Computer Vision

Assignment (1)

Team Members:

Mennatallah Mostafa	6234
Esraa Mahmoud	6597
Seif Mohamed	6624

Part 1: Cartooning images

Problem Overview:

Image Cartoonyfing using filters and OpenCV libraries

Approach:

The process to create a cartoon effect image can be initially branched into 3 parts:

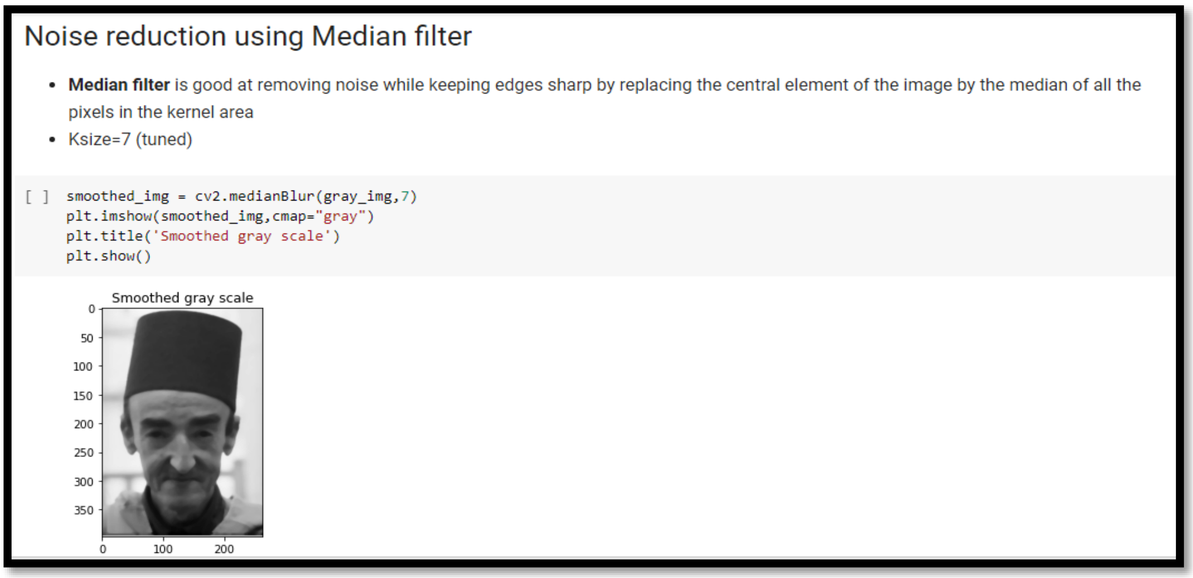
1) To detect, blur and bold the edges of the actual RGB color image. (Black and white sketch)
2) To smooth the flat areas. (Color painting)
3) To overlay the edge mask ”sketch” onto the bilateral filter ”painting”. (Cartoon effect)

1-Black and White sketch

a. Converting from BGR to RGB image

b. Converting from RGB to gray scale image

c. Noise reduction using Median Filter



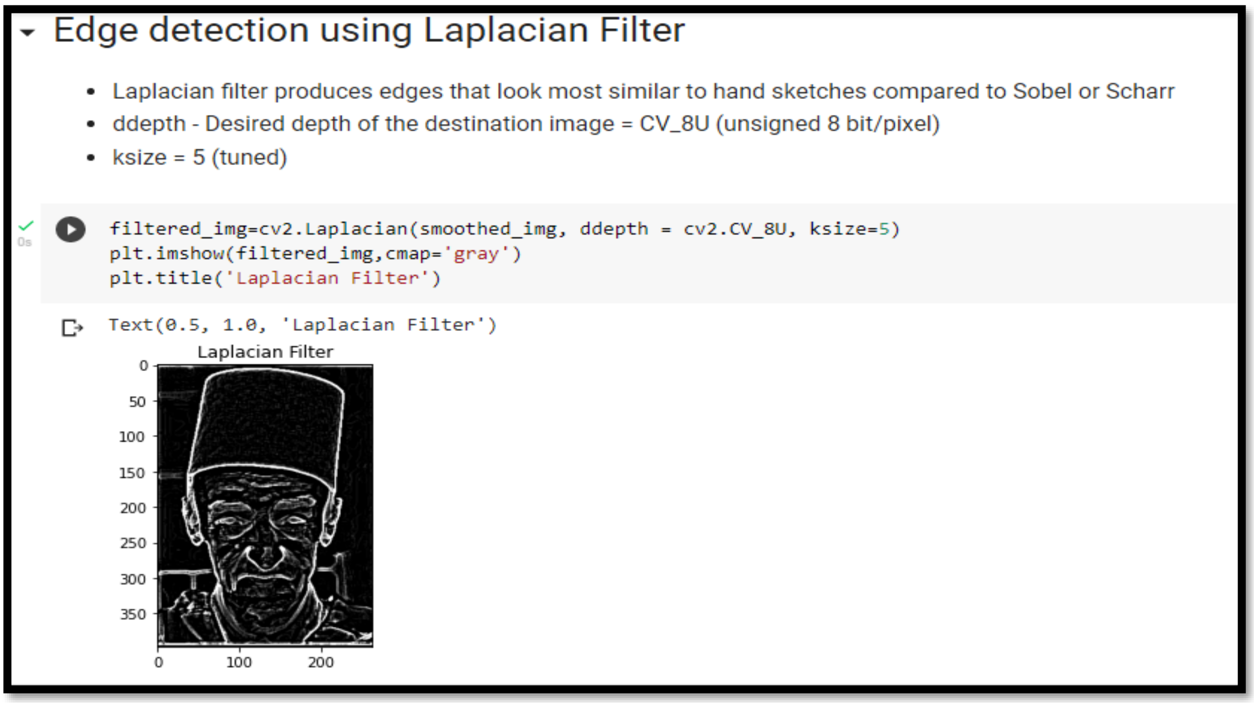
Noise reduction using Median filter

- **Median filter** is good at removing noise while keeping edges sharp by replacing the central element of the image by the median of all the pixels in the kernel area
- Ksize=7 (tuned)

```
[ ] smoothed_img = cv2.medianBlur(gray_img,7)
    plt.imshow(smoothed_img,cmap="gray")
    plt.title('Smoothed gray scale')
    plt.show()
```

Output: Smoothed Gray scale image with blurring effect and edges preserved. Increasing the kernel size increases the blurring effect and vanishes the edges

d. Edge detection using Laplacian Filter



Edge detection using Laplacian Filter

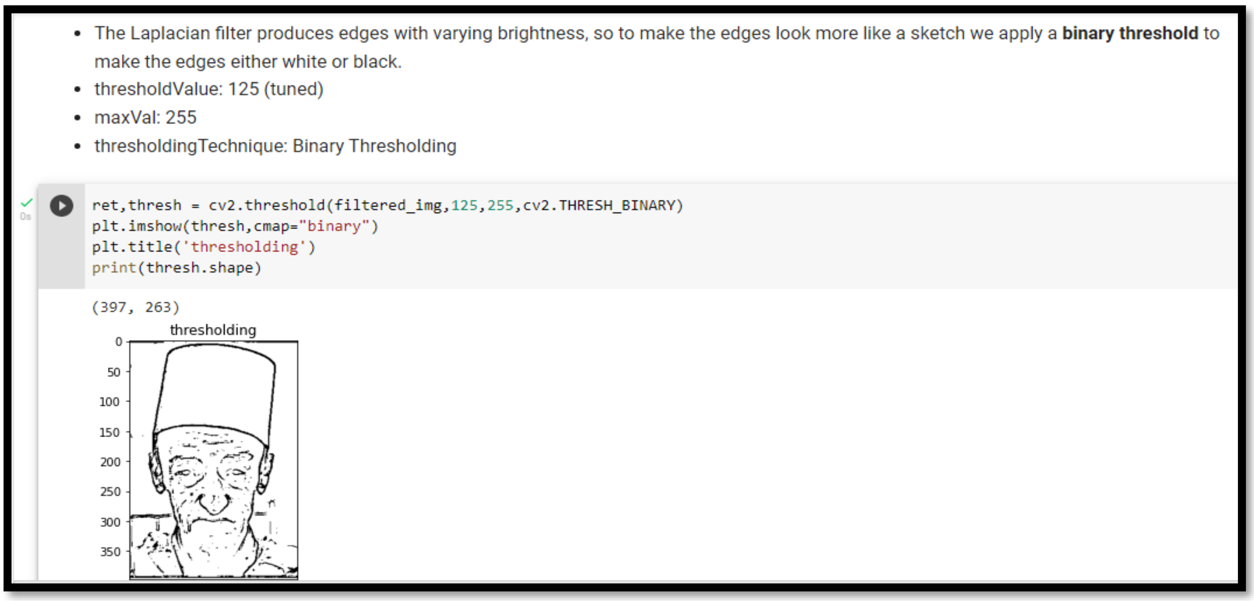
- Laplacian filter produces edges that look most similar to hand sketches compared to Sobel or Scharr
- ddepth - Desired depth of the destination image = CV_8U (unsigned 8 bit/pixel)
- ksize = 5 (tuned)

```
filtered_img=cv2.Laplacian(smoothed_img, ddepth = cv2.CV_8U, ksize=5)
plt.imshow(filtered_img,cmap='gray')
plt.title('Laplacian Filter')
```

```
Text(0.5, 1.0, 'Laplacian Filter')
```

Output: Bold edged image similar to hand sketches. Depth is 8bits/pixel (0-255).As the Kernel size increases undesired details and edges are detected.

e. Edges thresholding



- 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.
- thresholdValue: 125 (tuned)
- maxVal: 255
- thresholdingTechnique: Binary Thresholding

```
ret,thresh = cv2.threshold(filtered_img,125,255,cv2.THRESH_BINARY)
plt.imshow(thresh,cmap="binary")
plt.title('thresholding')
print(thresh.shape)
```

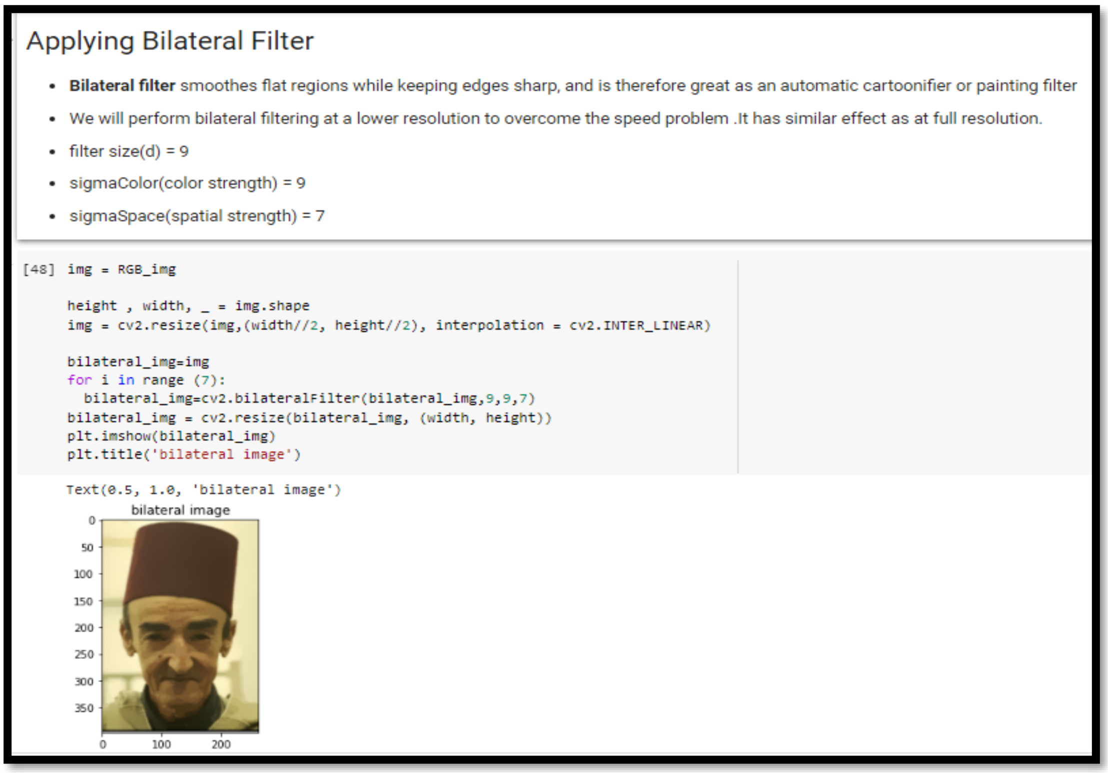
```
(397, 263)
```

Output: More similar to a Black and White hand sketch images

f. Convert sketch from Gray scale to RGB
   Correcting the Black background and white edges using bitwise_not() to reverse colors

2-Generating Color Painting

a. Applying Bilateral Filter
   - First resizing the image to half its original size then applying the filter



Applying Bilateral Filter

- **Bilateral filter** smoothes flat regions while keeping edges sharp, and is therefore great as an automatic cartoonifier or painting filter
- We will perform bilateral filtering at a lower resolution to overcome the speed problem .It has similar effect as at full resolution.
- filter size(d) = 9
- sigmaColor(color strength) = 9
- sigmaSpace(spatial strength) = 7

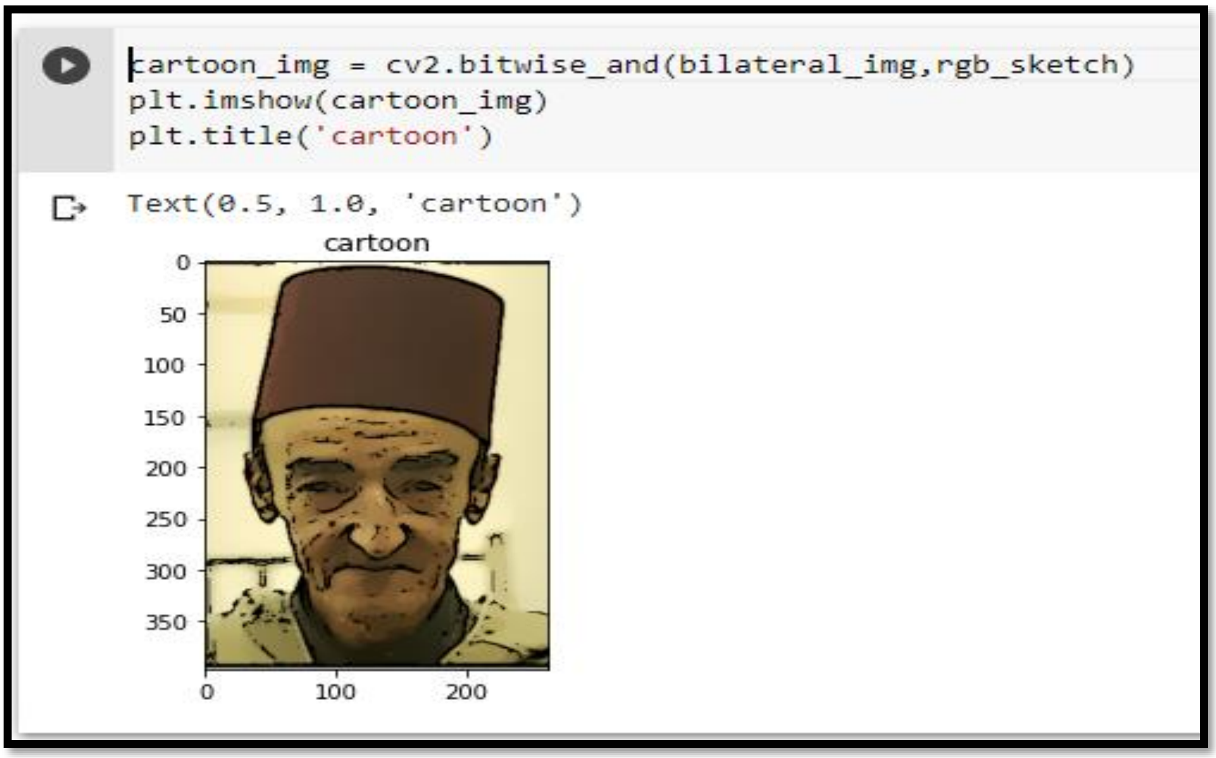
```
[48] img = RGB_img

     height , width, _ = img.shape
     img = cv2.resize(img,(width//2, height//2), interpolation = cv2.INTER_LINEAR)

     bilateral_img=img
     for i in range (7):
       bilateral_img=cv2.bilateralFilter(bilateral_img,9,9,7)
     bilateral_img = cv2.resize(bilateral_img, (width, height))
     plt.imshow(bilateral_img)
     plt.title('bilateral image')
```

```
Text(0.5, 1.0, 'bilateral image')
```

Output: Smoothed image with more color strength

3-Finally creating the cartoon effect

By over laying the smoothed color painting image onto the black and white sketch

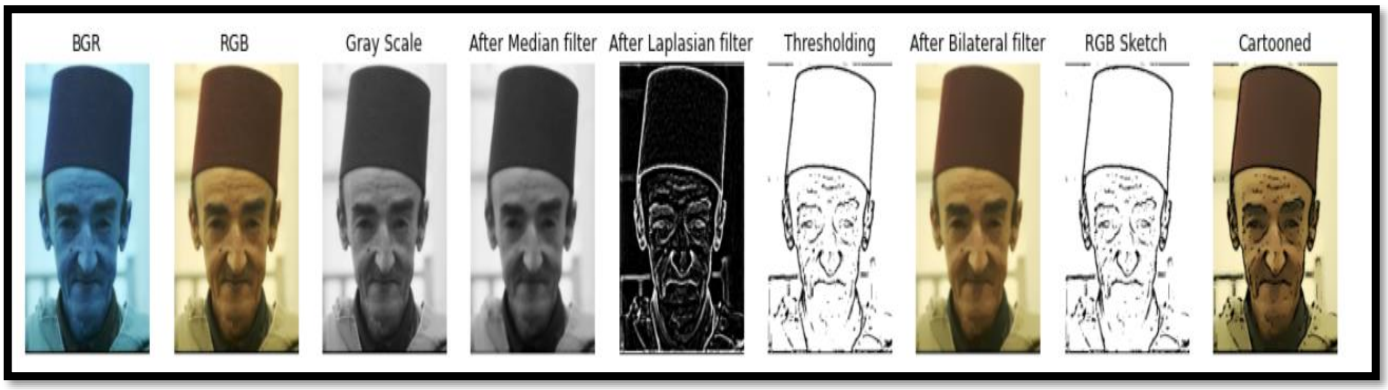


```
cartoon_img = cv2.bitwise_and(bilateral_img,rgb_sketch)
plt.imshow(cartoon_img)
plt.title('cartoon')
```

```
Text(0.5, 1.0, 'cartoon')
```

using bitwise_and()

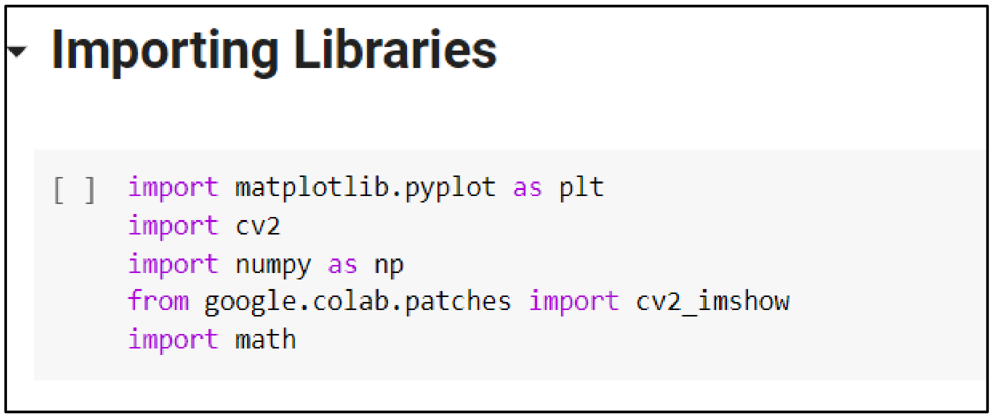
Image Sequence:



Part Two

Road Lane Detection Using Hough Transform

➢ First Importing libraries:

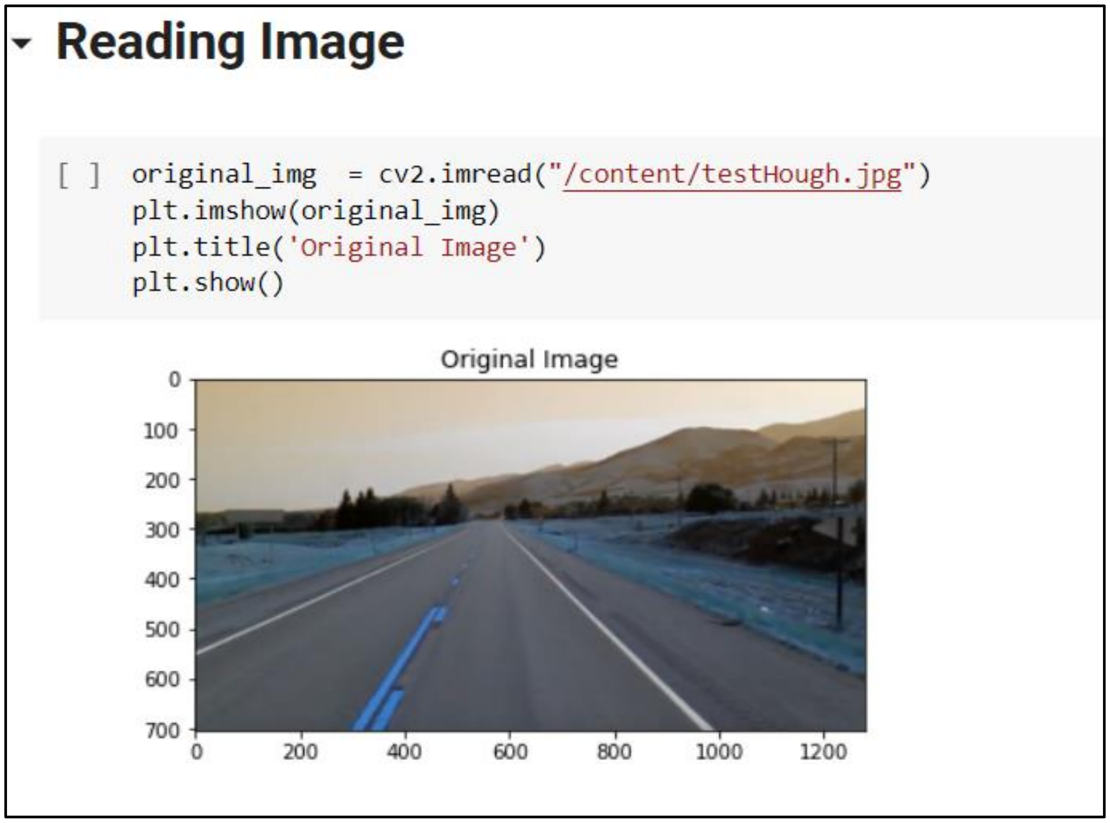


Importing Libraries

```
[ ] import matplotlib.pyplot as plt
    import cv2
    import numpy as np
    from google.colab.patches import cv2_imshow
    import math
```

➢ Reading the image:

Using open cv documentation we were able to read the image using open cv

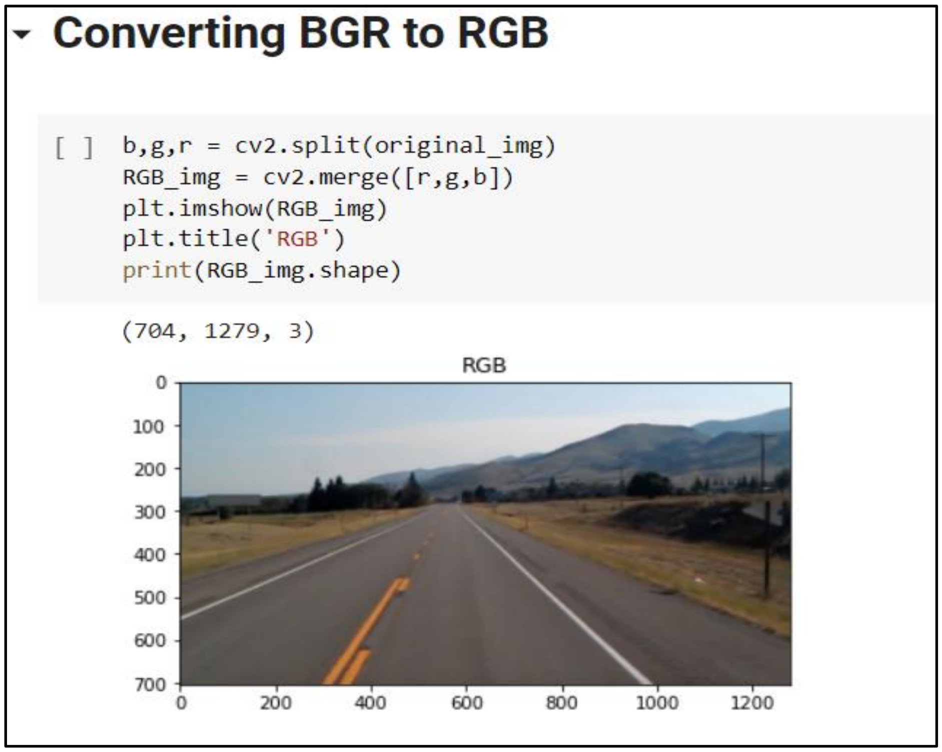


Reading Image

```
[ ] original_img  = cv2.imread("/content/testHough.jpg")
    plt.imshow(original_img)
    plt.title('Original Image')
    plt.show()
```

➢ Converting BGR to RGB

As in open cv the image is read in BGR so we converted it into RGB so we could have the original image.



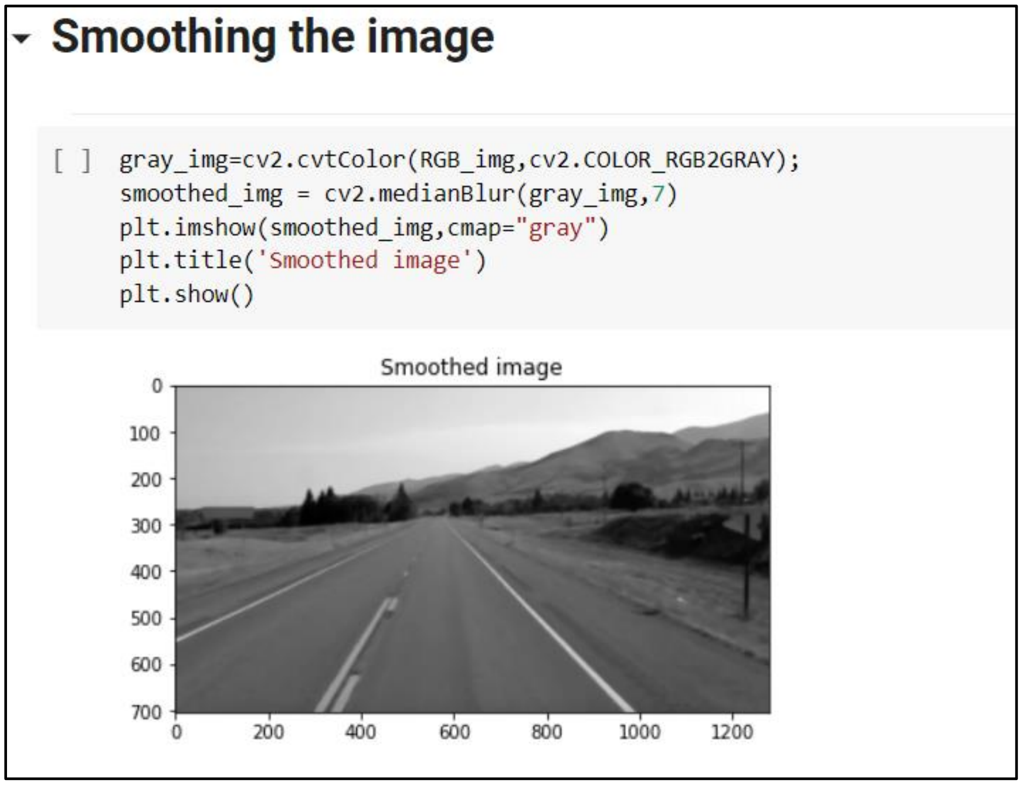
▾ **Converting BGR to RGB**

```
[ ]  b,g,r = cv2.split(original_img)
     RGB_img = cv2.merge([r,g,b])
     plt.imshow(RGB_img)
     plt.title('RGB')
     print(RGB_img.shape)
```

```
(704, 1279, 3)
```

➢ Smoothing the Image:

Smoothing the image using median filter to remove noise and plotting it in gray scale

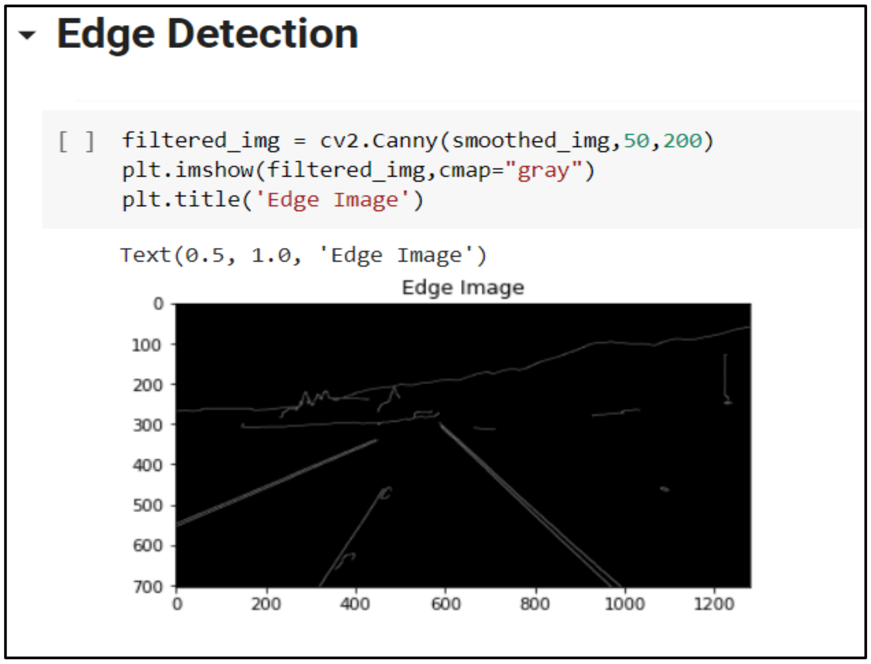


▾ **Smoothing the image**

```
[ ]  gray_img=cv2.cvtColor(RGB_img,cv2.COLOR_RGB2GRAY);
     smoothed_img = cv2.medianBlur(gray_img,7)
     plt.imshow(smoothed_img,cmap="gray")
     plt.title('Smoothed image')
     plt.show()
```

➢ Edge Detection:

Edges are detected using Cannie’s Algorithm



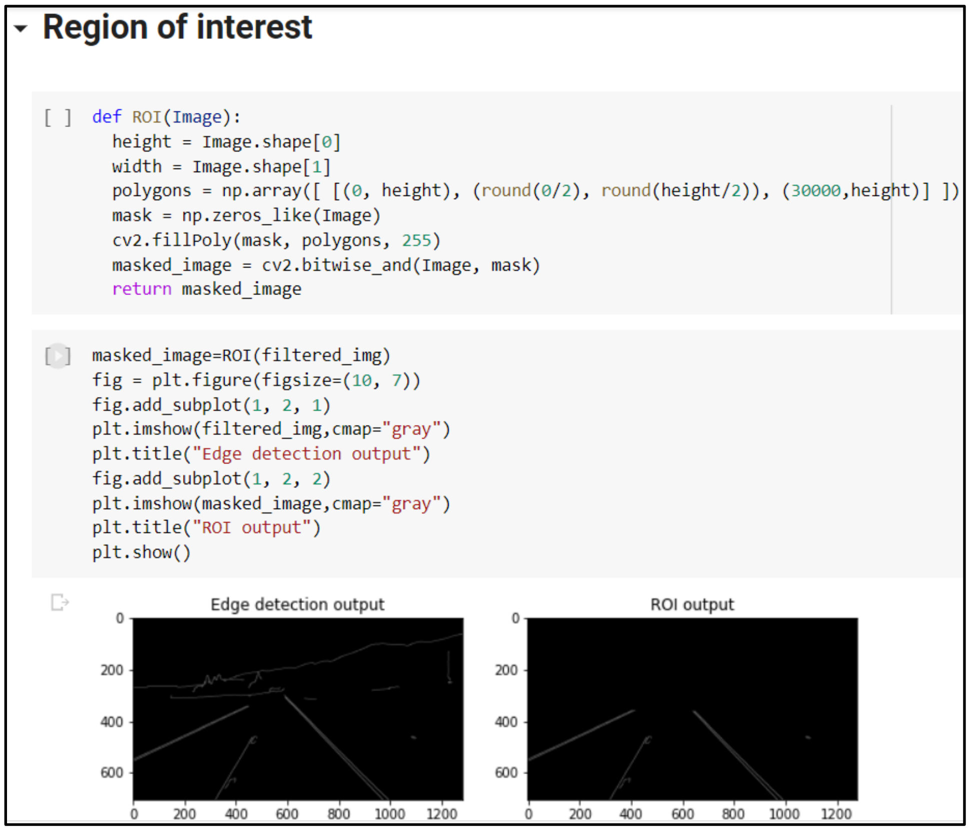
Edge Detection

```
[ ]  filtered_img = cv2.Canny(smoothed_img,50,200)
     plt.imshow(filtered_img,cmap="gray")
     plt.title('Edge Image')
```

```
Text(0.5, 1.0, 'Edge Image')
```

➢ Region Of Interest

Then, we get the region of interest to remove any unwanted lines in the image after detecting edges using ROI function that defines the sides of the polygon which we need in the image.



Region of interest

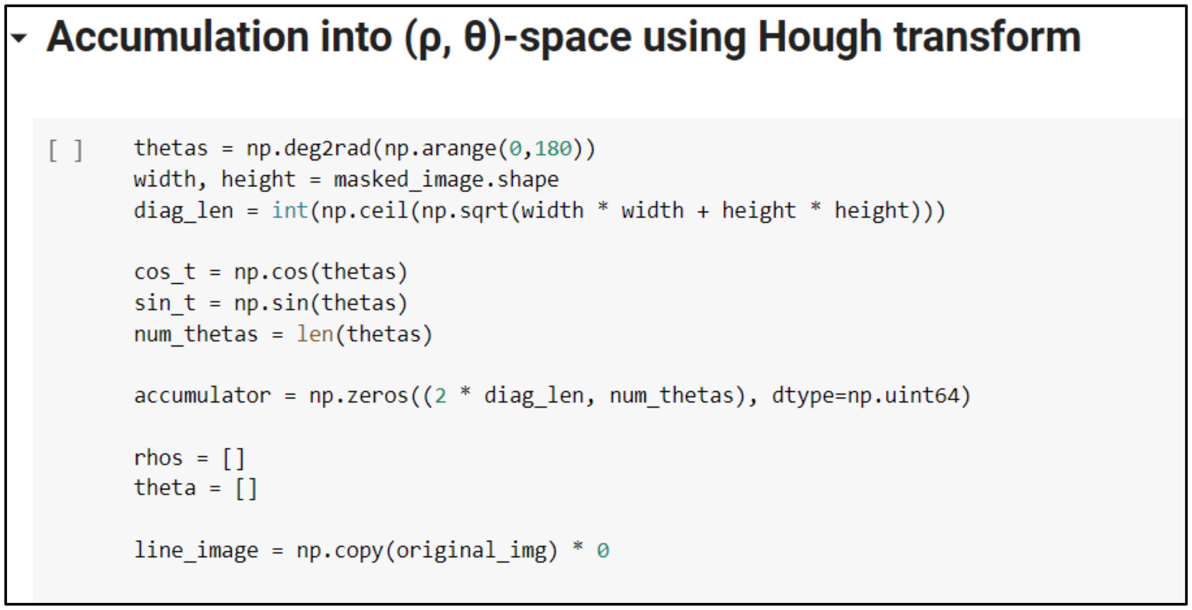
```
[ ]  def ROI(Image):
       height = Image.shape[0]
       width = Image.shape[1]
       polygons = np.array([ [(0, height), (round(0/2), round(height/2)), (30000,height)] ])
       mask = np.zeros_like(Image)
       cv2.fillPoly(mask, polygons, 255)
       masked_image = cv2.bitwise_and(Image, mask)
       return masked_image
```

```
[ ]  masked_image=ROI(filtered_img)
     fig = plt.figure(figsize=(10, 7))
     fig.add_subplot(1, 2, 1)
     plt.imshow(filtered_img,cmap="gray")
     plt.title("Edge detection output")
     fig.add_subplot(1, 2, 2)
     plt.imshow(masked_image,cmap="gray")
     plt.title("ROI output")
     plt.show()
```

➢ Accumulation into (ρ, θ)-space using Hough transform

Initializing variables that are needed in Accumulator (), Thresholding () and Drawing lines () functions where:

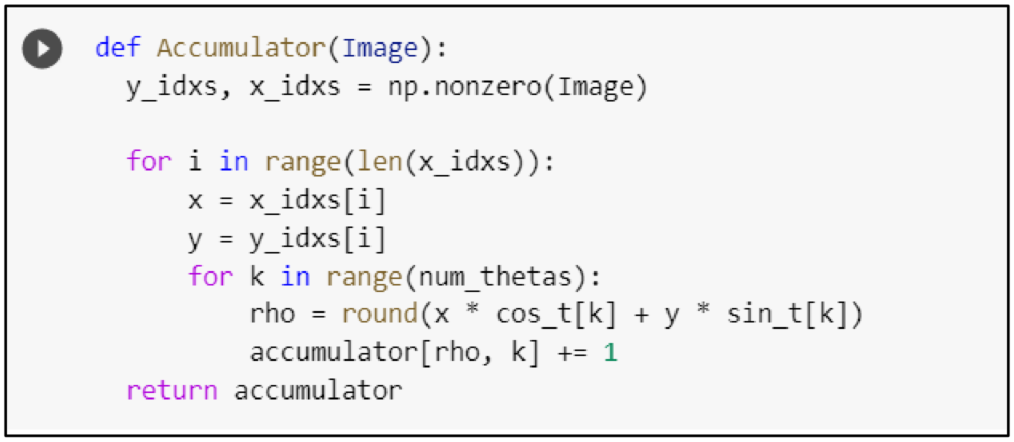
- Thetas→Values of thetas from 0 to 180 degrees
- diag_len→To adjust the size of the accumulator according to the size of the image
- rhos []→ Will be used after thresholding which contains the rhos of the detected lines
- theta []→ Will be used after thresholding which contains the rhos of the detected lines



Accumulation into (ρ, θ)-space using Hough transform

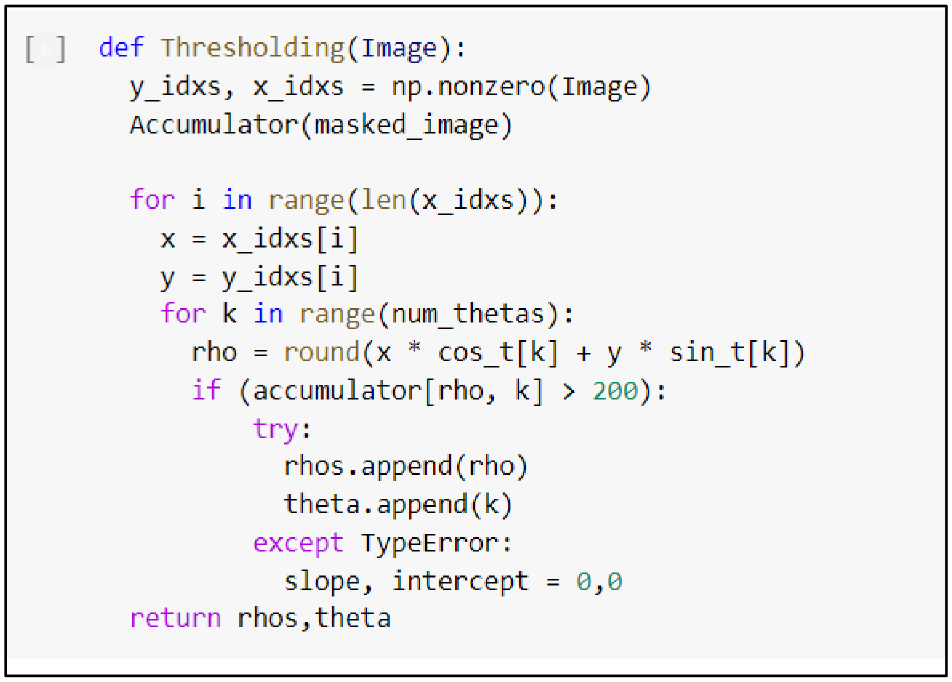
```
[ ]   thetas = np.deg2rad(np.arange(0,180))
      width, height = masked_image.shape
      diag_len = int(np.ceil(np.sqrt(width * width + height * height)))

      cos_t = np.cos(thetas)
      sin_t = np.sin(thetas)
      num_thetas = len(thetas)

      accumulator = np.zeros((2 * diag_len, num_thetas), dtype=np.uint64)

      rhos = []
      theta = []

      line_image = np.copy(original_img) * 0
```

Accumulator function: that takes any image as a parameter (the masked image will be passed to it) it is used to get the rhos and thetas of each point in the given image



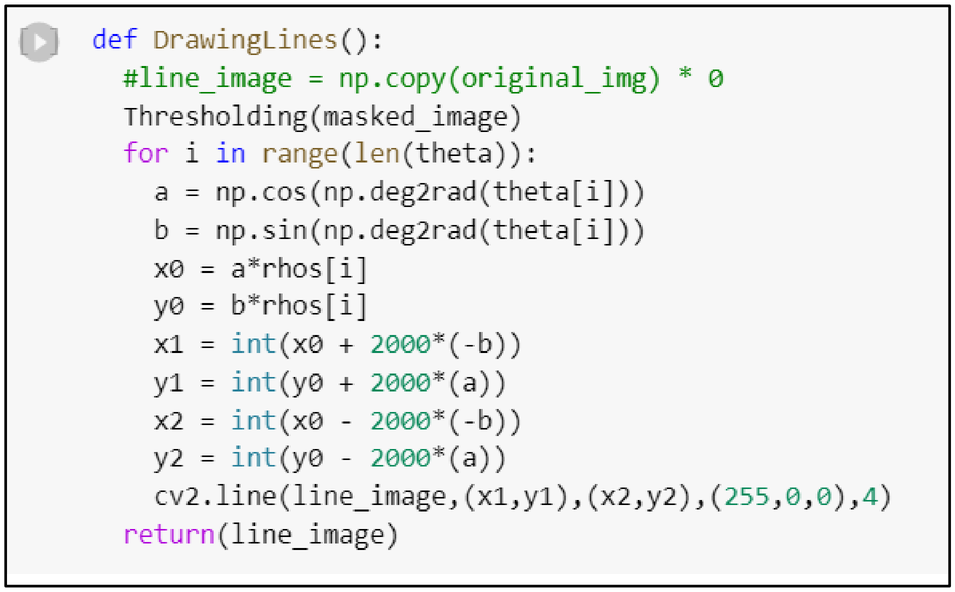
```
def Accumulator(Image):
  y_idxs, x_idxs = np.nonzero(Image)

  for i in range(len(x_idxs)):
      x = x_idxs[i]
      y = y_idxs[i]
      for k in range(num_thetas):
          rho = round(x * cos_t[k] + y * sin_t[k])
          accumulator[rho, k] += 1
  return accumulator
```

Thresholding function: is used to get the lines that have the maximum intersections so that most likely that they are forming lines



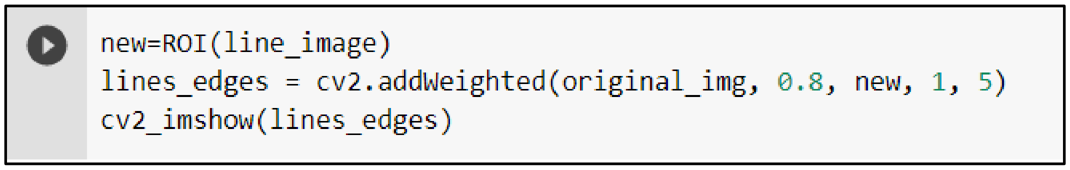
```python
def Thresholding(Image):
  y_idxs, x_idxs = np.nonzero(Image)
  Accumulator(masked_image)

  for i in range(len(x_idxs)):
    x = x_idxs[i]
    y = y_idxs[i]
    for k in range(num_thetas):
      rho = round(x * cos_t[k] + y * sin_t[k])
      if (accumulator[rho, k] > 200):
          try:
            rhos.append(rho)
            theta.append(k)
          except TypeError:
            slope, intercept = 0,0
  return rhos,theta
```

DrawingLines function: Is used to draw the lines of the detected lines according to the rhos and thetas that are returned by Thresholding function



```python
def DrawingLines():
  #line_image = np.copy(original_img) * 0
  Thresholding(masked_image)
  for i in range(len(theta)):
    a = np.cos(np.deg2rad(theta[i]))
    b = np.sin(np.deg2rad(theta[i]))
    x0 = a*rhos[i]
    y0 = b*rhos[i]
    x1 = int(x0 + 2000*(-b))
    y1 = int(y0 + 2000*(a))
    x2 = int(x0 - 2000*(-b))
    y2 = int(y0 - 2000*(a))
    cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),4)
  return(line_image)
```

Finally, we need to apply the same region of interest that we used before on the (line_image) that is returned from DrawingLines function and (cv2.addWeighted) is used to apply the drawn lines on the original image



```
new=ROI(line_image)
lines_edges = cv2.addWeighted(original_img, 0.8, new, 1, 5)
cv2_imshow(lines_edges)
```