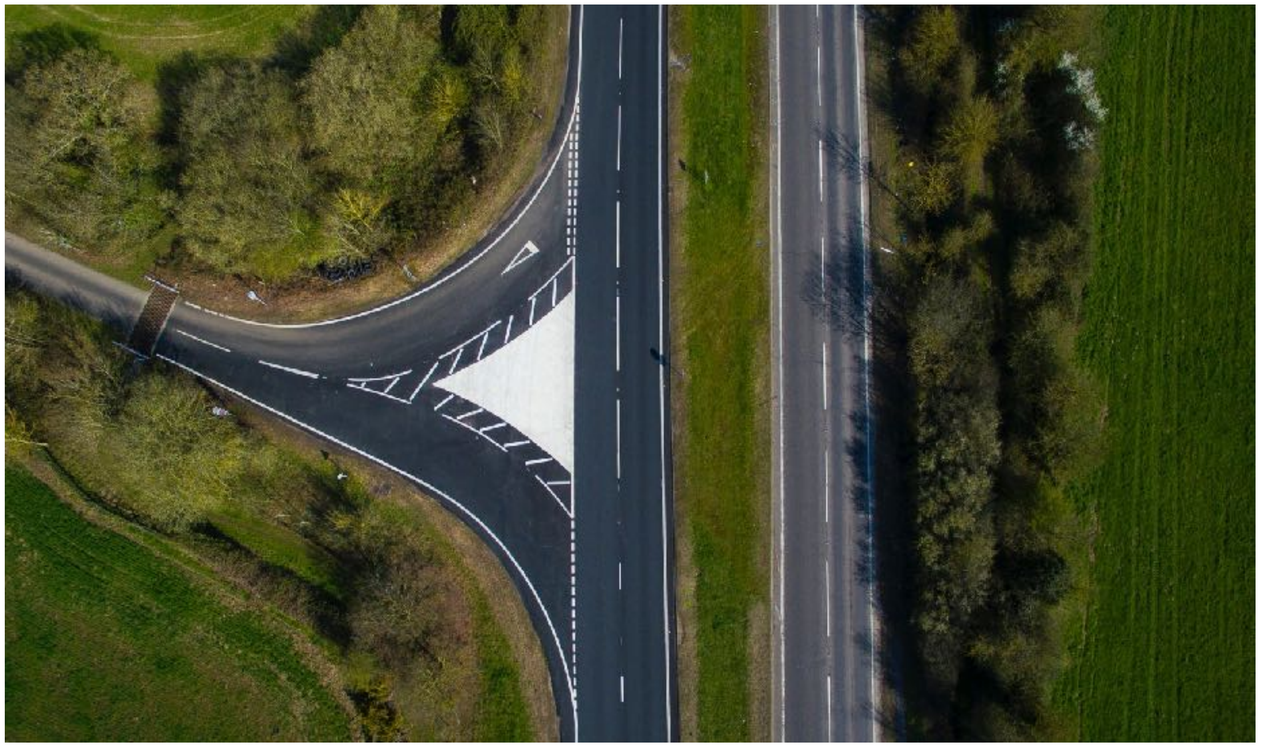


Lane Detection Using Images

- Chi-Sheng SHIH
- National Taiwan University

國立臺灣大學

NEWS Lab
嵌入式系統暨無線網路實驗室

Agenda

- Image Sensors Calibration
- Lane Detection for Self-Driving Cars

Lane Detection Using Image Sensors

Lane Detection

- Lane detection is a fundamental components for ADAS and self-driving cars. For example,
  - **Lane departure warning (LDW)**: alerts the driver when the car changes lanes without turning on signals.
  - **Lane Following**: follow the lane lines to drive the car.
- The core operation is to detect the lane line using onboard sensors.
  - **Precision localization and HD map**: given the map and location of the vehicle, the vehicle can locate itself between lane lines.
  - **Image sensors**: detect the lane lines according to the defined property of lane lines such as color and dimensions.
  - **Lidar sensors**: detect the lane lines according to the intensity of the reflection.

Finding Lane Lines from Images

1. Calibrate camera to remove distortion
2. Warping images
3. Finding lane lines
4. Curve fitting
5. Overlap Results on Image

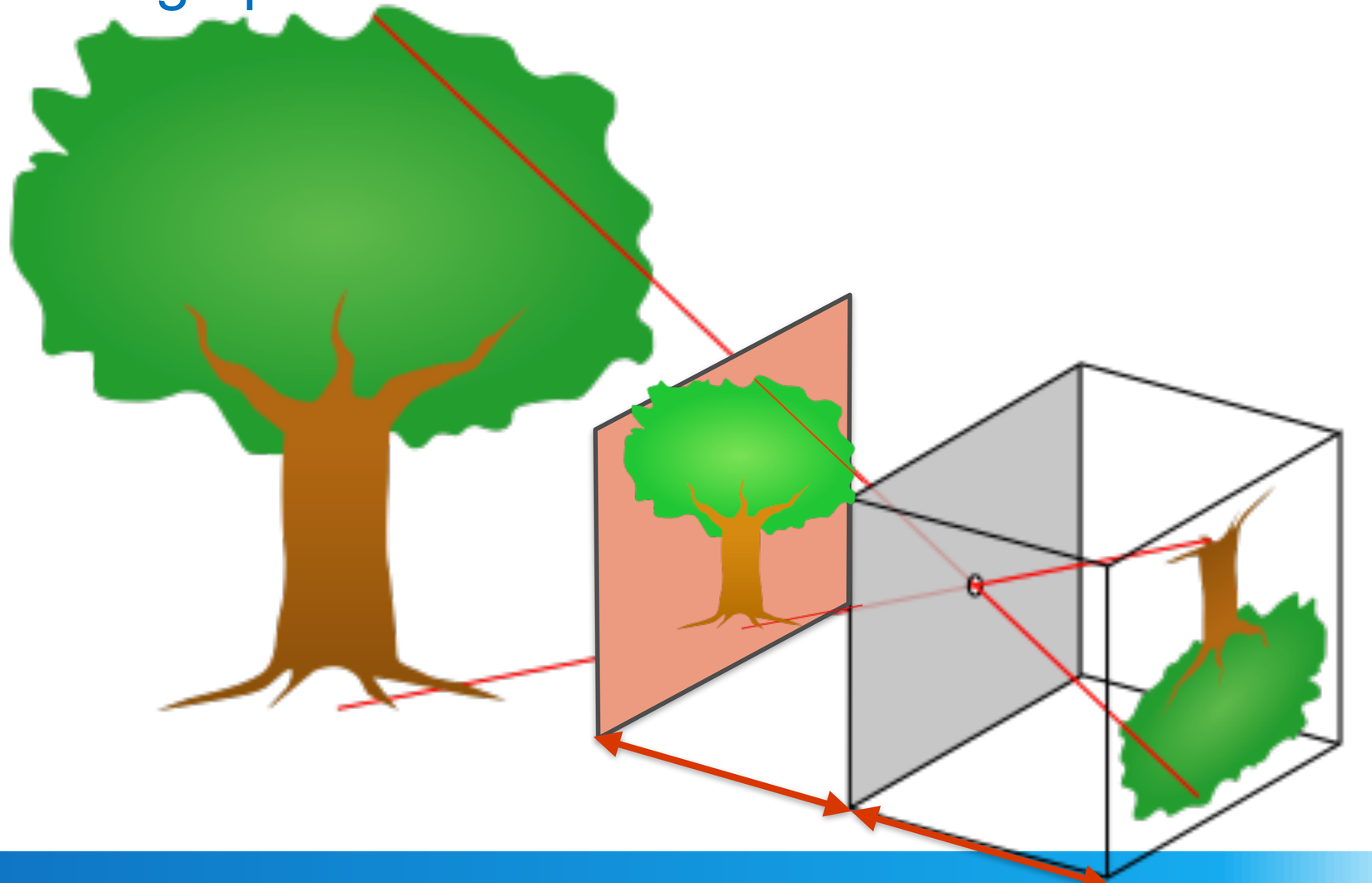
Camera Calibration

What is Camera Calibration?

- A camera projects 3D world-points onto the 2D image plane.
- **Calibration**: Finding the quantities internal to the camera that affect this imaging process
  - Image center
  - Focal length
  - Lens distortion parameters

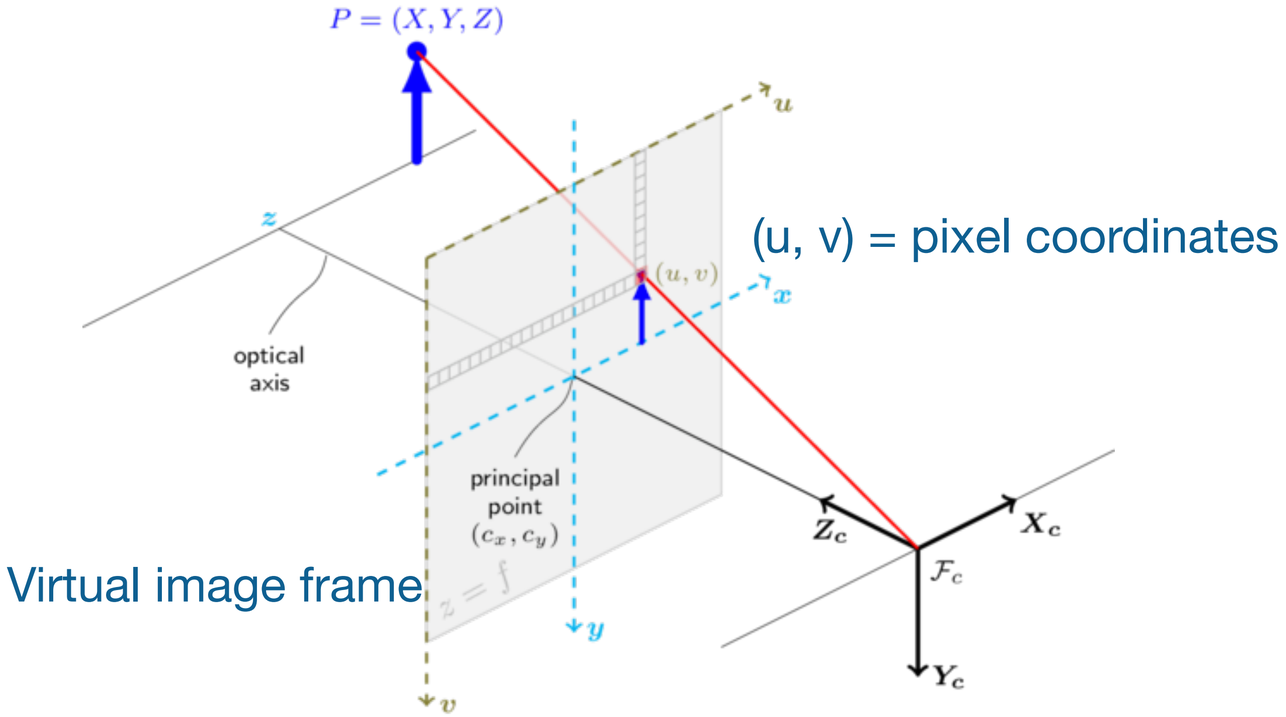
Pin-Hole Camera Model

- A camera projects 3D world-points onto the 2D image plane.



Pin-Hole Camera Model

(x, y, z) = world coordinates



(u, v) = pixel coordinates

Virtual image frame

Ideally, the light goes straights. However, in order to capture more light and increase field of views, lens are added to change the light path.

As a result, the points, (u, v), projected on (virtual) camera frame need to be adjusted so that they are ***isomorphic*** to the points, (x, y, z), in world coordinates.

The process to find the parameters is called ***calibration***.

What is Camera Calibration?

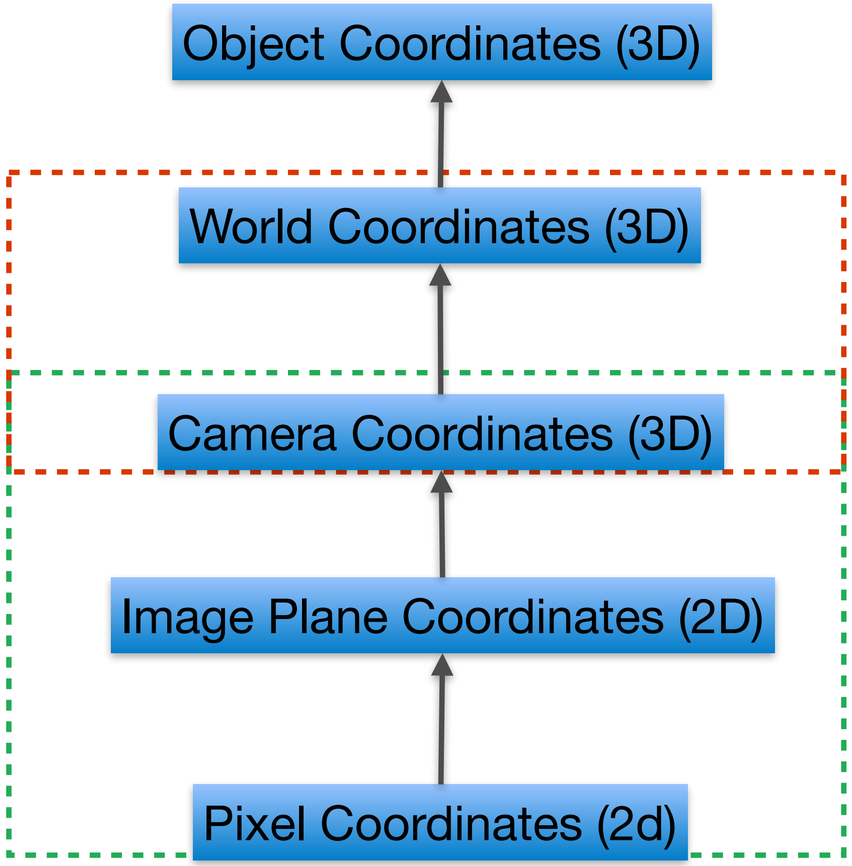
- A camera projects 3D world-points onto the 2D image plane.
- **Calibration**: Finding the quantities internal to the camera that affect this imaging process
  - Image center
  - Focal length
  - Lens distortion parameters

Motivation

- Causes of errors for calibrations:
  - Camera production errors
  - Cheap lenses

- Precise calibration is required for
  - 3D interpretation of images
  - Reconstruction of world models: e.g., lane lines
  - Robot interaction with the world (Hand-eye coordination)

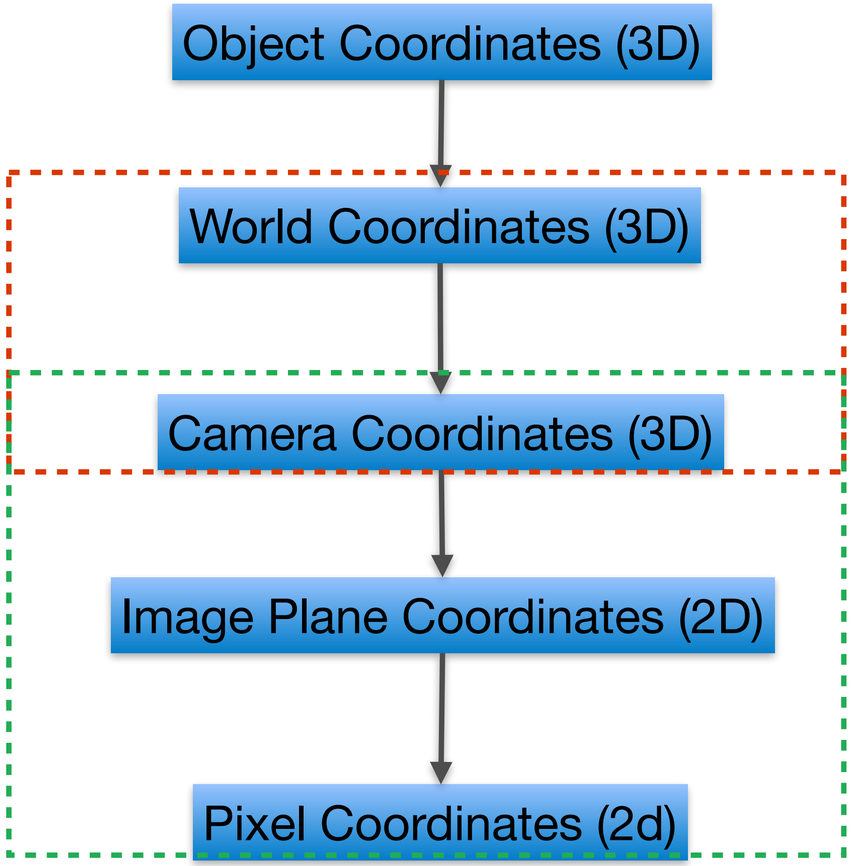
From 2D to 3D pixel frame



Extrinsic
camera parameters

Intrinsic
camera parameters

From 3D to 2D pixel frame



Extrinsic
camera parameters

Intrinsic
camera parameters

Projective Geometry

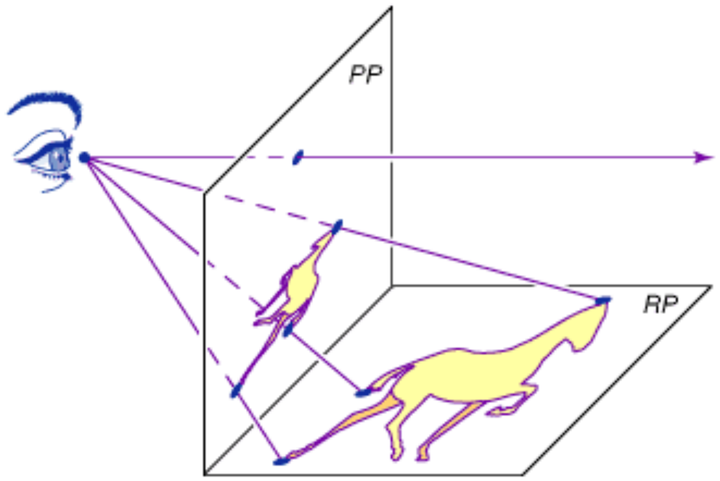
- Extension of Euclidean coordinates towards points at infinity

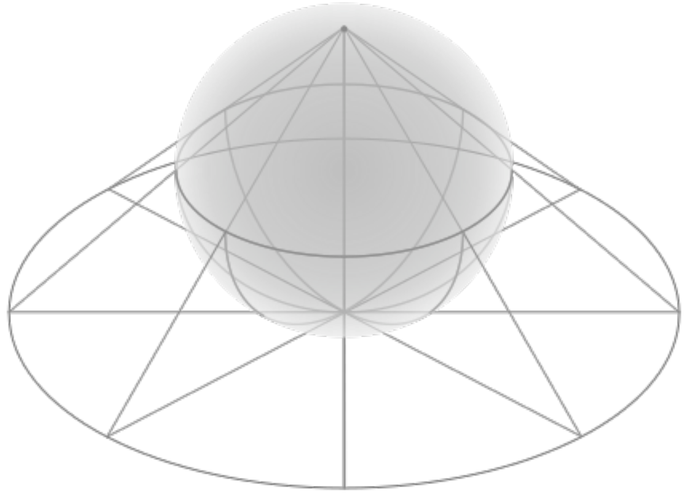
$$\mathbb{R}^n \to \mathbb{P}^n : (x_1, x_2, \ldots x_n) \to (\lambda x_1, \lambda x_2, \ldots, \lambda x_n, \lambda) \in \mathbb{R}^{n+1} \backslash 0_{n+1}$$

- Here, equivalence is defined up to scale:

$$\hat{x} \sim \hat{y} \Leftrightarrow \exists \lambda \in \mathbb{R} \backslash \{0\} : \hat{x} = \lambda \hat{y}$$

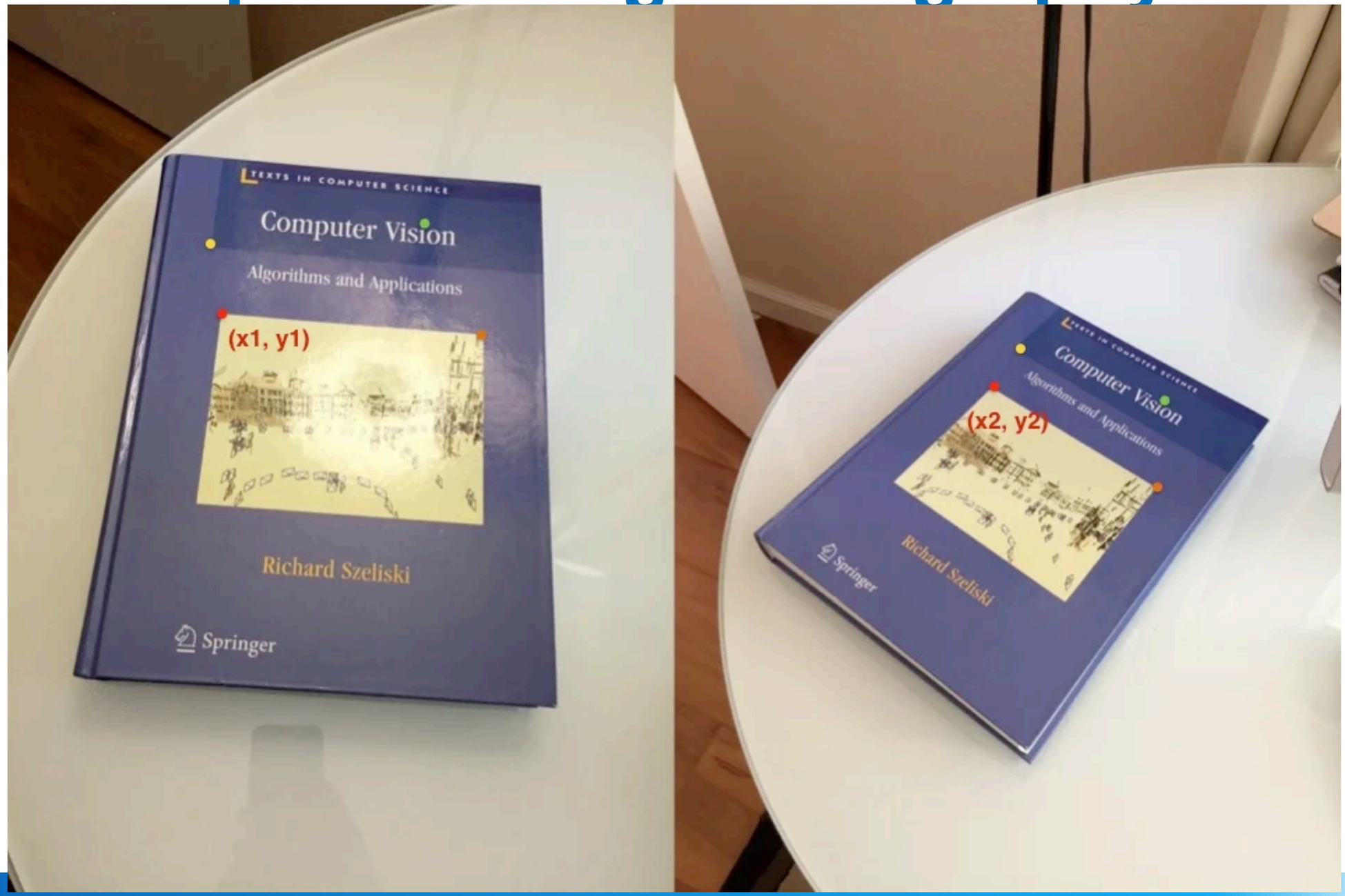
- Special case: Projective Plane $\mathbb{P}^2$
- A linear transformation within is called a homography $\mathbb{P}^2$



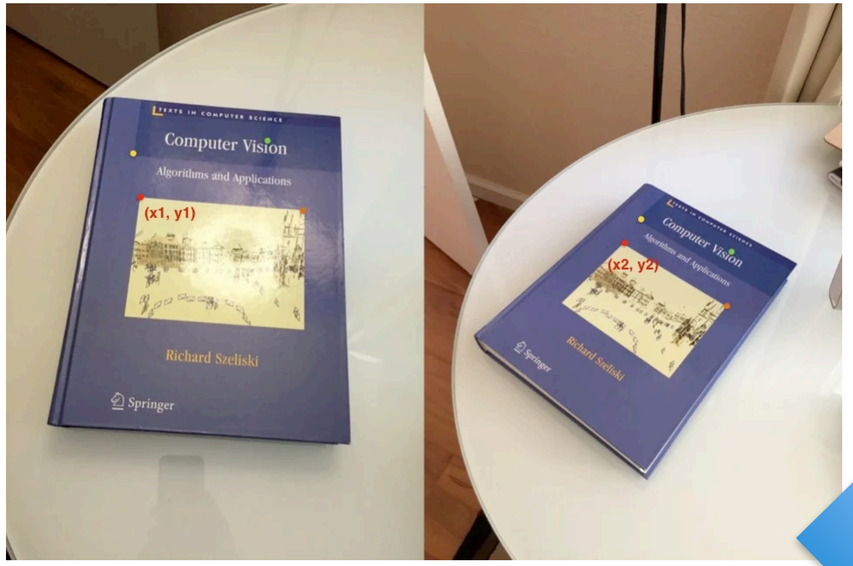


Projecting a sphere to a plane.

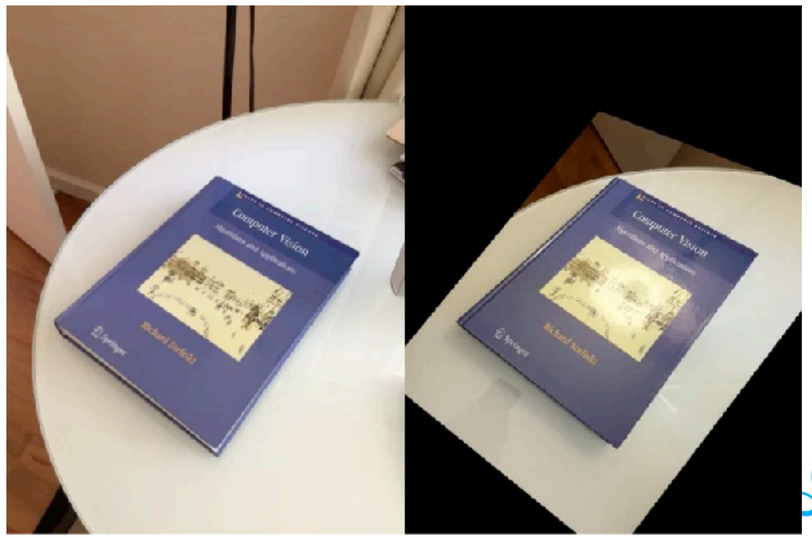
Example of Using Homography

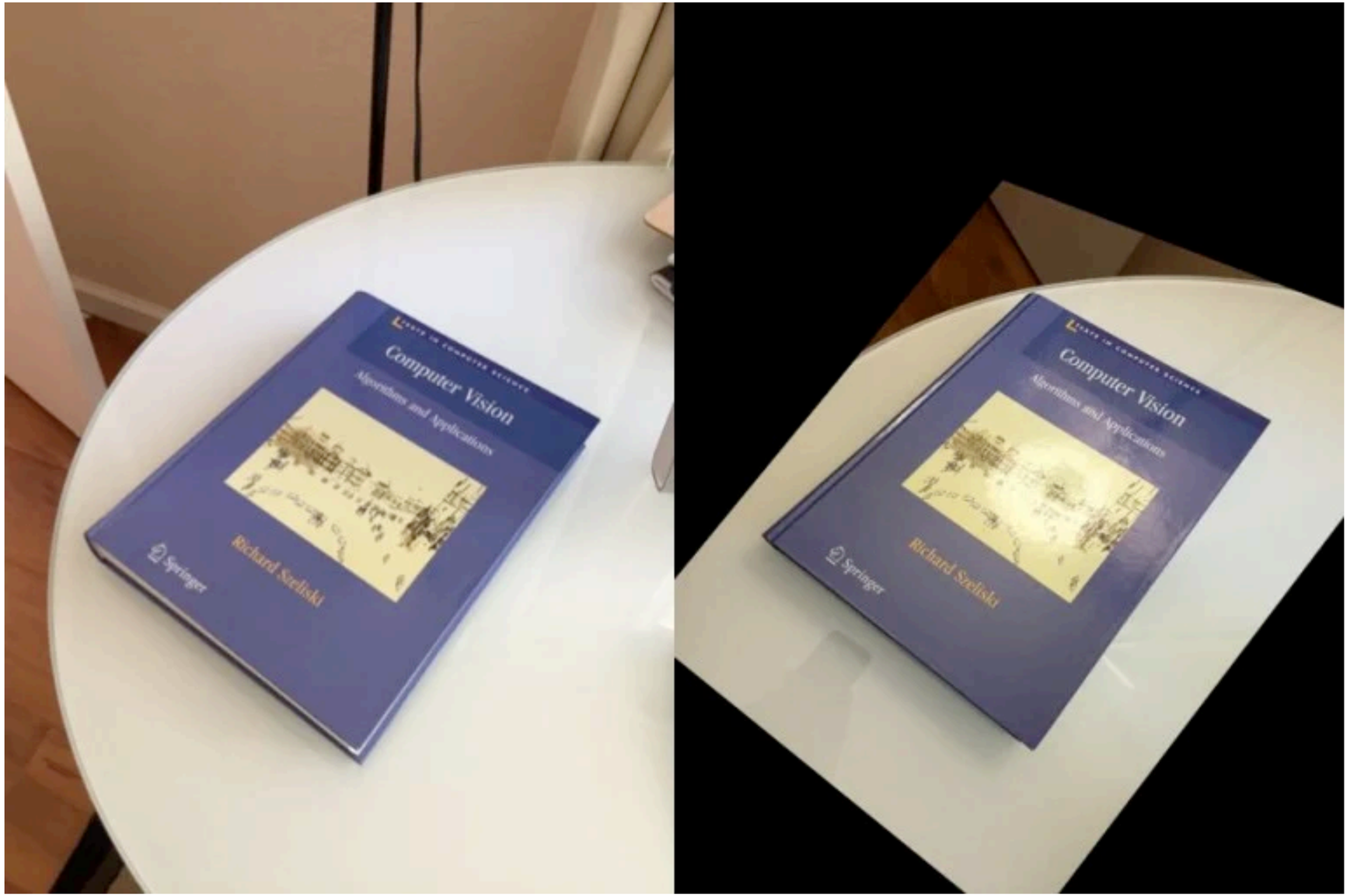


Example of Using Homography

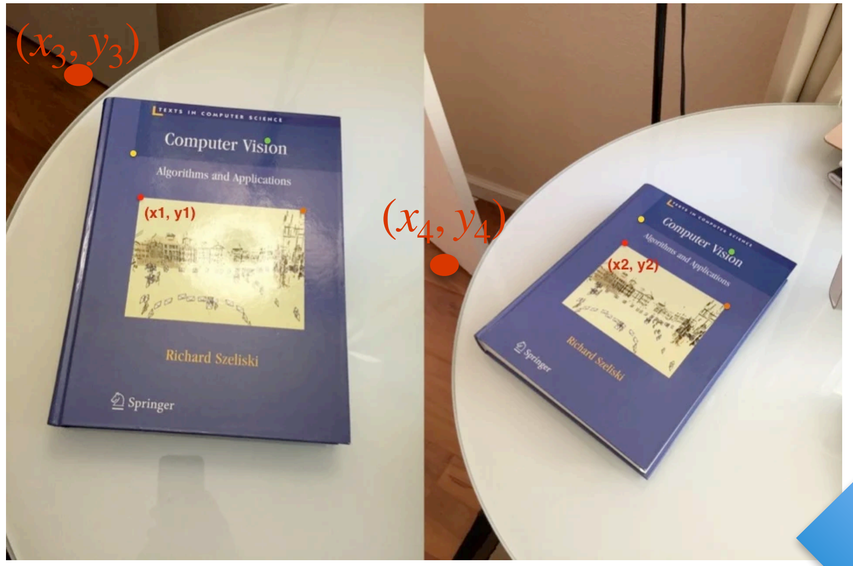


(x_1, y_1) and (x_2, y_2) are the points located at the same place in world coordinates

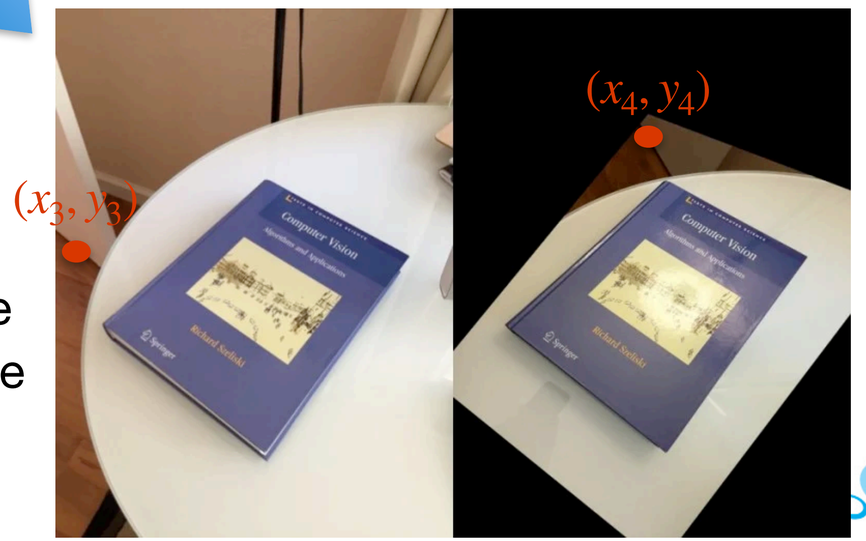



Computer Vision
Algorithms and Applications
Richard Szeliski
Springer
Computer Vision
Algorithms and Applications
Richard Szeliski
Springer

Example of Using Homography



(x_1, y_1) and (x_2, y_2) are the points located at the same place in world coordinates



(x_3, y_3) and (x_4, y_4) are ***NOT*** the points located at the same place in world coordinates.

Homography

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \sim \underbrace{\begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}}_{\text{Homography } \mathbf{H}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- H has 9-1(scale invariance)=8 DoF
- One pair of points gives us 2 equations
- Therefore, we need at least 4 point correspondences for calculating a homography matrix.

Pinhole Camera Model

- Perspective transformation using homogeneous coordinates:

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

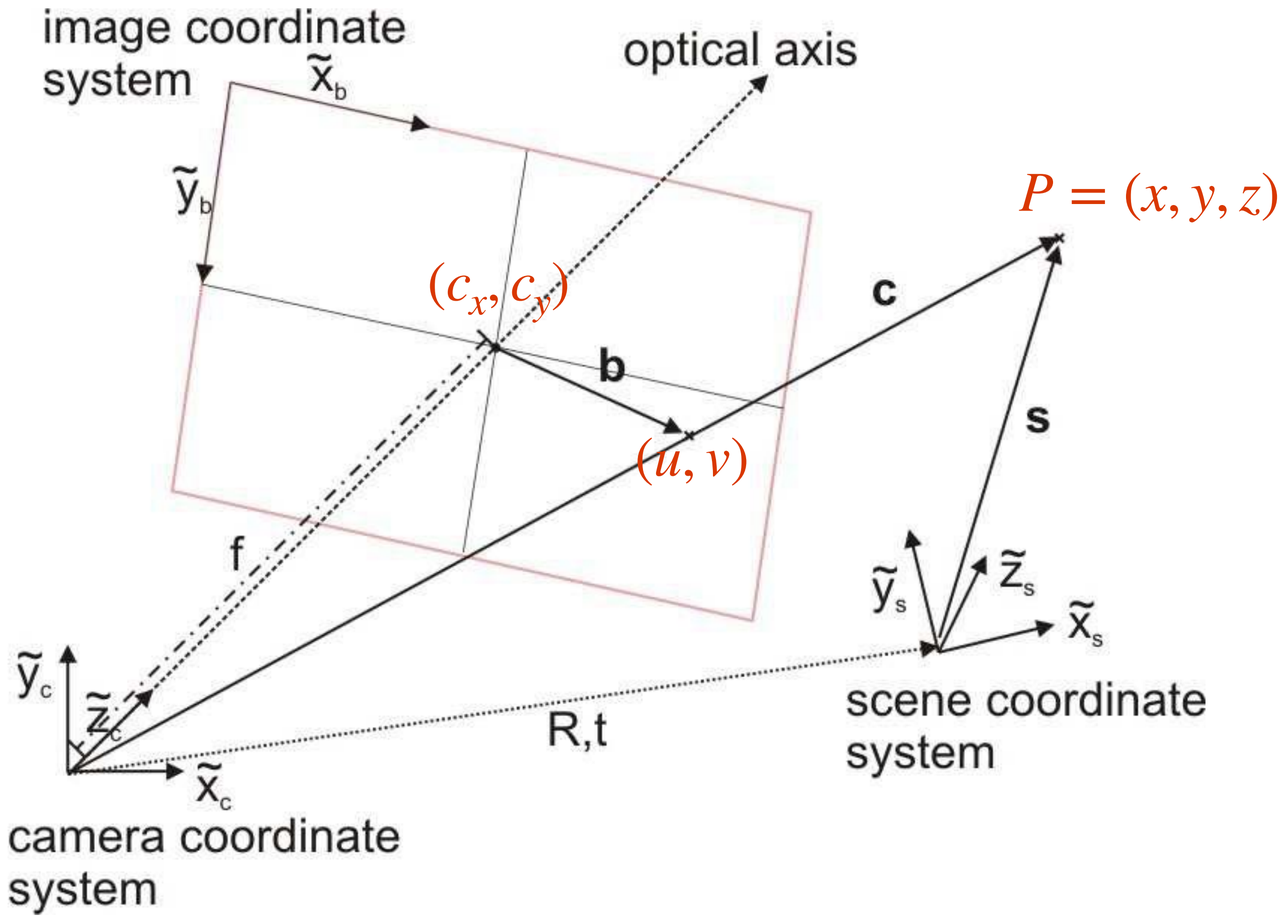
Intrinsic camera parameters

Extrinsic camera parameters

Intrinsic parameters: parameters to project points on pixel frame onto reference frame.

Extrinsic parameters: parameters to project points on camera reference frame onto world reference frame.

Pinhole Camera Model

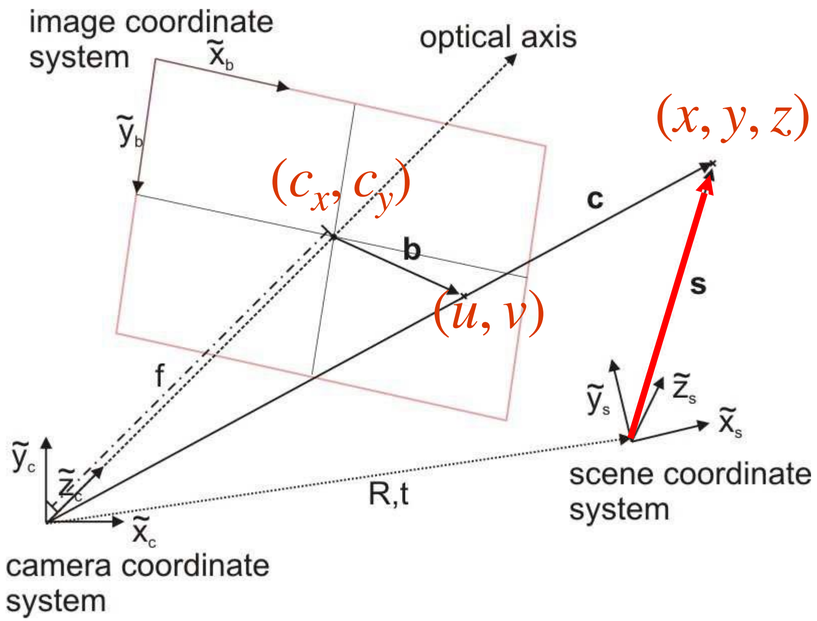


Pinhole Camera Model

- Perspective transformation using homogeneous coordinates:

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

world/scene coordinate system

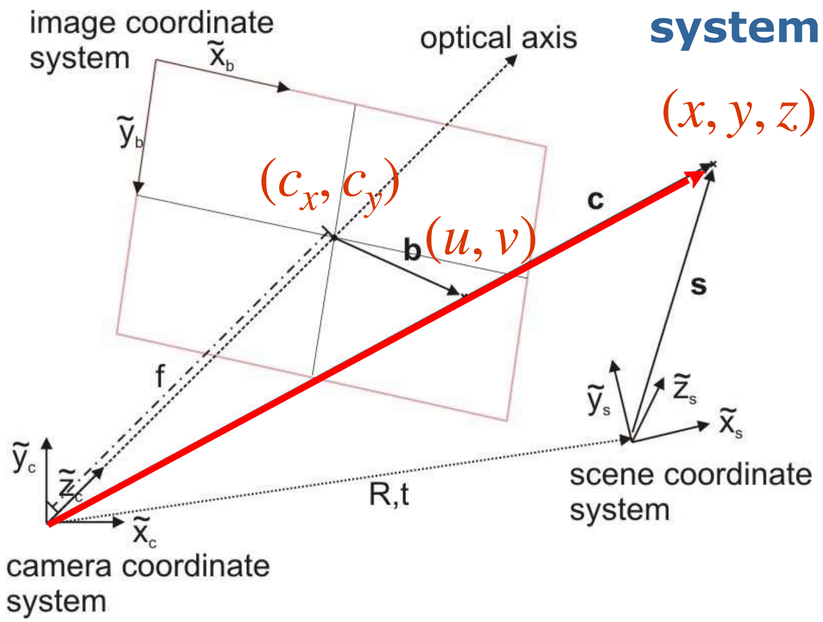


Pinhole Camera Model

- Perspective transformation using homogeneous coordinates:

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

camera coordinate system

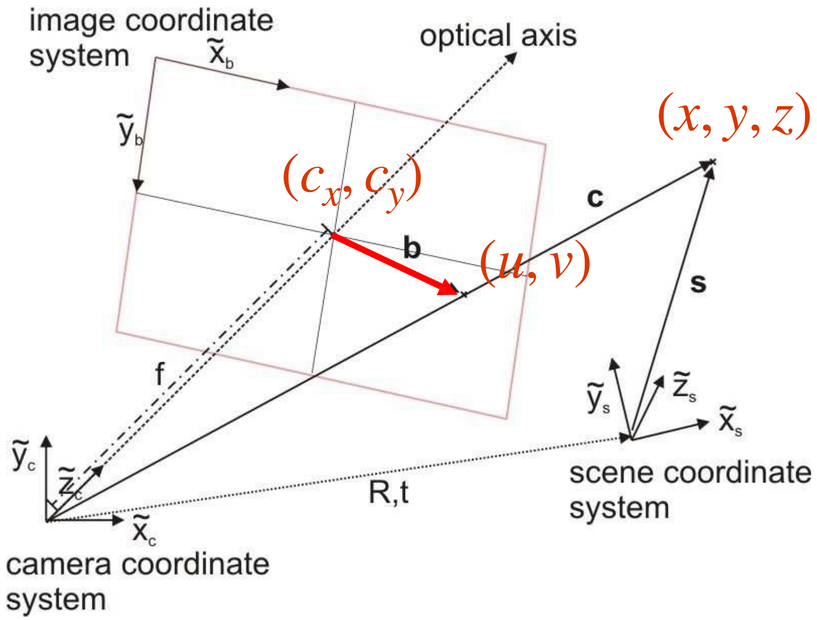


Pinhole Camera Model

- Perspective transformation using homogeneous coordinates:

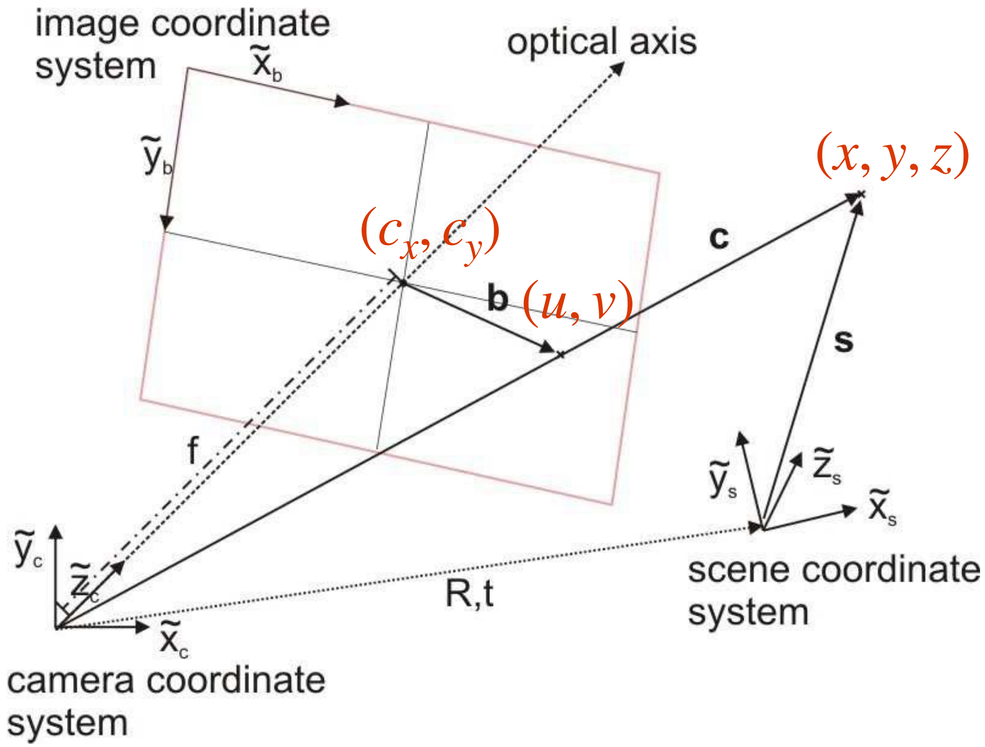
$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

image coordinate system



Pinhole Camera Model

- Interpretation of intrinsic camera parameters:



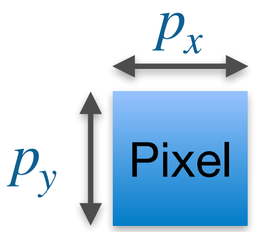
$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

(f_x, f_y): focal length in pixels.

$$f_x = \frac{F}{p_x} \text{ and } f_y = \frac{F}{p_y}$$

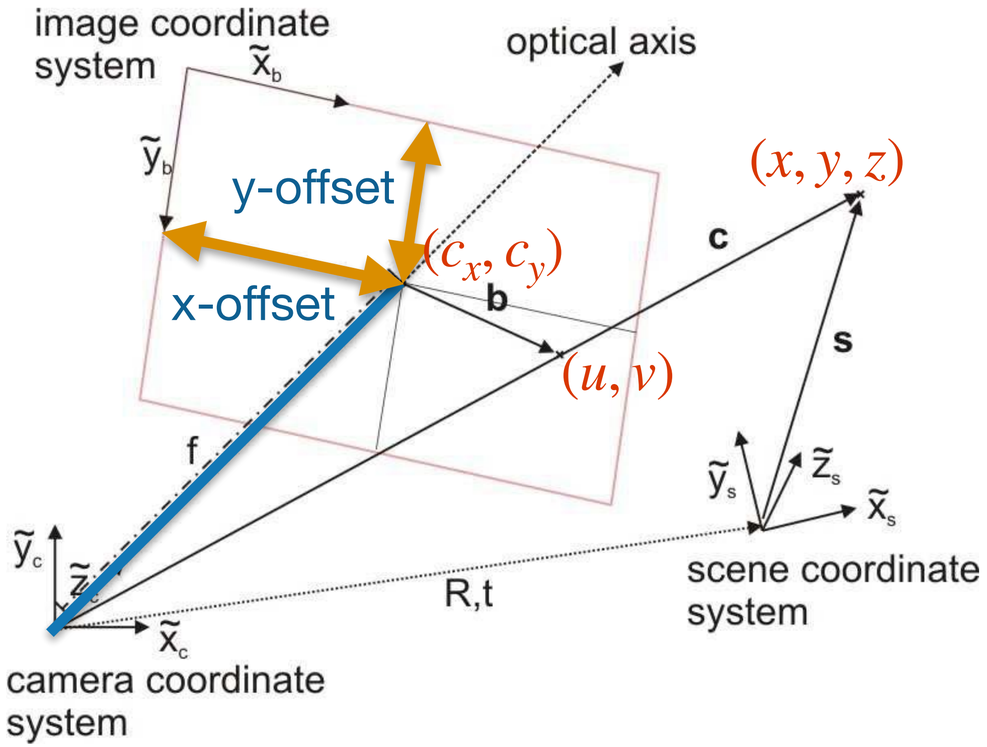
F: focal length in world units

(p_x, p_y): size of pixels in world units



Pinhole Camera Model

- Interpretation of intrinsic camera parameters:



$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

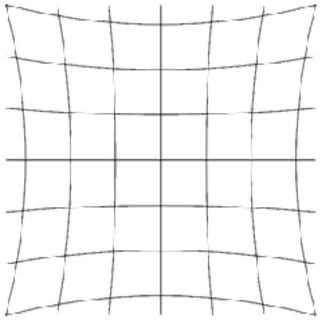
focal length

offset

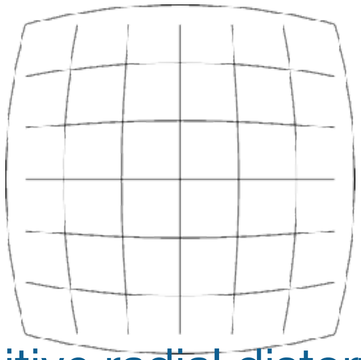
Lens distortion

Non-linear effects:

- Radial distortion
- Tangential distortion
- Compute corrected image point:



Negative radial distortion



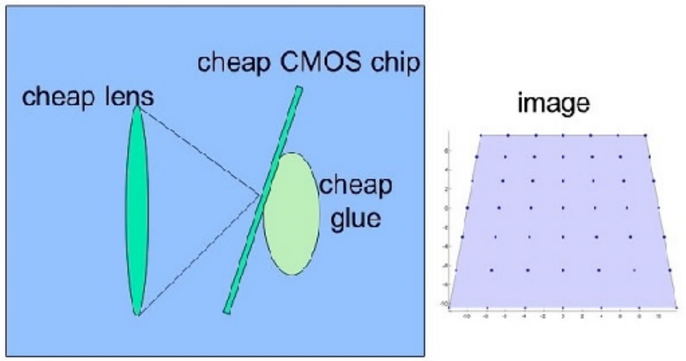
Positive radial distortion

$$(1)\quad \begin{aligned} x' &= x/z \\ y' &= y/z \end{aligned}$$

$$(2)\quad \begin{aligned} x'' &= x'(1 + k_1 r^2 + k_2 r^4) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\ y'' &= y'(1 + k_1 r^2 + k_2 r^4) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \end{aligned}$$

where $r^2 = x'^2 + y'^2$, k_1, k_2: radio distortion coefficients,

and p_1, p_2: tangential distortion coefficients.



Tangential distortion

$$(3)\quad \begin{aligned} u &= f_x \cdot x'' + c_x \\ v &= f_y \cdot y'' + c_y \end{aligned}$$

Camera Calibration

- Calculate intrinsic parameters and lens distortion from a series of images
  - 2D camera calibration
  - 3D camera calibration
  - Self calibration

Camera Calibration

- Calculate intrinsic parameters and lens distortion from a series of images
  - 2D camera calibration
  - 3D camera calibration

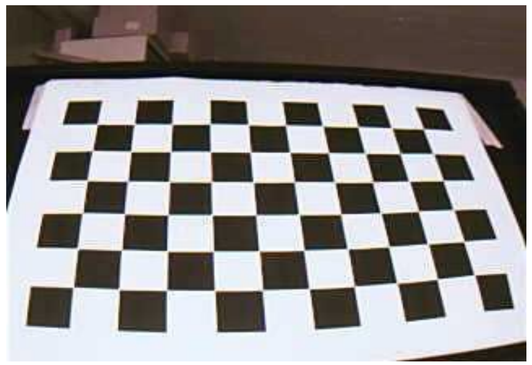
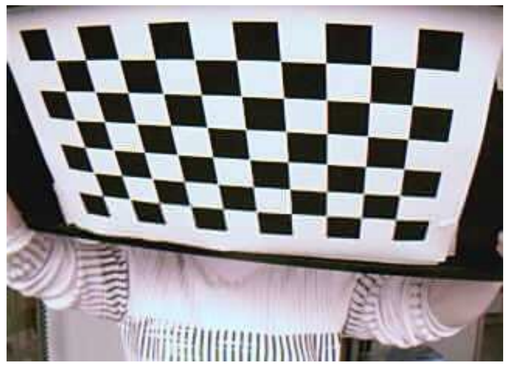
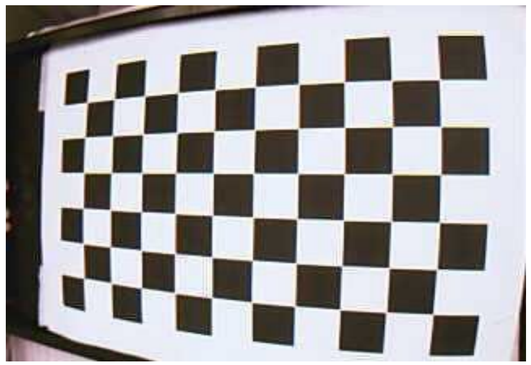
  need external pattern
  - Self calibration

Camera Calibration

- Calculate intrinsic parameters and lens distortion from a series of images
  - 2D camera calibration
  - 3D camera calibration
  - self calibration

2D Camera Calibration

- Use a 2D pattern (e.g., a checkerboard)

- Trick: set the world coordinate system to the corner of the checkerboard and the dimension of the checker board in world units are known.

2D Camera Calibration

- Use a 2D pattern (e.g., a checkerboard)

- Trick: set the world coordinate system to the corner of the checkerboard and the dimension of the checker board in world units are known.
- Now: All points on the checkerboard lie in one plane!

2D Camera Calibration

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ \boxed{0} \\ 1 \end{pmatrix}$$

- Since all points lie in a plane, their z components are 0 in world coordinates

2D Camera Calibration

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix}$$

- Since all points lie in a plane, their z components are 0 in world coordinates.

2D Camera Calibration

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix}$$

- Since all points lie in a plane, their z components are 0 in world coordinates
- Thus, we can delete the 3rd column of the Extrinsic parameter matrix.

2D Camera Calibration

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- Since all points lie in a plane, their z components are 0 in world coordinates
- Thus, we can delete the 3rd column of the Extrinsic parameter matrix

2D Camera Calibration

$$b = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \overbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}^{\text{Homography } \mathbf{H}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- Since all points lie in a plane, their z components are 0 in world coordinates.
- Thus, we can delete the 3rd column of the Extrinsic parameter matrix.

2D Camera Calibration

$$H = (h_1, h_2, h_3) = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}_{(r_1, r_2, t)}$$

$$(h_1, h_2, h_3) = K(r_1, r_2, t)$$

2D Camera Calibration

$$H = (h_1, h_2, h_3) = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}_{(r_1, r_2, t)}$$

$$(h_1, h_2, h_3) = K(r_1, r_2, t)$$

$$r_1 = K^{-1}h_1, r_2 = K^{-1}h_2$$

2D Camera Calibration

Note that (r_1, r_2, r_3) form an orthonormal basis, thus: $r_1^T r_2 = 0, \|r_1\| = \|r_2\| = 1.$

$$H = (h_1, h_2, h_3) = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}_{(r_1, r_2, t)}$$

$$(h_1, h_2, h_3) = K(r_1, r_2, t)$$

$$r_1 = K^{-1} h_1, r_2 = K^{-1} h_2$$

2D Camera Calibration

Note that (r_1, r_2, r_3) form an orthonormal basis, thus: $r_1^T r_2 = 0, \|r_1\| = \|r_2\| = 1.$

$$H = (h_1, h_2, h_3) = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}_{(r_1, r_2, t)}$$

$$(h_1, h_2, h_3) = K(r_1, r_2, t)$$

$$r_1 = K^{-1} h_1, r_2 = K^{-1} h_2$$

$r_1^T r_2 = 0$ ⇒

$$h_1^T K^{-T} K^{-1} h_2 = 0$$

2D Camera Calibration

Note that (r_1, r_2, r_3) form an orthonormal basis, thus: $r_1^T r_2 = 0, \|r_1\| = \|r_2\| = 1.$

$$H = (h_1, h_2, h_3) = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}_{(r_1, r_2, t)}$$

$$(h_1, h_2, h_3) = K(r_1, r_2, t)$$

$$r_1 = K^{-1}h_1, r_2 = K^{-1}h_2$$

$$h_1^T K^{-T} K^{-1} h_2 = 0$$

$\|r_1\| = \|r_2\| = 1$ ⇒

$$h_1^T K^{-T} K^{-1} h_1 = h_2 K^{-T} K^{-1} h_2$$

2D Camera Calibration

Note that (r_1, r_2, r_3) form an orthonormal basis, thus: $r_1^T r_2 = 0, \|r_1\| = \|r_2\| = 1.$

$$H = (h_1, h_2, h_3) = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}_{(r_1, r_2, t)}$$

$$(h_1, h_2, h_3) = K(r_1, r_2, t)$$

$$r_1 = K^{-1}h_1, r_2 = K^{-1}h_2$$

$$h_1^T K^{-T} K^{-1} h_2 = 0$$

$$h_1^T K^{-T} K^{-1} h_1 = h_2 K^{-T} K^{-1} h_2$$

$$h_1^T K^{-T} K^{-1} h_1 - h_2 K^{-T} K^{-1} h_2 = 0$$

2D Camera Calibration

Note that (r_1, r_2, r_3) form an orthonormal basis, thus: $r_1^T r_2 = 0, \|r_1\| = \|r_2\| = 1.$

$$H = (h_1, h_2, h_3) = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}_{(r_1, r_2, t)}$$

$$(h_1, h_2, h_3) = K(r_1, r_2, t)$$

$$r_1 = K^{-1} h_1, r_2 = K^{-1} h_2$$

$$h_1^T K^{-T} K^{-1} h_2 = 0$$

$$h_1^T K^{-T} K^{-1} h_1 = h_2 K^{-T} K^{-1} h_2$$

$$h_1^T K^{-T} K^{-1} h_1 - h_2 K^{-T} K^{-1} h_2 = 0$$

2D Camera Calibration

$$h_1^T K^{-T} K^{-1} h_2 = 0 \quad (1)$$

$$h_1^T K^{-T} K^{-1} h_1 - h_2 K^{-T} K^{-1} h_2 = 0 \quad (2)$$

- $B := K^{-T} K^{-1}$ is symmetric and positive definite.

2D Camera Calibration

$$h_1^T K^{-T} K^{-1} h_2 = 0 \quad (1)$$

$$h_1^T K^{-T} K^{-1} h_1 - h_2 K^{-T} K^{-1} h_2 = 0 \quad (2)$$

- $B := K^{-T}K^{-1}$ is symmetric and positive definite.
- Thus: $B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$

Note: K can be calculated from B using Cholesky factorization.

2D Camera Calibration

$$h_1^T K^{-T} K^{-1} h_2 = 0 \quad (1)$$

$$h_1^T K^{-T} K^{-1} h_1 - h_2 K^{-T} K^{-1} h_2 = 0 \quad (2)$$

- $B := K^{-T} K^{-1}$ is symmetric and positive definite.
- Thus: $B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$

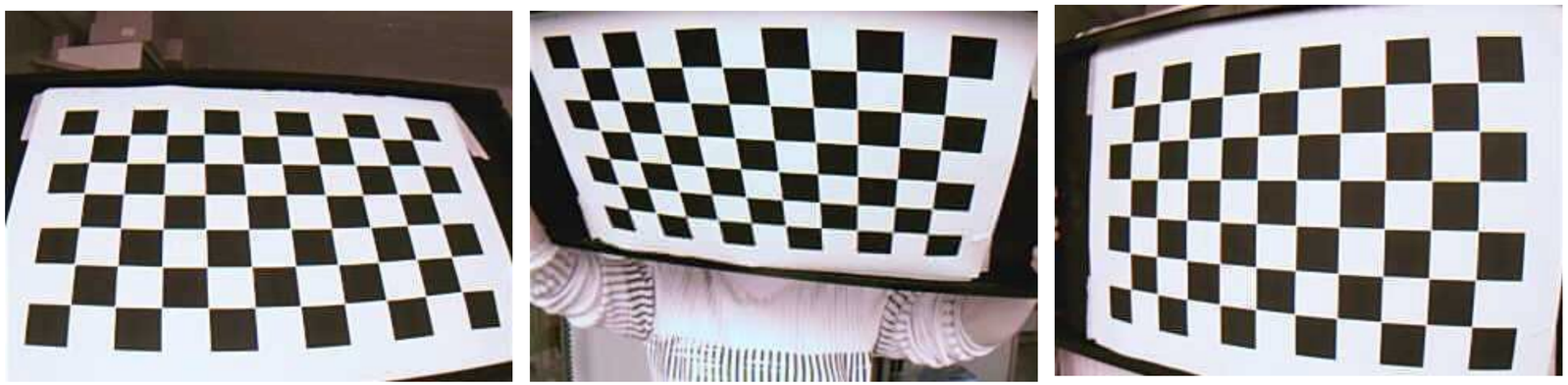
Note: K can be calculated from B using Cholesky factorization.

- define: $b = (b_{11}, b_{12}, b_{13}, b_{22}, b_{23}, b_{33}) \quad (3)$
- Reording of (1)-(3) leads to the system of the final equations :

$$Vb = 0$$

Direct Linear Transformation

- Each plane gives us two equations
- Since B has 6 degrees of freedom, we need at least 3 different views of a plane.



- We need at least 4 points per plane.

Direct Linear Transformation

- Real measurements are corrupted with noise

- Find a solution that minimizes the least-squares error

$$b = \arg\min_{b} Vb$$

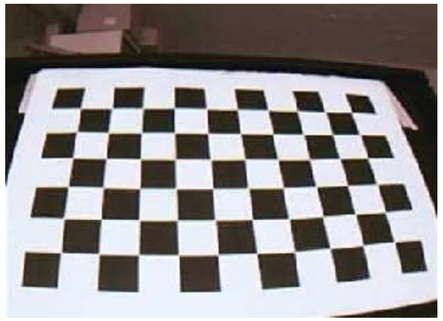
Non-Linear Optimization

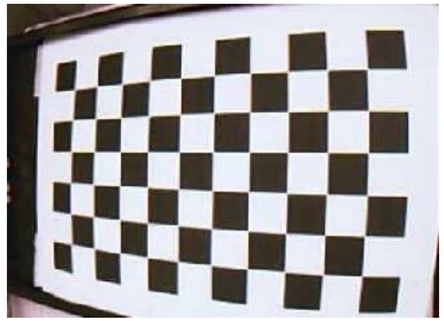
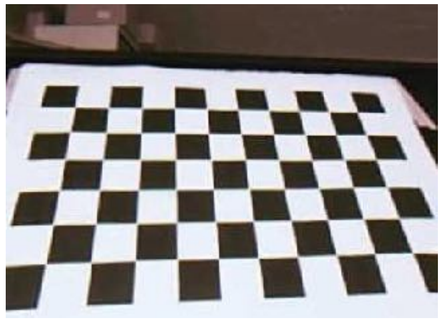
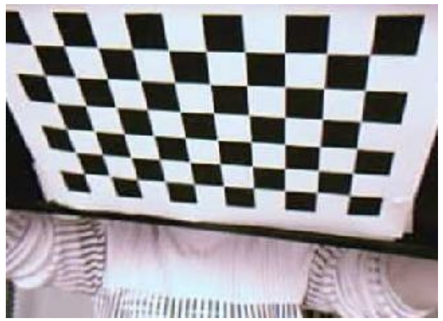
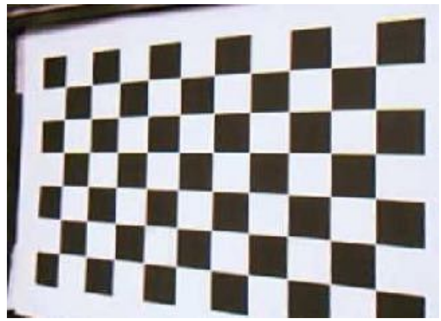
- Lens distortion can be calculated by minimizing a non-linear function

$$\min_{K,k,R_i,t_i} \sum_i \sum_j \|x_{i,j} - \hat{x}(K, k, t_i; X_{ij})\|^2$$

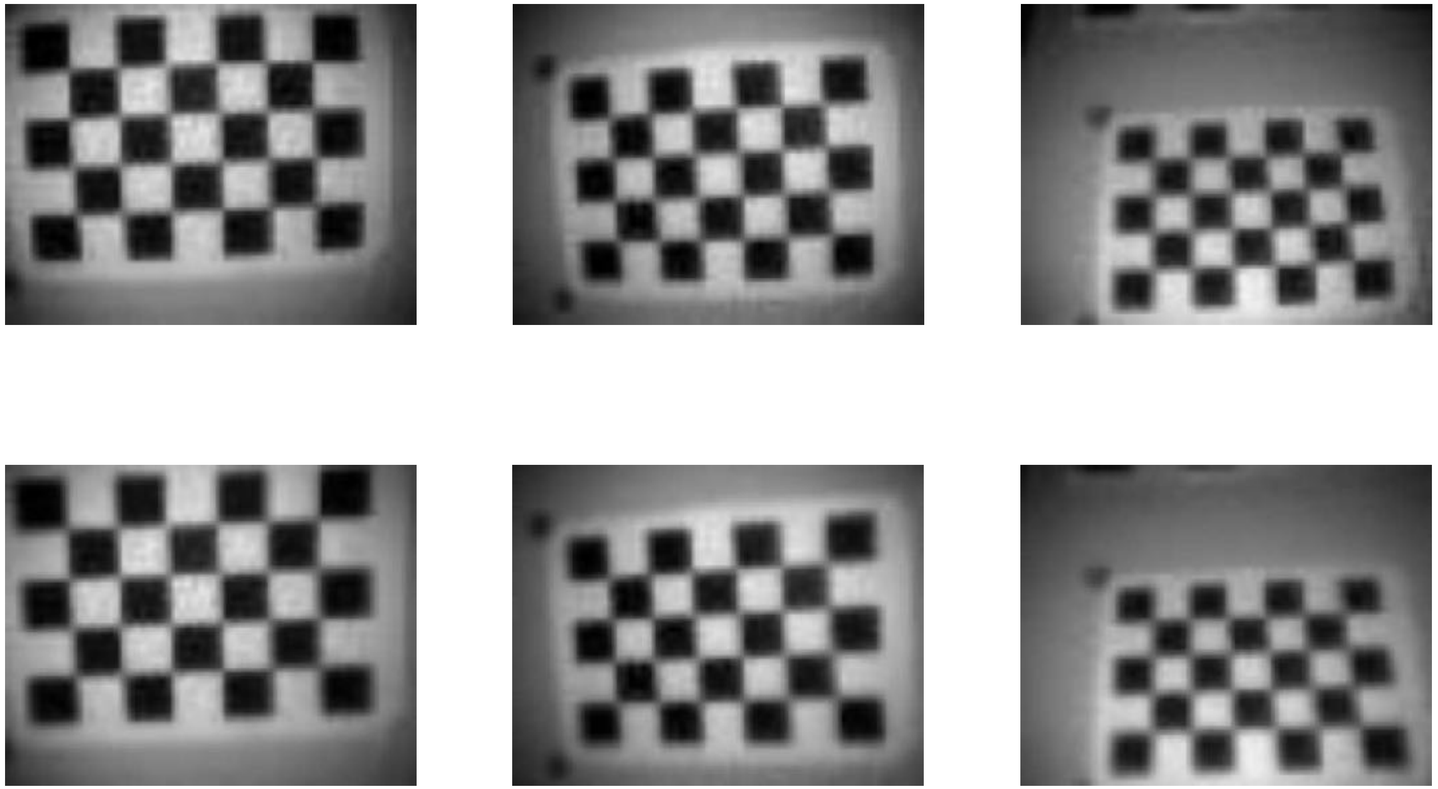
- Estimation of k using non-linear optimization techniques (e.g., Levenberg-Marquard)
- The parameters obtained by the linear function are used as starting values.

Results: Webcam



Results: ToF-Camera



Summary

- Pinhole Camera Model
- Non-linear model for lens distortion
- Approach to 2D Calibration that
  - accurately determines the model parameters and is easy to realize

Finding lane lines for self-driving cars

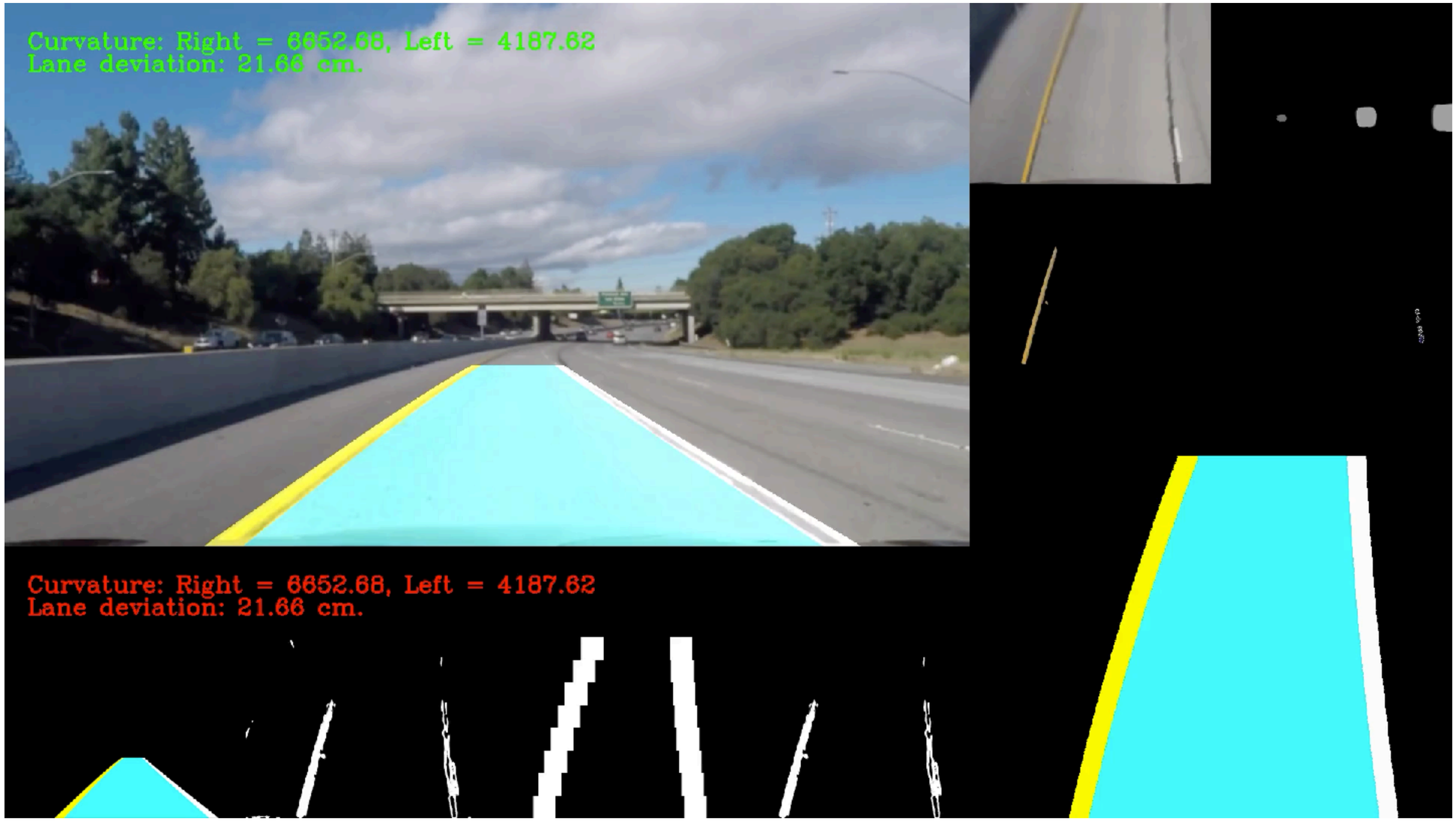
Reference from Udacity

NEWS Lab
嵌入式系統暨無線網路實驗室

Finding lane lines is perception and decision


Radius of curvature: 0501m
0.11m Right of center


Curvature: Right = -13777.09, Left = -7113.53
Lane deviation: 36.47 cm.

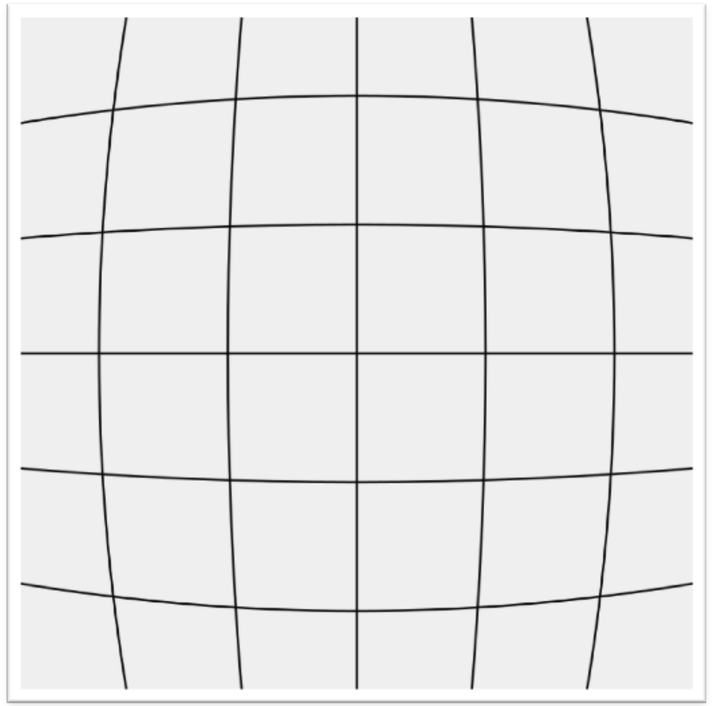

Curvature: Right = 6652.68, Left = 4187.62
Lane deviation: 21.66 cm.
Curvature: Right = 6652.68, Left = 4187.62
Lane deviation: 21.66 cm.

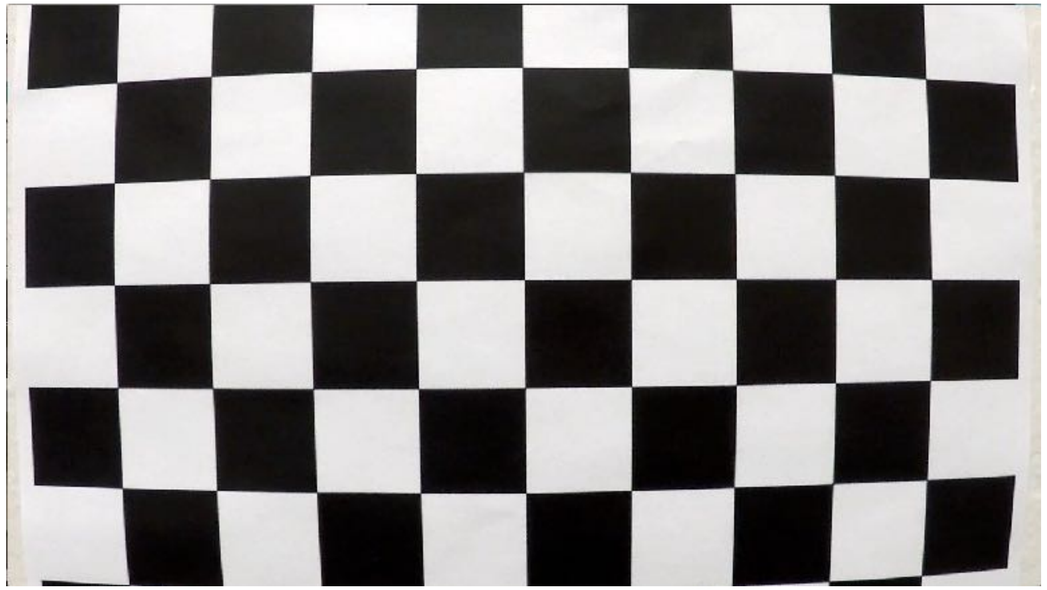
Finding Lane Lines from Images

1. Calibrate camera to remove distortion
2. Warping images
3. Finding lane lines
4. Curve fitting
5. Overlap Results on Image

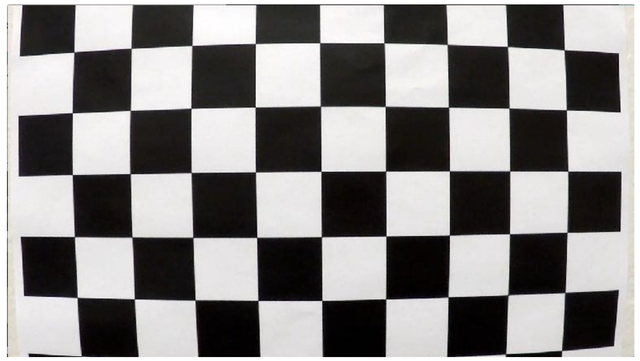
Calibrate camera to remove distortion

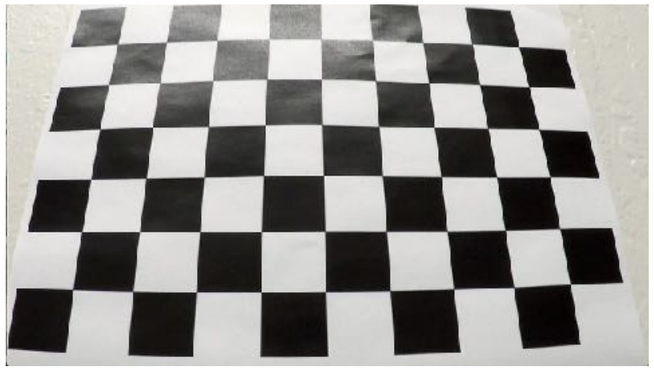
Camera Lens Distortion

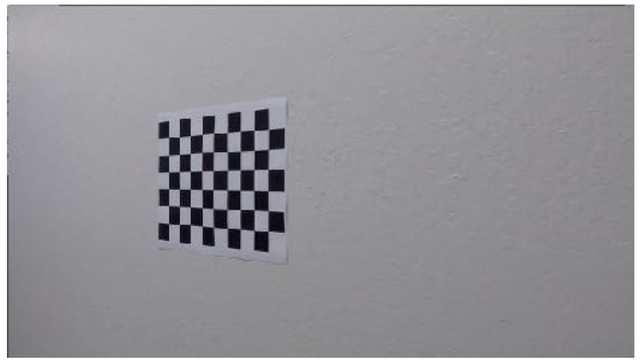


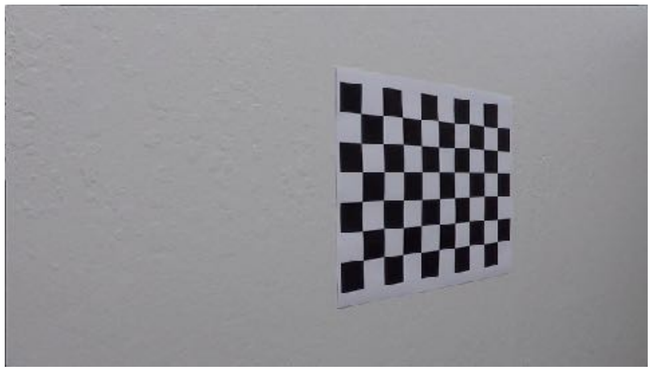


OpenCV helps us correct for these distortions

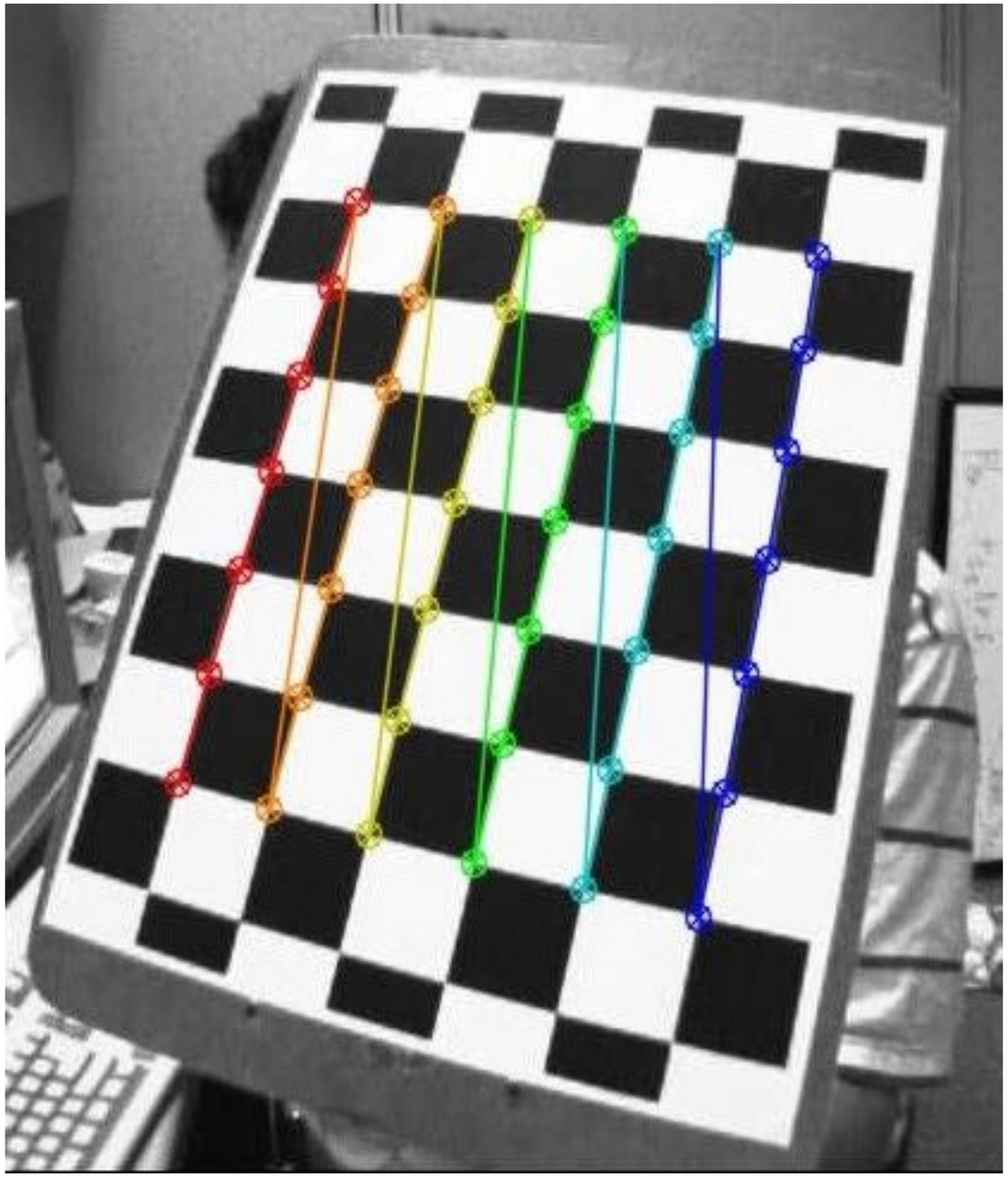








```
ret, corners = cv2.findChessboardCorners(gray, (9,6), None)
```



OpenCV: Locates the chessboard corners

Python: cv2.findChessboardCorners(image, patternSize[, corners[, flags]]) → retval, corners

Parameters:

- image – Source chessboard view. It must be an 8-bit grayscale or color image.
- patternSize – Number of inner corners per a chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows)).
- corners – Output array of detected corners.
- flags –Various operation flags that can be zero or a combination of the following values:
  - CALIB_CB_ADAPTIVE_THRESH Use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
  - CALIB_CB_NORMALIZE_IMAGE Normalize the image gamma with equalizeHist() before applying fixed or adaptive thresholding.
  - CALIB_CB_FILTER_QUADS Use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads extracted at the contour retrieval stage.
  - CALIB_CB_FAST_CHECK Run a fast check on the image that looks for chessboard corners, and shortcut the call if none is found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.

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

# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*7,3), np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('*.jpg')

for fname in images:
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv.findChessboardCorners(gray, (7,6), None)

    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)

        corners2 = cv.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
        imgpoints.append(corners)

        # Draw and display the corners
        cv.drawChessboardCorners(img, (7,6), corners2, ret)
        cv.imshow('img', img)
        cv.waitKey(500)
```

```
# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*7,3), np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('*.jpg')

for fname in images:
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv.findChessboardCorners(gray, (7,6), None)

    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)

        corners2 = cv.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
        imgpoints.append(corners)

        # Draw and display the corners
        cv.drawChessboardCorners(img, (7,6), corners2, ret)
        cv.imshow('img', img)
        cv.waitKey(500)

cv.destroyAllWindows()
```

Calibrate the camera

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

```
ret, mtx, dist, rvecs, tvecs =  cv2.calibrateCamera(
    objpoints,  imgpoints,  gray.shape[::-1],
    None, None
)
```

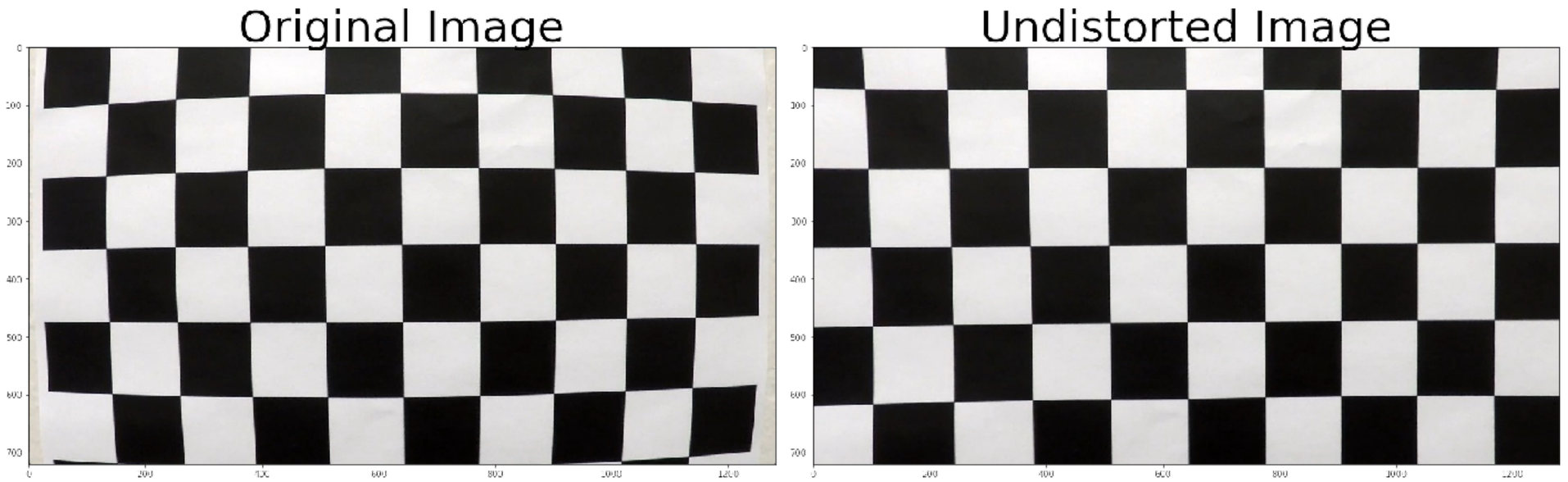
Python: `cv2.calibrateCamera`(objectPoints, imagePoints, imageSize[, cameraMatrix[, distCoeffs[, rvecs[, tvecs[, flags[, criteria]]]]]]) → retval, cameraMatrix, distCoeffs, rvecs, tvecs

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

- return values: cameraMatrix, distCoeffs, rvecs, tvecs
- **obejctPoints**: a vector of vectors of calibration pattern points in the calibration pattern coordinate space
- **imagePoints**: a vector of vectors of the projections of calibration pattern points
- **imageSize**: Size of the image used only to initialize the intrinsic camera matrix.
- **cameraMatrix**: Output 3x3 floating-point camera matrix A
- **distCoeffs**: Output vector of distortion coefficients
- **rvecs** – Output vector of rotation vectors¶

Use the camera calibration to undistort images

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



C++: void `undistort`(InputArray src, OutputArray dst, InputArray cameraMatrix, InputArray distCoeffs, InputArray newCameraMatrix=noArray())

Python: `cv2.undistort`(src, cameraMatrix, distCoeffs[, dst[, newCameraMatrix]]) → dst¶

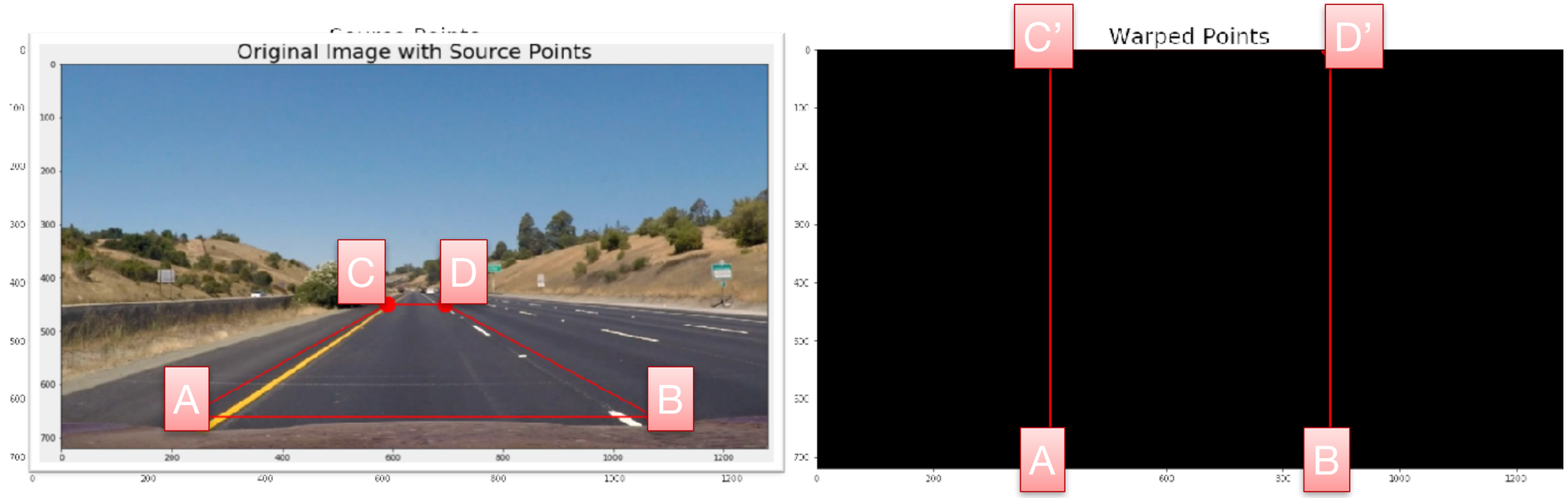
The function transforms an image to compensate radial and tangential lens distortion.

Parameters:

- **src** – Input (distorted) image.
- **dst** – Output (corrected) image that has the same size and type as `src` .
- **cameraMatrix** – Input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
- **distCoeffs** – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **newCameraMatrix** – Camera matrix of the distorted image. By default, it is the same as `cameraMatrix` but you may additionally scale and shift the result by using a different matrix.

Warping images

Need a “birds eye view” for curve fitting



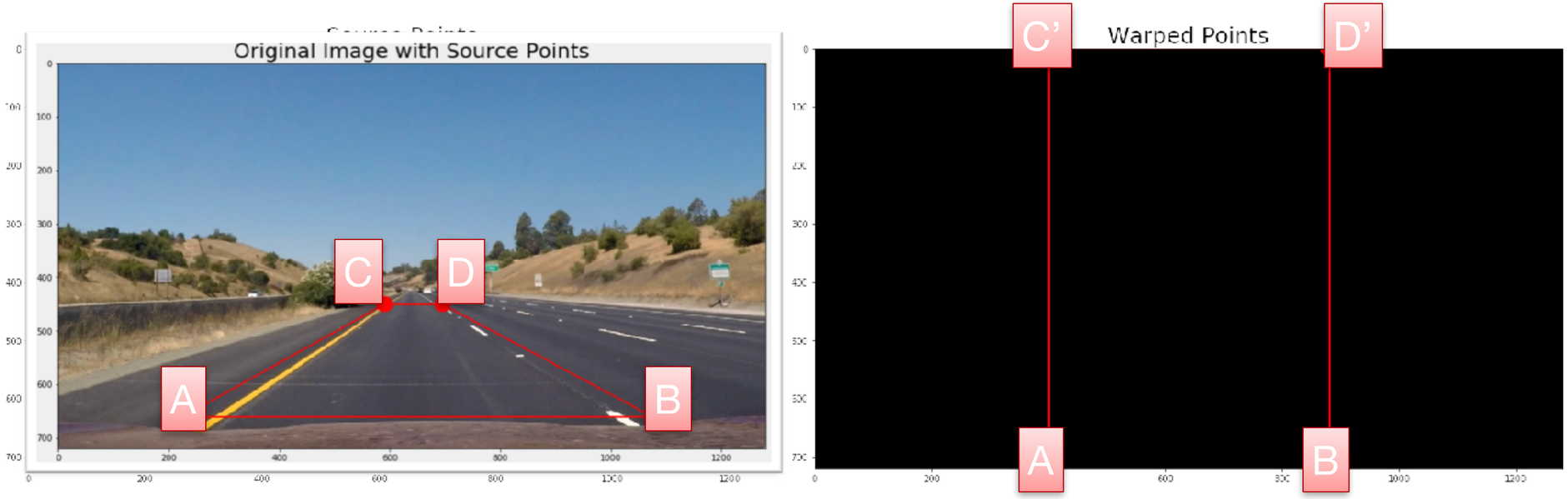
src={cv::Point2f(200, 650), cv::Point2f(1120, 650),

cv::Point2f(580, 450),cv::Point2f(760, 450)}}

dst={cv::Point2f(200, 650), cv::Point2f(1120, 650),

cv::Point2f(200, 0),cv::Point2f(1120, 0)}}

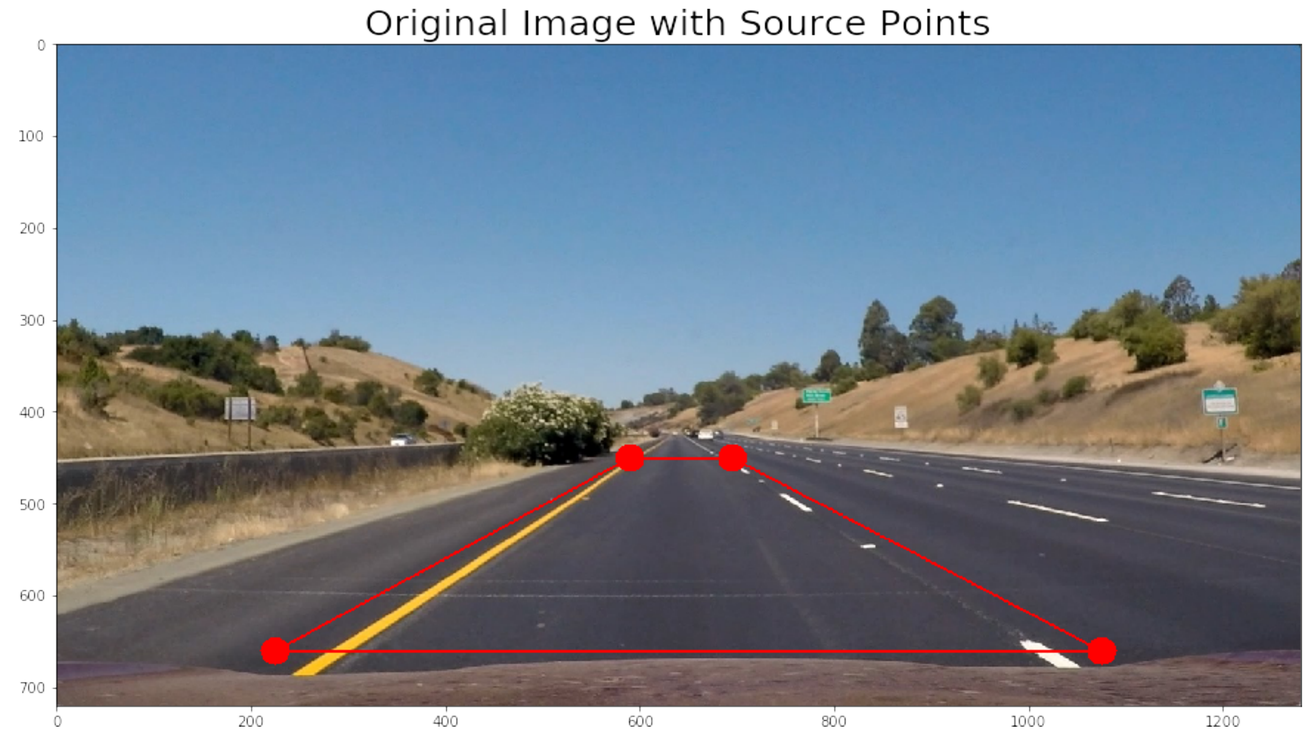
Need a “birds eye view” for curve fitting

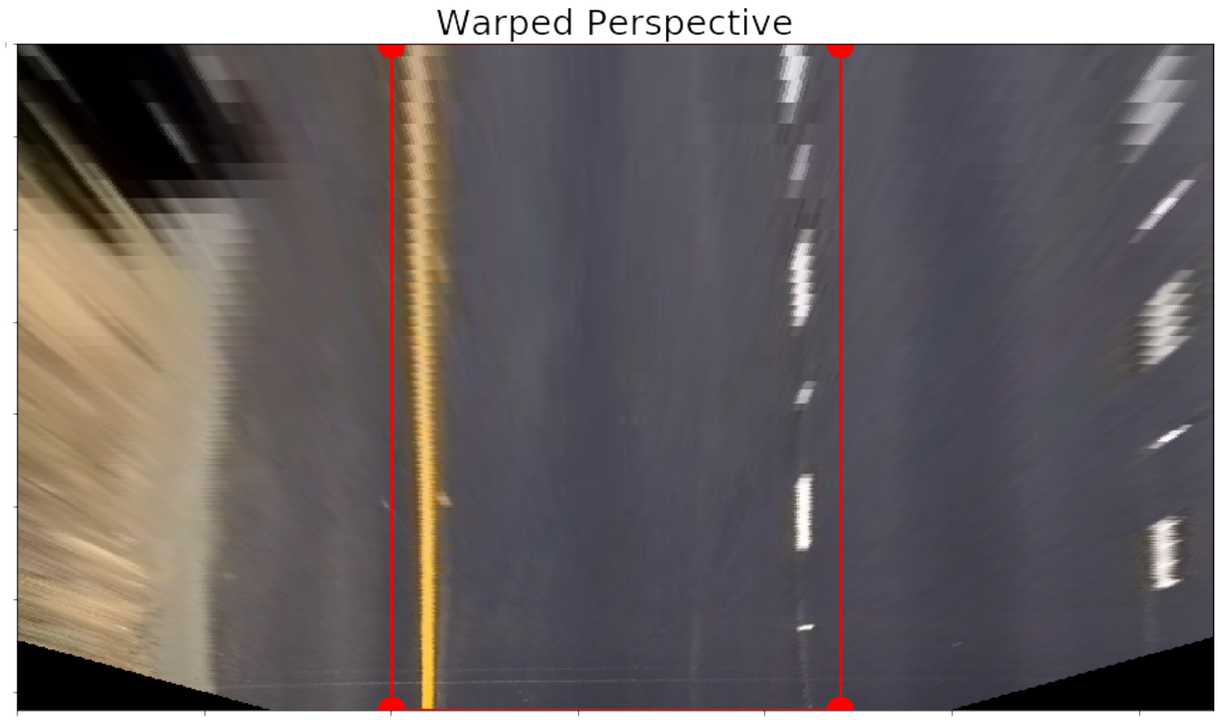


create a transformation matrix

M = cv2.getPerspectiveTransform(src, dst)

```
# apply the transform to the original image warped = cv2.warpPerspective(img, M, img_size)
```

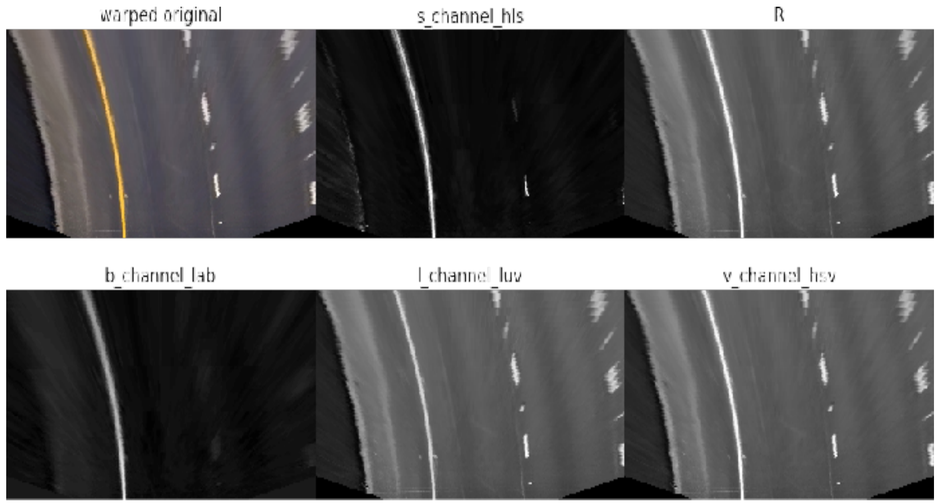


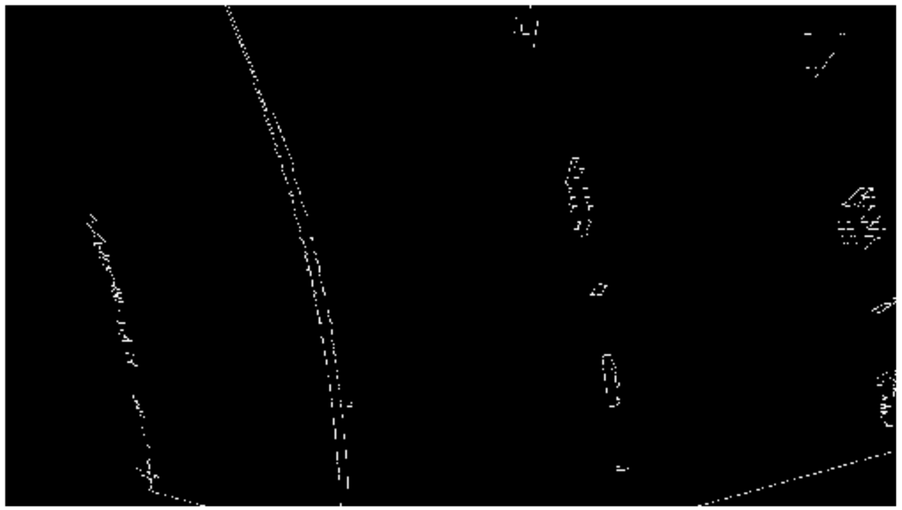


Finding the lane lines

Edge Detection

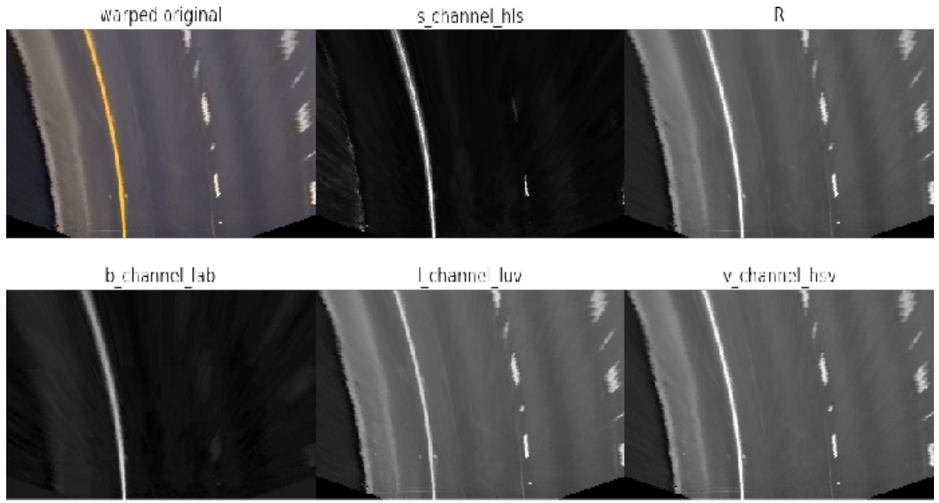
Color Selection





Edge Detection

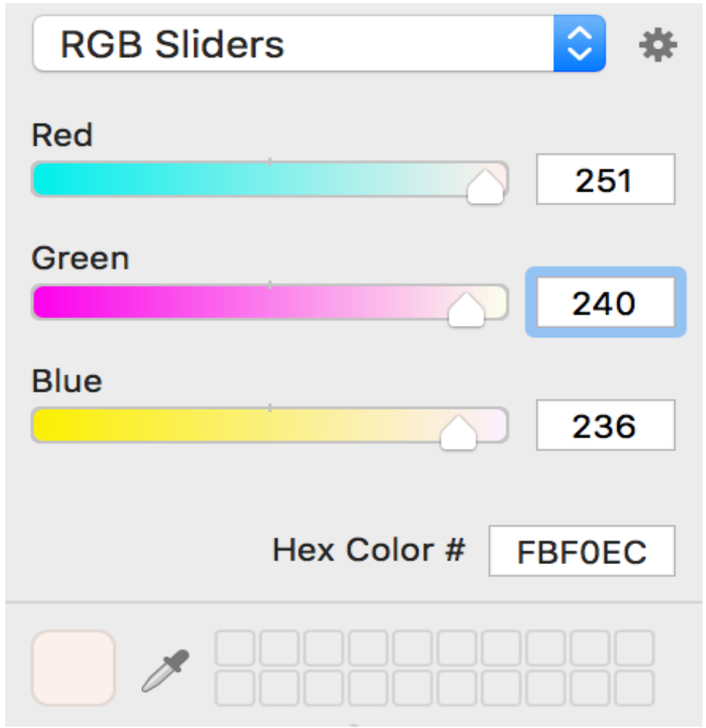
Color Selection

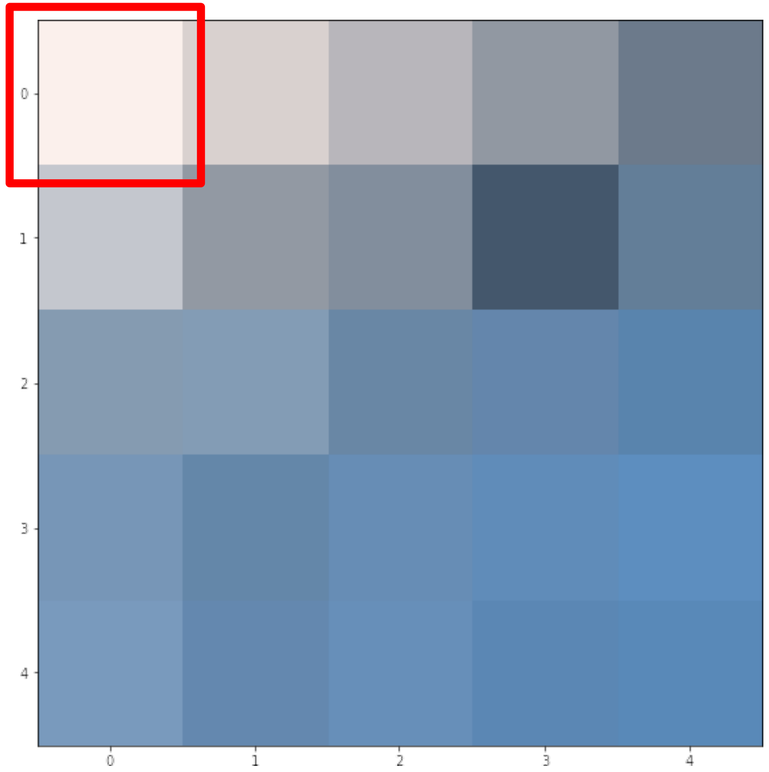


Anatomy of an image

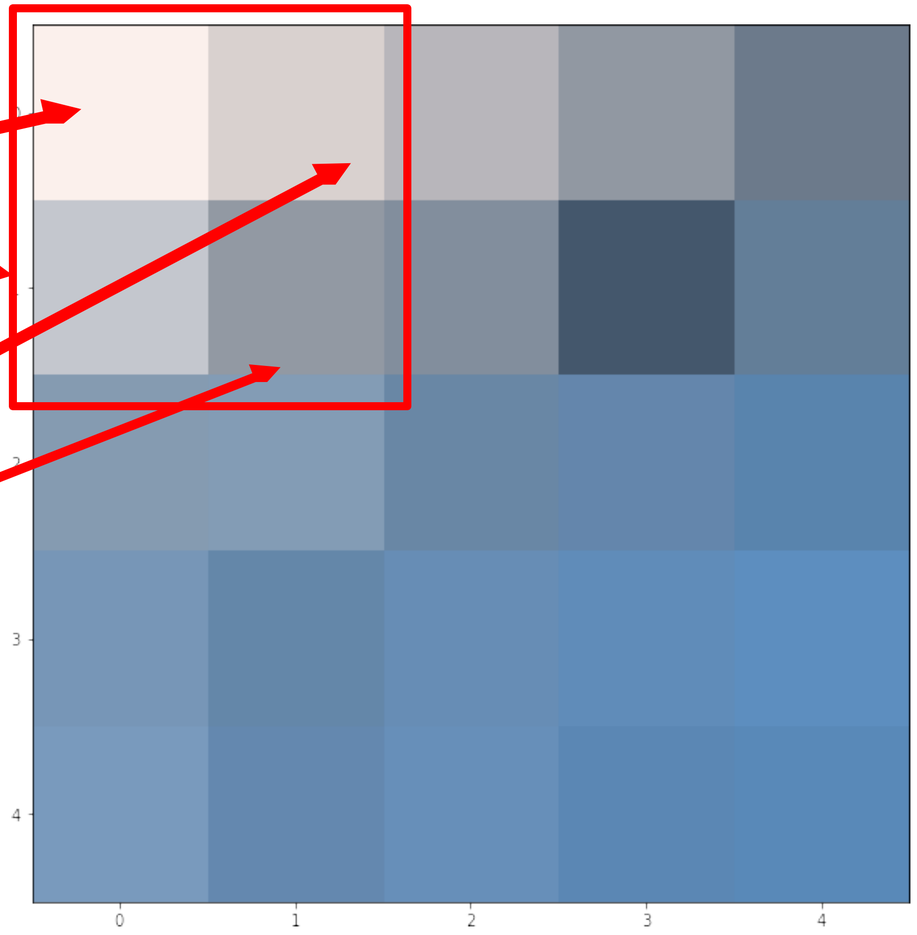
```
> img[0,0,:] # first pixel

array([251, 240, 236])
```

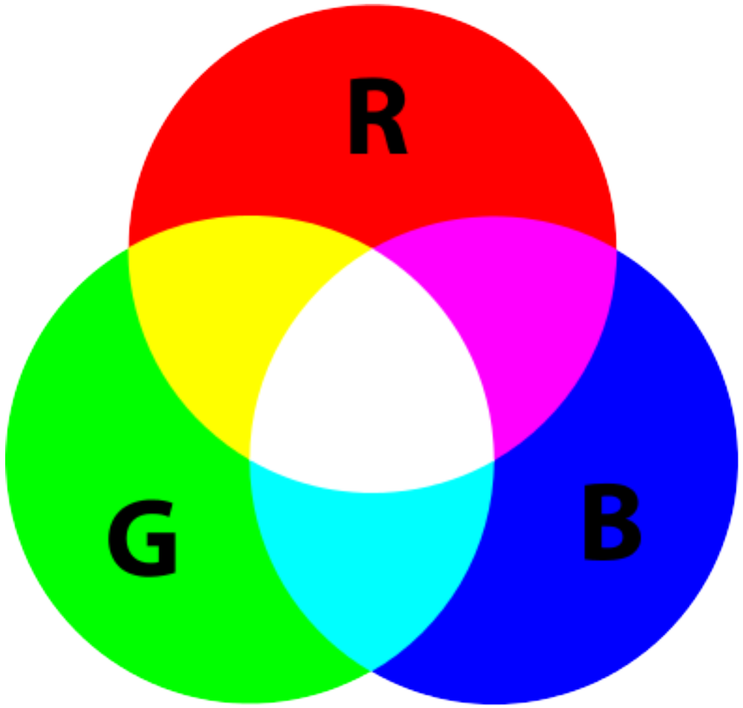
Red Green Blue

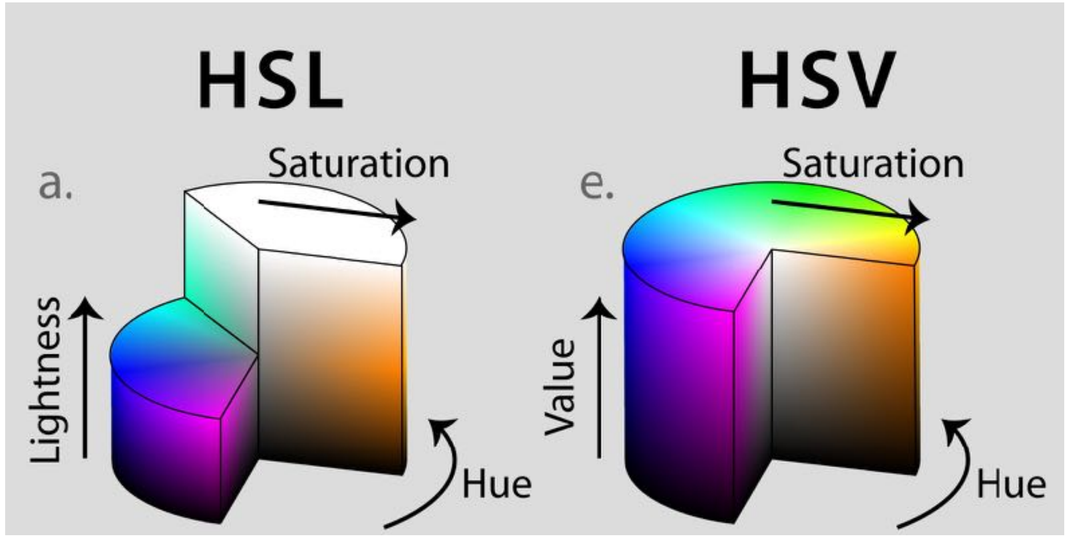


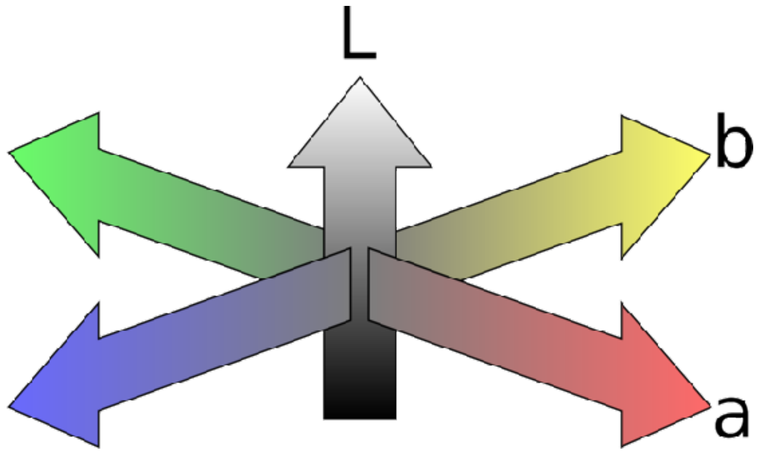


```
> img[0:2,0:2,:]
array([[
[251, 240, 236],
[217, 209, 207]],
[[196, 199, 206],
[146, 153, 163]]],)
```




R
G
B


HSL
HSV
a.
Saturation
Lightness
Hue
e.
Saturation
Value
Hue


L
b
a

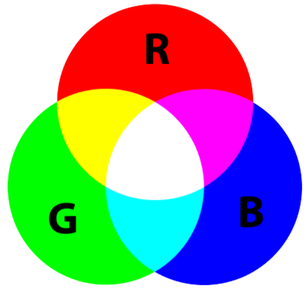
OpenCV makes changing color space easy

```
# convert RGB image to Hue Light Saturation (HLS)
HLS = cv2.cvtColor(RGB, cv2.COLOR_RGB2HLS)
```

```
> HLS[0,0,:]
array([ 8, 244, 166])
```

```
> RGB[0,0,:]
array([251, 240, 236])
```

Isolate each color channel




```
red = img[:,:,0]
green  = img[:,:,1]
blue = img[:,:,2]
```

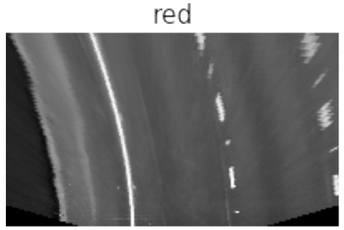
Red Channel

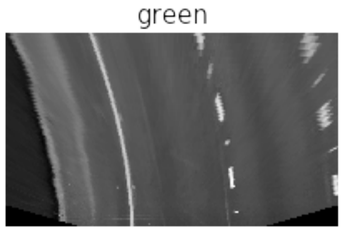
Green Channel

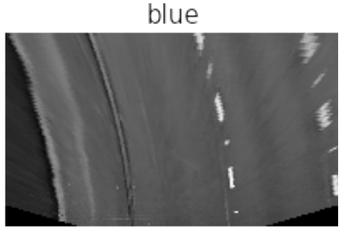
Blue Channel

Yellow Channel

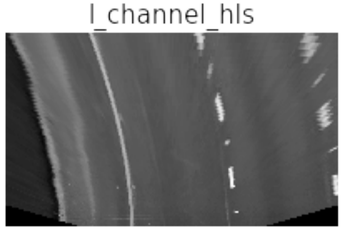
RGB

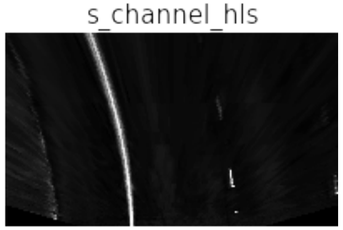




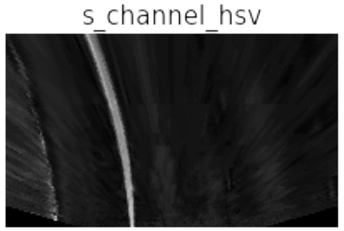


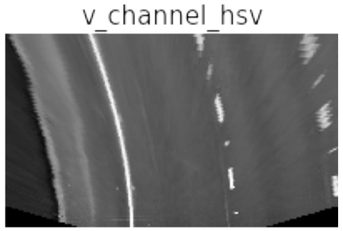
HLS



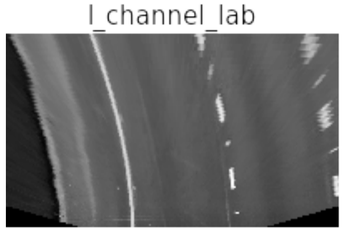


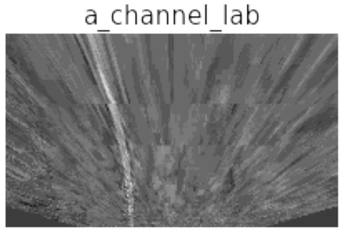
HSV

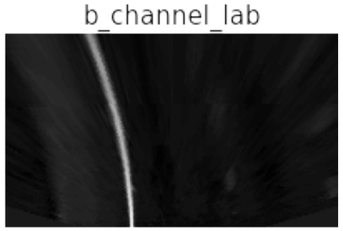




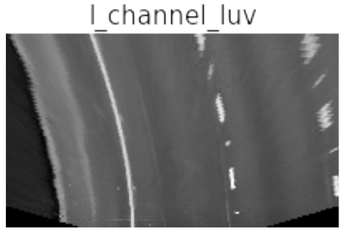
LAB

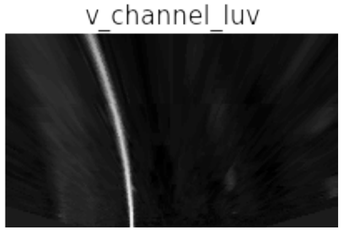






LUV





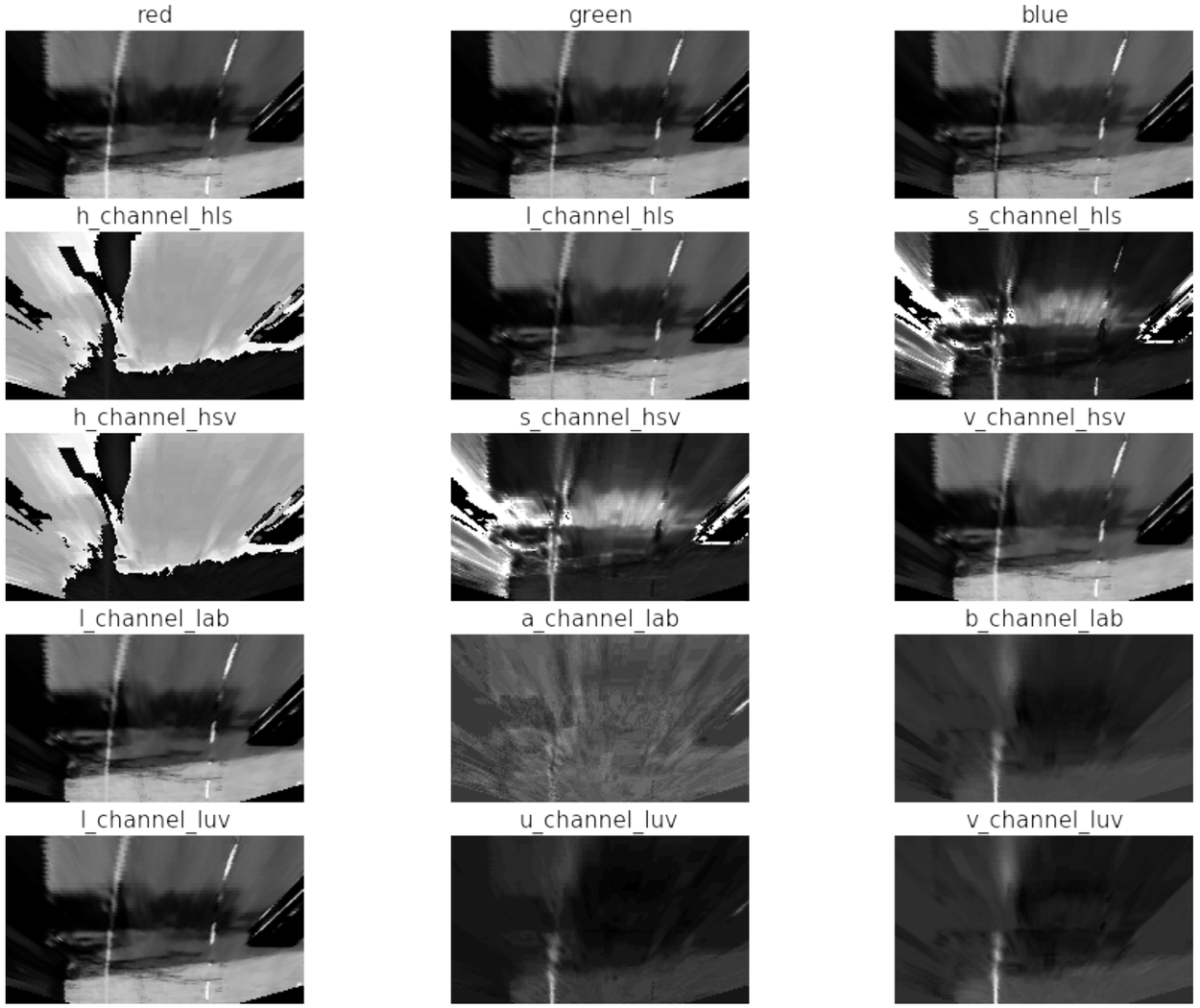
RGB

HLS

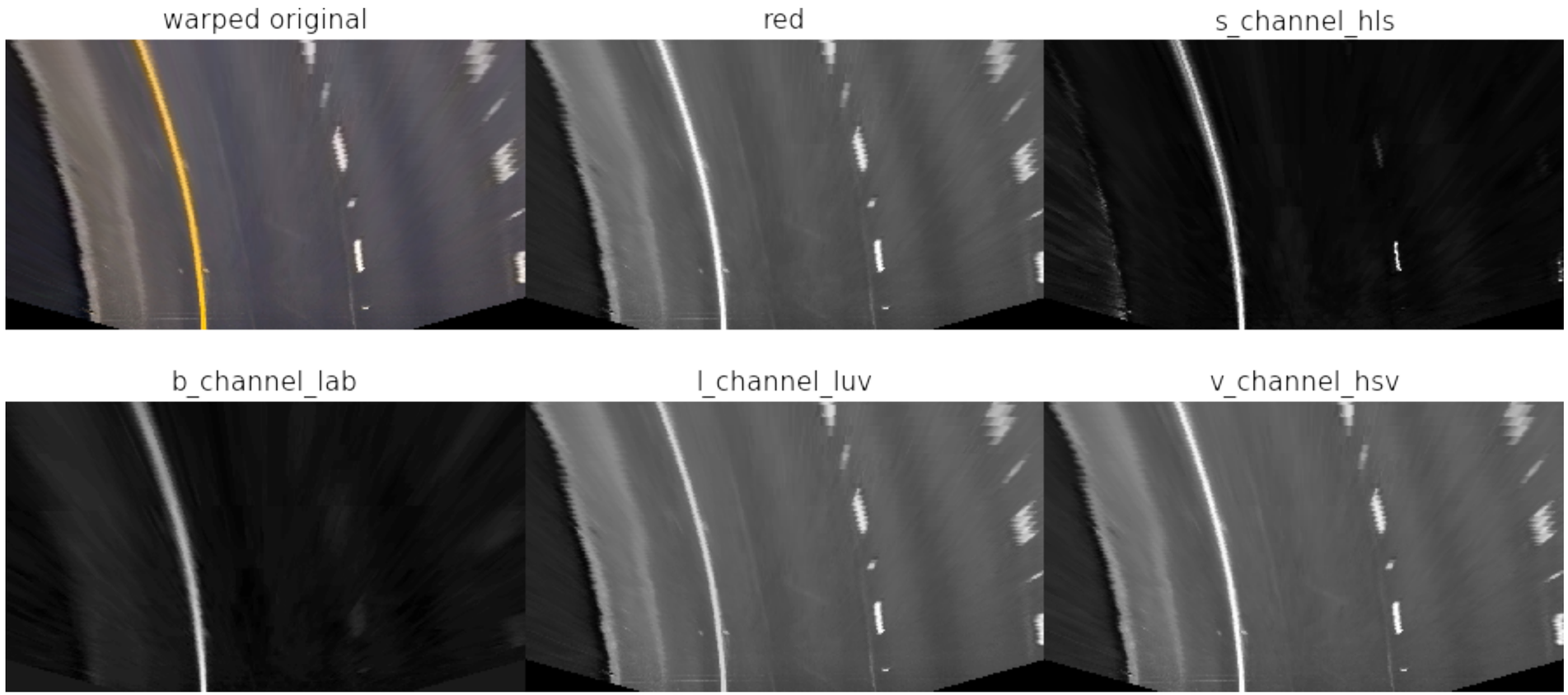
HSV

LAB

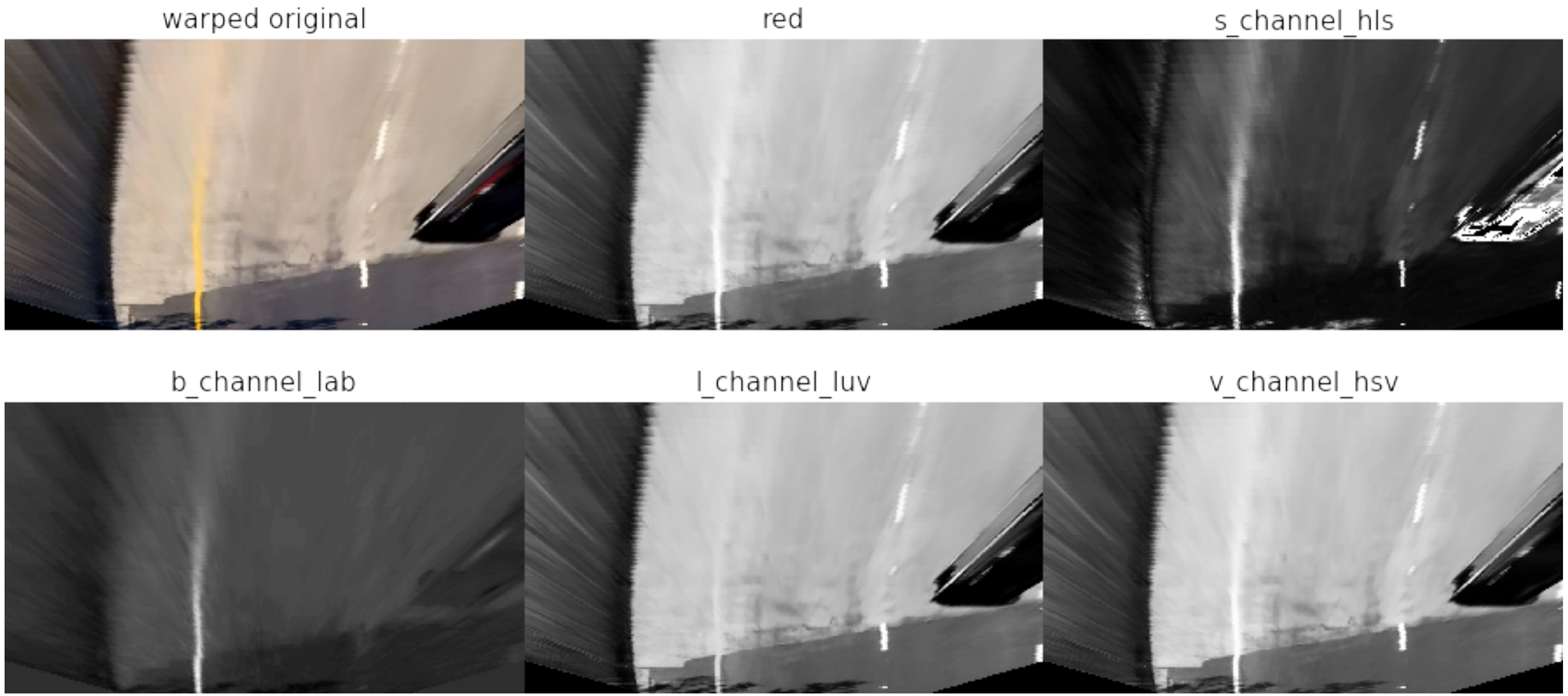
LUV



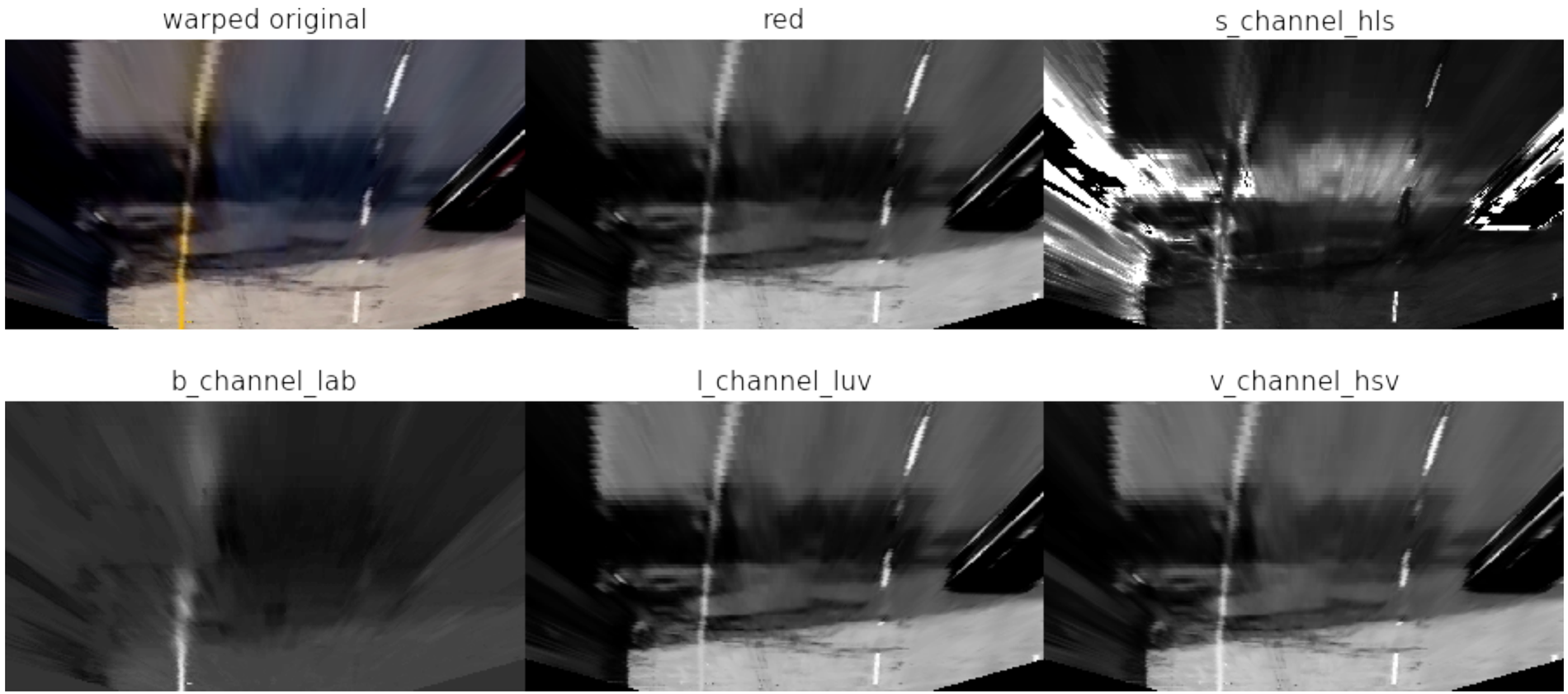
The best ones...



But some images are harder than others

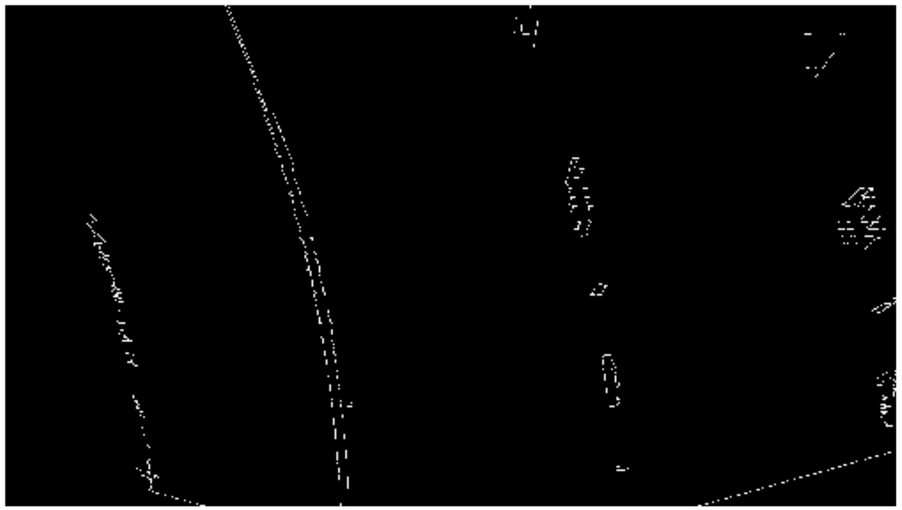


And some are really hard



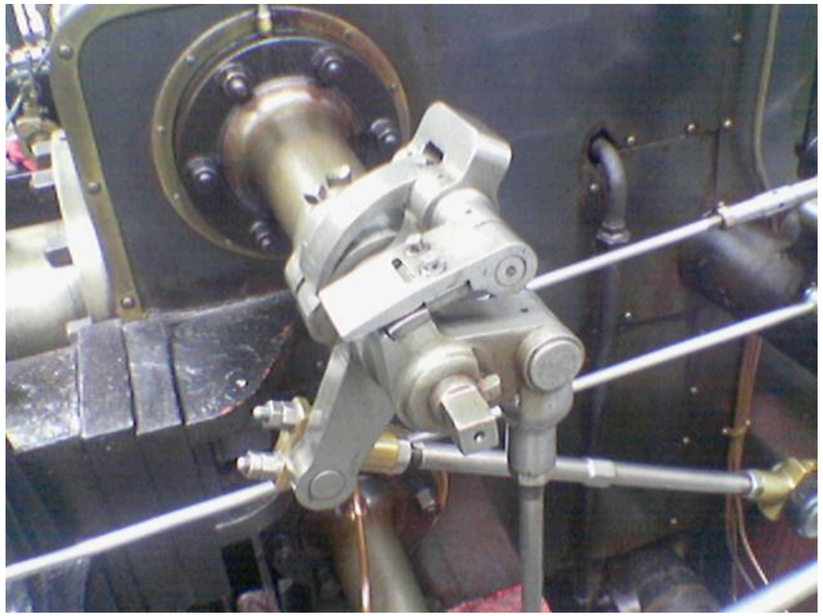
Color Selection

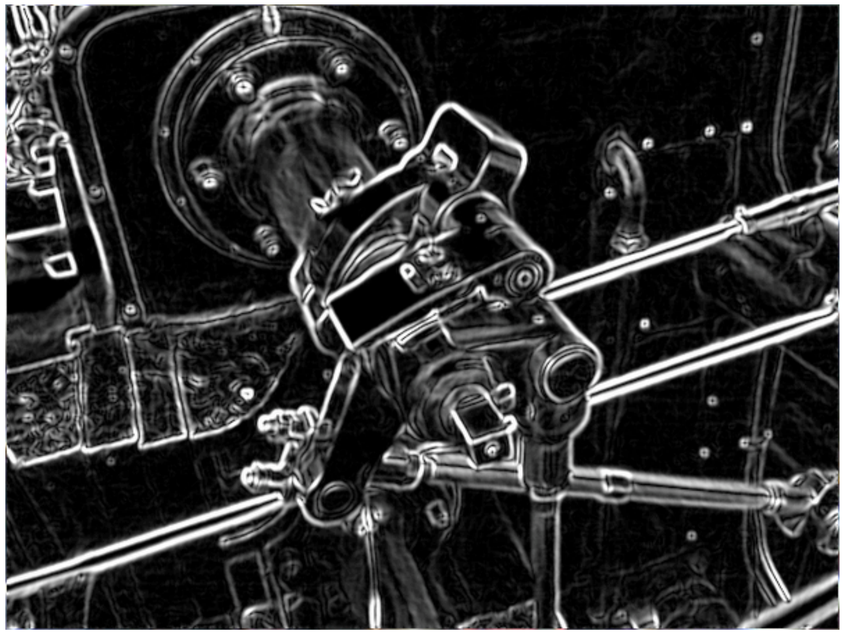
Edge Detection



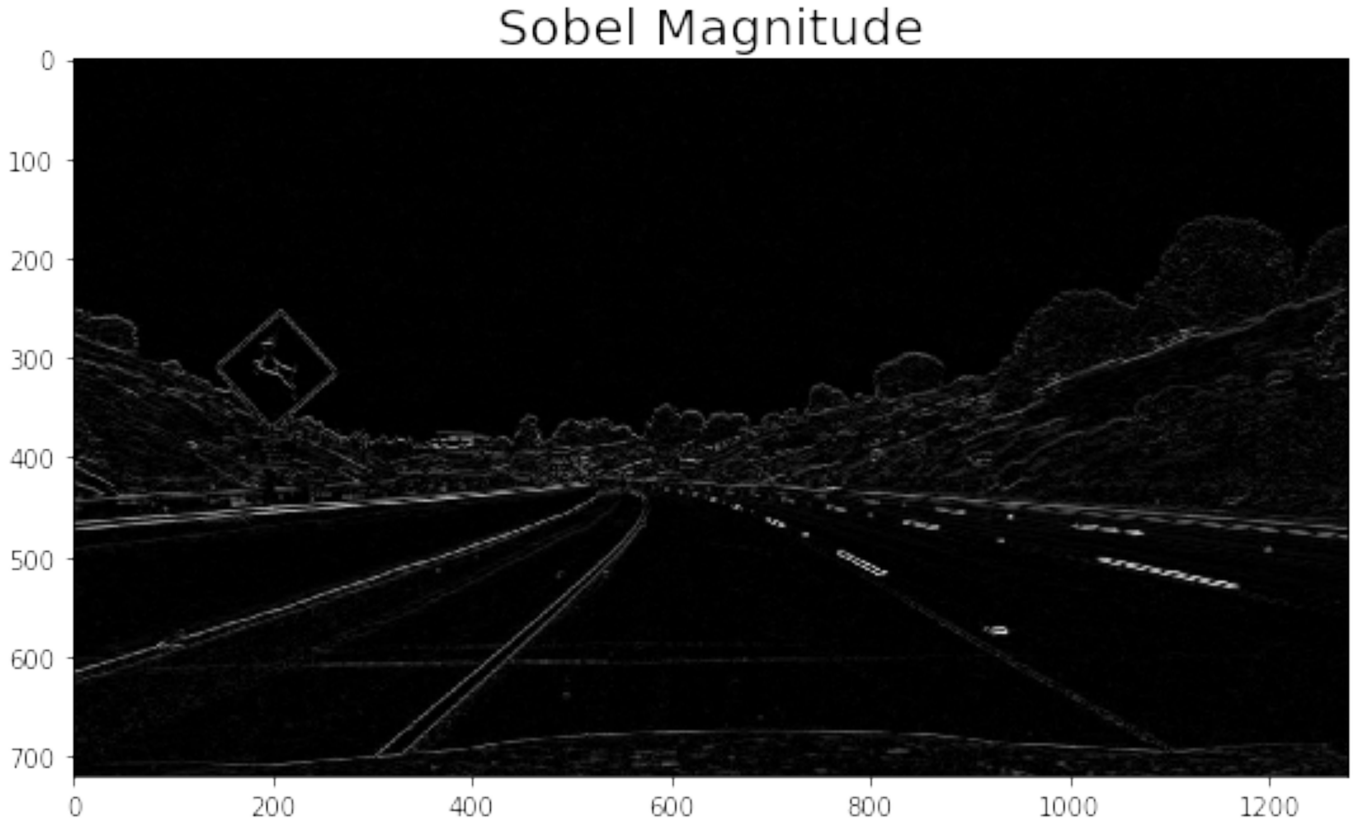
Sobel operator

Sobel operator creates an image emphasizing edges. Technically, it is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function.



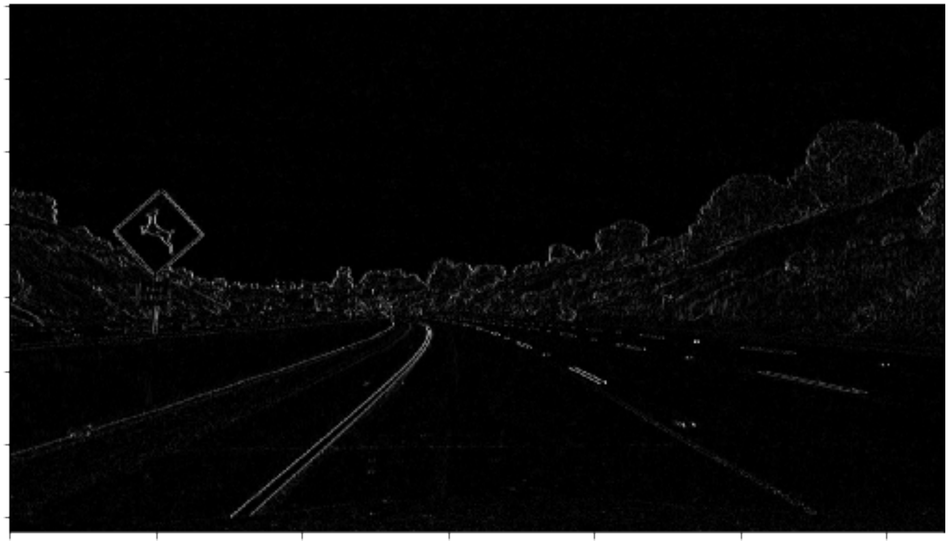


We can use built in OpenCV functions to find edges



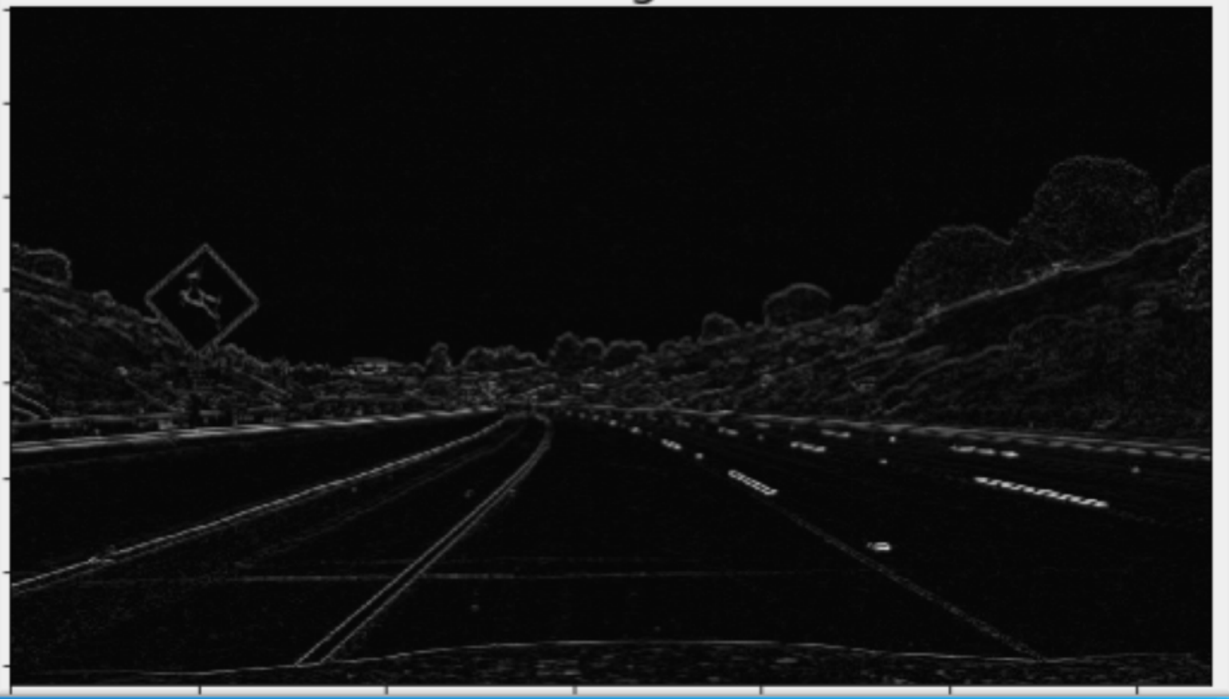
Sobel operator creates an image emphasising edges. Technically, it is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function.

Sobel X



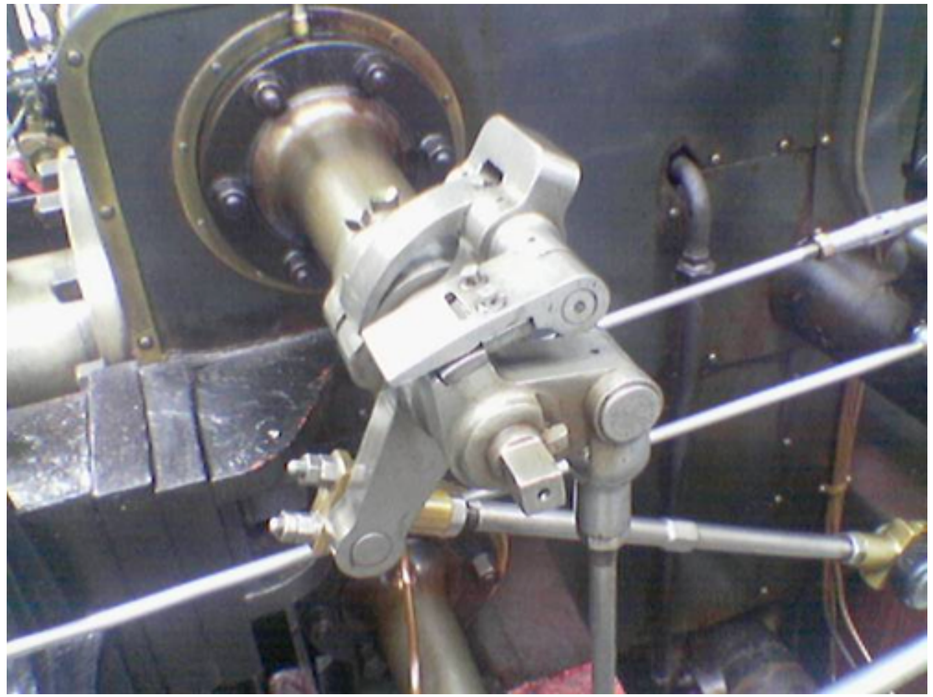
Sobel Y

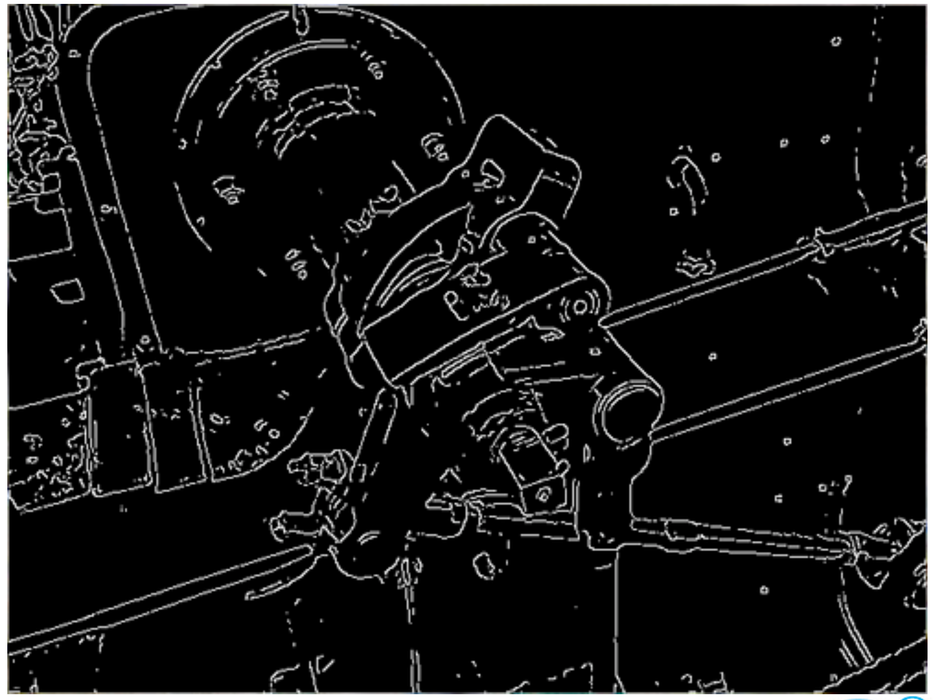
Sobel Magnitude



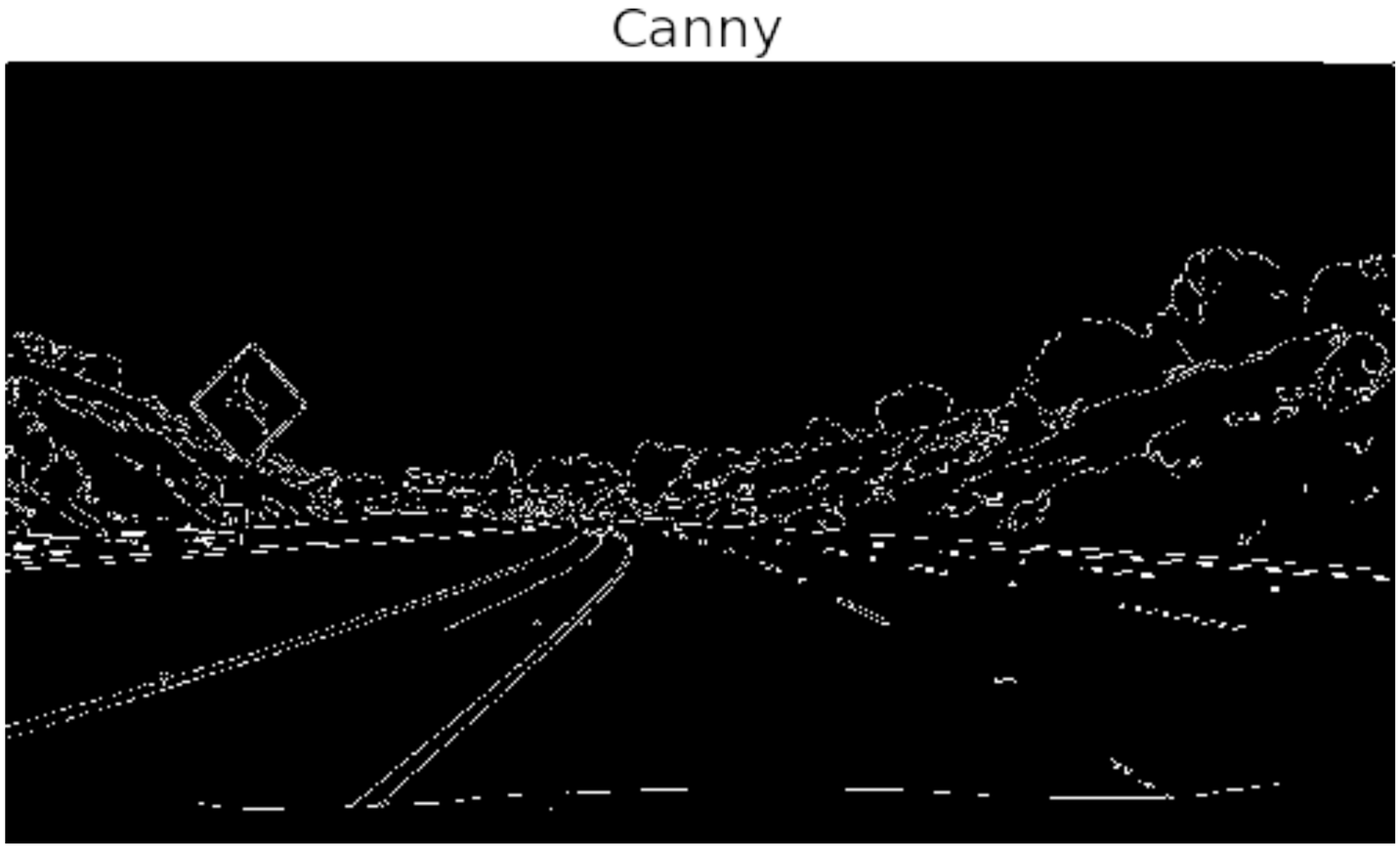
Canny edge detection

Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed.



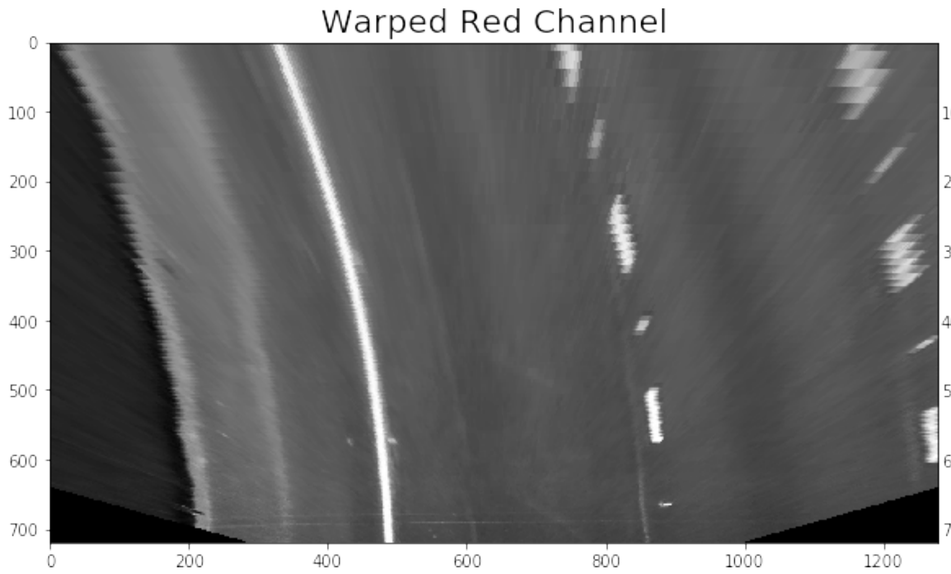


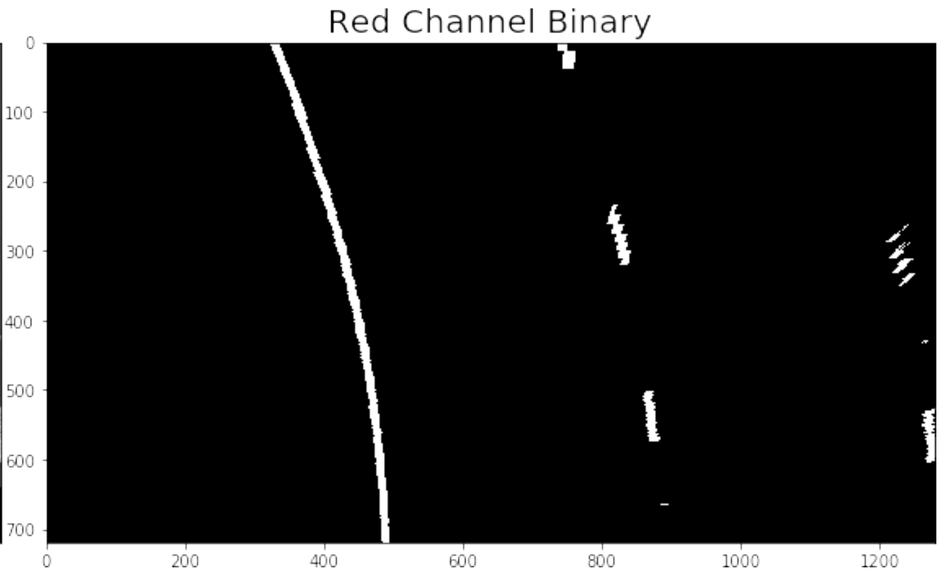
Canny edge detection



Use binary thresholding to isolate “hot” pixels

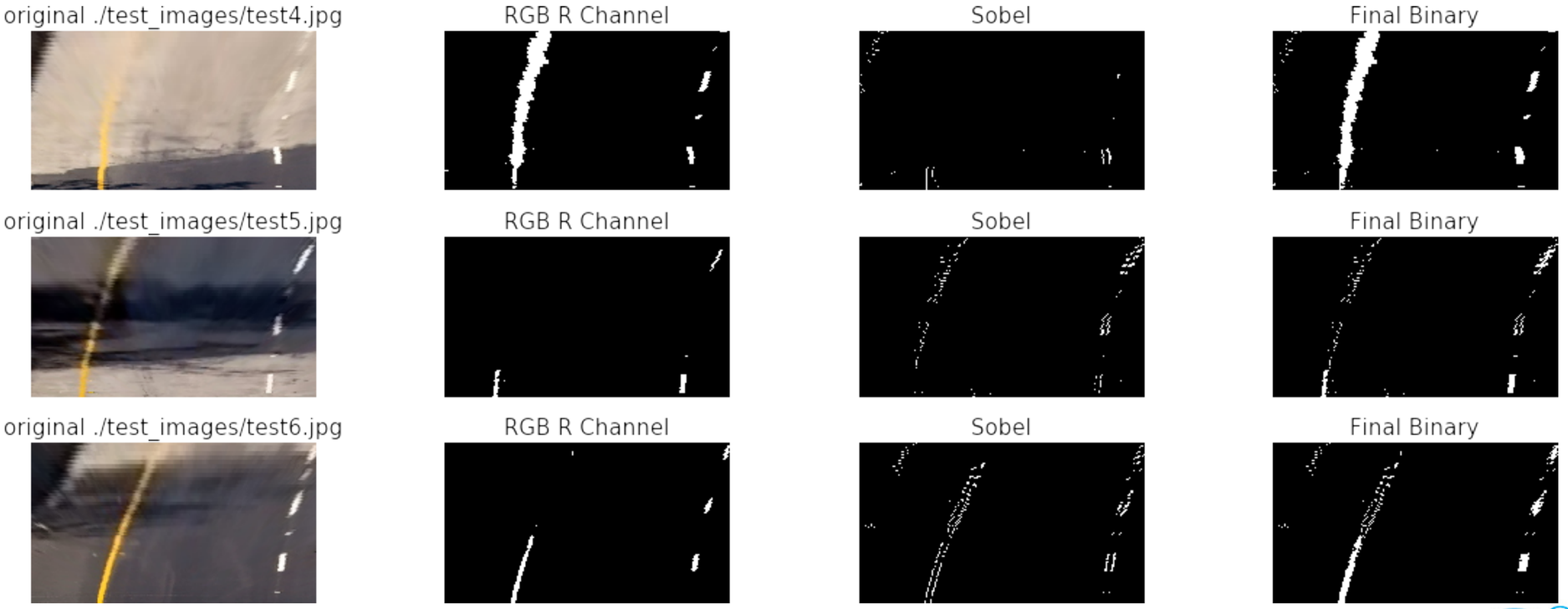
```
channel = warped[:,:,0]  thresh = (200,255)
output = np.zeros_like(channel)
output[(channel >= thresh[0]) & (channel <= thresh[1])] = 1
```





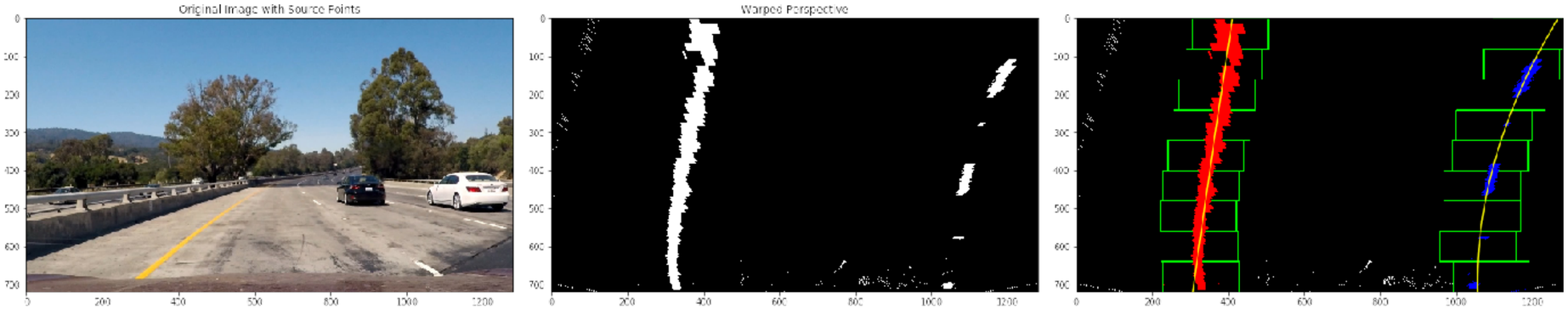
Combine binary threshold images

```
combined_binary = np.zeros_like(red)  combined_binary[ (red == 1) | (sobel == 1) ] = 1
```

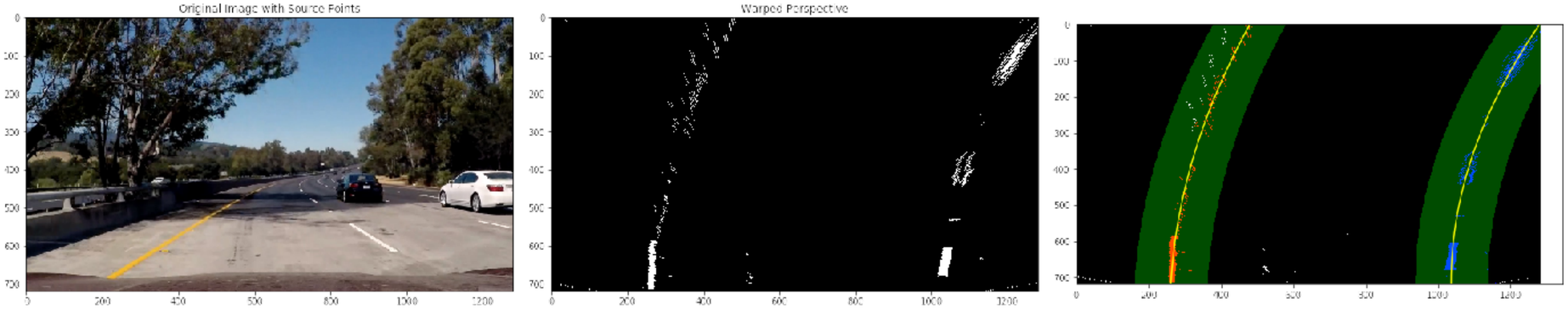


Curve Fitting

Method 1



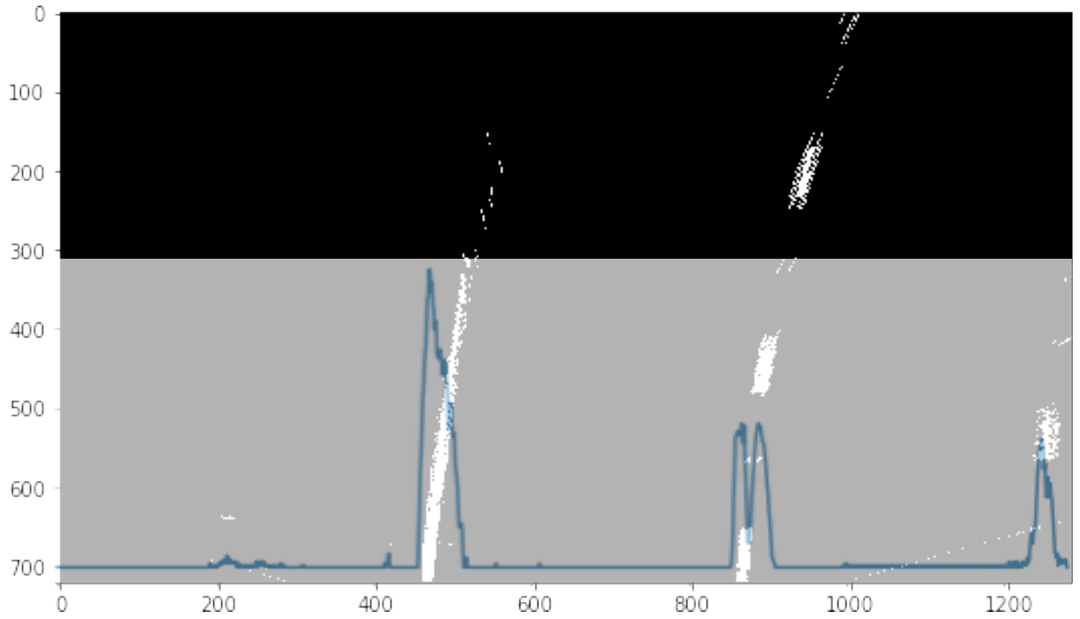
Method 2



Step 1: Find the start of the line

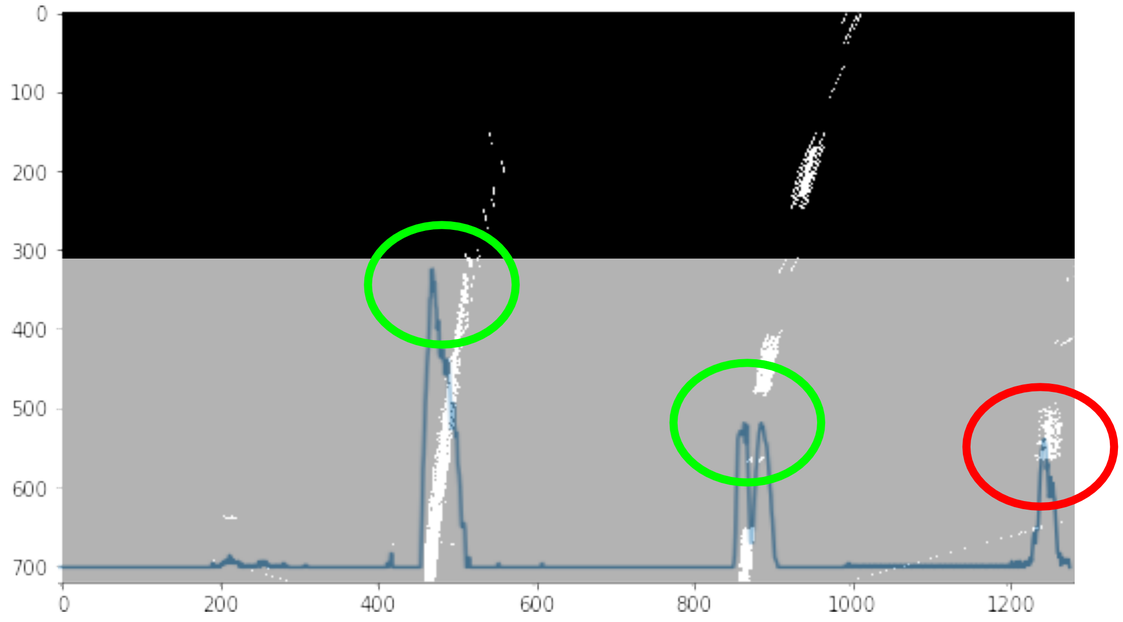
```
import numpy as np

histogram = np.sum(img[img.shape[0]//2:,:], axis=0)
```

Sum white pixels at x location

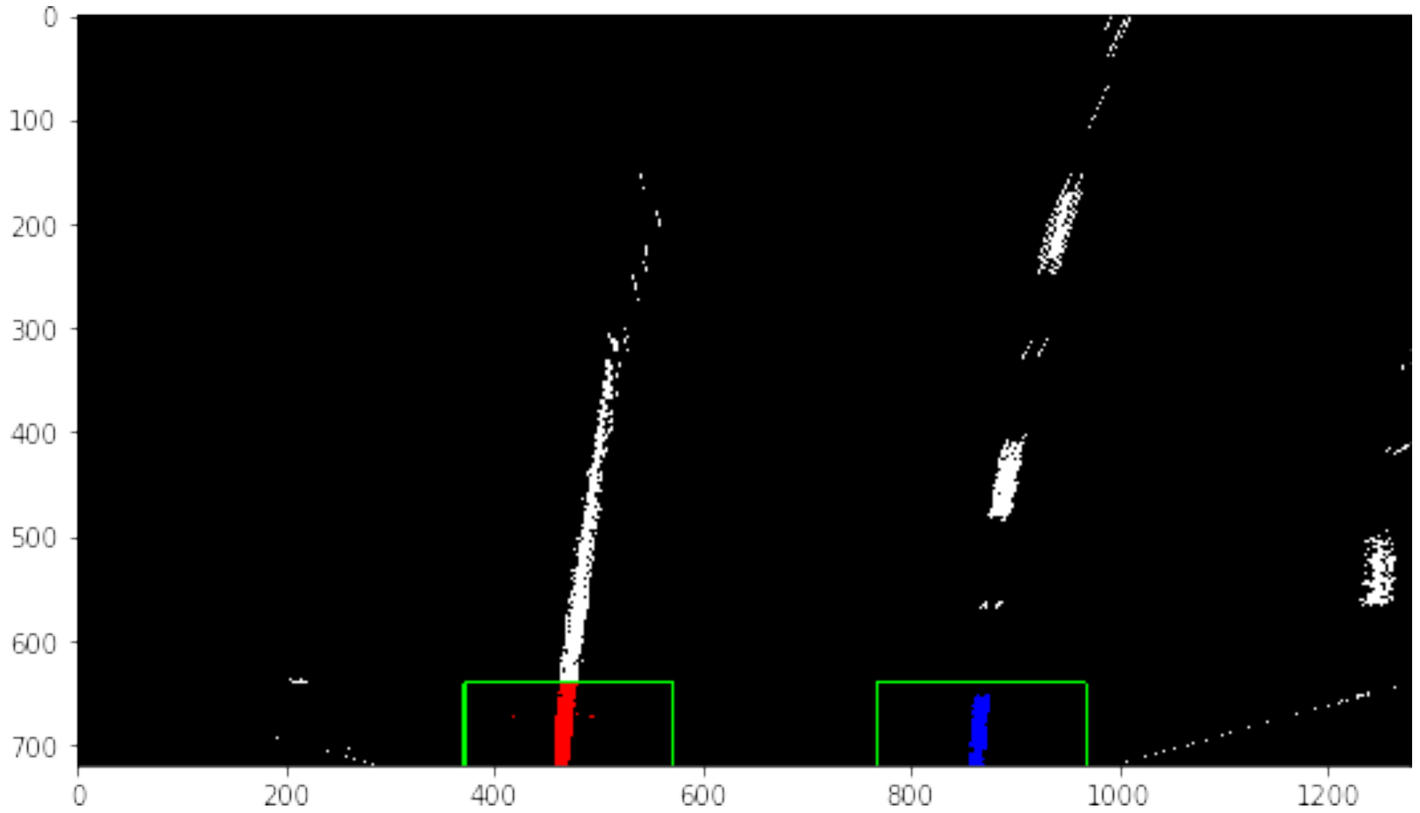


Step 1: Find the start of the line

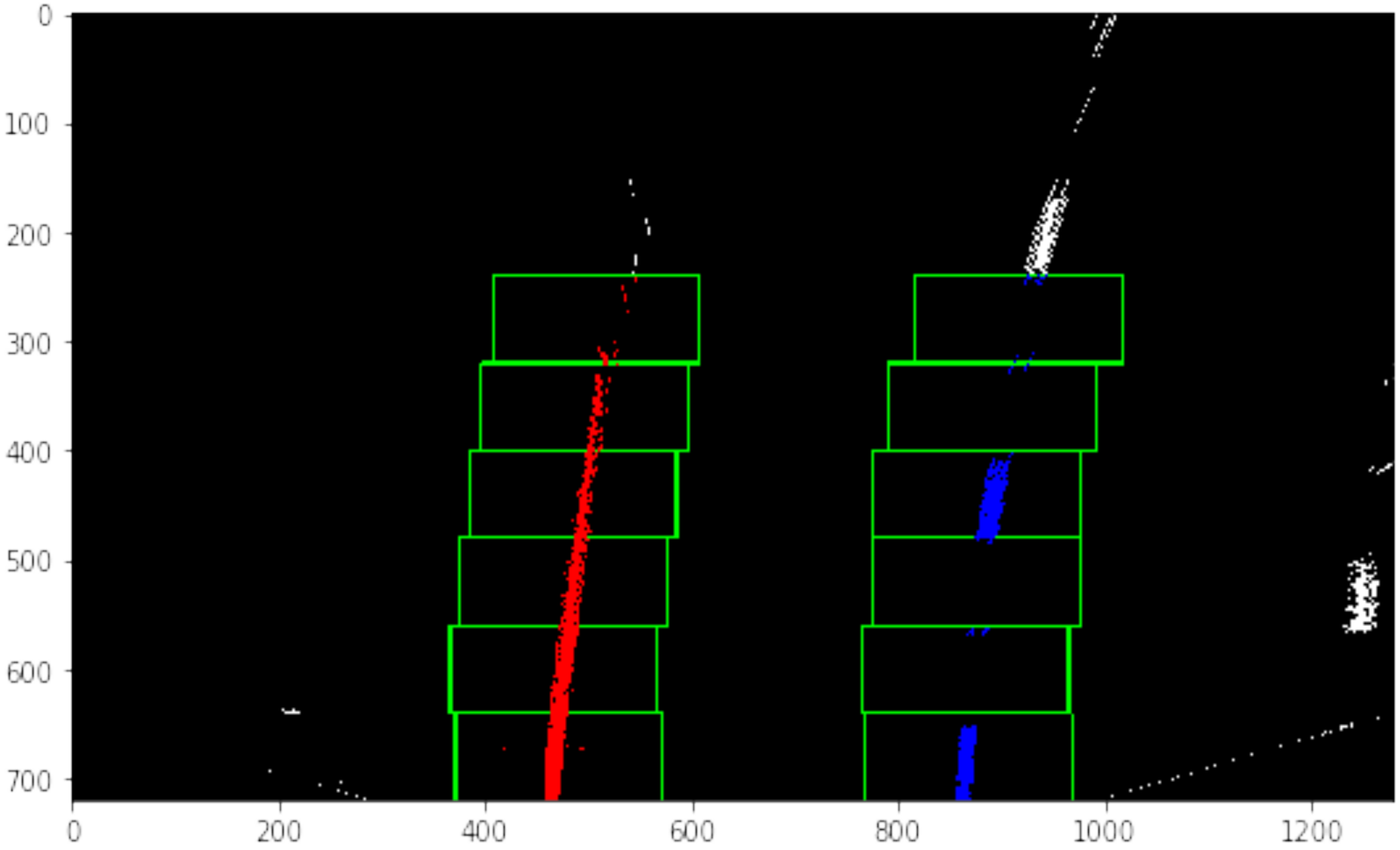
```
midpoint = ar(histogram.shape[0]/2)
leftx_base = np.argmax(histogram[:midpoint])
rightx_base = np.argmax(histogram[midpoint:]) + midpoint
```



Step 2: Create windows around these maxima

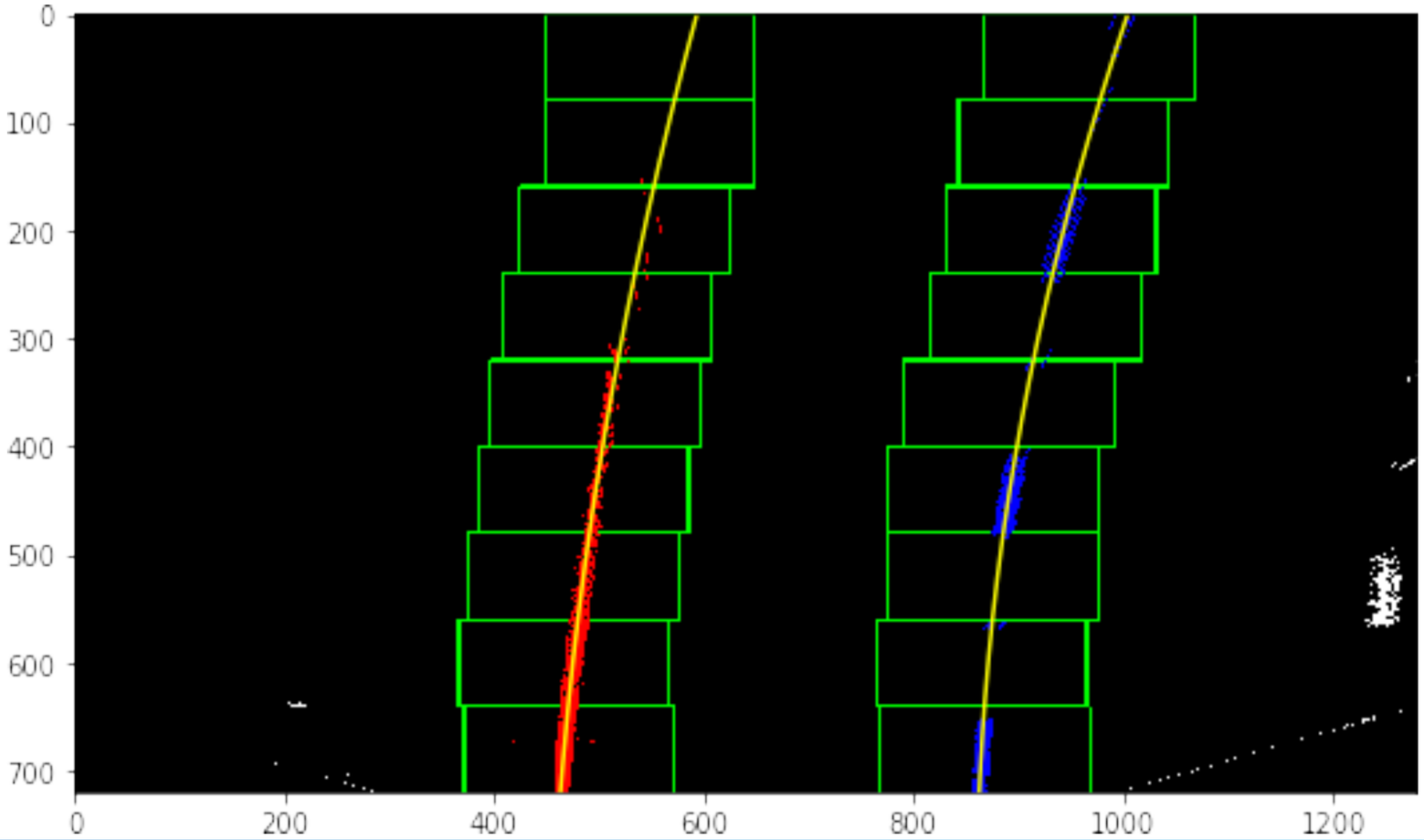


Step 3: Stack and move the windows

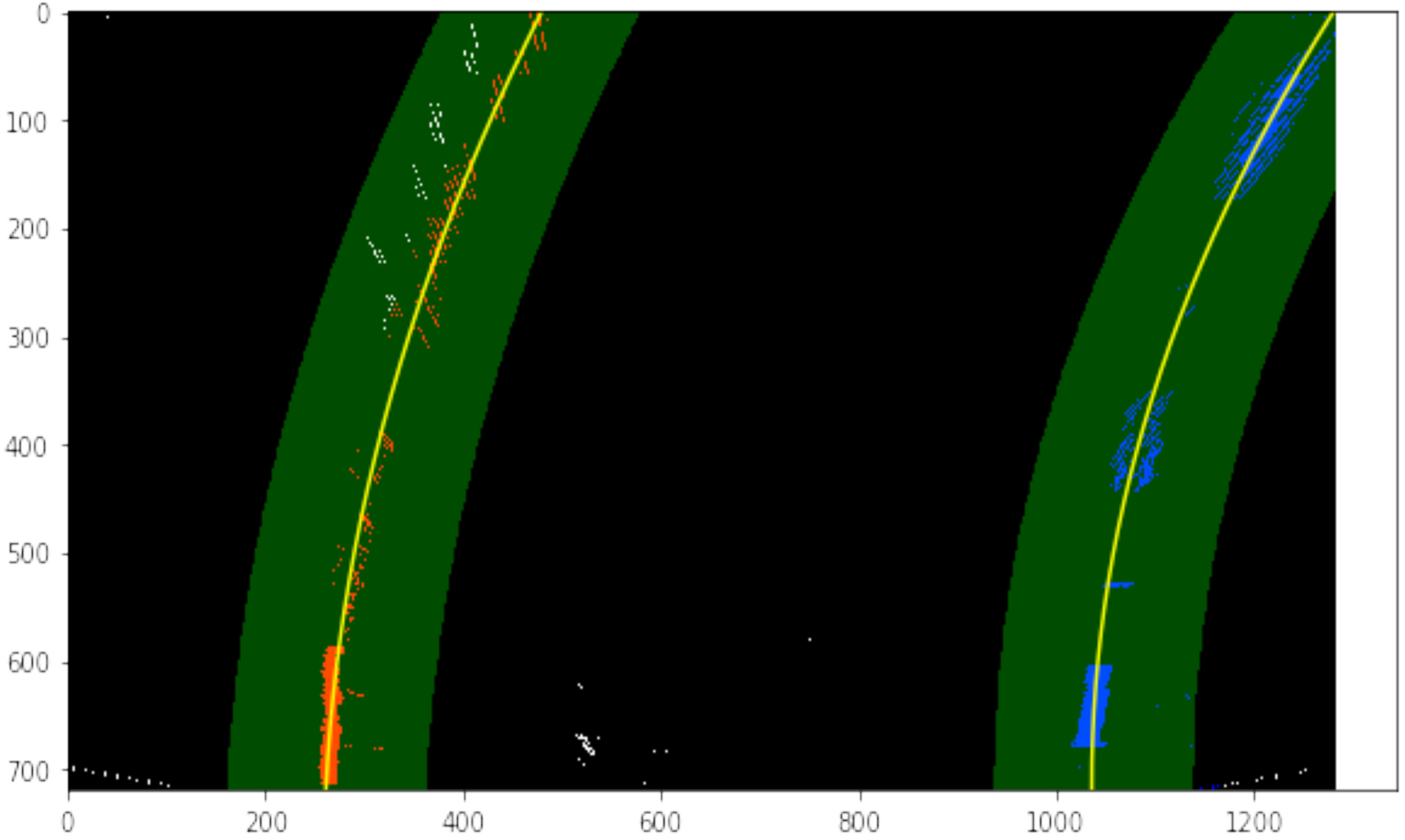


Step 4: Polyfit all the points from windows

```
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

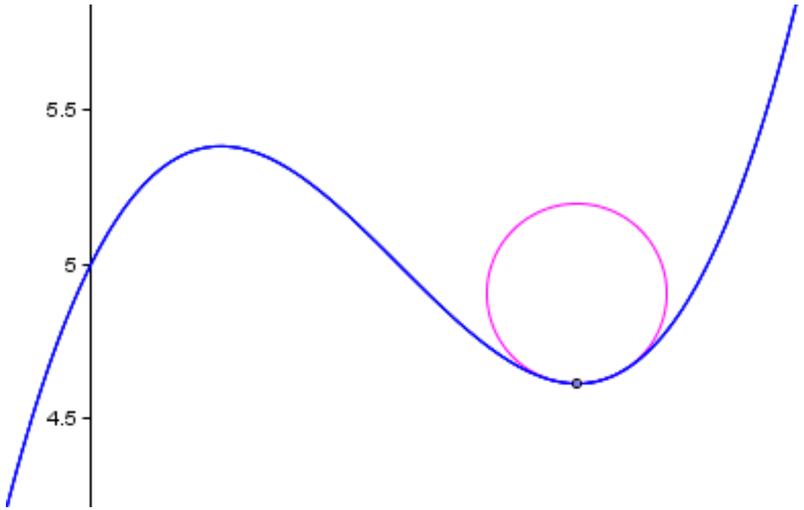


Method 2 uses an existing fit



Overlap Results on Image

Calculating radius of curvature



$$\text{Radius of curvature} = \frac{\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{3/2}}{\left|\frac{d^2y}{dx^2}\right|}$$

Polyfit equation: $y = Ax^2 + Bx + C$

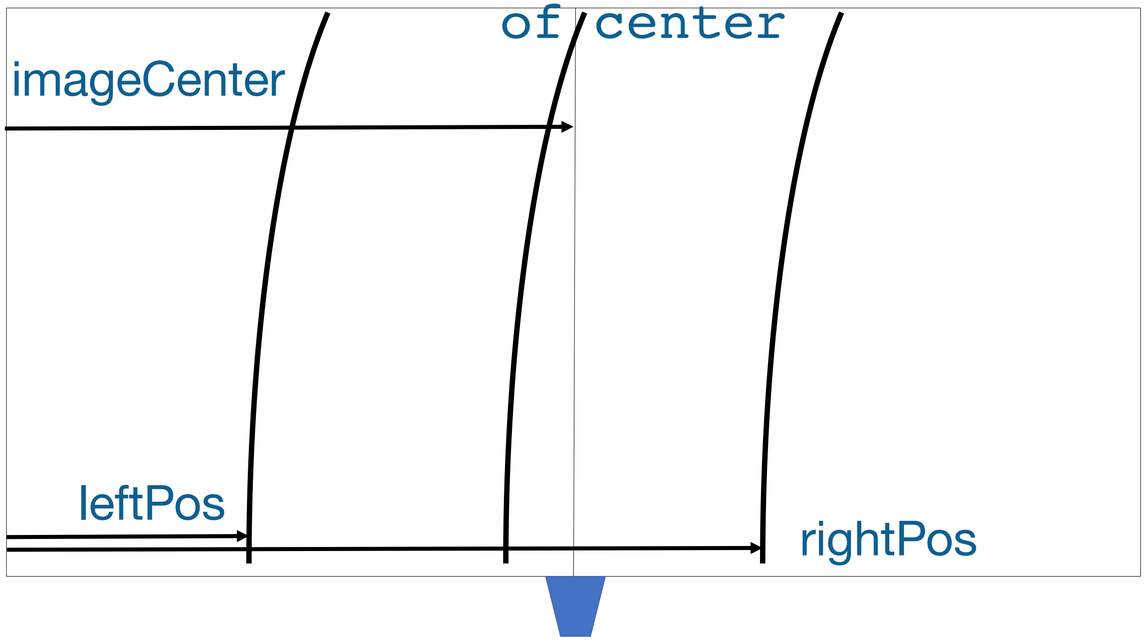
```
fit = [A, B, C]

curve_rad = ((1 + (2*fit[0]*y_eval +
fit[1])**2)**1.5) /  np.absolute(2*fit[0])
```

Distance to lane center

```
laneCenter = (rightPos - leftPos) / 2 + leftPos
distanceToCenter = laneCenter - imageCenter
```

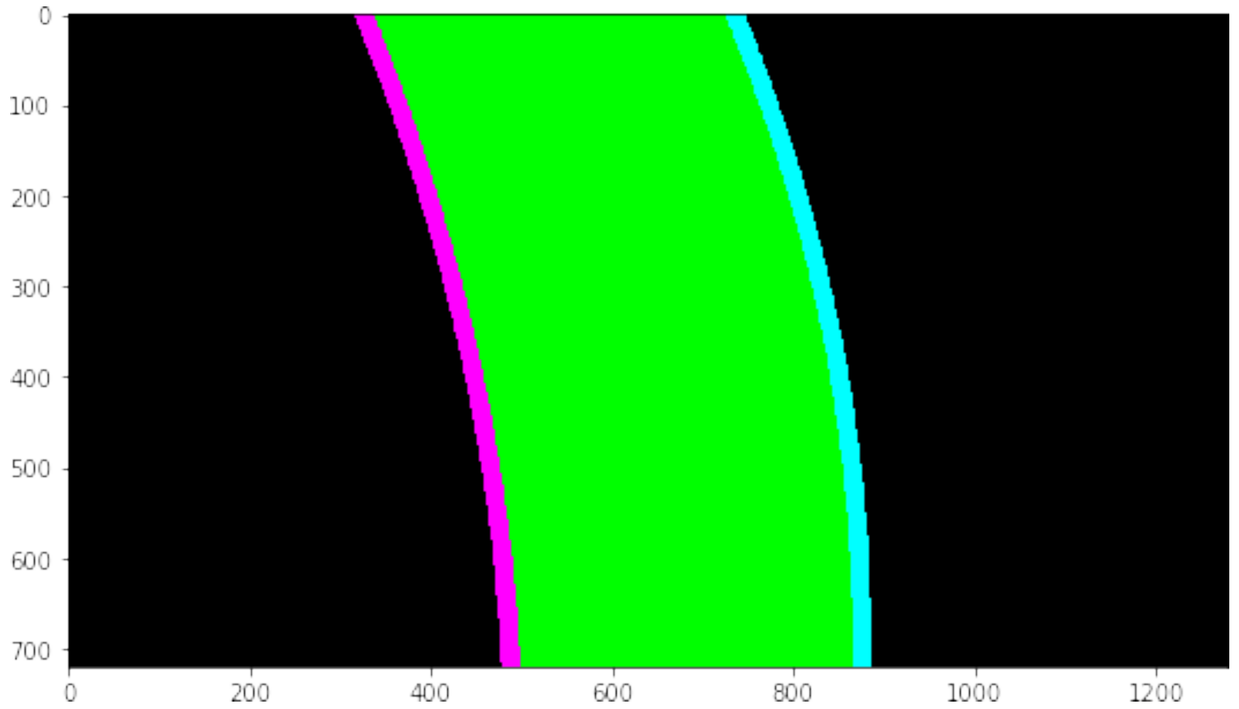
(0, 0)

If “+” then right of center



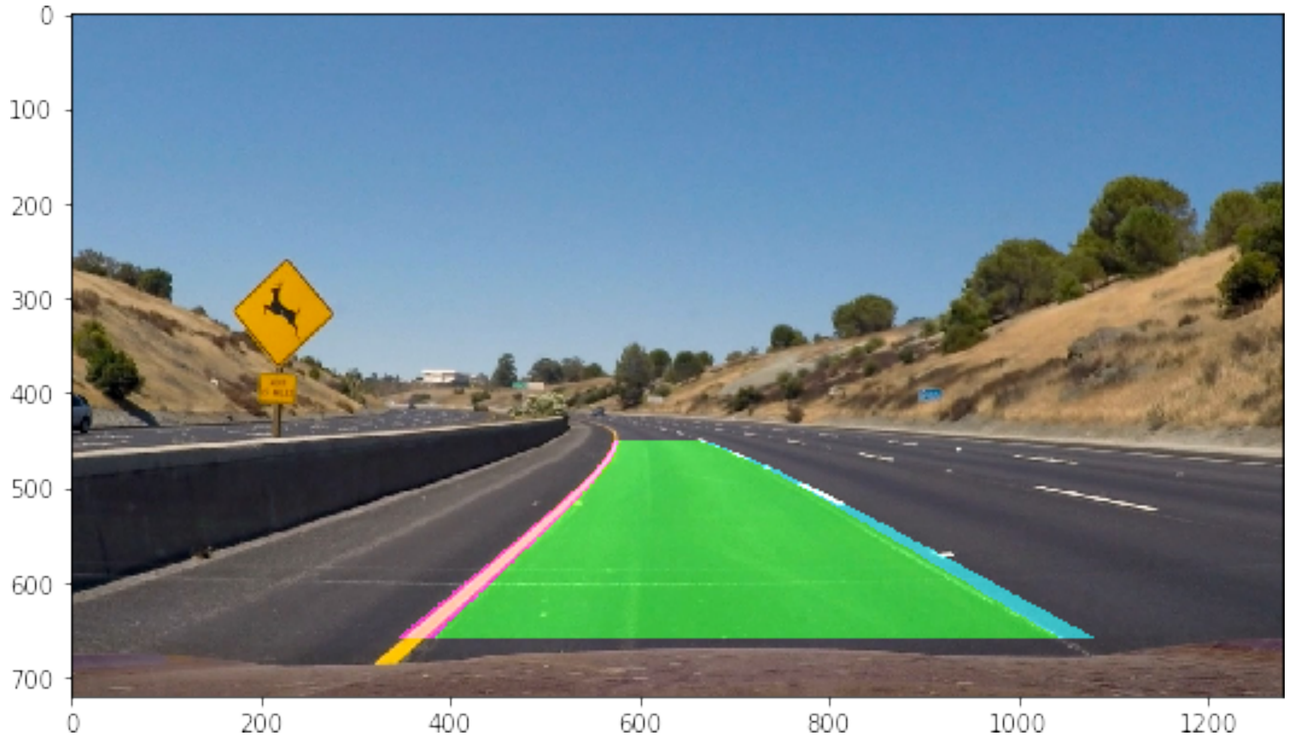
Plotting the lines and the drive area

```
cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))

cv2.polylines(color_warp, np.int32([pts_left]), …)
```



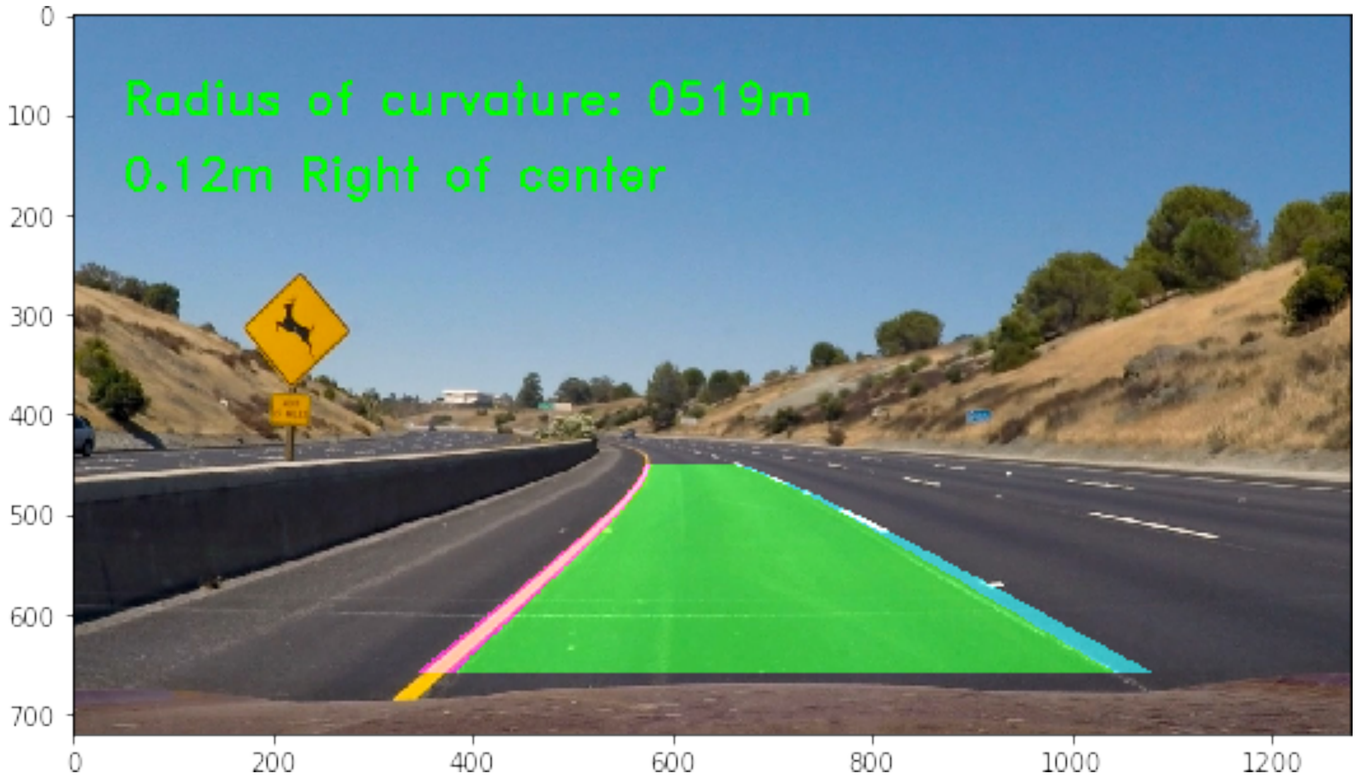
Unwarp the image and add to the original

```
result = cv2.addWeighted(img, 1, newwarp, 0.5, 0)
```



Add the radius and center distance

```
cv2.putText(img, text, (50,100), font, 1.5, (0,255, 0),
2, cv2.LINE_AA)
```



Building more

- Image warping
- Dynamic color channels
- Speed
- Smoothing

References

- Test images and videos come from the udacity open source self- driving car repo: https://github.com/udacity/self-driving-car