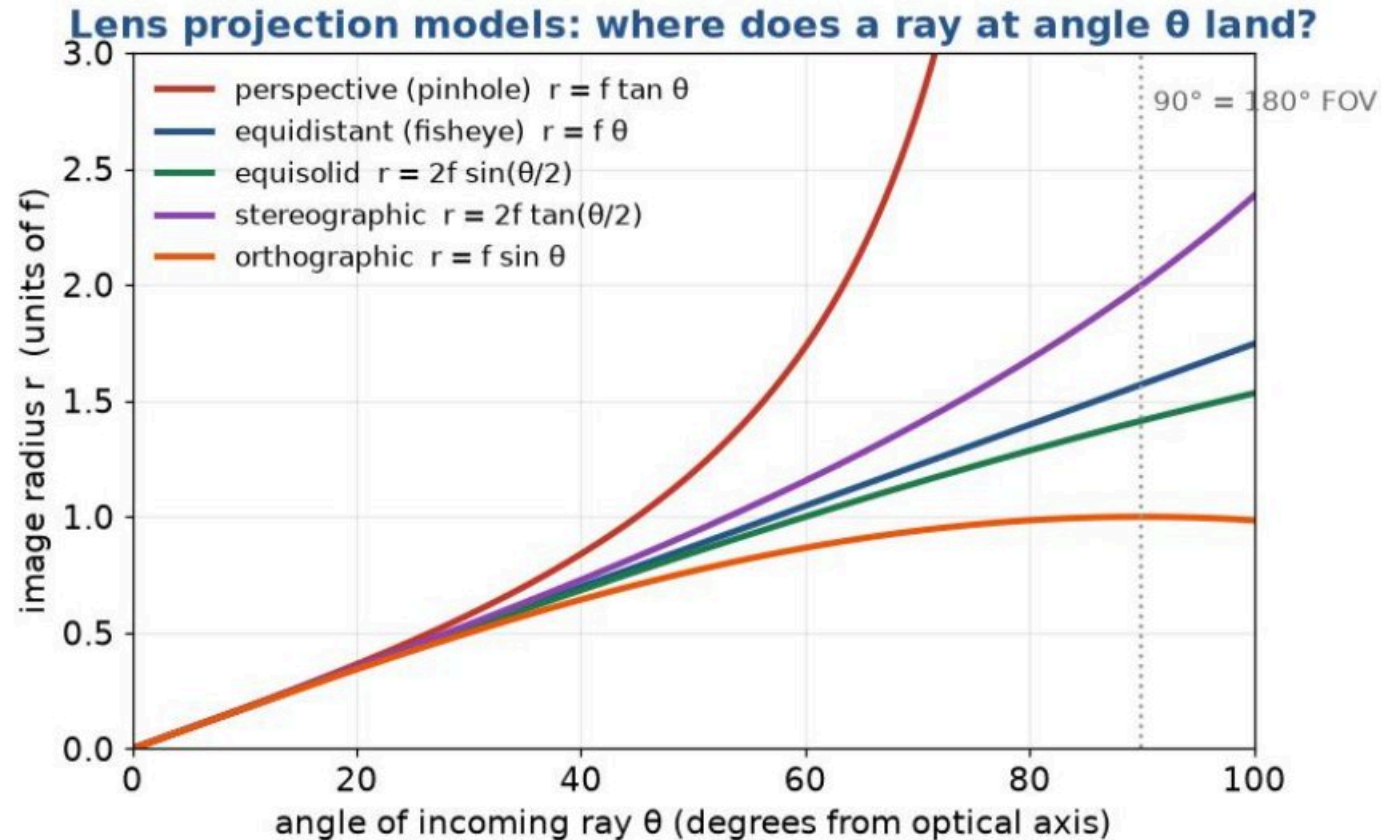
Camera types you will meet

Camera	Typical FOV	Model to calibrate with	OpenCV	Examples
Standard / narrow lens	40–70°	pinhole + radial/tangential (k_1, k_2, p_1, p_2)	<code>cv2.calibrateCamera</code>	Pi Cam v2/3, webcams, phone main camera
Wide-angle	90–120°	pinhole + k_3 , or <code>CALIB_RATIONAL_MODEL</code> (k_4 – k_6)	<code>cv2.calibrateCamera</code>	Pi Cam 3 Wide, GoPro (linear mode), dash-cams
Fisheye	150–200°	Kannala–Brandt equidistant $\theta(1 + k_1\theta^2 + \dots)$	<code>cv2.fisheye.calibrate</code>	car surround-view, drones, robot vacuum, T265
Omnidirectional / catadioptric	≥ 180 – 360°	unified (Mei) model with mirror parameter ξ	<code>cv2.omnidir.calibrate</code> (contrib)	mirror + lens rigs, 360° cameras (dual fisheye)
Stereo pair	any	two intrinsics + relative pose (R, T)	<code>cv2.stereoCalibrate</code>	ZED, OAK-D, two Pi cameras, Kaggle stereo set
RGB-D / ToF	60–90°	factory-calibrated colour + depth intrinsics & extrinsics	vendor SDK, <code>calibrateCamera</code> on IR	RealSense, Kinect, Orbbec
Thermal	25–60°	pinhole: needs a heated or emissive target	<code>calibrateCamera</code>	FLIR Lepton
Event camera	~ 60–90°	pinhole; board shown as a blinking screen	reconstruct frames → OpenCV	Prophesee, DAVIS

The more the field of view grows, the more the simple pinhole + polynomial model breaks down: at 90° off-axis, $\tan \theta \rightarrow \infty$.

In plain words: the wider the lens sees, the more it bends lines, and the more special the maths. Normal lens → standard model; very wide → fisheye model.

Why fisheye needs a different model



- A pinhole maps a ray at angle θ to radius $r = f \tan \theta$, which goes to infinity at 90° . No finite sensor can hold a 180° view.
- Fisheye lenses are designed so r **grows slowly**: equidistant $r = f\theta$, equisolid, stereographic...
- OpenCV's **fisheye** module models the real lens as $\theta_d = \theta (1 + k_1\theta^2 + k_2\theta^4 + k_3\theta^6 + k_4\theta^8)$, so D has 4 coefficients.
- Rule of thumb: FOV above $\sim 120^\circ$, or the standard model leaves the edges curved $\rightarrow$ switch to `cv2.fisheye`.

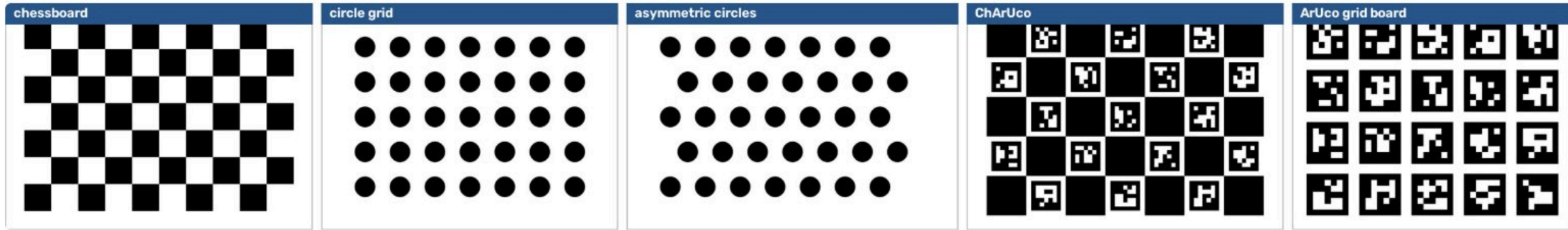
The cameras on our lab bench: Raspberry Pi

Module	Sensor	Resolution	Pixel	Focal length	H-FOV	Shutter	Calibrate as
Camera Module v2	IMX219	3280 × 2464	1.12 μm	3.04 mm	62°	rolling	standard
Camera Module 3	IMX708	4608 × 2592	1.4 μm	4.74 mm	66°	rolling	standard (fix the focus!)
Camera Module 3 Wide	IMX708	4608 × 2592	1.4 μm	2.75 mm	102°	rolling	wide: k3 or rational
HQ Camera	IMX477	4056 × 3040	1.55 μm	lens (6 mm, 16 mm...)	lens	rolling	depends on lens
Global Shutter Camera	IMX296	1456 × 1088	3.45 μm	lens	lens	global	depends on lens
M12 fisheye lens on HQ/GS	–	–	–	~1.5–2 mm	> 180°	–	fisheye

Camera Module 3 has autofocus. Intrinsic change with focus, so lock the lens
(`picam2.set_controls({"AfMode": 0, "LensPosition": ...})`) before you calibrate and keep it there.

Rolling shutter rows are exposed one after another, so a moving board or camera skews the image. Hold the board **still** for every shot, or use the Global Shutter camera.

Calibration targets



Target	Detect with	+	-
Chessboard	<code>findChessboardCorners</code> , <code>findChessboardCornersSB</code>	easy to print, very accurate saddle points	must be fully visible ; 180° ambiguity if rows = cols
(Asymmetric) circle grid	<code>findCirclesGrid</code>	robust to blur & noise	circle centres biased under strong perspective/distortion
ChArUco	<code>cv2.aruco.CharucoDetector</code>	partial views OK , every corner has an ID : ideal for fisheye & edges	needs the right dictionary & a sharp print
ArUco grid / AprilGrid	<code>cv2.aruco.ArucoDetector</code>	pose of each marker, multi-camera rigs (Kalibr)	marker corners less precise than saddles

Making a good target (it matters more than the code)

Do

- Print at **100 % scale**, then **measure** a square with a ruler or caliper: that is `square_size`
- Glue it to something **rigid and flat**: glass, aluminium, foam board
- **Matte** paper: no glare spots on the corners
- Keep a **white border** around the pattern (the detector needs it)
- Use **different** row/column counts (9×6 , not 8×8) so the orientation is unambiguous

Don't

- Use a board that bends or ripples: the model assumes $Z = 0$ for every corner
- Show it on a laptop screen at an angle (moiré, refresh, reflections). OK only as a last resort
- Count **squares** instead of **inner corners**
- Let auto-exposure / motion blur smear the corners

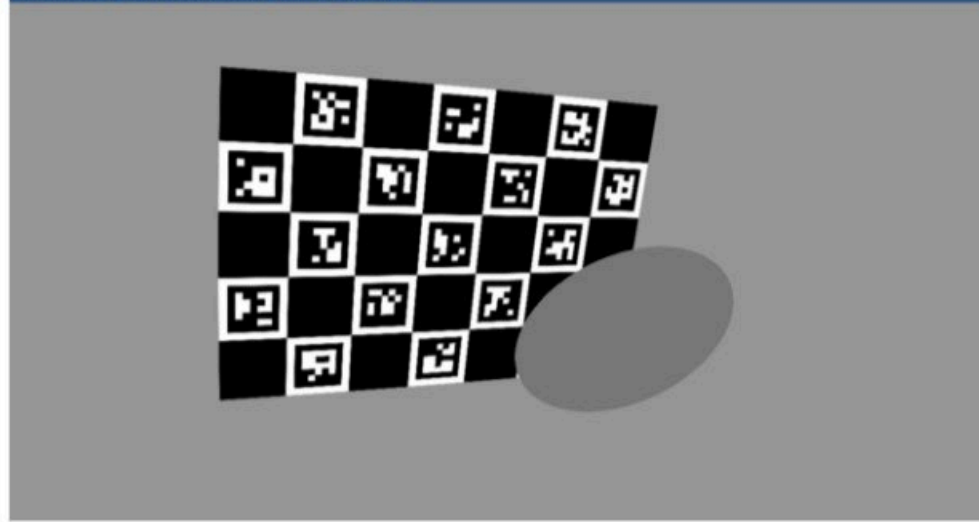
Pattern generators: calib.io, OpenCV

`gen_pattern.py` , or `board.generateImage()` in Lab 6E.

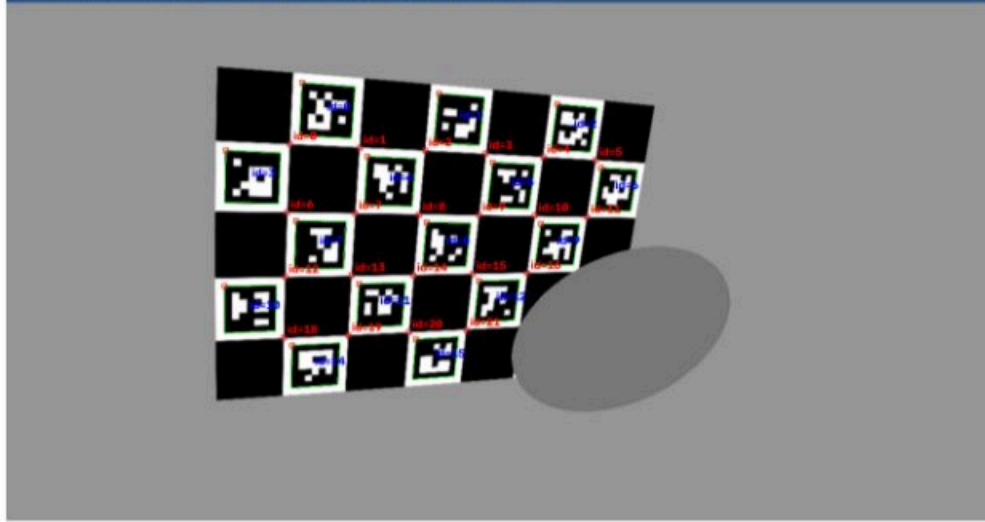
ChArUco: a chessboard with an ID on every square

► Lab 6E in Colab

rendered ChArUco view, partly covered



CharucoDetector: 21 of 24 corners, each with an ID



- ArUco markers inside the white squares tell the detector **which** corner it is seeing
- So **partial, occluded or out-of-frame** boards still give corners: perfect for covering image **edges** and fisheye lenses
- Corners are chessboard **saddle points** → as accurate as a chessboard

Watch: [How to do Camera Calibration with ChArUco Boards](#) (Nicolai Nielsen) • [OpenCV tutorial: calibration with ArUco and ChArUco](#)

ChArUco calibration: the code (OpenCV ≥ 4.7 / 5)

[► Lab 6E in Colab](#)

```
import cv2, glob

aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_5X5_100)
board = cv2.aruco.CharucoBoard((7, 5),          # squares x, y
                              (0.040, 0.030),  # square, marker (m)
                              aruco_dict)
cv2.imwrite("board.png", board.generateImage((2100, 1500), marginSize=10))

detector = cv2.aruco.CharucoDetector(board)
all_obj, all_img = [], []
for f in glob.glob("charuco/*.jpg"):
    gray = cv2.imread(f, cv2.IMREAD_GRAYSCALE)
    ch_corners, ch_ids, m_corners, m_ids = detector.detectBoard(gray)
    if ch_ids is not None and len(ch_ids) >= 6:  # partial view
        obj, img = board.matchImagePoints(ch_corners, ch_ids)
        all_obj.append(obj); all_img.append(img)

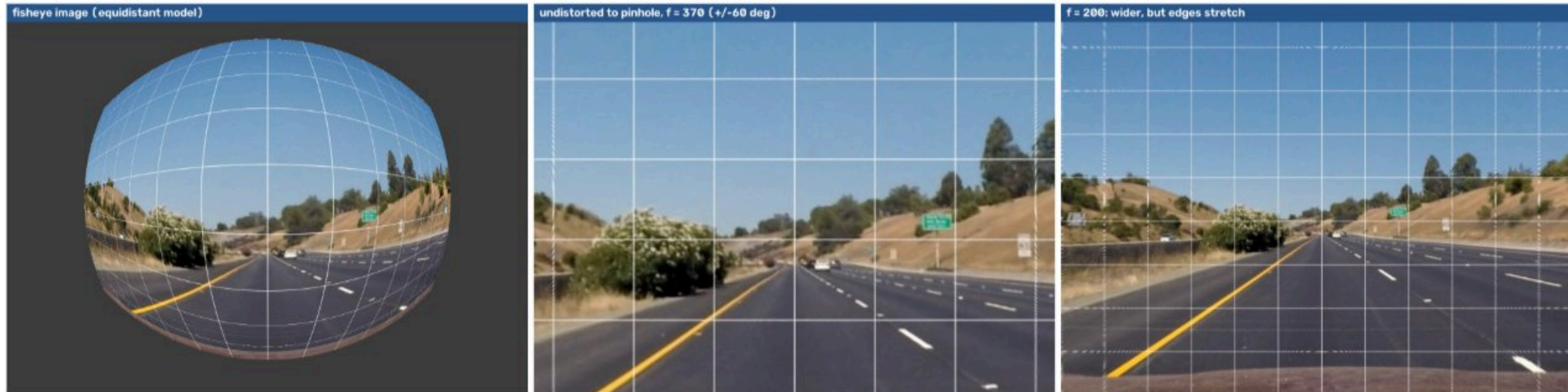
rms, K, dist, rvecs, tvecs = cv2.calibrateCamera(
    all_obj, all_img, gray.shape[::-1], None, None)
```

Notes

- The old `cv2.aruco.calibrateCameraCharuco` / `interpolateCornersCharuco` were removed in 4.7+. Use `CharucoDetector` + `matchImagePoints`.
- Measure both the square and marker size on the print.
- Pick a dictionary with few markers (e.g. 5x5_100) for better robustness.
- Lab 6E renders synthetic ChArUco views with a **known** camera, so you can check your calibration against the ground truth.

Fisheye lenses: cv2.fisheye

► Lab 6E in Colab



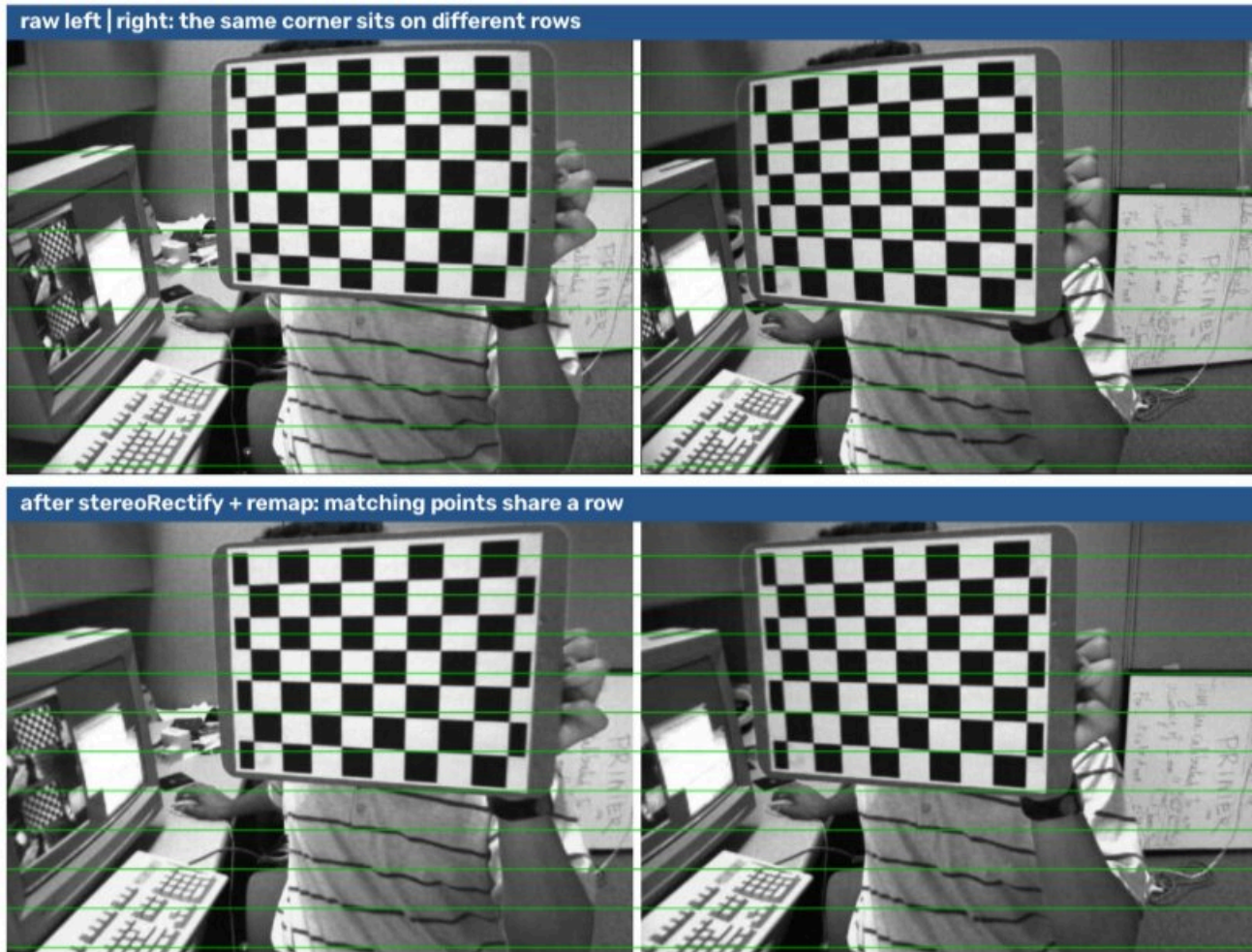
```
# per view: objpoints (1,N,3) float64, imgpoints (1,N,2) float64
# OpenCV 5: cv2.CALIB_* (OpenCV 4: cv2.fisheye.CALIB_*)
flags = cv2.CALIB_RECOMPUTE_EXTRINSIC | cv2.CALIB_FIX_SKEW
rms, K, D, rvecs, tvecs = cv2.fisheye.calibrate(
    objpoints, imgpoints, (w, h), None, None, flags=flags) # D =

newK = cv2.fisheye.estimateNewCameraMatrixForUndistortRectify(
    K, D, (w, h), np.eye(3), balance=0.0) # 0 cr
m1, m2 = cv2.fisheye.initUndistortRectifyMap(K, D, np.eye(3), newK,
und = cv2.remap(img, m1, m2, cv2.INTER_LINEAR)
```

- The whole 180° view **cannot** fit into one pinhole image: choose a narrower **newK** (middle) or accept stretched edges (right)
- Points must be float64, shape (1, N, 3): OpenCV 5 rejects the old (N, 1, 3). The #1 fisheye error
- Very wide (> 200°) or mirror cameras: `cv2.omnidir` (contrib)

Stereo calibration: two cameras, one rig

► Lab 6G in Colab



1. Capture the board with **both cameras at the same moment** (hardware sync or a still board)
2. Calibrate each camera $\rightarrow K_1, D_1, K_2, D_2$
3. `stereoCalibrate` $\rightarrow R, T$ between the cameras (+ E, F)
4. `stereoRectify` $\rightarrow$ rotations that make epipolar lines **horizontal**
5. `initUndistortRectifyMap` + `remap` per frame $\rightarrow$ disparity $\rightarrow$ depth

OpenCV's left/right sample pair: RMS 0.45 px, baseline 3.3 squares along $-x$.

Depth from disparity, $Z = fB/d$, is Lecture 10. Datasets: [Kaggle stereo chessboard pictures](#).

Stereo calibration: the code

► Lab 6G in Colab

```
# objpoints, imgL, imgR: corners found in BOTH images of each pair
_, K1, D1, _, _ = cv2.calibrateCamera(objpoints, imgL, size, None, None)
_, K2, D2, _, _ = cv2.calibrateCamera(objpoints, imgR, size, None, None)

rms, K1, D1, K2, D2, R, T, E, F = cv2.stereoCalibrate(
    objpoints, imgL, imgR, K1, D1, K2, D2, size,
    flags=cv2.CALIB_FIX_INTRINSIC)      # keep the mono results, solve only R, T
print("stereo RMS", rms, "baseline", np.linalg.norm(T) * square_mm, "mm")

R1, R2, P1, P2, Q, roi1, roi2 = cv2.stereoRectify(K1, D1, K2, D2, size, R, T, alpha=0)
mapL = cv2.initUndistortRectifyMap(K1, D1, R1, P1, size, cv2.CV_16SC2)
mapR = cv2.initUndistortRectifyMap(K2, D2, R2, P2, size, cv2.CV_16SC2)
rectL = cv2.remap(left, *mapL, cv2.INTER_LINEAR)
rectR = cv2.remap(right, *mapR, cv2.INTER_LINEAR)      # now matching points share a row
```

Two Raspberry Pi cameras on a bracket (or the Pi 5's two CSI ports) make a cheap stereo rig. Keep them **rigidly** mounted: any movement invalidates R, T. See [Raspberry Pi stereo camera calibration](#).

Check the rectified pair: draw horizontal lines. If the same corner is on different rows, the calibration or the synchronisation is off.

Challenge: calibrate your own camera

[▶ Lab 6H in Colab](#)

```
# capture_calib.py (Lab 6H): Pi camera or USB webcam
from picamera2 import Picamera2          # USB: cv2.VideoCapture(0)
import cv2, time, os

picam2 = Picamera2()
picam2.configure(picam2.create_still_configuration(
    main={"size": (1920, 1080), "format": "RGB888"}))
picam2.start()
picam2.set_controls({"AfMode": 0, "LensPosition": 0.0}) # Module 3

os.makedirs("calib_images", exist_ok=True)
for i in range(25):
    time.sleep(3)                                # move the board, hold
    frame = picam2.capture_array()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    ok, _ = cv2.findChessboardCorners(gray, (9, 6),
                                      cv2.CALIB_CB_FAST_CHECK)
    cv2.imwrite(f"calib_images/img_{i:02d}.jpg", frame)
    print(i, "board found" if ok else "NO board: move it")
```

Recipe

1. Print a 9×6 board ([pattern.png](#)), stick it to a flat board, measure a square
2. Take 20–30 pictures over the **whole field of view**, especially near the edges
3. Run `calibrate.py --board 9x6 --square 24.5 → camera.yaml`
4. Check $\text{RMS} < 0.5\text{--}1$ px and that straight lines are straight
5. Commit `camera.yaml` next to your project code

Based on [Petros626/Camera-Calibration-with-PiCamera2-OpenCV](#), [niconielsen32/camera-calibration](#), [nuwandda/Checkerboard-Calibration](#), [ibois-epfl/rgb-camera-calibration](#).

Calibration tools in the wild

Tool	Best for	Notes
OpenCV <code>calib3d</code> (<code>cv2.calibrateCamera</code> , <code>fisheye</code> , <code>stereoCalibrate</code> , <code>aruco</code>)	everything in this course	calib3d reference : models, flags, functions
OpenCV <code>interactive-calibration</code> app	guided live calibration	suggests the next pose, shows per-parameter uncertainty
ROS 2 <code>camera_calibration</code>	robots, mono/stereo, fisheye, ChArUco	X/Y/Size/Skew bars → <code>camera_info</code> YAML
MATLAB Camera Calibrator / Stereo Camera Calibrator	teaching, quick visual checks	error plots and 3-D pose views
Kalibr (ETH Zürich)	multi-camera + IMU, rolling shutter	AprilGrid targets; the standard for VIO / SLAM rigs
calib.io camera calibrator, <code>mrcal</code> (NASA JPL)	high-accuracy industrial / metrology	richer lens models, uncertainty maps

Whatever the tool, the inputs and outputs are the same: **known 3-D target points + detected 2-D corners** → **K, distortion, poses, reprojection error**.

Watch & learn: recommended videos



Intrinsics & extrinsics
Cyrill Stachniss · 5 min



Intrinsic camera calibration
Cyrill Stachniss · 5 min



Zhang's method
Cyrill Stachniss · lecture



Camera calibration explained,
step by step
Kevin Wood



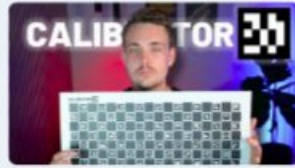
OpenCV Python calibration
(intrinsic, extrinsic, distortion)
Kevin Wood



Camera calibration in < 5 minutes
Nicolai Nielsen



Calibration with a Python script
Nicolai Nielsen



ChArUco board calibration
Nicolai Nielsen



Calibrate an OpenCV camera in
15 minutes
The Drone Dojo (Raspberry Pi)

📖 Step-by-step tutorials for all 9 videos

Video labs:

6I Pi in 15 min

6J Nicolai

6K Kevin Wood

6L ChArUco

6M Zhang (Cyrill)

Calibrate your Raspberry Pi camera in 15 minutes

```
# on the Pi (SSH + VNC), Raspberry Pi OS Bookworm
mkdir -p ~/calib && cd ~/calib
for f in pi_calibrate_20.py aruco_pose_test.py; do
  wget https://raw.githubusercontent.com/Kazimbalti/\
intelligent-vision-systems/main/LAB/Lab_6/$f; done
wget https://raw.githubusercontent.com/opencv/opencv/4.x/doc/pattern.png
sudo apt install -y python3-opencv python3-picamera2
```

```
# 1. calibrate: 26 mm squares, 640 x 480 (as in the video)
python3 pi_calibrate_20.py --mm 26 --width 640 --height 480
```

```
# 2. print a marker, measure it, test with a ruler
python3 aruco_pose_test.py --make-marker --id 72
python3 aruco_pose_test.py --size 200 --id 72 \
  --width 640 --height 480
```

1. Print **pattern.png** on A4, glue it flat, **measure** a square
2. The script keeps **20 good, different** frames by itself: show the **edges** first, then near, far, tilted
3. It saves **cameraMatrix.txt**, **cameraDistortion.txt**, **camera.yaml**
4. **Ruler test:** centre an ArUco marker ($x \approx 0$), slide it **6 cm**: the program must say ≈ 60 mm
5. Same resolution in every script!

After [The Drone Dojo: Calibrate OpenCV Camera In Under 15 Minutes](#), updated for Bookworm + OpenCV 5 (Picamera2, apt instead of sudo pip, solvePnP instead of estimatePoseSingleMarkers).

[► Lab 6I in Colab](#)[Step-by-step tutorial](#)

Questions students always ask

Do I calibrate every time I switch the camera on?

No. Once per camera + lens + focus + resolution. Save `camera.yaml` and reuse it.

I changed the resolution. Recalibrate?

Best yes. Same aspect ratio: just scale f_x , f_y , c_x , c_y by the resize factor (dist stays the same).

Two cameras of the same model: same K?

Close, but not equal. Every unit is assembled slightly differently, so calibrate each one.

Why a chessboard and not any picture?

We must know the true shape exactly, and chessboard corners can be found to 0.1 px.

How many photos?

3 is the mathematical minimum; use **15–30** sharp, tilted views covering the whole frame.

My RMS is 0.3 px. Am I done?

Almost: also check the board covered the edges, and do a ruler test (Lab 6I): does a known distance come out right?

Summary

Model

- $s \cdot x = K [R | t] X$: $K (f_x, f_y, c_x, c_y)$ is the camera, $[R | t]$ is where it is
- Lenses add **radial** (k_1, k_2, k_3) and **tangential** (p_1, p_2) distortion, applied in normalised coordinates
- Wide and fisheye lenses need k_3 / rational or the **fish-eye** model; stereo needs **R, T** between cameras

Practice

- `findChessboardCorners` (+ `cornerSubPix`) → `calibrateCamera` → $RMS < 0.5-1$ px
- 20+ sharp, tilted views covering the **whole frame**; same resolution & focus as deployment
- `undistort` or `initUndistortRectifyMap` + `remap` for video
- **Undistort first**, then `getPerspectiveTransform` + `warpPerspective` → bird's-eye → curvature

Next: Lecture 10 builds on this with epipolar geometry, disparity and depth from stereo, and the Lab 9 measuring system needs your calibrated camera.

Lab 6: all notebooks (click to open in Colab)

6A Pinhole projection & distortion

6C Calibrate & undistort (GoPro)

6E ChArUco & fisheye (known truth)

6G Stereo calibration & rectification

6I 📺 Pi camera in 15 minutes

6K 📺 Calibration explained (Kevin Wood)

6M 📺 Zhang from scratch (Cyrill)

6B Finding chessboard corners

6D Road camera: bird's-eye & curvature

6F Bug hunt 🐛 (pairs)

6H Calibrate your own Pi / USB camera

6J 📺 Get images → calibrate (Nicolai)

6L 📺 ChArUco pipeline (Nicolai)

 Labs page: Colab · View · Download for every notebook

📺 Webcam Calibration Studio (browser) · [webcam_calibration_studio.py](#) (offline) · Scripts: [pi_calibrate_20.py](#) · [aruco_pose_test.py](#) · [capture_calib.py](#) · [calibrate.py](#) · [get_images.py](#) · [all on GitHub](#) · [video tutorials](#)

Reading: [OpenCV calib3d](#) · [ArUco/ChArUco calibration](#) · [ROS camera_calibration](#) · Zhang, TPAMI 2000 · Szeliski 2e §2.1, §11.1 · Medium: [checkerboard calibration](#) · [udacity/CarND-Camera-Calibration](#)

Credits. Adapted and extended from the Camera Calibration lesson of the Udacity Self-Driving Car Engineer Nanodegree. Lesson illustrations and calibration images © Udacity, Inc. (MIT-licensed repositories); stereo sample images from OpenCV (Apache 2.0). Videos © their authors. Used here for teaching.

Questions?

Next: Lab 6. Calibrate the GoPro, the road camera, a stereo pair, and then your own camera.