Assignment 1

Team Members

Aya Khames Khairy	18010442
Basel Ayman Mohamed	18010458
Pancee Wahid Mohamed	18010467

Part I: Applying Image Processing Filters For Image Cartoonifying

```
[1] import cv2
    from google.colab.patches import cv2_imshow
    import numpy as np
    from matplotlib import pyplot as plt
    import math
    from scipy.ndimage.filters import maximum_filter
```

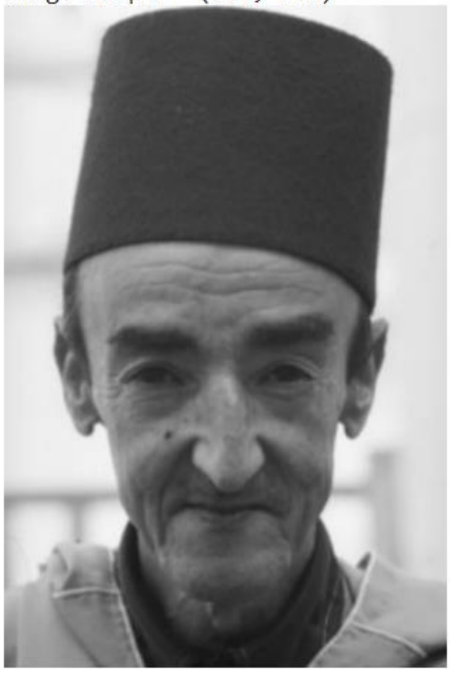
```
[2] # drive mount
    from google.colab import drive
    drive.mount('/content/drive')
    folder_path = "/content/drive/MyDrive/test/"
```

Read the image in grayscale and display it

The image is read in grayscale using cv2_imshow() to be operated on.

```
[39] file_path = folder_path + "9.jpg"
     img = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE) # 0
     print("Image Shape: ", img.shape)
     cv2_imshow(img)
```

```
Image Shape:  (371, 248)
```



1. Generating a black-and-white sketch

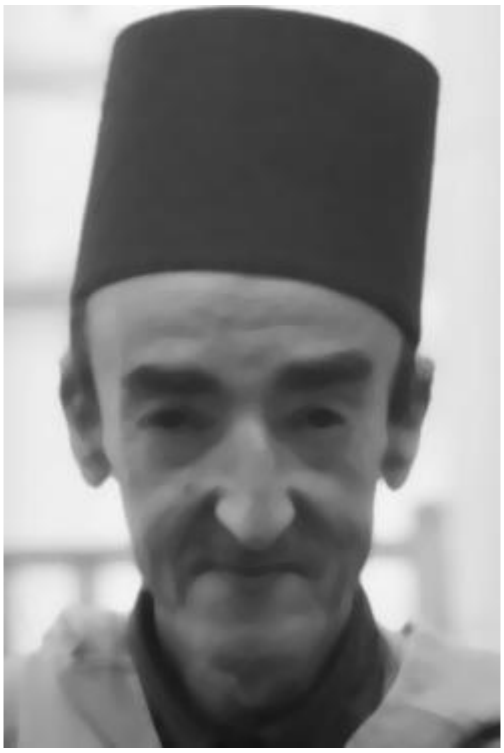
We used the median blurring filter to preserve the edges while getting rid of the noise. Then, we apply a Laplacian filter for edge detection as it produces image edges more similar to hand sketch. OpenCv Laplacian filter function takes the image in grayscale as we already read it. Finally, a binary threshold is applied on the output of the Laplacian filter function to have edges either white or black as it produces edges with varying intensities.

```
[11] # median filter -- good at removing noise while keeping edges sharp
     img_blurred = cv2.medianBlur(img, 7)

     # laplacian filter -- works only on gray scale
     img_laplacian = cv2.Laplacian(img_blurred, -1, ksize=5)

     # apply binary threshold -- edge is black or white
     _,img_sketch = cv2.threshold(img_laplacian, 127, 255, cv2.THRESH_BINARY_INV)
```

For the median blurring filter, kernel size is set to 7 as trying 3 didn't give enough blurring while trying 9 made the image lose so many details.
For the Laplacian filter, kernel size is set to 5 as trying 3 didn't detect edges except for the face outline while trying 7 captured too many edges that we don't need.

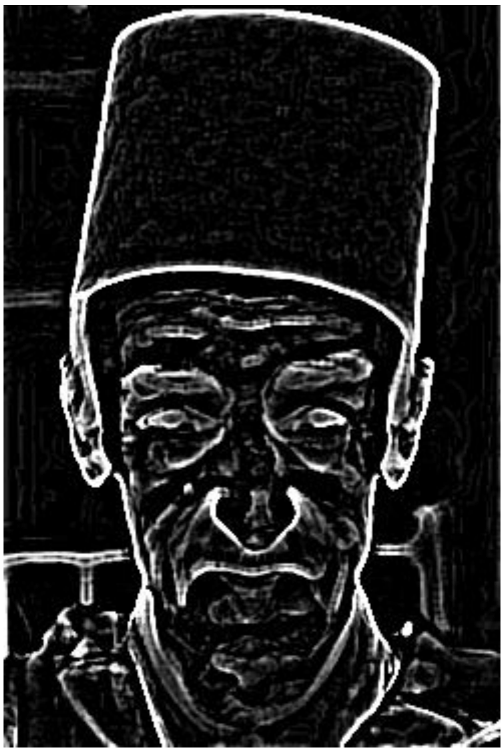
The image after applying median blurring filter:

```
[12] cv2_imshow(img_blurred)
```



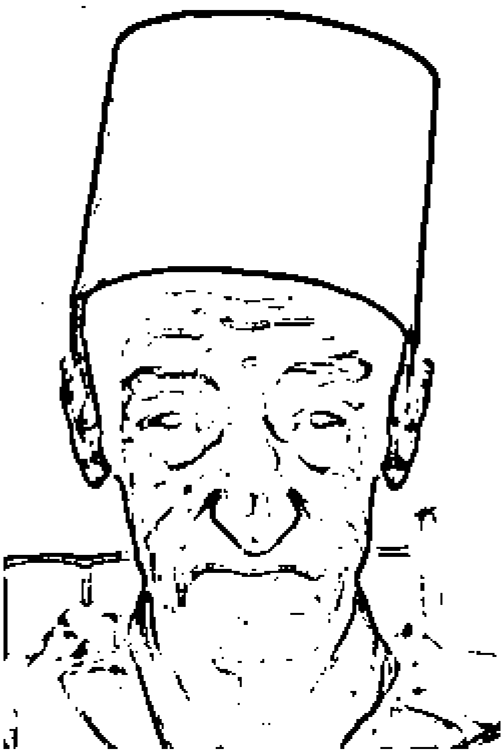
The image after applying laplacian filter:

```
[6]  cv2_imshow(img_laplacian)
```



The image after applying binary threshold having our final edges:

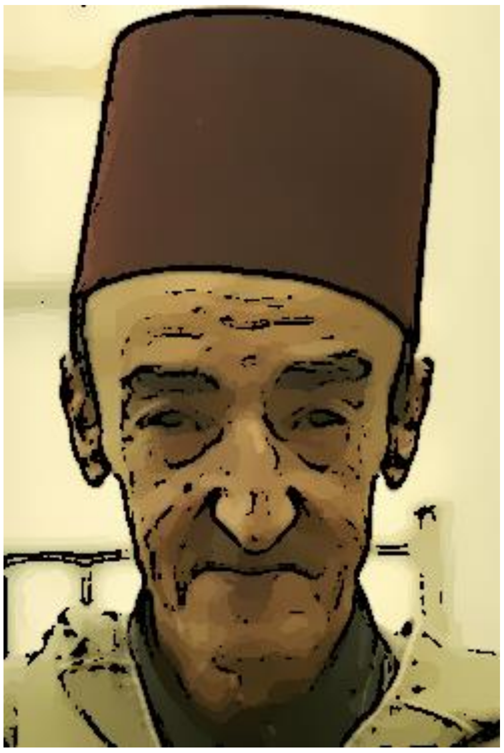
```
[7]  cv2_imshow(img_sketch)
```



2. Generating a color painting and a cartoon

Bilateral filter is used to get a cartoonified image as it smoothes the flat regions while keeping the edges sharp.
By tuning, we applied 15 small bilateral filters each of kernel size 9.
We used a large value for $\boldsymbol{\sigma}_r$ to make the regions more flat.
The image sketch array is divided by 255 to have 0's for edges' pixels and 1's otherwise.
Then, for every channel in the BGR image, we do element-wise multiplication of the channel array and the updated edges array, so that every pixel corresponding to 0 (edge) pixel is set to 0 to be black (edge) and every pixel corresponding to 1 (non-edge) stays as it is (multiplied by 1).

```
[77] # apply many small bilateral filters to produce a strong cartoon effect in less time
     cartoon = cv2.imread(file_path)

     for i in range(15):
       cartoon = cv2.bilateralFilter(cartoon, 9, 8, 9)

     edges = img_sketch.copy()
     edges = edges / 255.0

     final = cartoon.copy()

     final[:,:,0] = cartoon[:,:,0] * edges
     final[:,:,1] = cartoon[:,:,1] * edges
     final[:,:,2] = cartoon[:,:,2] * edges

     cv2_imshow(final)
```



Part II: Road Lane Detection Using Hough Transform

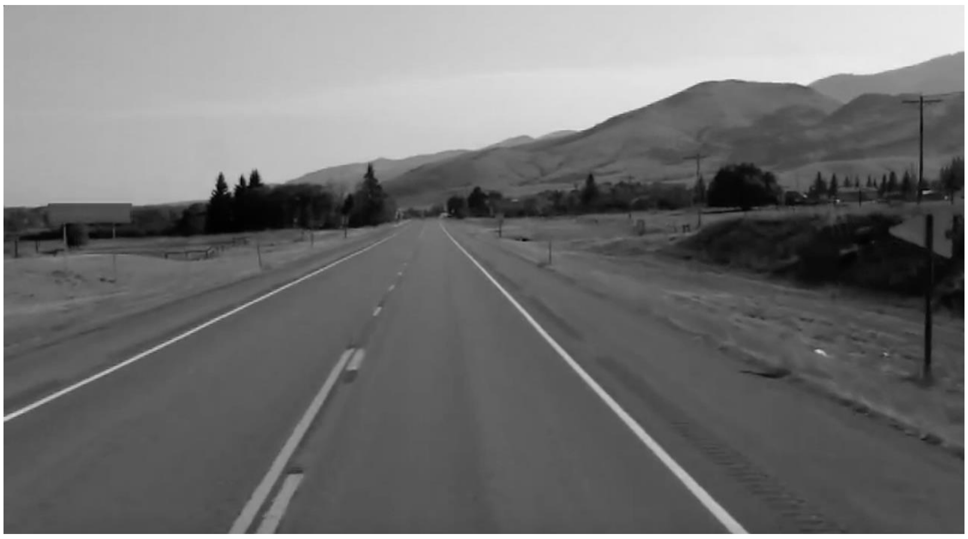
Smoothing the grayscale image

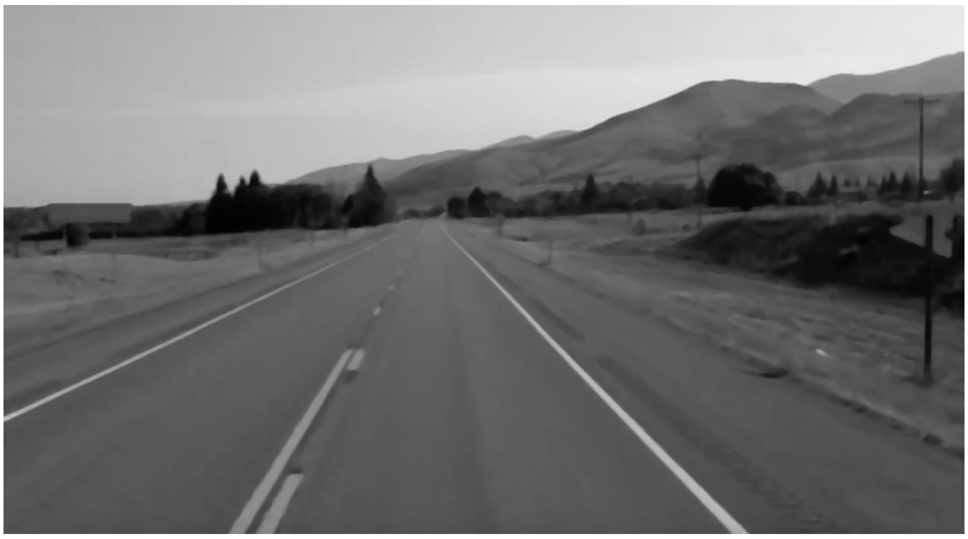
Same as part I, the image is read in grayscale using OpenCV function, imread().

The image is then smoothed using a median blurring filter.

Image height, width and diagonal of the image are calculated as they will be used later.

```
[ ]  hough_file_path = folder_path + "testHough.jpg"
     hough_img = cv2.imread(hough_file_path, cv2.IMREAD_GRAYSCALE) # 0
     cv2_imshow(hough_img)

     height = hough_img.shape[0]
     width = hough_img.shape[1]
     diagonal = int(round(math.sqrt(math.pow(height, 2) + math.pow(width, 2))))

     hough_smoothed = cv2.medianBlur(hough_img, 7)
     cv2_imshow(hough_smoothed)
```

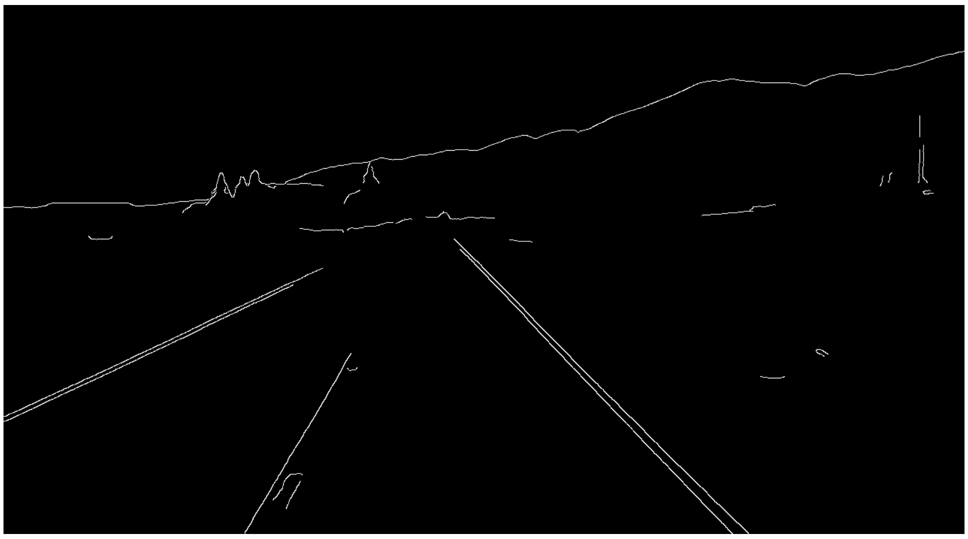




Edge Detection

Canny's algorithm is used to detect edges in the image. We used the Canny() function of OpenCV.

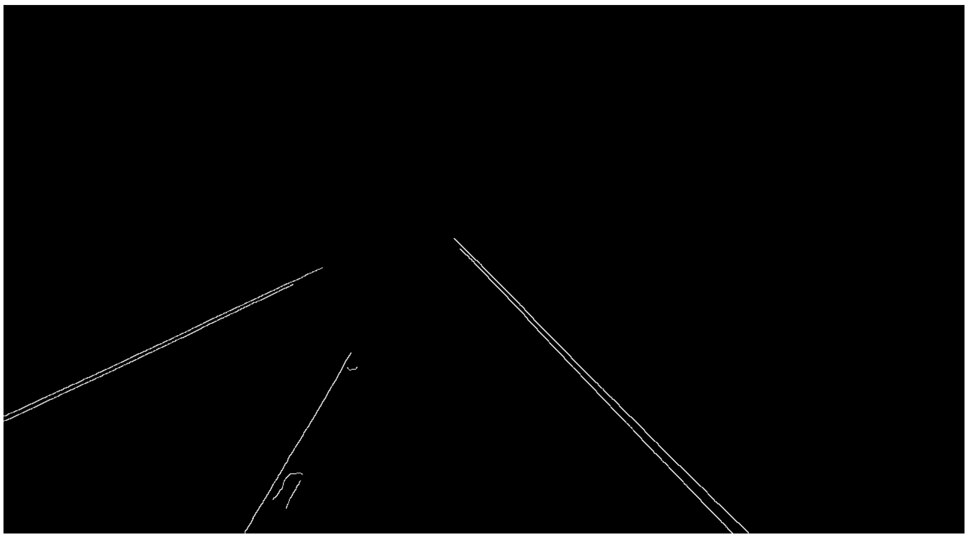
```
[ ]  edges = cv2.Canny(hough_smoothed, 90, 180)
     cv2_imshow(edges)
```



Region Of Interest

Unnecessary edges of the background were detected. We don't need these edges. So, to eliminate the noise, we defined a polygon area to mask by setting all pixels in this region to 0 (black). We used fillPoly() function to fill the pixels in this polygon with 0's. The vertices of this polygon were decided by having the two upper corners as vertices and the rest were decided by tuning till we got the best polygon that covers everything and leaves only the road.

```
[ ]  ROI = edges.copy()
     points = np.array([[0, 0], [0, 500], [426, 320],
                        [620, 290], [1040, height],
                        [width, height], [width, 0]])
     cv2.fillPoly(ROI, pts=[points], color=(0))
     cv2_imshow(ROI)
```



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

Accumulation matrix of dimensions 180 (θ) and (2 x diagonal length) of the image (ρ) is initialized by zeros. Then, for each edge pixel in the image (intensity = 255), we calculate ρ and increment H(θ, ρ) (number of votes) for each (integer) value of θ.

```
[ ]  H = np.zeros((181, (2*diagonal)+1))
     for j in range(width):
       for i in range(height):
         if ROI[i, j] == 255:
           for theta  in range (0, 181):
             p = int((i * math.cos(theta) + j * math.sin(theta)) + diagonal)
             H[theta, p] = H[theta, p] + 1
     print(np.unique(H))
```

```
[  0.   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.  12.  13.
  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.
  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.
  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.
  56.  57.  58.  59.  60.  61.  62.  63.  64.  65.  66.  67.  68.  69.
  70.  71.  72.  73.  74.  75.  76.  77.  78.  79.  81.  82.  84.  85.
  86.  88.  90.  91.  92.  93.  94.  96.  97.  98. 100. 101. 102. 103.
 105. 109. 111. 112. 113. 114. 115. 118. 119. 121. 122. 126. 127. 128.
 134. 135. 139. 141. 143. 144. 145. 147. 150. 155. 157. 161. 176. 188.
 198. 204. 215. 221. 230. 240. 262. 282. 285. 294. 297. 298. 342.]
```

Refining Coordinates and HT Post-Processing

To eliminate noise, we set the votes at the points with votes <= 150 to 0. This number was chosen also by tuning. Then, we look for the highest peaks of the accumulator function and perform non-maximum suppression for lower values. We used the maximum_filter() function of the SciPy library.

```
[ ]  for i in range(181):
       for j in range(2*diagonal + 1):
         if H[i, j] <= 150:
           H[i, j] = 0

     out = maximum_filter(H, 433)
     unique_vals = np.unique(out)
     print(unique_vals)
```

```
[  0. 155. 215. 285. 294. 297. 298. 342.]
```

We get the unique (θ, ρ) pairs.

```
[ ]  theta_p = np.zeros((1,2))
     for i in range(181):
       for j in range(2921):
         if H[i,j] in unique_vals and H[i, j] != 0:
           theta_p = np.append(theta_p, [[i, j - 1460]], axis=0)
     theta_p = np.delete(theta_p, (0), axis=0)
     print(theta_p)
```

```
[[  35. -502.]
 [  35. -497.]
 [  67. -638.]
 [  89.  634.]
 [ 106. -221.]
 [ 128.  212.]
 [ 172.  204.]]
```

Then the get_point() function is used to get the y-coordinate given x-coordinate, θ and ρ values.

```
[ ]  def get_point(rho, theta, x):
       y = (rho-(x*math.cos(theta)))/(math.sin(theta))
       return y
```

```
[ ]  img = cv2.imread(hough_file_path)
     x1 = 330
     x2 = 703
```

draw_line() function is used to draw a line on the original image. We used the line() function from OpenCV to draw the line on the image. θ and ρ are passed to draw_line() which calls get_point() to get the two points of the line segment to draw then call cv2.line() to draw it.

```
[ ]  def draw_line(theta, rho):
       y1 = int(get_point(rho, theta, x1))
       y2 = int(get_point(rho, theta, x2))
       cv2.line(img, (y2,x2), (y1,x1), color=(0, 255, 255), thickness=2)
```

For each (θ, ρ) pair we get, a line is drawn on the image.

```
[ ]  for row in theta_p:
       draw_line(row[0], row[1])
     cv2_imshow(img)
```