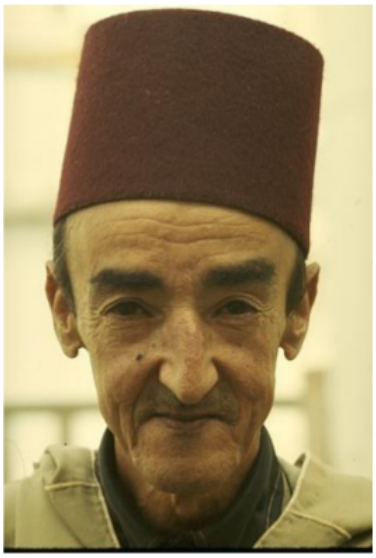
CV Assignment 1

Name 1: Abdelrahman Salem Mohamed
ID 1: 6309
Name 2: Reem Abdelhalim
ID 2: 6114

1 Part I: Applying Image Processing Filters for Image Cartoonifying

Original Image:



```
Orginal Image Shape:
(397, 263, 3)
```

Resize Image:

To 360x240 pixels

```
New Image Shape:
(360, 240, 3)
```

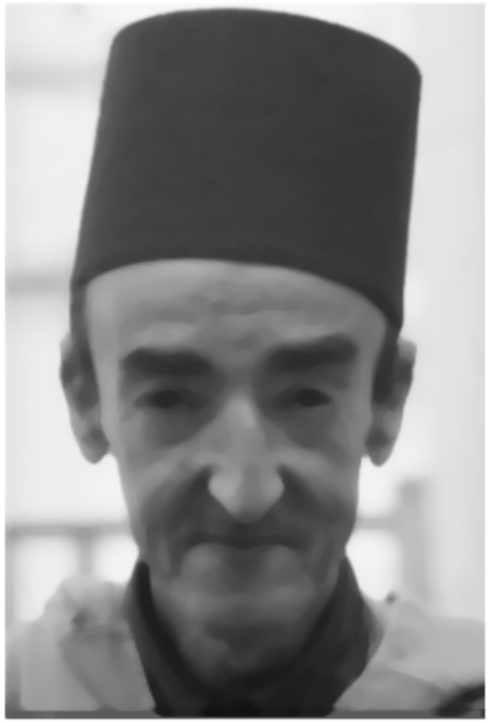
1.1 Generating a black-and-white sketch

Convert BGR IMAGE to Grayscale as
Laplacian filters use grayscale images, we must convert from OpenCV's defa ult BGR format to Grayscale.

1.1.1 Noise Reduction Using Median Filter

removing noise while keeping the edge sharp using cv2.medianBlur
as we tried many kernels as shown below

`kernel Size = (7, 7) is chosen`



1.1.2 Edge Detection Using Laplacian Filter

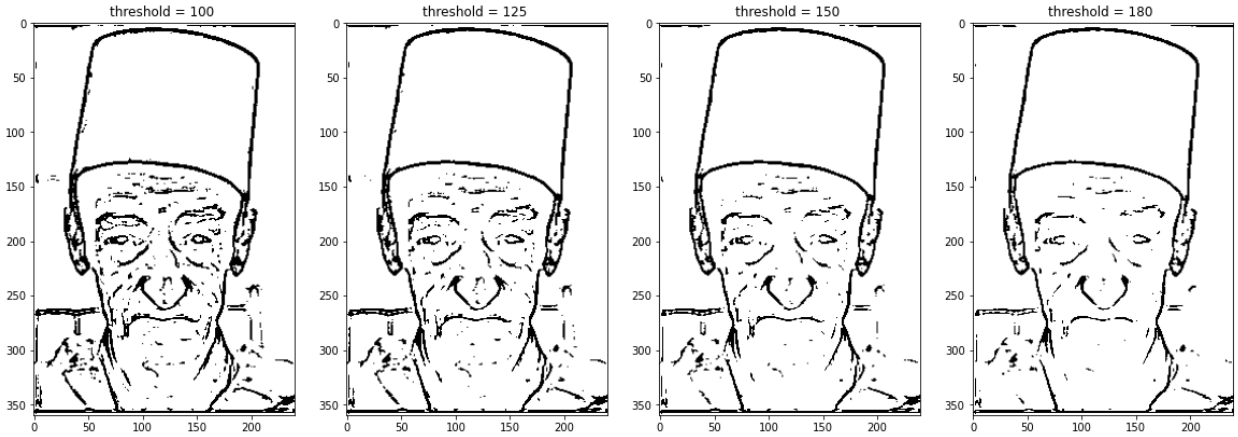
The Laplacian filter produces edges with varying brightness, so to make the edges look more like a sketch we apply a binary threshold to make the edges either white or black.

`as we tried many kernels as shown below`

```
kernel Size = (5, 5) is chosen
```

Binary Thresholding:

Cv2.threshholding is used then cv2.bitwise_not was used as the output image in the first place the edges is white and background is black and as the required the opposite so not was used to exchange black and white, many thresholds are tried as shown below



```
Threshold [125,255] is chosen
```

1.2 Generating a color painting and a cartoon

The most important trick we can use is to perform bilateral filtering at alower resolution. It will have a si milar effect as at full resolution. Then image will be resized to 75% of its size.

```
img_lowRes = cv2.resize(img_resized, (180, 270))
```

Rather than applying a large bilateral filter, we will apply many small bilateral filters to produce a strong cartoon effect in less time. We will truncate the filter so that instead of performing a whole filter (for example, a filter size of 21 x 21), it just uses the minimum filter size needed for a convincing result (for example, with a filter size of just 9 x 9). We have four parameters that control the bilateral filter: color strength, positional strength, size, and repetition count.

Then REPETITIONS = 5 and KERNAL_SIZE = 9 are chosen.

sigmaColor – A variable of the type integer representing the filter sigma in the color space.

sigmaSpace – A variable of the type integer representing the filter sigma in the coordinate space.

In order to test which sigmaColor and sigmaSpace is the best for our image, we tried many values as shown below:

Test Different Sigma Color:

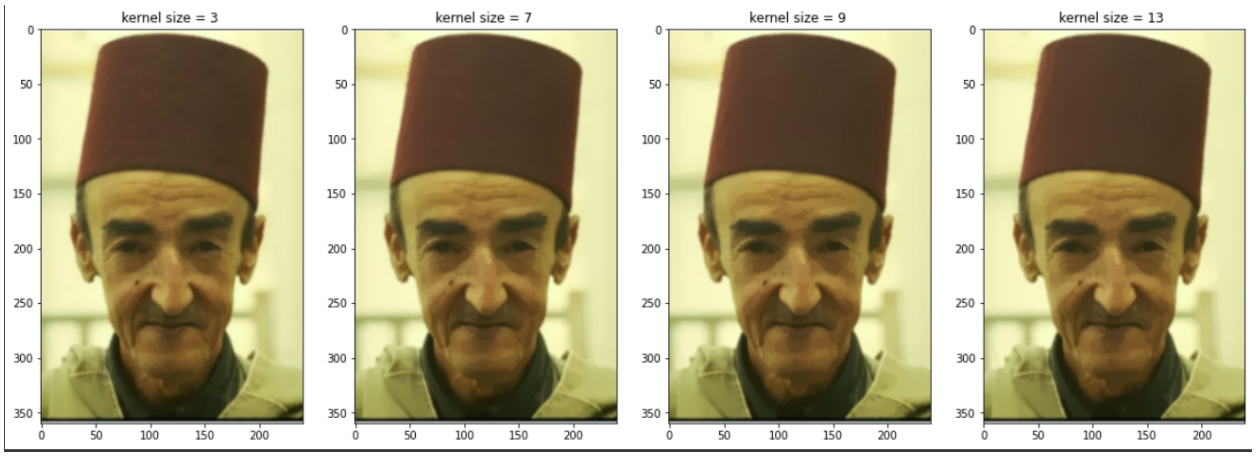
sigmaColor - Sigma of gray/color space.

sigma color '7' is chosen

Test Different Sigma Space:

sigmaSpace - Large value means farther pixels influence each other (as lon g as the colors are close enough)

Test Different Kernel Size for Bilateral Filter:



Kernel Size '9' is chosen

Applying different number of repetitions for bilateral filter:

No big difference appeared so '5' repetitions is chosen only

Apply Bilateral filter

KERNAL_SIZE = 9 SIGMA_COLOR = 7 SIGMA_SPACE = 7 REPETITIONS = 5

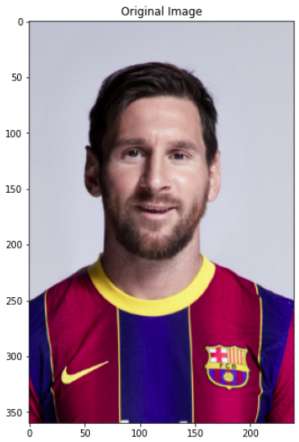
```
[23] SIGMA_COLOR = 7
     SIGMA_SPACE = 7

     img_bilateralFilter = cv2.bilateralFilter(img_lowRes, KERNAL_SIZE, SIGMA_COLOR, SIGMA_SPACE)
     for _ in range(REPETITIONS-1):
       img_bilateralFilter = cv2.bilateralFilter(img_bilateralFilter, KERNAL_SIZE, SIGMA_COLOR, SIGMA_SPACE)

     img_bilateralFilter = cv2.resize(img_bilateralFilter,dsize=(240,360))
     cv2_imshow(img_bilateralFilter)
```

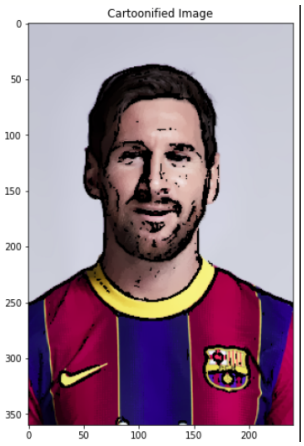
overlay the edge mask onto the bilateral filter output:

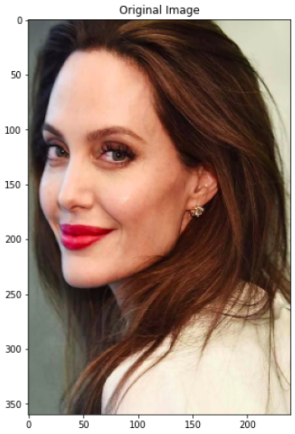
```
black_img = np.zeros(img_resized.shape)
for i in range(3):
  black_img[:,:,i] = cv2.bitwise_and(img_bilateralFilter[:,:,i], img_mask)
cv2_imshow(black_img)
```

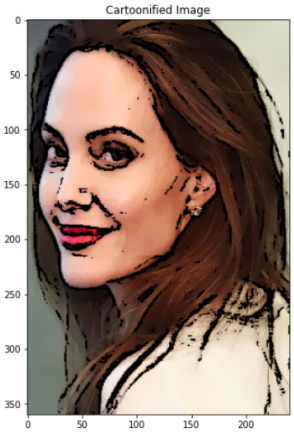
First a black image is generated then bitwise_and is applied between image edge mask and output from bilateral filter.

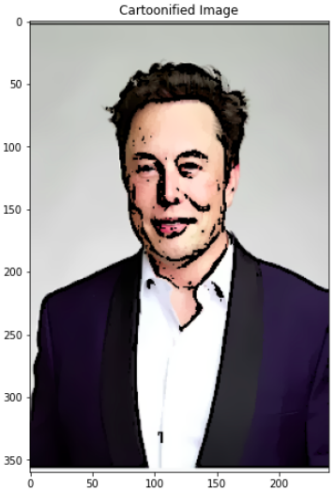
Applying Cartoonified operations of some random images



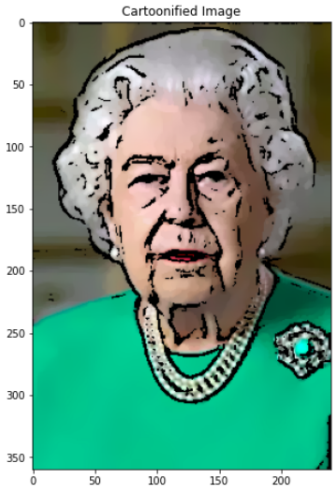









Original Image
0
50
100
150
200
250
300
350
0
50
100
150
200


Cartoonified Image
0
50
100
150
200
250
300
350
0
50
100
150
200

Part 2: Hough Transform

2.1 -Introduction

The Hough transform can be used to determine the parameters of a line when a number of points that fall on it are known. The normal form of a line can be described with the following equation: x cos θ + y sin θ = ρ where ρ is the length of a line that starts from the origin and perpendicular to the required line, and θ is its inclination. The true parameters ρ and θ will get maximum votes from the line points, and can be found with a Hough accumulation array.

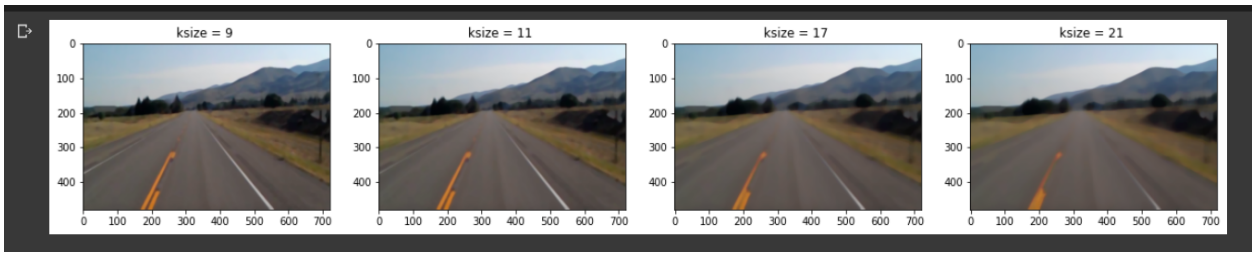
2.2 - Procedures

2.2.1 - Resizing the original image

from (704, 1279, 3) to (480, 720, 3)

2.2.2 - Removing noise using median filter

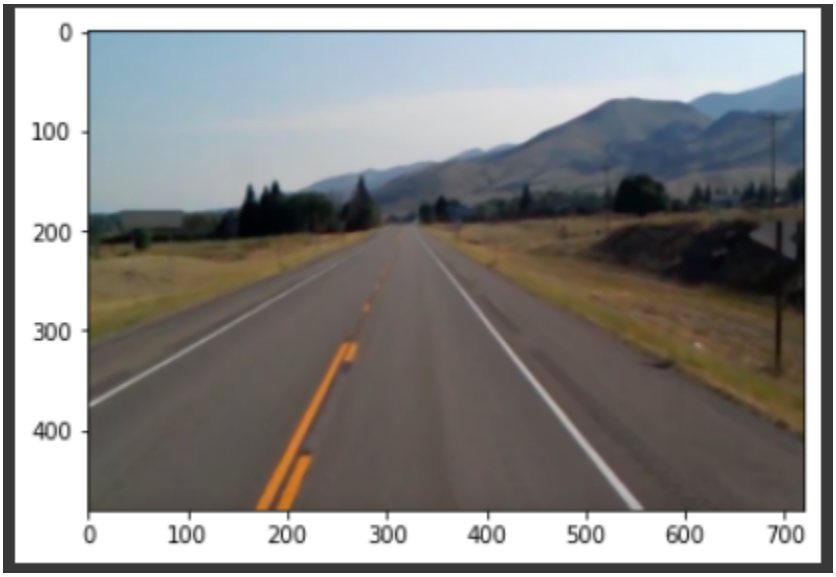
to reduce the noise in the image without losing too much information we performed median filter with different size kernel



as seen in the image higher kernel size gives more blurring effect and losing informations of the image

When we used ksize=11 the road lane line almost faded and the canny edge detection algorithm couldn't identify it.

so we decided to use ksize = 5 for more small noises and didn't affect the overall pixels



2.2.3 - Edges Detection

Canny Algorithm has been used for our road image to identify the lanes lines, Canny algorithm mainly composed of 5 steps:

1. Noise Reduction: applying gaussian blur
2. Gradient Calculation: sobel kernels in x and y directions for every pixel
3. Non-Maximum Suppression
4. Double Threshold: identify strong, weak and non-relevant pixels
5. Edge Tracking by Hysteresis: transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one

One last important thing to mention, is that the algorithm is based on grayscale pictures. Therefore, the prerequisite is to convert the image to grayscale before following the above-mentioned steps.

2.2.4 - Region of Interest

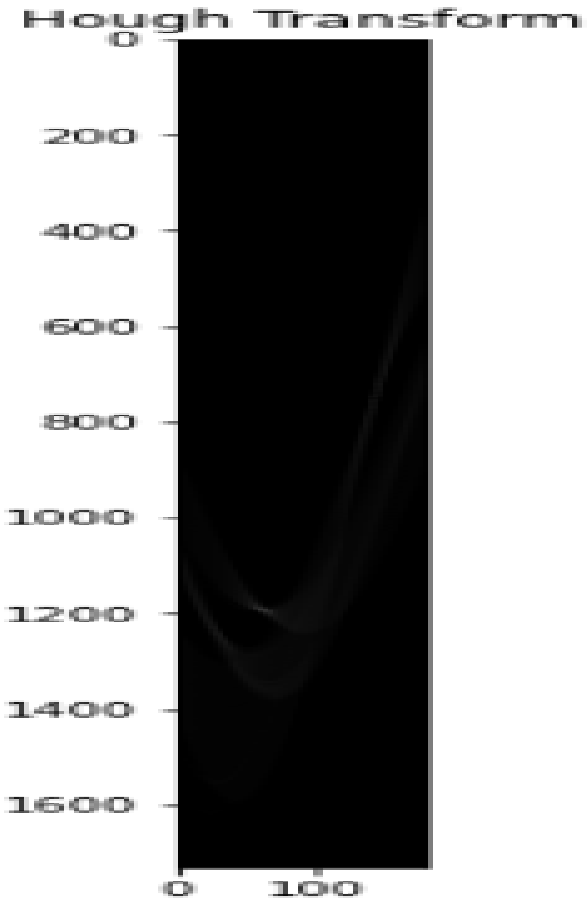
for sake of simplicity we have assumed that the image snap take from driver POV so most of the upper image is contains some edges irrelevant to the road lanes so we set all pixels above the half of the image to zero:

```
edged_img[:240,:] = 0
```

2.2.5 - Hough Algorithm

The main aim is to fill the accumulator matrix (mapping between image pixels and lines in the image using line normal form to (p, theta), then select the most voted pixel to identify as linse and map back to the original image.

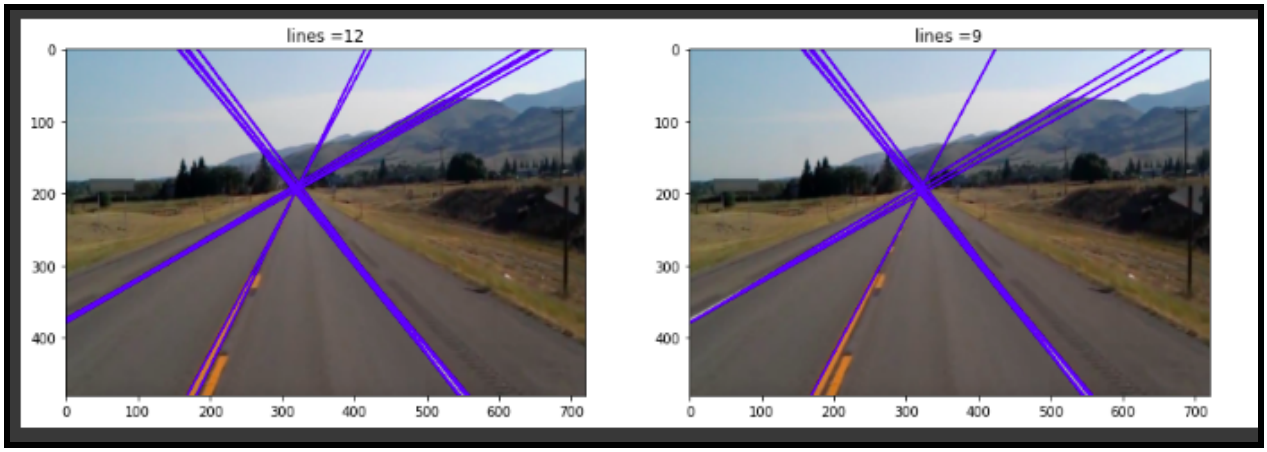
```
def hough_transform(image):
  H,W = image.shape

  # Max diatance is diagonal one
  Maxdist = int(np.ceil(np.sqrt(H**2 + W ** 2)))
  # Theta in range from 0 to 180 degrees
  thetas = np.deg2rad(np.arange(0, 180))
  # Range of radius
  rs = np.linspace(-Maxdist, Maxdist, 2*Maxdist)
  accumulator = np.zeros((2 * Maxdist, len(thetas)),dtype=np.uint64)
  y_idxs, x_idxs = np.nonzero(image)  # (row, col) indexes to edges

  for i in range(len(x_idxs)):
    x = x_idxs[i]
    y = y_idxs[i]
    for k in range(len(thetas)):
      p = np.round(x*np.cos(thetas[k]) + y*np.sin(thetas[k])) + Maxdist # to avoid negative values
      accumulator[int(p),k] += 1
  return accumulator, thetas, rs
```

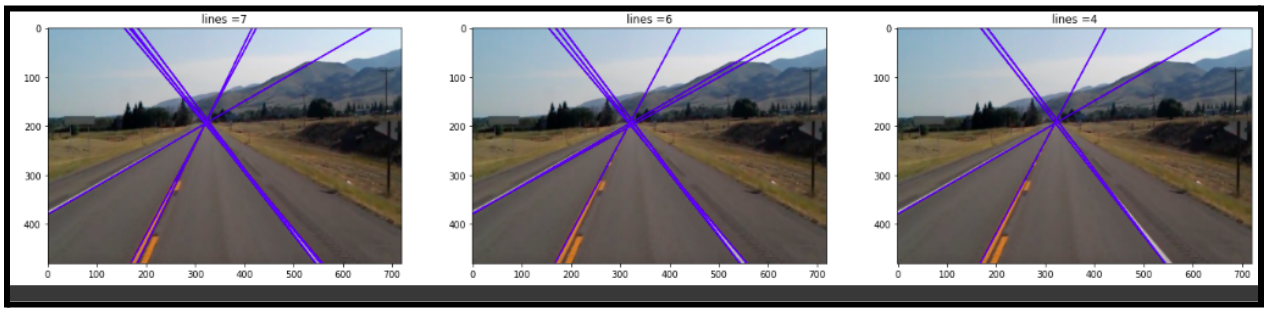


2.2.6 - Refine Coordinates

as results of image resolutions maybe some pixels near to each other map to 2 lines, to solve this problem we performed some window filter techniques to select the local maximum in predefined window size, select the maximum number locally inside the window and assign the results as zero, so when applying high kernel size it will consider more local scale inside the accumulator matrix

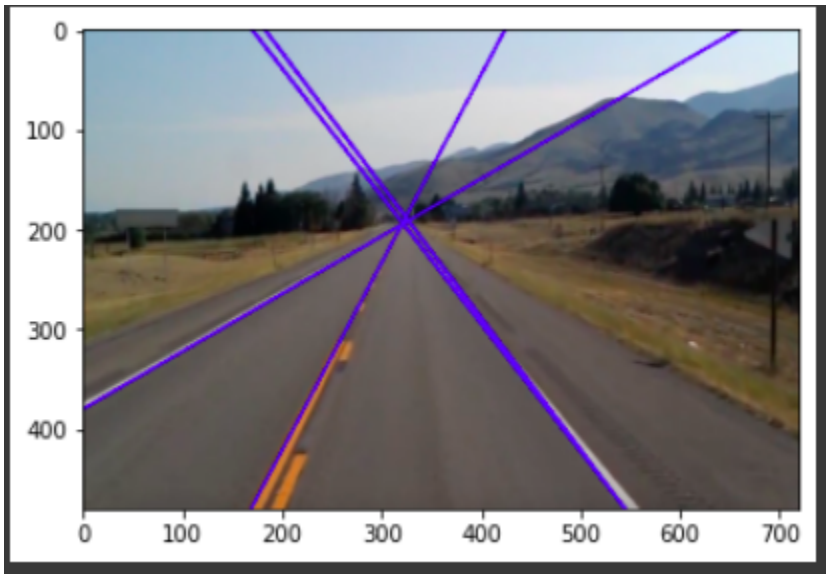
```
def remove_neighbour_lines(accumulator,neigh=3):
  acc_copy = accumulator.copy()
  for i in range(0,accumulator.shape[0],neigh):
    if i+neigh > accumulator.shape[0]:
      continue
    for j in range(0,accumulator.shape[1],neigh):
      if j+neigh > accumulator.shape[1]:
        continue
      filter = np.zeros((neigh,neigh))
      idx = np.argmax(accumulator[i:i+neigh,j:j+neigh])
      x,y = np.unravel_index(idx, (neigh,neigh))
      filter[x][y] = 1
      acc_copy[i:i+neigh,j:j+neigh]= accumulator[i:i+neigh,j:j+neigh]*filter
  return acc_copy
```





2.2.7 - Draw line on the Original Image

to add the lines from (p,theta) to (x,y) we get two points in the x,y domain and calculate the slope and the intercept part to get the equation of straight line on the image



2.3 Source Code

[Notebook]