1 Computer vision Fundamentals

Formal definition of an image:

If we define f (x, y) ∈ R to be the intensity at position (x, y), an image can be defined as a function f: R^2 → R. That is, a function that maps from 2 dimensions to a single intensity or pixel value.

A color image 3 channels or planes (r,g,b).Each channel having specific intensity values.
As a function, a color image would be defined as f: R^2 → R^3

Computer vision typically operates on discrete images.

- So, we sample the 2D space on a regular grid
- And we quantize each sample (round to nearest integer to get pixel value)
- Image is thus represented as a matrix of integer values.
    - The rows are the y values.
    - The columns are the x values.

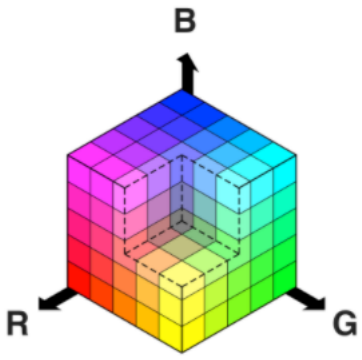
Reading an Image

```
import matplotlib.image as mpimg
import cv2


image = mpimg.imread('image.jpg') # ← Read the image

#Color channels
red_channel = image[:,:,0]
green_channel = image[:,:,1]
blue_channel = image[:,:,2]

#Convert to gray scale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

1.1 Color Spaces

- A color space is a specific organization of colors;
  - Color spaces provide a way to categorize colors and represent them in digital images.

- **RGB** is red-green-blue color space.
  - You can think of this as a 3D space, in this case a cube, where any color can be represented by a 3D coordinate of R, G, and B values.
  - For example, white has the coordinate (255, 255, 255), which has the maximum value for red, green, and blue.



RGB color space

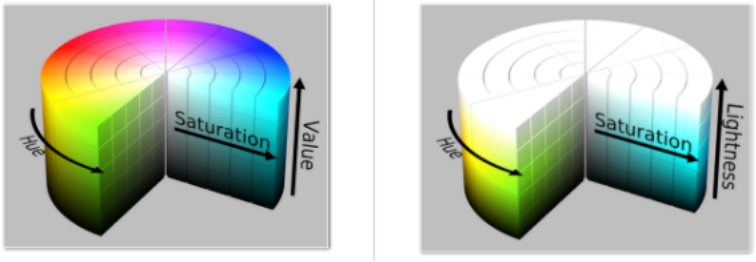
- **Note**:
  - If you read in an image using *matplotlib.image.imread()* you will get an ***RGB*** image,
  - But if you read it in using OpenCV *cv2.imread*() this will give you a ***BGR*** image.

- **Hue**
  - This is the value that represents color independent of any change in brightness.
  - For example:
    - Take a basic red paint color.
    - If you add some white or some black to make that color lighter or darker, the underlying color is still the same, red. So, the Hue remains the same.

- **Lightness and Value**
  - These represent different ways to measure the relative lightness or darkness of a color.
  - For example, a dark red will have a similar hue but much lower value for lightness than a light red.

- **Saturation**
  - A measurement of colorfulness.
  - As colors get lighter and closer to white, they have a lower saturation value, whereas colors that are the most intense, like a bright primary color (imagine a bright red, blue, or yellow), have a high saturation value.

- Most commonly used color spaces are **HSV** (Hue, Saturation and Value) or **HLS** (Hue, Lightness and Saturation).

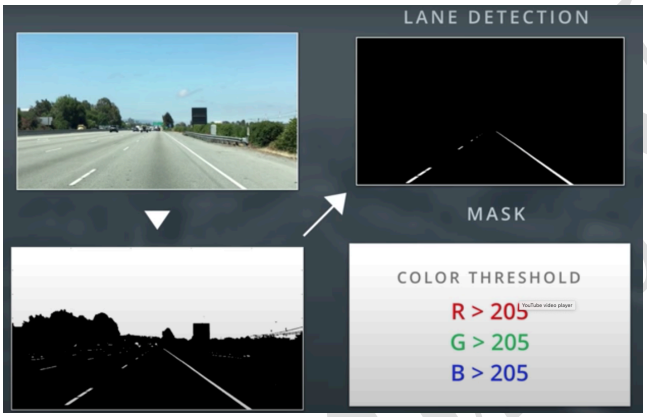


(Left) HSV color space, (Right) HLS color space

- We can use OpenCV function *cvtColor()* to convert images to different color spaces.
    - For example,
    - *cv2.cvtColor(im, cv2.COLOR_RGB2HLS)* converts from RGB to HLS.

1.1.1 Color Thresholding

We can use color thresholds and masking to detect certain features from images. For example, if we need to detect lane markings on a road, we could use thresholds on the RGB color space and then mask the region of interest resulting in an image that detects lane lines.



- **Drawbacks of RGB color space**
    - Works best on white lane pixels.
    - Doesn't work well with varying color and light conditions.
    - Example: Image and corresponding RGB channels

        - R and G channels detect yellow and white lines well. The blue channel doesn't detect the yellow line.
        - At the back of the image, R and G channels don't detect lines where there is a shadow or bright light.

- Example using HLS color space
  - **In HLS space**,
    - The L component can be isolated, and this is the component that varies the most under different lighting conditions.
    - The H and S component stay fairly consistent in shadow or excessive brightness.
    - If we use values from H and S channels and discard the L channel, we can detect lane lines more reliably than from the RGB space.

- The dark sections of the H channel combined with the S channel can be used to detect lane lines.

Example Code

```
# Convert image
hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)

#Extract channels
H = hls[:,:,0]
L = hls[:,:,1]
S = hls[:,:,2]

#Get binary version of channel
binary_S = np.zeros_like(S)
binary_H = np.zeros_like(H)


# Define some threshold manually
thresh_S = (90, 255)
thresh_H = (15, 100)

#Apply threshold - set 1 where there is a match, 0 otherwise
binary_S[(S > thresh_S[0]) & (S <= thresh_S[1])] = 1
binary_H[(H > thresh_H[0]) & (H <= thresh_H[1])] = 1
```

1.2 Noise in images

Since images are functions, noise is just another function that is combined with the original image function to get a new function.

So, if noise is $\eta(x, y)$ and the original image is f (x, y), we get:

$$\mathbf{g\ (x, y) = f\ (x, y) + \eta(x, y)}$$

Kinds of noise functions:

- Salt and pepper noise
- Impulse noise
- Gaussian noise

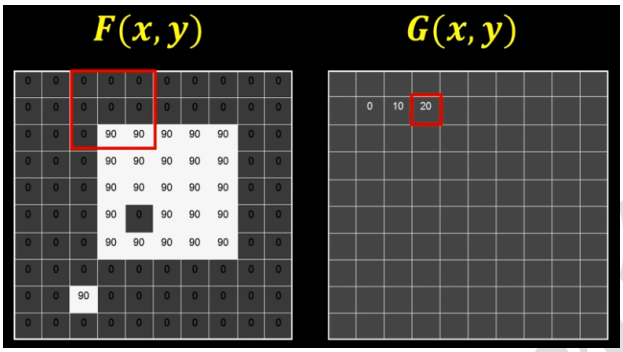
1.3 Image Filters

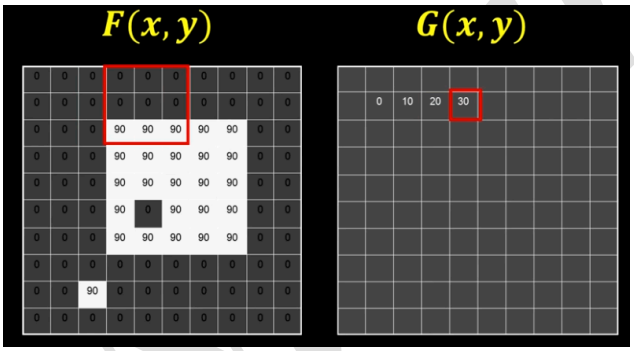
A filter is used to remove noise from an image or transform the image in some form.
It can also be used to extract information from an image.

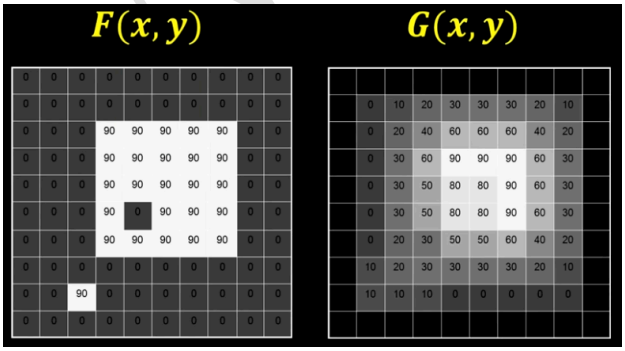
One common approach is to average the neighbors around a given pixel.
This normally works, when the following assumptions hold true:

- The true value of pixels is similar to the true value of pixels nearby.
- The noise added to each pixel is done independently.

Example: Applying a filter to an image







This process of averaging the pixel values is called **Correlation Filtering.**

When we use **uniform weights**, mathematically, it is given by:

- Assume window size is 2k+1

$$G[i,j] = \frac{1}{(2k+1)^2} \sum_{\{u=-k\}}^{k} \sum_{\{v=-k\}}^{k} F[i+u, j+v]$$

- Input image is F[i,j]
- We add pixel values around i and j
- We divide with a uniform weight: $(2k+1)^2$

Uniform weights typically don't work. So, we use non uniform weights. This is given by:

$$G[i,j] = \sum_{\{u=-k\}}^{k} \sum_{\{v=-k\}}^{k} H[u,v] \,.\, F[i+u, j+v]$$

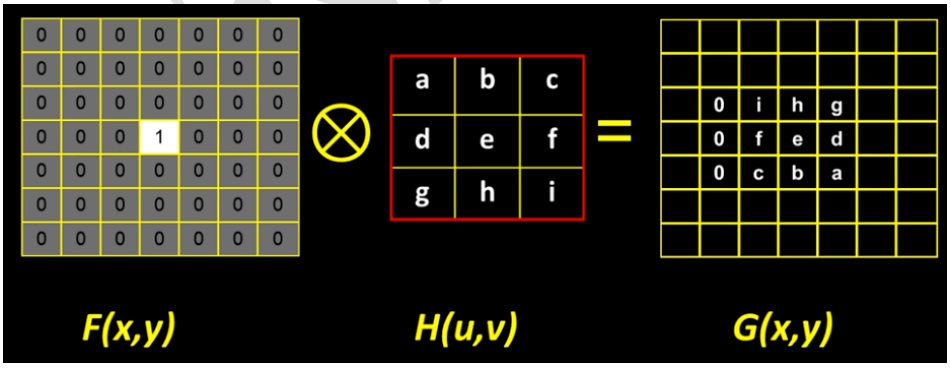
- H[u,v] represents the non-uniform weight. This is called the **filter** or **mask**.
  - It is a matrix of weights
- This is called **cross-correlation**, denoted by $G = H \otimes F$s

1.3.1 Correlation vs Convolution

Correlation, denoted by $G = H \otimes F$ is computed as:

$$G[i,j] = \sum_{\{u=-k\}}^{k} \sum_{\{v=-k\}}^{k} H[u,v] \,.\, F[i+u, j+v]$$

Example using Correlation:



If you notice, after the correlation operation, the output is a **flipped version** of the filter.

Convolution, denoted by $G = H * F$, is computed as:

$$G[i,j] = \sum_{\{u=-k\}}^{k} \sum_{\{v=-k\}}^{k} H[u,v] \,.\, F[i-u, j-v]$$

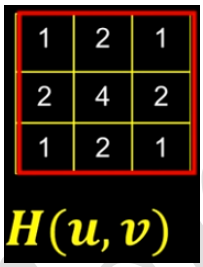
Here, the access to indices is flipped so that we get the desired output, i.e. the same as the filter. Typically, convolution is used to process images.

Properties of convolution:

- Linear and shift invariant
- **Commutative**: $G * F = F * G$
- **Associative**: $(G * F) * H = G * (F * H)$
- **Identity**: $F * e = F$, where, Identity or unit impulse, e = [0, …0,1, 0…0]
- **Differentiation**: $\frac{\partial}{\partial x}(f * g) = \frac{\partial f}{\partial x} * g$

1.3.2 Gaussian Filter

Example of a Gaussian filter: It is circularly symmetric.




```
#Adding Gaussian noise or blur,
#standard deviation or variance = 0
blur = cv2.GaussianBlur(image, (kernel_size,kernel_size),0)
```

2 Edge Detection

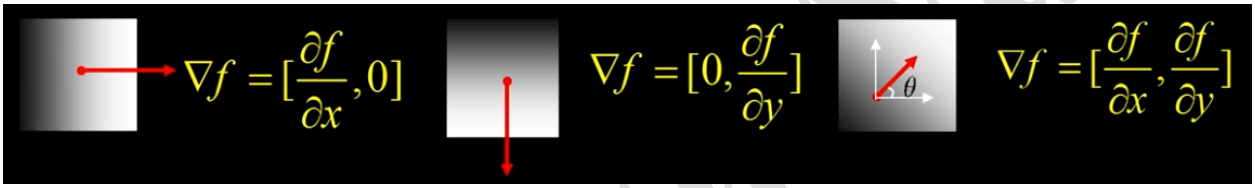
Edges in an image convey a lot of information about the contents of the image. We don't need to process all the information if we can detect the edges.

To detect an edge, we essentially look for changes in intensity around a neighborhood of pixels.

Formally, an edge is a place of rapid change in the image intensity function.

Differential operators are masks or kernels when applied to an image, compute the gradient function of the image. This gradient function is then thresholded to select the edge pixels.

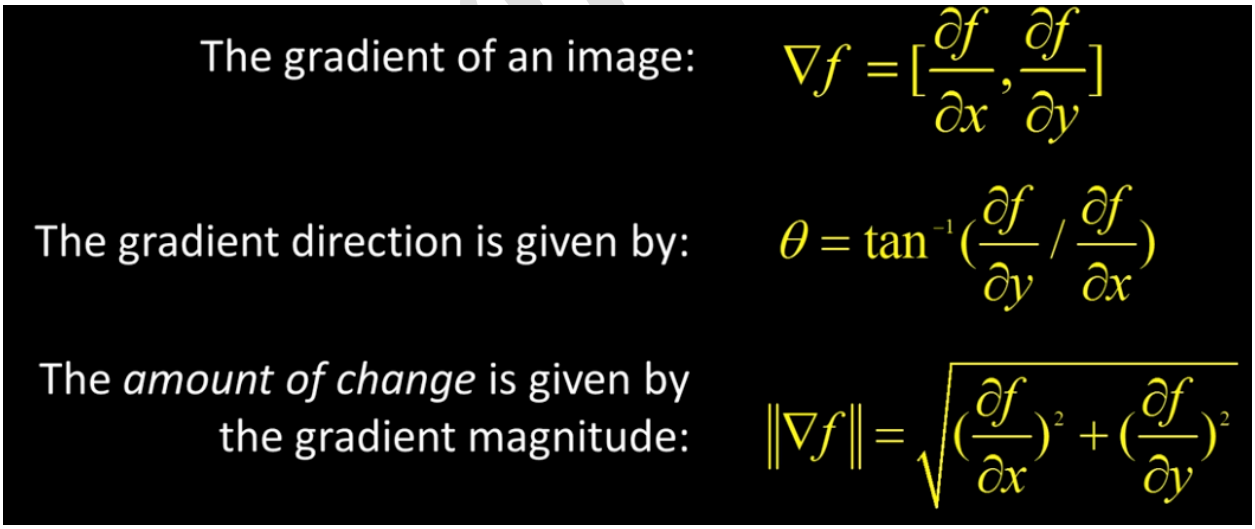
Image Gradient points in the direction of most rapid increase in intensity and the magnitude is the rate of change in intensity.



In the first example above, intensity changes only along the x-axis.

In the second example above, intensity changes only along the y-axis.

In the third example above, intensity changes along both axes

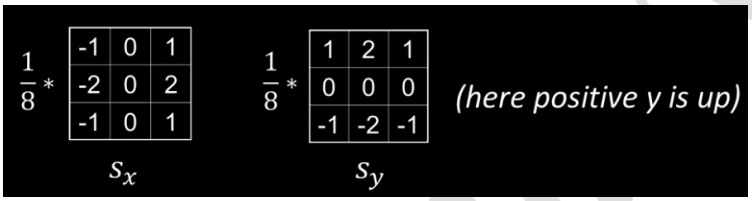


General steps for edge detection:

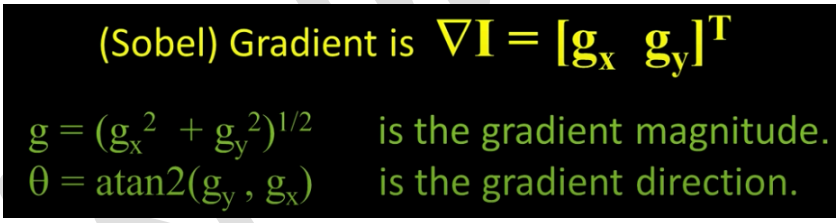
- Convert image to grayscale
- Remove noise and compute gradient
- Apply thresholds to find regions of "significant" gradient.
- "Thin" to get localized edge pixels.
- Link or connect edge pixels.

2.1 Sobel Operator

- Applying the Sobel operator to an image is a way of taking the derivative of the image in the x or y direction.
- It is defined as follows:

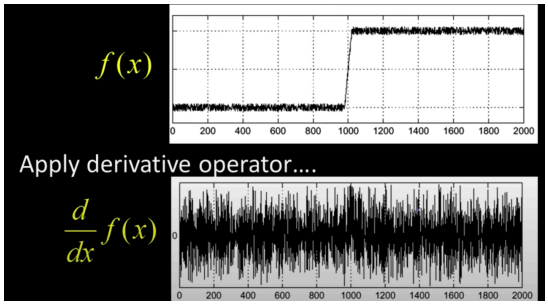


- We use the above to compute the gradient (magnitude and direction) as follows:

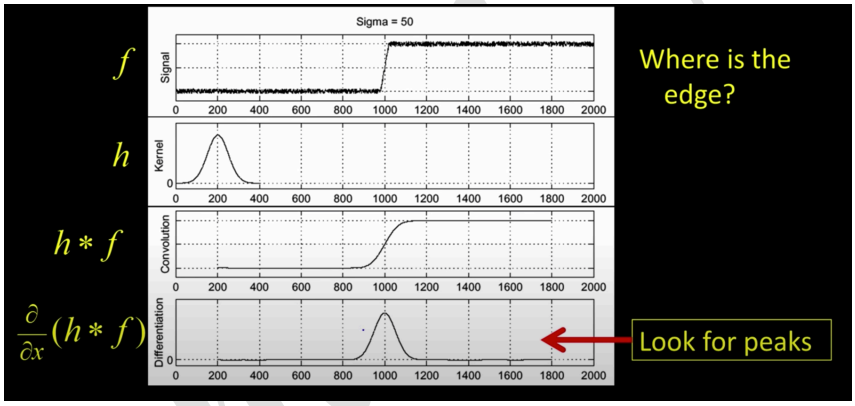


- **Note**:
  - g_x is the application of s_x
  - g_y is the application of s_y

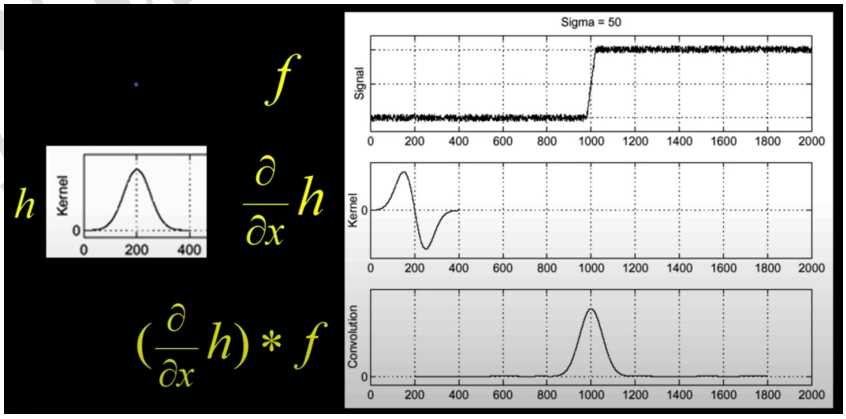
- Practically, this is not useful for edge detection because of noise. With noise, we get something like the below:



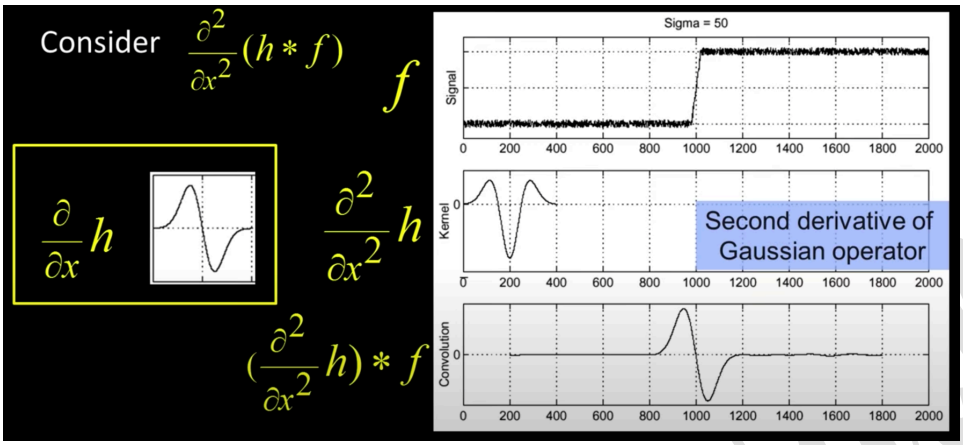
- The noise makes the derivative go all over the place.
- We can use a filter to solve this problem.
- If we use a Gaussian filter, we get the following with convolution:
  - Assume our image is *f* and filter is *h*



- Using the differentiation property of convolution, we can save one operation and detect edges as follows:



- This can be extended to compute second derivative too to find the maximum value.



2.2 Canny Edge Detection

- Convert image to grayscale
- Filter image with derivative of Gaussian and compute gradient.
- Non maximum suppression: “Thin” multi-pixel wide “ridges” to single pixel width.
- Linking and thresholding:
    - Define 2 thresholds: low and high
    - Apply high threshold to detect strong edge pixels.
    - Link these to form strong edges.
    - Apply the low threshold to find weak but plausible edge pixels.
    - Extend the strong edges to follow weak edge pixels

Original	Gradient	Thresholding	Thinning
			
Linking and thresholding (final result)			
			

```
#Getting canny edges using OpenCV,
#Recommended low to high threshold = 1:2 or 1:3

edges = cv2.Canny(gray, low_threshold, high_threshold)
```

2.3 Hough Transform: Line detection

Hough: Pronounced Huff

Parametric model: A model that can represent a class of instances where each instance is defined by a set of parameters. Example: A line or circle can be thought of as a parametric model.

2.3.1 Edges to Lines

Voting: A general technique where features vote for all models compatible with it.

- First, cycle through all features. Each feature casts votes for model parameters.
- Look for parameters that receive most votes.
- Example: If model is for a line, features are points in the image and they vote for various parameters that represent a line the point belongs to.

Why does voting work?

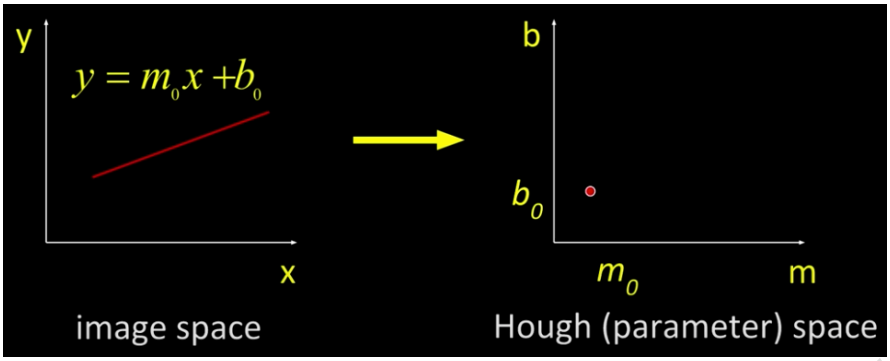
- Noise and clutter features cast votes too. But their votes are typically inconsistent compared to actual "good" features.
- If some features are missing, majority model still generalizes to span multiple fragments.

To fit lines, we need to identify the following:

- Given points that belong to a line, what is the line?
- How many lines are there?
- Which points belong to which lines?

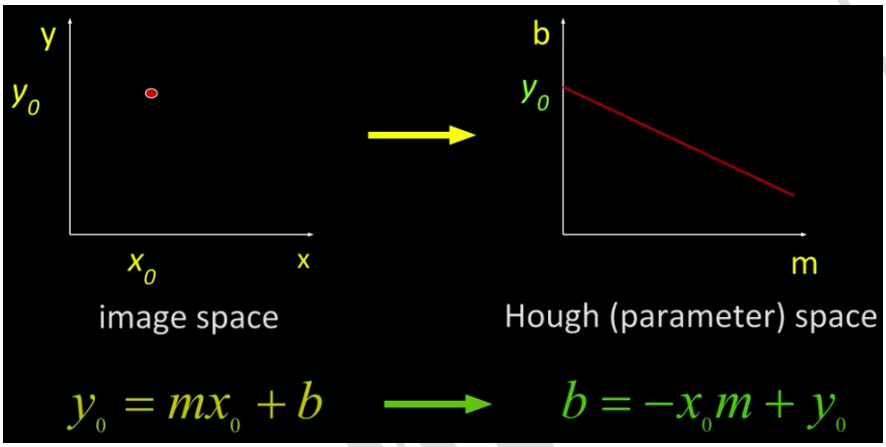
Hough Transform is a voting technique that answers the above questions.

2.3.2 Hough Space



A line in the image space is a point in the Hough space.

- This is because there is a unique (m_0, b_0) that parameterizes the line.



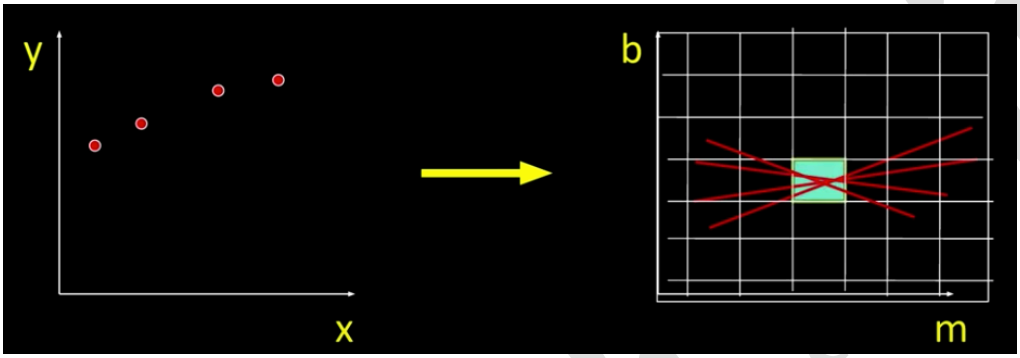
A point in image space is a line in Hough Space.

- This because, many lines can go through that point in image space.
- All such lines have parameters that fall on a line in Hough Space as depicted above.

2.3.3 Hough algorithm

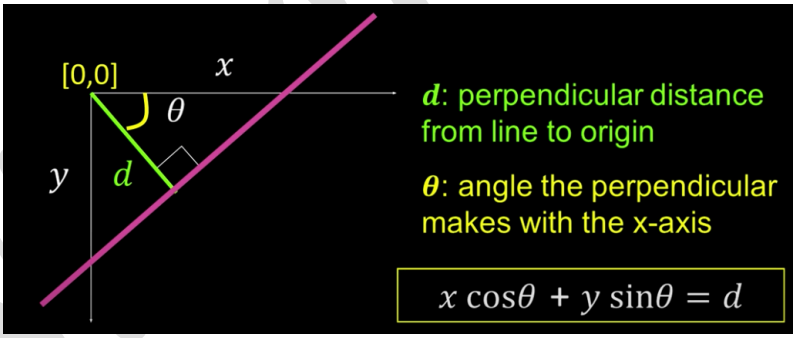
In the **coordinate plane**, the basic algorithm works as follows:

- Let each point in the image space (on an edge), vote for a set of possible parameters in Hough space (lines in Hough space)
- Accumulate votes in discrete set of bins.
- Parameters with the most votes indicate a line in image space.



For the Hough Transform algorithm, we will use a polar representation of lines because with the coordinate representation, we run into problems like infinite slope etc.

The polar representation of a line is as below.

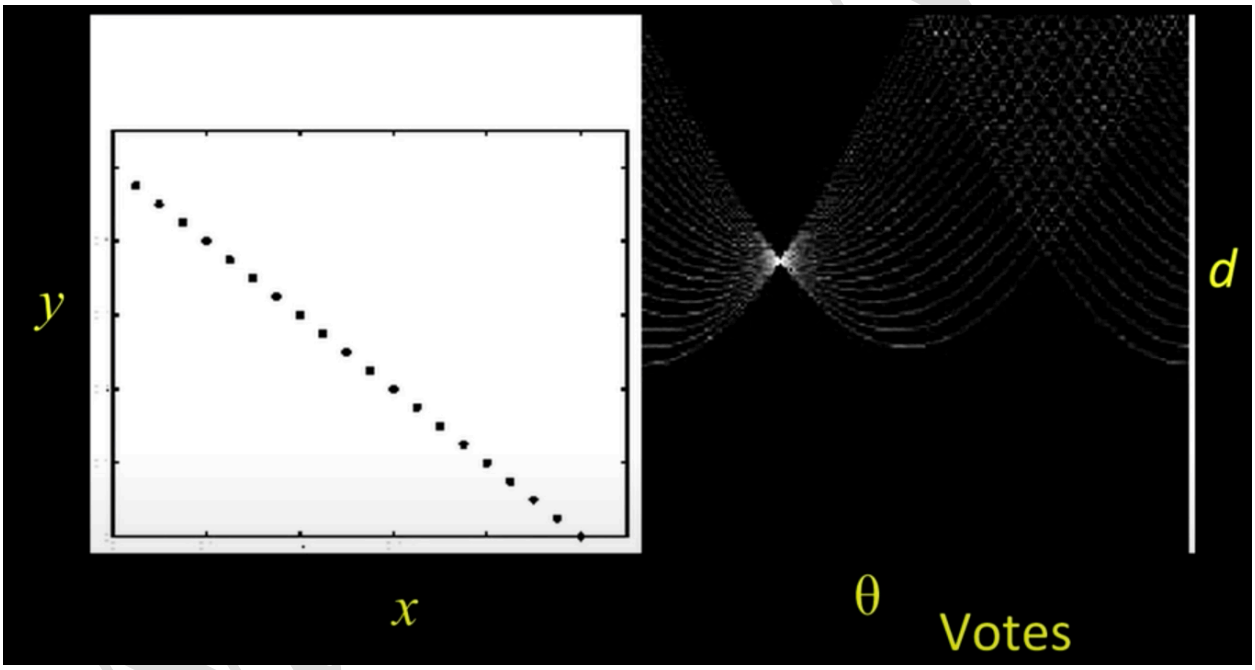


- So, a point in image space will now correspond to Sinusoid in Hough Space.
- A line in image space will correspond to a point in Hough space (where all the sinusoids meet).

In the **polar coordinate plane**, the algorithm works as follows:

- Initialize H[d,θ] = 0
  - H[d,θ] is the **Hough accumulator** array that stores the votes for each set of parameters.
- For each edge point (x, y) in the image:
  - $\theta = gradient\ at\ (x, y)$
  - $d = x\ cos\theta - y\ sin\theta$
  - H [d, θ] += 1
- Find value of (d, θ) where H [d, θ] is maximum.
- These parameters define the line identified on the image. The line is given by:
  - $d = x\ cos\theta - y\ sin\theta$

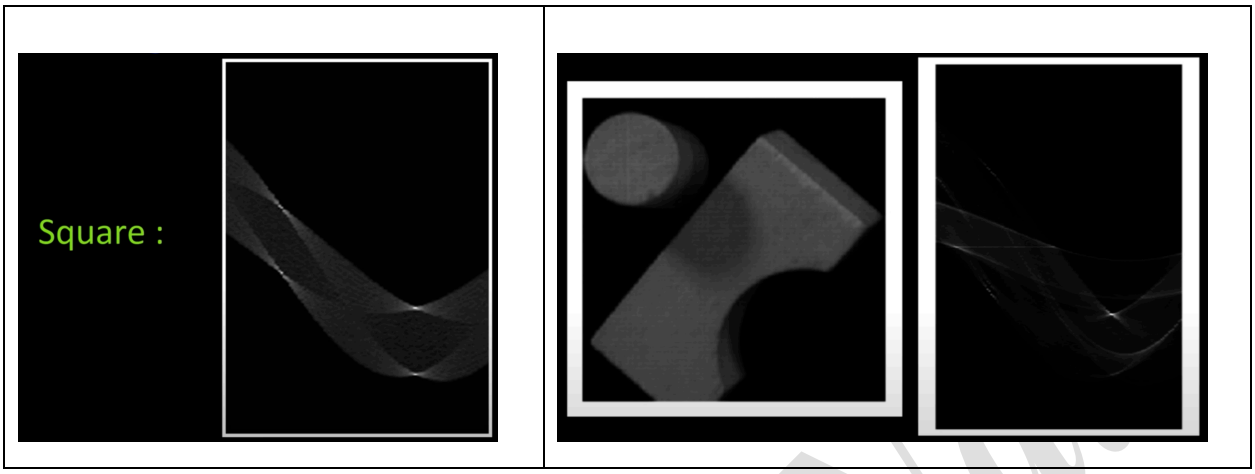
Examples:



In the above, In the Hough space,

- Bright value = High vote count
- Black = no votes

Other examples:



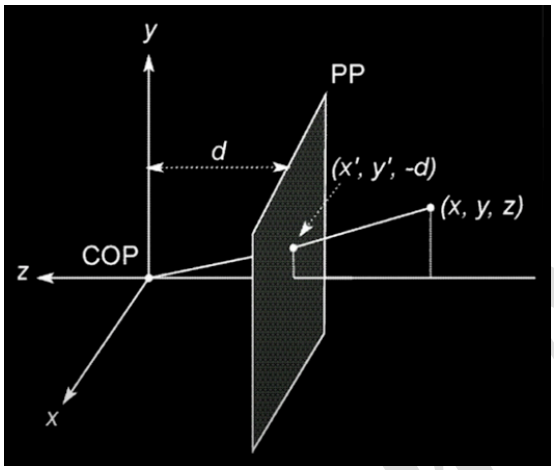
Sample Code:

```
# Do relevant imports
import matplotlib.image as mpimg
import numpy as np
import cv2

# Read in and grayscale the image
# Define a kernel size and apply Gaussian smoothing
# Define our parameters for Canny and apply
low_threshold = 50
high_threshold = 150
masked_edges = cv2.Canny(blur_gray, low_threshold,
high_threshold)

# Define the Hough transform parameters
# Make a blank the same size as our image to draw on
rho = 1
theta = np.pi/180
threshold = 1
min_line_length = 10
max_line_gap = 1


# Run Hough on edge detected image
lines = cv2.HoughLinesP(masked_edges, rho, theta,
                        threshold,np.array([]),
                          min_line_length,
                           max_line_gap)

#creating a blank to draw lines on
line_image = np.copy(image)*0

# Iterate over the output "lines" and draw lines on the blank
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),10)

# Create a "color" binary image to combine with line image
color_edges = np.dstack((masked_edges,
                           masked_edges,
                             masked_edges))

# Draw the lines on the edge image
combo = cv2.addWeighted(color_edges, 0.8, line_image, 1, 0)
plt.imshow(combo)
```

3 Camera Calibration

3.1 Cameras and Images

Image: A 2D projection of 3D points



Assume the following:

- Center of projection (COP) is at the origin
- Coordinates of the point of interest in the real world is (x,y,z)
- Image plane is at a distance 'd' from the COP
- The projection of the 3D point on the image plane in 2D is given by:
  - (x,y,z) → (x`, y`)
  - (x`, y`) = (-d (x/z), -d (y/z))

- The above is not a linear transformation, since we need to divide by a different z for every point. To make it linear, we use **homogenous coordinates**. This gives us the **perspective transformation** from 3D to 2D as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z/f \end{bmatrix} \Rightarrow \left(f\frac{x}{z}, f\frac{y}{z} \right) \Rightarrow (u, v)$$

- 'f' is the focal length of the lens or camera.

- This transformation is invariant to scale. So, its equivalent to the following:

$$\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} fx \\ fy \\ z \end{bmatrix} \Rightarrow \left(f\frac{x}{z}, f\frac{y}{z} \right)$$

3.1.1 Parallel Lines

Assume we have lines in 3D space, their corresponding transformation is as shown.

Line in 3-space

$$x(t) = x_0 + at$$
$$y(t) = y_0 + bt$$
$$z(t) = z_0 + ct$$

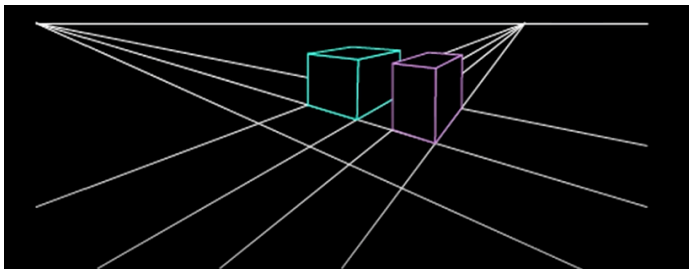
Perspective projection of the line

$$x'(t) = \frac{fx}{z} = \frac{f(x_0 + at)}{z_0 + ct}$$
$$y'(t) = \frac{fy}{z} = \frac{f(y_0 + bt)}{z_0 + ct}$$

In the limit as $t \to \pm\infty$ we have (for $c \neq 0$):

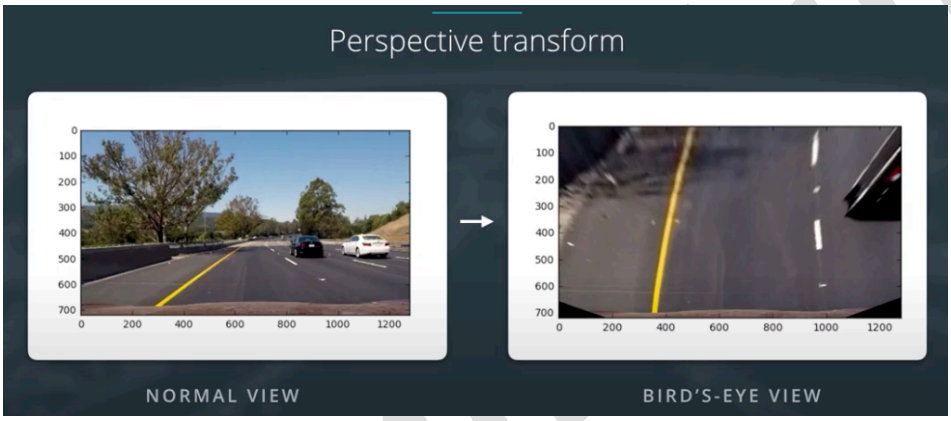
$$x'(t) \to \frac{fa}{c}, \quad y'(t) \to \frac{fb}{c}$$

- As seen above, when t→∞, x and y become constant.
- That is, all parallel lines meet at the same point in the 2D perspective.
- The point they meet at is called **vanishing points**.
- Sets of parallel lines on the same plane lead to collinear vanishing points. The line joining these collinear points is called the **horizon** for that plane.



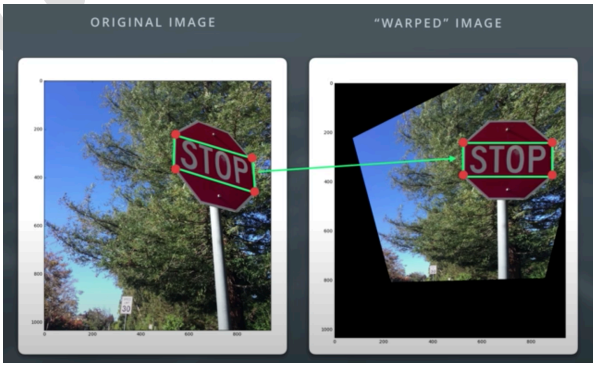
3.2 Perspective Transform

- A perspective transform maps the points in a given image to different, desired, image points with a new perspective.
- The perspective transform we will use is the **bird's-eye view** transform
  - This lets us view a lane from above;
  - This will be useful for calculating the lane curvature.
- Aside from creating a bird's eye view representation of an image, a perspective transform can also be used for all kinds of different viewpoints.



3.2.1 Steps to perform perspective transform

- Choose 4 points on an image that form a rectangle.
  - 4 points are sufficient for a linear transformation
- Choose 4 points that correspond to a rectangle the above need to transform to in the transformed or warped image.
  - These can be approximated from the above to form a true rectangle.



- Use OpenCV to compute the transformation matrix between the two images.
    - The function *getPerspectiveTransform* takes in the coordinates defined above and returns a transformation matrix.
- Use OpenCV to transform the original image to a warped image.
    - The function *warpPerspective* uses the transformation matrix from above to transform the image to the warped image.

Code example

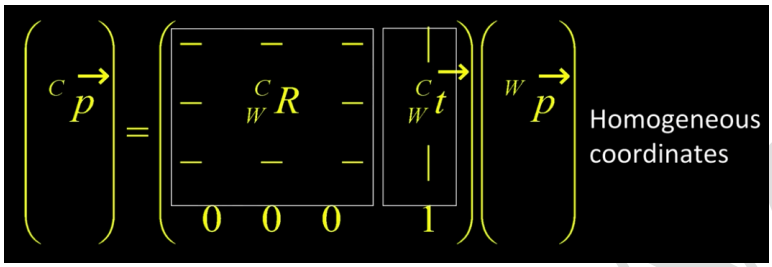
```
# Get src and dst coordinates
def get_coordinates(image):
    src = np.float32([
                    [(image.shape[1]/2)-150,(image.shape[0]/2)+120],
                    [(image.shape[1]/2)+170,(image.shape[0]/2)+120],
                    [image.shape[1]-30,image.shape[0]],
                    [40,image.shape[0]]
    ])

    dst = np.float32([
                [0,0],
                [image.shape[1],0],
                [image.shape[1]-50,image.shape[0]],
                [40,image.shape[0]]
    ])

    return src,dst

# Transform image perspective
def warp(image):
    src,dst = get_coordinates(image)
    img_size = (image.shape[1], image.shape[0])

    # Get transformation matrix
    M = cv2.getPerspectiveTransform(src, dst)

    # Warp the image
    warped = cv2.warpPerspective(image, M, img_size)
    return warped


image = mpimg.imread('test_images/test.jpg')
warped = warp(image)
```

- **Note**:
    - When you apply a perspective transform, choosing four source points manually, as we did above, is not the best option.
    - There are many other ways to select source points.
    - For example, many perspective transform algorithms will programmatically detect four source points in an image based on edge or corner detection and by analyzing attributes like color and surrounding pixels.

3.3 Geometric Camera Calibration

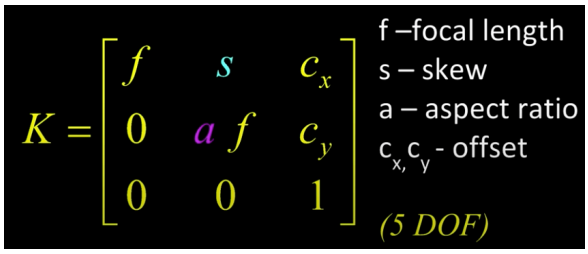
This is composed of 2 transformations:

- **Extrinsic parameters**:
  - Transformation from world coordinates to camera's 3D system.
  - Identifies camera's location in the world (camera pose)

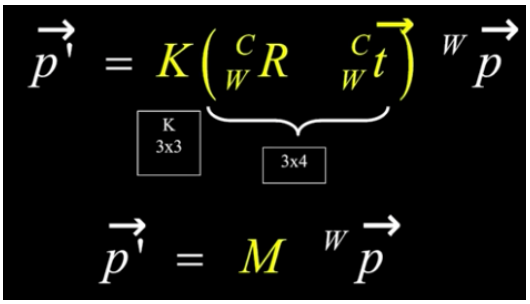


    - ${}^{c}P$ is coordinates in camera coordinate system (3D).
    - ${}^{w}P$ is coordinates in the real world.
    - ${}^{c}_{w}R$ is the rotation matrix (from world, w to camera, c).
    - ${}^{c}_{w}t$ is the translation matrix (from world, w to camera, c).
    - The above two combined form the **extrinsic parameter matrix**(4x4).

- **Intrinsic parameters**:
  - Transformation from 3D coordinates in the camera to the 2D image plane via projection.
  - What we did in the previous section was the ideal perspective projection.
  - The following takes into account other real-world considerations.
    - $P = K\ {}^{c}P$, where,
      - P is coordinates in image space (2D)
      - ${}^{c}P$ is coordinates in camera coordinate system (3D)
      - K is the **intrinsic parameter matrix**:



Combining Extrinsic and extrinsic parameters, we get:



Where,

3.4 Image Distortion

Image distortion occurs when a camera looks at 3D objects in the real world and transforms them into a 2D image; this transformation isn't perfect. Distortion actually changes what the shape and size of these 3D objects appear to be. So, the first step in analyzing camera images, is to undo this distortion so that you can get correct and useful information out of them.

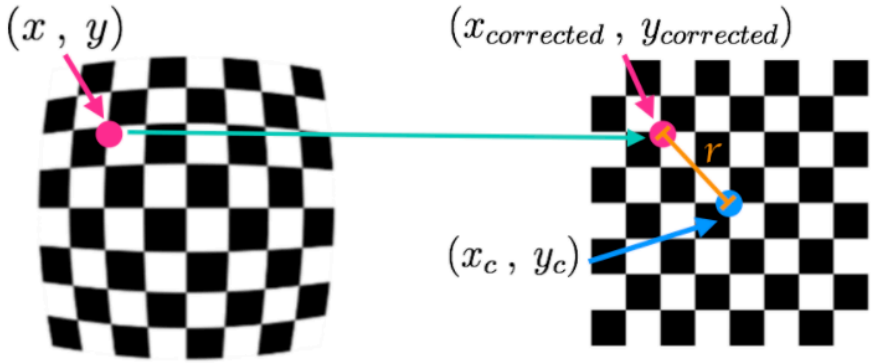
Types of distortion

- **Radial Distortion**
  - Most common type of distortion
  - Real cameras use curved lenses to form an image, and light rays often bend a little too much or too little at the edges of these lenses.
  - This creates an effect that distorts the edges of images, so that lines or objects appear **more or less curved than they actually are.**

- **Tangential Distortion**
  - Occurs when a camera's lens is not aligned perfectly parallel to the imaging plane, where the camera film or sensor is.
  - This makes an image look tilted so that some objects **appear farther away or closer than they actually are**.

3.4.1 Distortion Correction

There are three coefficients needed to correct for **radial distortion**: **k1**, **k2**, and **k3.**
There are two more coefficients that account for **tangential distortion**: **p1** and **p2.**



Let (x_c,y_c) be the center of the undistorted image. This is called the **distortion center**.
Let (x,y) be a point in the distorted image.
The corresponding point in the undistorted image will be $(x_{corrected}, y_{corrected})$.
The distance between $(x_{corrected}, y_{corrected})$ and (x_c,y_c) is r.

Radial distortion is corrected as follows:

$$x_{corrected} = x\ (1 + k_1r^2 + k_2r^4 + k_3r^6)$$
$$y_{corrected} = y\ (1 + k_1r^2 + k_2r^4 + k_3r^6)$$

Tangential distortion is corrected as follows:

$$x_{corrected} = x + (2p_1xy + p_2(r^2 + 2x^2))$$
$$y_{corrected} = y + (p_1(r^2 + 2y^2) + 2p_2xy)$$

Distortion Coefficients = $[k_1\ k_2\ p_1\ p_2\ k_3]$

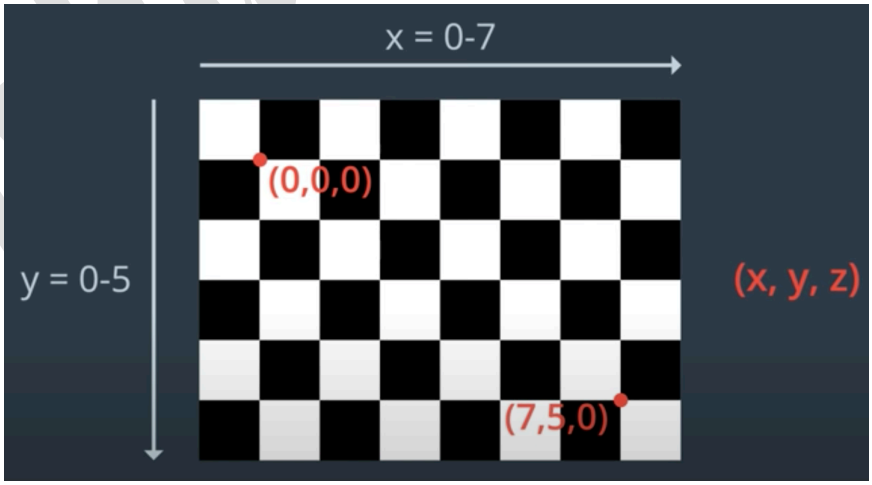
3.4.2 Measuring Distortion

Distortion can be measured by using images of known shapes. (like a chess board)

- Take multiple pictures of a chess board against a flat surface using the camera.
- Look at the difference between the apparent size in the image and actual size of the chess board and shapes and sizes of the squares in the image and what they actually are.
- Use this information to create a transformation that maps points in the distorted image to points in the undistorted image.
- Use the transformation info to undistort the image.

3.5 Steps to calibrate a camera

- Read in calibration images of a chess board.
    - At least 20 images are recommended.
    - These are images taken by the camera at different angles and distances.
    - Each chess board image has 8 x 6 corners to detect.
- For each calibration image, do the following:
    - Map the coordinates from 2D image (called **image points**) to the 3D coordinates of the real undistorted image (called **object points**) for each of the chess board corners.
        - The **object points** will range from (0,0,0) to (7,5,0) as shown below. The z coordinate will be 0 for all points.



        - Use NumPy to generate the object points for all the chess board corners.

    - Detect the corners of the calibration image using OpenCV:
        - Convert image to gray scale.
        - Use the function *findChessboardCorners* to get the coordinates of all the corners from the calibration image.
        - If corners are detected, add them to an image points array.
- Now we have the image points and object points for all the images.
- We can now use OpenCV to calibrate the camera.
    - The *calibrateCamera* function takes in the object points and image points along with the image shape.
        - It returns the ***distortion coefficients*** and the ***camera matrix*** that can be used to transform the 3D object points to 2D image points.
        - It also returns the position of the camera in the world with values for the ***rotation*** and ***translation vectors***.
- To undistort an image, we can use the OpenCV function *undistort*.
    - This takes in the distorted image, camera matrix and distortion coefficients as inputs and returns the undistorted image as the output.

Code example

```
import numpy as np
import cv2
import glob
import pickle
import matplotlib.pyplot as plt


#prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*8,3), np.float32)

#Generate x,y coordinates
objp[:,:2] = np.mgrid[0:8, 0:6].T.reshape(-1,2)
```

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

# Make a list of calibration images
images = glob.glob('calibration/image*.jpg')


# Step through the list and search for chessboard corners
for idx, fname in enumerate(images):
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (8,6), None)

    # If found, add object points, image points
    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)



# Test undistortion on an image
img = cv2.imread('calibration/test_image.jpg')
img_size = (img.shape[1], img.shape[0])


# Do camera calibration given the object points and image points
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
                                                   imgpoints,
                                                   img_size,
                                                    None,None)



#Undistort the test image
dst = cv2.undistort(img, mtx, dist, None, mtx)
cv2.imwrite('calibration/test_undist.jpg',dst)


# Save the camera calibration result for later use with other images
dist_pickle = {}
dist_pickle["mtx"] = mtx
dist_pickle["dist"] = dist
pickle.dump( dist_pickle, open("calibration/wide_dist_pickle.p","wb"))
```

4 Lane Lines and curvature

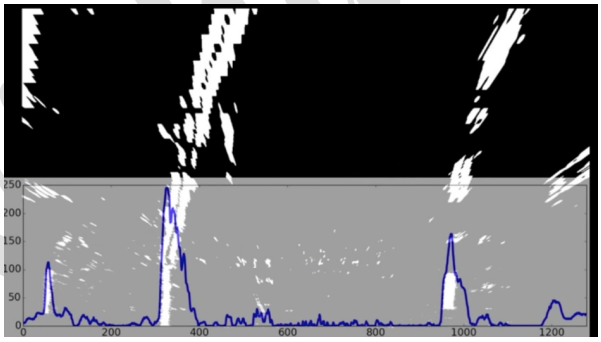
After pre-processing an image to detect lanes, we may need to approximate the lane. With Hough transform, we get a straight line. Here, we look at some approaches to fit a polynomial and detect curvature of the lane.

Thresholded and perspective transformed image

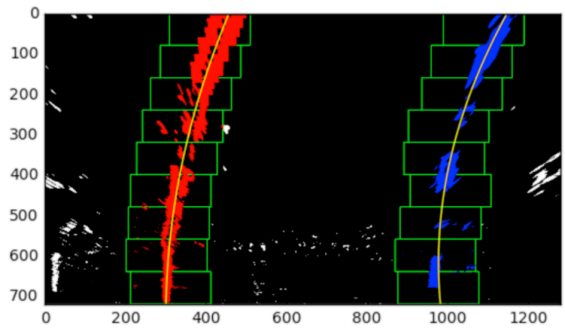
After applying calibration, thresholding, and a perspective transform to a road image, we get a binary image where the lane lines stand out clearly. However, we still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

Take a histogram along all the columns in the lower half of the image.

- The two most prominent peaks in this **histogram** will be good indicators of the x-position of the base of the lane lines.



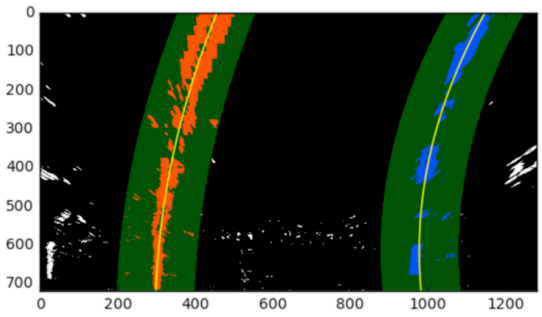
Use a **sliding window** moving upwards to determine pixels of interest along the lane.



Once we have the left lane and right lane indices, we can **fit a polynomial** using np.polyfit().

Now, running the above algorithm for every frame is unnecessary. Lane lines don't dramatically shift frame to frame. So, once we have detected lane lines, we can do the below going forward:

- Instead of starting with a blind search, search in a margin around the previous lane line position.
- For example, we search the green shaded region below for the lane lines. This makes the search more targeted.



Once we have our polynomial, we can measure curvature:

Radius of curvature is given by:

$$R = \frac{\left(1+\left(\frac{dx}{dy}\right)^2\right)^{\frac{3}{2}}}{\left|\frac{d^2x}{dy^2}\right|}$$

Now, the polynomial we fit can be of any degree depending on the problem. In this case, lets assume the polynomial is

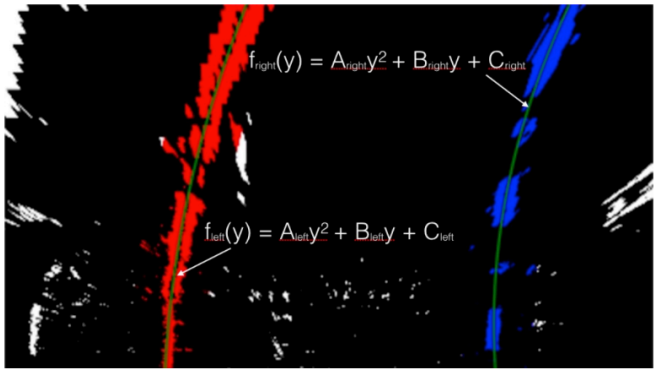
$$f(y) = Ay^2 + By + C$$

Note: We are fitting for f(y), rather than f(x), because the lane lines in the warped image are near vertical and may have the same x value for more than one y value.

This gives us:

$$f`(y) = 2Ay + B$$
$$f``(y) = 2A$$

We can substitute the above in the equation of R to compute the radius of curvature.



As seen above, we get a left and right polynomial.
We can compute R for each polynomial and take their average as the final radius of curvature.

5 Additional References

- Fully Convolutional Networks for Semantic Segmentation
- Lane Detection with Deep Learning (Part 1)
- Lane Detection with Deep Learning (Part 2)
- VPGNet: Vanishing Point Guided Network for Lane and Road Marking Detection and Recognition
- Learning to Map Vehicles into Bird's Eye View
- End-to-End Deep Learning for Self-Driving Cars

End of Document