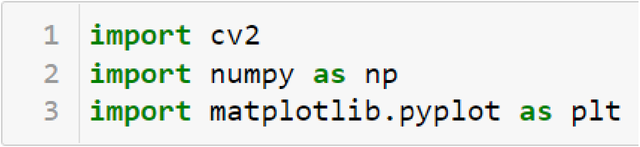
Lane Line Detection Using OpenCV

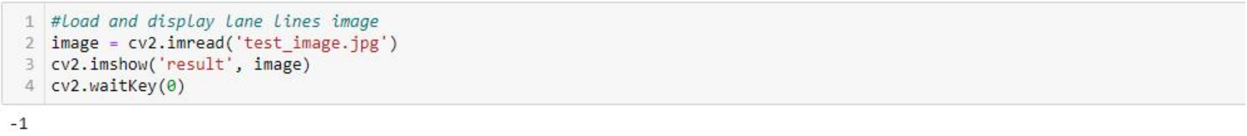
This document describes the Python script used to detect lane lines in images and videos using OpenCV, an open-source computer vision library.

First, let’s import the following python libraries.



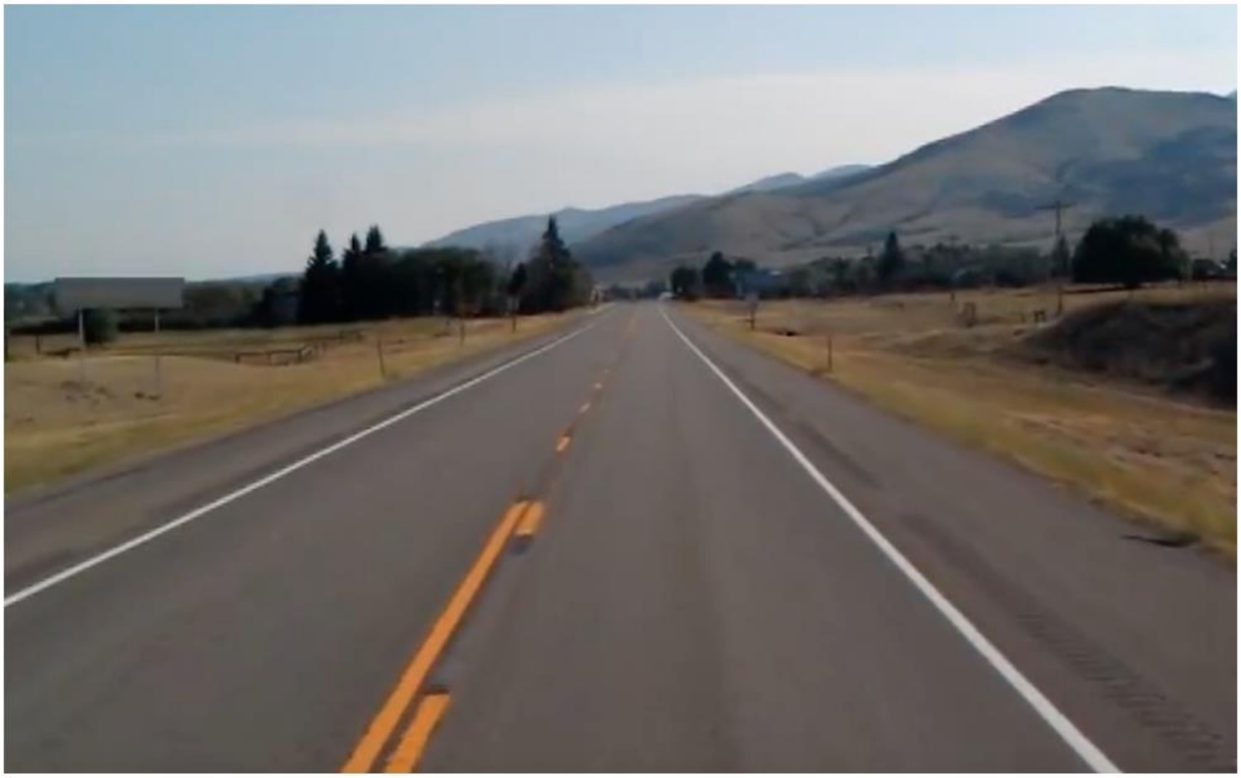
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

Next, let’s import an image with lane lines in order to develop our script.

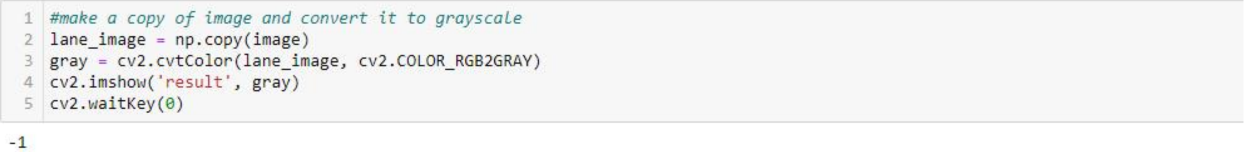


```
#load and display lane lines image
image = cv2.imread('test_image.jpg')
cv2.imshow('result', image)
cv2.waitKey(0)
```

```
-1
```



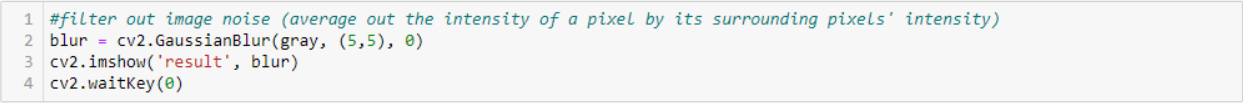
As we will be using a method called **Canny Edge detection** which is not concerned about colours, we can first **convert the image to grayscale**.



```
#make a copy of image and convert it to grayscale
lane_image = np.copy(image)
gray = cv2.cvtColor(lane_image, cv2.COLOR_RGB2GRAY)
cv2.imshow('result', gray)
cv2.waitKey(0)
```

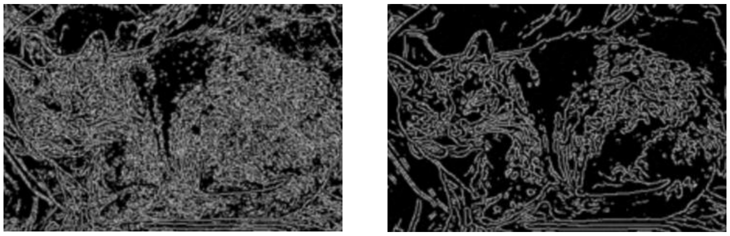
```
-1
```

Since edge detection is susceptible to noise in the image, the next step is to **remove the noise in the image with a 5x5 Gaussian filter**. Gaussian blur works by smoothing / averaging out the intensity of a pixel by its surrounding pixels' intensity.

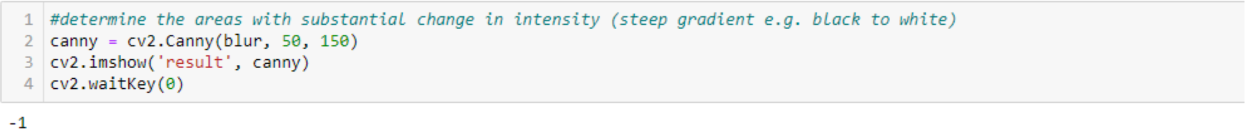


```
#filter out image noise (average out the intensity of a pixel by its surrounding pixels' intensity)
blur = cv2.GaussianBlur(gray, (5,5), 0)
cv2.imshow('result', blur)
cv2.waitKey(0)
```

Example of how smoothing affects edge detection

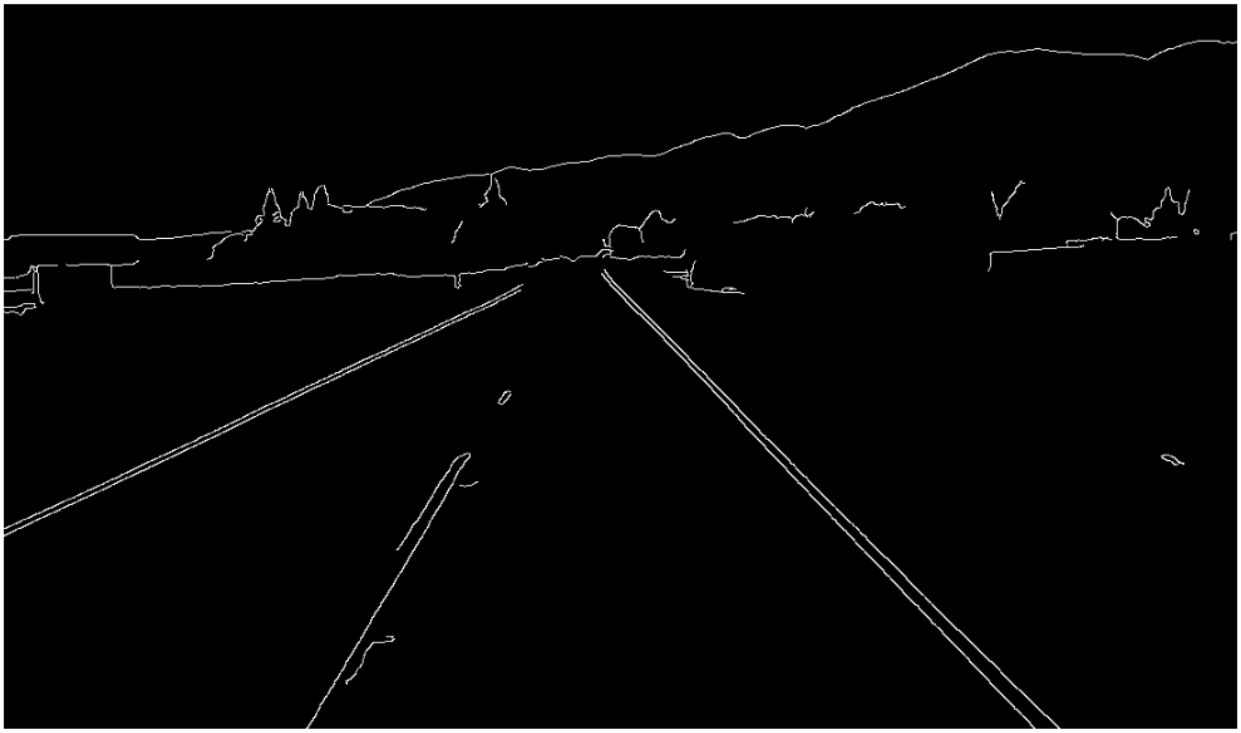


Now, we can apply the **cv2.Canny()** method to the image to determine areas with substantial change in pixel intensity (e.g. black to white). These are edges with steep gradient of intensity.

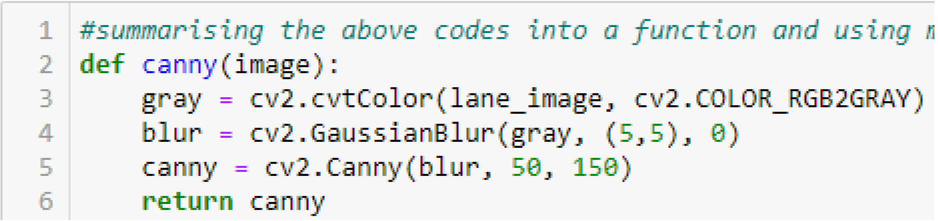


```
#determine the areas with substantial change in intensity (steep gradient e.g. black to white)
canny = cv2.Canny(blur, 50, 150)
cv2.imshow('result', canny)
cv2.waitKey(0)
```

```
-1
```

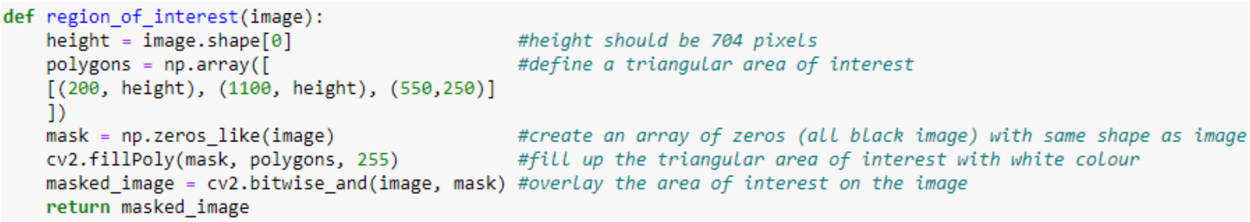


We can **combine the above functions into 1** and call it canny().

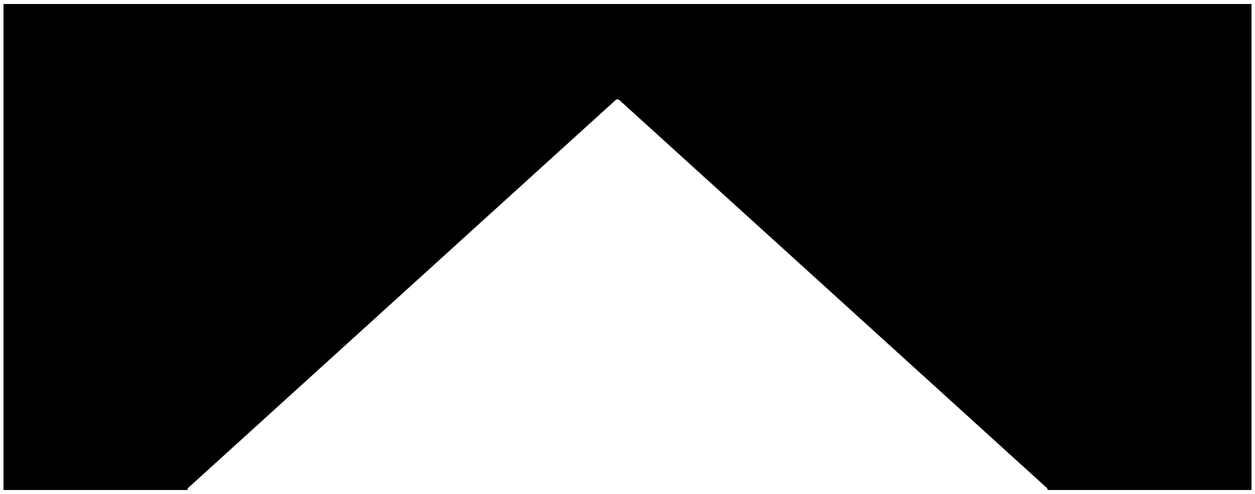


```
#summarising the above codes into a function and using
def canny(image):
    gray = cv2.cvtColor(lane_image, cv2.COLOR_RGB2GRAY)
    blur = cv2.GaussianBlur(gray, (5,5), 0)
    canny = cv2.Canny(blur, 50, 150)
    return canny
```

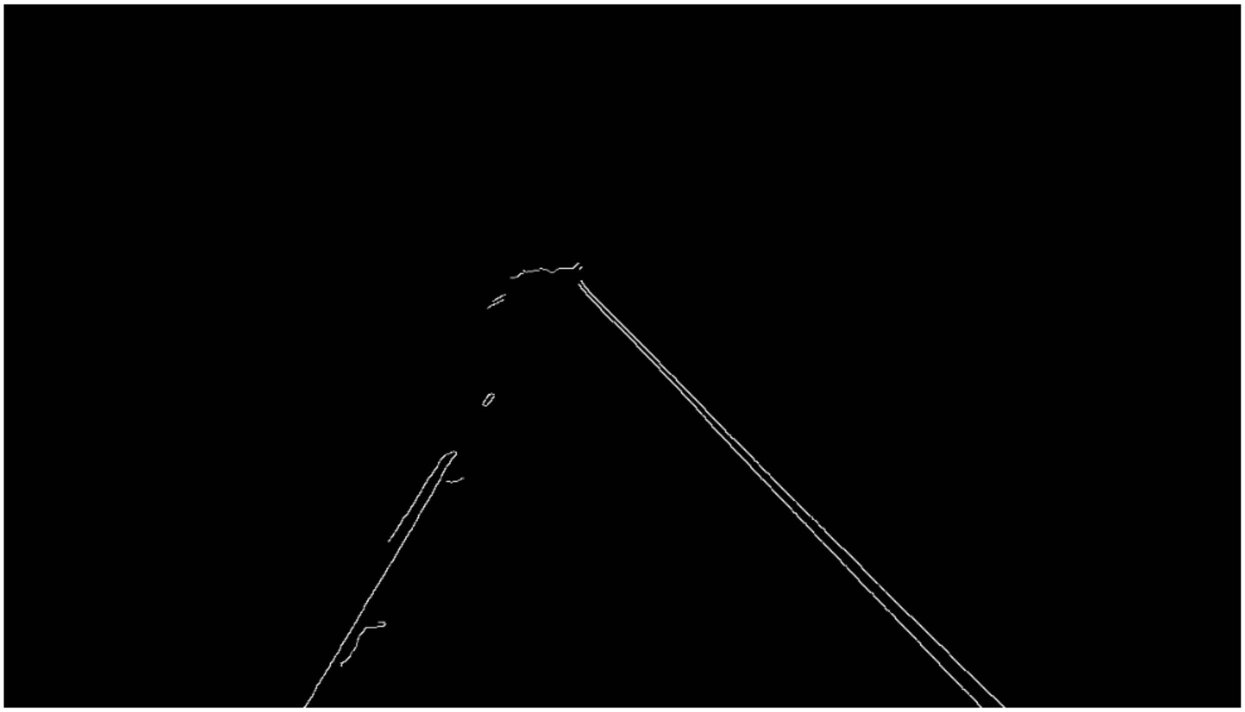
Next, we want to **filter out the region of interest** which contains the lane lines. To do this, we can use the function below. What it basically does is to **create an image with a white colour triangular region that surrounds the lane lines**. Everything else outside the region will be black. We then overlay that region over the original image and return areas where both layers are white.

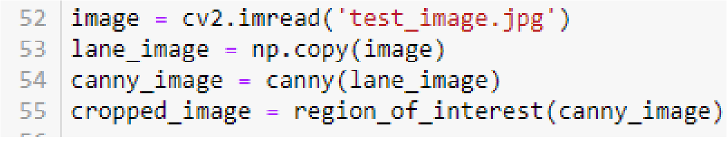


```
def region_of_interest(image):
    height = image.shape[0]                     #height should be 704 pixels
    polygons = np.array([                       #define a triangular area of interest
    [(200, height), (1100, height), (550,250)]
    ])
    mask = np.zeros_like(image)                 #create an array of zeros (all black image) with same shape as image
    cv2.fillPoly(mask, polygons, 255)           #fill up the triangular area of interest with white colour
    masked_image = cv2.bitwise_and(image, mask) #overlay the area of interest on the image
    return masked_image
```



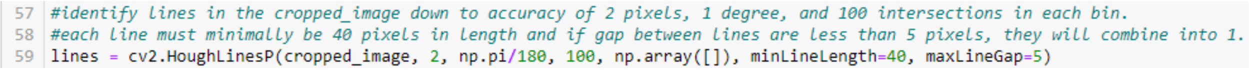
After Filtering Out Region of Interest





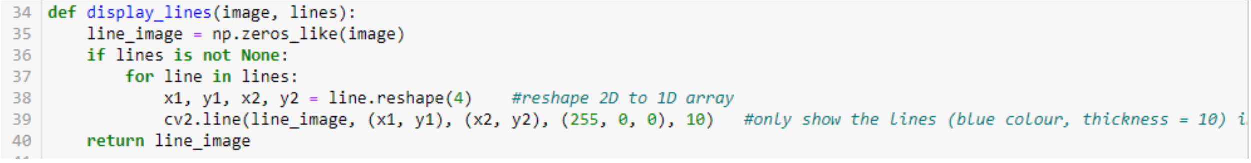
```python
image = cv2.imread('test_image.jpg')
lane_image = np.copy(image)
canny_image = canny(lane_image)
cropped_image = region_of_interest(canny_image)
```

With this, we can now use **the cv2.HoughLinesP() method to detect straight lines. Note that this function outputs the extremes of the detected lines (x0,y0,x1,y1)**. Refer to code comments below for details of the function parameters.

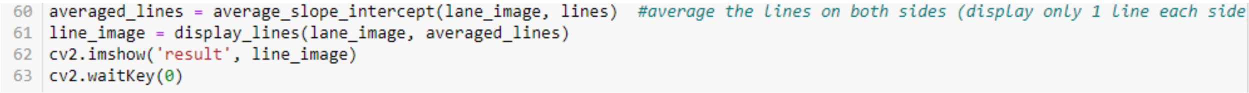


```python
#identify lines in the cropped_image down to accuracy of 2 pixels, 1 degree, and 100 intersections in each bin.
#each line must minimally be 40 pixels in length and if gap between lines are less than 5 pixels, they will combine into 1.
lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180, 100, np.array([]), minLineLength=40, maxLineGap=5)
```

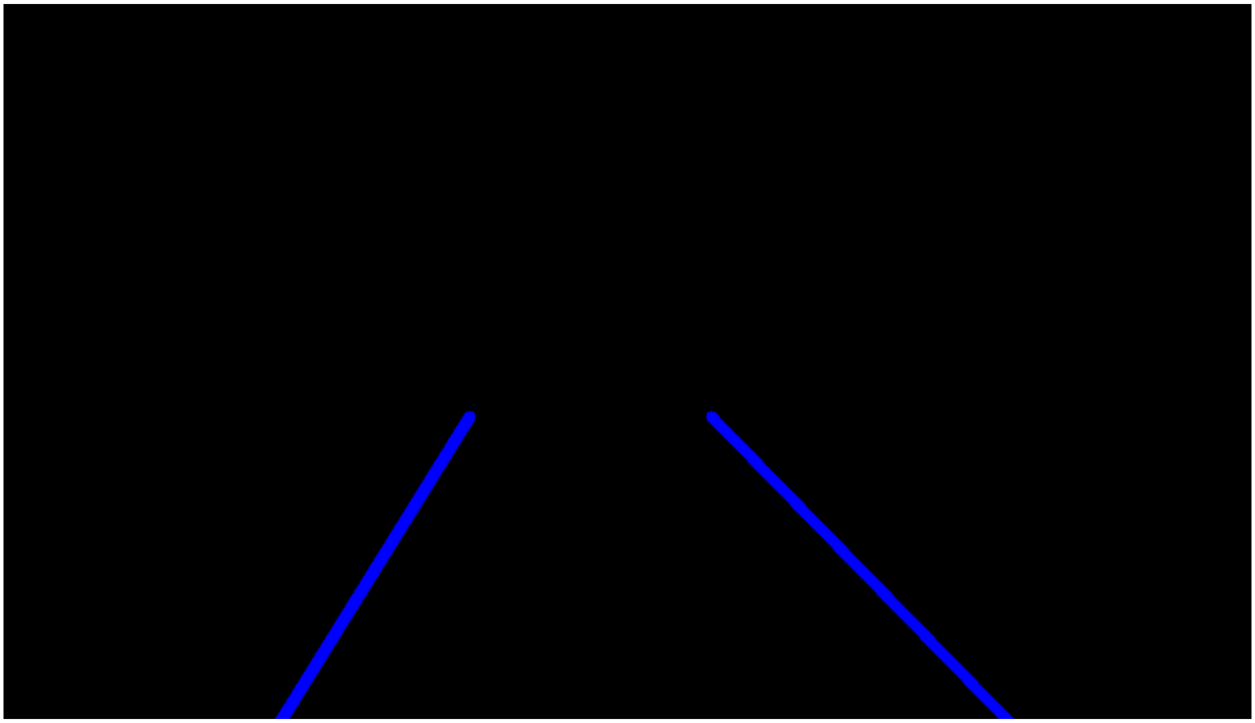
Now that we obtain the coordinates of the lines, we can use the following method to **display the lines**.



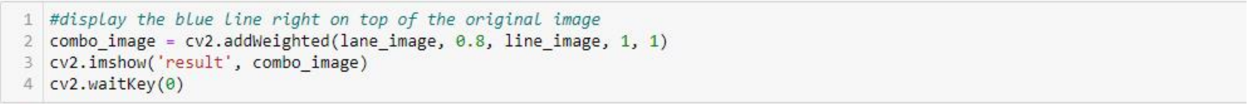
```python
def display_lines(image, lines):
    line_image = np.zeros_like(image)
    if lines is not None:
        for line in lines:
            x1, y1, x2, y2 = line.reshape(4)    #reshape 2D to 1D array
            cv2.line(line_image, (x1, y1), (x2, y2), (255, 0, 0), 10)   #only show the lines (blue colour, thickness = 10) i
    return line_image
```



```python
averaged_lines = average_slope_intercept(lane_image, lines)  #average the lines on both sides (display only 1 line each side
line_image = display_lines(lane_image, averaged_lines)
cv2.imshow('result', line_image)
cv2.waitKey(0)
```

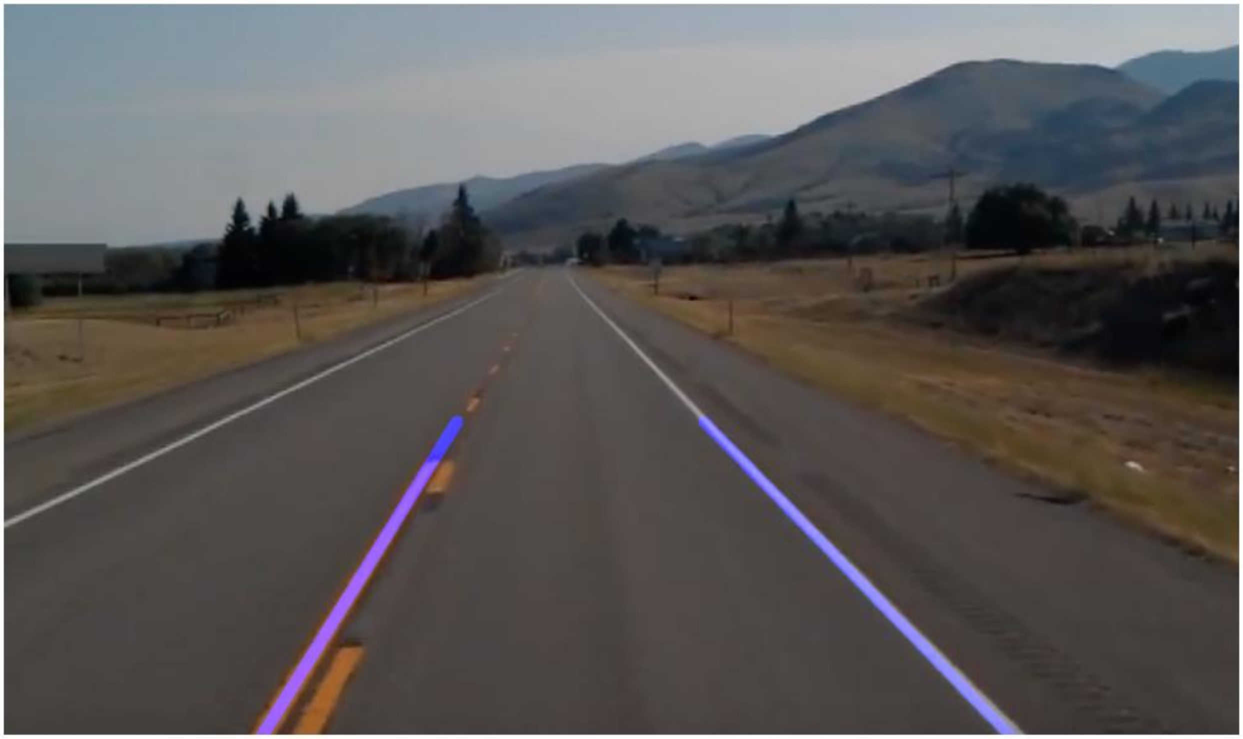


Next, we will use the **cv2.addWeighted() function** to overlay the line on top of our original image.

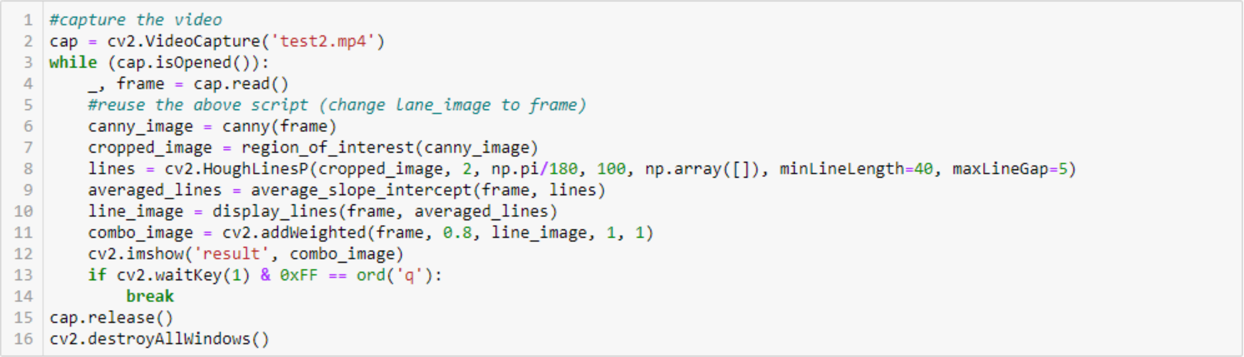


```
#display the blue line right on top of the original image
combo_image = cv2.addWeighted(lane_image, 0.8, line_image, 1, 1)
cv2.imshow('result', combo_image)
cv2.waitKey(0)
```

```
-1
```



What's remaining is to **do the same thing for a video** instead of a single image. To do that, we just need to use **cv2.VideoCapture()** to read individual frames, and then use back the above codes to overlay the lines on top of the frames.



```
#capture the video
cap = cv2.VideoCapture('test2.mp4')
while (cap.isOpened()):
    _, frame = cap.read()
    #reuse the above script (change lane_image to frame)
    canny_image = canny(frame)
    cropped_image = region_of_interest(canny_image)
    lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180, 100, np.array([]), minLineLength=40, maxLineGap=5)
    averaged_lines = average_slope_intercept(frame, lines)
    line_image = display_lines(frame, averaged_lines)
    combo_image = cv2.addWeighted(frame, 0.8, line_image, 1, 1)
    cv2.imshow('result', combo_image)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```