lanes

March 23, 2019

1 Find Lanes for Self-Driving Cars using OpenCV python

1.0.1 Loading with all the required packages:

```
In [1]: import cv2
        import numpy as np
        import matplotlib.pyplot as plt
```

1.0.2 Step 1: Loading image and converting to gray scale

Next we convert the image to gray scale and have a look at it. Notice that i have created the copy of the image and performing any processing on that image copy. Since processing the images may reflect changes in the original image which we don't want. Converting to gray scale reduces the image dimension from 3(R,G,B) to single dimension. Detection of edges is much more accurate when done in gray scale images. Hence this conversion.

1.0.3 Step 2: Smoothing the gray scale image

Once the image has been converted to gray scale, we then perform image smoothing. Images usually contain a lot of noise which make it difficult for algorithms to detect the edges. Smoothing the images does the detection task easier for us. You would be thinking what the (5,5) means in the first line of the code. Let me explain what image smoothing means.

The Gaussian blur performs smoothing by averaging out the pixel intensity values through a weighted average kernel. (5,5) is the kernel size that is used to perform the smoothing operation. You are free to try out with different kernel size.

1.0.4 Step 3: Detecting edges using canny edge detection algorithm

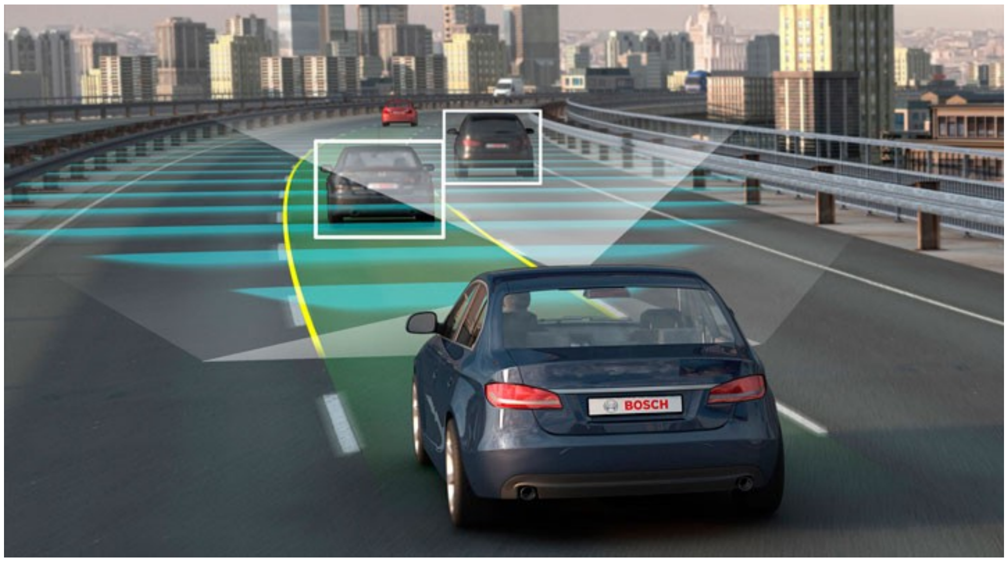
The purpose of edge detection in general is to significantly reduce the amount of data in an image, while preserving the structural properties to be used for further image processing.

The algorithm runs in 5 separate steps:

Smoothing: Blurring of the image to remove noise.

Finding gradients: The edges should be marked where the gradients of the image has large magnitudes.

Non-maximum suppression: Only local maxima should be marked as edges.



title

Double thresholding : Potential edges are determined by thresholding.

Edge tracking by hysteresis: Final edges are determined by suppressing all edges that are not connected to a very certain (strong) edge.

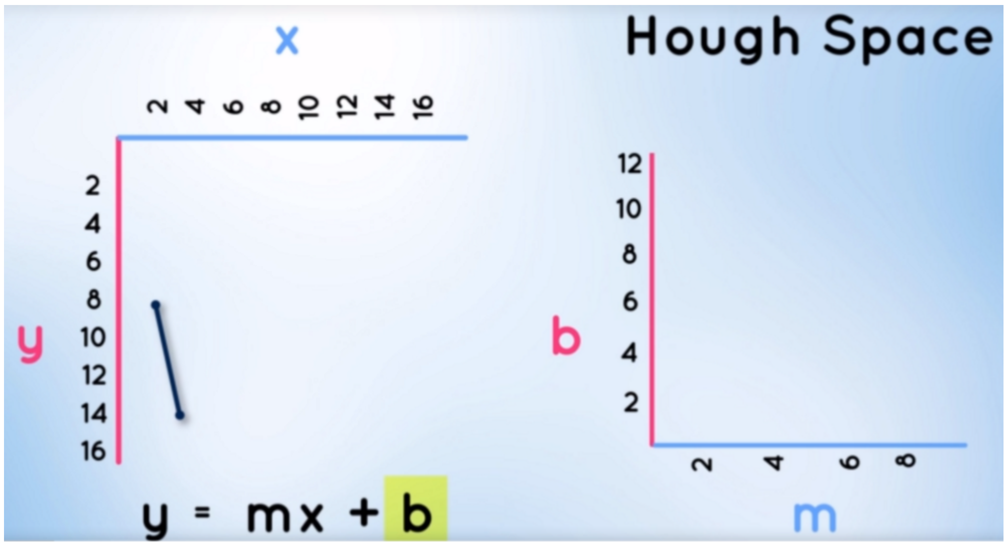
```
In [2]: #image=cv2.imread('test_image.jpg')
        #lane_image=np.copy(image)
        #canny_image=canny(lane_image)
        #cropped_image=region_of_interest(canny_image)

        def canny(image):
            gray=cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
            blur=cv2.GaussianBlur(gray,(5,5),0)
            canny=cv2.Canny(blur,50,150)
            return canny
```

1.0.5 Step 4 : Creating masked image using region of interest

Next we create an image mask which is a black image having same dimension as that of the original image. We then take the original image and identify the region of interest for us. In this case we have a triangle in place for the ROI which is then superimposed on the mask using opencv fillpoly() function and a region of interest mask is generated.

1.0.6 ROI mask

Now we have the ROI mask and the canny image as well. The next step is to perform a bitwise and operator between both the images to get the Region of interest canny image. The bit wise and operator looks like 00000000 in the black section and 11111111 in the white section of the above image. When a bitwise_and operation is performed between the values and the values in the canny image the and operation with 00000000 will always generate a 00000000 that is a black pixel



Image

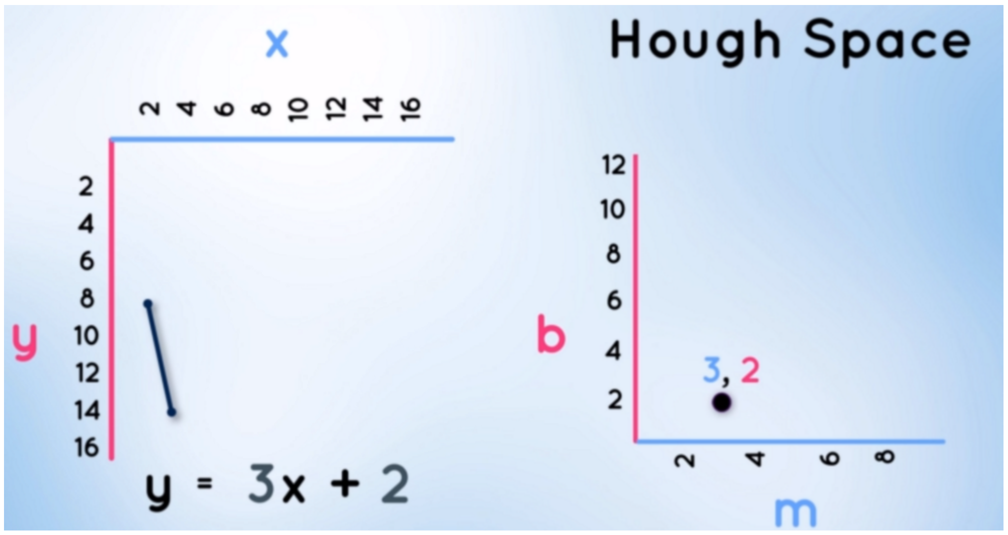
in the resulting image whereas a 11111111 will not have any effect on the resulting image and will generate a similar structure as the of the canny image in the white section.

```
In [3]: def region_of_interest(image):
            height=image.shape[0]
            polygons=np.array([[(200,height),(1100,height),(550,250)]])
            mask=np.zeros_like(image)
            cv2.fillPoly(mask,polygons,255)
            masked_image=cv2.bitwise_and(image,mask)
            return masked_image
```

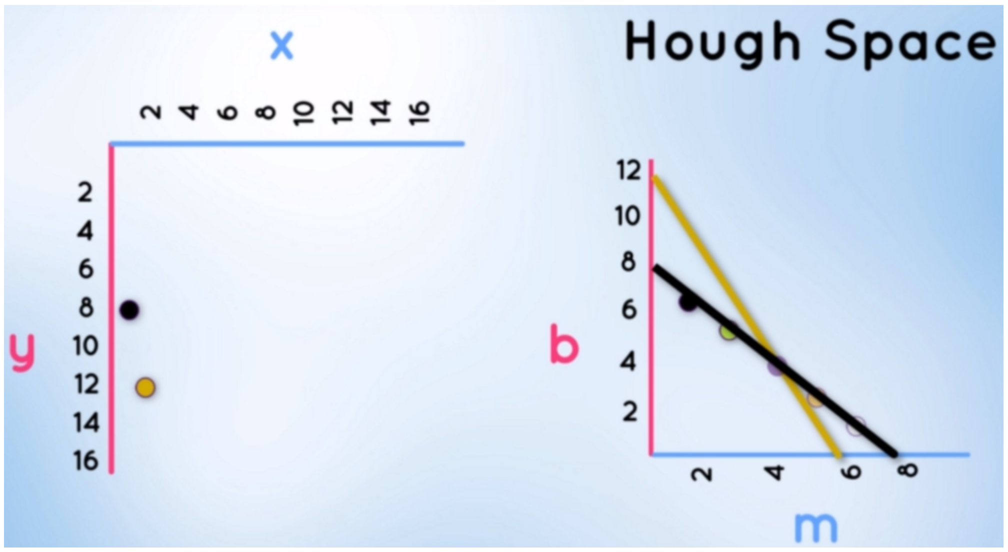
1.0.7 Step 5 : Applying Hough transform on the image

Before applying the hough transform method let us go through some of the theory behind it. The images in the image space is represented by the usual x and y axis as a 2D matrix of rows and columns that represents dimension of the image. The image in image space is represented by y = mx + b if we plot m and b as separate parameters along the x and y axis, this is called the parameter or hough space.

```
In [4]: # Define the Hough transform parameters
        # Make a blank the same size as our image to draw on
        #rho = 1 # distance resolution in pixels of the Hough grid
        #theta = np.pi/180 # angular resolution in radians of the Hough grid
        #threshold = 2     # minimum number of votes (intersections in Hough grid cell)
        #min_line_length = 40 #minimum number of pixels making up a line
        #max_line_gap = 5    # maximum gap in pixels between connectable line segments

        # Run Hough on edge detected image
        # Output "lines" is an array containing endpoints of detected line segments
        #lines = cv2.HoughLinesP(cropped_image, rho, theta, threshold, np.array([]), min_line_
```



title



Image

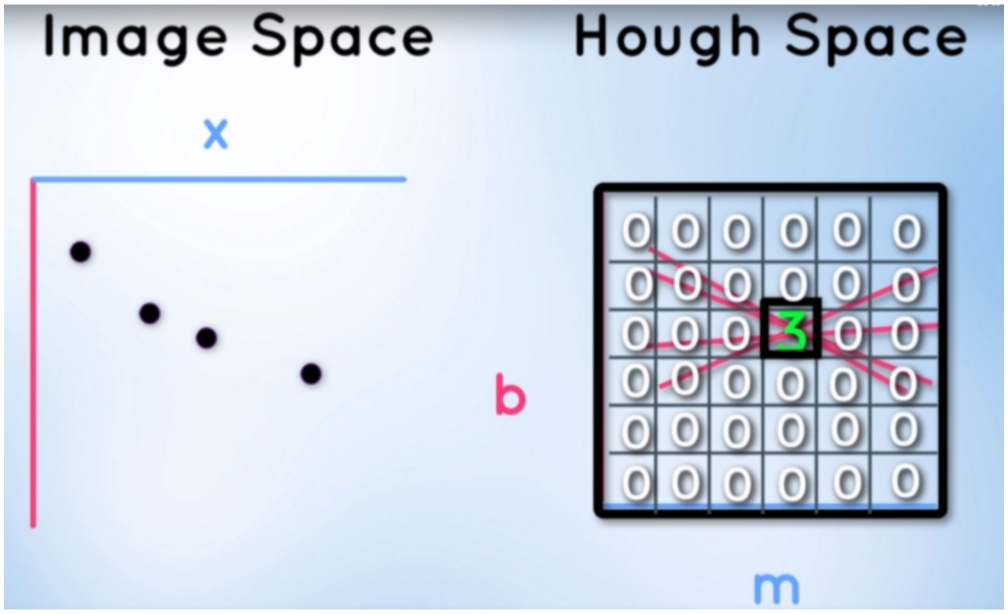


Image Space
Hough Space
x
b
m
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 3 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

Image

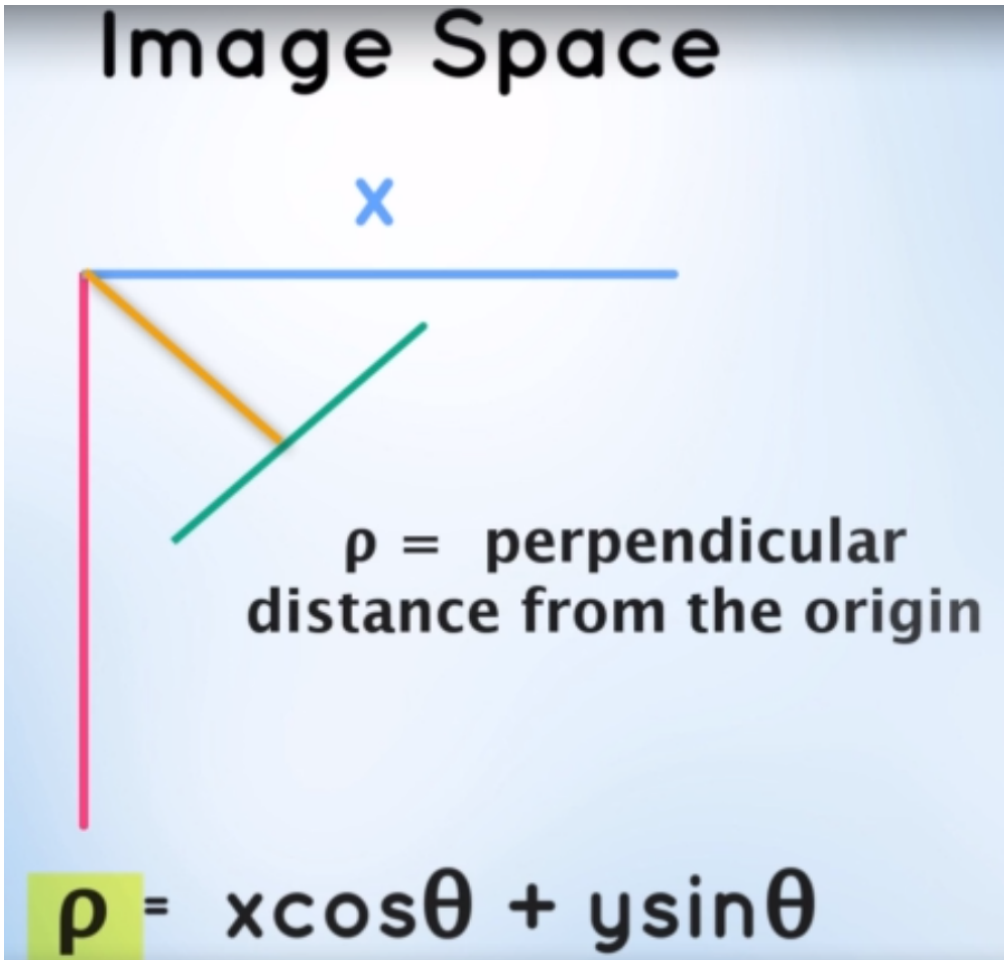
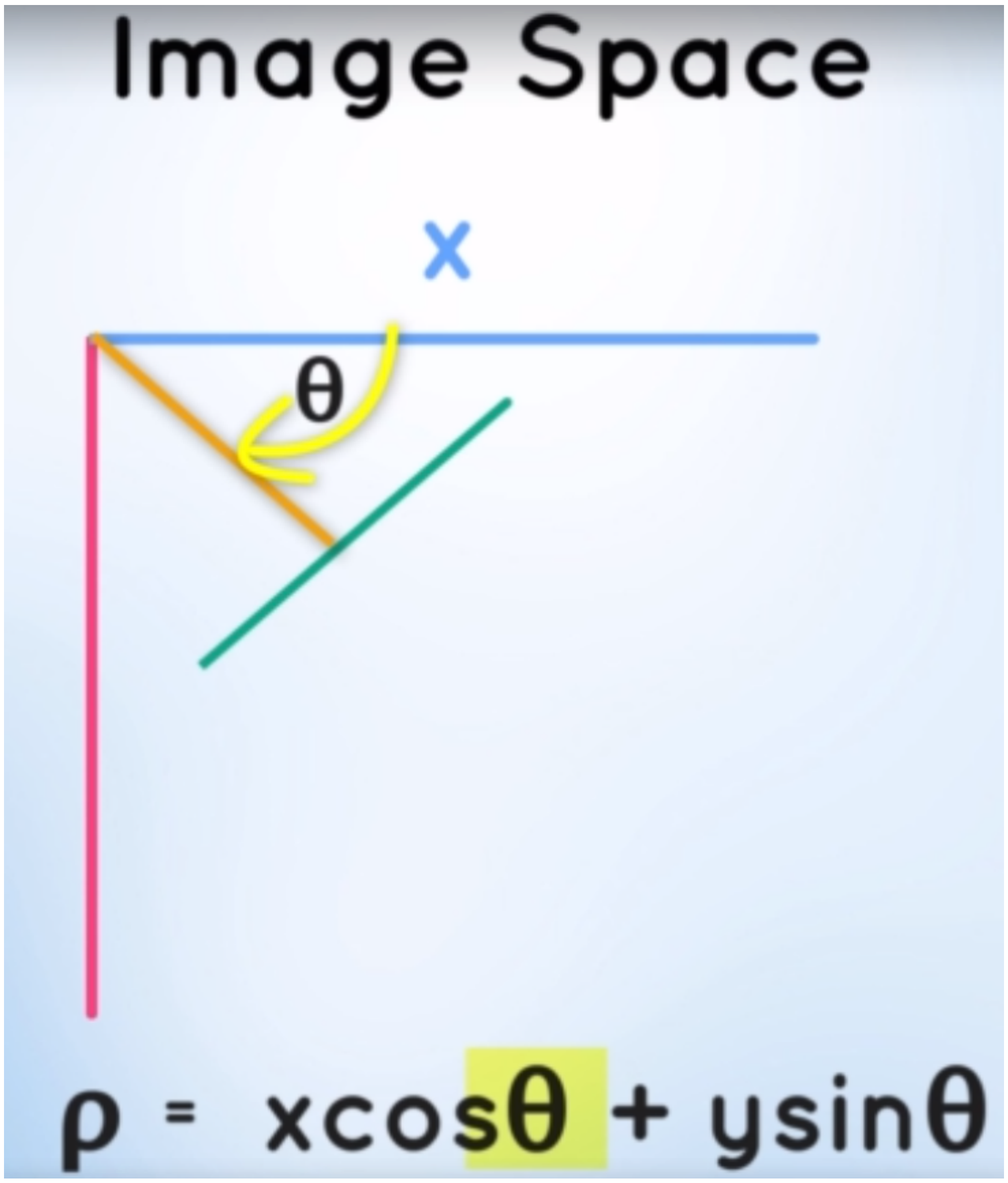
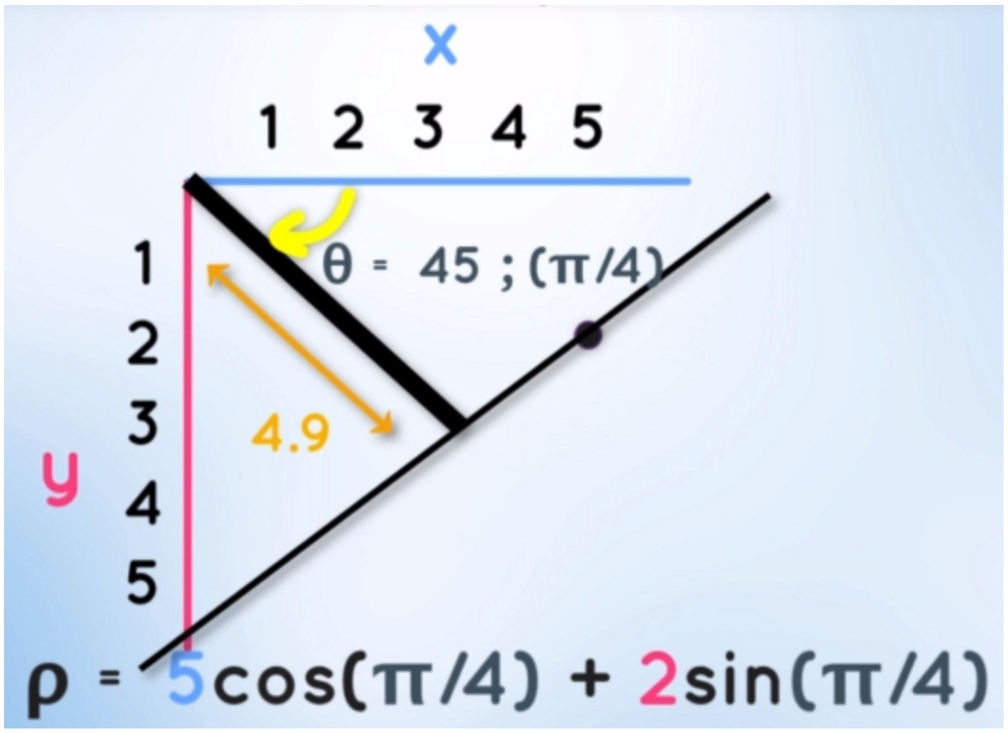


Image Space
x
ρ = perpendicular distance from the origin
ρ = xcosθ + ysinθ

Image



Image


x
1 2 3 4 5
y
1
2
3
4
5
θ = 45 ; (π/4)
4.9
ρ = 5cos(π/4) + 2sin(π/4)

Image

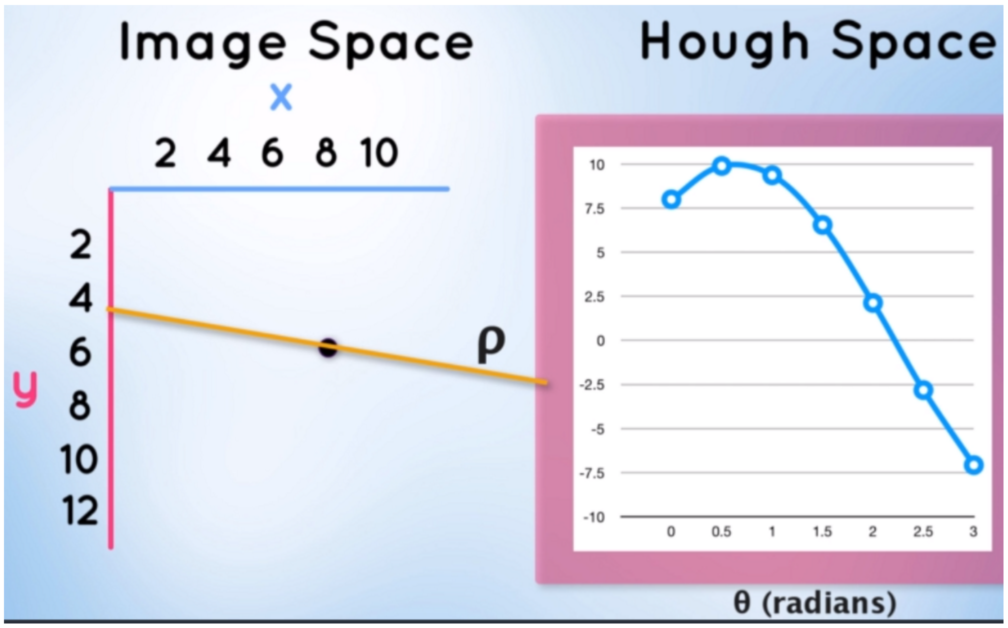
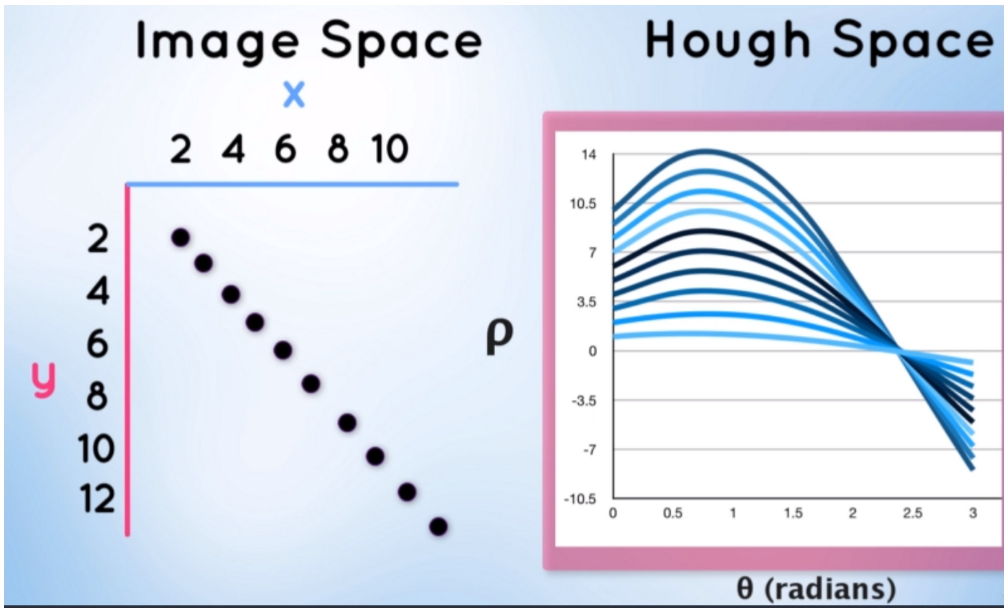


Image Space
Hough Space
x
2 4 6 8 10
y
2
4
6
8
10
12
ρ
10
7.5
5
2.5
0
-2.5
-5
-7.5
-10
0 0.5 1 1.5 2 2.5 3
θ (radians)

Image



Image

```
#averaged_lines=average_slope_intercept(lane_image, lines)
#line_image=display_lines(lane_image,averaged_lines)

#combo_image = cv2.addWeighted(lane_image, 0.8, line_image, 1, 1)
#cv2.imshow("result",combo_image)
#cv2.waitKey(0)
#cv2.destroyAllWindows()
```

```
In [5]: def make_coordinates(image,line_parameters):
            (slope,intercept)=line_parameters
            y1=image.shape[0]
            y2=int(y1*(3/5))
            x1=int((y1-intercept)/slope)
            x2=int((y2-intercept)/slope)
            return np.array([x1,y1,x2,y2])
```

```
In [6]: def average_slope_intercept(image, lines):
            left_fit=[]
            right_fit=[]
            for line in lines:
                x1,y1,x2,y2=line.reshape(4)
                parameters=np.polyfit((x1,x2),(y1,y2),1)
                slope=parameters[0]
                intercept=parameters[1]
                if slope<0:
                    left_fit.append((slope,intercept))
                else:
                    right_fit.append((slope,intercept))
```

```
        left_fit_average=np.average(left_fit,axis=0)
        right_fit_average=np.average(right_fit,axis=0)
        left_line=make_coordinates(image,left_fit_average)
        right_line=make_coordinates(image,right_fit_average)
        return np.array([left_line,right_line])
```

```
In [7]: def display_lines(image,lines):
            line_image=np.zeros_like(image)
            if lines is not None:
                for x1,y1,x2,y2 in lines:
                    cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),10)
            return line_image
```

1.0.8 Final image

Finally we are done with the image section. How you got an idea of how an image needs to be processed along with the computations happening in the background. Now we will extend this to videos. We will begin by importing the required packages for video analysis and processing in python

```
In [9]: cap=cv2.VideoCapture("test2.mp4")
        while(cap.isOpened()):
            _,frame=cap.read()
            canny_image=canny(frame)
            cropped_image=region_of_interest(canny_image)
            # Define the Hough transform parameters
            # Make a blank the same size as our image to draw on
            rho = 1 # distance resolution in pixels of the Hough grid
            theta = np.pi/180 # angular resolution in radians of the Hough grid
            threshold = 2     # minimum number of votes (intersections in Hough grid cell)
            min_line_length = 40 #minimum number of pixels making up a line
            max_line_gap = 5    # maximum gap in pixels between connectable line segments

            # Run Hough on edge detected image
            # Output "lines" is an array containing endpoints of detected line segments
            lines = cv2.HoughLinesP(cropped_image, rho, theta, threshold, np.array([]), min_li
            averaged_lines=average_slope_intercept(frame, lines)
            line_image=display_lines(frame,averaged_lines)
            combo_image = cv2.addWeighted(frame, 0.8, line_image, 1, 1)
            cv2.imshow("result",combo_image)
            if cv2.waitKey(1)==ord('q'):
                break
        cap.release()
        cv2.destroyAllWindows()
```

1.0.9 References:

Self Driving Car Engineer Course | Udacity
Build the future of transportation with Self Driving Car skillsin.udacity.com