无人驾驶技术入门（十四）
初识图像之初级车道线检测

陈 光

上一期的无人驾驶技术入门，我们以障碍物的跟踪为例，介绍了卡尔曼滤波器的原理、公式和代码的编写。接下来的几期无人驾驶技术入门，我会带大家接触无人驾驶技术的另一个重要的领域——计算机视觉。

在无人驾驶技术入门（五）| 没有视觉传感器，还谈什么无人驾驶？中，我介绍了车载视觉传感器能够实现车道线、障碍物、交通标志牌、可通行空间、交通信号灯的检测等。这些检测结果都离不开计算机视觉技术。

本次分享，我将以优达学城(Udacity)无人驾驶工程师学位中提供的初级车道线检测项目为例，对课程中使用到的计算机视觉技术进行分享。分享内容包括OpenCV 库的基本使用，以及车道线检测中所用到的计算机视觉技术，包括其基本原理和使用效果，以帮助大家由浅入深地了解计算机视觉技术。

在介绍计算机视觉技术前，我想先讨论一下这次分享的输入和输出。

- **输入**

一张摄像机拍摄到的道路图片，图片中需要包含车道线。如下图所示。

图片出处：

https://github.com/udacity/CarND-LaneLines-P1/blob/master/test_images/whiteCarLaneSwitch.jpg

- **输出**

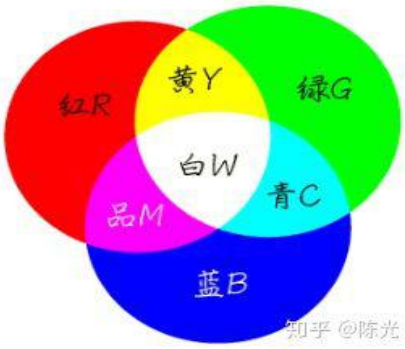
图像坐标系下的左右车道线的直线方程和有效距离。将左右车道线的方程绘制到原始图像上，应如下图所示。

输出结果

在输入和输出都定义清楚后，我们就开始使用计算机视觉技术，一步步完成对原始图像的处理。

1 原始图像

认识图像前，我们需要先回顾一下在初中所学的物理知识——光的三原色，光的三原色分别是红色(Red)、绿色(Green)和蓝色(Blue)。通过不同比例的三原色组合形成不同的可见光色。如下图所示。

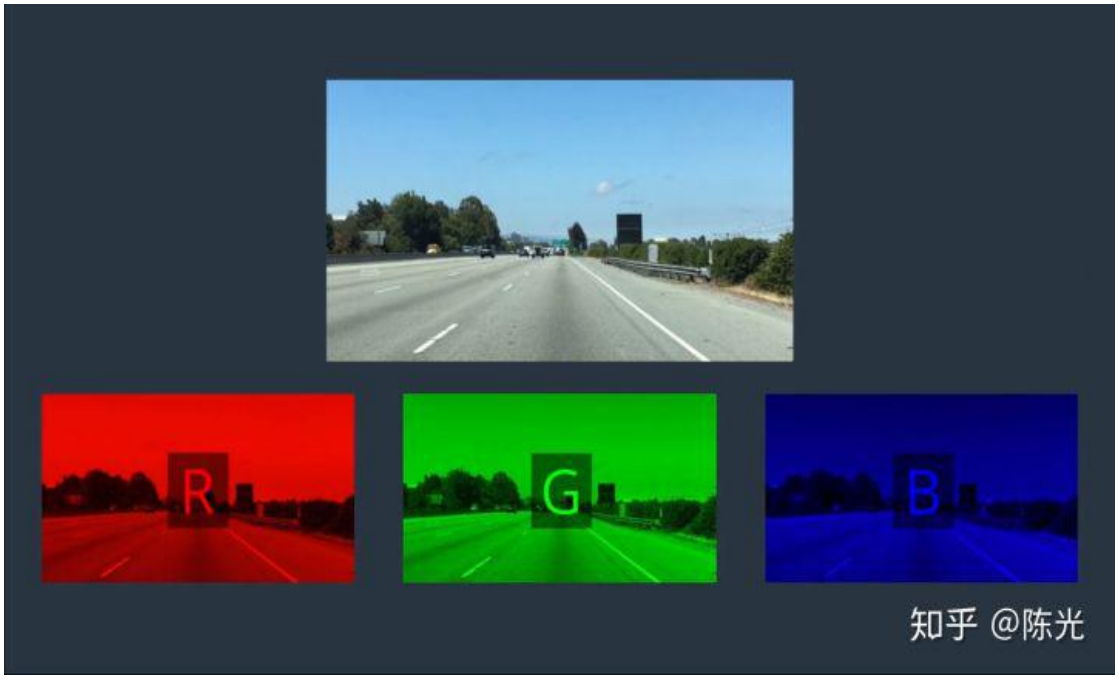


图片出处：https://zhidao.baidu.com/question/197911511.html

图像中的每个像素点都是由 RGB（红绿蓝）三个颜色通道组成。为了方便描述 RGB 颜色模型，在计算机中约束了每个通道由暗到亮的范围是 0~255。

当某个像素点的 R 通道数值为 255，G 和 B 通道数值为 0 时，实际表现出的颜色就是最亮的红色；当某个像素点的 RGB 三通道都为 255 时，所表示的是最亮的白色；当某个像素点的 RGB 三通道都为 0 时，就会显示最暗的黑色。在 RGB 颜色模型中，不会有比[255,255,255]的组合更亮的颜色了。

根据以上理论基础，一幅彩色图像，其实就是由三幅单通道的图像叠加，如下图所示。



图片出处：优达学城(Udacity)无人驾驶工程师学位

以基于 python 的 OpenCV 为例，读取名为 test_img.jpg 的图片到计算机内存中的代码如下：

```
import cv2
img = cv2.imread('image_name.jpg')
```

读取图像后，我们可以将图像看做一个二维数组，每个数组元素中存了三个值，分别是 RGB 三个通道所对应的数值。

OpenCV 定义了，图像的原点(0，0)在图片的左上角，横轴为 X，朝右，纵轴为 Y，朝下，如下图所示。

原始图像

需要注意的是，由于 OpenCV 的早期开发者习惯于使用 BGR 顺序的颜色模型，因此使用 OpenCV 的 imread()读到的像素，每个像素的排列是按 BGR，而不是常见的 RGB，代码编写时需要注意。

2 灰度处理

考虑到处理三个通道的数据比较复杂，我们先将图像进行灰度化处理，灰度化的过程就是将每个像素点的 RGB 值统一成同一个值。灰度化后的图像将由三通道变为单通道，单通道的数据处理起来就会简单许多。

通常这个值是根据 RGB 三通道的数值进行加权计算得到。人眼对 RGB 颜色的敏感度不同，对绿色最敏感，权值较高，对蓝色最不敏感，权值较低。坐标为(x,y)的像素点进行灰度化操作的具体计算公式如下：

$$Gray(x,y) = 0.299 * Red(x,y) + 0.587 * Green(x,y) + 0.114 * Blue(x,y)$$

图像灰度处理计算公式

调用 OpenCV 中提供的 cvtColor()函数，能够方便地对图像进行灰度处理。

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

由于使用 cv2.imread()读到的 img 的数据排列为 BGR，因此这里的参数为 BGR2GRAY

灰度处理后的图像如下图所示：

灰度处理

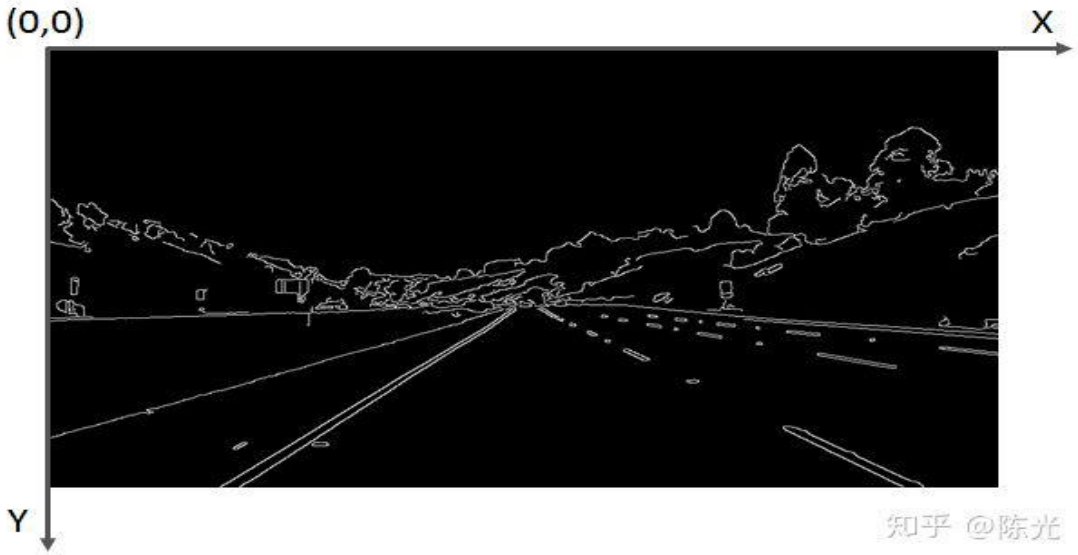
3 边缘提取

为了突出车道线，我们对灰度化后的图像做边缘处理。“边缘”就是图像中明暗交替较为明显的区域。车道线通常为白色或黄色，地面通常为灰色或黑色，因此车道线的边缘处会有很明显的明暗交替。

常用的边缘提取算法有 Canny 算法和 Sobel 算法，它们只是计算方式不同，但实现的功能类似。可以根据实际要处理的图像，选择算法。哪种算法达到的效果更好，就选哪种。

以 Canny 算法为例，选取特定的阈值后，对灰度图像进行处理，即可得到的边缘提取的效果图。

```
low_threshold = 40
high_threshold = 150
canny_image = cv2.Canny(gray, low_threshold, high_threshold)
```

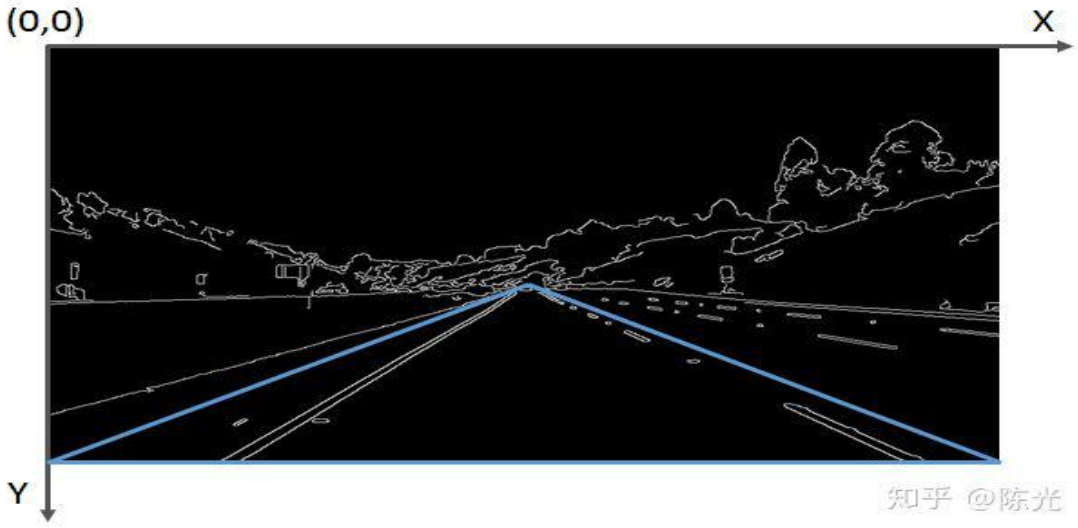


Canny 边缘提取

4 感兴趣区域选择

边缘提取完成后，需要检测的车道线被凸显出来了。为了实现自车所在车道的车道线检测，我们需要将感兴趣的区域(Region of Interest)提取出来。提取感兴趣区域最简单的方式就是“截取”。

首先选定一个感兴趣区域，比如下图所示的蓝色三角形区域。对每个像素点的坐标值进行遍历，如果发现当前点的坐标不在三角区域内，则将该点涂“黑”，即将该点的像素值置为 0。



感兴趣区域选定

为了实现截取功能，可以封装一下 OpenCV 的部分函数，定义一个 region_of_interest 函数：

```
def region_of_interest(img, vertices):
    #定义一个和输入图像同样大小的全黑图像mask，这个mask也称掩膜
    #掩膜的介绍，可参考：https://www.cnblogs.com/skyfsm/p/6894685.html
    mask = np.zeros_like(img)

    #根据输入图像的通道数，忽略的像素点是多通道的白色，还是单通道的白色
    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


    #[vertices]中的点组成了多边形，将在多边形内的mask像素点保留，
    cv2.fillPoly(mask, [vertices], ignore_mask_color)

    #与mask做"与"操作，即仅留下多边形部分的图像
    masked_image = cv2.bitwise_and(img, mask)
```

```
    return masked_image
```

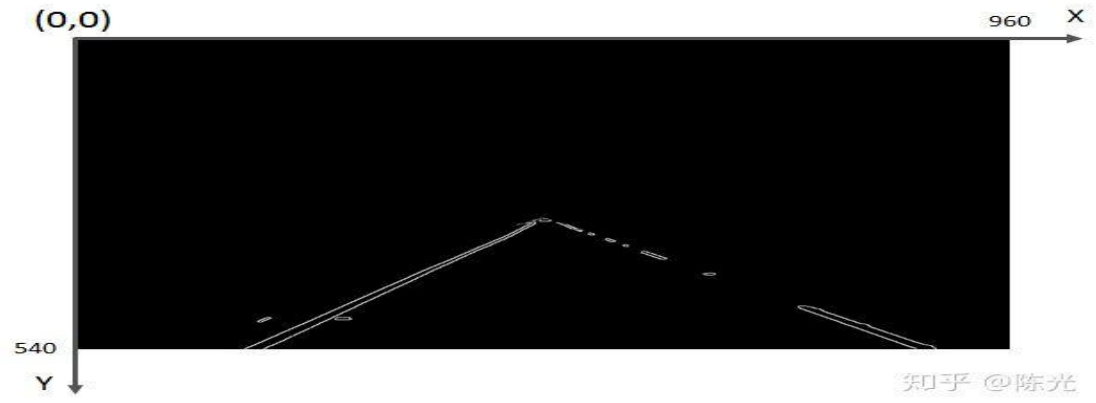
源码出自：

https://link.zhihu.com/?target=https%3A//github.com/udacity/CarND-LaneLines-P1/blob/master/P1.ipynb

封装完函数后，我们将感兴趣的区域输入，实现边缘提取后的图像的截取。

```
#图像像素行数 rows = canny_image .shape[0]  540行
#图像像素列数 cols = canny_image .shape[1]  960列
left_bottom = [0, canny_image .shape[0]]
right_bottom = [canny_image .shape[1], canny_image .shape[0]]
apex = [canny_image .shape[1]/2, 310]
vertices = np.array([ left_bottom, right_bottom, apex ], np.int32)
roi_image = region_of_interest(canny_image, vertices)
```

截取后的图像入下图所示：



感兴趣区域截取

5 霍夫变换

经过灰度处理、边缘检测、感兴趣区域截取后，我们终于将左右车道线从复杂的图像中提取出来了。接下来，我们使用霍夫变换来提取图像中的直线（段）。

霍夫变换是一种特征检测方法，其原理和推导过程可以参看经典霍夫变换(Hough Transform)。

在图像中使用霍夫变换不仅能够识别图像中的直线，还能识别出图像中的圆、椭圆等特征。

OpenCV 为我们提供了霍夫变换检测直线的函数，可以通过设置不同的参数，检测不同长度的线段。由于车道线存在虚线的可能，因此线段的检测长度不能设置地太长，否则短线段会被忽略掉。

OpenCV 的霍夫变换直线检测函数使用方法如下：

```
rho = 2 # distance resolution in pixels of the Hough grid
theta = np.pi/180 # angular resolution in radians of the Hough grid
threshold = 15     # minimum number of votes (intersections in Hough grid cell)
min_line_length = 40 #minimum number of pixels making up a line
max_line_gap = 20    # maximum gap in pixels between connectable line segments
# Hough Transform 检测线段，线段两个端点的坐标存在lines中
lines = cv2.HoughLinesP(roi_image, rho, theta, threshold, np.array([]),
                        min_line_length, max_line_gap)
```

封装一个绘图函数，实现把线段绘制在图像上的功能，以实现线段的可视化。

```
def draw_lines(img, lines, color=[255, 0, 0], thickness=2):
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness) # 将线段绘制在img上
```

将得到线段绘制在原始图像上。

```
import numpy as np
line_image = np.copy(img) # 复制一份原图，将线段绘制在这幅图上
draw_lines(line_image, lines, [255, 0, 0], 6)
```

结果如下图：

霍夫变换直线检测

可以看出，虽然右车道线的线段不连续，但已经很接近我们想要的输出结果了。

6 数据后处理

霍夫变换得到的一系列线段结果跟我们的输出结果还是有些差异。为了解决这些差异，需要对我们检测到的数据做一定的后处理操作。

实现以下两步后处理，才能真正得到我们的输出结果。

1. 计算左右车道线的直线方程

根据每个线段在图像坐标系下的斜率，判断线段为左车道线还是右车道线，并存于不同的变量中。随后对所有左车道线上的点、所有右车道线上的点做一次最小二乘直线拟合，得到的即为最终的左、右车道线的直线方程。

2. 计算左右车道线的上下边界

考虑到现实世界中左右车道线一般都是平行的，所以可以认为左右车道线上最上和最下的点对应的 y 值，就是左右车道线的边界。

基于以上两步数据后处理的思路，我们重新定义 draw_lines()函数，将数据后处理过程写入该函数中。

```
def draw_lines(img, lines, color=[255, 0, 0], thickness=2):
    left_lines_x = []
    left_lines_y = []
    right_lines_x = []
    right_lines_y = []
    line_y_max = 0
```

```
    line_y_min = 999
    for line in lines:
        for x1,y1,x2,y2 in line:
            if y1 > line_y_max:
                line_y_max = y1
            if y2 > line_y_max:
                line_y_max = y2
            if y1 < line_y_min:
                line_y_min = y1
            if y2 < line_y_min:
                line_y_min = y2
            k = (y2 - y1)/(x2 - x1)
            if k < -0.3:
                left_lines_x.append(x1)
                left_lines_y.append(y1)
                left_lines_x.append(x2)
                left_lines_y.append(y2)
            elif k > 0.3:
                right_lines_x.append(x1)
                right_lines_y.append(y1)
                right_lines_x.append(x2)
                right_lines_y.append(y2)
    #最小二乘直线拟合
    left_line_k, left_line_b = np.polyfit(left_lines_x, left_lines_y, 1)
    right_line_k, right_line_b = np.polyfit(right_lines_x, right_lines_y, 1)

    #根据直线方程和最大、最小的y值反算对应的x
    cv2.line(img,
             (int((line_y_max - left_line_b)/left_line_k), line_y_max),
             (int((line_y_min - left_line_b)/left_line_k), line_y_min),
             color, thickness)
    cv2.line(img,
             (int((line_y_max - right_line_b)/right_line_k), line_y_max),
             (int((line_y_min - right_line_b)/right_line_k), line_y_min),
             color, thickness)
```

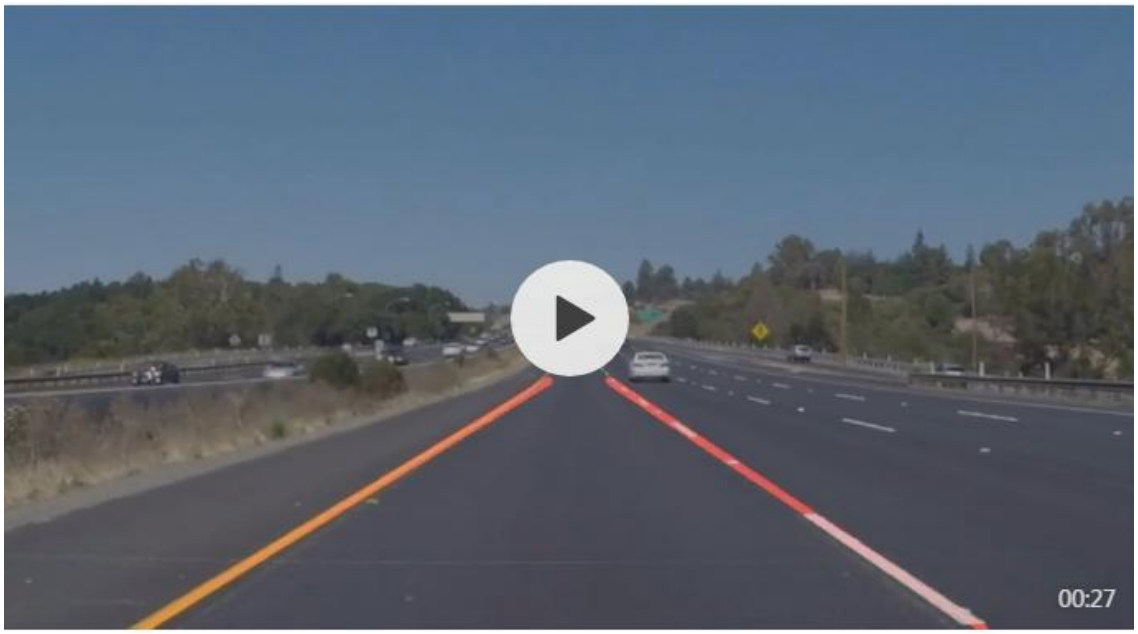
根据对线段的后处理，即可得到符合输出要求的两条直线方程的斜率、截距和有效长度。将后处理后的结果绘制在原图上，如下图所示：

数据后处理

7 处理视频

视频其实就是一帧帧连续不断的图像，使用读取视频的库，将视频截取成一帧帧图像，然后使用上面的灰度处理、边缘提取、感兴趣区域选择、霍夫变换和数据后处理，得到车道线检测结果，再将图片结果拼接成视频，就完成了视频中的车道线检测。

利用前文提到的车道线检测算法得到的视频处理结果如下：



视频地址见原文：https://zhuanlan.zhihu.com/p/52623916

无人驾驶技术入门之车道线检测。

由视频可以看出，当汽车在下坡时，车头会发生俯仰，造成感兴趣区域的变化，因此检测到的有效长度有所变化。可见本算法需要针对车辆颠簸的场景进行优化。

8 结语

以上就是《初识图像之初级车道线检测》的全部内容，关于这个项目的全部内容，可以在优达学城(Udacity)无人驾驶工程师学位首页试听，建议读者亲身学习一遍。

在实际编写车道线检测代码的过程中，你会发现，每一步都需要调很多参数，

才能满足后续算法的处理要求。可见，本算法无法应用在不同光照条件的场景中，鲁棒性较差；同时，由于霍夫变换检测直线本身的缺陷，面对弯道场景时，无法很好地将弯道检测出来。

所以，本算法设计并不完善。为了设计出一套能够适应更多场景的车道线检测算法，需要使用更多高级的算法，这些内容将会在下期的《高级车道线检测》中做介绍。

好了\(^o^)/~，这篇分享就到这啦，我们下期见~

本文原载：知乎号“陈光”，作者授权转载。

临菲信息技术港

临菲信息技术港公众号

临菲学堂