无人驾驶技术入门（十五）
再识图像之高级车道线检测

陈 光

上一期的《无人驾驶技术入门（十四）| 初识图像之初级车道线检测》，我对计算机视觉技术的一些基本理论和 OpenCV 库的使用进行了分享。这些技术包括图像的灰度处理、边缘提取算法、霍夫变换直线检测算法以及数据的后处理方法。应用这些技术后，可以将图像中近似直线的车道线检测出来。检测的效果如下图红线所示。

在上一期的末尾，我们从边缘提取算法和霍夫变换直线检测的原理上，讨论了为什么以上算法无法处理光照条件变化剧烈和弯道的场景。

在本次分享中，我将以优达学城（Udacity）无人驾驶工程师学位中提供的高级车道线检测项目为例，介绍普适性更好，且更为鲁棒的车道线检测技术，用于处理那些无人驾驶中常见的（如路面颜色变化、路边障碍物阴影等导致的）光线变化剧烈和弯道的场景。

按照惯例，在介绍计算机视觉技术前，我们先讨论一下这次分享的输入和输出。

- **输入**

一个连续的视频，视频中的左车道线为黄色实线，右车道线为白色虚线。无人车会经过路面颜色突变、路边树木影子干扰、车道线不清晰和急转弯的路况。（视频大小 7.25M）

视频出处：

https://link.zhihu.com/?target=https%3A//github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/project_video.mp4

- **输出**

左、右车道线的三次曲线方程，及其有效距离。最后将车道线围成的区域显示在图像上，如下图所示。

输入和输出都定义清楚后，我们开始探讨高级车道线检测的算法，并对每帧视频图像中的车道线进行检测。

1 摄像机标定

相信大家都多少听说过鱼眼相机，最常见的鱼眼相机是辅助驾驶员倒车的后向摄像头。也有很多摄影爱好者会使用鱼眼相机拍摄图像，最终会有高大上的大片效果，如下图所示。

图片来源：优达学城(Udacity)无人驾驶工程师课程

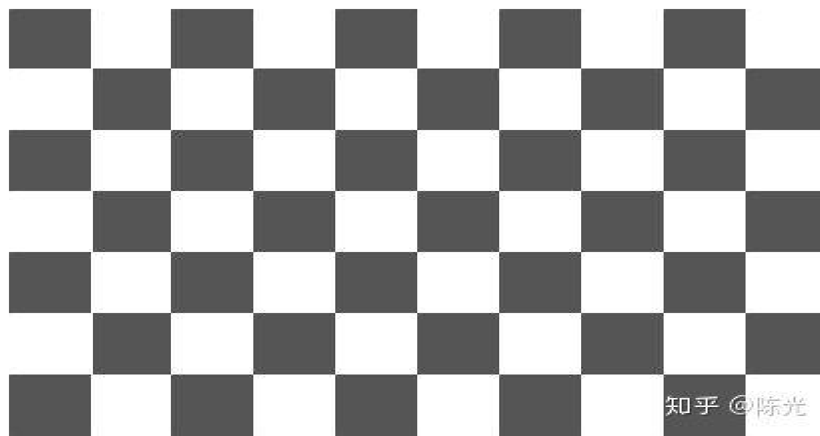
使用鱼眼相机拍摄的图像虽然高大上，但存在一个很大的问题——畸变(Distortion)。如上图所示，走道上的栏杆应该是笔直延伸出去的。然而，栏杆在图像上的成像却是弯曲的，这就是图像畸变，畸变会导致图像失真。

使用车载摄像机拍摄出的图像，虽然没有鱼眼相机的畸变这么夸张，但是畸变是客观存在的，只是人眼难以察觉。使用有畸变的图像做车道线的检测，检测结果的精度将会受到影响，因此进行图像处理的第一步工作就是去畸变。

为了解决车载摄像机图像的畸变问题，摄像机标定技术应运而生。

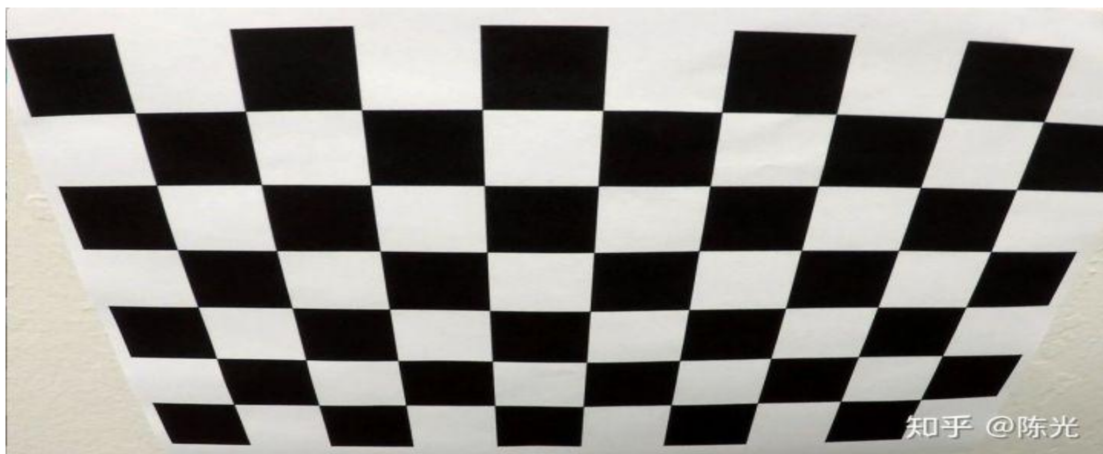
摄像机标定是通过对已知的形状进行拍照，通过计算该形状在真实世界中位置与在图像中位置的偏差量（畸变系数），进而用这个偏差量去修正其他畸变图像的技术。

原则上，可以选用任何的已知形状去校准摄像机，不过业内的标定方法都是基于棋盘格的。因为它具备规则的、高对比度图案，能非常方便地自动化检测各个棋盘格的交点，十分适合标定摄像机的标定工作。如下图所示为标准的 10x7（7 行 10 列）的棋盘格。



OpenCV 库为摄像机标定提供了函数 cv2.findChessboardCorners()，它能自动地检测棋盘格内 4 个棋盘格的交点（2 白 2 黑的交接点）。我们只需要输入摄像机拍摄的完整棋盘格图像和交点在横纵向上的数量即可。随后我们可以使用函数 cv2.drawChessboardCorners()绘制出检测的结果。

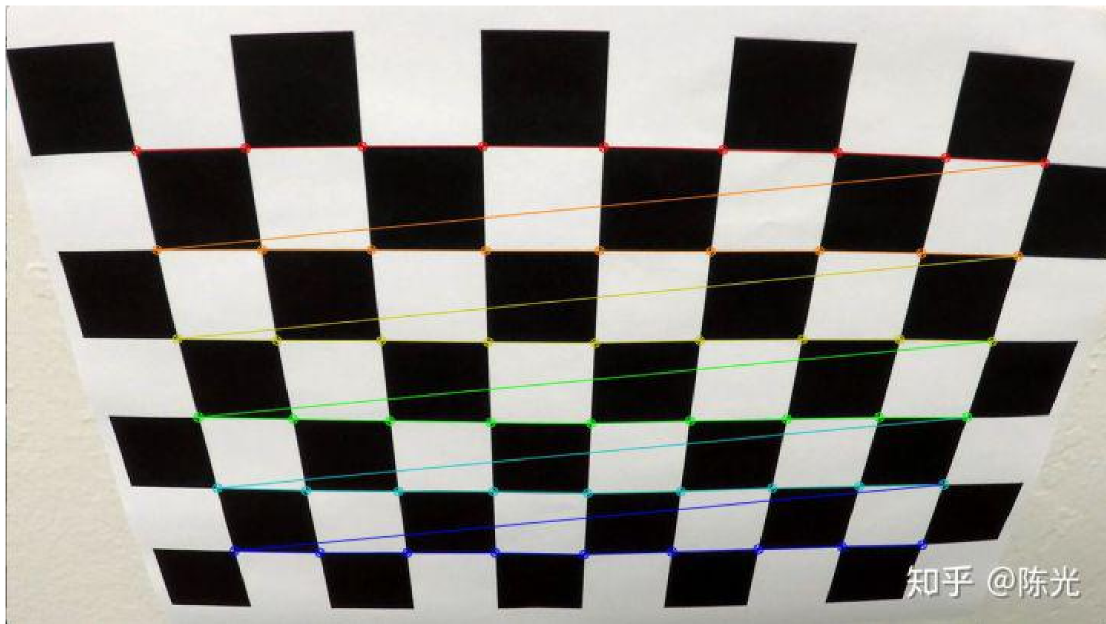
棋盘格原图如下所示：



图片出处：

https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/camera_cal/calibration2.jpg

使用 OpenCV 自动交点检测的结果如下：



获取交点的检测结果后，使用函数 cv2.calibrateCamera()即可得到相机的畸变系数。

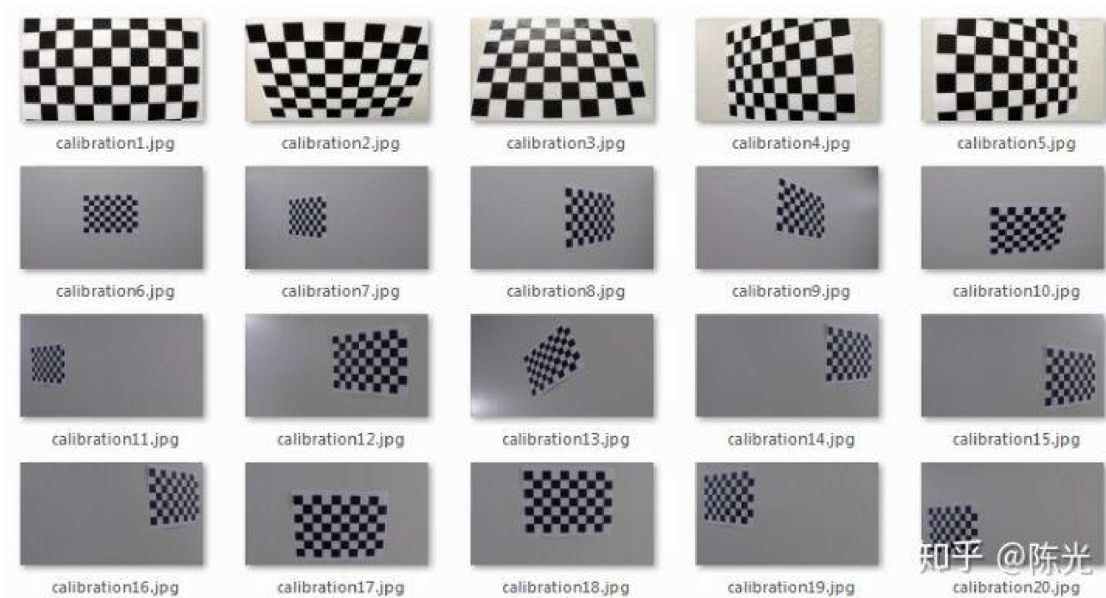
为了使摄像机标定得到的畸变系数更加准确，我们使用车载摄像机从不同的角度拍摄 20 张棋盘格，将所有的交点检测结果保存，再进行畸变系数的的计算。

我们将读入图片、预处理图片、检测交点、标定相机的一系列操作，封装成一个函数，如下所示：

```
#################################################################
# Step 1 : Calculate camera distortion coefficients
#################################################################
def getCameraCalibrationCoefficients(chessboardname, nx, ny):
    # prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
    objp = np.zeros((ny * nx, 3), np.float32)
    objp[:,:2] = np.mgrid[0:nx, 0:ny].T.reshape(-1,2)

    # Arrays to store object points and image points from all the images.
    objpoints = [] # 3d points in real world space
    imgpoints = [] # 2d points in image plane.

    images = glob.glob(chessboardname)
    if len(images) > 0:
        print("images num for calibration : ", len(images))
    else:
        print("No image for calibration.")
        return

    ret_count = 0
    for idx, fname in enumerate(images):
```

```
        img = cv2.imread(fname)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        img_size = (img.shape[1], img.shape[0])
        # Finde the chessboard corners
        ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)

        # If found, add object points, image points
        if ret == True:
            ret_count += 1
            objpoints.append(objp)
            imgpoints.append(corners)

    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size
    print('Do calibration successfully')
    return ret, mtx, dist, rvecs, tvecs
```

需要标定的一系列图片如下图所示：



图片出处：

https://github.com/udacity/CarND-Advanced-Lane-Lines/tree/master/camera_cal

调用之前封装好的函数，获取畸变参数。

```
nx = 9
ny = 6
ret, mtx, dist, rvecs, tvecs = getCameraCalibrationCoefficients('camera_cal/calibrati
```

随后，使用 OpenCV 提供的函数 cv2.undistort()，传入刚刚计算得到的畸变参

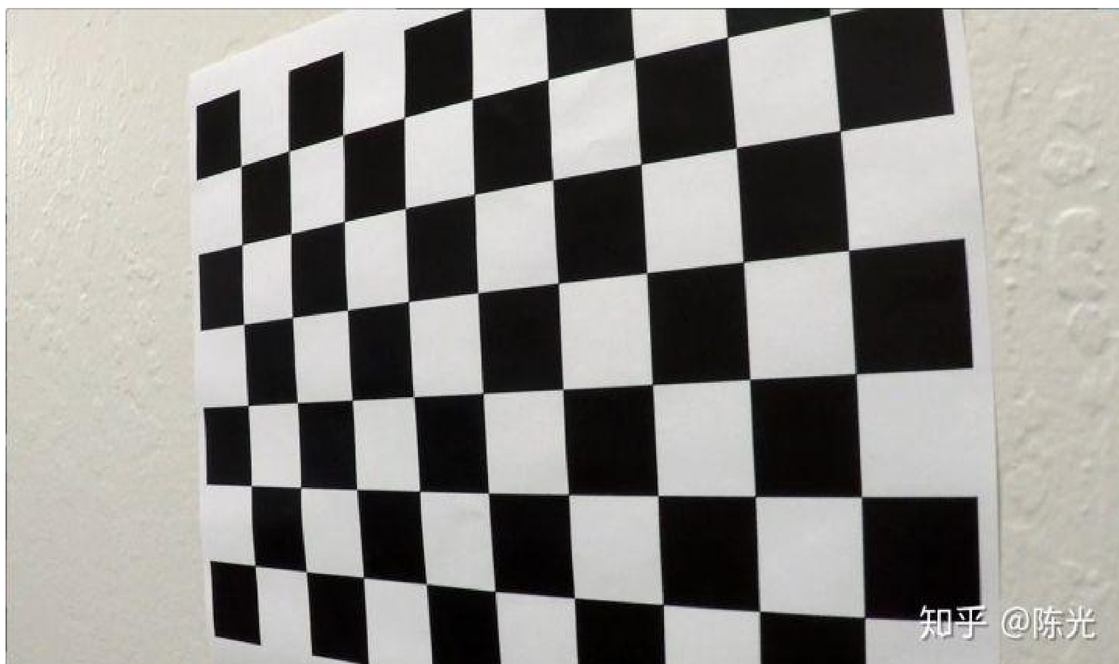
数，即可将畸变的图像进行畸变修正处理。

```
########################################################################
# Step 2 : Undistort image
########################################################################
def undistortImage(distortImage, mtx, dist):
    return cv2.undistort(distortImage, mtx, dist, None, mtx)
```

以畸变的棋盘格图像为例，进行畸变修正处理。

```
# Read distorted chessboard image
test_distort_image = cv2.imread('./camera_cal/calibration4.jpg')

# Do undistortion
test_undistort_image = undistortImage(test_distort_image, mtx, dist)
```

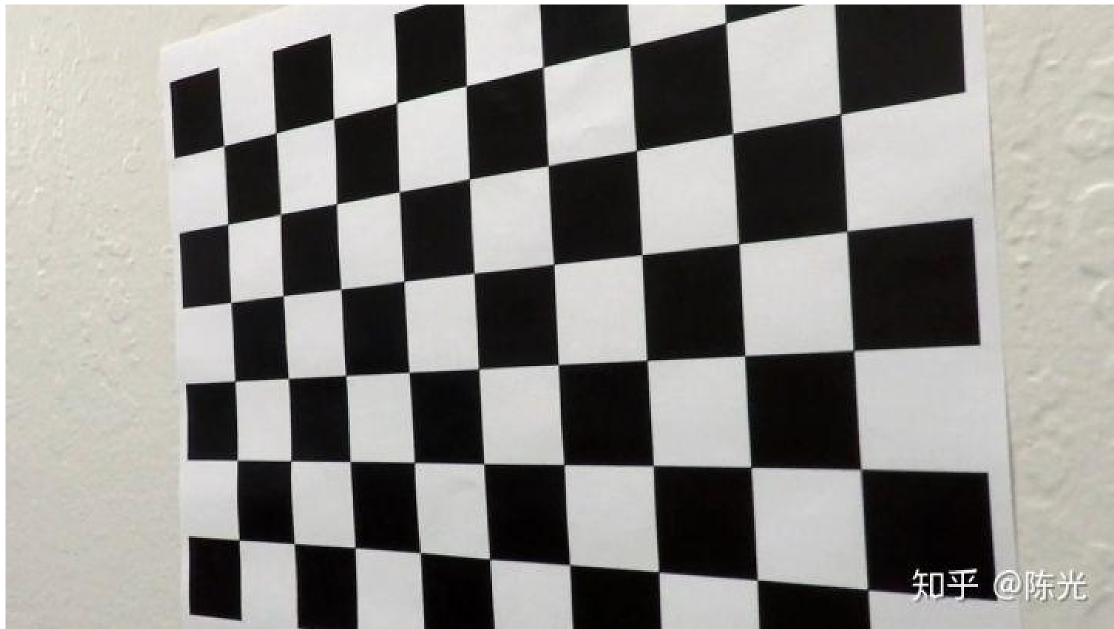
畸变图像如下图所示：



图像出处：

https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/camera_cal/calibration4.jpg

复原后的图像如下图所示：



同理，我们将摄像机拍摄到的实际路况进行畸变修正处理。

```
test_distort_image = cv2.imread('test_images/straight_lines1.jpg')

# Do undistortion
test_undistort_image = undistortImage(test_distort_image, mtx, dist)
```

原始畸变图像如下所示：

图片出处：

https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/test_images/straight_lines1.jpg

畸变修正后的图像如下所示：

可以看到离镜头更近的左侧、右侧和下侧的图像比远处的畸变修正更明显。

2 筛选图像

从我们作为输入的视频可以看出，车辆会经历颠簸、车道线不清晰、路面颜色突变，路边障碍物阴影干扰等复杂工况。因此，需要将这些复杂的场景筛选出来，确保后续的算法能够在这些复杂场景中正确地检测出车道线。

使用以下代码将视频中的图像数据提取，进行畸变修正处理后，存储在名为 original_image 的文件夹中，以供挑选。

```
video_input = 'project_video.mp4'
cap = cv2.VideoCapture(video_input)
count = 1
while(True):
    ret, image = cap.read()
    if ret:
        undistort_image = undistortImage(image, mtx, dist)
        cv2.imwrite('original_image/' + str(count) + '.jpg', undistort_image)
        count += 1
    else:
        break
cap.release()
```

在 original_image 文件夹中，挑选出以下 6 个场景进行检测。这 6 个场景既

包含了视频中常见的正常直道、正常弯道工况，也包含了具有挑战性的阴影、明暗剧烈变化的工况。如下图所示：

无阴影、颜色无明暗变换的直道

无阴影、颜色无明暗变换的弯道

有小面积阴影、颜色由暗到亮的直道

无阴影、道路标志线不清晰的弯道

有大面积阴影、颜色由暗到亮的弯道

有大面积阴影、颜色由亮到暗的弯道

如果后续的高级车道线检测算法能够完美处理以上六种工况，那将算法应用到视频中，也会得到完美的车道线检测效果。

3 透视变换

在完成图像的畸变修正后，就要将注意力转移到车道线。与《无人驾驶技术入门（十四）| 初识图像之初级车道线检测》中技术类似，这里需要定义一个感兴趣区域。很显然，我们的感兴趣区域就是车辆正前方的这个车道。为了获取感兴趣区域，我们需要对自车正前方的道路使用一种叫做透视变换的技术。

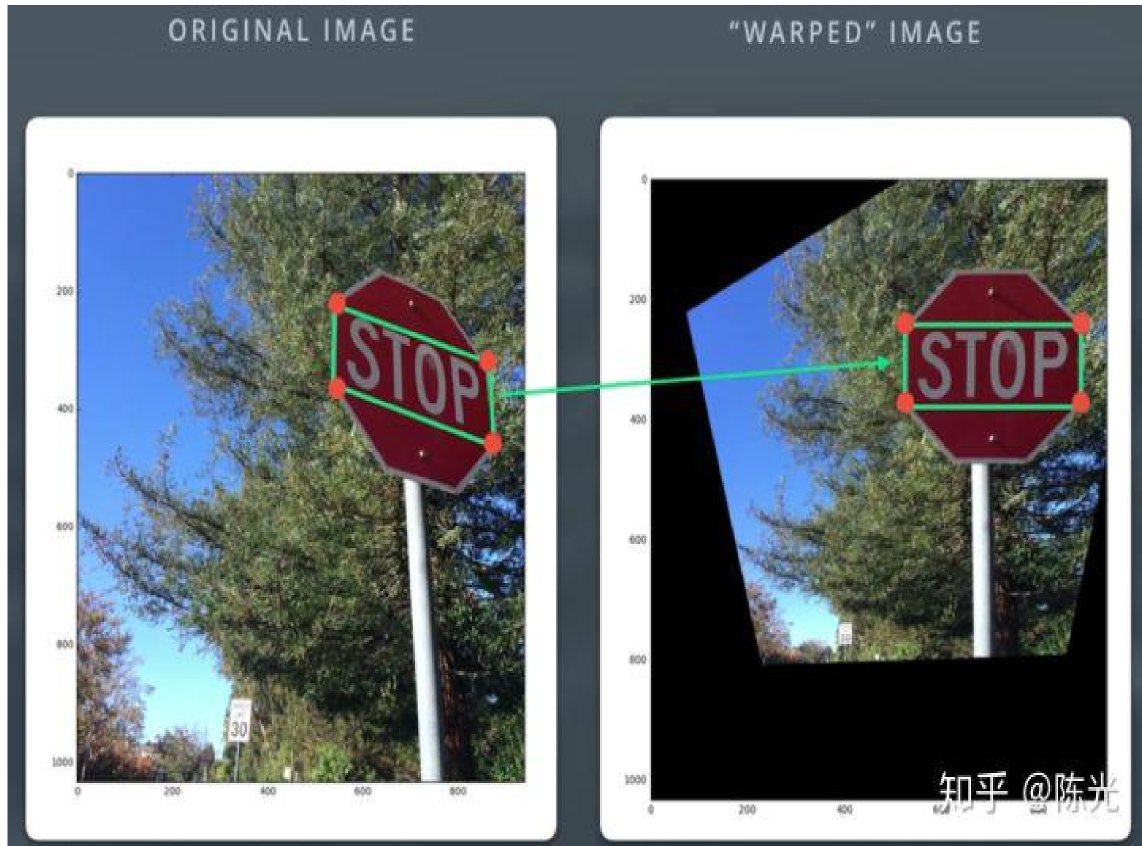
“透视”是图像成像时，物体距离摄像机越远，看起来越小的一种现象。在真实世界中，左右互相平行的车道线，会在图像的最远处交汇成一个点。这个现象就是“透视成像”的原理造成的。

以立在路边的交通标志牌为例，它在摄像机所拍摄的图像中的成像结果一般如下下图所示：

图片来源：优达学城(Udacity)无人驾驶工程师课程

在这幅图像上，原本应该是正八边形的标志牌，成像成为一个不规则的八边形。

通过使用透视变换技术，可以将不规则的八边形投影成规则的正八边形。应用透视变换后的结果对比如吐下：



透视变换的原理：首先新建一幅跟左图同样大小的右图，随后在做图中选择标志牌位于两侧的四个点（如图中的红点），记录这 4 个点的坐标，我们称这 4 个点为 src_points。图中的 4 个点组成的是一个平行四边形。

由先验知识可知，左图中 4 个点所围成的平行四边形，在现实世界中是一个长方形，因此在右边的图中，选择一个合适的位置，选择一个长方形区域，这个长方形的 4 个端点一一对应着原图中的 src_points，我们称新的这 4 个点为 dst_points。

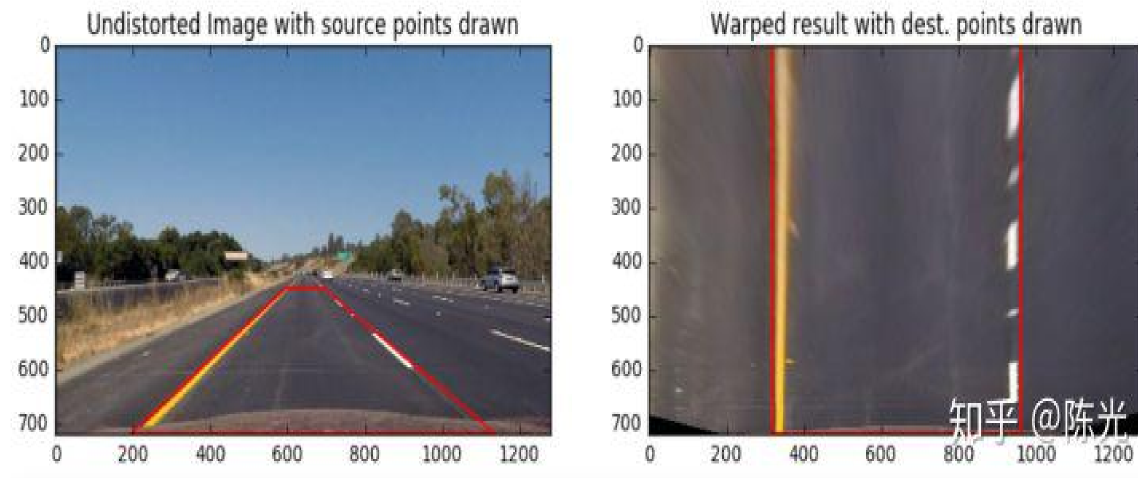
得到 src_points，dst_points 后，我们就可以使用 OpenCV 中计算投影矩阵的函数 cv2.getPerspectiveTransform(src_points, dst_points) 算出 src_points 到 dst_points 的投影矩阵和投影变换后的图像了。

使用 OpenCV 库实现透视变换的代码如下：

```
######################################################################
# Step 3 : Warp image based on src_points and dst_points
######################################################################
# The type of src_points & dst_points should be like
# np.float32([ [0,0], [100,200], [200, 300], [300,400]])
def warpImage(image, src_points, dst_points):
    image_size = (image.shape[1], image.shape[0])
    # rows = img.shape[0] 720
    # cols = img.shape[1] 1280
    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    warped_image = cv2.warpPerspective(image, M,image_size, flags=cv2.INTER_LINEAR)

    return warped_image, M, Minv
```

同理，对于畸变修正过的道路图像，我们同样使用相同的方法，将我们感兴趣的区域做透视变换。

如下图所示，我们选用一张在直线道路上行驶的图像，沿着左右车道线的边缘，选择一个梯形区域，这个区域在真实的道路中应该是一个长方形，因此我们选择将这个梯形区域投影成为一个长方形，在右图横坐标的合适位置设置长方形的 4 个端点。最终的投影结果就像“鸟瞰图”一样。



图片出处：

https://github.com/udacity/CarND-Advanced-Lane-Lines/tree/master/examples/warped_straight_lines.jpg

使用以下代码，通过不断调整 src 和 dst 的值，确保在直线道路上，能够调试出满意的透视变换图像。

```
test_distort_image = cv2.imread('test_images/test4.jpg')

# 畸变修正
test_undistort_image = undistortImage(test_distort_image, mtx, dist)

# 左图梯形区域的四个端点
src = np.float32([[580, 460], [700, 460], [1096, 720], [200, 720]])
# 右图矩形区域的四个端点
dst = np.float32([[300, 0], [950, 0], [950, 720], [300, 720]])

test_warp_image, M, Minv = warpImage(test_undistort_image, src, dst)
```

最终，我们把筛选出的 6 幅图统一应用调整好的 src、dst 做透视变换，结果如下：

无阴影、颜色无明暗变换的直道

无阴影、颜色无明暗变换的弯道

有小面积阴影、颜色由暗到亮的直道

无阴影、道路标志线不清晰的弯道

有大面积阴影、颜色由暗到亮的弯道

有大面积阴影、颜色由亮到暗的弯道

可以看到，越靠图片下方的图像越清晰，越上方的图像越模糊。这是因为越远的地方，左图中的像素点越少。而无论是远处还是近处，需要在右图中填充的像素点数量是一样的。左图近处有足够多的点去填充右图，而左图远处的点有限，只能通过插值的方式创造“假的”像素点进行填充，所以就不那么清晰了。

4 提取车道线

在《无人驾驶技术入门（十四）︱初识图像之初级车道线检测》中，我们介

绍了通过 Canny 边缘提取算法获取车道线待选点的方法，随后使用霍夫直线变换进行了车道线的检测。在这里，我们也尝试使用边缘提取的方法进行车道线提取。

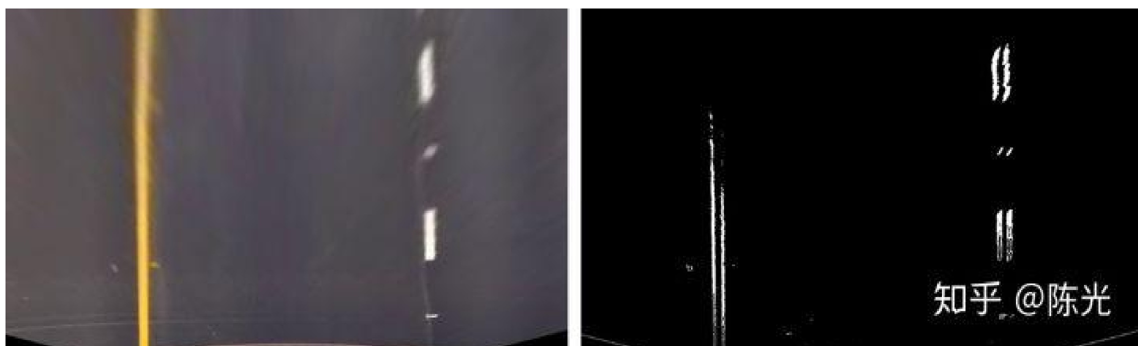
需要注意的是，Canny 边缘提取算法会将图像中各个方向、明暗交替位置的边缘都提取出来，很明显，Canny 边缘提取算法在处理有树木阴影的道路时，会将树木影子的轮廓也提取出来，这是我们不愿意看到的。

因此我们选用 Sobel 边缘提取算法。Sobel 相比于 Canny 的优秀之处在于，它可以选择横向或纵向的边缘进行提取。从投影变换后的图像可以看出，我们关心的正是车道线在横向上的边缘突变。

封装一下 OpenCV 提供的 cv2.Sobel()函数，将进行边缘提取后的图像做二进制图的转化，即提取到边缘的像素点显示为白色（值为 1），未提取到边缘的像素点显示为黑色（值为 0）。

```
def absSobelThreshold(img, orient='x', thresh_min=30, thresh_max=100):
    # Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # Apply x or y gradient with the OpenCV Sobel() function
    # and take the absolute value
    if orient == 'x':
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 1, 0))
    if orient == 'y':
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 0, 1))
    # Rescale back to 8 bit integer
    scaled_sobel = np.uint8(255*abs_sobel/np.max(abs_sobel))
    # Create a copy and apply the threshold
    binary_output = np.zeros_like(scaled_sobel)
    # Here I'm using inclusive (>=, <=) thresholds, but exclusive is ok too
    binary_output[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

    # Return the result
    return binary_output
```

使用同一组阈值对以上 6 幅做过投影变换的图像进行 x 方向的边缘提取，可以得到如下结果：



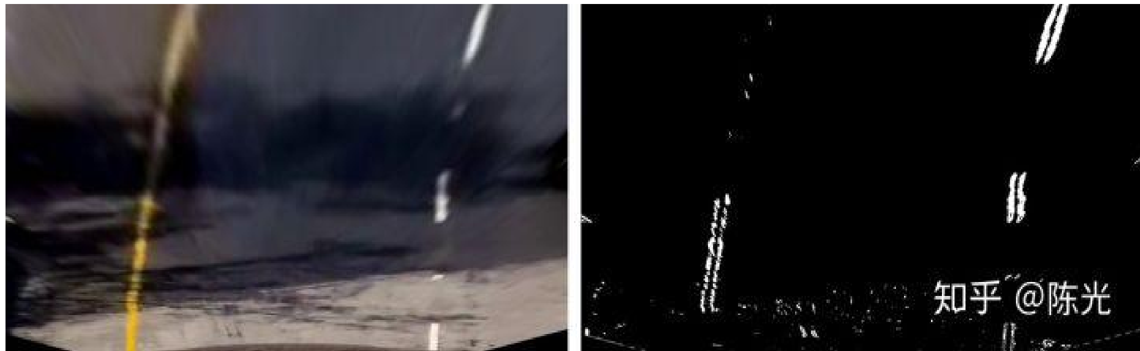
无阴影、颜色无明暗变换的直道

无阴影、颜色无明暗变换的弯道

有小面积阴影、颜色由暗到亮的直道

无阴影、道路标志线不清晰的弯道

有大面积阴影、颜色由暗到亮的弯道



有大面积阴影、颜色由亮到暗的弯道

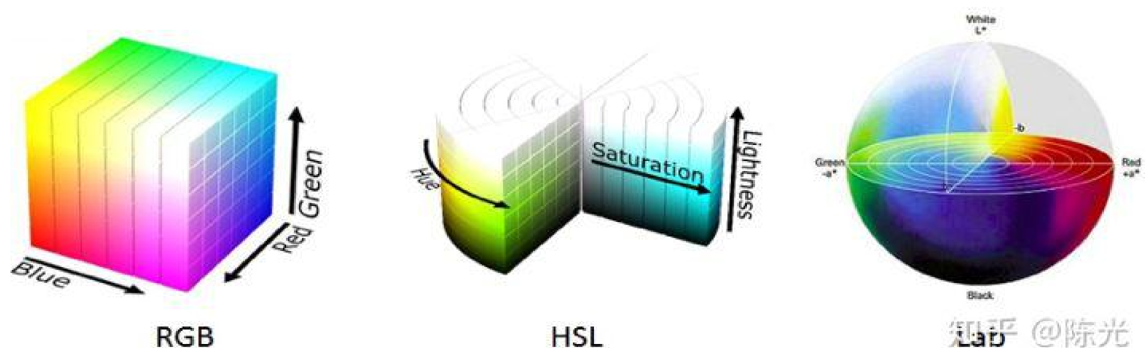
由以上结果可以看出，在明暗交替明显的路面上，如图 1 和图 2，横向的 Sobel 边缘提取算法在提取车道线的表现上还不错。不过一旦道路的明暗交替不那么明显了，如图 3 和图 4 的白色路面区域，很难提取到有效的车道线待选点。当面对有树木阴影覆盖的区域时，如图 5 和图 6，虽然能提取出车道线的大致轮廓，但会同时引入的噪声，给后续处理带来麻烦。

因此，横向的 Sobel 边缘提取算法，无法很好地处理路面阴影、明暗交替的道路工况。

无法使用边缘提取的方法提取车道线后，我们开始将颜色空间作为突破口。

在以上 6 个场景中，虽然路面明暗交替，而且偶尔会有阴影覆盖，但黄色和白色的车道线是一直都存在的。因此，我们如果能将图中的黄色和白色分割出来，然后将两种颜色组合在一幅图上，就能够得到一个比较好的处理结果。

一幅图像除了用 RGB（红绿蓝）三个颜色通道表示以外，还可以使用 HSL（H 色相、S 饱和度、L 亮度）和 Lab（L 亮度、a 红绿通道、b 蓝黄）模型来描述图像，三通道的值与实际的成像颜色如下图所示。



图片出处：

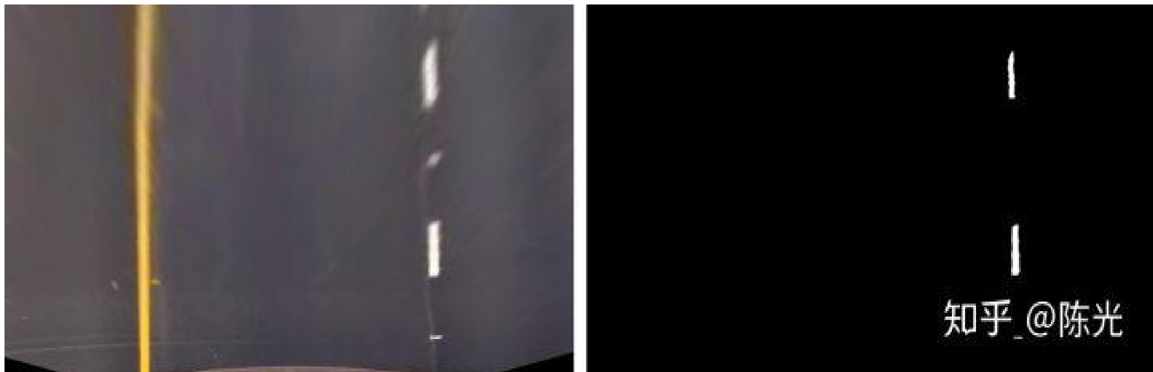
https://blog.csdn.net/wsp_1138886114/article/details/80660014

我们可以根据 HSL 模型中的 L（亮度）通道来分割出图像中的白色车道线，同时可以根据 Lab 模型中的 b（蓝黄）通道来分割出图像中的黄色车道线，再将两次的分割结果，去合集，叠加到一幅图上，就能得到两条完整的车道线了。

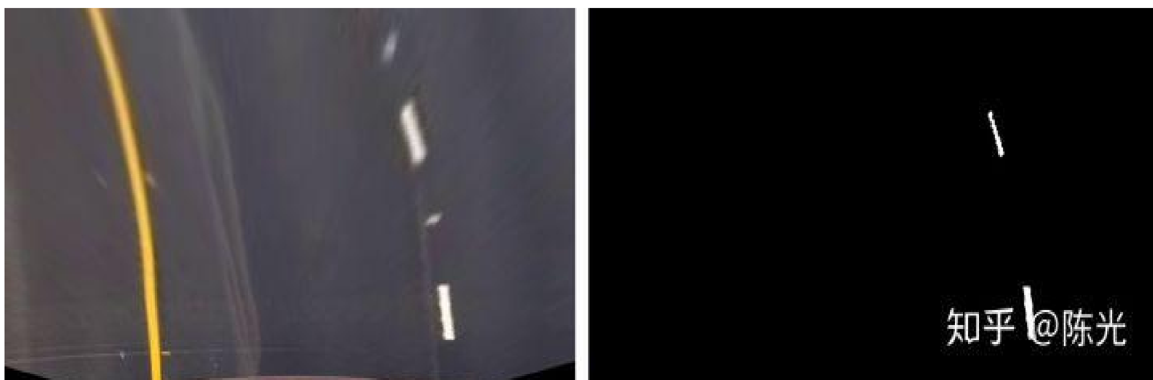
使用 OpenCV 提供的 cv2.cvtColor()接口，将 RGB 通道的图，转为 HLS 通道的图，随后对 L 通道进行分割处理，提取图像中白色的车道线。封装成代码如下：

```
def hlsLSelect(img, thresh=(220, 255)):
    hls = cv2.cvtColor(img, cv2.COLOR_BGR2HLS)
    l_channel = hls[:,:,1]
    l_channel = l_channel*(255/np.max(l_channel))
    binary_output = np.zeros_like(l_channel)
    binary_output[(l_channel > thresh[0]) & (l_channel <= thresh[1])] = 1
    return binary_output
```

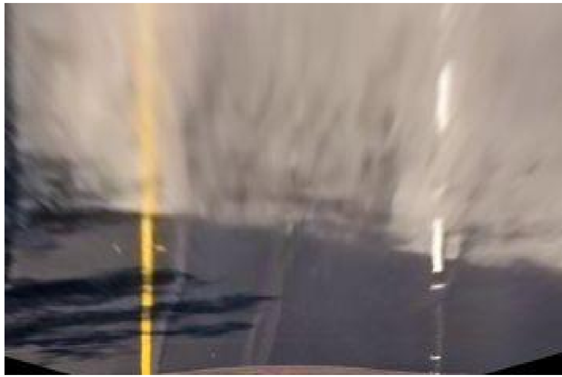
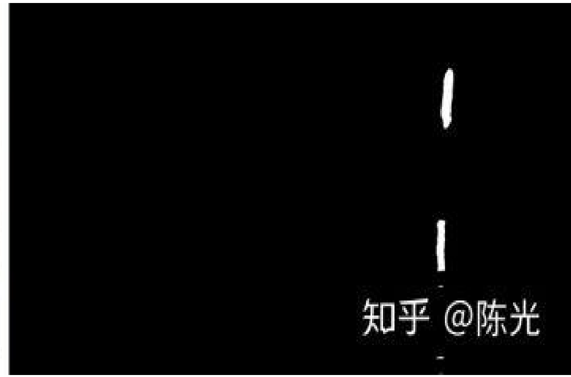
使用同一组阈值对以上 6 种工况进行处理，处理结果如下图所示。



无阴影、颜色无明暗变换的直道



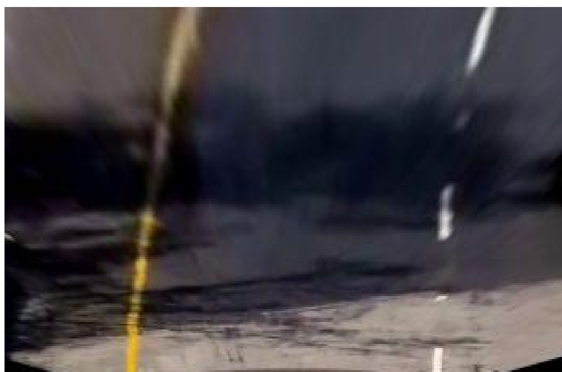
无阴影、颜色无明暗变换的弯道

有小面积阴影、颜色由暗到亮的直道

无阴影、道路标志线不清晰的弯道

有大面积阴影、颜色由暗到亮的弯道

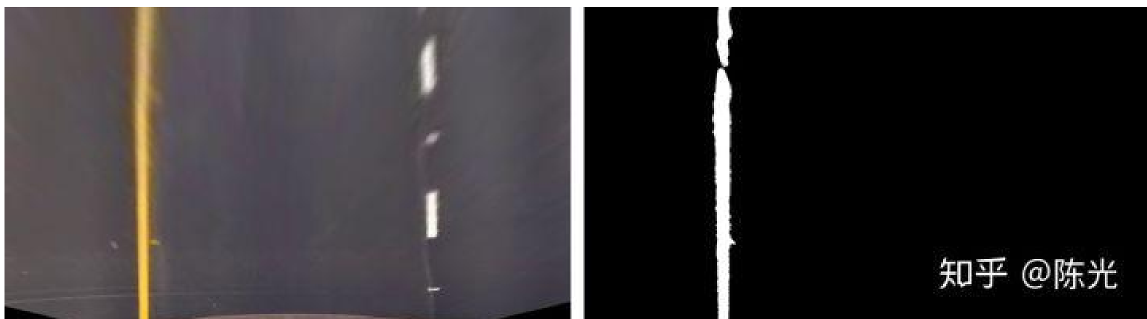


有大面积阴影、颜色由亮到暗的弯道

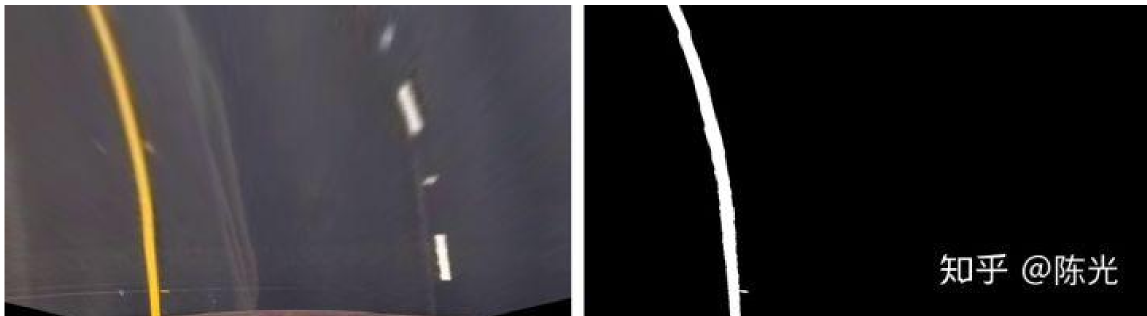
使用 OpenCV 提供的 cv2.cvtColor()接口，将 RGB 通道的图，转为 Lab 通道的图，随后对 b 通道进行分割处理，提取图像中黄色的车道线。封装成代码如下：

```
def labBSelect(img, thresh=(195, 255)):
    # 1) Convert to LAB color space
    lab = cv2.cvtColor(img, cv2.COLOR_BGR2Lab)
    lab_b = lab[:,:,2]
    # don't normalize if there are no yellows in the image
    if np.max(lab_b) > 100:
        lab_b = lab_b*(255/np.max(lab_b))
    # 2) Apply a threshold to the L channel
    binary_output = np.zeros_like(lab_b)
    binary_output[((lab_b > thresh[0]) & (lab_b <= thresh[1]))] = 1
    # 3) Return a binary image of threshold result
    return binary_output
```

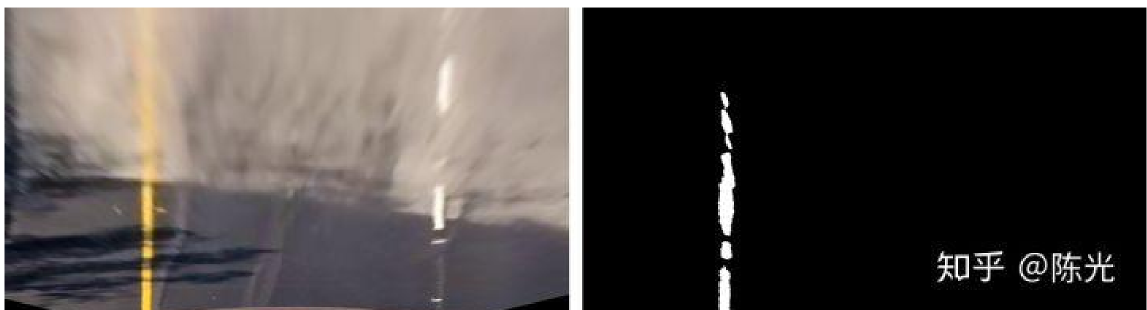
使用同一组阈值对以上 6 种工况进行处理，处理结果如下图所示。



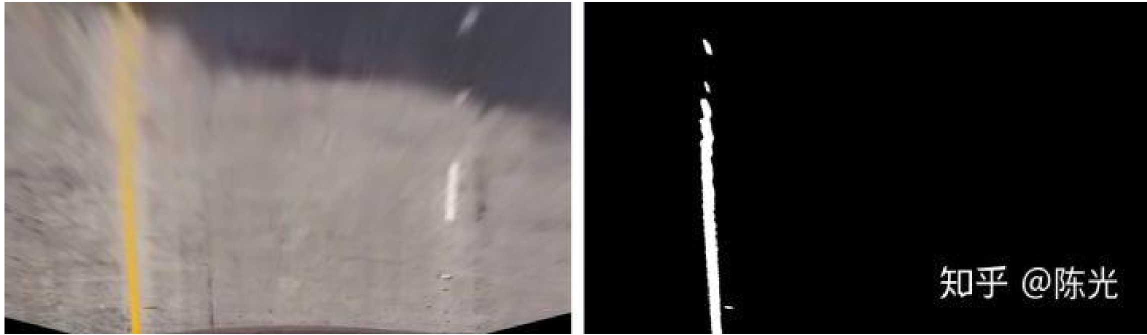
无阴影、颜色无明暗变换的直道



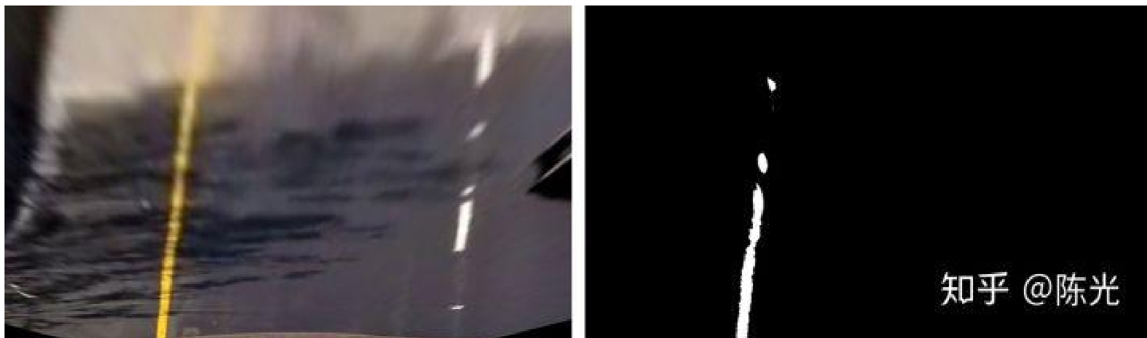
无阴影、颜色无明暗变换的弯道



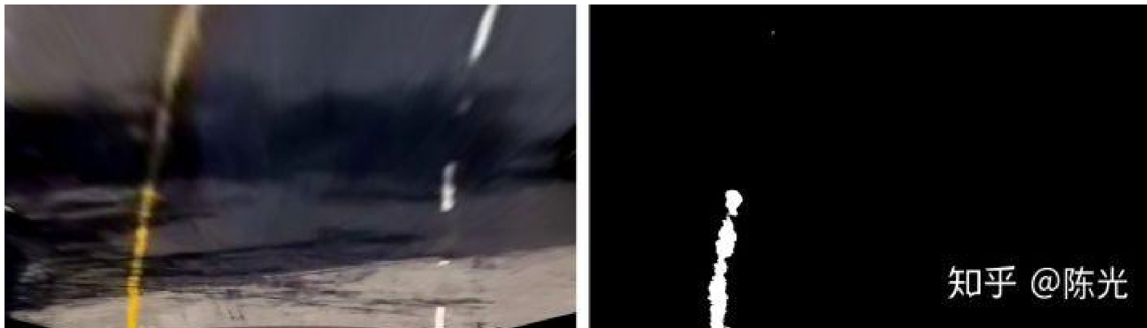
有小面积阴影、颜色由暗到亮的直道



无阴影、道路标志线不清晰的弯道



有大面积阴影、颜色由暗到亮的弯道



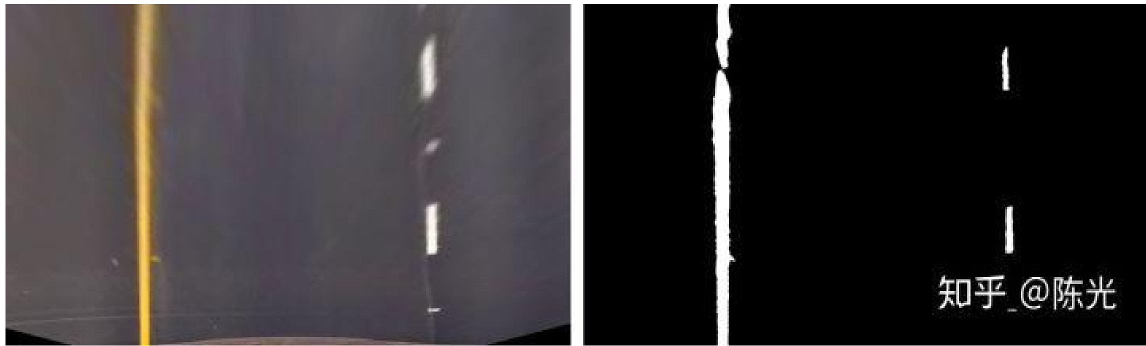
有大面积阴影、颜色由亮到暗的弯道

根据以上试验可知，L 通道能够较好地分割出图像中的白色车道线，b 通道能够较好地分割出图像中的黄色车道线。即使面对树木阴影和路面颜色突变的场景，也能尽可能少地引入噪声。

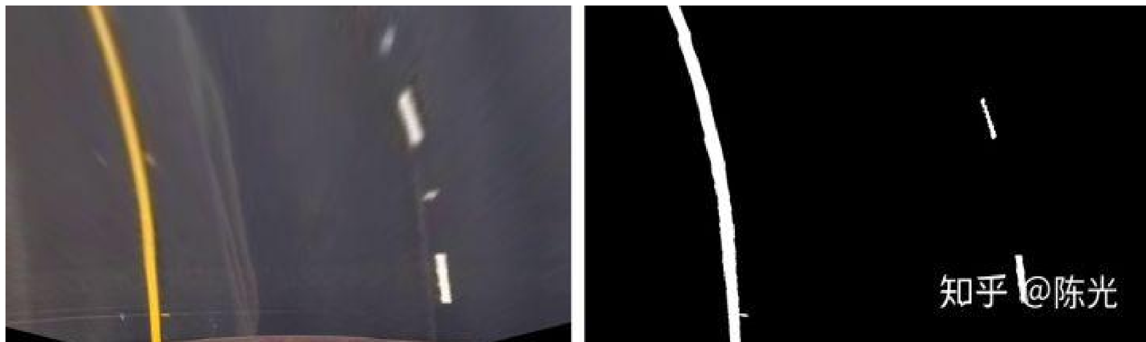
最后，我们使用以下代码，将两个通道分割的图像合并。

```
hlsL_binary = hlsLSelect(test_warp_image)
labB_binary = labBSelect(test_warp_image)
combined_binary = np.zeros_like(hlsL_binary)
combined_binary[(hlsL_binary == 1) | (labB_binary == 1)] = 1
```

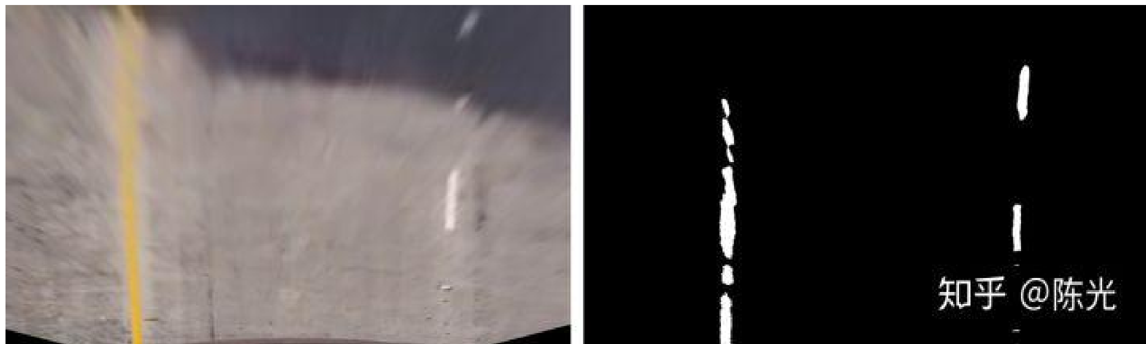
最终合并的效果如下图所示：



无阴影、颜色无明暗变换的直道



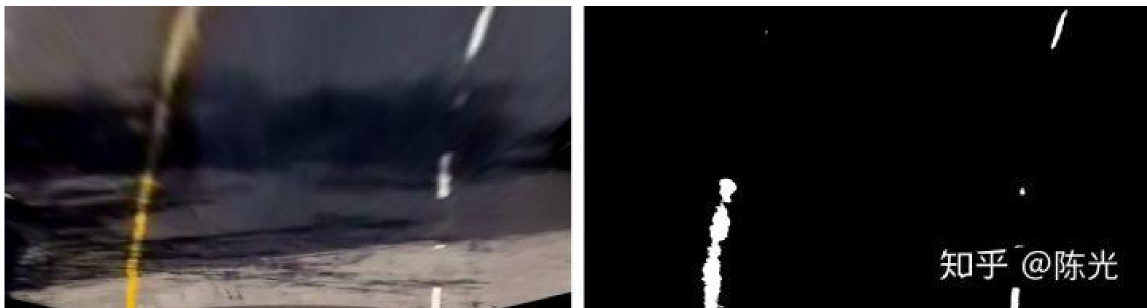
无阴影、颜色无明暗变换的弯道



有小面积阴影、颜色由亮到暗的直道

无阴影、道路标志线不清晰的弯道

有大面积阴影、颜色由暗到亮的弯道



有大面积阴影、颜色由亮到暗的弯道

以上仅仅是车道线提取的方法之一。除了可以通过 HSL 和 Lab 颜色通道，这种基于规则的方法，分割出车道线外，还可以使用基于深度学习的方法。它们目的都是为了能够稳定地将车道线从图像中分割出来。

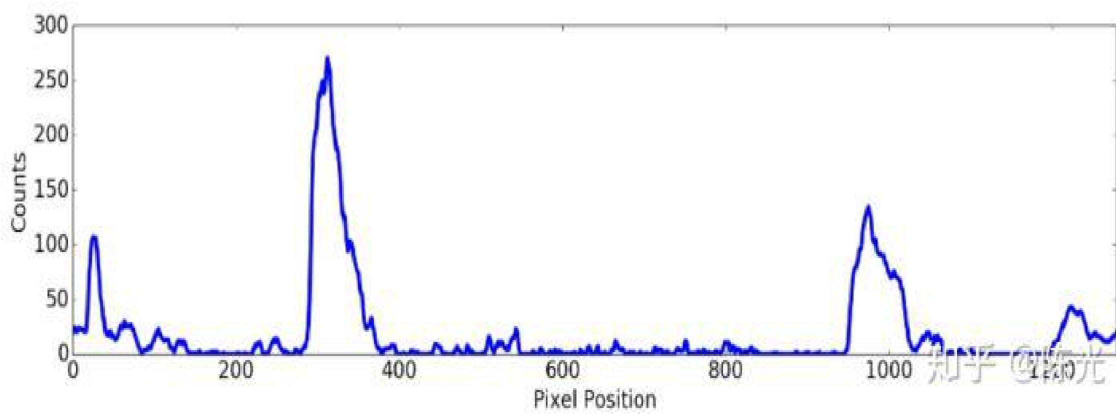
5　检测车道线

在检测车道线前，需要粗定位车道线的位置。为了方便理解，这里引入一个概念——直方图。

以下面这幅包含噪点的图像为例，进行直方图的介绍。

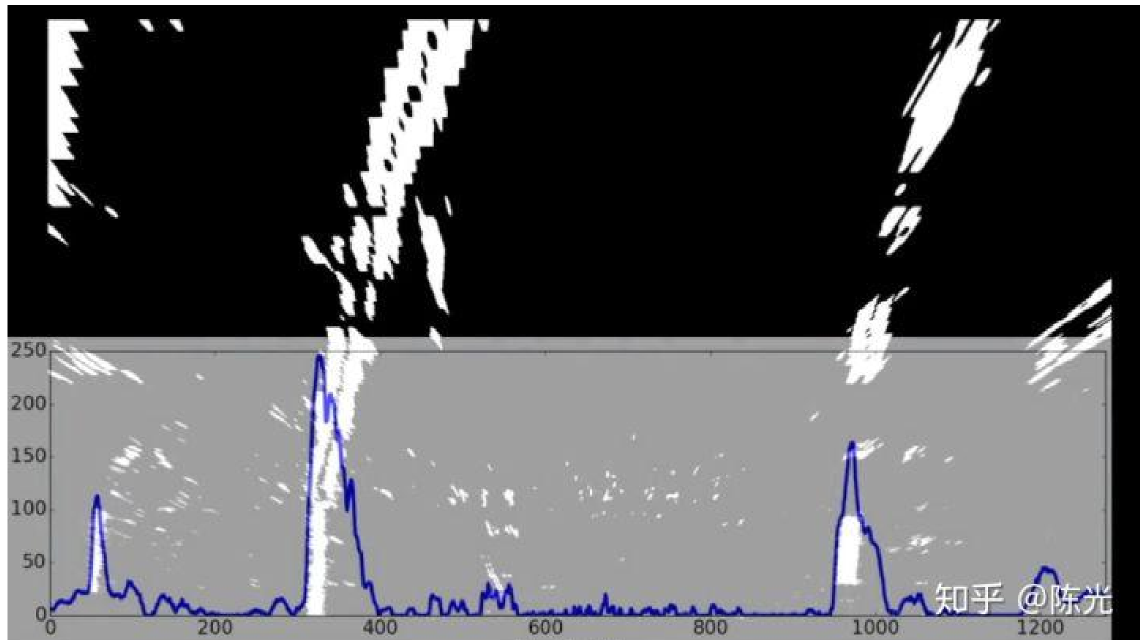
图片出处：优达学城(Udacity)无人驾驶工程师学位

我们知道，我们处理的图像的分辨率为1280*720，即720行，1280列。如果我将每一列的白色的点数量进行统计，即可得到1280个值。将这1280个值绘制在一个坐标系中，横坐标为1-1280，纵坐标表示每列中白色点的数量，那么这幅图就是“直方图”，如下图所示：



图片出处：优达学城(Udacity)无人驾驶工程师学位

将两幅图叠加，效果如下：



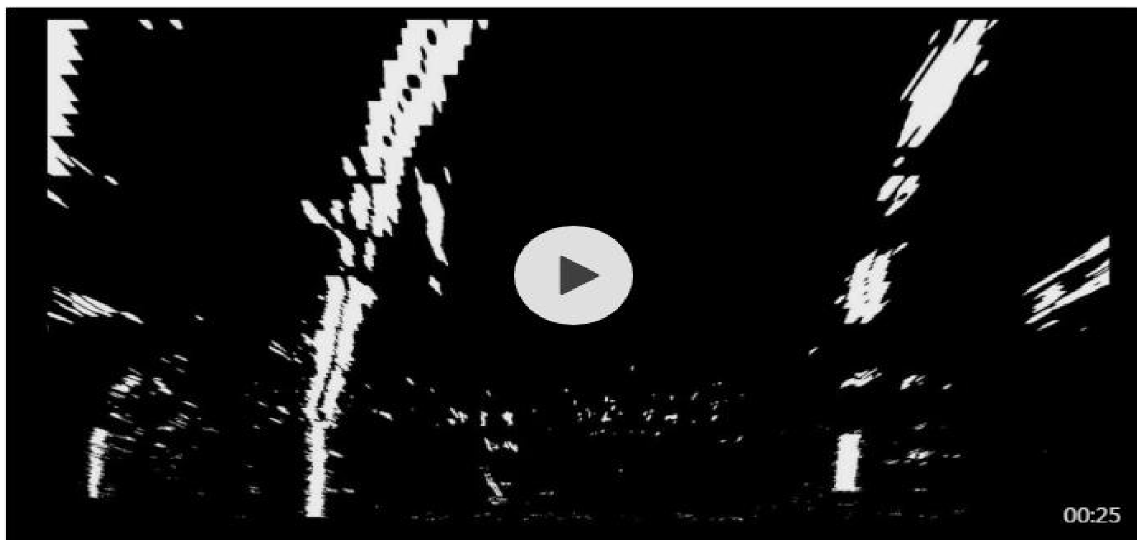
图片出处：优达学城(Udacity)无人驾驶工程师学位

找到直方图左半边最大值所对应的列数，即为左车道线所在的大致位置；找到直方图右半边最大值所对应的列数，即为右车道线所在的大致位置。

使用直方图找左右车道线大致位置的代码如下，其中leftx_base和rightx_base即为左右车道线所在列的大致位置。

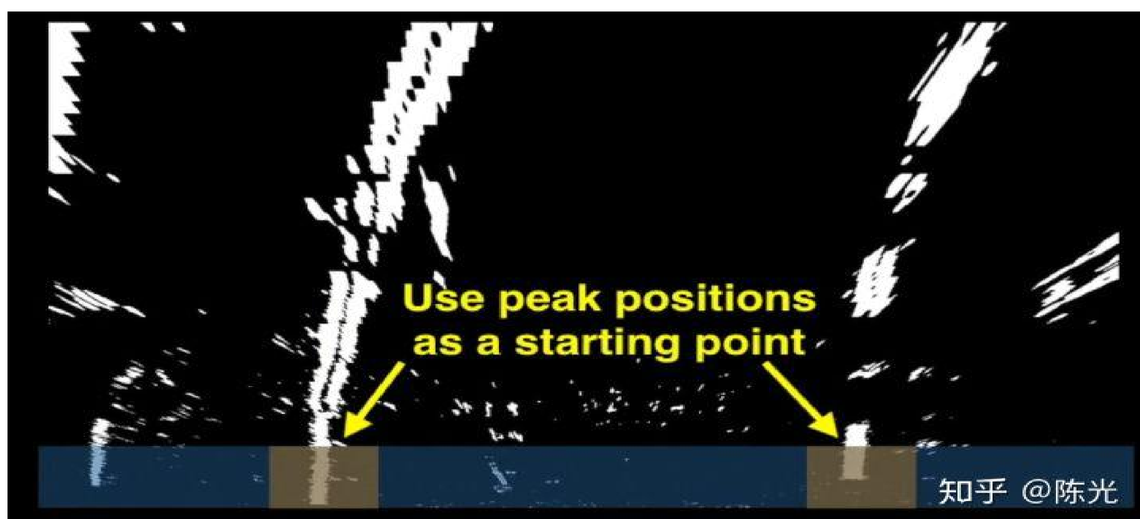
```
# Take a histogram of the bottom half of the image
histogram = np.sum(combined_binary[combined_binary.shape[0]//2:,:], axis=0)
# Create an output image to draw on and visualize the result
out_img = np.dstack((combined_binary, combined_binary, combined_binary))
# Find the peak of the left and right halves of the histogram
# These will be the starting point for the left and right lines
midpoint = np.int(histogram.shape[0]//2)
leftx_base = np.argmax(histogram[:midpoint])
rightx_base = np.argmax(histogram[midpoint:]) + midpoint
```

确定了左右车道线的大致位置后，使用一种叫做“滑动窗口”的技术，在图中对左右车道线的点进行搜索。先看一个介绍"滑动窗口"原理的视频（视频大小1.18M）。

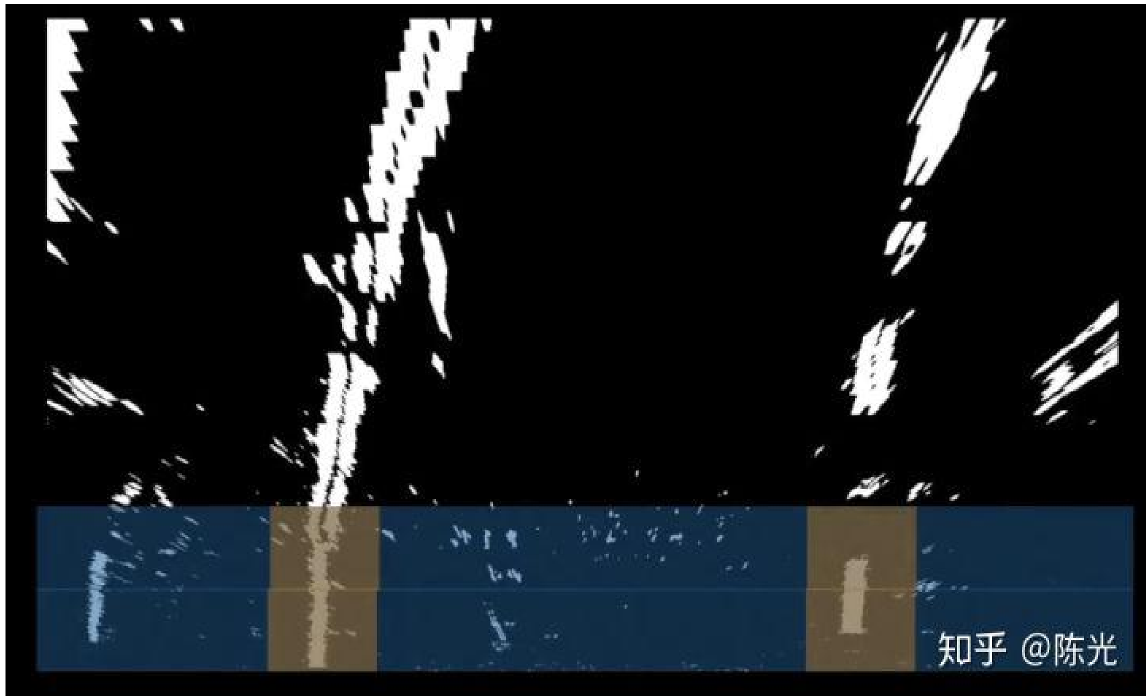


视频地址见原文：https://zhuanlan.zhihu.com/p/54866418

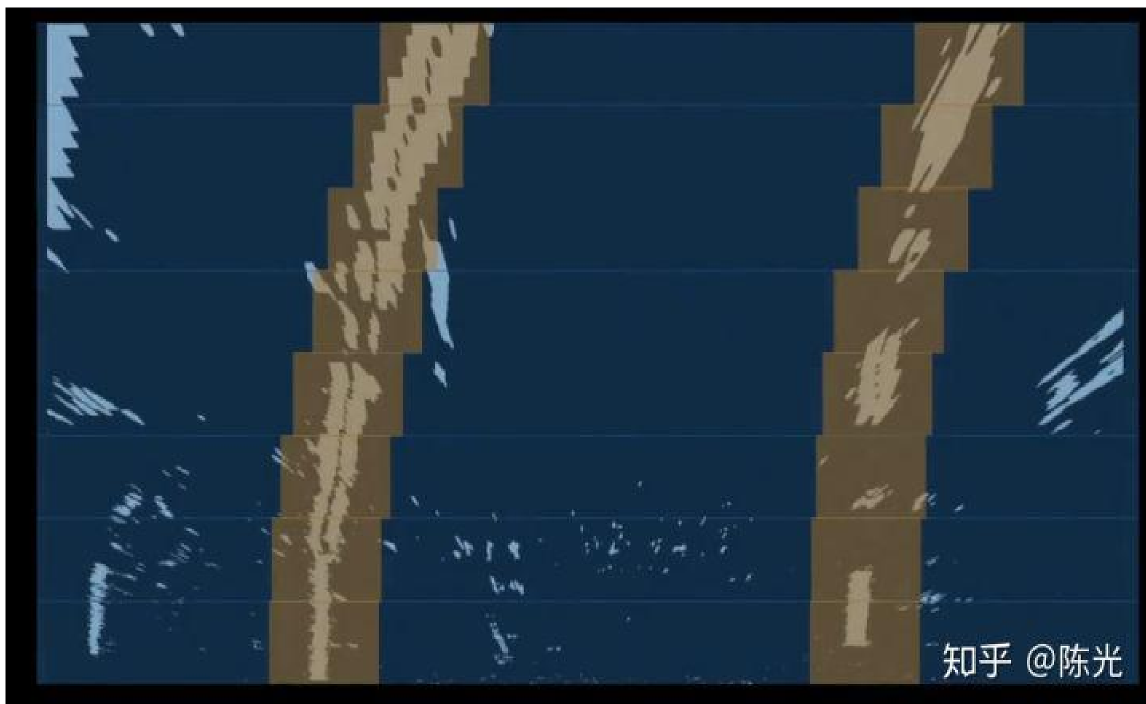
首先根据前面介绍的直方图方法，找到左右车道线的大致位置，将这两个大致位置作为起始点。定义一个矩形区域，称之为“窗口”（图中棕色的部分），分别以两个起始点作为窗口的下边线中点，存储所有在方块中的白色点的横坐标。



随后对存储的横坐标取均值，将该均值所在的列以及第一个”窗口“的上边缘所在的位置，作为下一个“窗口”的下边线中点，继续搜索。



以此往复，直到把所有的行都搜索完毕。



所有落在窗口（图中棕色区域）中的白点，即为左右车道线的待选点，如下图蓝色和红色所示。随后将蓝色点和红色点做三次曲线拟合，即可得到车道线的曲线方程。

使用直方图、滑动窗口检测车道线的代码如下：

```
#################################################################
# Step 5 : Detect lane lines through moving window
#################################################################
def find_lane_pixels(binary_warped, nwindows, margin, minpix):
    # Take a histogram of the bottom half of the image
    histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)
    # Create an output image to draw on and visualize the result
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))
    # Find the peak of the left and right halves of the histogram
    # These will be the starting point for the left and right lines
    midpoint = np.int(histogram.shape[0]//2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    # Set height of windows - based on nwindows above and image shape
    window_height = np.int(binary_warped.shape[0]//nwindows)
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    # Current positions to be updated later for each window in nwindows
```

```
    leftx_current = leftx_base
    rightx_current = rightx_base

    # Create empty lists to receive left and right lane pixel indices
    left_lane_inds = []
    right_lane_inds = []
```

```
# Step through the windows one by one
for window in range(nwindows):
    # Identify window boundaries in x and y (and right and left)
    win_y_low = binary_warped.shape[0] - (window+1)*window_height
    win_y_high = binary_warped.shape[0] - window*window_height
    win_xleft_low = leftx_current - margin
    win_xleft_high = leftx_current + margin
    win_xright_low = rightx_current - margin
    win_xright_high = rightx_current + margin

    # Draw the windows on the visualization image
    cv2.rectangle(out_img,(win_xleft_low,win_y_low),
    (win_xleft_high,win_y_high),(0,255,0), 2)
    cv2.rectangle(out_img,(win_xright_low,win_y_low),
    (win_xright_high,win_y_high),(0,255,0), 2)

    # Identify the nonzero pixels in x and y within the window #
    good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
    (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_high)).nonzero()[0]
```

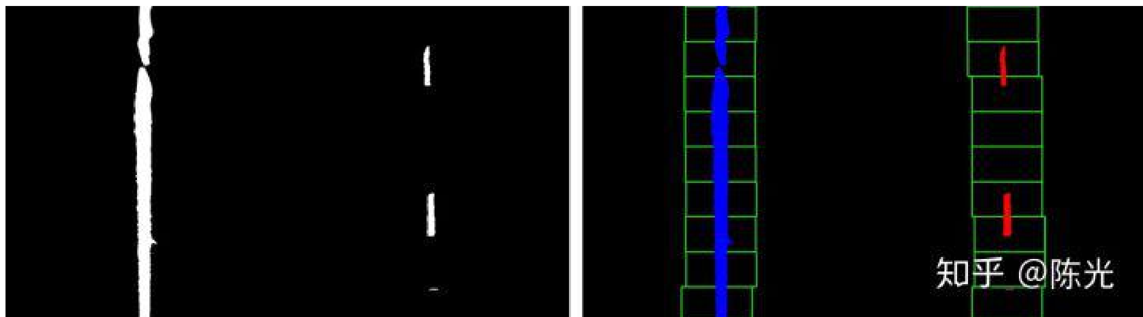
```
    # Append these indices to the lists
    left_lane_inds.append(good_left_inds)
    right_lane_inds.append(good_right_inds)

    # If you found > minpix pixels, recenter next window on their mean position
    if len(good_left_inds) > minpix:
        leftx_current = np.int(np.mean(nonzerox[good_left_inds]))
    if len(good_right_inds) > minpix:
        rightx_current = np.int(np.mean(nonzerox[good_right_inds]))

# Concatenate the arrays of indices (previously was a list of lists of pixels)
try:
    left_lane_inds = np.concatenate(left_lane_inds)
    right_lane_inds = np.concatenate(right_lane_inds)
except ValueError:
    # Avoids an error if the above is not implemented fully
    pass

# Extract left and right line pixel positions
```

```
leftx = nonzerox[left_lane_inds]
lefty = nonzeroy[left_lane_inds]
rightx = nonzerox[right_lane_inds]
righty = nonzeroy[right_lane_inds]

return leftx, lefty, rightx, righty, out_img
```

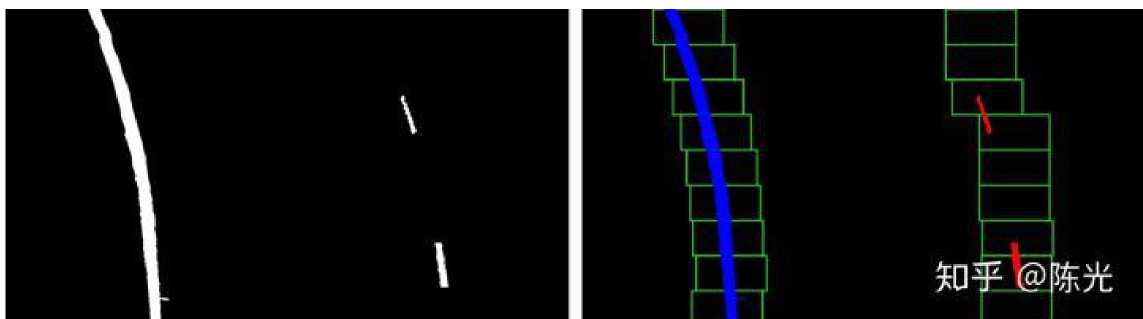
```
def fit_polynomial(binary_warped, nwindows=9, margin=100, minpix=50):
    # Find our lane pixels first
    leftx, lefty, rightx, righty, out_img = find_lane_pixels(
        binary_warped, nwindows, margin, minpix)

    # Fit a second order polynomial to each using `np.polyfit`
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)

    # Generate x and y values for plotting
    ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )
    try:
        left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
        right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
    except TypeError:
        # Avoids an error if `left` and `right_fit` are still none or incorrect
        print('The function failed to fit a line!')
        left_fitx = 1*ploty**2 + 1*ploty
        right_fitx = 1*ploty**2 + 1*ploty
```

```
    ## Visualization ##
    # Colors in the left and right lane regions
    out_img[lefty, leftx] = [255, 0, 0]
    out_img[righty, rightx] = [0, 0, 255]

    # Plots the left and right polynomials on the lane lines
    #plt.plot(left_fitx, ploty, color='yellow')
    #plt.plot(right_fitx, ploty, color='yellow')

    return out_img, left_fit, right_fit, ploty
```

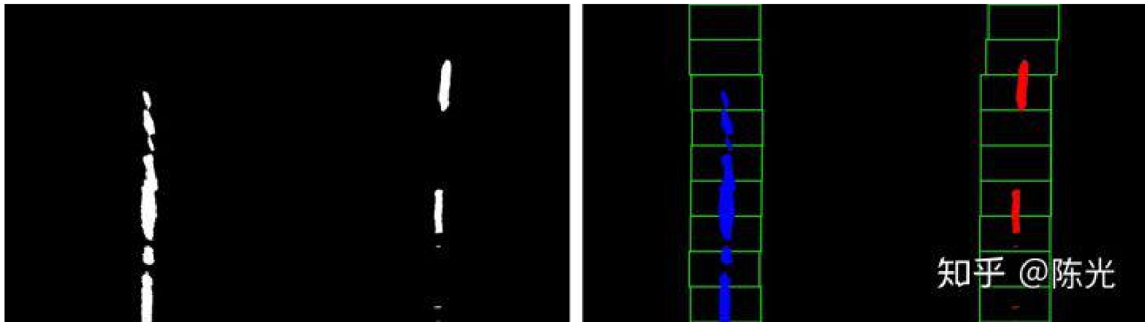
对以上 6 种工况进行车道线检测，处理结果如下图所示。



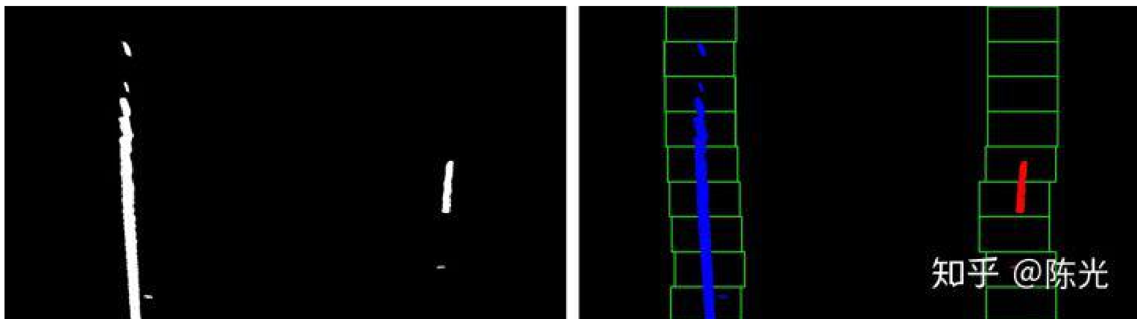
无阴影、颜色无明暗变换的直道



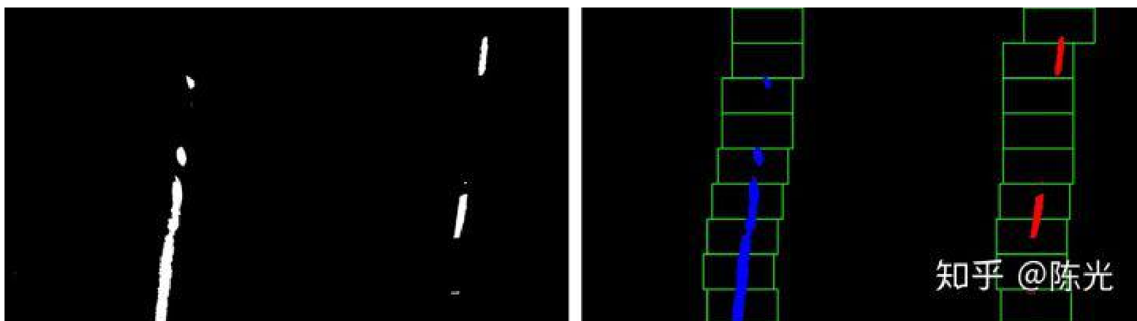
无阴影、颜色无明暗变换的弯道



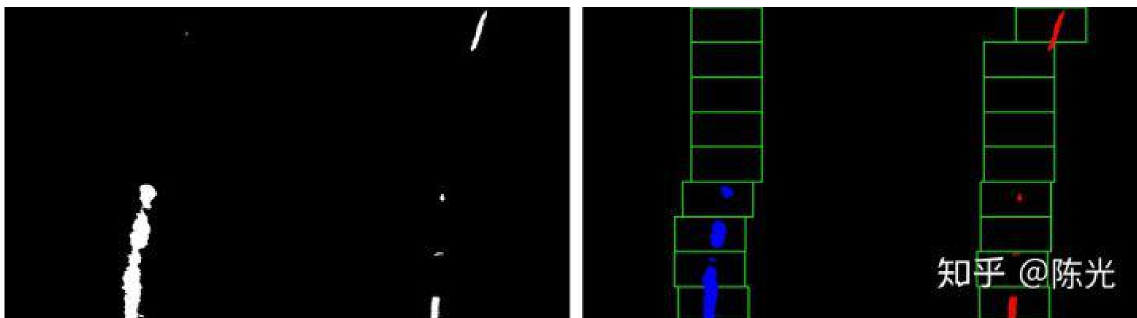
有小面积阴影、颜色由暗到亮的直道



无阴影、道路标志线不清晰的弯道



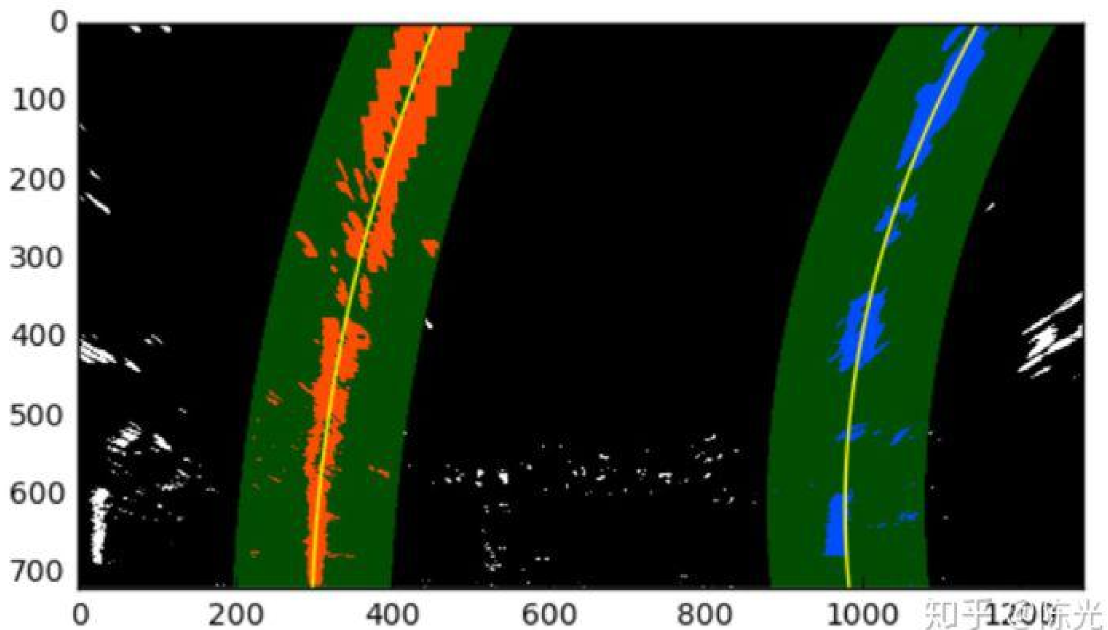
有大面积阴影、颜色由暗到亮的弯道



有大面积阴影、颜色由亮到暗的弯道

6 跟踪车道线

视频数据是连续的图片，基于连续两帧图像中的车道线不会突变的先验知识，我们可以使用上一帧检测到的车道线结果，作为下一帧图像处理的输入，搜索上一帧车道线检测结果附近的点，这样不仅可以减少计算量，而且得到的车道线结果也更稳定，如下图所示。



图片出处：优达学城(Udacity)无人驾驶工程师学位

图中的细黄线为上一帧检测到的车道线结果，绿色阴影区域为细黄线横向扩展的一个区域，通过搜索该区域内的白点坐标，即可快速确定当前帧中左右车道线的待选点。

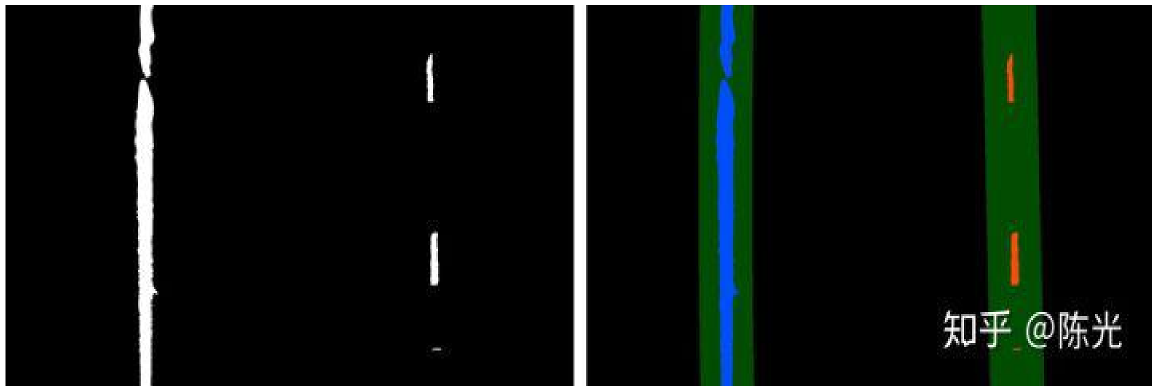
使用上一帧的车道线检测结果进行车道线跟踪的代码如下：

```
#################################################################
# Step 6 : Track lane lines based the latest lane line result
#################################################################
def fit_poly(img_shape, leftx, lefty, rightx, righty):
     ### TO-DO: Fit a second order polynomial to each with np.polyfit() ###
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)
    # Generate x and y values for plotting
    ploty = np.linspace(0, img_shape[0]-1, img_shape[0])
    ### TO-DO: Calc both polynomials using ploty, left_fit and right_fit ###
```

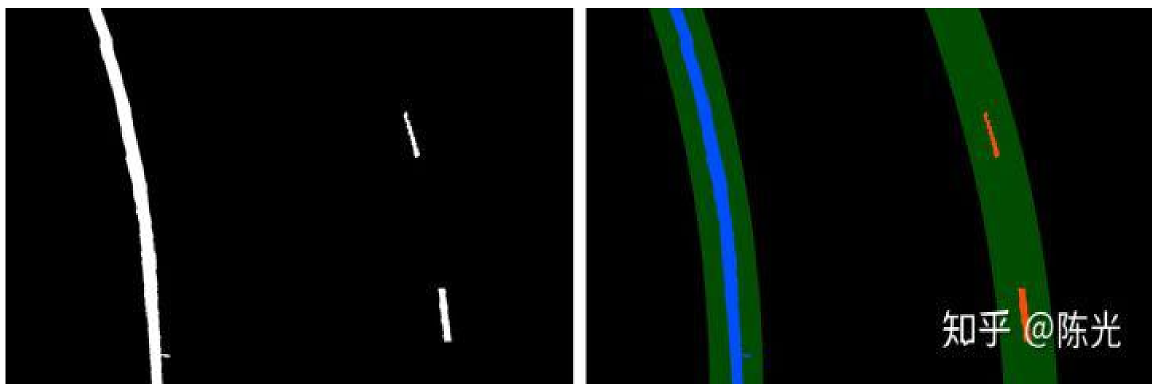
```
    left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
    right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

    return left_fitx, right_fitx, ploty, left_fit, right_fit

def search_around_poly(binary_warped, left_fit, right_fit):
    # HYPERPARAMETER
    # Choose the width of the margin around the previous polynomial to search
    # The quiz grader expects 100 here, but feel free to tune on your own!
    margin = 60

    # Grab activated pixels
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    ### TO-DO: Set the area of search based on activated x-values ###
    ### within the +/- margin of our polynomial function ###
    ### Hint: consider the window areas for the similarly named variables ###
    ### in the previous quiz, but change the windows to our new search area ###
    left_lane_inds = ((nonzerox > (left_fit[0]*(nonzeroy**2) + left_fit[1]*nonzeroy +
                    left_fit[2] - margin)) & (nonzerox < (left_fit[0]*(nonzeroy**2) +
                    left_fit[1]*nonzeroy + left_fit[2] + margin)))
    right_lane_inds = ((nonzerox > (right_fit[0]*(nonzeroy**2) + right_fit[1]*nonzero
                    right_fit[2] - margin)) & (nonzerox < (right_fit[0]*(nonzeroy**2)
                    right_fit[1]*nonzeroy + right_fit[2] + margin)))

    # Again, extract left and right line pixel positions
    leftx = nonzerox[left_lane_inds]
    lefty = nonzeroy[left_lane_inds]
    rightx = nonzerox[right_lane_inds]
    righty = nonzeroy[right_lane_inds]

    # Fit new polynomials
    left_fitx, right_fitx, ploty, left_fit, right_fit = fit_poly(binary_warped.shape,

    ## Visualization ##
    # Create an image to draw on and an image to show the selection window
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
    window_img = np.zeros_like(out_img)
    # Color in left and right line pixels
    out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
    out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]

    # Generate a polygon to illustrate the search window area
    # And recast the x and y points into usable format for cv2.fillPoly()
    left_line_window1 = np.array([np.transpose(np.vstack([left_fitx-margin, ploty]))]
    left_line_window2 = np.array([np.flipud(np.transpose(np.vstack([left_fitx+margin,
                              ploty])))])
```

```
    left_line_pts = np.hstack((left_line_window1, left_line_window2))
    right_line_window1 = np.array([np.transpose(np.vstack([right_fitx-margin, ploty])
    right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx+margi
                                  ploty])))])
    right_line_pts = np.hstack((right_line_window1, right_line_window2))

    # Draw the lane onto the warped blank image
    cv2.fillPoly(window_img, np.int_([left_line_pts]), (0,255, 0))
    cv2.fillPoly(window_img, np.int_([right_line_pts]), (0,255, 0))
    result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)

    # Plot the polynomial lines onto the image
    #plt.plot(left_fitx, ploty, color='yellow')
    #plt.plot(right_fitx, ploty, color='yellow')
    ## End visualization steps ##

    return result, left_fit, right_fit, ploty
```

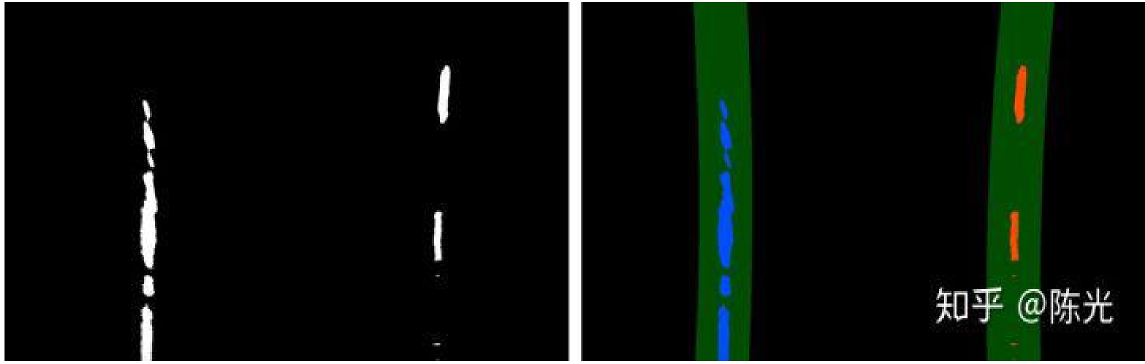
对以上 6 种工况进行车道线跟踪，处理结果如下图所示。



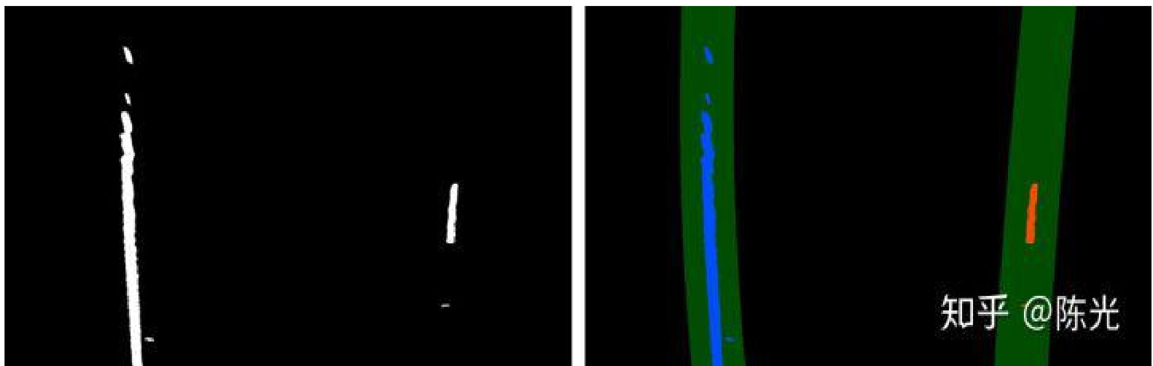
无阴影、颜色无明暗变换的直道



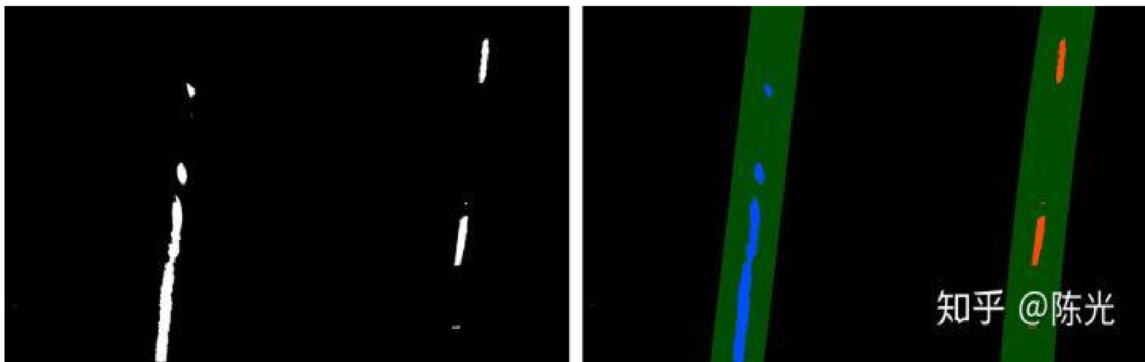
无阴影、颜色无明暗变换的弯道



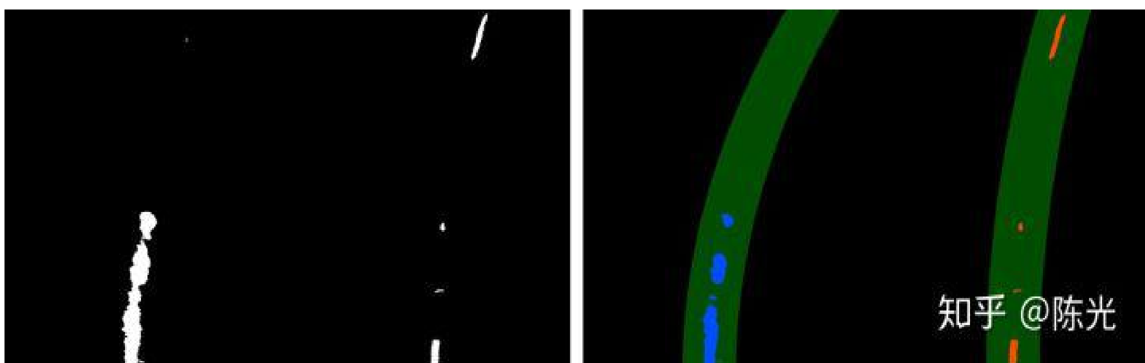
有小面积阴影、颜色由暗到亮的直道



无阴影、道路标志线不清晰的弯道



有大面积阴影、颜色由暗到亮的弯道



有大面积阴影、颜色由亮到暗的弯道

以上，我们就完成了在透视变换结果上的车道线检测和跟踪。

7 逆投影到原图

我们在计算透视变换矩阵时计算了两个矩阵 M 和 Minv，使用 M 能够实现透视变换，使用 Minv 能够实现逆透视变换。

```
M = cv2.getPerspectiveTransform(src, dst)
Minv = cv2.getPerspectiveTransform(dst, src)
```

我们将两条车道线所围成的区域涂成绿色，并将结果绘制在“鸟瞰图”上后，使用逆透视变换矩阵反投到原图上，即可实现在原图上的可视化效果。代码如下：

```
#################################################################
# Step 8 : Draw lane line result on undistorted image
#################################################################
def drawing(undist, bin_warped, color_warp, left_fitx, right_fitx):
    # Create an image to draw the lines on
    warp_zero = np.zeros_like(bin_warped).astype(np.uint8)
    color_warp = np.dstack((warp_zero, warp_zero, warp_zero))

    # Recast the x and y points into usable format for cv2.fillPoly()
    pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
    pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
    pts = np.hstack((pts_left, pts_right))

    # Draw the lane onto the warped blank image
    cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))

    # Warp the blank back to original image space using inverse perspective matrix (M
    newwarp = cv2.warpPerspective(color_warp, Minv, (undist.shape[1], undist.shape[0]
    # Combine the result with the original image
    result = cv2.addWeighted(undist, 1, newwarp, 0.3, 0)
    return result
```

以上 6 个场景的左右车道线绘制结果如下所示：

无阴影、颜色无明暗变换的直道

无阴影、颜色无明暗变换的弯道

有小面积阴影、颜色由暗到亮的直道

无阴影、道路标志线不清晰的弯道

有大面积阴影、颜色由暗到亮的弯道

有大面积阴影、颜色由亮到暗的弯道

8 处理视频

在一步步完成摄像机标定、图像畸变校正、透视变换、提取车道线、检测车道线、跟踪车道线后，我们在图像上实现了复杂环境下的车道线检测算法。现在我们将视频转化为图片，然后一帧帧地对视频数据进行处理，然后将车道线检测结果存为另一段视频。

处理代码如下：

```
nx = 9
ny = 6
ret, mtx, dist, rvecs, tvecs = getCameraCalibrationCoefficients('camera_cal/calibrati

src = np.float32([[580, 460], [700, 460], [1096, 720], [200, 720]])
dst = np.float32([[300, 0], [950, 0], [950, 720], [300, 720]])

video_input = 'project_video.mp4'
video_output = 'result_video.mp4'

cap = cv2.VideoCapture(video_input)

fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter(video_output, fourcc, 20.0, (1280, 720))

detected = False

while(True):
    ret, image = cap.read()
    if ret:
        undistort_image = undistortImage(image, mtx, dist)
        warp_image, M, Minv = warpImage(undistort_image, src, dst)
        hlsL_binary = hlsLSelect(warp_image)
        labB_binary = labBSelect(warp_image, (205, 255))
```

```
        combined_binary = np.zeros_like(sx_binary)
        combined_binary[(hlsL_binary == 1) | (labB_binary == 0)] = 1
        left_fit = []
        right_fit = []
        ploty = []
        if detected == False:
            out_img, left_fit, right_fit, ploty = fit_polynomial(combined_binary, nwi
            if (len(left_fit) > 0 & len(right_fit) > 0) :
                detected = True
            else :
                detected = False
        else:
            track_result, left_fit, right_fit, ploty,  = search_around_poly(combined_
            if (len(left_fit) > 0 & len(right_fit) > 0) :
                detected = True
            else :
                detected = False

        result = drawing(undistort_image, combined_binary, warp_image, left_fitx, rig

        out.write(result)
```

```
    else:
        break

cap.release()
out.release()
```

最终的视频车道线检测结果如下所示（视频大小 7.59M）：

视频地址见原文：https://zhuanlan.zhihu.com/p/54866418

视频中左上角出现的道路曲率和车道偏离量的计算都是获取车道线曲线方程后的具体应用，这里不做详细讨论。

9 结语

以上就是《再识图像之高级车道线检测》的全部内容，本次分享中介绍的摄像机标定、投影变换、颜色通道、滑动窗口等技术，在计算机视觉领域均得到了广泛应用。

处理复杂道路场景下的视频数据是一项及其艰巨的任务。仅以提取车道线的过程为例，使用设定规则的方式提取车道线，虽然能够处理项目视频中的场景，但面对变化更为恶劣的场景时，还是无能为力。现阶段解决该问题的方法就是通过深度学习的方法，拿足够多的标注数据去训练模型，才能尽可能多地达到稳定的检测效果。

文章中所使用的图片、视频素材，技术细节和部分代码来自优达学城（Udacity）无人驾驶工程师学位的第二个项目。目前我已学完了两个实战项目，从一个工程师的角度来看，这套课程十分适合那些想转行到无人驾驶领域，又不知道从何入手的人。

听说做完毕业项目后，工作人员会把代码部署在硅谷 Udacity 的无人车 Carla 上，实车测试。希望自己能零 bug，顺利毕业~

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