Computer Vision Assignment (1)

Team Members

Nouran Hisham	**ID: 6532**
Nourhan Waleed	**ID: 6609**
Kareem Sabra	**ID: 6594**

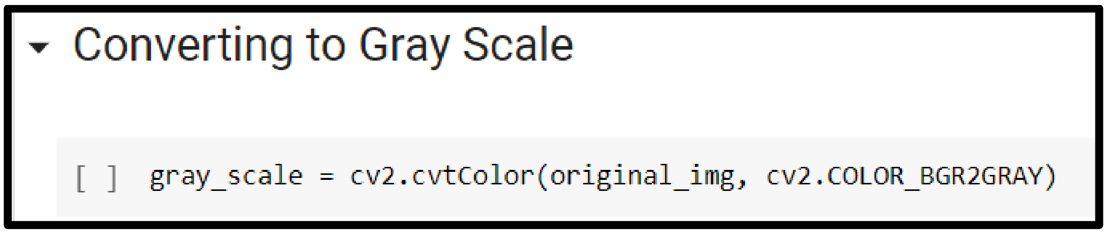
Google colab notebook link:
https://colab.research.google.com/drive/1Dc4UE0MM-ZTtAn8WJ_loPgrQ01ep8Wp4?usp=sharing

Part (1)

Applying Image Processing Filters for Image Cartoonifying

The basic idea is to fill the flat parts with some colour and then draw thick lines on the strong edges. In other words, the flat areas should become much flatter and the edges should become much more distinct. We will detect edges and smooth the flat areas, then draw enhanced edges back on top to produce a cartoon or comic book effect.

1.1 Generating a black-and-white sketch

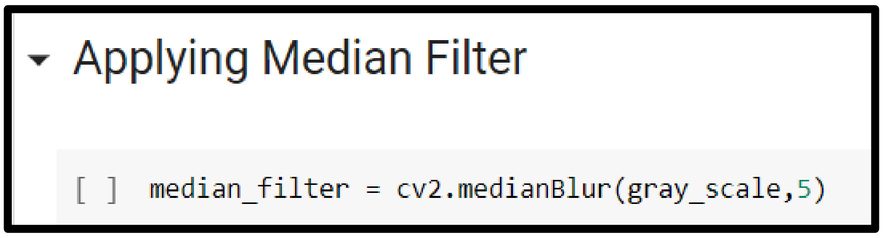

Converting to Gray Scale

```
[ ]  gray_scale = cv2.cvtColor(original_img, cv2.COLOR_BGR2GRAY)
```

Output:

1.1.1 Noise Reduction Using Median Filter

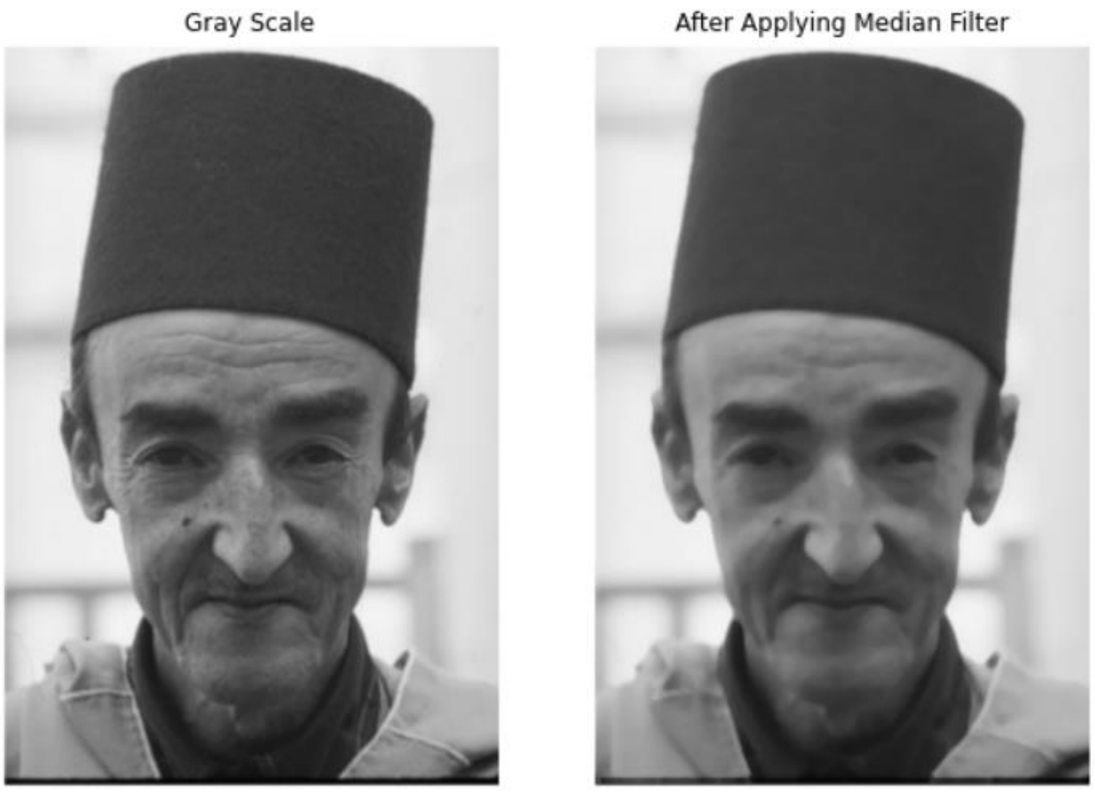
The **median filter** is the filtering technique used for noise removal from images and signals. Median filter is very crucial in the image processing field as it is well known for the preservation of edges during noise removal.



Applying Median Filter

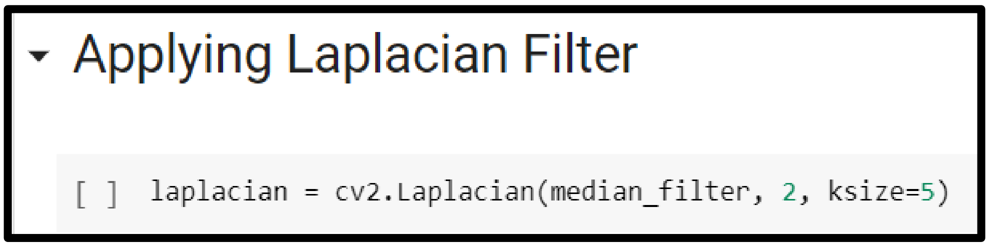
```
[ ]  median_filter = cv2.medianBlur(gray_scale,5)
```

Output:



1.1.2 Edge Detection Using Laplacian Filter

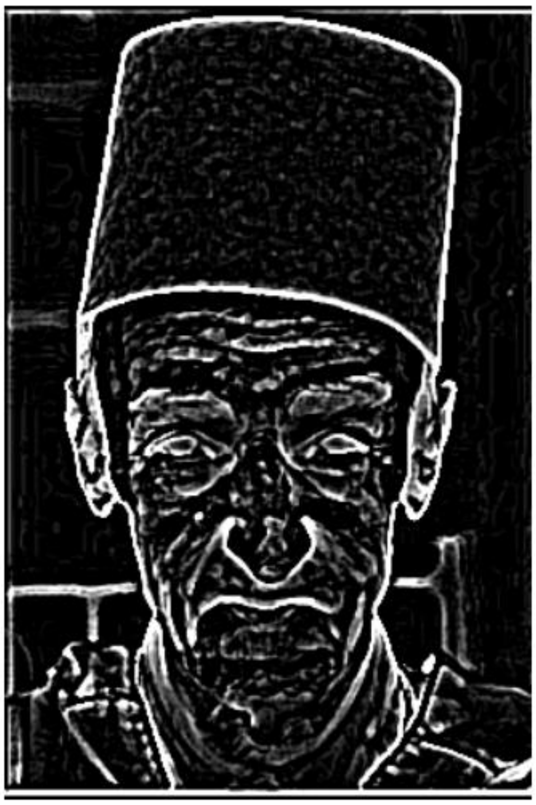
A **Laplacian filter** is an edge detector used to compute the second derivatives of an image, measuring the rate at which the first derivatives change. This determines if a change in adjacent pixel values is from an edge or continuous progression.



Applying Laplacian Filter

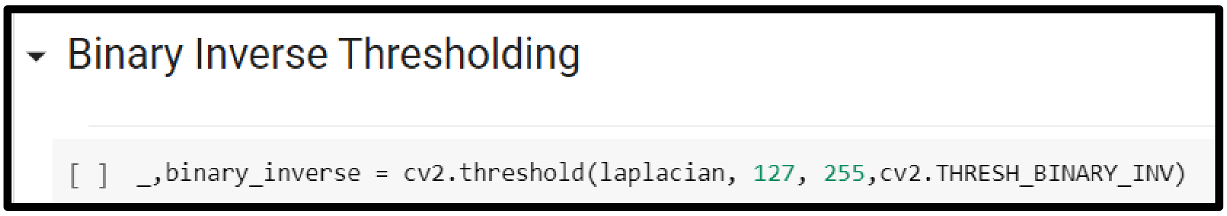
```
[ ]  laplacian = cv2.Laplacian(median_filter, 2, ksize=5)
```

Output:



1.2 Generating a colour painting and a cartoon

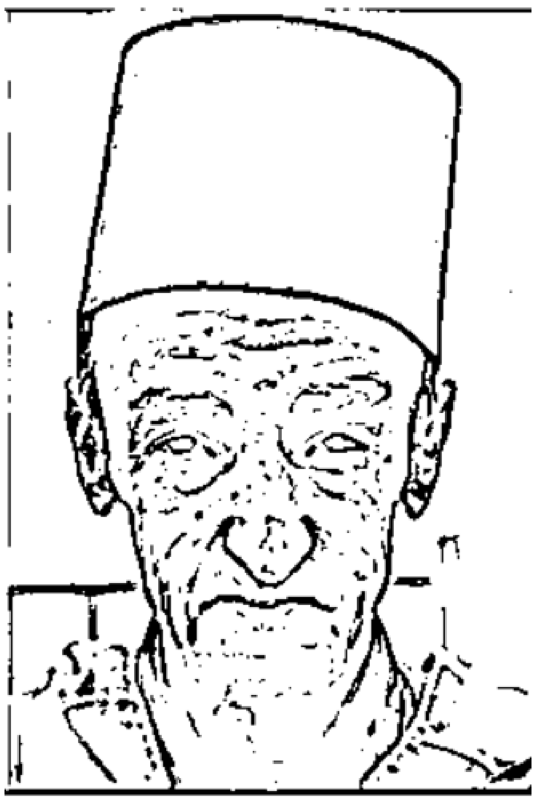
In **inverse binary thresholding**, if the value of the pixel is less than the threshold, it will be given a maximum value (white). If it is greater than the threshold, it will be assigned 0, (black).



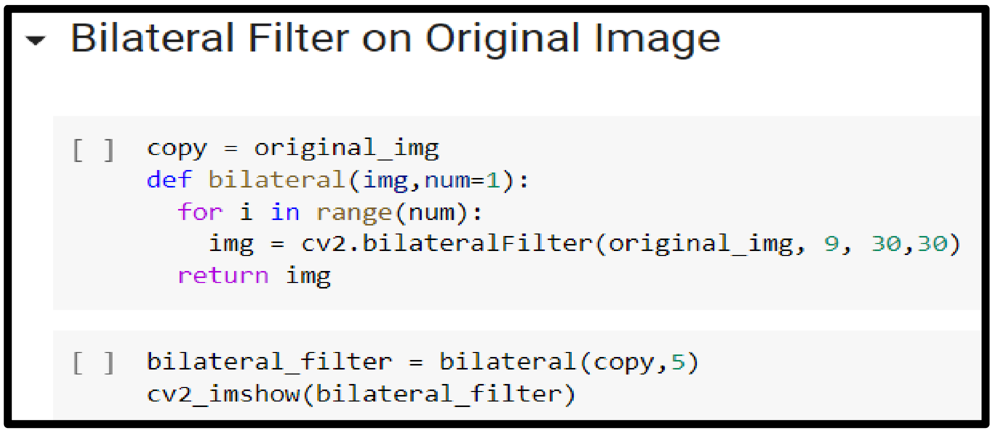
Binary Inverse Thresholding

```
[ ]  _,binary_inverse = cv2.threshold(laplacian, 127, 255,cv2.THRESH_BINARY_INV)
```

Output:



A **bilateral filter** is a non-linear, edge-preserving, and noise-reducing smoothing filter for images. It replaces the intensity of each pixel with a weighted average of intensity values from nearby pixels. This weight can be based on a Gaussian distribution.

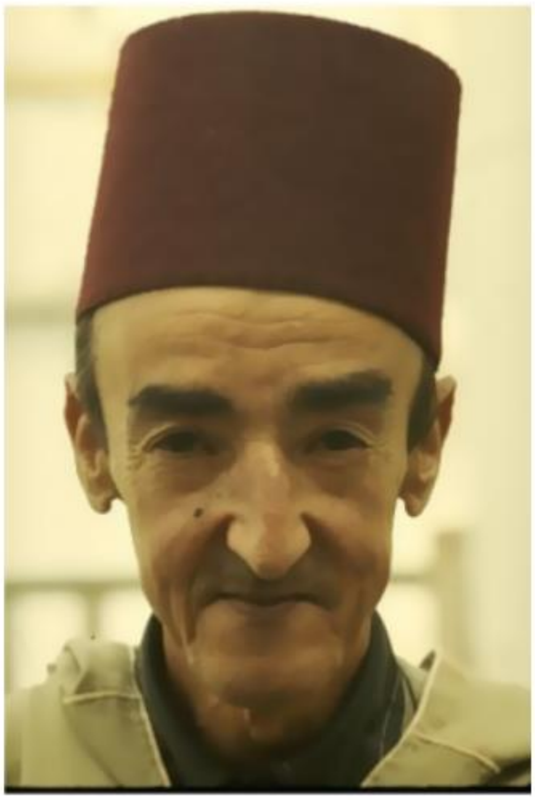


Bilateral Filter on Original Image

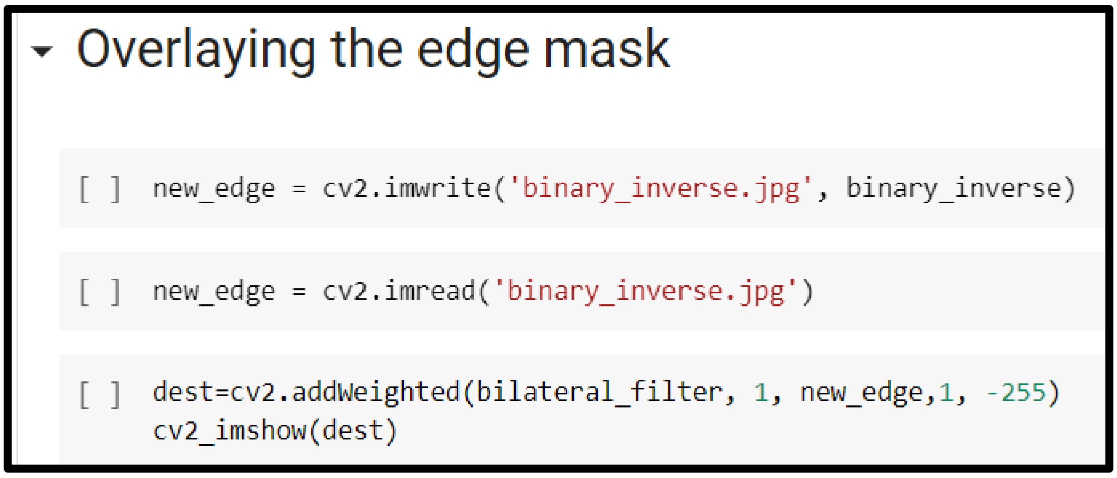
```
[ ]  copy = original_img
     def bilateral(img,num=1):
       for i in range(num):
         img = cv2.bilateralFilter(original_img, 9, 30,30)
       return img
```

```
[ ]  bilateral_filter = bilateral(copy,5)
     cv2_imshow(bilateral_filter)
```

Output:



To overlay the edge mask ”sketch” onto the bilateral filter ”painting”, we can start with a black background and copy the ”painting” pixels that aren’t edges in the ”sketch” mask.

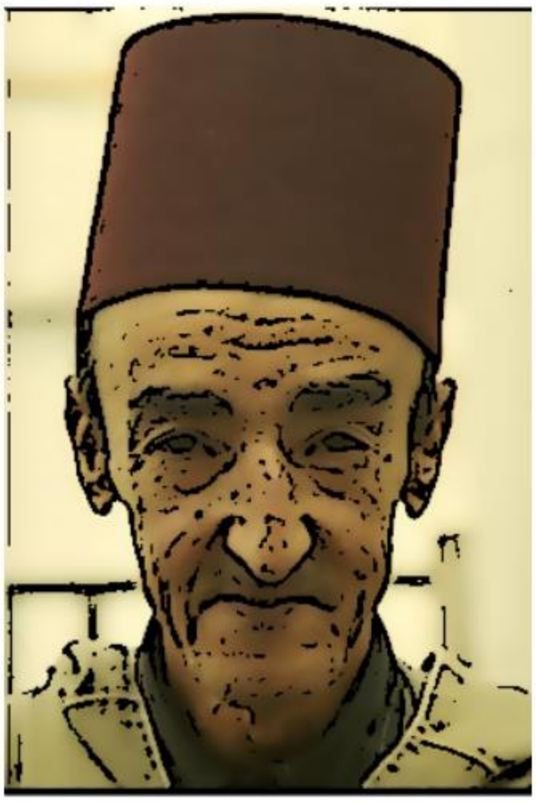

Overlaying the edge mask

```
[ ]  new_edge = cv2.imwrite('binary_inverse.jpg', binary_inverse)
```

```
[ ]  new_edge = cv2.imread('binary_inverse.jpg')
```

```
[ ]  dest=cv2.addWeighted(bilateral_filter, 1, new_edge,1, -255)
     cv2_imshow(dest)
```

Final output:



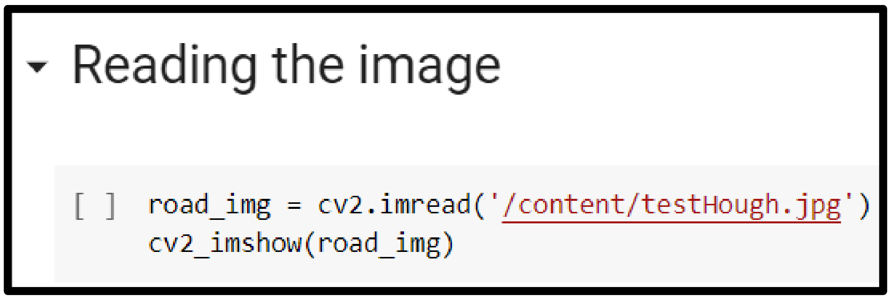
Part (2)

Road Lane Detection Using Hough Transform

The objective of this part of the assignment is the detection of road lanes in an image using Hough Transform.

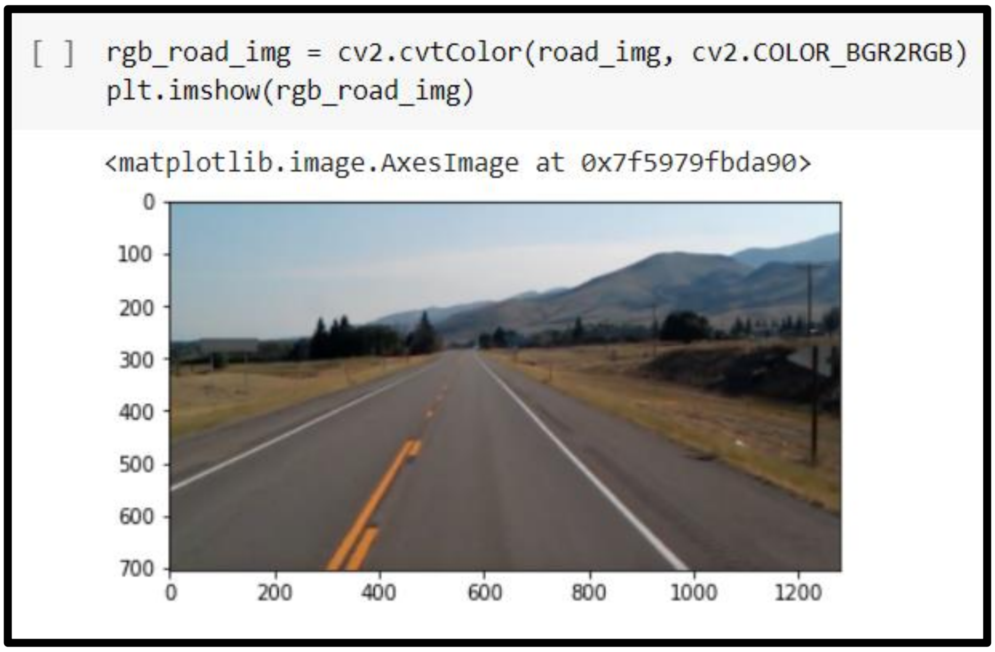
2.1 Hough Transform

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: **$x \cos \theta + y \sin \theta = \rho$** 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.



Reading the image

```
[ ]  road_img = cv2.imread('/content/testHough.jpg')
     cv2_imshow(road_img)
```

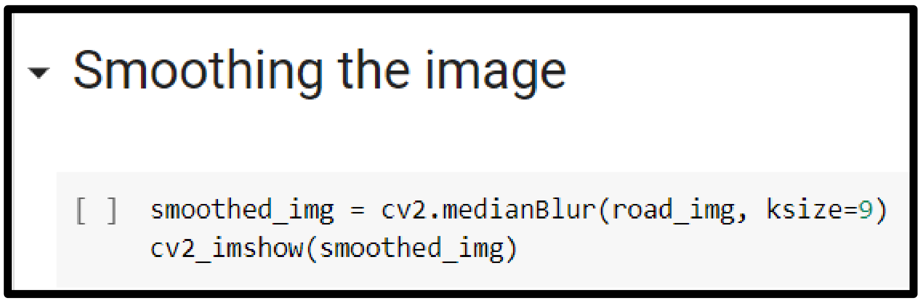


```
[ ]  rgb_road_img = cv2.cvtColor(road_img, cv2.COLOR_BGR2RGB)
     plt.imshow(rgb_road_img)

<matplotlib.image.AxesImage at 0x7f5979fbda90>
```

2.2 Implementation Details

2.2.1 Smoothing the image

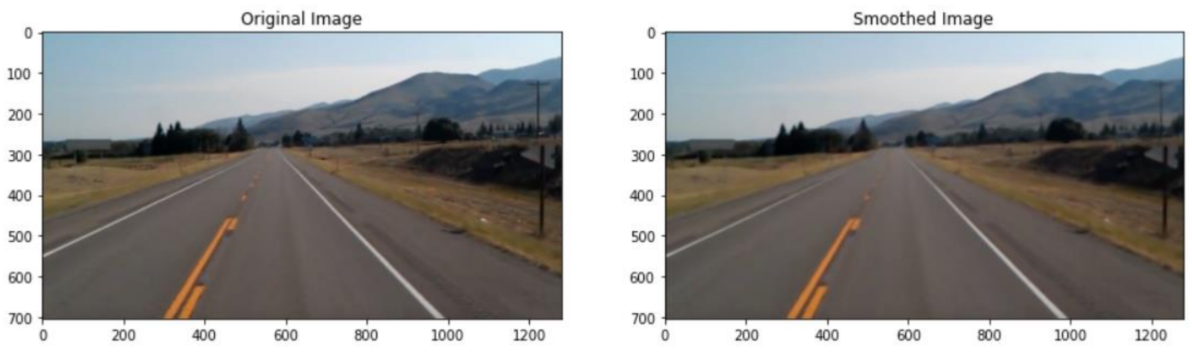
It is important to identify as many edges in the image as possible. But we must also filter any image noise which can create false edges and ultimately affect edge detection. This reduction of noise and image smoothening will be done by a filter called **Median Blur**.



Smoothing the image

```
[ ]  smoothed_img = cv2.medianBlur(road_img, ksize=9)
     cv2_imshow(smoothed_img)
```

Output:

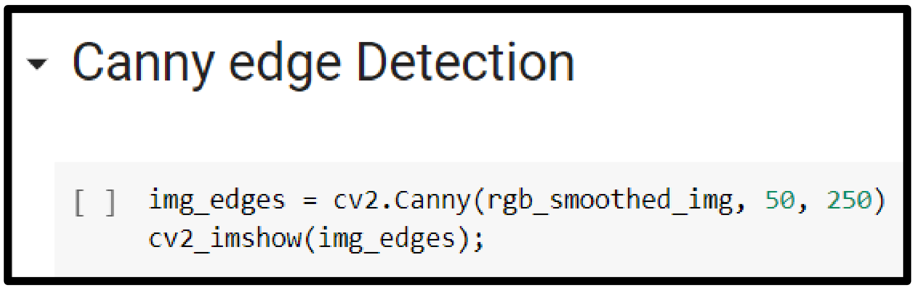


2.2.2 Edge Detection

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.

The **canny edge detection** first removes noise from image by smoothening. It then finds the image gradient to highlight regions with high spatial derivatives. The algorithm then tracks along these regions and suppresses any pixel that is not at the maximum (non-maximum suppression).

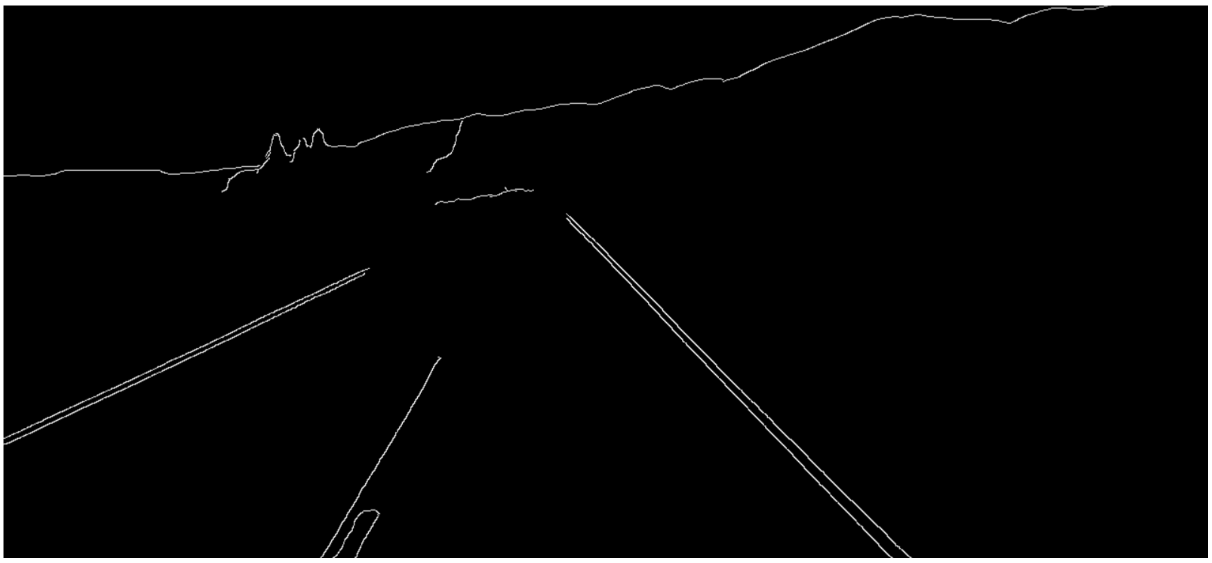
The two arguments **low_threshold** and **high_threshold** allows us to isolate the adjacent pixels that follow the strongest gradience. If the gradient is larger than the upper threshold, then it is identified as an edge pixel. If it's below the lower threshold, it gets rejected. The gradient between the thresholds is accepted only when it is connected to a strong edge.

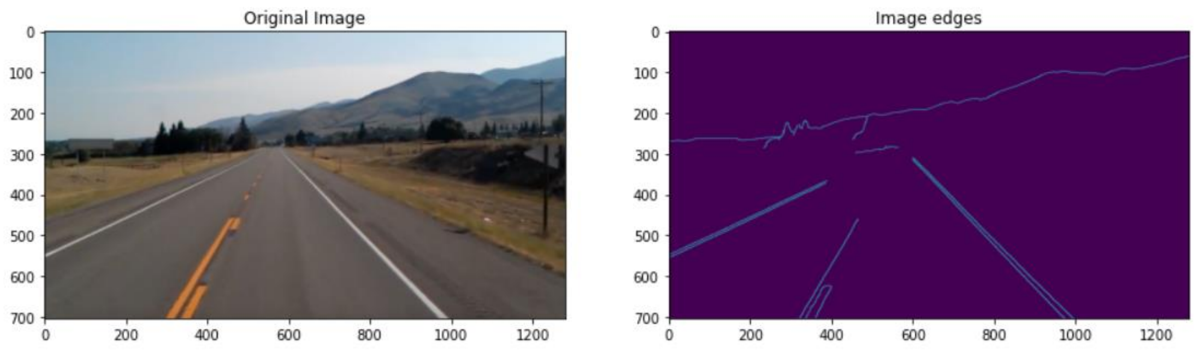


Canny edge Detection

```
[ ]  img_edges = cv2.Canny(rgb_smoothed_img, 50, 250)
     cv2_imshow(img_edges);
```

Output:



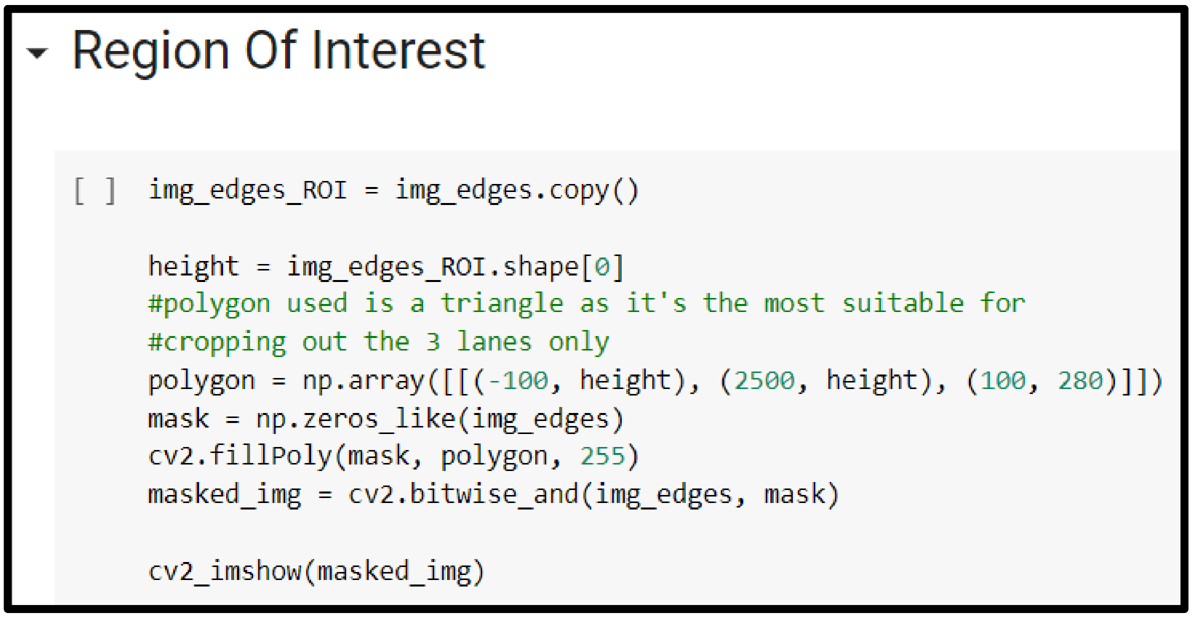


2.3 Region of Interest

The output of the edge detection will contain unnecessary edges that belongs to objects outside the road. Therefore, to eliminate this noise, it was found that our area of interest somehow makes the shape of a triangle, so we defined a polygon of 3 sides to eliminate the regions in the image that doesn't contain the road lanes we're trying to detect.

We used **fillPoly()** function to draw our polygon (triangle) over the image.

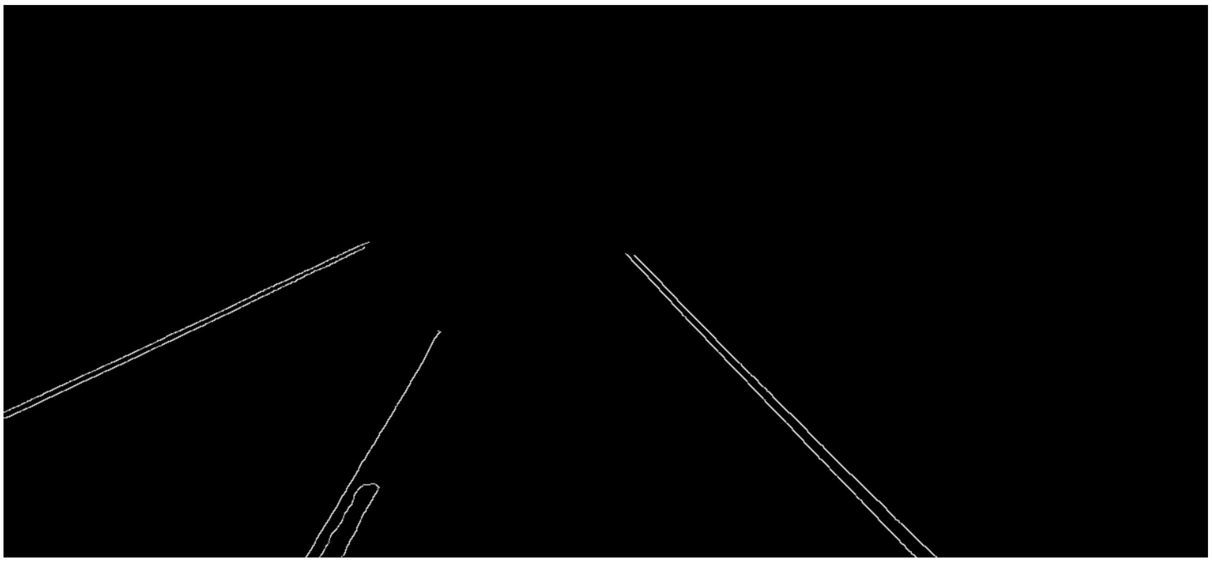
To apply the masking, we used the **bitwise_and()** function which creates a black matrix where only the parts of this black image that match up with white pixels in the mask are filled with the pixels from the region of interest. This means that in the mask where there are white pixels. the region of interest value will be copied in the pure black image generated by the function in the corresponding coordinate.

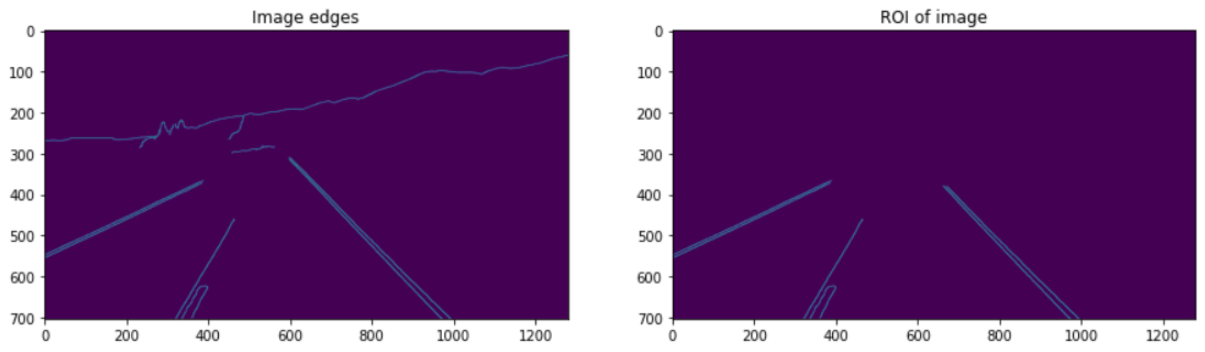


Region Of Interest

```
[ ]  img_edges_ROI = img_edges.copy()

     height = img_edges_ROI.shape[0]
     #polygon used is a triangle as it's the most suitable for
     #cropping out the 3 lanes only
     polygon = np.array([[(-100, height), (2500, height), (100, 280)]])
     mask = np.zeros_like(img_edges)
     cv2.fillPoly(mask, polygon, 255)
     masked_img = cv2.bitwise_and(img_edges, mask)

     cv2_imshow(masked_img)
```

Output:



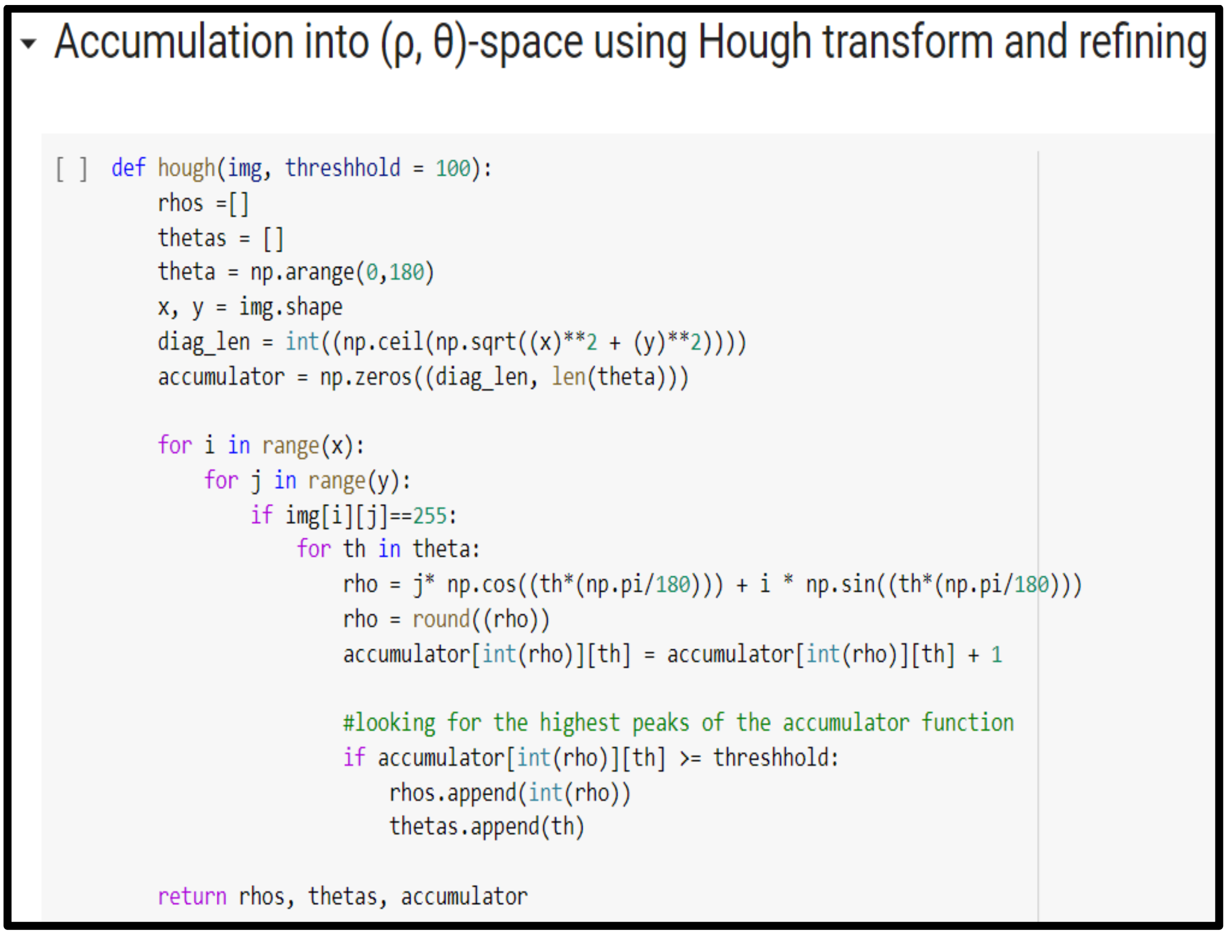


2.3.1 Accumulation into (ρ, θ)-space using Hough transform

In this part, we followed the pseudo code provided in the assignment pdf, we defined **thetas** in the **range (0, 180)**, **diag_len** to adjust the accumulator's size according to the image size and looped around each edge point in the image. We calculated the rho by the following equation **(rho = x*cos(theta) + y*sin(theta))** and increased the accumulator array accordingly.

Then after filling the accumulator by looping around the whole image, we looped around every cell in the accumulator array again, but this time to eliminate the values which were less than a certain **threshold**, 100 in our case.

Then we'd take only the theta and rho values in of the accumulator cells that exceeded or are equal to that threshold to be able to know the lines that had the greatest number of votes because these would be our lanes.



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

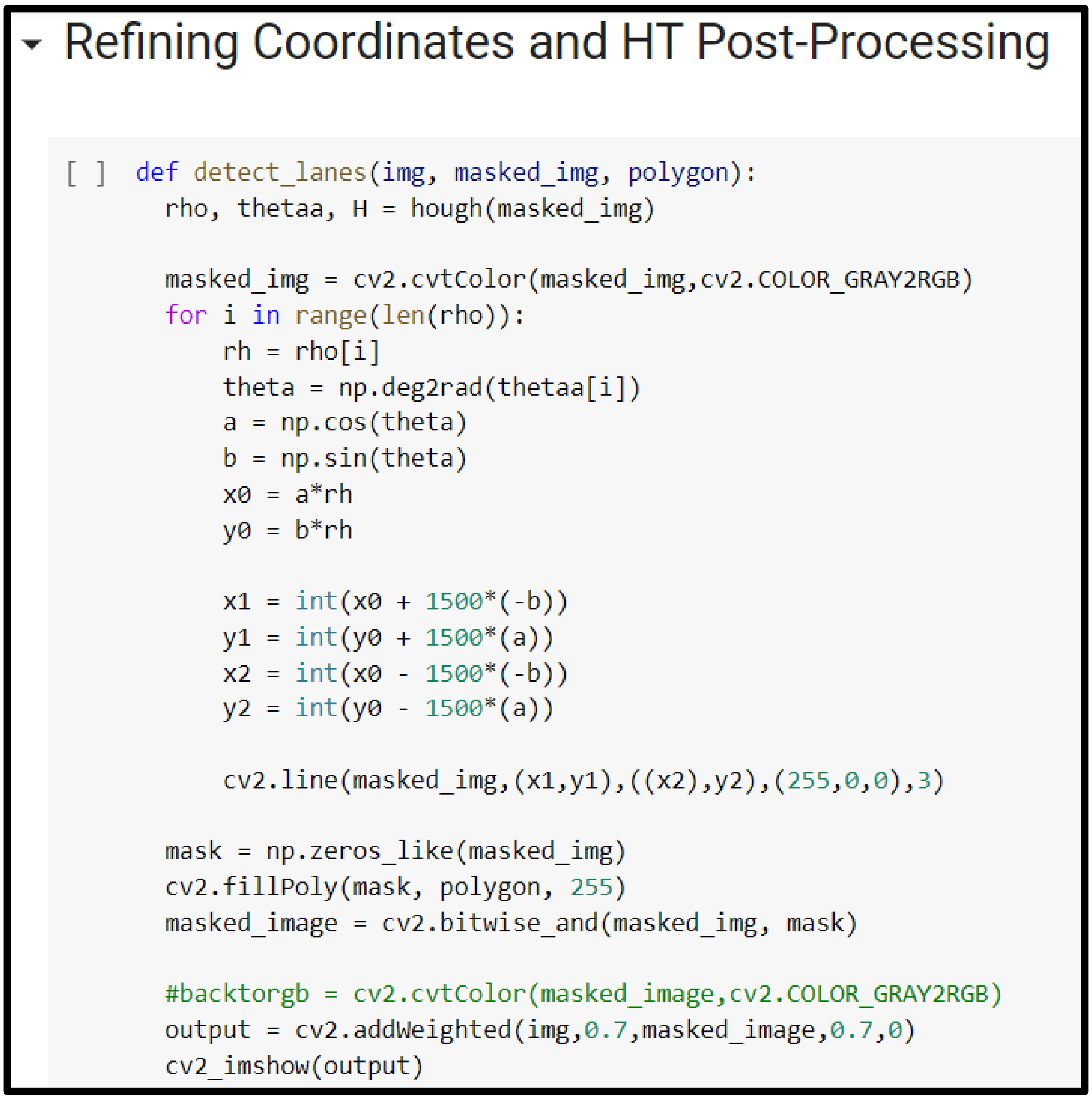
```
[ ]  def hough(img, threshhold = 100):
         rhos =[]
         thetas = []
         theta = np.arange(0,180)
         x, y = img.shape
         diag_len = int((np.ceil(np.sqrt((x)**2 + (y)**2))))
         accumulator = np.zeros((diag_len, len(theta)))

         for i in range(x):
             for j in range(y):
                 if img[i][j]==255:
                     for th in theta:
                         rho = j* np.cos((th*(np.pi/180))) + i * np.sin((th*(np.pi/180)))
                         rho = round((rho))
                         accumulator[int(rho)][th] = accumulator[int(rho)][th] + 1

                         #looking for the highest peaks of the accumulator function
                         if accumulator[int(rho)][th] >= threshhold:
                             rhos.append(int(rho))
                             thetas.append(th)

         return rhos, thetas, accumulator
```

2.3.2 Refining Coordinates and HT Post-Processing

In the following function, we use **the maximum rhos and thetas** from the previous equation to actually draw the lines with the aid of the **line() function in openCV**.

We provide the **line() function** with our image, the start and end points where we want the lines be drawn, their colour and their thickness. We add 1500 or 1000 to the points to be able to extend them in the image and control their length.

We then crop out the extra extended lines by masking the image again with the polygon (triangle) we previously defined to adjust them to be only on the wanted lanes not beyond them.

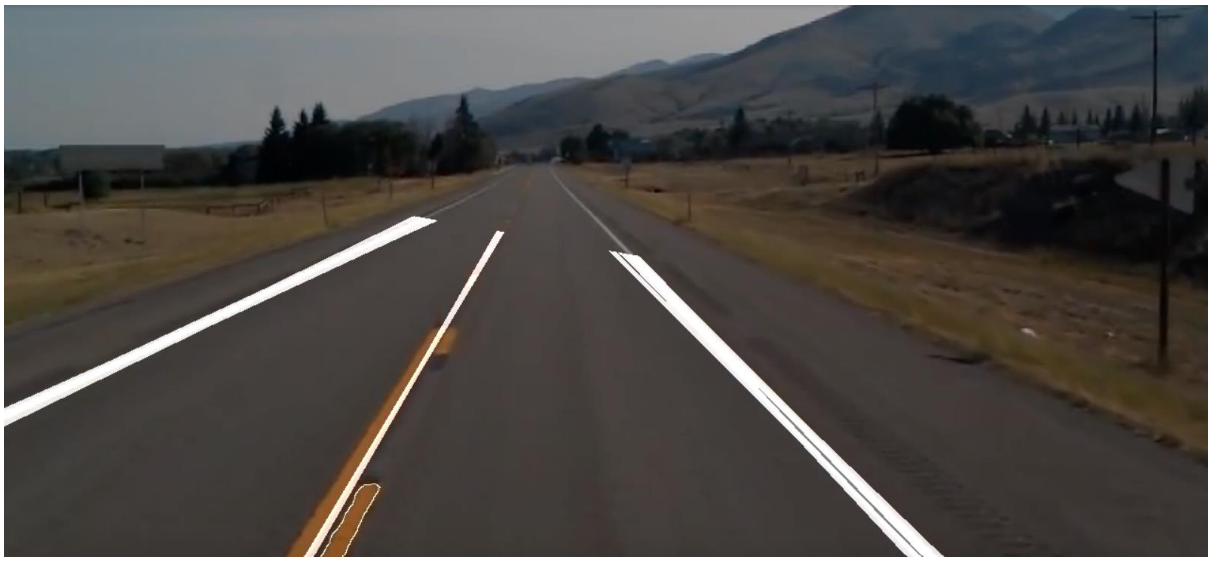


Refining Coordinates and HT Post-Processing

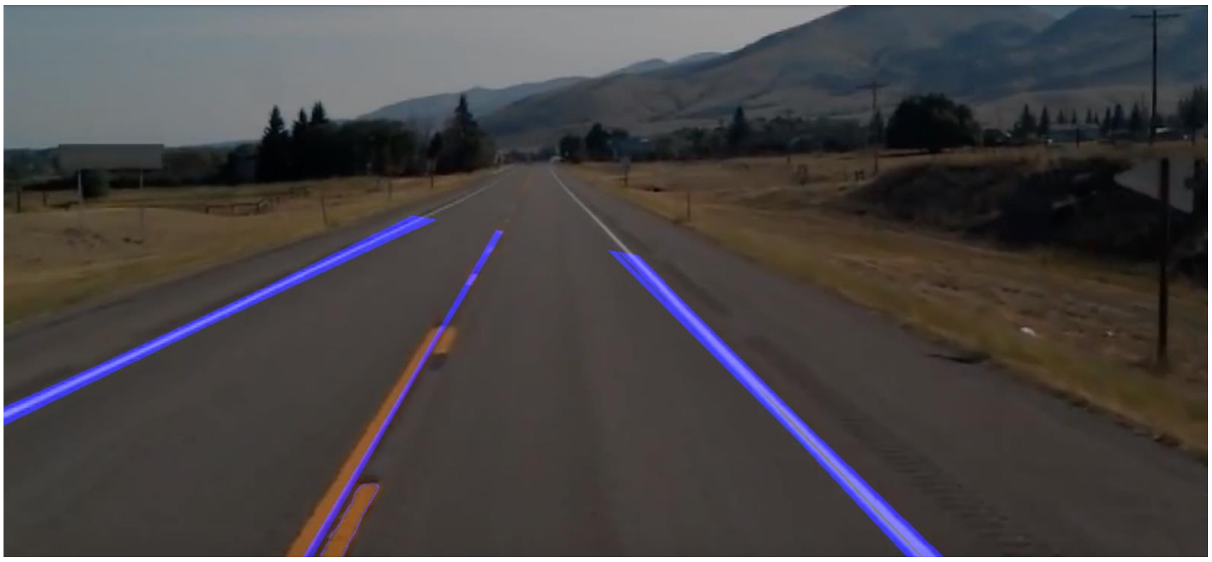
```
[ ]  def detect_lanes(img, masked_img, polygon):
       rho, thetaa, H = hough(masked_img)

       masked_img = cv2.cvtColor(masked_img,cv2.COLOR_GRAY2RGB)
       for i in range(len(rho)):
           rh = rho[i]
           theta = np.deg2rad(thetaa[i])
           a = np.cos(theta)
           b = np.sin(theta)
           x0 = a*rh
           y0 = b*rh

           x1 = int(x0 + 1500*(-b))
           y1 = int(y0 + 1500*(a))
           x2 = int(x0 - 1500*(-b))
           y2 = int(y0 - 1500*(a))

           cv2.line(masked_img,(x1,y1),((x2),y2),(255,0,0),3)

       mask = np.zeros_like(masked_img)
       cv2.fillPoly(mask, polygon, 255)
       masked_image = cv2.bitwise_and(masked_img, mask)

       #backtorgb = cv2.cvtColor(masked_image,cv2.COLOR_GRAY2RGB)
       output = cv2.addWeighted(img,0.7,masked_image,0.7,0)
       cv2_imshow(output)
```

Final output:

We got this output when we drew the lines on the **gray scale** masked image so the **line() function** wasn't able to draw in colours other than white.



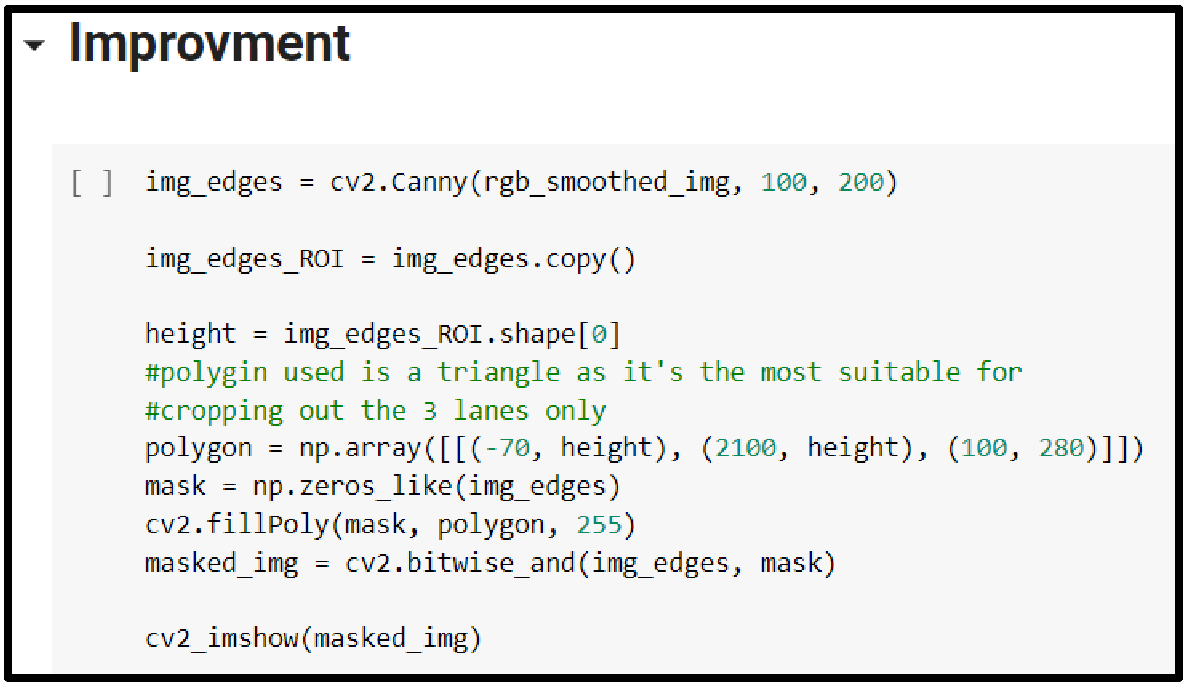
We then modified the code so that we convert the masked image from **gray scale to BGR** to be able to draw the lines in colours.



Improvements:

For the improvements, we adjusted the parameters of the canny edge detection a little bit along with the measurements of our defined polygon.

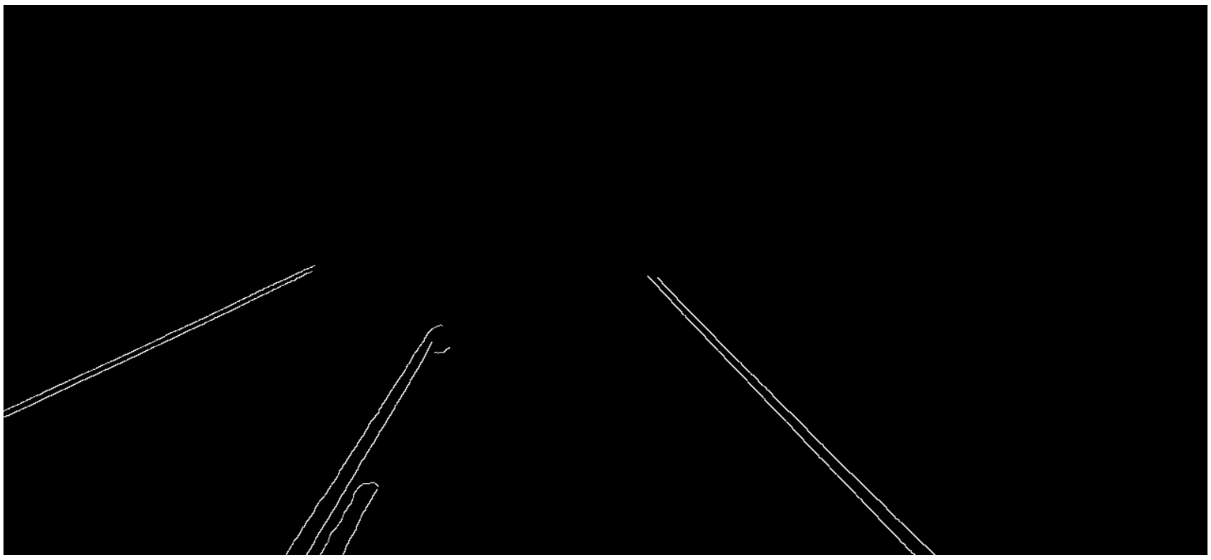
We also, made the accumulator cells that are less than our defined threshold be zero, so we ended up with detecting the edges of the lanes more precisely.

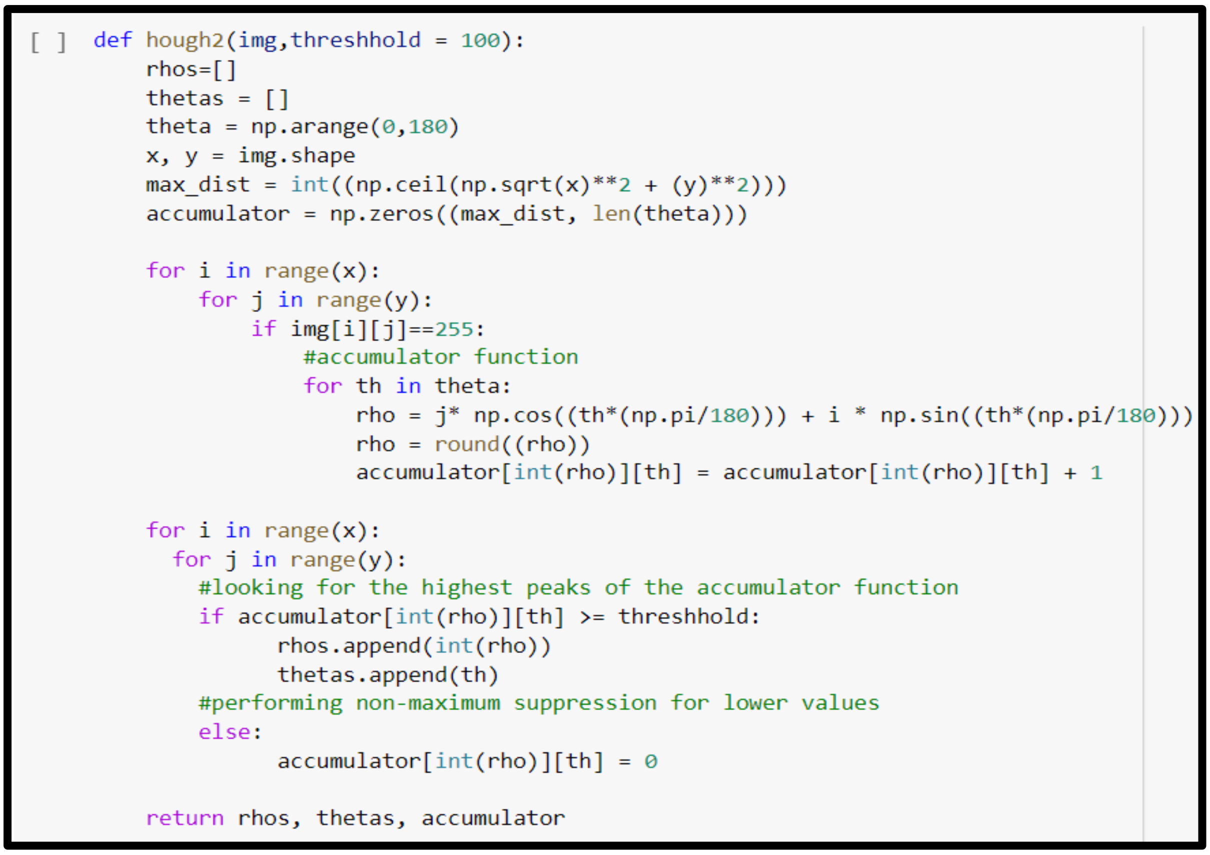


Improvment

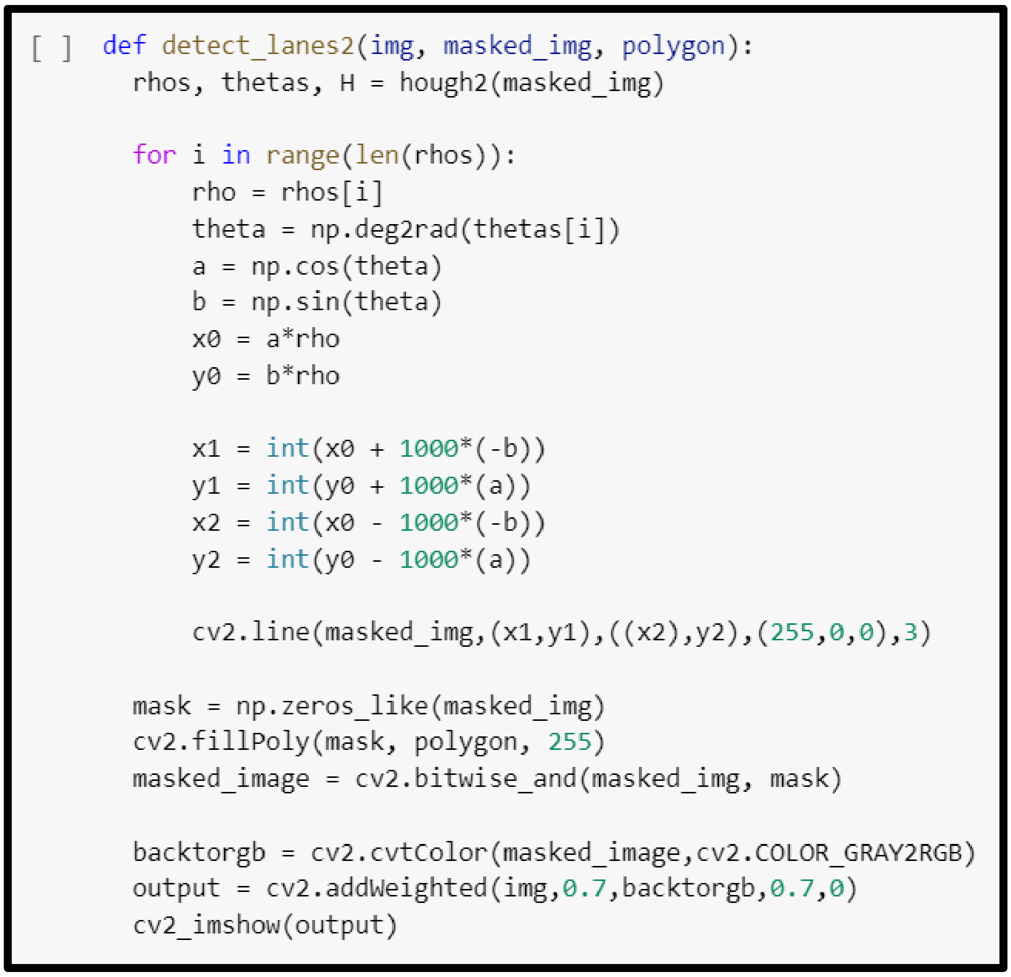
```
[ ]  img_edges = cv2.Canny(rgb_smoothed_img, 100, 200)

     img_edges_ROI = img_edges.copy()

     height = img_edges_ROI.shape[0]
     #polygin used is a triangle as it's the most suitable for
     #cropping out the 3 lanes only
     polygon = np.array([[(-70, height), (2100, height), (100, 280)]])
     mask = np.zeros_like(img_edges)
     cv2.fillPoly(mask, polygon, 255)
     masked_img = cv2.bitwise_and(img_edges, mask)

     cv2_imshow(masked_img)
```

Output:



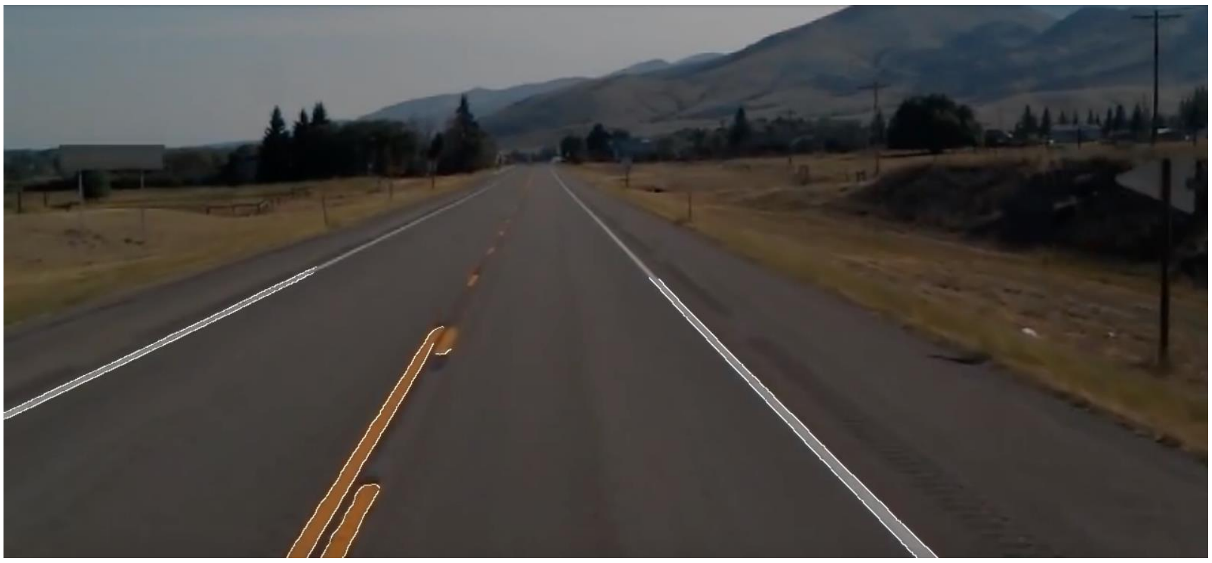


```
[ ] def hough2(img,threshhold = 100):
        rhos=[]
        thetas = []
        theta = np.arange(0,180)
        x, y = img.shape
        max_dist = int((np.ceil(np.sqrt(x)**2 + (y)**2)))
        accumulator = np.zeros((max_dist, len(theta)))

        for i in range(x):
            for j in range(y):
                if img[i][j]==255:
                    #accumulator function
                    for th in theta:
                        rho = j* np.cos((th*(np.pi/180))) + i * np.sin((th*(np.pi/180)))
                        rho = round((rho))
                        accumulator[int(rho)][th] = accumulator[int(rho)][th] + 1

        for i in range(x):
          for j in range(y):
            #looking for the highest peaks of the accumulator function
            if accumulator[int(rho)][th] >= threshhold:
                  rhos.append(int(rho))
                  thetas.append(th)
            #performing non-maximum suppression for lower values
            else:
                  accumulator[int(rho)][th] = 0

        return rhos, thetas, accumulator
```



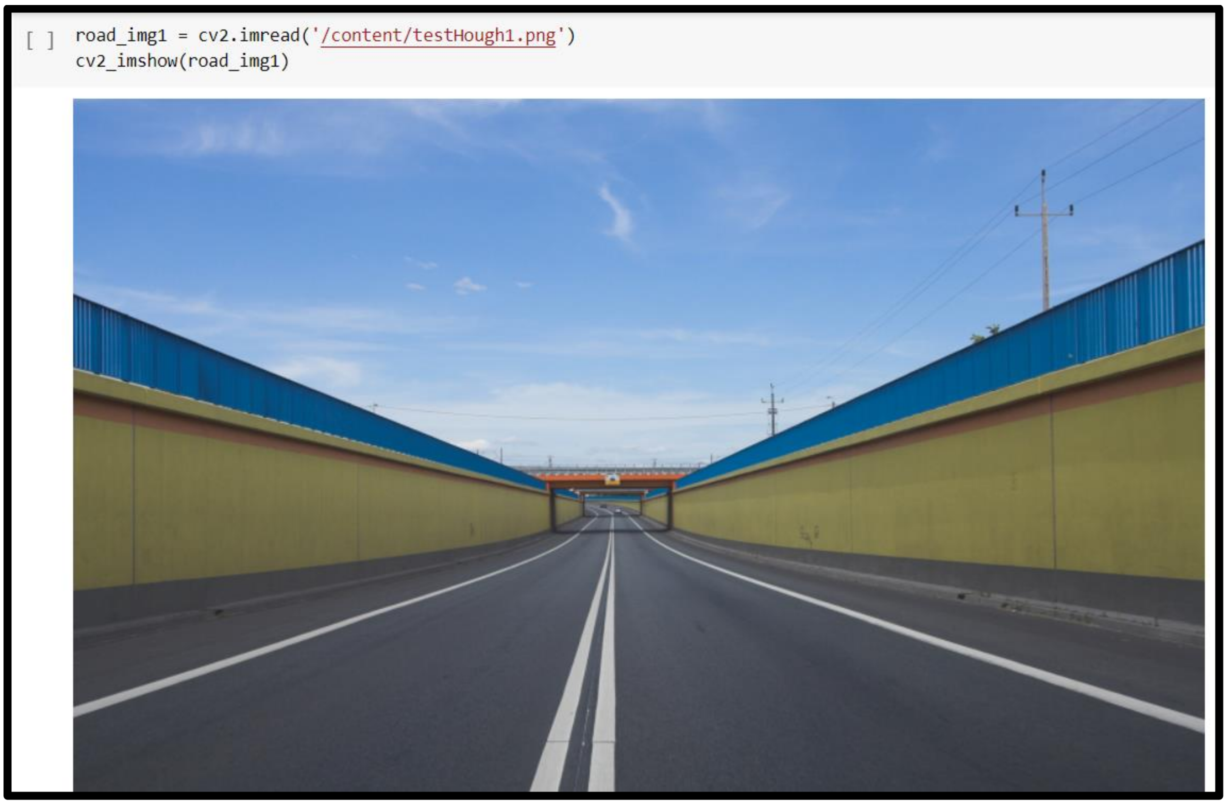
```
[ ] def detect_lanes2(img, masked_img, polygon):
      rhos, thetas, H = hough2(masked_img)

      for i in range(len(rhos)):
          rho = rhos[i]
          theta = np.deg2rad(thetas[i])
          a = np.cos(theta)
          b = np.sin(theta)
          x0 = a*rho
          y0 = b*rho

          x1 = int(x0 + 1000*(-b))
          y1 = int(y0 + 1000*(a))
          x2 = int(x0 - 1000*(-b))
          y2 = int(y0 - 1000*(a))

          cv2.line(masked_img,(x1,y1),((x2),y2),(255,0,0),3)

      mask = np.zeros_like(masked_img)
      cv2.fillPoly(mask, polygon, 255)
      masked_image = cv2.bitwise_and(masked_img, mask)

      backtorgb = cv2.cvtColor(masked_image,cv2.COLOR_GRAY2RGB)
      output = cv2.addWeighted(img,0.7,backtorgb,0.7,0)
      cv2_imshow(output)
```

Improved output:

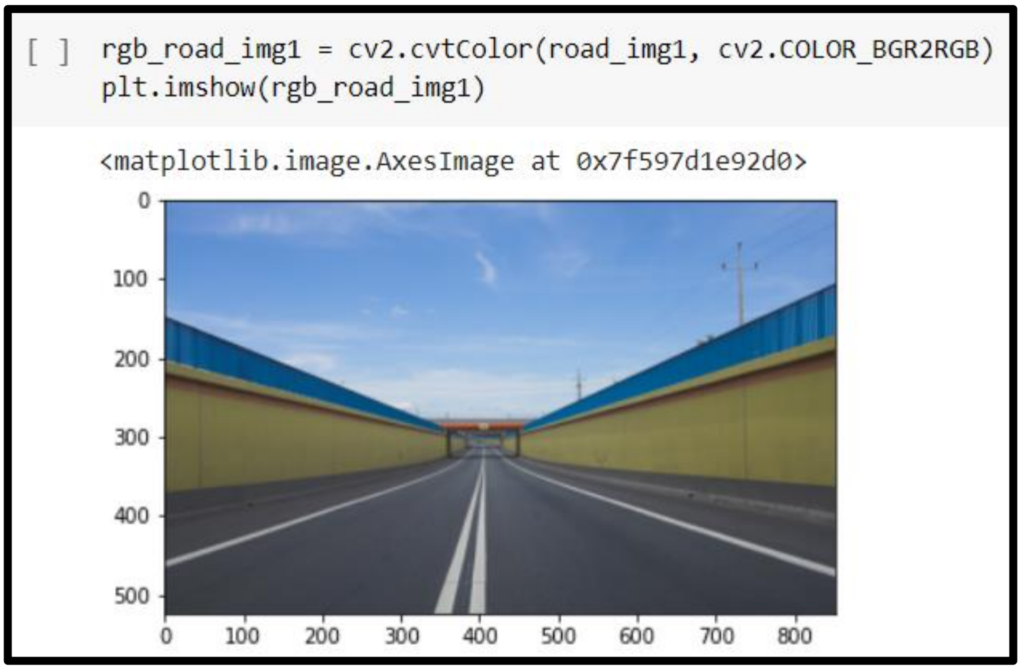


Testing another image

The same previous steps were followed



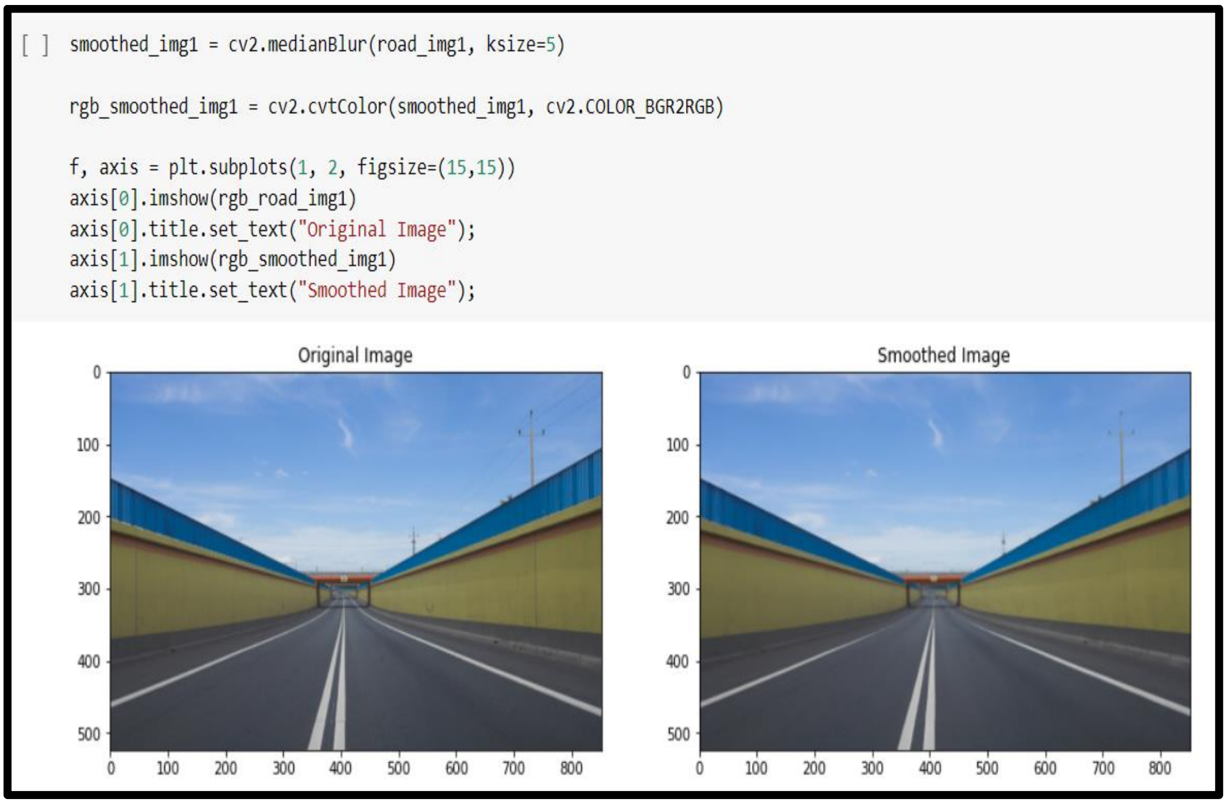
```
[ ] road_img1 = cv2.imread('/content/testHough1.png')
    cv2_imshow(road_img1)
```



```
[ ]  rgb_road_img1 = cv2.cvtColor(road_img1, cv2.COLOR_BGR2RGB)
     plt.imshow(rgb_road_img1)
```

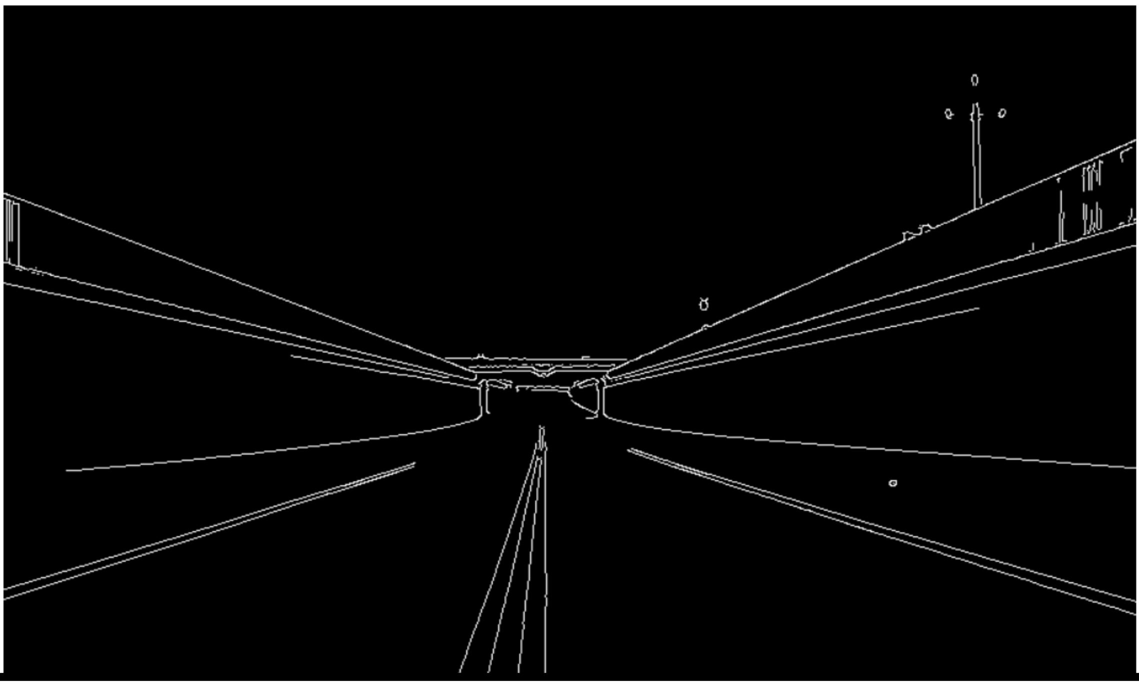
```
<matplotlib.image.AxesImage at 0x7f597d1e92d0>
```

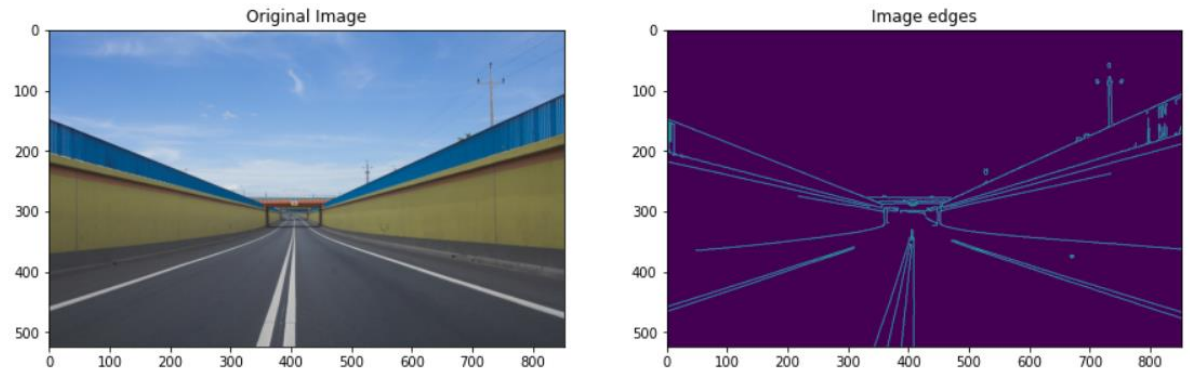
We used a smaller kernel size because when we tried **ksize = 9**, some details of the lanes were lost.



```
[ ]  smoothed_img1 = cv2.medianBlur(road_img1, ksize=5)

     rgb_smoothed_img1 = cv2.cvtColor(smoothed_img1, cv2.COLOR_BGR2RGB)

     f, axis = plt.subplots(1, 2, figsize=(15,15))
     axis[0].imshow(rgb_road_img1)
     axis[0].title.set_text("Original Image");
     axis[1].imshow(rgb_smoothed_img1)
     axis[1].title.set_text("Smoothed Image");
```

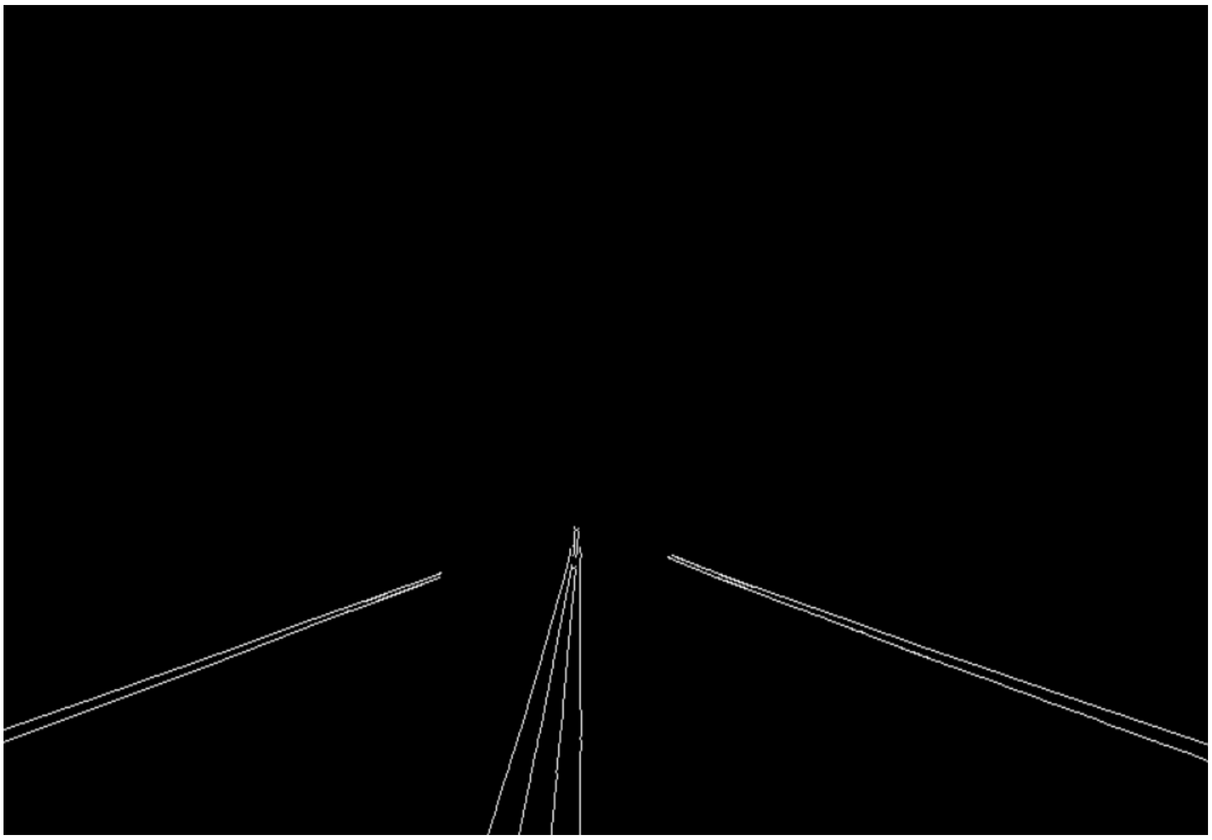
```
[ ]  img_edges1 = cv2.Canny(rgb_smoothed_img1, 100, 200)
     cv2_imshow(img_edges1);
```





We defined the polygon as a triangle as well, but we adjusted the measurements for the lanes of this image.

```
[ ]  img_edges_ROI1 = img_edges1.copy()

     height1 = img_edges_ROI1.shape[0]
     #polygon used is a triangle as it's the most suitable for
     #cropping out the 3 lanes only
     polygon1 = np.array([[(-300, height1), (1300, height1), (400, 320)]])
     mask1 = np.zeros_like(img_edges1)
     cv2.fillPoly(mask1, polygon1, 255)
     masked_img1 = cv2.bitwise_and(img_edges1, mask1)

     cv2_imshow(masked_img1)
```



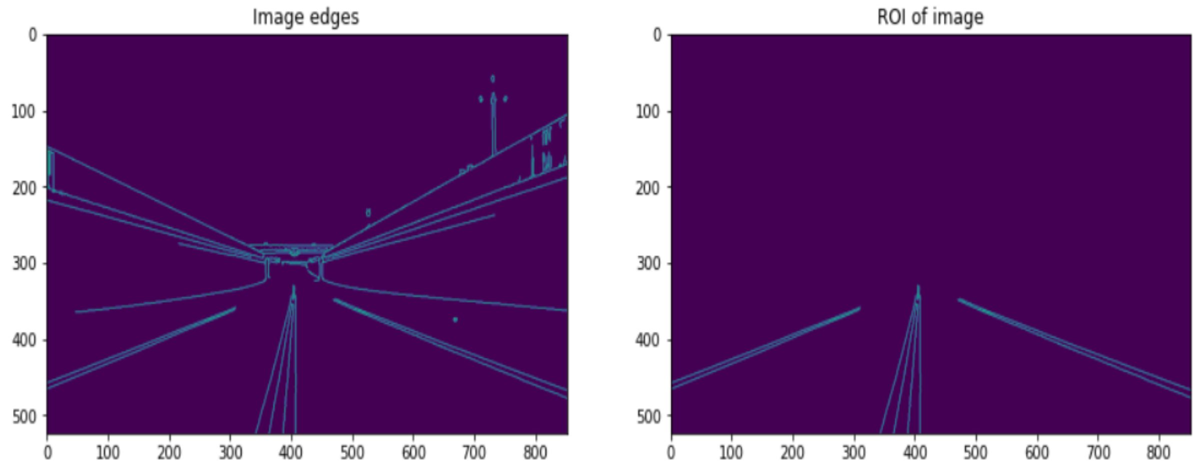
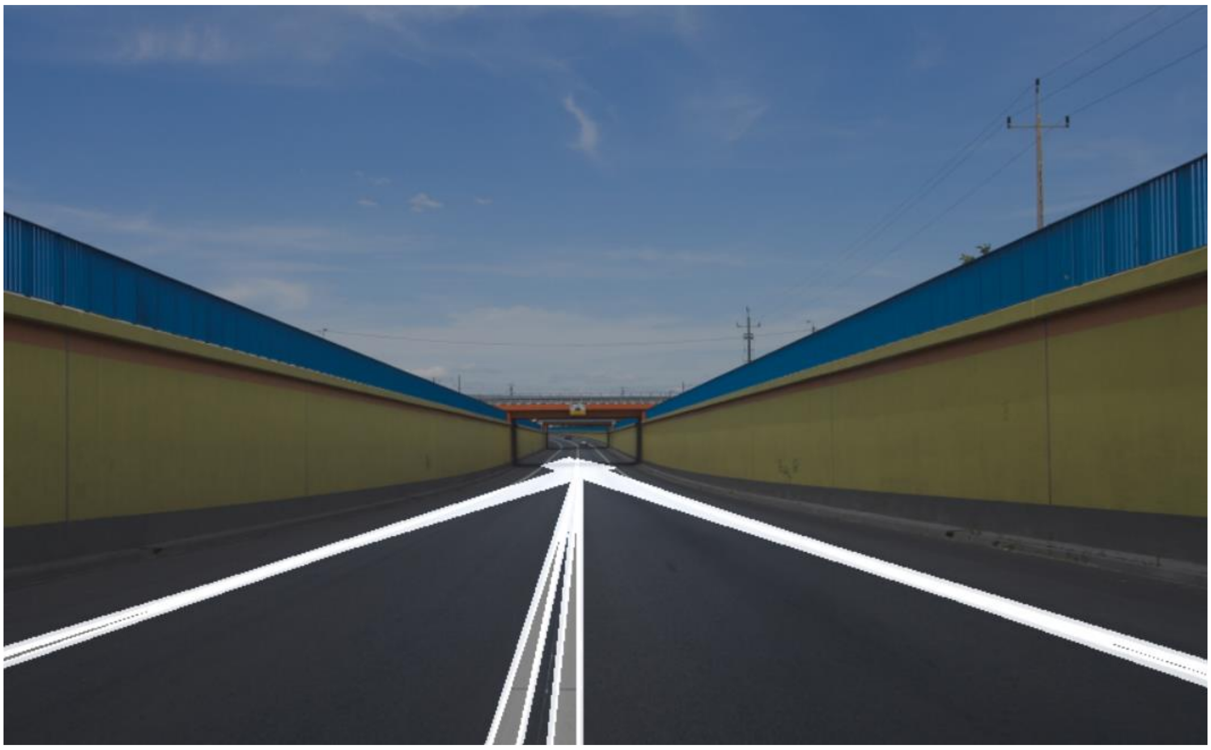


Image edges
ROI of image
0
100
200
300
400
500
0
100
200
300
400
500
600
700
800

Output:

Before improvement:



After improvement: