Practice of Autonomous Driving

Project I. Finding Lane Line

P46091204

蔡承穎

航太碩 碩一簡介

簡介

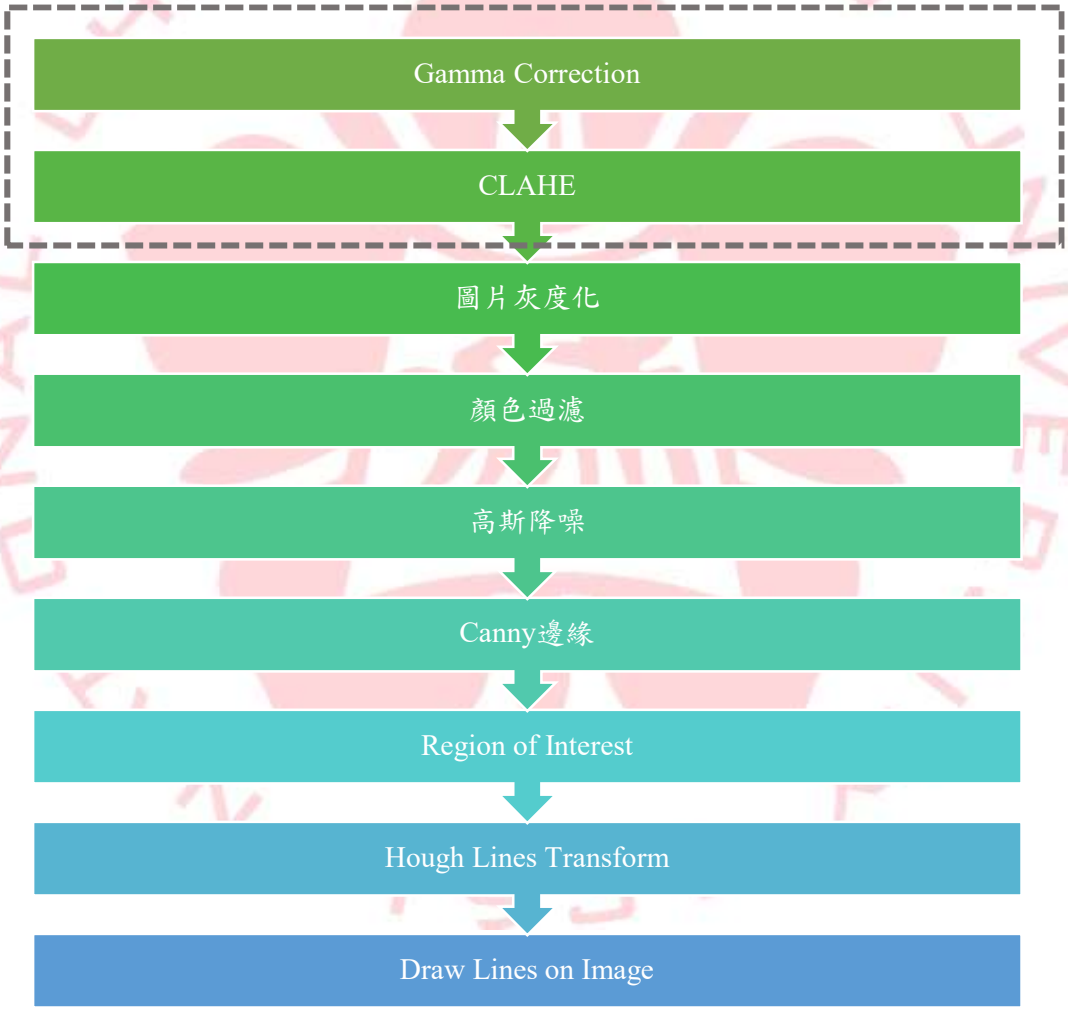
Land detection 是為了偵測 lanes on the road，並提供每條 lane 的正確位置與大小，本文會如下格式作報告，壹、Project 架構，貳、Project 方法，參、結果，肆、心得討論。除此之外，影片結果已放上 Youtube，並且此專案已放上 Github，source code：https://github.com/ab458629/Practice-of-Autonomous-Driving/tree/main/Land%20Detection/Land_Detection

測試影片已上傳至 Youtube
以下三個影片是沒有做 Gamma 校正與 CLAHE 當作前處理的方法
(1) solidWhiteRight.mp4
https://www.youtube.com/watch?v=5tI5Tae0CSs
(2) solidYellowLeft.mp4
https://www.youtube.com/watch?v=ect0s_uV50A
(3) challenge.mp4
https://www.youtube.com/watch?v=HwoEk8aO8ug

最後一個影片則是有做影像前處理(Gamma 校正、限制對比度自適應直方圖均衡化(CLAHE))，避免光源問題
(4) Bonus
https://www.youtube.com/watch?v=Lj3_ciOsvGg

壹、Project 架構

以下為 Finding Lane Line 程式碼的架構圖(圖一)，灰色虛線外框表示，在 Github 上的三個影片中沒有使用到的演算法，在 test.py 定義了輸入影片或是輸入照片的入口，接著每一幀皆會經過灰度化，接著利用 HSV 顏色空間找出黃色及白色上下限，將白色與黃色的 Mask OR 起來，接著 AND 原本的灰度照片，即可得到過濾除了白色與黃色的灰度照片。接著經過 kernel size 為 3，標準差為 0 的高斯過濾器，即可得到降噪的圖。然後經過 Canny 函式可以得到圖像邊緣，這些邊緣要先決定 ROI（Region of Interest），再經過 Hough 轉換找出直線，主要是因為 Hough 轉換函式的時間複雜度。得到一些線後，即可分成左線與右線，再找尋這些線的平均斜率、截距等方式，取得最終的線與原圖疊加。



圖一、Finding Lane Line 架構

貳、Project 方法

一、圖片灰度化

```
def grayscale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

讀取圖像後，由 OpenCV 將 RGB 轉換成 Gray，另法就是在讀取的時候 cv2.imread 時的第二參數設為 0，接下來以 whiteCarLaneSwitch.jpg 作測試（圖二）。

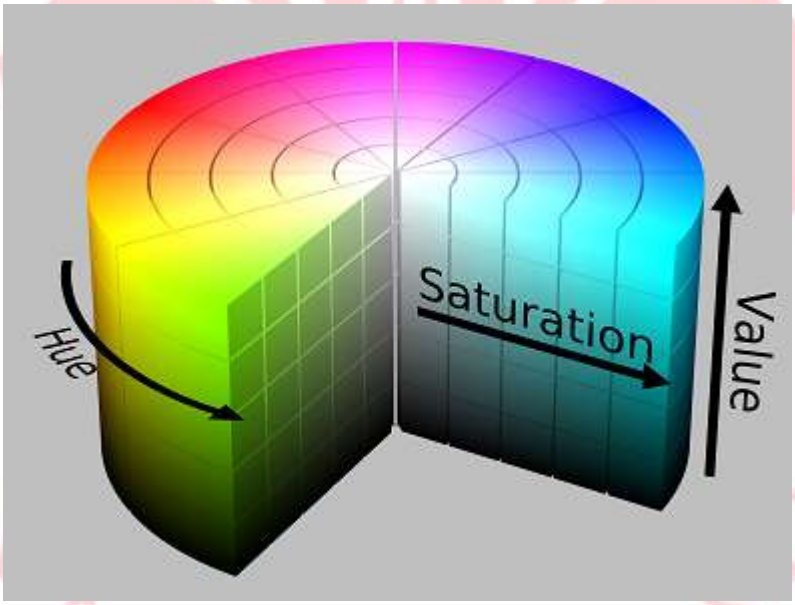
圖二、原圖

圖三、灰度圖

二、顏色過濾

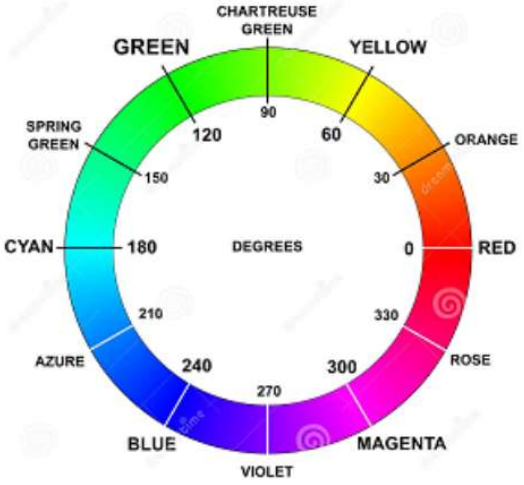
```
def color_filter(img):
    hsv_img = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    gray_image=grayscale(img)
    lower_yellow = np.array([26, 43, 46], dtype = "uint8")
    upper_yellow = np.array([100, 255, 255], dtype="uint8")
    mask_yellow = cv2.inRange(hsv_img, lower_yellow, upper_yellow)
    mask_white = cv2.inRange(gray_image, 200, 255)
    mask_yw = cv2.bitwise_or(mask_white, mask_yellow)
    mask_yw_image = cv2.bitwise_and(gray_image, mask_yw)
    return mask_yw_image
```

為了提取只讓白色與黃色通過的 Mask，故先將圖片轉換成 HSV 空間（圖四）。H 表示色相，S 表示飽和度，V 表示明度，可以將顏色看成這是什麼顏色？深淺如何？明暗如何？三個問題。



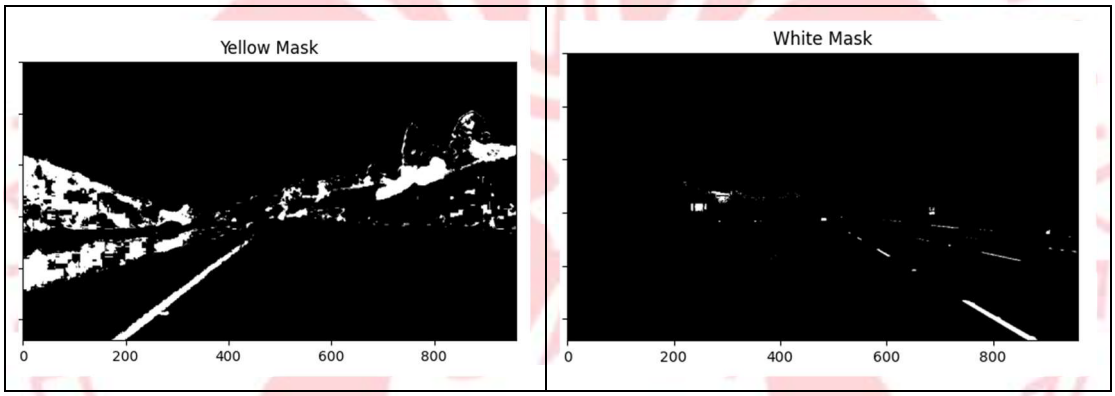
圖四、HSV 色彩空間

首先先將黃色的上下限定義出來，色相部分可由 HSV 色相圖（圖五）得到，這裡定義 $(h,s,v)_{\min}=(26,43,46)$ 、$(h,s,v)_{\max}=(100,255,255)$。接著白色部分直接使用灰度得到，也就是定義灰度 200~255 為白色。

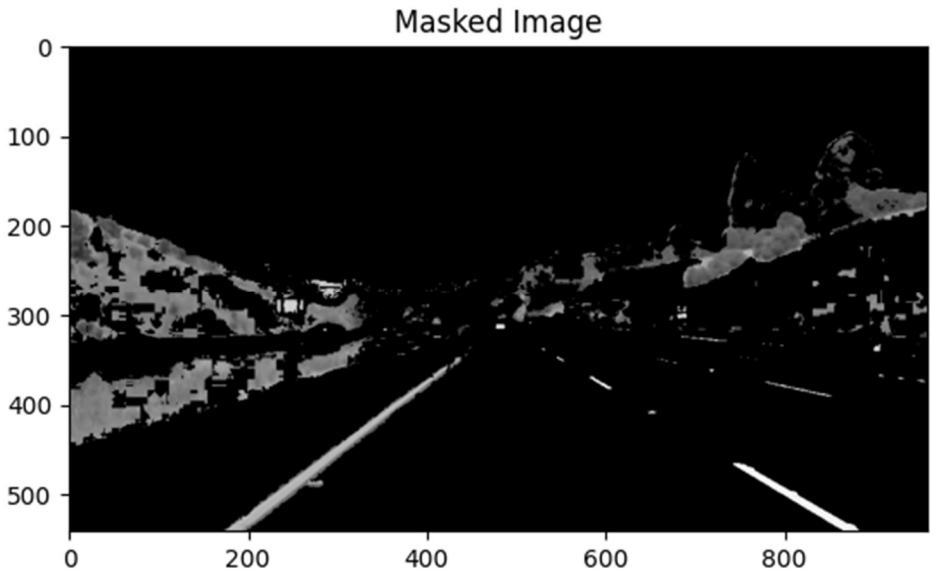


圖五、色相圖

得到白色與黃色的 Mask 後(圖五)，將其 OR 起來，再跟原圖 AND 起來，就可以得到結果（圖六）。



圖五、Yello Mask 與 White Mask



圖六、經過 Mask 過後的結果

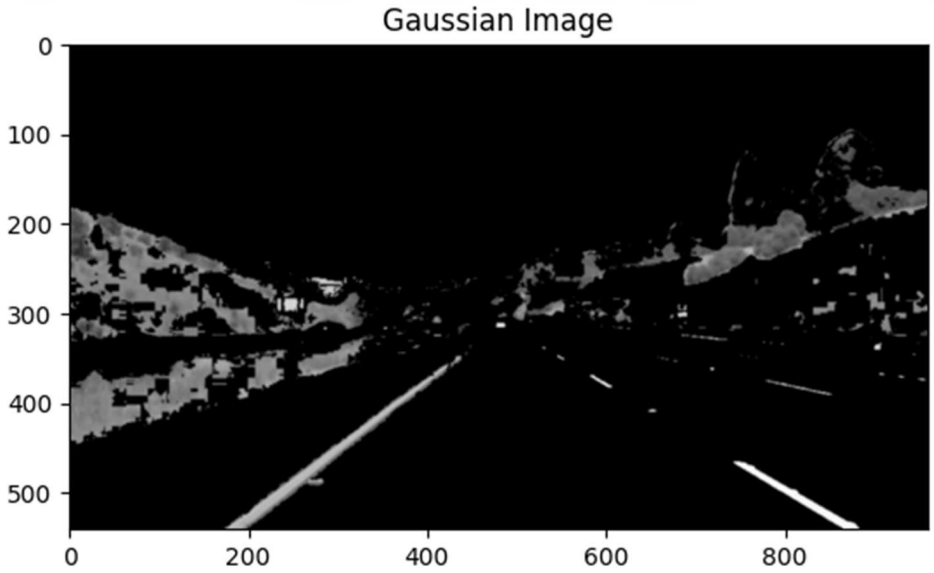
三、高斯降噪

```
def gaussian_blur(img, kernel_size=3, sigma=0):
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), sigma)
```

常見過濾雜訊方法有：(1)均值濾波、(2)中值濾波、(3)高斯濾波，強化邊緣方法有 Laplace Filter、Sobel Filter 等，是用來作特徵提取的，像是有名的 SIFT。

高斯模糊其實就是用一個 kernel size * kernel size 大小的高斯分佈的矩陣，對圖像作 Convolution，stride 為 1，因為圖像大小不能變，故要 padding 0。

將圖像模糊化，其實就是低通濾波，為了不讓圖像失真，故要將中心權重增加，將邊緣的權重降低，這就是高斯濾波的由來。這裡將 kernel size 設為 3，標準差設為 0，結果如圖七。



圖七、高斯降噪後結果

四、Canny 邊緣

```
def canny(img,low_threshold=100,high_threshold=200):
    # void Canny(InputArray image, OutputArray edges, double threshold1, double threshold2, int apertureSize=3, bool L2gradient=false )
        return cv2.Canny(img,low_threshold,high_threshold)
```

Canny 演算法的過程：

(1) 取得圖像每個 pixel 的梯度值與梯度方向，可用 Soble 運算子得到

Sobel 運算子分別計算 x, y 方向梯度，將方向歸類成上下、左右、右上左下、左下右上，並計算每個其梯度大小，OpenCV 預設 L1 norm。

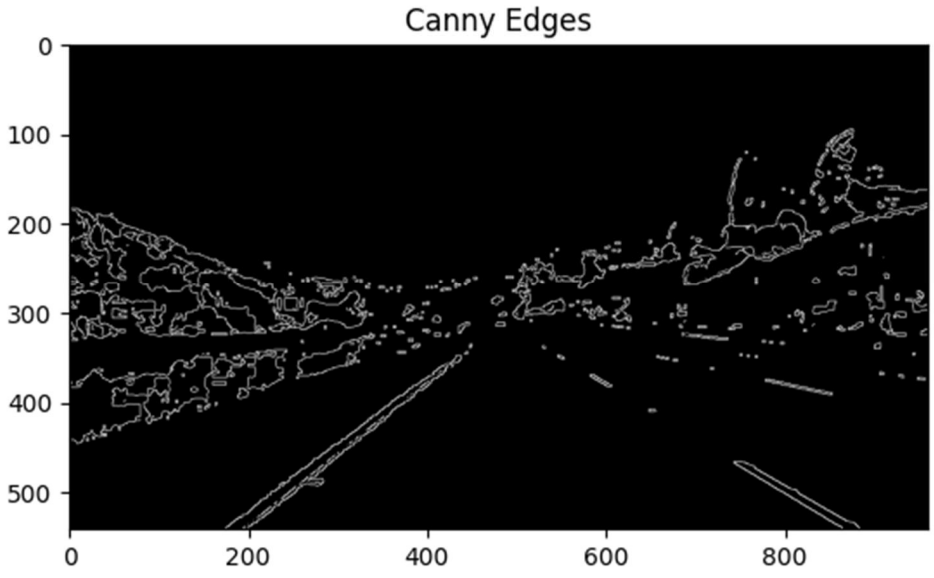
(2) 由 Non-maximum suppression 尋找可能的邊緣

因為 Edge 附近都會有非零的梯度值，會造成很粗的 Edge，故使用非極大抑制，也就是把每個 pixel 和梯度方向鄰居比較梯度值，不是最大就去除。

(3) 根據兩個 threshold 得到 strong edge 與 weak edge

大於 high_threshold，就是邊緣，小於 low_threshold，則不是，在兩者之間稱作 weak edge。

(4) 與 strong edge 相連的 strong edge 當作確定的 edge，結果如圖八。



圖八、Canny 演算法結果

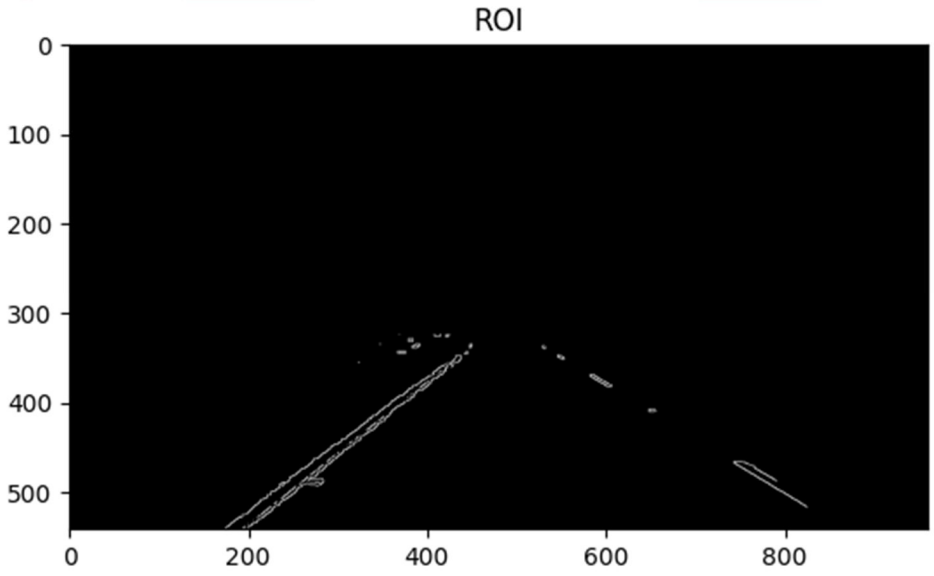
五、ROI

```
def ROI(img, vertices):
    if len(img.shape) > 2:
        channel_count = img.shape[2] # i.e. 3 or 4 depending on your image
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255

    mask = np.zeros_like(img)
    cv2.fillPoly(mask, np.array([ROI_vertices], np.int32), ignore_mask_color)
    masked_img = cv2.bitwise_and(img, mask)
    return masked_img
```

因為並非所有邊緣都是我們需要的，故我們可以定義一個多邊形，來選出 ROI，頂點定義如下，是個梯形，接著將其作成 mask，再與原圖 AND 起來，得結果如圖九。

```
lower_left = [imshape[1]/9,imshape[0]]
lower_right = [imshape[1]-imshape[1]/9,imshape[0]]
top_left = [imshape[1]/2-imshape[1]/8,imshape[0]/2+imshape[0]/10]
top_right = [imshape[1]/2+imshape[1]/8,imshape[0]/2+imshape[0]/10]
vertices = [np.array([lower_left,top_left,top_right,lower_right],dtype=np.int32)]
```



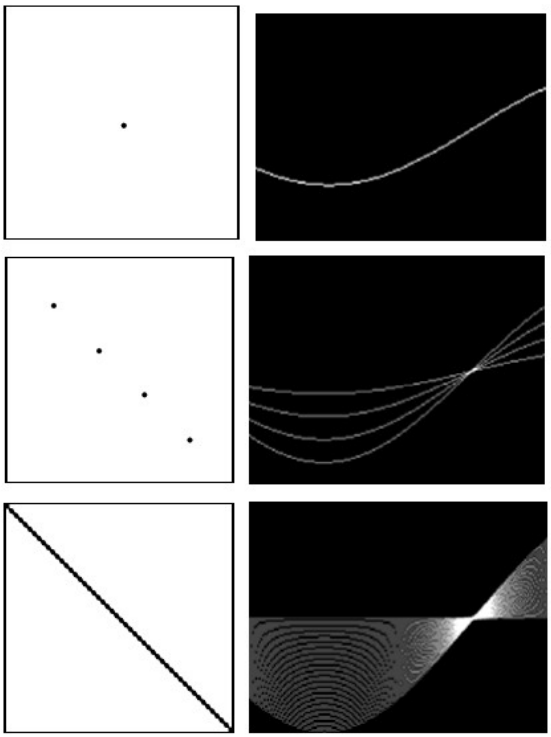
圖九、ROI

六、Hough Lines Transform

```
def hough_lines(img, rho=1.0, theta=np.pi/180, threshold=160, minLineLength=40,
maxLineGap=25):
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]),
minLineLength, maxLineGap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img,lines)
    return line_img
```

霍夫轉換演算法的核心精神就是，給定很多點，判斷這些點是否共線，經過霍夫轉換，變成判斷一堆取線是否在(r,θ)平面相交於同一點，霍夫轉換的數學式子如下，霍夫轉換過程，如圖十。

$$r = x_0 \cos\theta + y_0 \sin\theta = \sqrt{x_0^2 + y_0^2}\cos(\theta - \phi)$$



圖十、霍夫轉換

接著使用累加器二維矩陣，偵測圖片是否有直線可以用$r = x\cos\theta + y\sin\theta$來描述，如果有，就把該條直線的$(r,\theta)$對應累加器的元素加一，元素越大，代表著出現直線的 believe 越大，最後選出其元素裡，大於 threshold 的一些 local maximum，就有機會找到圖片上直線。

這裡使用的是 cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineLength, maxLineGap) ，介紹其參數的意義。

(1) rho：為以像素為單位距離 r 的精度。

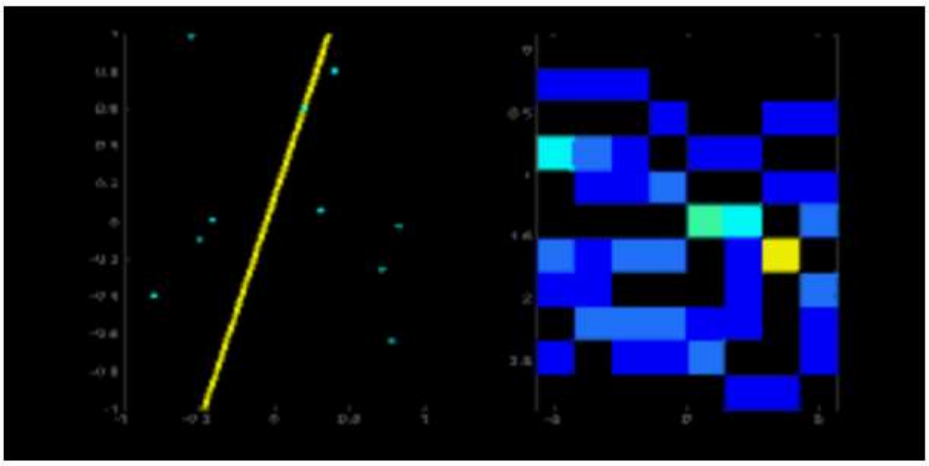
(2) theta：一般使用 np.pi/180 表示搜尋所有可能角度。

(3) threshold：越小，判斷是直線的數目就會越多，很好理解。

(4) minLineLength：表示用來控制直線最小長度。

(5) maxLineGap：若兩點的線段長度超過此值，則兩點不在一條線上。

注意，若量化的間距太細，容易讓票數分散，以及霍夫轉換的效率取決於圖片的品質，圖十一表示霍夫轉換與其累加器。



圖十一、累加器

七、Draw Lines on Image

```
def draw_lines(img, lines, color=[0, 0, 255], thickness=5):
    # the left lane will have a negative slope and right positive
        def get_slope(x1,y1,x2,y2):
                return (y2-y1)/(x2-x1)

        global cache
        global first_frame

        y_global_min = img.shape[0]
        y_max = img.shape[0]
        l_slope, r_slope, l_lane, r_lane = [], [], [], []
        det_slope = 0.4
        alpha = 0.3

        if lines is None:
                print ('Lines is not enough!')
                return 1

        for line in lines:
                for x1,y1,x2,y2 in line:
                        slope = get_slope(x1,y1,x2,y2)
                        if slope > det_slope:
                                r_slope.append(slope)
                                r_lane.append(line)
                        elif slope < -det_slope:
                                l_slope.append(slope)
                                l_lane.append(line)

                y_global_min = min(y1,y2,y_global_min)

        if((len(l_lane) == 0) or (len(r_lane) == 0)):
                print ('No lane detected!')
                return 1

        l_slope_mean = np.mean(l_slope)
        r_slope_mean = np.mean(r_slope)
        l_mean = np.mean(np.array(l_lane),axis=0)
```

```
        r_mean = np.mean(np.array(r_lane),axis=0)

        if ((r_slope_mean == 0) or (l_slope_mean == 0 )):
            print('Dividing by zero!')
            return 1

        l_b = l_mean[0][1] - (l_slope_mean * l_mean[0][0])
        r_b = r_mean[0][1] - (r_slope_mean * r_mean[0][0])

        l_x1 = int((y_global_min - l_b)/(l_slope_mean))
        l_x2 = int((y_max - l_b)/(l_slope_mean))
        r_x1 = int((y_global_min - r_b)/(r_slope_mean))
        r_x2 = int((y_max - r_b)/(r_slope_mean))

        if l_x1 > r_x1:
            l_x1 = int((l_x1+r_x1)/2)
            r_x1 = l_x1
            l_y1 = int((l_slope_mean * l_x1 ) + l_b)        # 左小 y
            r_y1 = int((r_slope_mean * r_x1 ) + r_b)        # 右小 y
            l_y2 = int((l_slope_mean * l_x2 ) + l_b)        # 左大 y
            r_y2 = int((r_slope_mean * r_x2 ) + r_b)        # 右大 y
    else:
            l_y1 = y_global_min
            l_y2 = y_max
            r_y1 = y_global_min
            r_y2 = y_max

        current_frame = np.array([l_x1,l_y1,l_x2,l_y2,r_x1,r_y1,r_x2,r_y2])

        if first_frame == 1:
            next_frame = current_frame
            first_frame = 0
        else :
            prev_frame = cache
            next_frame = (1-alpha)*prev_frame+alpha*current_frame

        cv2.line(img, (int(next_frame[0]), int(next_frame[1])),
(int(next_frame[2]),int(next_frame[3])), color, thickness)
```

```
        cv2.line(img, (int(next_frame[4]), int(next_frame[5])),
(int(next_frame[6]),int(next_frame[7])), color, thickness)

        cache = next_frame
```

首先，先將霍夫轉換演算法傳入的 Lines，分別計算其斜率，並用斜率 0.4 當作 threshold，若斜率大過它，則加入至右斜線的 list，反之亦然。

取最小的 y 值當作 global minimum，並計算所有左線、右線的斜率平均值，與左線、右線的平均值。如果斜率為 0，raise a error。

接著我們求左右線的斜率，以左線為例，利用 $b = y - mx$ 來計算，l_b = l_mean[0][1] - (l_slope_mean * l_mean[0][0])，計算完截距，計算左右線的 x 值，可以利用 $x = \frac{y-b}{m}$，l_x1 = int((y_global_min - l_b)/(l_slope_mean))，l_x2 = int((y_max - l_b)/(l_slope_mean))，這裡算出了最大與最小的 x 值。

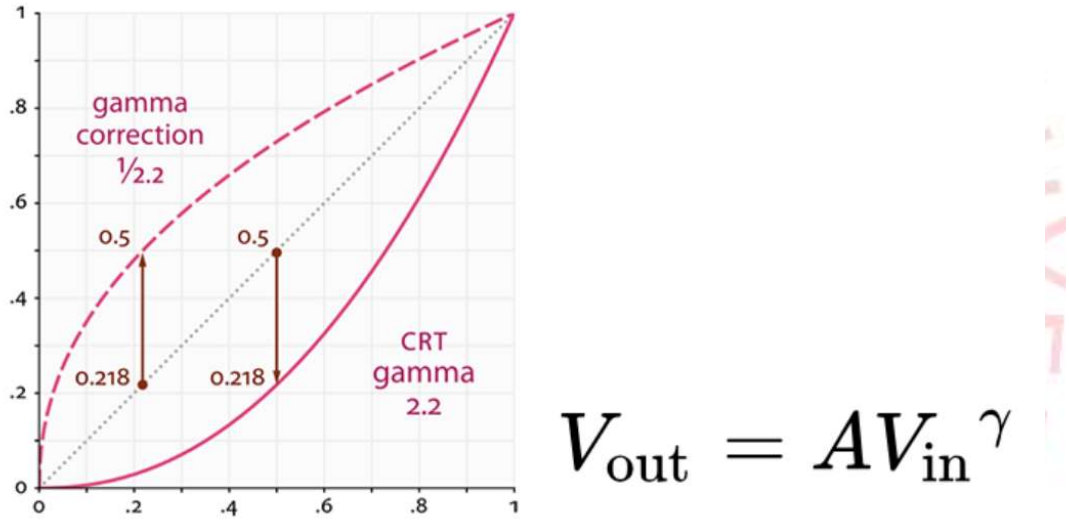
如果左線的最小 x 值大於右線的最小 x 值，則將左線的 x 值設為 int((l_x1+r_x1)/2)，右線的 x 值就設為 r_x1 = l_x1，接下來就套直線公式。若左線的最小 x 值小於右線的最小 x 值，表示不會出現交叉情形，直接將兩條線的 y 值這樣令 l_y1 = y_global_min，l_y2 = y_max，r_y1 = y_global_min，r_y2 = y_max，此時我們就有兩條線了。

將剛剛計算的兩條線令成 current_frame = np.array([l_x1,l_y1,l_x2,l_y2,r_x1,r_y1,r_x2,r_y2])，若是第一個 frame，就直接畫上去，若不是則 next_frame = (1-alpha)*prev_frame+alpha*current_frame，用這種指數遞減的方式，計算現在的兩條線，alpha 在這設成 0.4，也就是上一個 frame 的權重是 0.4，現在是 0.6，比較 Robust，畢竟車道線並不會突然的大改變。

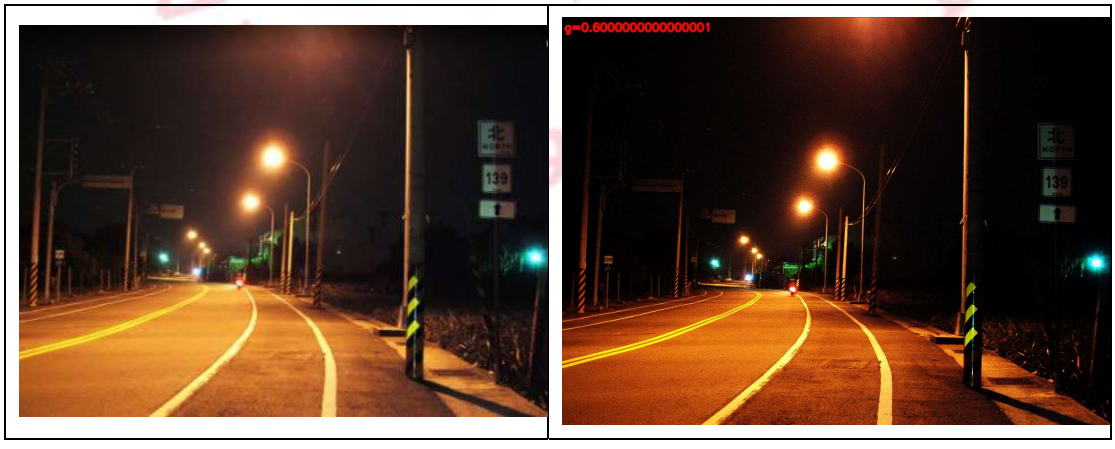
八、Gamma 校正

```
def adjust_gamma(image, gamma=1.0):
        # build a lookup table mapping the pixel values [0, 255] to
        # their adjusted gamma values
        invGamma = 1.0 / gamma
        table = np.array([((i / 255.0) ** invGamma) * 255
                for i in np.arange(0, 256)]).astype("uint8")
        # apply gamma correction using the lookup table
        return cv2.LUT(image, table)
```

可以考慮是否要影像前處理，這裡使用了 Gamma 校正，函式部分就只是建一個 look up table 而已，公式參考圖十二，圖十三可以看到原圖有些部分過亮，可以使用 gamma 校正調暗，在測試影片 2，我設 gamma=0.2。



圖十二、Gamma correction

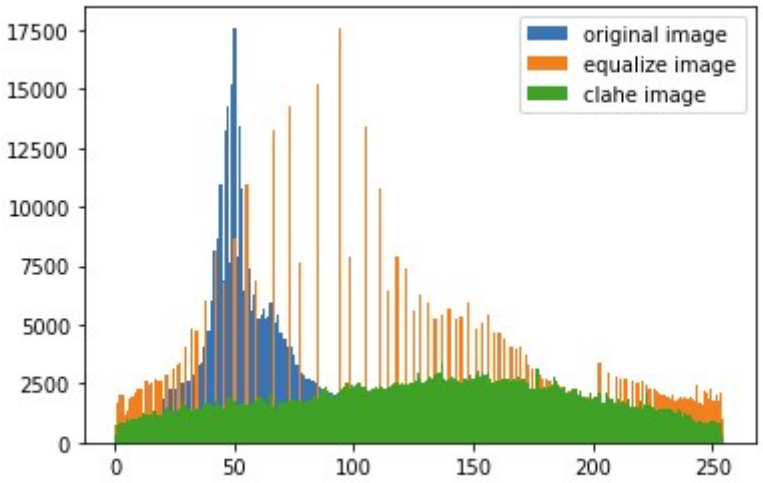


圖十三、原圖與 Gamma=0.6 結果

九、CLAHE (Contrast Limited Adaptive Histogram Equalization)

```
def CLAHE(image, clipLimit=2.0, tileGridSize=(8,8)):
    clahe = cv2.createCLAHE(clipLimit=clipLimit, tileGridSize=tileGridSize)
    cl_img = clahe.apply(image)
    return cl_img
```

因為像素的灰度值很聚集，故可以使用直方圖均衡化，讓灰度可以分散，但直方圖均衡容易受雜訊干擾，故我們就使用 CLAHE，會將圖片切成很多個小塊，OpenCV 預設 tileGridSize=(8,8)，接著對每個小塊作均衡化，接著限制對比度，也就是限制累積分布函數的斜率，表示限制了直方圖的幅度，預設 clipLimit=2.0。



圖十四、CLAHE、HE、原圖的灰度直方圖比較

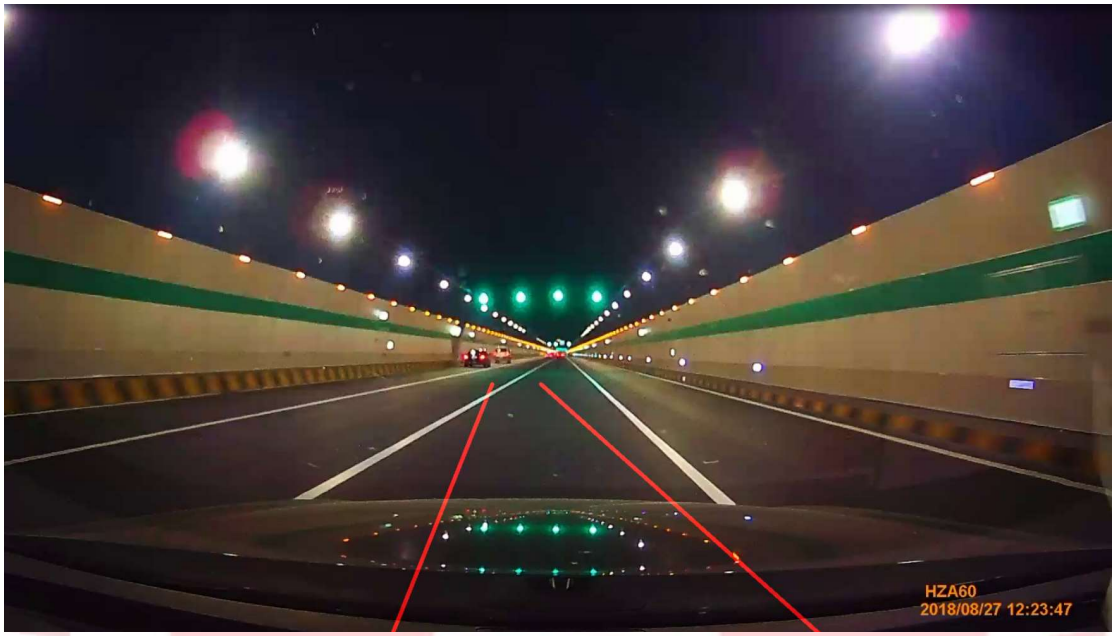
參、結果

結果如下，圖十五與十六為測試檔案夾裡的測試集，可以搭配測試影片 1 觀賞結果 https://www.youtube.com/watch?v=HwoEk8aO8ug&list=LL&index=3。

圖十五、彎白線

圖十六、左黃右白線

倘若不加入影像前處理，那在其他影片就會 Fail，如圖十七。圖十八則是加入了上節的最後兩個方法，Gamma 校正與 CLAHE 的結果，可以搭配測試影片二 https://www.youtube.com/watch?v=Lj3_ciOsvGg&list=LL&index=3 觀賞。



圖十七、Fail

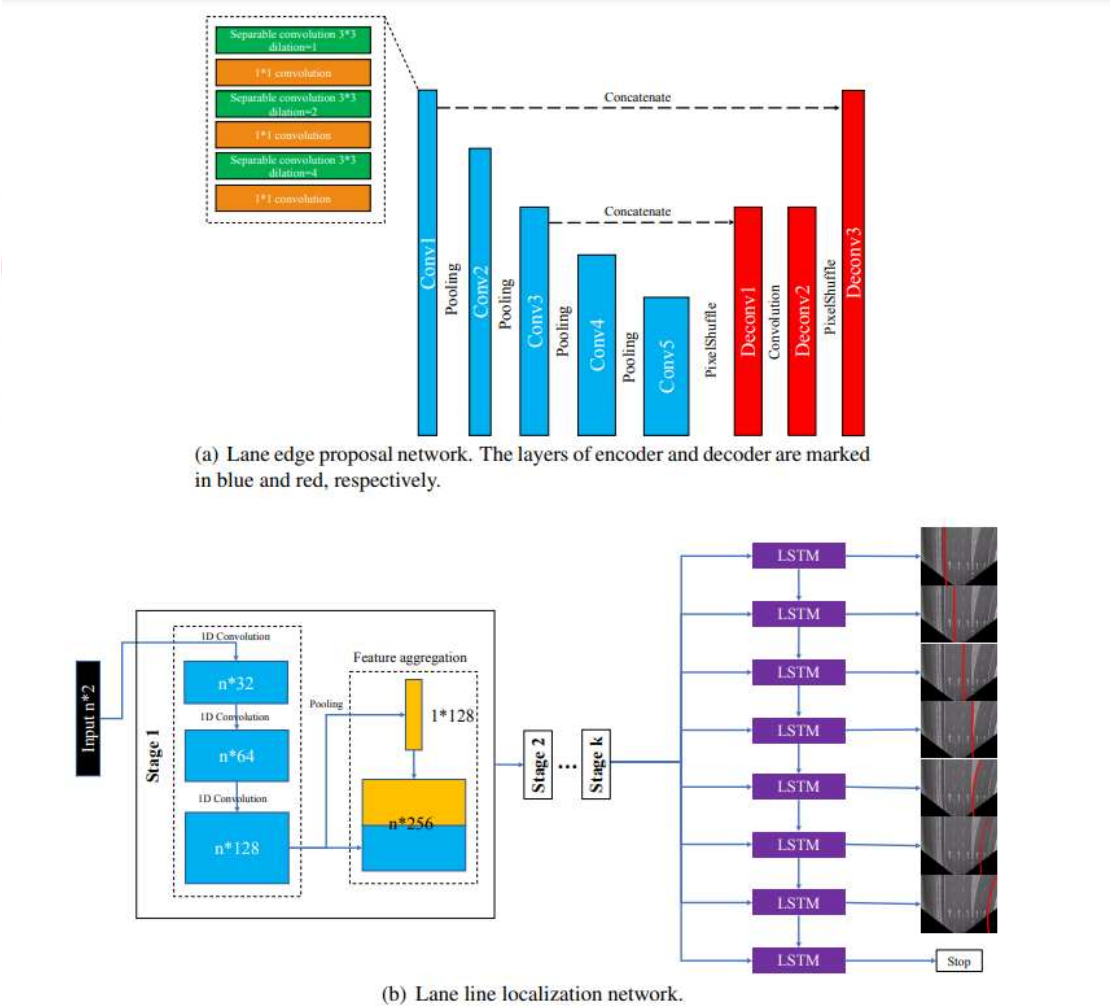
圖十八、加入 Gamma correction 與 CLAHE

肆、心得討論

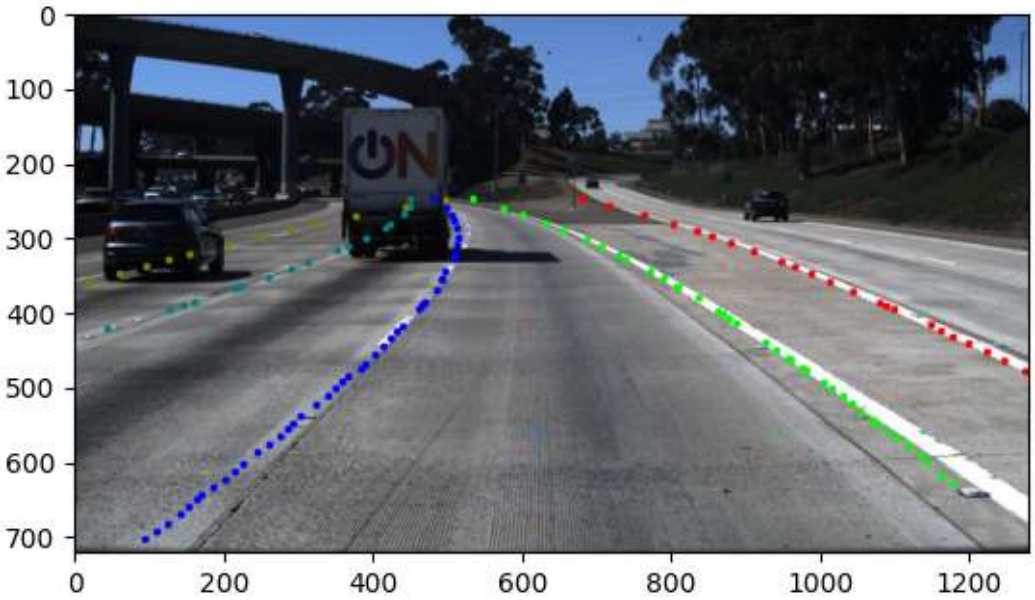
以上許多參數都是需要手動測試的，像是 gamma 要設成多少，霍夫轉換的 threshold、minLineLength、maxLineGap 等等函式，都算是 case by case 的，若是有 illumination variation、blur 等等問題，就需要讓系統更強健。

像是我本身在研究 visual SLAM，要是特徵點太少，feature based 的方式可能就不適合，例如：長廊，可以考慮使用 photometric 的直接法，直接使用像素灰度來優化，倘若像素灰度不變，就能使用 LK 光流，但這不太可能，故此時相機的曝光參數、時間就得考慮進來，才有 DSO 這個 visual odometry。

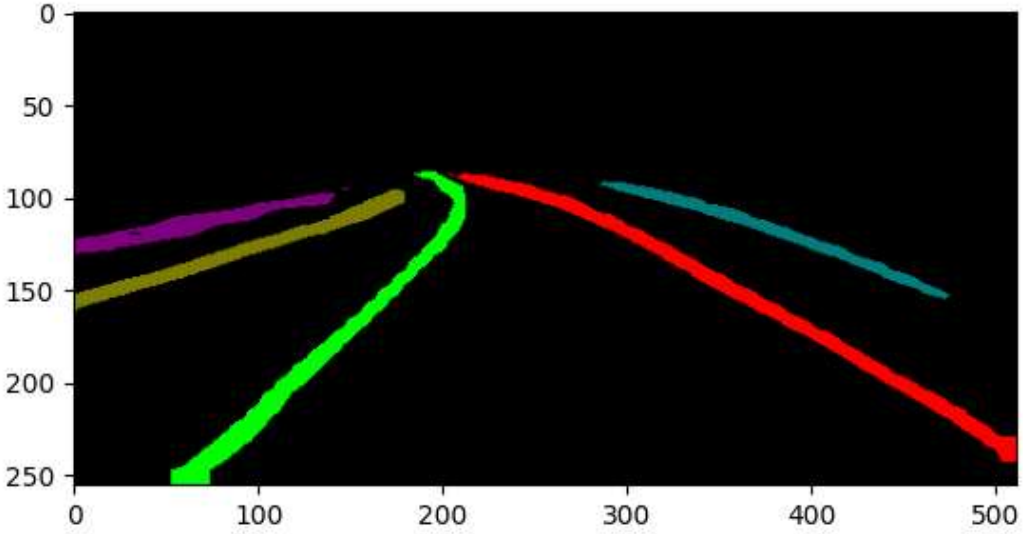
Lane detection 可以使用其他方式，讓結果更 Robust，像是 learning based 的方式，例如：2018 年的 LaneNet [1]，這裡不細講其理論，網路架構如圖十九、結果如圖二十、二十一。



圖十九、LaneNet 網路架構



圖二十、LaneNet 結果



圖二十一、LaneNet 結果，Masked Image

另外我們也能使用 Cammera Parameter Control 的方式達到高品質圖片，在源頭處就處理掉，也就是不要使用 post-processing 的方式處理圖片，而是在相機能調參數的時候就調整，像是如何處理在進入隧道以及出隧道有大變化的光源問題[2]，不使用 Auto Exposue Time 以及 Auto Gain。長曝光容易動態模糊，增益太大導致 Noise 增加，除了有學習的方式可以調參，也可以自行定義一個 Metric，利用傳統方法解出適當參數。

圖二十一、以學習的方式調整 Gain 與 Exposue Time

伍、參考資料

[1] S. M. Azimi, P. Fischer, M. Körner and P. Reinartz, "Aerial LaneNet: Lane-Marking Semantic Segmentation in Aerial Imagery Using Wavelet-Enhanced Cost-Sensitive Symmetric Fully Convolutional Neural Networks," in IEEE Transactions on Geoscience and Remote Sensing, vol. 57, no. 5, pp. 2920-2938, May 2019, doi: 10.1109/TGRS.2018.2878510.

[2] J. Tomasi, B. Wagstaff, S. L. Waslander and J. Kelly, "Learned Camera Gain and Exposure Control for Improved Visual Feature Detection and Matching," in IEEE Robotics and Automation Letters, vol. 6, no. 2, pp. 2028-2035, April 2021, doi: 10.1109/LRA.2021.3058909.