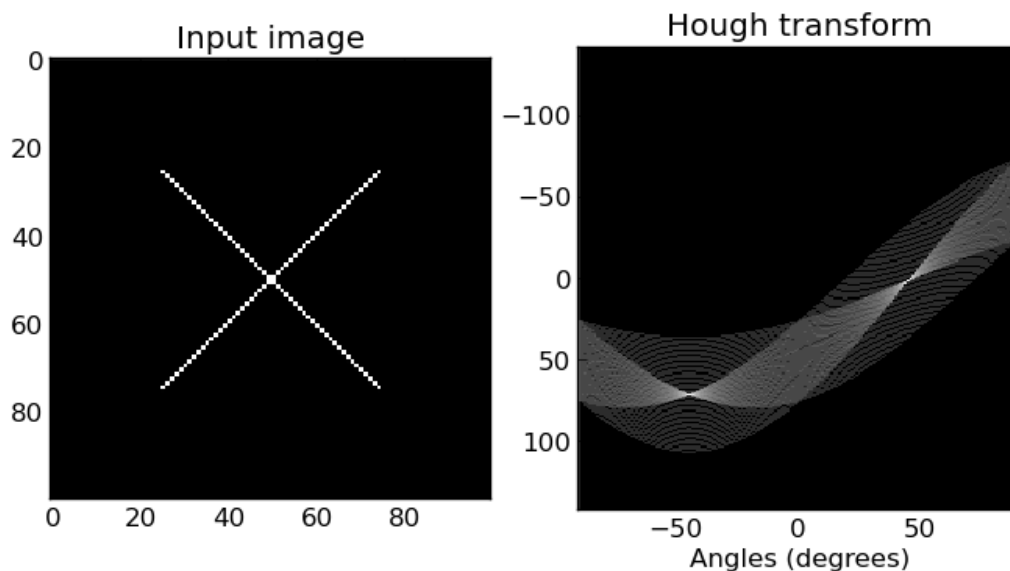


به نام خدا
مهدی فیروزبخت
تمرین سری دوم
درس بینائی ماشین

۱.

روش هاف پایه : تبدیل هاف (به انگلیسی: Hough transform) روشی برای استخراج ویژگی ها در آنالیز تصاویر، بینایی رایانه ای و پردازش تصویر دیجیتال است. این روش در یک تصویر به دنبال نمونه هایی از یک الگو می گردد. این نمونه ها ممکن است کامل نباشند و همچنین تا حدی دچار اعوجاج شده باشند. به عنوان نمونه از کاربردهای این روش می توان به تشخیص وجود خط مستقیم در یک تصویر اشاره کرد.



در این تبدیل انواع مختلفی از تبدیلات برای خط مستقیم وجود دارد که یک روش تبدیل نقطه به خط به معادله خط $b = -ax + y$ است که برای این فضای جدید یک ماتریس به اندازه آن وجود دارد که محل عبور هر خط از هر قسمت فضا و دستگاه را مشخص میکند که این ماتریس یا توری میتواند هر خانه آن به اندازه خانه فضای پارامترها یا ویژگی ها باشد یا به اندازه بزرگتر یا کوچکتر از آن باشد که اگر خیلی بزرگ باشد ممکن است خط های نزدیک به هم را یک خط در نظر بگیرد و اگر خیلی کوچک باشد ممکن است بعضی از خطوط را به عنوان نویز در نظر بگیرد و آن را حذف نماید. در نوع دیگری از این تبدیل برای خطوط مستقیم ، به جای معادله فضای $b = -ax + y$ از معادله جدیدی استفاده میکند .

این معادله سینوسی به صورت $\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$ است. برای به دست آوردن این مقادیر، به ازای هر نقطه در تصویر تعدادی خط از نقطه عبور می‌دهیم و مقدار θ و ρ برای مقادیر مختلف را به دست می‌آوریم. مقدار ρ برابر فاصله مبدا تا خط که به صورت عمود بر خط است محاسبه می‌شود و مقدار θ برابر زاویه خط عمود بر خط گذرا از نقطه و محور x ها خواهد بود.

سپس با استفاده از این θ و ρ در دستگاه ρ و θ با استفاده از فرمول یادشده معادلات را رسم مینماییم. سپس پس از انجام این کار برای نقاط مختلف تصویر به تعدادی نقطه تقاطع برای نقاط خاصی از تصویر اصلی در فضای ρ و θ میرسیم که این نقاط نشان دهنده ρ و θ خاصی است که خط گذرنده از هر نقطه دارای این پارامترها است. این نقاط در ماتریس متناظر با فضای دوم دارای بالاترین مقادیر عبور تعداد زیادی از خطوط هستند که نشان‌دهنده این هستند که نقاط مختلف در این نقطه تشابه ویژگی زیادی دارند. پس از رسم این خطوط با پارامترهای بهینه یادشده به خطوط مستقیم مورد نظر خود خواهیم رسید. در بیشتر مواقع از این روش دوم استفاده می‌شود.

در هر دو روش یکی از راه‌های کنترل خطوط این است که می‌توانیم یک مقدار محدود کننده ای برای تعداد خطوط گذرنده از هر تقاطع مشخص کنیم که نشان‌دهنده این است که اگر تعداد خطوط گذرنده از یک نقطه از تعداد خاصی بیشتر باشد، که این مقدار خاص را می‌توانیم خودمان مشخص کنیم، این مقدار به عنوان پارامتر یک خط در نظر گرفته شود.

علت استفاده از روش دوم: در روش اول برای هر نقطه در فضای دوم یک خط از منفی بینهایت تا مثبت بینهایت می‌توانیم بکشیم که این حالت نیازمند یک حافظه و سرعت زیادی است تا بتواند ماتریس متناظر با این دستگاه را ذخیره و محاسبات آن را انجام دهد. اما در روش دوم این حالت محدود شده است. زیرا مقدار ρ که دامنه معادله سینوسی را مشخص میکند در فاصله محدودی است و همچنین مقدار θ در فاصله 0 تا 2π قرار دارد که مقدار آن نیز محدود است.

تابع `cv2.HoughLinesP` که در صورت سوال به عنوان مسئله بیان شده است، یکی از تابع‌های کتابخانه OpenCV است که تمام مراحل ذکر شده را بر روی تصویر انجام داده و تعدادی نقطه به عنوان خروجی بیان می‌کند. خروجی این تابع به صورت آرایه ای از نقاط است که هر خانه این آرایه دارای ۴ پارامتر است که ۲ مقدار اول مشخص کننده x_1, y_1 نقطه شروع خط و ۲ پارامتر دوم مشخص کننده x_2, y_2 نقطه پایان خط هستند.

در کتابخانه OpenCV ۲ نوع از تبدیل هاف بیان شده است. در روش اول که با تابع `HoughLines()` مشخص می‌شود، تمام مراحل بالا دقیقاً انجام شده و خروجی آن به صورت (θ, ρ) خواهد که همانند مدل پایه ای است که در تبدیل هاف بیان شده است. در نتیجه بردار زوج‌ها (θ, ρ) را می‌دهد.

اما تبدیل دوم که با تابع `cv2.HoughLinesP` قابل استفاده است و به از تبدیل احتمالی خط هاف استفاده میکند خواهد بود که یک اجرای کارآمدتر تبدیل خط هاف است. تمام مراحل ذکر شده را بر روی تصویر انجام داده و تعدادی نقطه به عنوان خروجی بیان میکند. خروجی این تابع به صورت آرایه ای از نقاط است که هر خانه این آرایه دارای ۴ پارامتر است که ۲ مقدار اول مشخص کننده x_1 , y_1 نقطه شروع خط و ۲ پارامتر دوم مشخص کننده x_2 , y_2 نقطه پایان خط هستند.

برای انجام کار با تبدیل هاف ، ابتدا تصاویر را به یک الگوریتم لبه یابی داده تا لبه ها را کشف کند سپس لبه های کشف شده را وارد تبدیل هاف کرده و محاسبات را روی آن انجام داده و نتیجه را مانند یک ماسک به روی تصویر اصلی می اندازیم تا روی تصویر اصلی خطوط صاف خواسته شده مشخص شود.

پس در مجموع تفاوت روش پایه با روش دوم در استفاده از تبدیل احتمالی خط هاف است . حال تبدیل احتمالی خط هاف چیست ؟ :

یک تبدیل Hough در صورتی احتمالی در نظر گرفته می شود که از نمونه گیری تصادفی از نقاط لبه استفاده کند. این الگوریتم ها را می توان بر اساس نحوه نگاشت فضای تصویر به فضای پارامتر تقسیم کرد. یکی از ساده ترین روش های احتمالی، انتخاب m نقاط لبه از مجموعه نقاط لبه M است. پیچیدگی مرحله رای گیری از $O(M.N\theta)$ به $O(m.N\theta)$ کاهش می یابد. این کار به این دلیل کار می کند که یک زیر مجموعه تصادفی از M نسبتاً نقاط لبه و نویز و انحراف اطراف را نشان می دهد. مقدار کوچکتر m منجر به محاسبه سریع اما دقت کمتر می شود. بنابراین مقدار m باید به طور مناسب با توجه به M انتخاب شود.

پارامتر های این تابع به صورت زیر می باشند : ۱. `Dst` . ۲. `Lines` . ۳. `Rho` . ۴. `Theta` . ۵. `Threshold` . ۶. `minLineLength` . ۷. `maxLineGap`

۱. `Dst` : تصویر خروجی آشکارساز لبه است که باید به عنوان ورودی به تابع داده شود. باید یک تصویر خاکستری باشد. (اگرچه در واقع یک تصویر باینری است)

۲. `Lines` : متغیری است که نقاط خطوط در آن ذخیره میشود.

۳. `Rho` : وضوح پارامتر r بر حسب پیکسل. ما از ۱ پیکسل استفاده می کنیم.

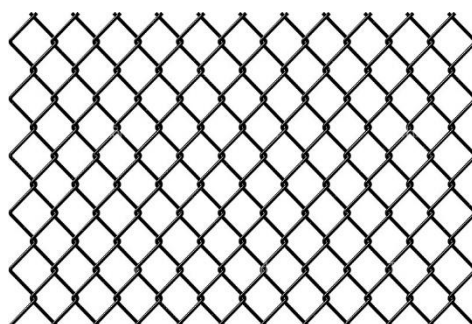
۴. `Theta` : وضوح پارامتر θ بر حسب رادیان. ما از ۱ درجه استفاده می کنیم. $np.pi/180$ به ما ۱ درجه را میدهد.

۵. `Threshold` : حداقل تعداد تقاطع ها برای "تشخیص" یک خط.

۶. `minLineLength` : حداقل تعداد نقاطی که می تواند یک خط را تشکیل دهد. خطوط با کمتر از این تعداد امتیاز نادیده گرفته می شوند.

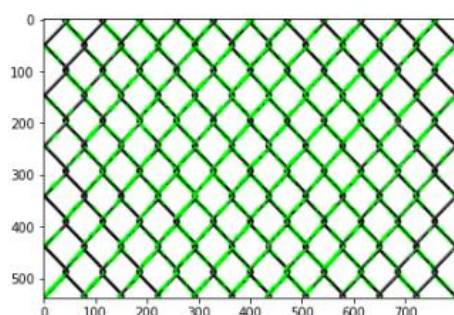
۷. `maxLineGap` : حداکثر فاصله بین دو نقطه در یک خط در نظر گرفته شود.

در تصاویر زیر تفاوت استفاده از داده های متفاوت را مشاهده میکنیم . تصویر اصلی به صورت زیر میباشد :

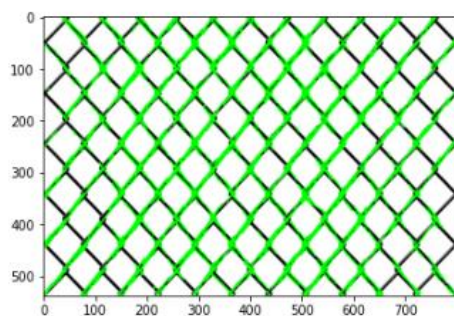


تاثیر تغییر RHO :

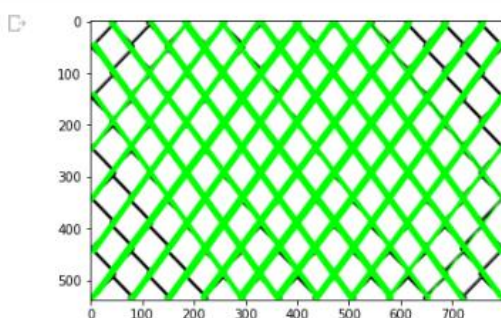
```
[159] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=1)
```



```
[160] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=10)
```

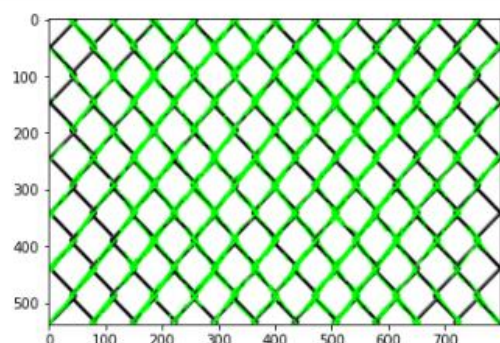


```
[161] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=100)
```

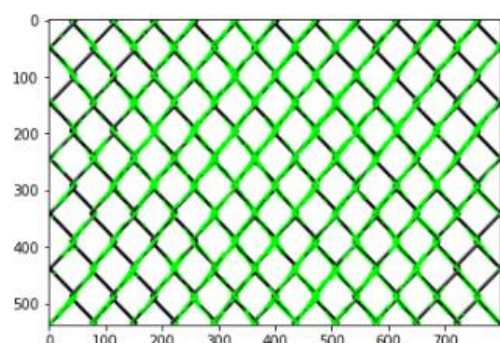


تاثیر تغییر Theta :

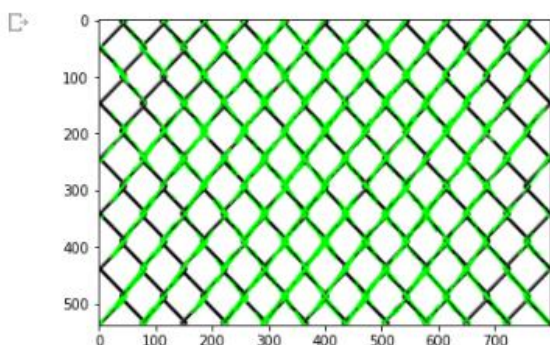
```
[162] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=10)
```



```
[163] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/60 ,threshold=100 ,minLineLength=1 ,maxLineGap=10)
```

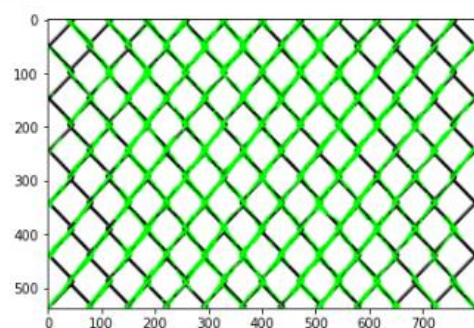


```
[164] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/30 ,threshold=100 ,minLineLength=1 ,maxLineGap=10)
```

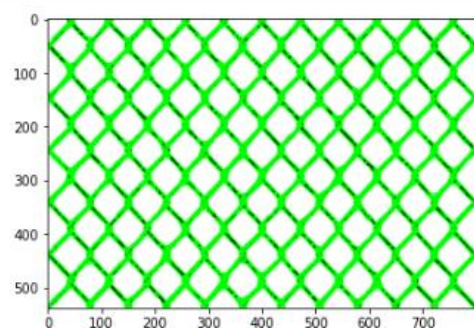


تاثیر تغییر Threshold :

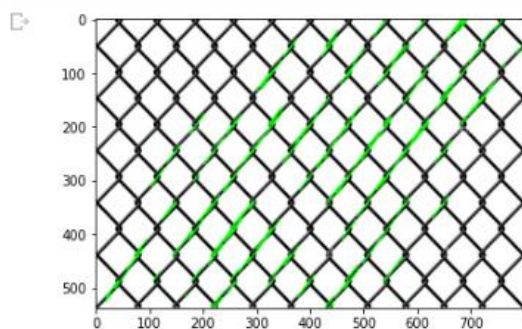
```
[153] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=10)
```



```
[154] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=10 ,minLineLength=1 ,maxLineGap=10)
```

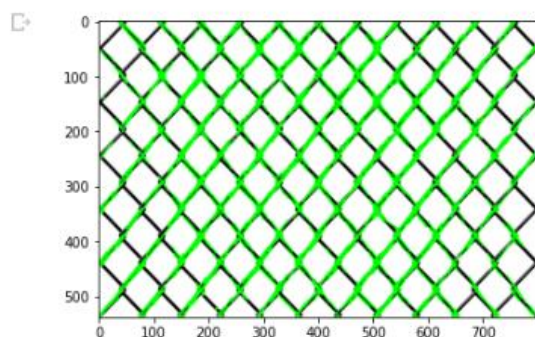


```
[156] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=200 ,minLineLength=1 ,maxLineGap=10)
```

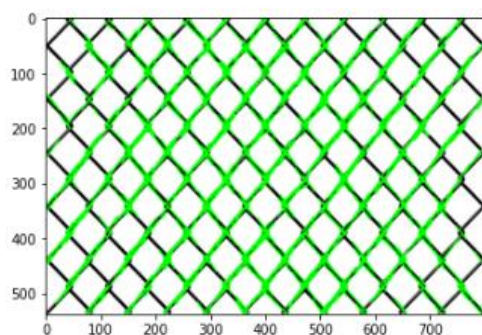


تاثیر تغییر minLineLength :

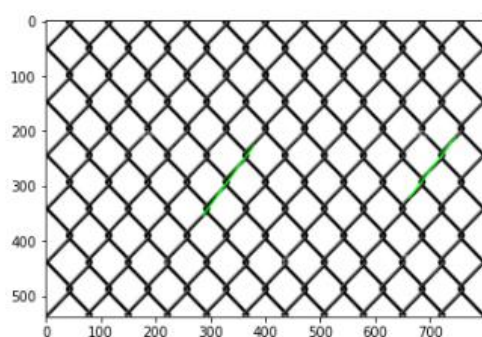
```
[153] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=10)
```



```
[157] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=10 ,maxLineGap=10)
```

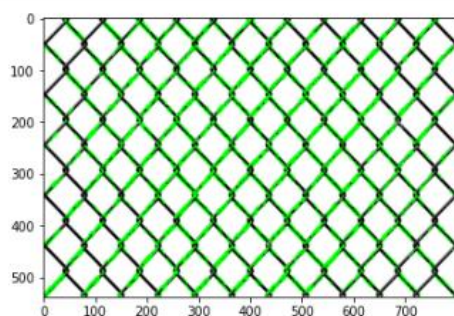


```
[158] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=100 ,maxLineGap=10)
```

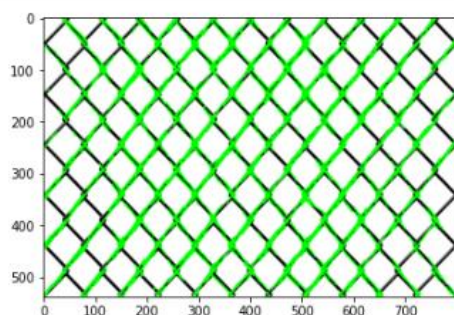


تاثیر تغییر maxLineGap :

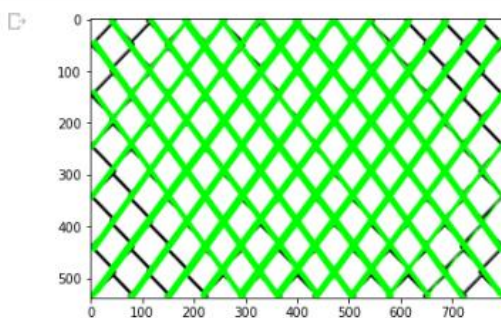
```
[159] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=1)
```



```
[160] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=10)
```



```
[161] useHoughLinesP(Dst=new_image ,Rho = 1 , Theta=np.pi/180 ,threshold=100 ,minLineLength=1 ,maxLineGap=100)
```



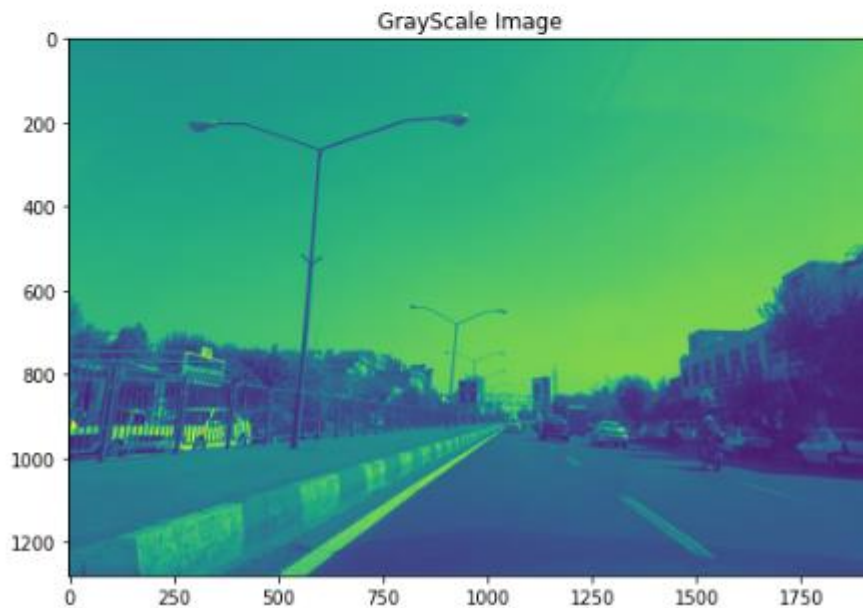
۲. الف. ابتدا تصویر خواسته شده را بار گذاری مینماییم. سپس تصویر بارگذاری شده را به سطح خاکستری میبریم. علت این کار این است که به ساده سازی الگوریتم ها کمک می کند و همچنین پیچیدگی های مربوط به نیازهای محاسباتی را حذف می کند به این دلیل که مقیاس خاکستری یک تصویر را به حداقل پیکسل آن فشرده می کند. تجسم آسان را افزایش می دهد. کدها و تصویر جدید در زیر نمایش داده شده است :

```
import matplotlib.pyplot as plt
import numpy as np
import cv2

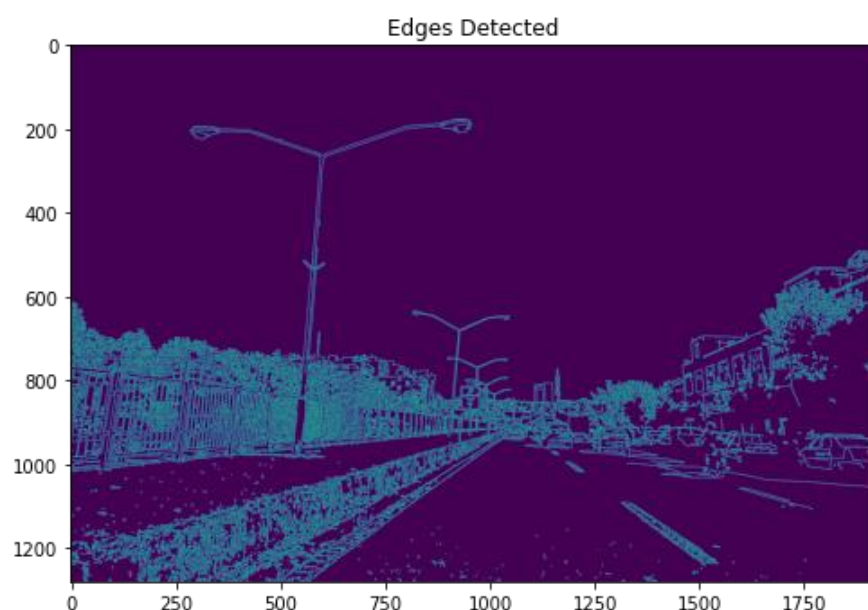
image = cv2.imread('img1.jpg')
```

Chaneg Image to GrayScale

```
[42] gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
plt.figure(figsize=(8,6))
plt.title("GrayScale Image")
plt.imshow(gray)
plt.show()
```



در مرحله بعد قبل از لبه یابی نیاز است برای از بین بردن نویز و لبه های اضافی از یک راهکار هموارسازی استفاده کنیم تا نویزهای لبه را از بین ببرد. تصویر بدون نویز گیری به صورت زیر می باشد :



مشاهده میکنیم تصویر لبه ها پر از نویز و لبه های بی استفاده است. پس از هموارسازی استفاده میکنیم:

Use Gaussian Filter

```
[178] kernel_size = 7
      blur_gray = cv2.GaussianBlur(gray, (kernel_size, kernel_size), 0)
      plt.figure(figsize=(8,6))
      plt.title("GrayScale Image with Gaussian Blur")
      plt.imshow(blur_gray)
```


<matplotlib.image.AxesImage at 0x7f2fb58011d0>

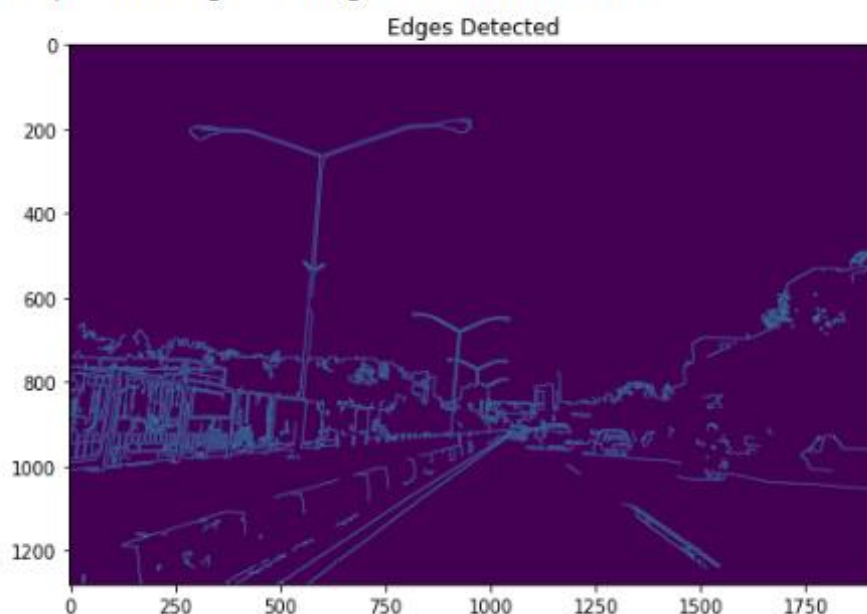


سپس تصویر جدید را به لبه یاب Canny وارد میکنیم تا لبه ها را استخراج نماید :

Use Canny with Image Smoothing

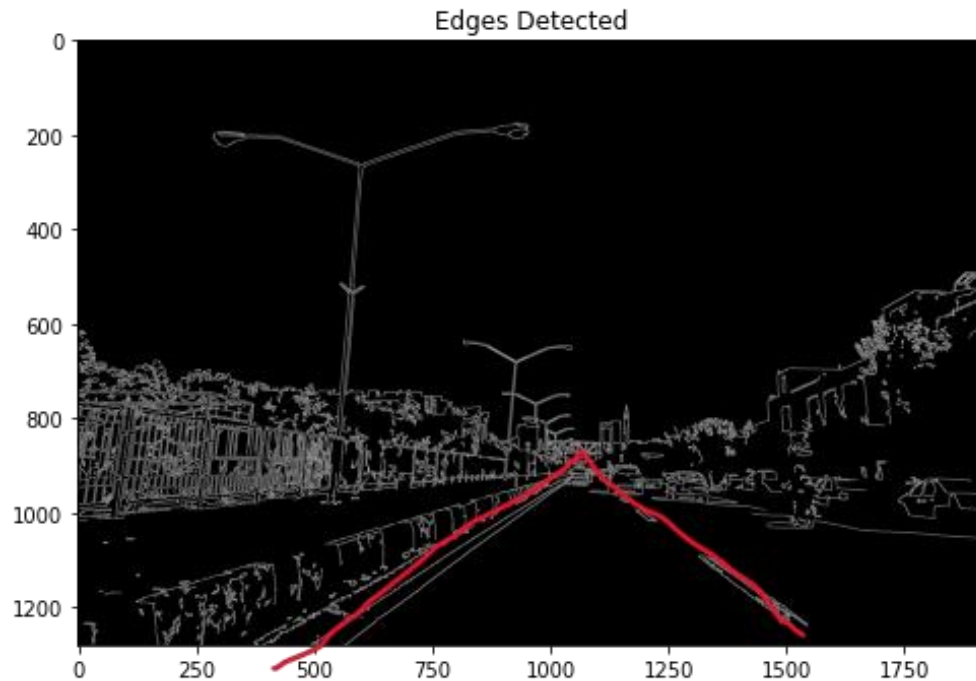
```
low_threshold = 50
high_threshold = 150
edges = cv2.Canny(blur_gray, low_threshold, high_threshold)
plt.figure(figsize=(8,6))
plt.title("Edges Detected")
plt.imshow(edges)
```

 <matplotlib.image.AxesImage at 0x7f2fb51a2fd0>



مشاهده میشود تصویر جدید از لبه های بهتری نسبت به لبه های اولیه برخوردار است و لبه های مورد نیاز برای سوالات بعدی (خطوط جاده ای) در این حالت بهتر خواهند بود. برای پیدا کردن محدوده Canny از روش آزمون و خطا استفاده شده است تا بهترین نتیجه را ارائه نماید.

ب. در این قسمت طبق بیان صورت سوال ، نیاز است تا محدوده تصویر را تا حدی کوچک کنیم. برای این کار تصویر لبه های ساخته شده انتخاب میکنیم و با استفاده از روش آزمون و خطا سعی میکنیم تا به محدوده قرمز شده زیر برسیم :



پس با استفاده از کد های زیر و روش آزمون و خطا به ماسک زیر دست میابیم . ابتدا با روش آزمون و خطا مثلث محدوده را ساخته و درون vertices قرار میدهیم. سپس یک ماسک مانند تصویر اصلی میسازیم. با استفاده از تابع fillPoly درون ماسک ساخته شده محدوده مثلث را سفید میکنیم. سپس ماسک ایجاد شده را بر تصویر اصلی اعمال میکنیم. تصویر ماسک و اعمال آن بر روی تصویر لبه ها را مشاهده میکنید :

Find Region of Interest

```
vertices = np.array([(0,image.shape[0]),(1050, 920), (1070, 920), (image.shape[1],image.shape[0])], dtype=np.int32)

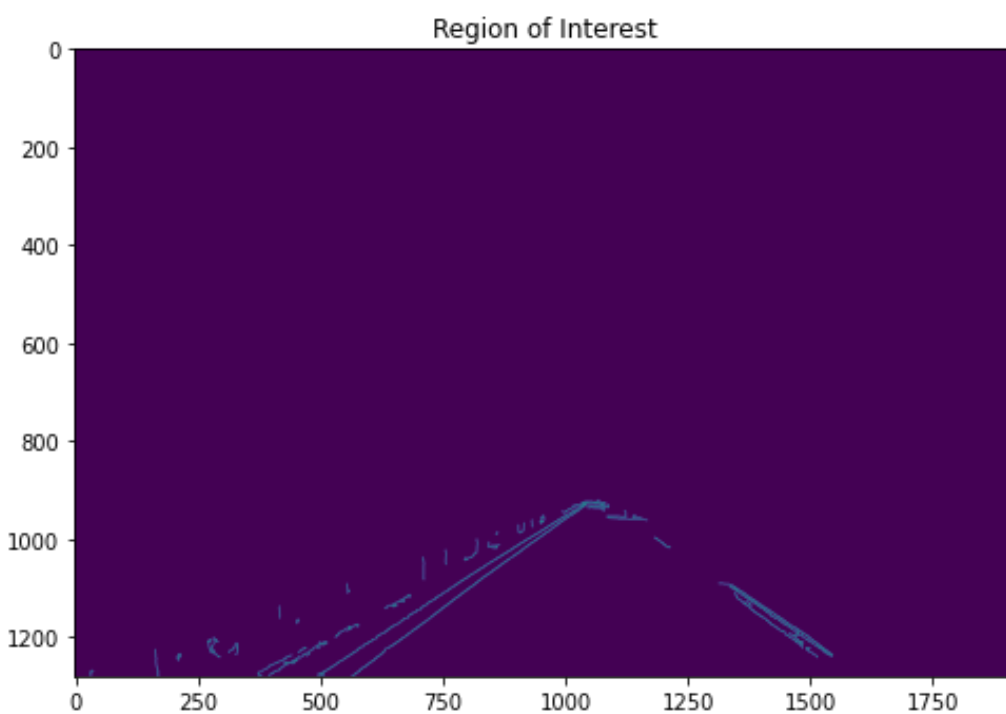
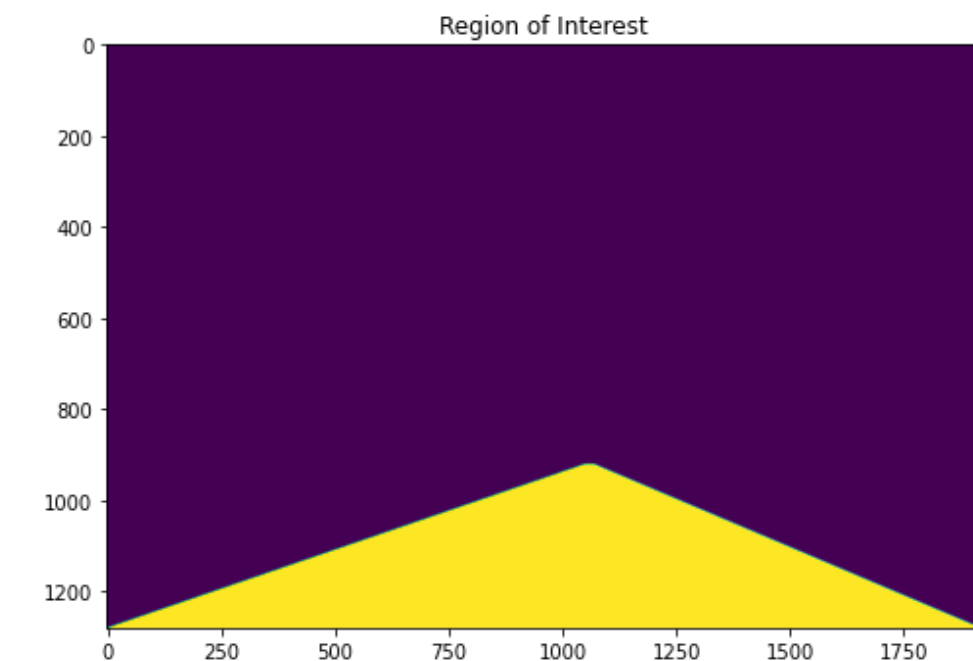
mask = np.zeros_like(edges)
ignore_mask_color = 255

#filling pixels inside the polygon defined by "vertices" with the fill color
cv2.fillPoly(mask, vertices, ignore_mask_color)

#returning the image only where mask pixels are nonzero
masked_image = cv2.bitwise_and(edges, mask)

plt.figure(figsize=(8,6))
plt.title("Region of Interest")
plt.imshow(mask, cmap='Greys_r')

plt.figure(figsize=(8,6))
plt.title("Region of Interest")
plt.imshow(masked_image, cmap='Greys_r')
```



مشاهده میشود تصویر لبه های محدود شده و همچنین ماسک ساخته شده در بالای آن قرار داده شده است.

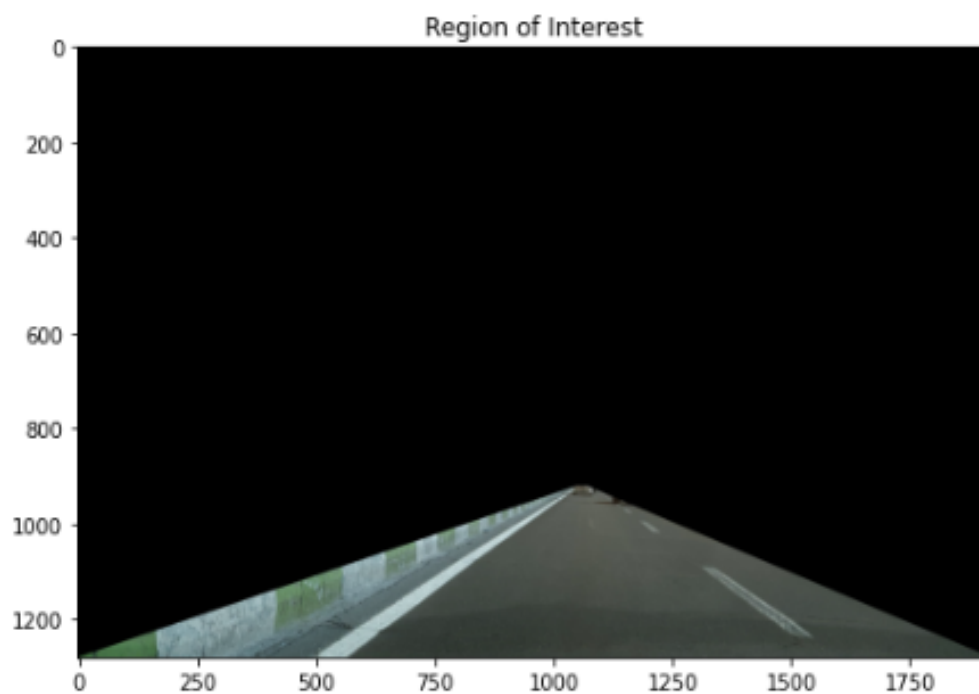
همچنین در ادامه استفاده از ماسک به روی تصویر اصلی را ایجاد کرده ایم :

Mask on Main image

```
[47] colorful_mask = cv2.merge((mask , mask , mask))

main_masked_image = cv2.bitwise_and(image, colorful_mask)
plt.figure(figsize=(8,6))
plt.title("Region of Interest")
plt.imshow(main_masked_image)
```

<matplotlib.image.AxesImage at 0x7f2fb70c1990>



ج. در این سوال لازم است تصویر محدود شده لبه ها را به تابع HoughLinesP بدهیم. ابتدا با استفاده از تصویر ورودی و پارامترهایی که با روش آزمون و خطا پیدا کرده ایم خط ها را پیدا کرده و در متغیر lines میریزیم. این متغیر ارایه ای از خطوط است که هر خانه از این ارایه دارای ۴ پارامتر است که ۲ پارامتر اول نقطه شروع و ۲ پارامتر دوم نشان دهنده نقطه پایان خط ها را نمایش میدهند. نتیجه کار به روی تصویر محدود شده لبه ها و تصویر محدود شده اصلی و تصویر اصلی به شرح زیر است :

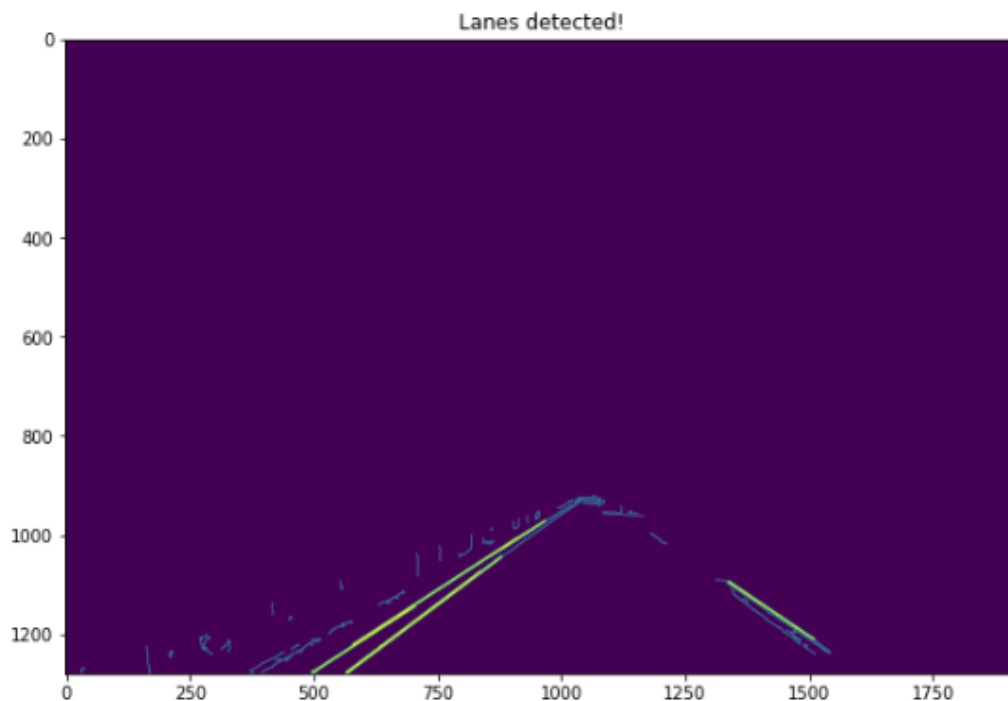
Use HoughLinesP

```
[194] img = masked_image.copy()

lines = cv2.HoughLinesP(masked_image, 1, np.pi/180, 150, minLineLength=1, maxLineGap=50)
# Draw lines on the image
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(img, (x1, y1), (x2, y2), (255, 0, 0), 3)

plt.figure(figsize=(10,7))
plt.title("Lanes detected!")
plt.imshow(img)
```

<matplotlib.image.AxesImage at 0x7f2fb5175490>



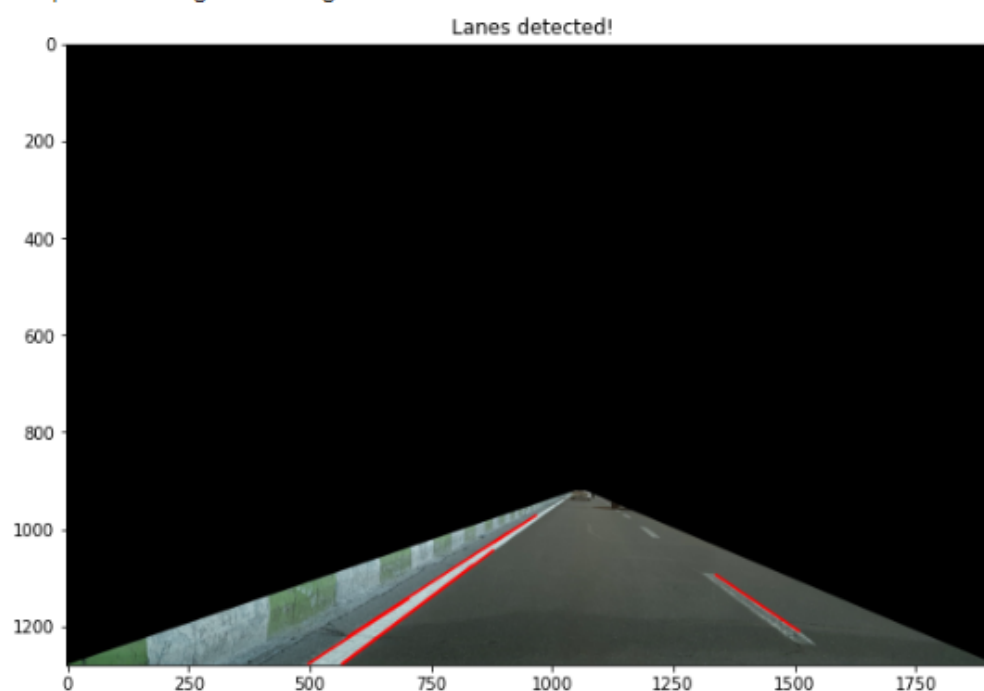
Use HoughLinesP on Main masked image

```
195] img = main_masked_image.copy()

lines = cv2.HoughLinesP(masked_image, 1, np.pi/180, 150, minLineLength=1, maxLineGap=50)
# Draw lines on the image
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(img, (x1, y1), (x2, y2), (255, 0, 0), 3)

plt.figure(figsize=(10,7))
plt.title("Lanes detected!")
plt.imshow(img)
```

<matplotlib.image.AxesImage at 0x7f2fb5167c10>



Use HoughLinesP on Main image

```
[263] img = image.copy()

lines = cv2.HoughLinesP(masked_image, 1, np.pi/180, 150, minLineLength=1, maxLineGap=50)
# Draw lines on the image
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(img, (x1, y1), (x2, y2), (0, 0, 255), 3)

plt.figure(figsize=(10,7))
plt.title("Lanes detected!")
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

<matplotlib.image.AxesImage at 0x7f2fb3e00d10>



د. در ادامه برای بهبود تشخیص خطوط طبق مباحث مطرح شده در صورت سوال ، نیازمند این هستیم که برای هر سمت تصویر یک خط از پایین ترین نقطه خط تا نقطه مشخصی رسم نماییم. برای این کار ابتدا یک تصویر خام به اندازه تصویر اصلی ایجاد میکنیم. خطوط سمت چپ و راست در یک نقطه به یکدیگر میرسند و این نقطه همان نقطه ای است که در قسمت ب مثلث را با آن ساختیم. پس یک حد برای عرض تصویر (محور y) برای تصویر خود ایجاد کنیم که حد بالای آن انتهای تصویر منطبق بر محور x ها خواهد بود برای حد پایین آن با آزمون و خطا و بسته به قسمت ب سوالات مقدار ۹۰۸ را برای نقطه تلاقی ۲ خط انتخاب میکنیم. سپس برای هر خطی که در سوال قبل از طریق HoughLinesP به دست آورده این مقدار شیب را محاسبه میکنیم و به ۲ دسته شیب های مثبت و منفی تقسیم میکنیم. سپس برای هر خط b را از طریق فرمول $b = y_1 - \text{slope} * x_1$ به دست می‌آوریم. سپس از طریق ۳ مقدار slope , b و مقدار حد بالای عرض و حد پایین عرض نقاطی را پیدا میکنیم که درون فرمول $x = (y - b) / \text{slope}$ معنا پیدا کند و تعدادی نقطه پیدا میشود. این نقاط به دست آمده متناظر با نقاط محور x برای هر خط خواهد بود. سپس از این نقاط به دست آمده میانگین میگیریم و از آن به عنوان نقطه شروع و پایان برای پارامتر x خط ها استفاده میشود. سپس این نقاط را متصل کرده و تصاویر مورد نیاز را به دست می آوریم:

Calculate Means

```
[ ] line_img = np.zeros((main_masked_image.shape[0], main_masked_image.shape[1], 3), dtype=np.uint8)

color=[255, 0, 0]
thickness=5

x_bottom_pos = []
x_upper_pos = []
x_bottom_neg = []
x_upper_neg = []

y_bottom = 908
y_upper = 1300

slope = 0
b = 0

for line in lines:
    for x1,y1,x2,y2 in line:
        #test and filter values to slope
        if ((y2-y1)/(x2-x1)) > 0 :

            slope = ((y2-y1)/(x2-x1))
            b = y1 - slope*x1

            x_bottom_pos.append((y_bottom - b)/slope)
            x_upper_pos.append((y_upper - b)/slope)

        elif ((y2-y1)/(x2-x1)) < 0:

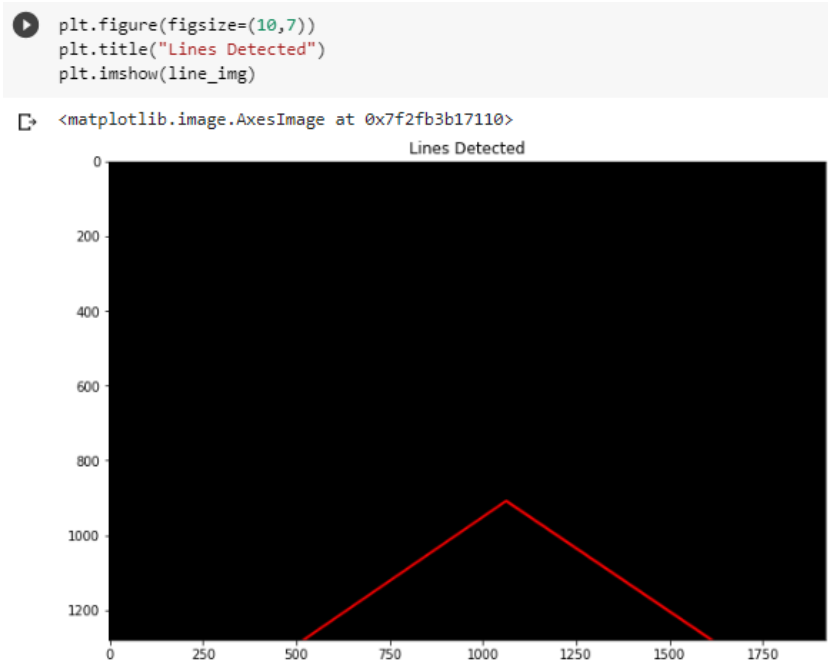
            slope = ((y2-y1)/(x2-x1))
            b = y1 - slope*x1

            x_bottom_neg.append((y_bottom - b)/slope)
            x_upper_neg.append((y_upper - b)/slope)

#creating a new 2d array with means
lines_mean = np.array([[int(np.mean(x_bottom_pos)), int(np.mean(y_bottom)), int(np.mean(x_upper_pos)), int(np.mean(y_upper))],
                       [int(np.mean(x_bottom_neg)), int(np.mean(y_bottom)), int(np.mean(x_upper_neg)), int(np.mean(y_upper))]])

#Drawing the lines
for i in range(len(lines_mean)):
    cv2.line(line_img, (lines_mean[i,0], lines_mean[i,1]), (lines_mean[i,2], lines_mean[i,3]), color, thickness)
```


تصویر به دست آمده از این کد به صورت زیر می باشد:



که اگر این تصویر را به همراه تصویر اصلی به صورت وزن دار ترکیب کنیم به صورت زیر تبدیل خواهد شد :

```
# initial_img * α + img * β + λ
lines_edges = cv2.addWeighted(cv2.cvtColor(image, cv2.COLOR_BGR2RGB), 0.8, line_img, 1., 0)
plt.figure(figsize=(10,7))
plt.title("Lanes detected!")
plt.imshow(lines_edges)
```



تصویر اصلی که پس از پایان این کار ها به دست آمده است به صورت زیر است :

Lanes detected!



۳.

در این سوال با توجه به صورت مسئله نیازمند انجام کارهای بینایی ماشین بر روی ویدیو هستیم. برای اینکار ابتدا ویدیو های vid1 و vid2 را به روی google colab بارگذاری میکنیم. ویدیو اول از حرکت ماشین به روی اتوبان در روز است که کار نسبت ساده تری از ویدیو دوم است که در شب و حالت مه است. برای انجام این پروژه باید تصویر را به فریم های خاصی تبدیل کرده و کار هایی که برای سوال قبل کرده ایم را برای هر فریم انجام دهیم سپس دوباره فریم ها را به یکدیگر چسبانده و ویدیو جدید را ایجاد کنیم.

برای شروع کار ابتدا ویدیو را به پروژه وارد میکنیم. سپس با استفاده از کتابخانه OpenCV پارامتر های fps , height , width , fourcc مربوط به ویدیو را استخراج میکنیم.

```
video_cap = cv2.VideoCapture("vid1.mp4")
```

```
fps = video_cap.get(cv2.CAP_PROP_FPS)
height = int(video_cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
width = int(video_cap.get(cv2.CAP_PROP_FRAME_WIDTH))
FOURCC = cv2.VideoWriter_fourcc(*'MP4V')
```

```
print(fps , height , width , FOURCC)
```

```
25.0 540 960 1446269005
```

در ادامه فریم های ویدیو را از یکدیگر جداسازی کرده و درون آرایه frames ذخیره میکنیم :

```
frames = []
while True:
    success, frame = video_cap.read()
    if success:
        frames.append(frame)
    else:
        break
video_cap.release()
cv2.destroyAllWindows()
```

خروجی video_cap.read() ۲ مقدار است. خروجی اول ، یک مقدار بولین است که نشان دهنده این است که آیا فریمی برای استخراج وجود دارد یا خیر و اگر وجود داشته باشد خروجی دوم فریمی هست که در هر مرحله جداسازی میشود . اگر مقدار آن درست باشد فریم جدا شده را درون آرایه قرار میدهم در غیر این صورت از حلقه خارج شده و حافظه اختصاصی به ویدیو را رها میکنیم.

در مرحله بعد برای باز نویسی فریم ها یک VideoWriter میسازیم تا بتوانیم با استفاده از آن فریم ها را به یکدیگر بچسبانیم. برای پارمتر های ورودی آن ابتدا نام جدید به همراه فرمت فایل را به عنوان ورودی اول میدهم و ورودی های بعدی را با استفاده از پارامتر هایی که در گذشته از ویدیو اصلی استخراج کرده ایم پر میکنیم :

```
out = cv2.VideoWriter('vid1_line_detected.mp4', int(FOURCC), fps, (width,height))
```

در مرحله بعدی نیازمند این هستیم که به روی هر فریم پردازش های مورد نیاز را انجام داده و در انتها آنها را به یکدیگر بچسبانیم تا تشکیل ویدیو اصلی را بدهند. برای این کار از کد زیر استفاده میکنیم. هر فریم وارد تابع process_image شده و در آن پردازش ها صورت میگیرد. در مرحله بعدی با استفاده از تابع write فریم به روی VideoWriter ساخته شده نوشته میشود و در انتها بعد از اینکه تمام فریم ها به روی VideoWriter نوشته شد آن را رها میکنیم تا ویدیو کاملاً ساخته شده و در خروجی قرار گیرد :

```
for image in frames:
    img = process_image(image)
    out.write(img)
    out.release()
```

حال به بررسی تابع process_image میپردازیم. این تابع تشکیل شده از تمام مراحل است که در سوالات قبل آن را پیاده سازی کرده ایم که تنها دارای تغییراتی در پارامترهای ورودی در برخی از توابع است زیرا تصویر جدید با تصویر قبلی متفاوت است . در ابتدا ورودی های تابع را بررسی میکنیم :

```
def process_image(image,
                    kernel_size = 5,
                    low_threshold = 100, high_threshold = 250,
                    rho = 1, theta = np.pi/180, threshold = 30,
                    min_line_len = 100, max_line_gap = 200):
```

در ورودی مقادیر سائز کرنل ، حدود مورد نیاز برای Canny ، مقادیر مورد نیاز برای تبدیل هاف احتمالی را قرار داده ایم. برای به دست آوردن بهترین مقدار برای این مقادیر به طور دستی و با آزمون و خطا به روی چند فریم جدا گانه ورودی ها را تست کرده ایم تا به بهترین مقدار و بهینه ترین مقدار برسیم.

در مرحله بعد چند مرحله از مراحل را بررسی میکنیم :

```
#Change Image to GrayScale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

#Use Gaussian Filter
blur_gray = cv2.GaussianBlur(gray, (kernel_size, kernel_size), 0)

#Use Canny with Image Smoothing
edges = cv2.Canny(blur_gray, low_threshold, high_threshold)
```

ابتدا تصاویر را به صورت سطح خاکستری تبدیل کرده ایم. سپس به روی آنها فیلتر گاسی با کرنل ۵ را وارد کرده ایم سپس با استفاده از لبه یاب Canny لبه ها را یافته ایم.

```
#Find Region of Interest
vertices = np.array([[0,image.shape[0]],[450, 310], (490, 310), (image.shape[1],image.shape[0])], dtype=np.int32)
mask = np.zeros_like(edges)
ignore_mask_color = 255
cv2.fillPoly(mask, vertices, ignore_mask_color)
masked_image = cv2.bitwise_and(edges, mask)

#Use HoughLinesP
img = masked_image.copy()

lines = cv2.HoughLinesP(masked_image, rho, theta, threshold, minLineLength=min_line_len, maxLineGap=max_line_gap)
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(img, (x1, y1), (x2, y2), (0, 0, 255), 3)

#Calculate Means
line_img = np.zeros((masked_image.shape[0], masked_image.shape[1], 3), dtype=np.uint8)
```

سپس با بررسی تصویر و آزمون و خطا مقدار مورد نیاز برای محدود کردن تصویر و تشکیل مثلث با vertices به مقادیر نوشته شده رسیده ایم. سپس ماسک را ساخته و بر روی تصویر لبه ها اعمال کرده ایم. سپس از تصویر ماسک شده کپی گرفته ایم. تبدیل هاف احتمالی را به روی تصویر ماسک شده انجام داده ایم و نتایج خطوط را به روی تصویر کپی شده اعمال کرده ایم. سپس یک تصویر خام از روی تصویر ماسک شده ساخته ایم.

```
color=[0, 0, 255]
thickness=5

x_bottom_pos = []
x_upper_pos = []
x_bottom_neg = []
x_upper_neg = []

y_bottom = 540
y_upper = 315

slope = 0
b = 0

for line in lines:
    for x1,y1,x2,y2 in line:
        #test and filter values to slope
        if ((y2-y1)/(x2-x1)) > 0 :

            slope = ((y2-y1)/(x2-x1))
            b = y1 - slope*x1

            x_bottom_pos.append((y_bottom - b)/slope)
            x_upper_pos.append((y_upper - b)/slope)

        elif ((y2-y1)/(x2-x1)) < 0:

            slope = ((y2-y1)/(x2-x1))
            b = y1 - slope*x1

            x_bottom_neg.append((y_bottom - b)/slope)
            x_upper_neg.append((y_upper - b)/slope)

    if not x_bottom_pos or not x_upper_pos:
        pos_side = past_line_mean[0]
    else :
        pos_side = [int(np.mean(x_bottom_pos)), int(np.mean(y_bottom)), int(np.mean(x_upper_pos)), int(np.mean(y_upper))]
    if not x_bottom_neg or not x_upper_neg:
        neg_side = past_line_mean[1]
    else :
        neg_side = [int(np.mean(x_bottom_neg)), int(np.mean(y_bottom)), int(np.mean(x_upper_neg)), int(np.mean(y_upper))]
    if flag.any() :
        pos_side = [int(np.mean([pos_side[0], past_line_mean[0][0]])), int(np.mean(y_bottom)),
                    int(np.mean([pos_side[2], past_line_mean[0][2]])), int(np.mean(y_upper))]
        neg_side = [int(np.mean([neg_side[0], past_line_mean[1][0]])), int(np.mean(y_bottom)),
                    int(np.mean([neg_side[2], past_line_mean[1][2]])), int(np.mean(y_upper))]
```


سپس در مرحله بعد نیازمند این بوده ایم تا حدودی برای خط هایی که از میانگین خطوط ایجاد میشوند اعمال کنیم که با استفاده از حدودی که در مرحله قبلی یافته بودیم و استفاده از عرض کلی تصویر ، آنها را ایجاد کرده ایم. سپس با بررسی شیب و با استفاده از فرمول های $b = y - \text{slope} * x$ و فرمول $\text{new_x} = (y - b) / \text{slope}$ مقدار x مربوط به هر خط را یافته سپس از این خطوط میانگین گرفته و خطوط میانگین را میابیم .

```
#Drawing the lines
for i in range(len(lines_mean)):
    cv2.line(line_img, (lines_mean[i,0], lines_mean[i,1]), (lines_mean[i,2], lines_mean[i,3]), color, thickness)

# initial_img * α + img * β + λ
lines_edges = cv2.addWeighted(image, 0.8, line_img, 1., 0)

return lines_edges
```

در مرحله بعد خطوط را رسم میکنیم و در انتها خطوط رسم شده را به صورت وزن دار به تصویر اصلی اعمال میکنیم و تصویر نهایی را به عنوان خروجی تابع قرار میدهیم. ویدیو تشکیل شده از این کد با نام vid1_line_detected.mp4 در پیوست قرار داده شده است. در ادامه تصویر چند فریم به عنوان نمونه قرار داده شده است. پس از آن به بررسی تصویر دوم میپردازیم .



برای ویدیو دوم ابتدا تصویر vid2 بار گذاری کرده و پارامتر های مربوط به آن را دریافت کرده سپس فریم های آن را جداسازی کرده و یک VideoWriter میسازیم. سپس آن را وارد تابع process_image وارد میکنیم. برای این ویدیو به علت شرایط آب و هوایی و اینکه تصویر در شب است نیازمند تنظیم پارامتر ها به طور حساس تری هستیم پس در هر مرحله آن را با حساسیت بیشتری پیش میبریم :

```
video_cap = cv2.VideoCapture("vid2.mp4")
```

```
fps = video_cap.get(cv2.CAP_PROP_FPS)
height = int(video_cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
width = int(video_cap.get(cv2.CAP_PROP_FRAME_WIDTH))
FOURCC = cv2.VideoWriter_fourcc(*'MP4V')
```

```
print(fps , height , width , FOURCC)
```

```
30.0 720 1280 1446269005
```

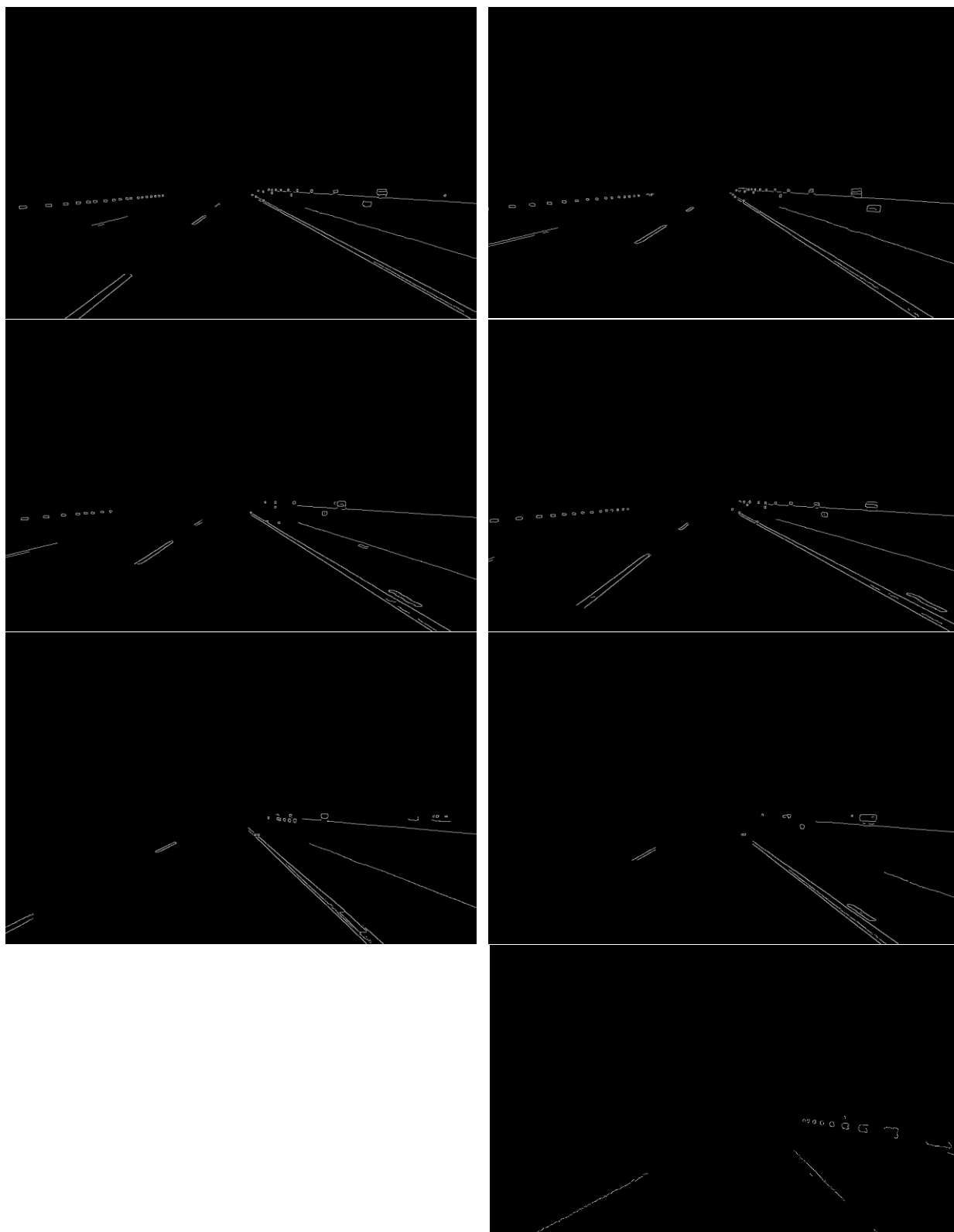
```
frames = []
while True:
    success, frame = video_cap.read()
    if success:
        frames.append(frame)
    else:
        break
video_cap.release()
cv2.destroyAllWindows()
```

```
out = cv2.VideoWriter('vid2_line_detected.mp4', int(FOURCC), fps, (width,height))
```

ابتدا چند فریم مختلف برای بررسی جداسازی کرده و هر مرحله را برای هر کدام به صورت جداگانه ای پیش میبریم تا به پارامتر های خوبی برسیم. این ویدیو از ۳۱۹ فریم تشکیل شده و برای بررسی آن از ۷ فریم استفاده میکنیم. فریم های ۰ ، ۵۳ ، ۱۰۶ ، ۱۵۹ ، ۲۱۲ ، ۲۶۵ و ۳۱۸ .

ابتدا هر کدام را به صورت جدا گانه ای به سطح خاکستری برده و لبه ها را میابیم و بررسی میکنیم که آیا لبه های مناسبی پیدا شده یا خیر . برای لبه یابی به روی محدوده با آزمون و خطا جا به جا شده تا به حدود خوبی برسیم. برای این مسئله به حدود ۲۰ و ۳۰ رسیده ایم.

برای هموار سازی و استفاده از فیلتر گوسی به روی کرنل های مختلف جا به جا شده تا به بهترین کرنل برسیم. از کرنل های مختلف استفاده کرده و به کرنل ۹ میرسیم تا به بهترین حالت تصویر و لبه برسیم.



سپس در مرحله بعد نیازمند این هستیم تا در این مدل یک ناحیه ایجاد کنیم تا در این ناحیه به دور از لبه های خارج ناحیه به پردازش لبه های داخل ناحیه بپردازیم. برای این ناحیه از کد زیر استفاده کرده ایم. برای پیدا کردن این ناحیه با آزمون و خطا مقادیر را پیدا کرده ایم :

```
#Find Region of Interest
vertices = np.array([[(0,image.shape[0]),(725, 415), (745, 415), (image.shape[1],image.shape[0])]], dtype=np.int32)
mask = np.zeros_like(edges)
ignore_mask_color = 255
cv2.fillPoly(mask, vertices, ignore_mask_color)
masked_image = cv2.bitwise_and(edges, mask)
```

سپس با استفاده از تصویر ماسک شده و تبدیل هاف احتمالی خطوط مستقیم را مشخص میکنیم . سپس با استفاده از میانگین گیری و کمک گرفتن از شیب خطوط یک سری خطوط مستقیم را برای هر سمت مشخص کرده و سپس آن ها را با حالت وزن دار به تصویر اصلی اضافه کرده ایم. یکی از نکاتی که در این قسمت اضافه شده است ، استفاده از فریم های قبلی برای کمک به فریم های کنونی است. در این حالت به علت شرایط جوی و نور تصویر در برخی از فریم ها ، از خطوط فریم قبلی استفاده کرده و برای فریم کنونی در سمتی که خطی تشخیص داده نشده استفاده میکنیم. برای استفاده از این حالت توجه میکنیم که تعداد فریم هایی که از این کار استفاده کرده و پشت سر هم هستند تعداد زیادی نباشند که در این حالت دچار خطا شده ایم. برای استفاده از این امکان ، به درون تابع یک ورودی با نام `past_line_mean` اضافه کرده ایم که خطوط میانگین فریم قبلی را به تابع فرستاده و در صورت نیاز از آن استفاده میکنیم . کد مربوط به این بخش به صورت زیر میباشد :

```
y_bottom = 800
y_upper = 430

slope = 0
b = 0

for line in lines:
    for x1,y1,x2,y2 in line:
        #test and filter values to slope
        if ((y2-y1)/(x2-x1)) > 0 :

            slope = ((y2-y1)/(x2-x1))
            b = y1 - slope*x1

            x_bottom_pos.append((y_bottom - b)/slope)
            x_upper_pos.append((y_upper - b)/slope)

        elif ((y2-y1)/(x2-x1)) < 0:

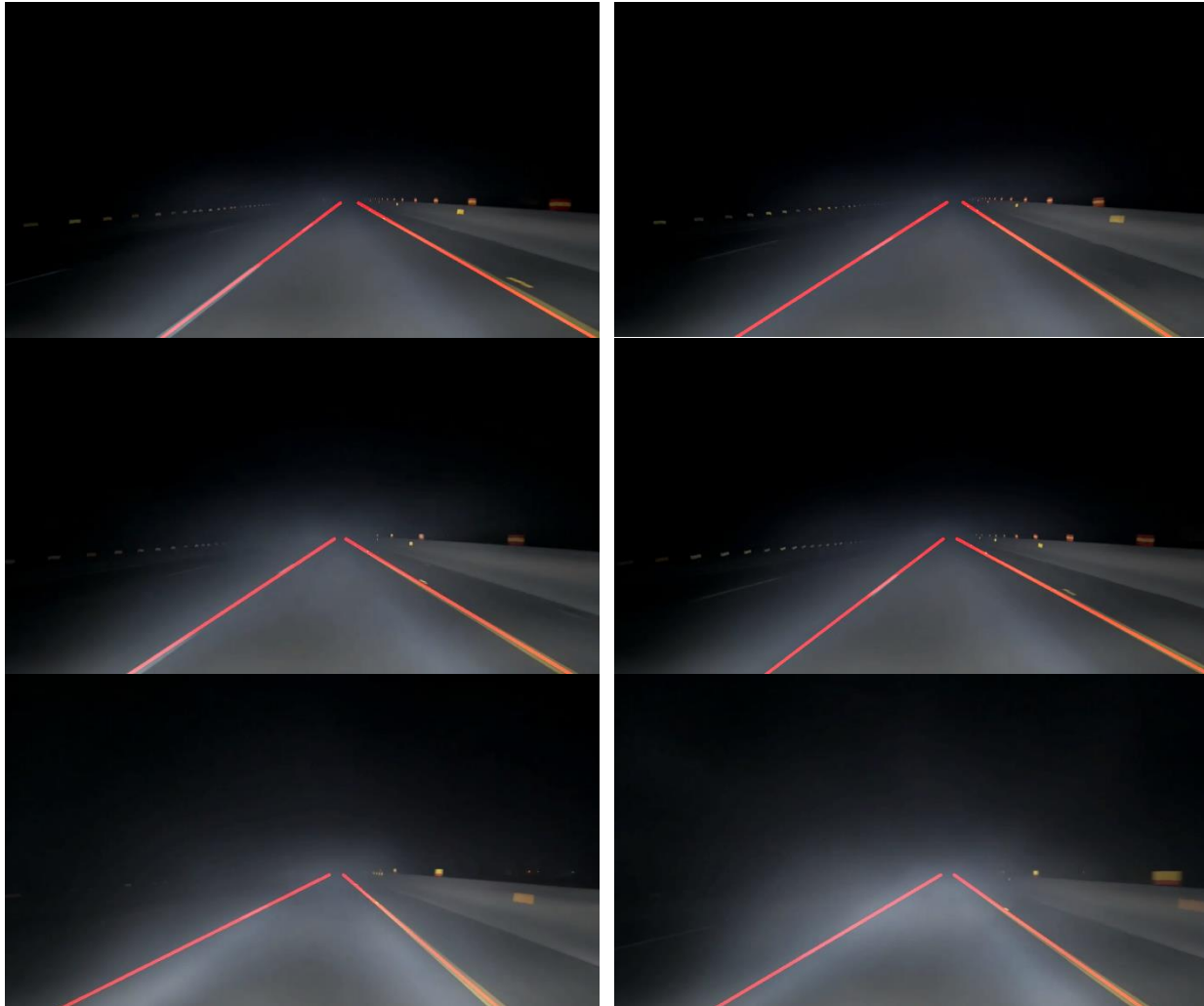
            slope = ((y2-y1)/(x2-x1))
            b = y1 - slope*x1

            x_bottom_neg.append((y_bottom - b)/slope)
            x_upper_neg.append((y_upper - b)/slope)

    if not x_bottom_pos or not x_upper_pos:
        pos_side = past_line_mean[0]
    else :
        pos_side = [int(np.mean(x_bottom_pos)), int(np.mean(y_bottom)), int(np.mean(x_upper_pos)), int(np.mean(y_upper))]

    if not x_bottom_neg or not x_upper_neg:
        neg_side = past_line_mean[1]
    else :
        neg_side = [int(np.mean(x_bottom_neg)), int(np.mean(y_bottom)), int(np.mean(x_upper_neg)), int(np.mean(y_upper))]
    #creating a new 2d array with means
    lines_mean = np.array([pos_side , neg_side])
```

تعدادی از خطوط رسم شده در این سوال در ادامه قرار داده شده است :



در این سوال برای رفع نویز ها و پیوستگی خطوط همچنین برای کاهش پرش خطوط بین فریم ها ، درون تابع محاسبه میانگین خطوط در هر فریم ، از فریم ها گذشته نیز کمک گرفته و میانگین ۲ فریم پشت سر هم را محاسبه کرده و به جای فریم کنونی استفاده میکنیم. در این حالت پرش ها کاهش یافته و خطوط پیوستگی بهتری را خواهد داشت. کد محاسبه میانگین فریم ها به صورت زیر میباشد :

```
if flag.any() :
    pos_side = [int(np.mean([pos_side[0] , past_line_mean[0][0]])) , int(np.mean(y_bottom))
    , int(np.mean([pos_side[2] , past_line_mean[0][2]])) , int(np.mean(y_upper))]]
    neg_side = [int(np.mean([neg_side[0] , past_line_mean[1][0]])) , int(np.mean(y_bottom))
    , int(np.mean([neg_side[2] , past_line_mean[1][2]])) , int(np.mean(y_upper))]]
```

کدهای مربوط به قسمت اول سوال درون فایل Q3_vid1 و فایل ویدیو به نام vid1_line_detected و قسمت دوم سوال درون فایل Q3_vid2 و فایل ویدیو به نام vid1_line_detected قرار داده شده است.

۴.

کانتور های فعال با تطبیق مارها (snakes) با ویژگی های تصاویر کار میکنند. این تابع از تصاویر دو بعدی تک و چند کاناله پشتیبانی می کند. مارها می توانند دوره ای یا دارای انتهای ثابت و/یا آزاد باشند. طول مار خروجی برابر با مرز ورودی است. از آنجایی که تعداد نقاط ثابت است، مطمئن شوید که مار اولیه دارای نقاط کافی برای ثبت جزئیات کانتور نهایی است.

$$E_{Total} = E_{image} + E_{Contour}$$

$$E_{Contour} = \alpha * E_{Elastic} + \beta * E_{Smooth}$$

پارامتر های این تابع عبارتند از :

۱. image : این پارامتر تصویر ورودی است.

۲. snake : این ورودی همان کانتور یا مار اولیه است که به صورت دستی یا از پیش تعیین میشود. برای شرایط مرزی دوره ای، نقاط پایانی نباید تکرار شوند.

۳. alpha : این ورودی همان متغیر مضروب در فرمول انرژی کانتور است که افزایش آن باعث انعطاف پذیری بیشتر کانتور میشود. مقادیر بالاتر باعث می شود مار سریعتر منقبض شود.

۴. beta : این ورودی همان متغیر مضروب در فرمول انرژی کانتور است که افزایش آن باعث صافی بیشتر کانتور میشود. مقادیر بالاتر مار را صاف تر می کند.

۵. w_line : این ورودی باعث جذب بیشتر کانتور به مناطق با روشنایی بیشتر میشود. از مقادیر منفی برای جذب به سمت مناطق تاریک استفاده میشود.

۶. w_edge : این ورودی جذب کانتور به لبه ها را کنترل می کند. از مقادیر منفی برای دفع مار از لبه ها استفاده میشود.

۷. gamma : پارامتر گام. این پارامتر برای طول گام در هر تکرار استفاده میشود. اگر مقادیر مثبت استفاده شود در این حالت نقاط به سمت داخل جمع شده و نیروی به آنها به سمت داخل می باشد. اما اگر مثبت باشد نیرو از داخل به سمت خارج بوده و به مانند باد شدن بالن می باشد.

۸. max_px_move : حداکثر فاصله پیکسل برای حرکت در هر تکرار.

۹. max_num_iter : حداکثر تکرار برای بهینه سازی شکل مار.

۱۰. convergence : معیارهای همگرایی. به معنی این است که اگر مقدار انرژی به کمتر از این حد رسیده باشد دیگر اعمال الگوریتم نیازی نیست.

۱۱. boundary_condition : شرایط مرزی برای کانتور. می تواند یکی از «ادواری»، «آزاد»، «ثابت»، «آزاد ثابت» باشد. "دوره ای" دو انتهای مار را متصل می کند، "ثابت" نقاط انتهایی را در جای خود نگه می دارد، و "آزاد" اجازه حرکت آزاد انتهای مار را می دهد. "ثابت" و "آزاد" را می توان با تجزیه "free-fixed, fixed-free" ترکیب کرد. تجزیه «ثابت-ثابت» یا «آزاد» به ترتیب رفتاری مشابه «ثابت» و «آزاد» دارد.

۱۲. coordinates : این گزینه فقط برای سازگاری باقی می ماند و هیچ تاثیری ندارد. در ورژن ۰/۱۶ با گزینه 'xy' معرفی شد، اما از ۰/۱۸، فقط گزینه 'rc' معتبر است. مختصات باید در قالب ردیف-ستون تنظیم شوند.

برای مثال استفاده از آن نیز، یک تصویر سکه را بارگذاری کرده و تشخیص شکل بر روی آن انجام شده است.



© Münzkabinett der Staatlichen Museen zu Berlin



© Münzkabinett der Staatlichen Museen zu Berlin

۵.

تصویر img2 :

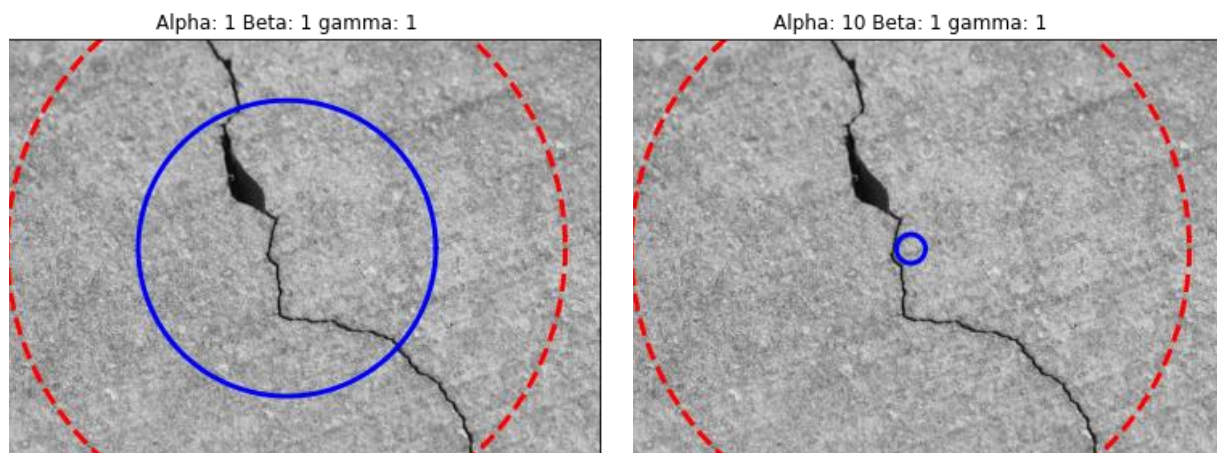
ابتدا تصویر را در googleColab بارگذاری میکنیم. سپس آن را برای بهبود عملکرد به سطح خاکستری میبریم .
در مرحله بعد با آزمون و خطا یک محدوده دستی برای مقدار اولیه مار ایجاد میکنیم :

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.segmentation import active_contour
import cv2
from google.colab.patches import cv2_imshow
```

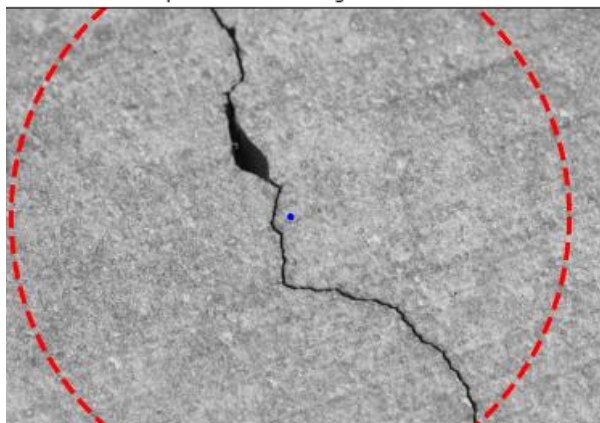
```
image = cv2.imread("img2.jpg")
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
s = np.linspace(0, 2*np.pi, 300)
r = (gray_image.shape[0]/2) + 300*np.sin(s)
c = (gray_image.shape[1]/2) + 300*np.cos(s)
init = np.array([r, c]).T
```

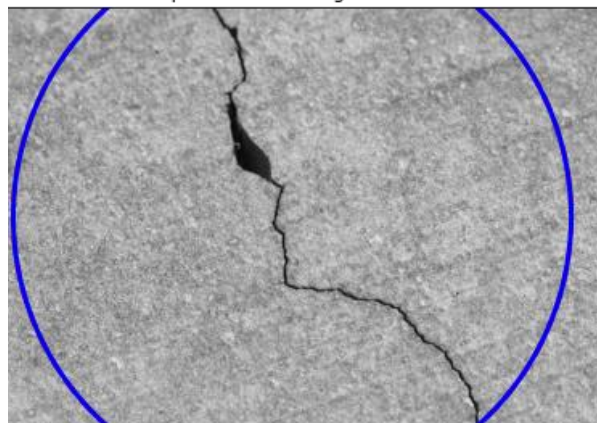
سپس در مرحله بعد با روش آزمون و خطا برای محاسبه مقدار صحیح تلاش میکنیم. مقادیر به دست آمده در این مرحله به صورت زیر میباشد :



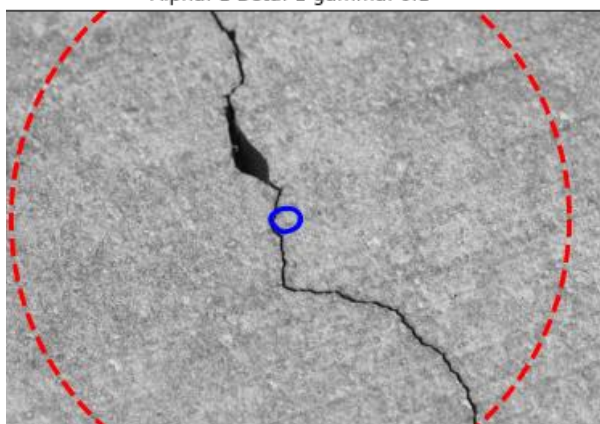
Alpha: 100 Beta: 1 gamma: 1



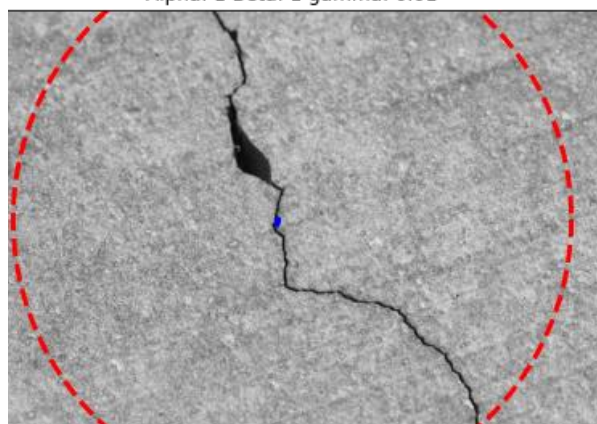
Alpha: 0.1 Beta: 1 gamma: 1



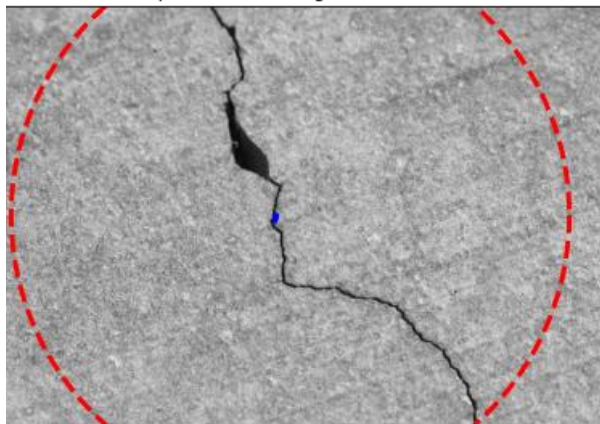
Alpha: 1 Beta: 1 gamma: 0.1



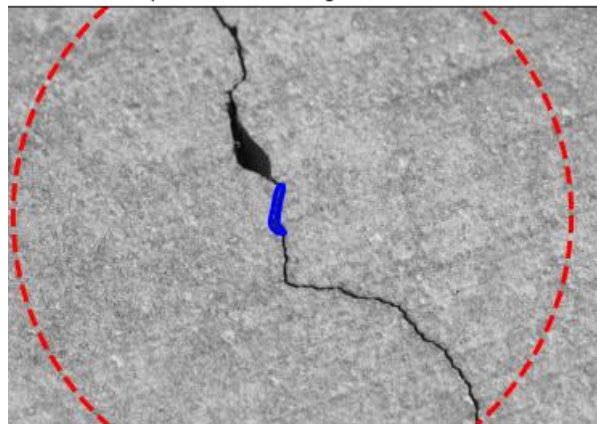
Alpha: 1 Beta: 1 gamma: 0.01



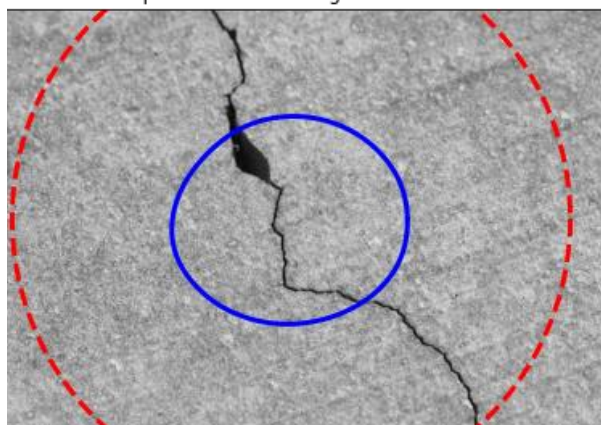
Alpha: 1 Beta: 10 gamma: 0.01



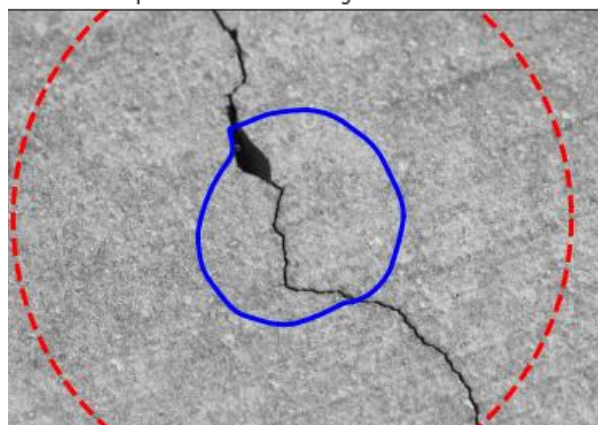
Alpha: 0.1 Beta: 10 gamma: 0.01



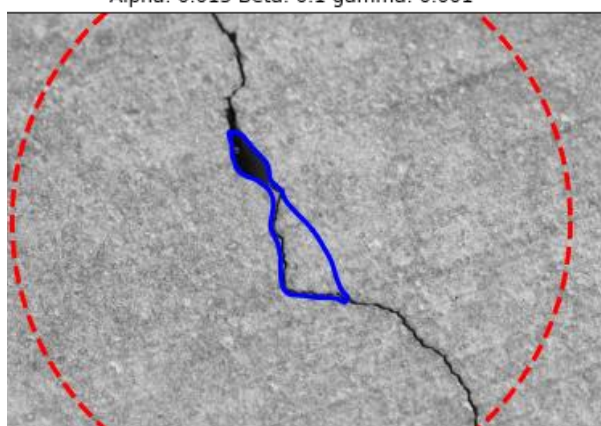
Alpha: 0.01 Beta: 10 gamma: 0.01



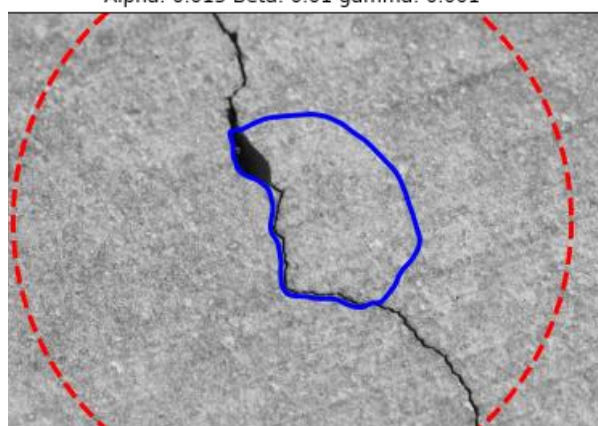
Alpha: 0.015 Beta: 0.1 gamma: 0.01



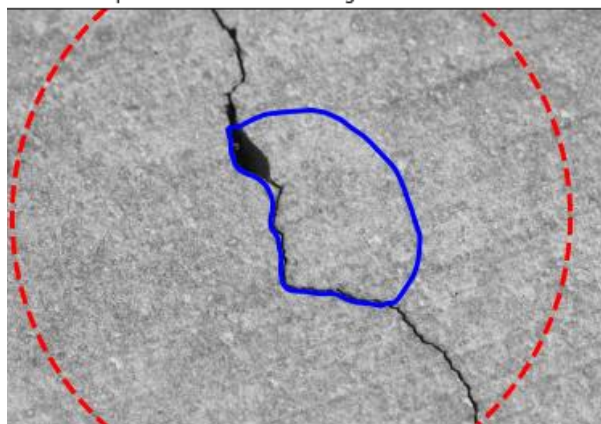
Alpha: 0.015 Beta: 0.1 gamma: 0.001



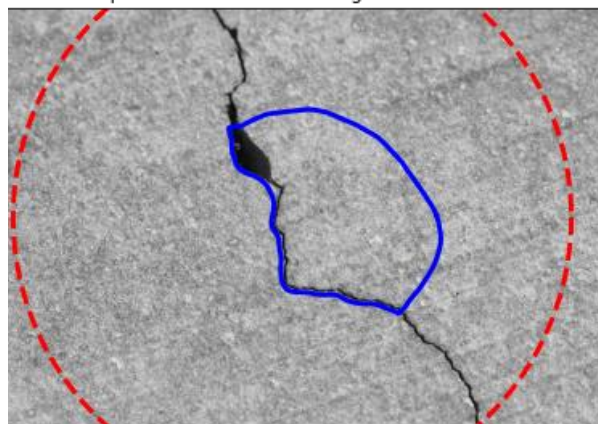
Alpha: 0.015 Beta: 0.01 gamma: 0.001

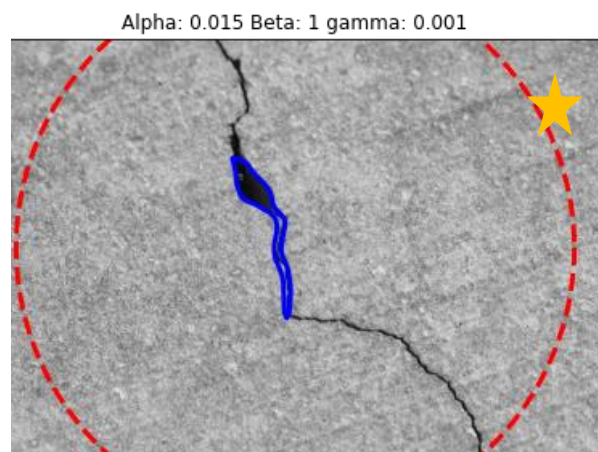
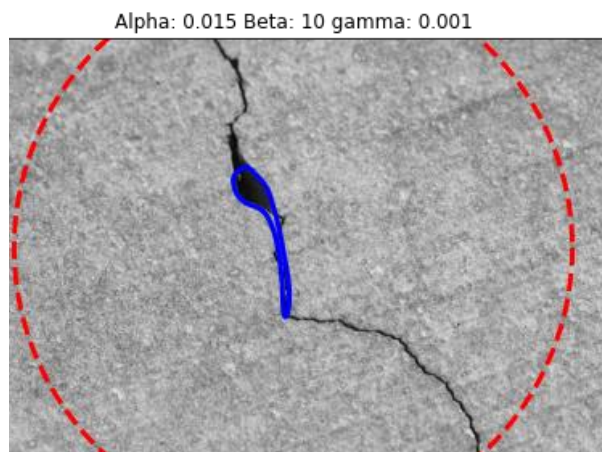


Alpha: 0.015 Beta: 0.008 gamma: 0.001



Alpha: 0.0115 Beta: 0.008 gamma: 0.001

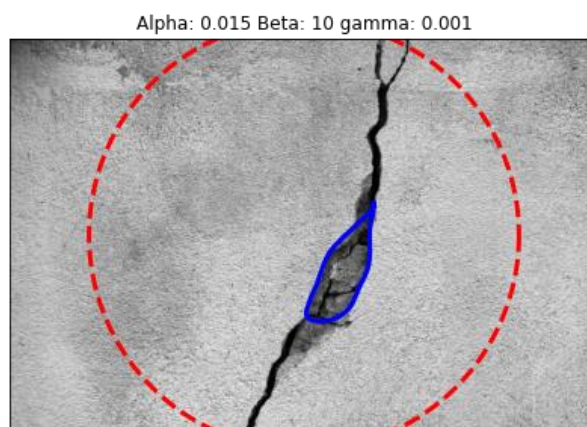
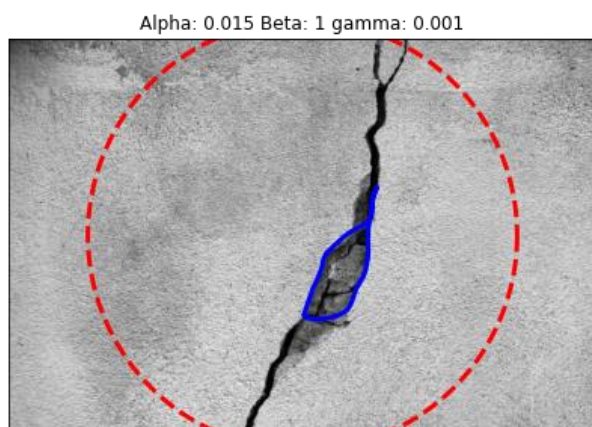




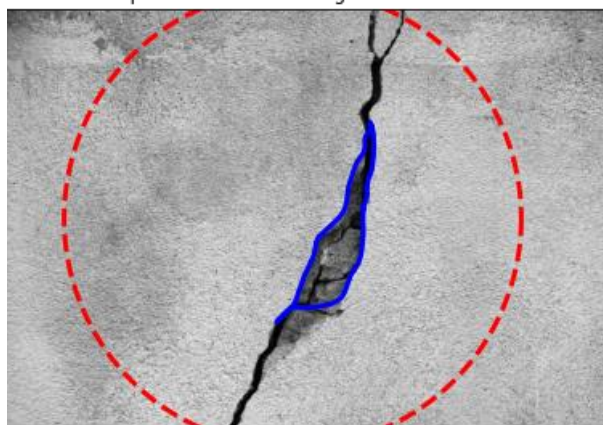
از لحاظ بهترین حالتی که میتوانیم به آن برسیم به مقدار $\alpha = 0.015$ و $\beta = 1$ و $\gamma = 0.001$ است. به روی بهترین حالت پیدا شده یک ستاره قرار داده شده است.

تصویر img3 :

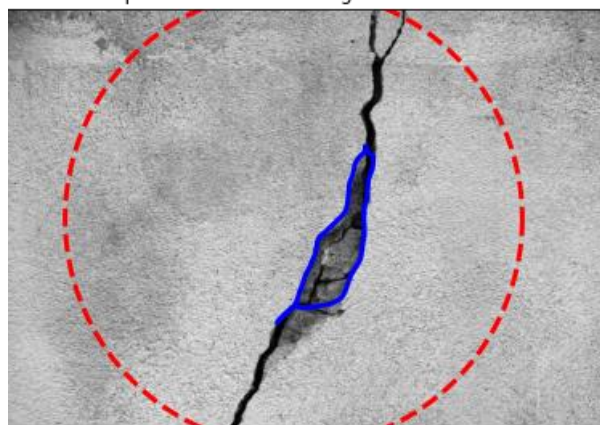
همانند قسمت اول سوال ، تصویر را بارگذاری کرده ، به سطح خاکستری میبریم سپس یک مار اولیه ایجاد میکنیم و داده ها را وارد تابع `active_contour` مربوط به کتابخانه `scikit` میکنیم. نتایج به دست آمده به شرح زیر میباشد :



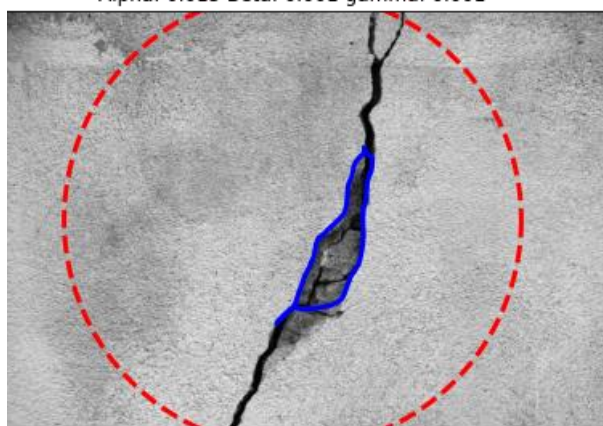
Alpha: 0.015 Beta: 0.1 gamma: 0.001



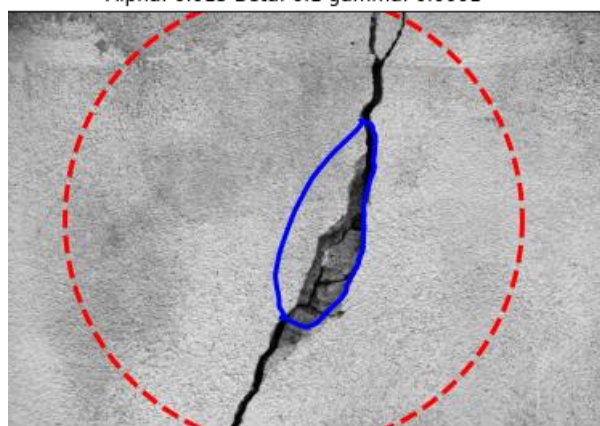
Alpha: 0.015 Beta: 0.01 gamma: 0.001



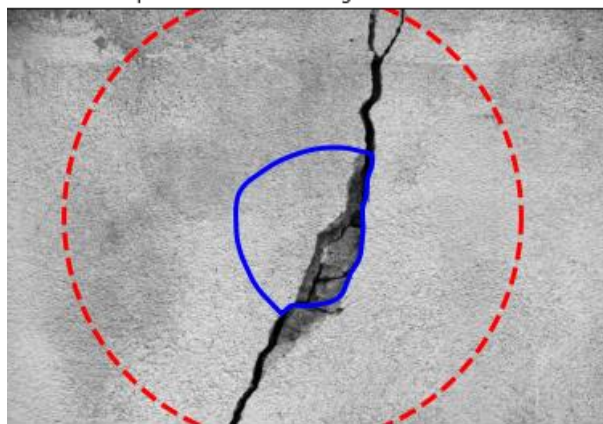
Alpha: 0.015 Beta: 0.001 gamma: 0.001



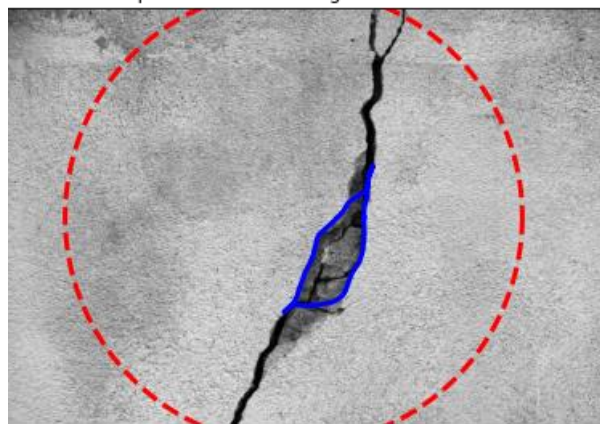
Alpha: 0.015 Beta: 0.1 gamma: 0.0001

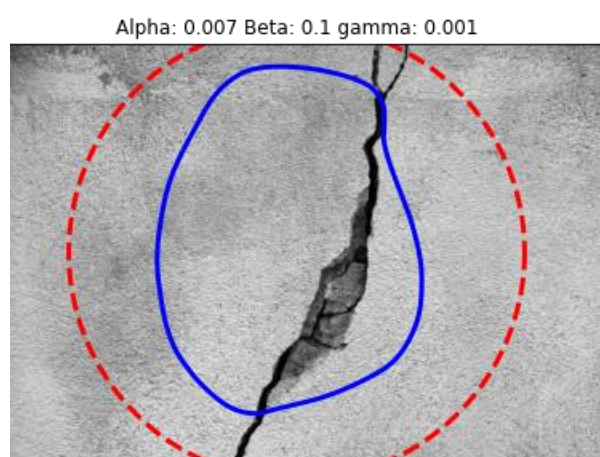
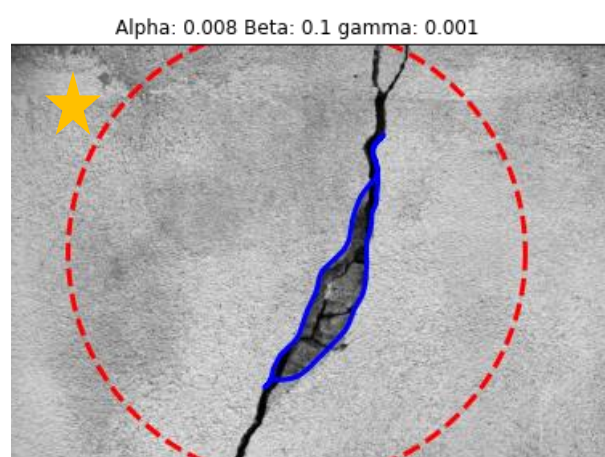
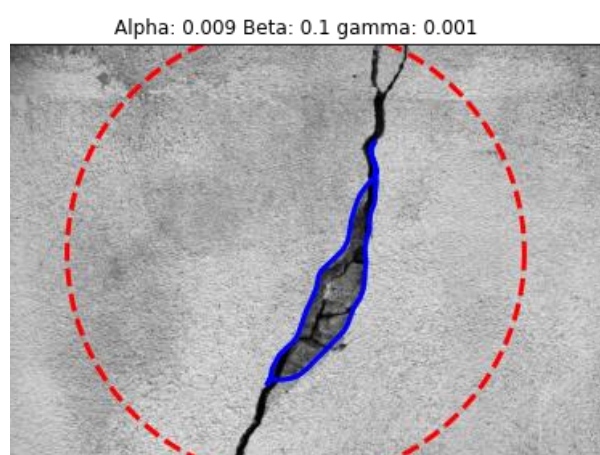
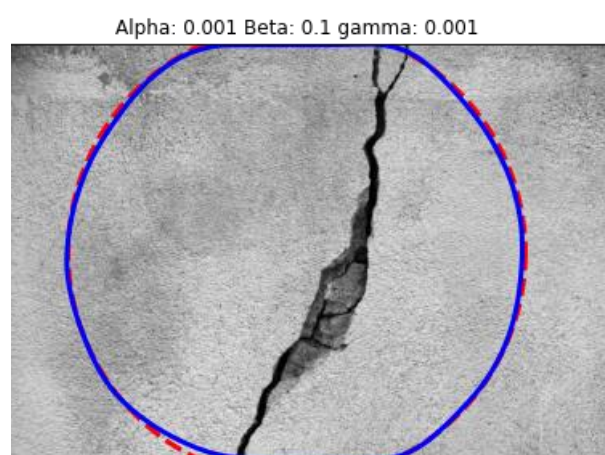
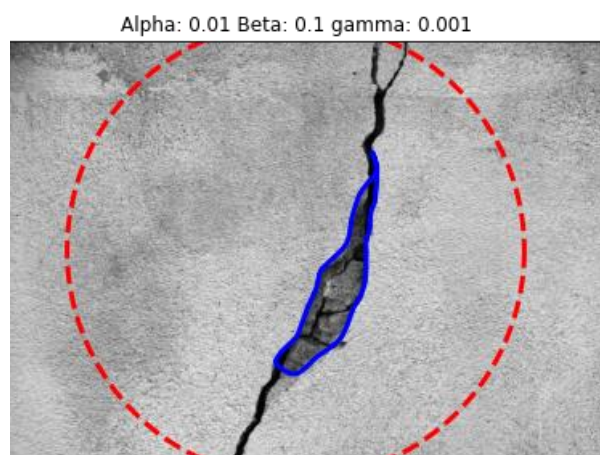
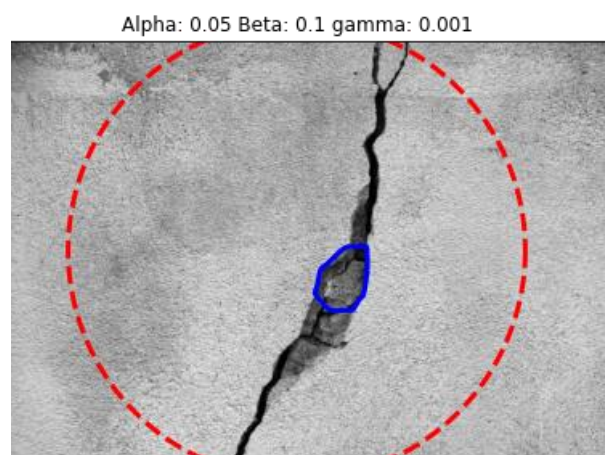


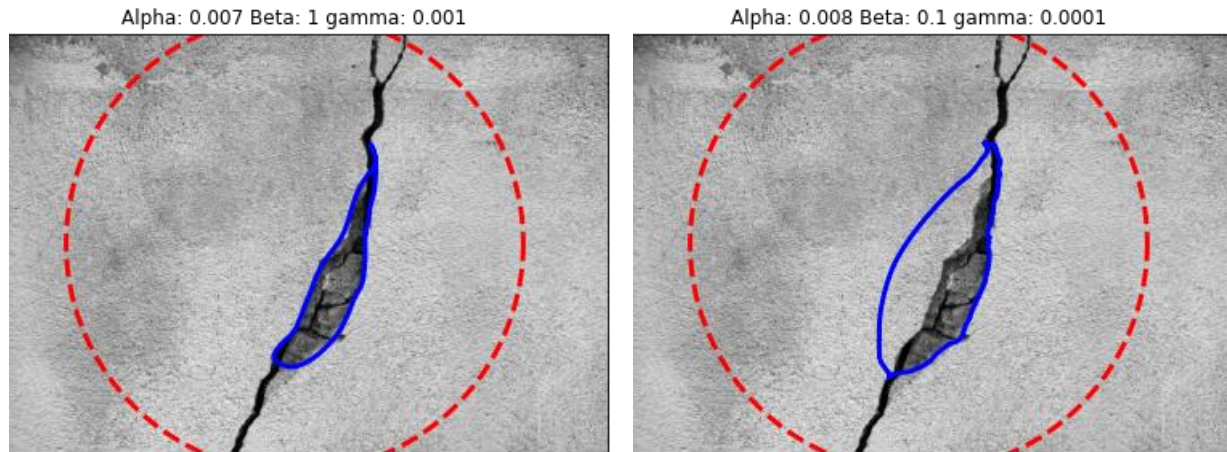
Alpha: 0.015 Beta: 0.1 gamma: 0.01



Alpha: 0.02 Beta: 0.1 gamma: 0.001







با توجه به اطلاعات به دست آمده ، مقدار $\alpha = 0.008$ و $\beta = 0.1$ و $\gamma = 0.001$ بهترین مقدار برای یافتن شکل شکاف بوده است.

به روی بهترین حالت پیدا شده یک ستاره قرار داده شده است.

ایده اصلی مدل کانتور فعال بر روی مسئله تقسیم‌بندی تصویر به یک منحنی بسته در مورد مشکلات کمینه‌سازی عملکردی تبدیل شده است. مدل کانتور فعال مبتنی بر اطلاعات لبه از اطلاعات گرادیان بهره می‌برد و دارای کاستی‌هایی از جمله نمی‌تواند مرز ضعیف، مرز فازی و شی مرز ناپیوسته را جدا کند. مدل کانتور فعال Chan-Vese بدون لبه می‌تواند بر کاستی‌های مدل بر اساس گرادیان غلبه کند. در کتابخانه scikit و تابع activecontour آن از نوع کانتور فعال مبتنی بر اطلاعات گرادیان استفاده میکند. در مدل اطلاعات گرادیان به علت نازک بودن شکل در محاسبه مقدار و جهت گرادیان دچار مشکل شده و نمیتواند آن را به درستی محاسبه کند. هنگامی که شکل مانند ترک باریک و دراز بوده این مشکل شدت میابد زیرا حالت باریک به مقدار زیادی طولانی شده و محاسبه و تخمین گرادیان از همسایه ها نیز دشوار شده و نمیتواند گرادیان را به درستی تشخیص دهد.