BİL 467/561 – Image Processing

Summer 2022

Final Project Report

Your Name and Surname:	Harun Serkan Metin
Your Student ID:	201101034
Project Title:	Lane Detection from Stitched Videos
Project Personnel:	-

1. Proje tanımı ve Önbilgi

• Bu projede, arabanın kaput kısmından 3 farklı açıyla kaydedilmiş videoları birleştirerek daha geniş bir görüş elde etmeyi ve bu görüntüden yol şeritlerini tespit etmeyi amaçladım.

• Yol şeritlerini tespit etme sırasında herhangi bir derin öğrenme veya nesne tespit yöntemi kullanmadım. Bu yüzden aldığım çıktılar bazen istenen sonuçtan uzak olabiliyor.

• Soldan, ortadan ve sağdan kaydedilmiş videoları "Stitching" işlemi yaparak birleştiriyorum. Bazı durumlarda şeritler tam olarak algılanabiliyor mesela çok eğimli virajlarda. Bu gibi durumda görüntüyü maskelemek zorlaştığından şerit tespiti için yazdığım kod iyi çalışmayabiliyor.

• Her video için KEY POINT (Anahtar nokta) bulmak ve bunları eşleştirmek, "Stitching" için kullanılan bir yöntemdir. Anahtar nokta tespiti için en popüler olan SIFT algoritmasını kullandım. Bu noktaları eşleştirmek için iki farklı fonksiyonu koda implement ettim. İşleyişi uzun zaman alan Brute Force (BF) Metodu ve daha optimal olan K. Nearest Neighbor (KNN) Metodu, kod içinde mod parametresi değiştirilerek kullanılabilir.

Şerit algılama kısmında, doğru alanı ve bu alandaki doğru şeritleri bulmak için Canny Edge Detection yöntemini ve resmi maskeleme yöntemini kullandım. Maskelemeden sonra beyazla gösterilen kenarları renklendirerek birden fazla çizgi elde ettim. Bu çizgileri iki sınıfta (sol çizgiler, sağ çizgiler) gruplandırdım. Gruplamadan sonra sol çizgileri tek bir çizgide, sağ çizgileri tek bir çizgide birleştirerek 2 şeridi de tespit ettim. Bu şeritlerin arasındaki alanı da Yeşil ile renklendirerek yolun ayırt edilmesini kolaylaştırdım.

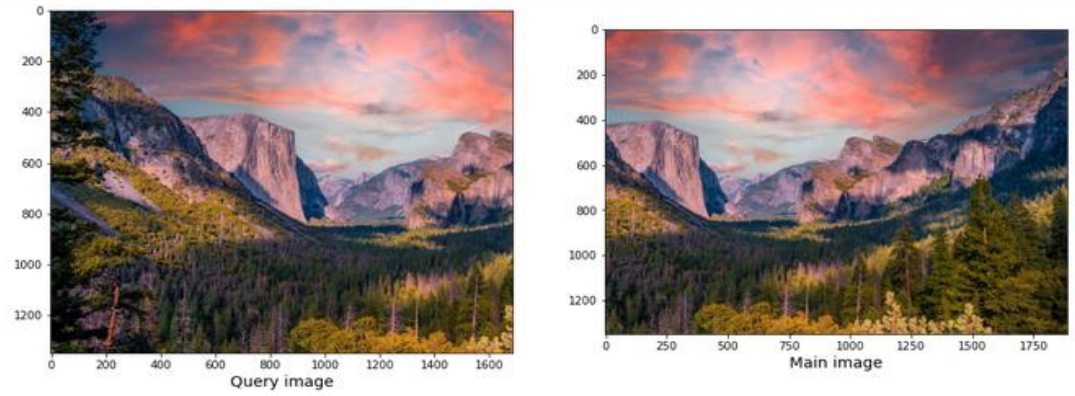
2. Tekniksel Çalışma

2.1 Image Stitching:

Benzer içeriğe sahip farklı resimlerin ortak noktalarını üst üste getirerek eşleşme sağlayacağım (Matching Key Points). Bu eşleme ve birleştirme işlemleri aşağıda gösterdiğim şekilde gerçekleşmektedir.

- Ortak “Key Pointleri” Match edip panorama bir görüntü elde edeceğim resimler.

```
In [65]: main_photo = cv2.cvtColor( cv2.imread('main.jpg'),cv2.COLOR_BGR2RGB)
         main_photo_gray = cv2.cvtColor(main_photo, cv2.COLOR_RGB2GRAY)

         query_photo = cv2.cvtColor(cv2.imread('query.jpg'),cv2.COLOR_BGR2RGB)
         query_photo_gray = cv2.cvtColor(query_photo, cv2.COLOR_RGB2GRAY)

         # Now view/plot the images
         fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, constrained_layout=False, figsize=(16,9))
         ax1.imshow(query_photo, cmap="gray")
         ax1.set_xlabel("Query image", fontsize=14)

         ax2.imshow(main_photo, cmap="gray")
         ax2.set_xlabel("Main image", fontsize=14)
         plt.show()
```



Resimlerin ortak noktalarını bulmak için KeyPointlerini belirlemek gerekiyor

- Bahsettiğim üzere “Key Pointleri” bulmak ile başladım. Bunun için bir çok method var fakat ben bu projede SIFT methodunu kullandım.
- SIFT şöyle çalışmaktadır:
    - Uzaydaki uç noktaların (minimum-maksimum) elde edilmesi.
    - “Key Pointlerin” konumlarının belirlenmesi.
    - Döngüsel değişime karşı dayanıklılık kazanılması.
    - “Key Pointlerin” tanımlayıcıların bulunması.

In [66]:
```python
def find_key_points(main_photo_gray,query_photo_gray,printMode):
        descriptor = cv2.SIFT_create()
        keypoints_main_img, features_main_img = descriptor.detectAndCompute(main_photo_gray, None)
        keypoints_query_img, features_query_img = descriptor.detectAndCompute(query_photo_gray, None)
        if(printMode):
                print("Query Image Number of KeyPoints",len(keypoints_query_img))
                print("Query Image Shape of Features:",features_query_img.shape)

                for keypoint in keypoints_query_img:
                        x,y = keypoint.pt
                        size = keypoint.size
                        orientation = keypoint.angle
                        response = keypoint.response
                        octave = keypoint.octave
                        class_id = keypoint.class_id
                #Last Key Point Features
                print("X:",x," Y:",y)
                print("Size",size)
                print("Angle:",orientation)
                print("response:",response)
                print("octave:",octave)
                print("class_id:",class_id)
        return (keypoints_main_img, features_main_img ,keypoints_query_img, features_query_img)
```

SIFT Methodu ile resimdeki Key Pointleri belirliyoruz (Feature Extraction)

In [67]:
```python
def matching_keys_BF(features_main_img, features_query_img,printMode):
        bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
        best_matches = bf.match(features_main_img,features_query_img)
        rawMatches = sorted(best_matches, key = lambda x:x.distance)
        if(printMode):
                print("Raw matches (Brute force):", len(rawMatches))
        return rawMatches
```

matching_keys_BF() : Main imagede bulunan tüm Key Pointleri, Query imagede bulunan tüm Key Pointlerle karşılaştırarak en yakın olanı bulmaya çalışıyor.

- Vektörlerin birbirinden uzaklığını hesaplamak için EUCLID Algoritmasını kullanıyor

In [68]:
```python
def matching_keys_KNN(features_main_img, features_query_img, ratio,printMode):
        bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)
        rawMatches = bf.knnMatch(features_main_img, features_query_img, k=2)
        if(printMode):
                print("Raw matches (KNN):", len(rawMatches))
        matches = []

        for m,n in rawMatches:
            if m.distance < n.distance * ratio:
                matches.append(m)
        return matches
```

matching_keys_KNN() : Main imagede bulunan Key Pointleri, Query imagede bulunan Key Pointlerle , K. Nearest Neighboor algoritmasını kullanarak karşılaştırarak optimal olanı bulmaya çalışıyor.

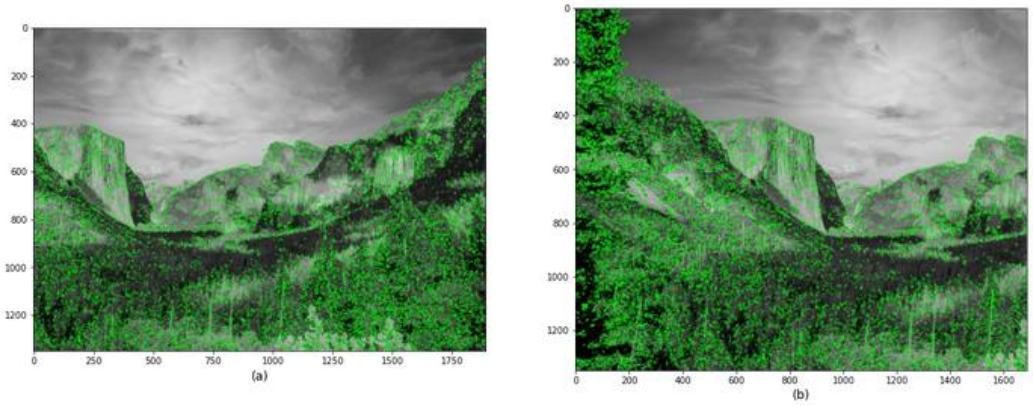
- "K=2" verdiğimiz için iki tane sonuç dönecek. Bu iki sonucu belli bir oranla çarparak bir eşik değeri belirliyoruz.
- Bu eşik değerinin üstündekinoktayı en iyi eşleşme seçiyoruz ve matches listesine ekliyoruz.

Yukardaki görselde belirtilmiş “find_key_points()” fonksiyonu SIFT methodunu kullanarak her iki resim için “Key Pointlerini ve Özelliklerini” tuple olarak return ediyor.

Bu noktalardan birbirine benzer olanları bulup, bunları eşleme işlemine Matching denir. Matching için birden fazla yöntem bulunmakta fakat ben bu projede Brute Force (BF) ve K. Nearest Neighbor (KNN) yöntemlerine yer verdim. Kodu çalıştırırken hangi eşleme methodunu kullanacağınıza karar verebilirsiniz.

```
In [69]: (keypoints_main_img, features_main_img ,keypoints_query_img, features_query_img)=find_key_points(main_photo_gray,query_photo_gray,1)
         # display the keypoints and features detected on both images

         fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(20,8), constrained_layout=False)

         ax1.imshow(cv2.drawKeypoints(main_photo_gray, keypoints_main_img, None, color=(0,255,0)))

         ax1.set_xlabel("(a)", fontsize=14)

         ax2.imshow(cv2.drawKeypoints(query_photo_gray,keypoints_query_img,None,color=(0,255,0)))
         ax2.set_xlabel("(b)", fontsize=14)

         plt.show()
```

```
Query Image Number of KeyPoints 18633
Query Image Shape of Features: (18633, 128)
X: 1684.06298828125  Y: 831.1734619140625
Size 1.998838054733276
Angle: 18.757591247558594
response: 0.027936942875385284
octave: 7733759
class_id: -1
```



Yukardaki görselde yeşil olarak renklendirilmiş noktalar bu resimlerin SIFT ile bulunmuş Key Pointleridir. Yukardaki kodun çıktısında herhangi bir Key Pointin özellikleri belirtilmiştir.

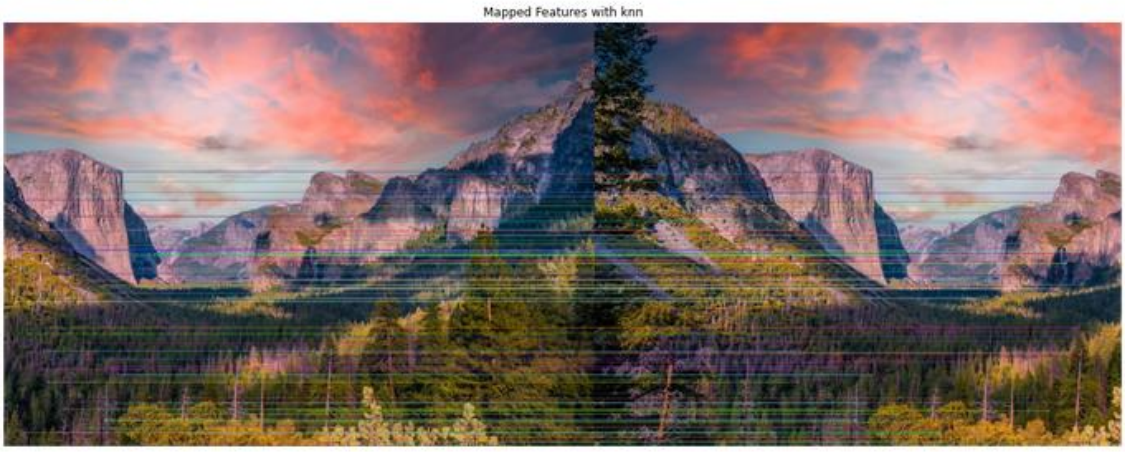
```
In [70]: def match_features(main_photo,keypoints_main_img,features_main_img,query_photo,keypoints_query_img,features_query_img,mode,printMode):
             if(printMode):
                 print(mode,"matched features Lines")

             fig = plt.figure(figsize=(20,8))

             if mode == 'bf':
                 matches = matching_keys_BF(features_main_img, features_query_img,printMode=printMode)
                 mapped_features_image = cv2.drawMatches(main_photo,keypoints_main_img,query_photo,keypoints_query_img,matches[:100],None,f

             elif mode == 'knn':
                 matches = matching_keys_KNN(features_main_img, features_query_img, ratio=0.75,printMode=printMode)
                 mapped_features_image = cv2.drawMatches(main_photo, keypoints_main_img, query_photo, keypoints_query_img, np.random.choice

             if(printMode):
                 plt.title("Mapped Features with {}".format(mode))
                 plt.imshow(mapped_features_image)
                 plt.axis('off')
             return matches
```

İki resim arasındaki eşleşmeleri çizgilerle ifade ettim.

- Eğer birleştirlecek olan resimler *transformasyon* görmemişse (mesela yan çevirmek) bu çizgiler genellikle paraleldir.
- Paralelliği bozan çizgiler yüksek ihtimal *Yanlış Eşleşmelerdir.*

In [71]:
```
feature_to_match='knn'
matches=match_features(main_photo,keypoints_main_img,features_main_img,query_photo,keypoints_query_img,features_query_img,mode=feature_to_
```

```
knn matched features Lines
Raw matches (KNN): 18457
```



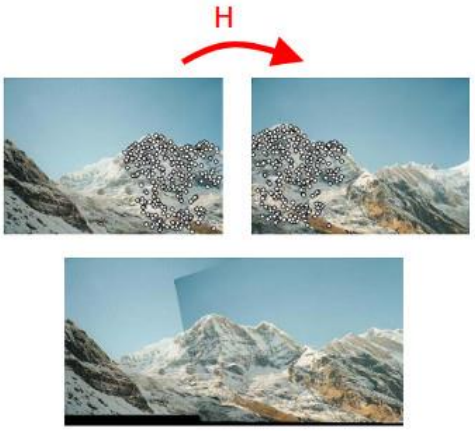
İki resimde de eşleşen noktaların arasına farklı renklerde çizgiler çizerek gösterdim. Eğer resimlerden biri çevrilmemişse yani iki resim de aynı açıdan bakıyorsa bu çizgiler paraleldir. Eğer paralelliği bozan çizgiler varsa yüksek ihtimalle yanlış bir eşleşme olmuştur.

In [72]:
```
def homography_stitching(keypoints_main_img, keypoints_query_img, matches, reprojThresh):

        keypoints_main_img = float([kp.pt for kp in keypoints_main_img])
        keypoints_query_img = float([kp.pt for kp in keypoints_query_img])

        if len(matches) > 4:
            points_main = float([keypoints_main_img[m.queryIdx] for m in matches])
            points_query = float([keypoints_query_img[m.trainIdx] for m in matches])
            (H, status) = cv2.findHomography(points_main, points_query, cv2.RANSAC, reprojThresh)

            return (matches, H, status)
        else:
            return None
```

Eşleşen Key Pointlere göre birleştirilecek olan resmin oryantasyonu ve açısı değiştiriliyor

In [62]:
```
(matches, Homography_Matrix, status) = homography_stitching(keypoints_main_img, keypoints_query_img, matches, reprojThresh=4)

print("Homography_Matrix:\n",Homography_Matrix)

width = query_photo.shape[1] + main_photo.shape[1]
height = min(query_photo.shape[0], main_photo.shape[0])
print("Width :", width, "\nHeight :",height )

result = cv2.warpPerspective(main_photo, Homography_Matrix,  (width, height))

result[0:height, 0:query_photo.shape[1]] = query_photo[0:height,:]
result = cv2.cvtColor(result,cv2.COLOR_RGB2BGR)
```

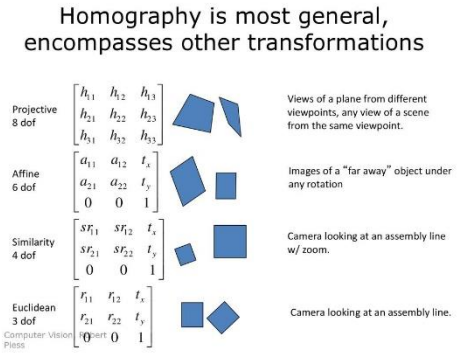
```
Homography_Matrix:
 [[-1.96580175e-05  9.99970608e-01  5.10010976e+02]
 [-9.99999871e-01 -1.71930265e-05  1.34849913e+03]
 [-1.18367688e-08 -1.24634910e-08  1.00000000e+00]]
Width : 3036
Height : 1350
```

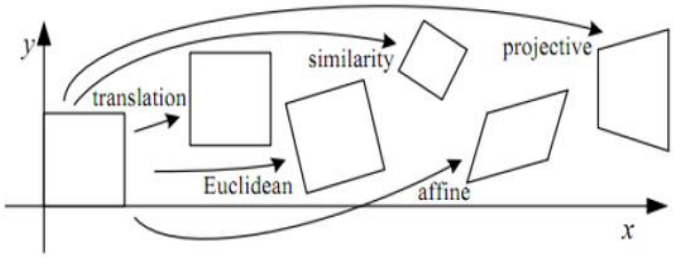


Homography, aynı düzlemsel yüzeyi görüntüleyen iki resmi ilişkilendirmemizi sağlar. Başka bir deyişle farklı açıda veya düzlemde olan resmi, başka bir açıya veya başka bir düzleme taşıma işlemidir.

Yukardaki kod parçası, eğer iki resim arasında 4 taneden fazla nokta eşleşmesi varsa bu verilerin Homography Matrisini, RANSAC regresyon kullanarak buluyor. Bulunan Matris ile resim WARP ediliyor ve iki resim üst üste getirilerek eşleşme sağlanıyor.

*RANSAC: **Ra**ndom **Sa**mple **C**onsensus, bir veri kümesinde model parametrelerine bulmak için kullanılan rassal bir yöntemdir. Rastgele bir parametre kümesi oluşturulur ve bu kümeden veriyi en iyi ifade eden parametre seçilir. Böylece veri kümesi modellenmiş olur.





Sonuç olarak aşağıda verildiği gibi bir çıktı elde ediyorum:

```
cv2_imshow(result)
```

2.1 Lane Detection:

In [90]:
```
def grayscale(img):
  return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

def canny(img, low_threshold=70, high_threshold=180):
  return cv2.Canny(img, low_threshold, high_threshold)

def gaussian_blur(img, kernel_size=5):
  return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

def draw_lines(img, lines, color=[255, 0, 0], thickness=10):
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)
```

In [91]:
```
def get_vertices(image):
    rows, cols = image.shape[:2]
    bottom_left  = [cols*0.15, rows]
    top_left     = [cols*0.45, rows*0.6]
    bottom_right = [cols*0.95, rows]
    top_right    = [cols*0.55, rows*0.6]

    ver = np.array([[bottom_left, top_left, top_right, bottom_right]], dtype=np.int32)
    return ver
```

get_vertices(): Maskelenecek alanı belirliyor

In [92]:
```
def region_of_interest(img, vertices):
    mask = np.zeros_like(img)
    ignore_mask_color = 255
    cv2.fillPoly(mask, vertices, ignore_mask_color)
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image
```

region_of_interest(): Maskelenme islemini "bitwise and" olarak yapıyor

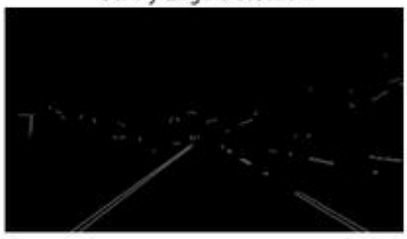
In [93]:
```
image = cv2.imread("/content/testImage.jpg")
gray_img = grayscale(image)
    #Gaussian Blur
smoothed_img = gaussian_blur(img = gray_img, kernel_size = 5)
    #Canny Edge Detection
canny_img = canny(img = smoothed_img, low_threshold = 180, high_threshold = 240)
    #Maskelenmis Resim elde et
masked_img = region_of_interest(img = canny_img, vertices = get_vertices(image))
```

- Resmi ilk önce Gray Scale haline getiriyoruz
- Gaussian blur kullanarak bozulmaları engellemeye çalıyoruz
- Canny Edge Detector kullanarak tüm kenarları buluyoruz
- Bulunan kenarları **region_of_interest()** fonksiyonuyla maskeleyerek yoldaki kenarlara yani çizgilerin sınırlarına ulaşıyoruz

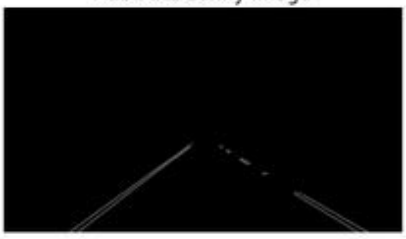
Orginal Image

Gray Scaled and Smoothed

Canny Edge Detection



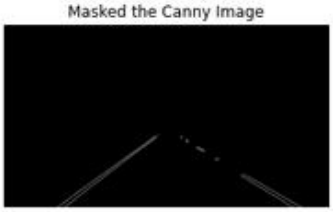
Mask the Canny Image



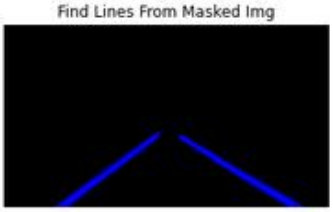
In [95]:
```python
def slope_lines(image,lines):

    img = image.copy()
    poly_vertices = []
    order = [0,1,3,2]

    left_lines = [] #  / bu egimli cizgiler
    right_lines = [] #  \ bu egimli cizgiler
    for line in lines:
        for x1,y1,x2,y2 in line:

            if x1 == x2:
                pass #dikey cizgiler
            else:
                m = (y2 - y1) / (x2 - x1)
                c = y1 - m * x1

                if m < 0:
                    left_lines.append((m,c))
                elif m >= 0:
                    right_lines.append((m,c))

    left_line = np.mean(left_lines, axis=0)
    right_line = np.mean(right_lines, axis=0)

    print(left_line, right_line)

    for slope, intercept in [left_line, right_line]:

        rows, cols = image.shape[:2]
        y1= int(rows)

        #y2 degeri: gercek height degerinin %60 ustu ya da y1 degerinin %60 altidir
        y2= int(rows*0.6) #int(0.6*y1)

        #Dogru denklemi y=mx +c bu sekilde de ifade edebiliriz x=(y-c)/m
        x1=int((y1-intercept)/slope)
        x2=int((y2-intercept)/slope)
        poly_vertices.append((x1, y1))
        poly_vertices.append((x2, y2))
        draw_lines(img, np.array([[[x1,y1,x2,y2]]]))

    poly_vertices = [poly_vertices[i] for i in order]
    cv2.fillPoly(img, pts = np.array([poly_vertices],'int32'), color = (0,155,100))
    return cv2.addWeighted(image,0.7,img,0.4,0.)

def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):

    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineLength=min_line_len, maxLineGap=max_line_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img, lines)
    justLines=line_img.copy()
    line_img = slope_lines(line_img,lines)
    return justLines,line_img
```

- **hough_lines()** fonksiyonuyla Canny edge detector ile belirlediğimiz kenarları çizgi olarak elde ediyoruz
- Ulaştığımız çizgileri yatay mı dikey mi (eğer dikeyse solda mı sağda mı) diye ayırıyoruz
- Tabiki birden fazla çizgi tespit edilecek sağ tarafta olanları kendi arasında sol tarafta olanları kendi arasında gruplayarak iki ayrı çizgi elde etmeye çalışıyoruz
- Sonra bu iki çizgi arasını yeşil renkle boyuyoruz

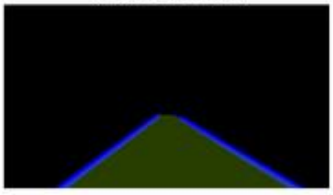
Masked the Canny Image



Find Lines From Masked Img



Color Between Lines



Mix the Original and Detected Img

In [99]:
```
def detect_line(image):
    #Grayscale
    gray_img = grayscale(image)

    #Gaussian Blur
    smoothed_img = gaussian_blur(img = gray_img, kernel_size = 5)
    #Canny Edge Detection
    canny_img = canny(img = smoothed_img, low_threshold = 180, high_threshold = 240)
    #Maskelenmis Resim elde et
    masked_img = region_of_interest(img = canny_img, vertices = get_vertices(image))
    #Cizgilere Hough Transform uygula
    _,houghed_lines = hough_lines(img = masked_img, rho = 1, theta = np.pi/180, threshold = 20, min_line_len = 20, max_line_gap = 180)
    #Kenarlara Cizgi Ciz
    output = weighted_img(img = houghed_lines, initial_img = image, a=0.8, b=1., c=0.)

    return output
```

In [100]:

```
[ -0.80367214 691.86525043] [ 0.57278626 33.13205845]
```

Input Image

Output Image [Lane Line Detected]

```
[ -0.70722619 646.68756969] [ 0.64353083 -1.93621793]
```

Input Image

Output Image [Lane Line Detected]

```
[ -0.70205846 646.06231573] [ 0.59158939 31.51032335]
```

3. Bulgular ve Çıktılar

- 2. Kısımda bahsettiğim işlemler fotoğraf kareleri içindi. Ben ise bu işlemleri videolar üzerinde yapmak istediğim için videoların her bir frame'ini alıp bu işlemlerden geçirip başka bir video olarak kaydettim.
- En başta bahsettiğim üzere bir arabanın üzerinden 3 farklı açıdan alınmış görüntüm bulunmamakta. Bu yüzden ben de bir videonun 3 farklı kısmını kırpıp 3 farklı video elde ettim.

- Bu 3 videoyu "Stitch" ettikten sonra Şerit Tespiti yaptım ve elde ettiğim sonuç:

4. Sonuç

Bu ödevlendirmede: SURF, SIFT gibi algoritmaların gerçek hayatta nasıl örneklendiğini, birden fazla metot üzerinde çalışarak Keypointlerin nasıl eşleştirildiğini (BF, KNN, FLANN), homograpy matrisinin ne olduğunu ve warping için ne kadar önemli olduğunu öğrendim.

Ayrıca, uzun soluklu bir projenin devamlılığını nasıl sağlayacağımı, başa çıkmakta zorlandığım problemleri nasıl halledeceğimi, ortaya koyulan çalışmanın her zaman geliştirilebilecek bir noktası olabileceğinin farkına vardım.