Carnd Term1 Proj4 Advanced Lane Finding

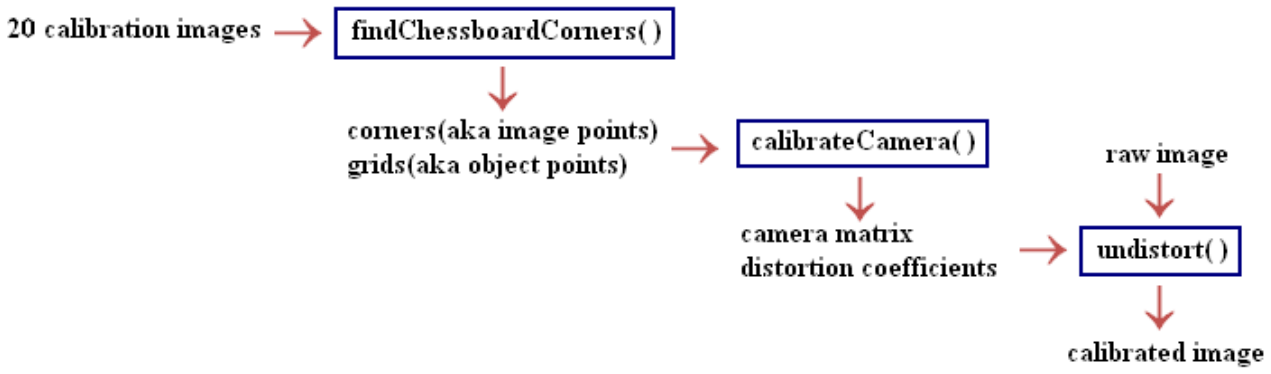
Joe Zhou – ibalpowr@gmail.com

This project is a continuation of Project 1 Lane Finding. In this project, we will detect lane marks under more challenging conditions involving curves and tree shadows. The techniques of Canny edges and Hough transform, which are used in Project 1, become less useful. So we explore a new set of tools: camera calibration, Sobel and Scharr gradients, perspective transform, and sliding windows. Chapter 10 "Filters and Convolution", Chapter 18 "Camera Models and Calibration" and Chapter 19 "Projects and Three-Dimensional Vision Projections" of *Learning OpenCV3* are excellent references. Also a good reference paper is Lee-Wong-Xiao[2015].

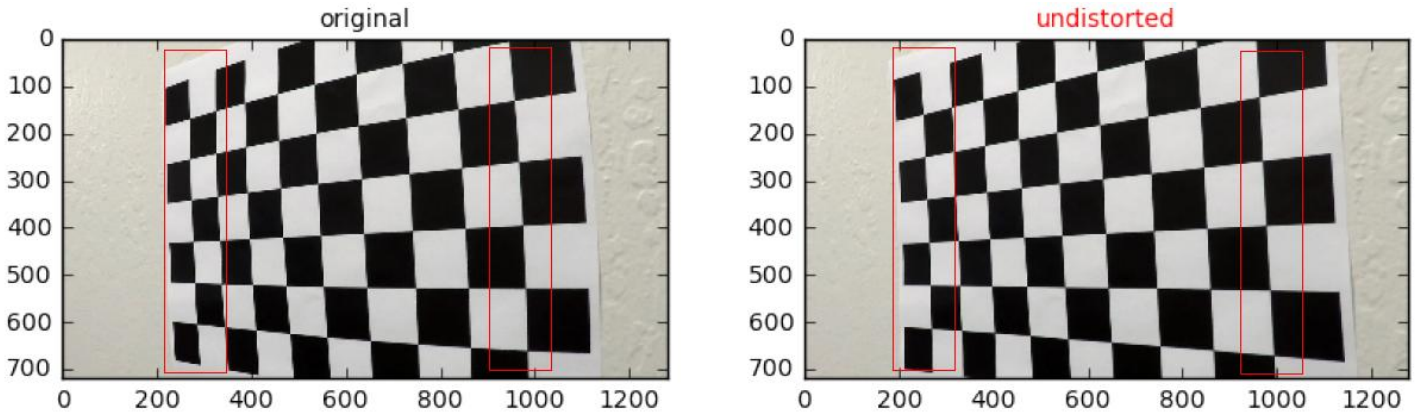
I. Camera Calibration

Camera calibration, usually the first step of computer vision pipeline, is important for all kind of tasks, such as image detection and precision navigation. The camera calibration and images correction routine are based on OpenCV's findChessboardCorners, calibrateCamera, and undistort functions.

The routine uses multiple different poses of a (9, 6) chessboard. The chessboard poses are o.k. to be unknown but they should be in landscape, not portrait, and evenly split to left tilt, right tilt, and direct facing. The higher the number of calibration images the better. In our case, we collect 20 calibration images. The calibration and correction pipeline has three steps, as illustrated below:



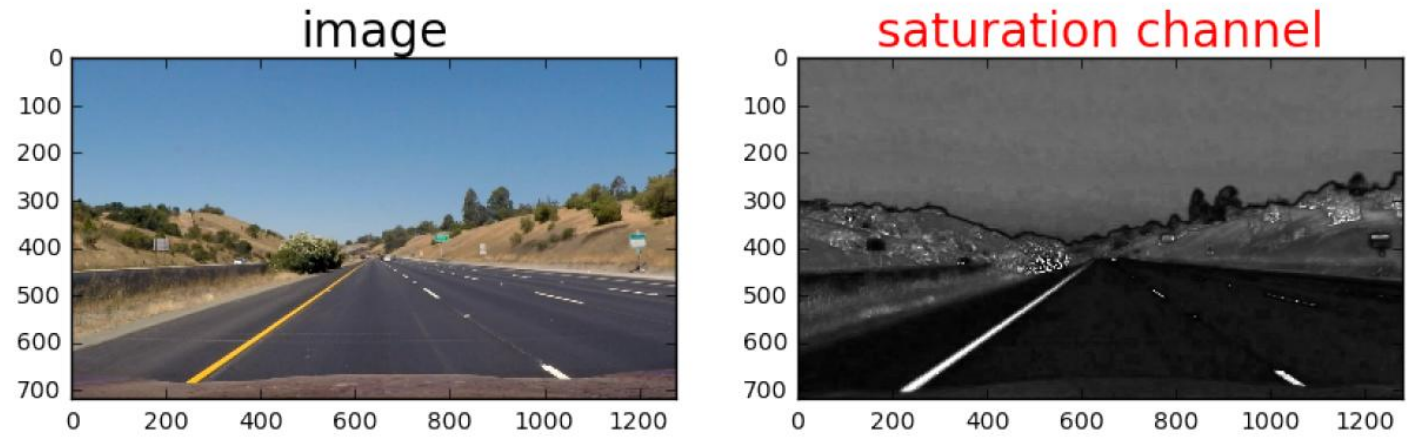
An example of original image and its undistorted version is here:



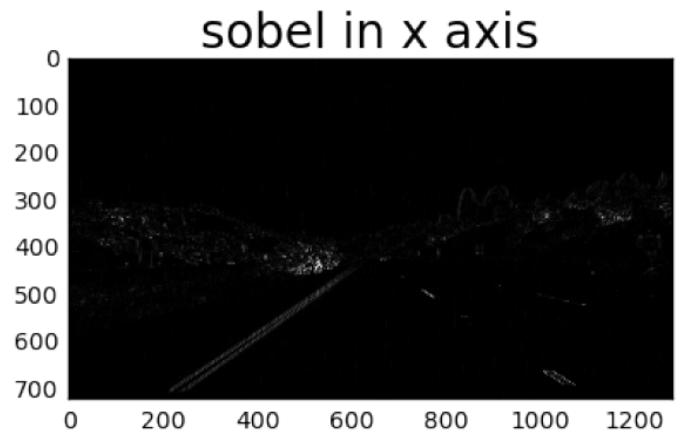
As shown above the distortion corrections are more prominent near the image boundaries. For source codes, please check out the “camera calibration” section of the github python notebook.

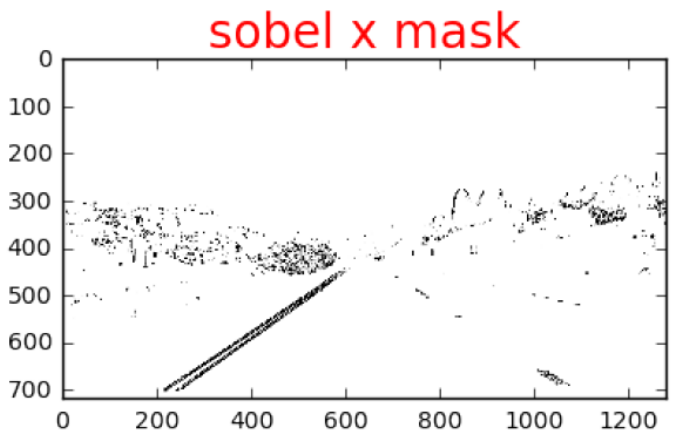
II. Gradients and Color Masks

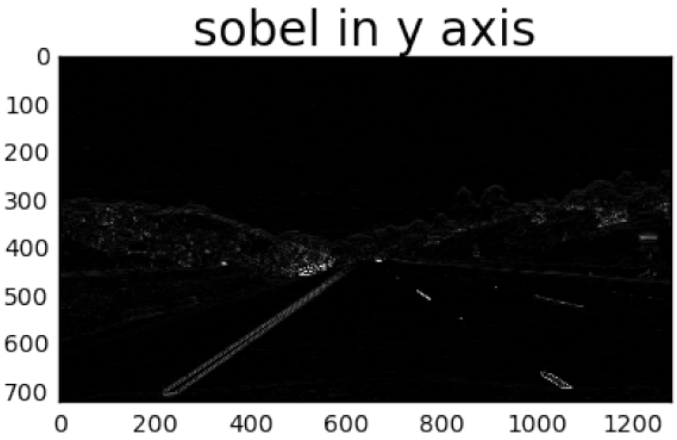
To detect edges, we use color and gradients features. Since R(ed) G(reen) B(lue) is not additive in the domain of human visual perception, which means half of R value does not produce half of the redness a person perceives, I convert RGB to H(ue) L(ightness) S(aturation) color space and use Saturation channel. In HLS cylinder, saturation is the radial dimension on the horizontal hue wheel and perpendicular to the vertical lightness axis. Here is an example of Saturation channel:

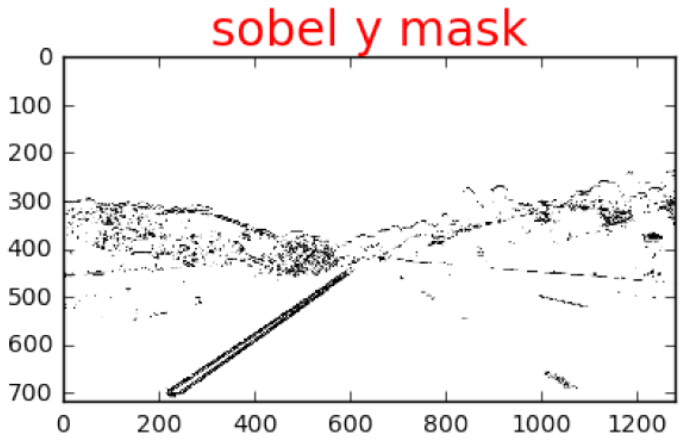


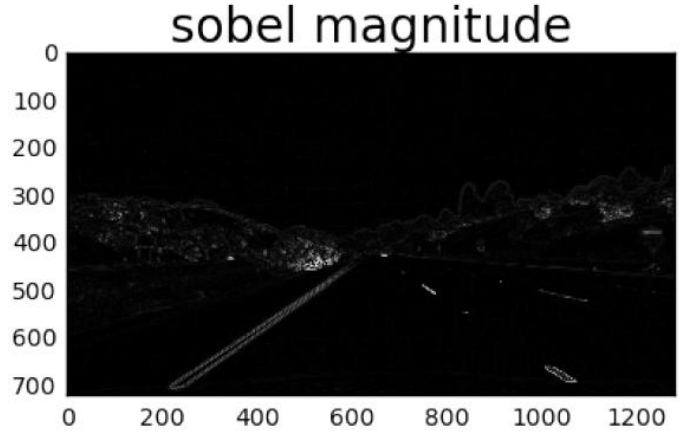
For gradients, I use Sobel for X direction, Y direction, and magnitude. One key difference between Sobel (in Canny) in Project 1 and this project is the thresholding. In Project 1, Canny use hysteresis thresholding. And here I just apply a thresholding band. Below are examples of Sobel X, Y and magnitude on the Saturation channel.

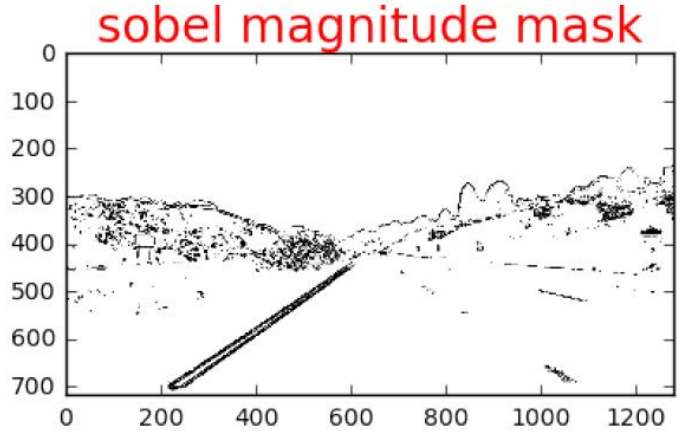












For gradient direction, I choose Scharr over Sobel because Scharr is as fast as Sobel and better in handling noise. Scharr kernels are as the following.

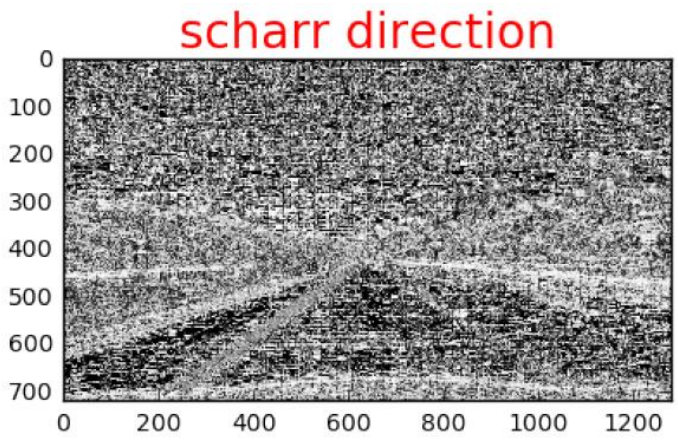
-3	0	+3
-10	0	+10
-3	0	+3

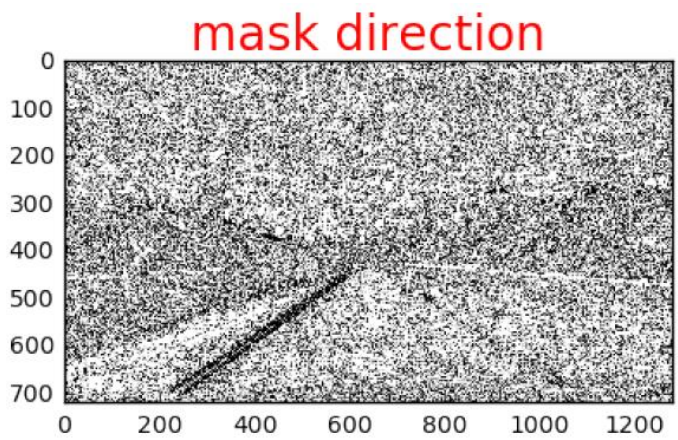
A

-3	-10	-3
0	0	0
+3	+10	+3

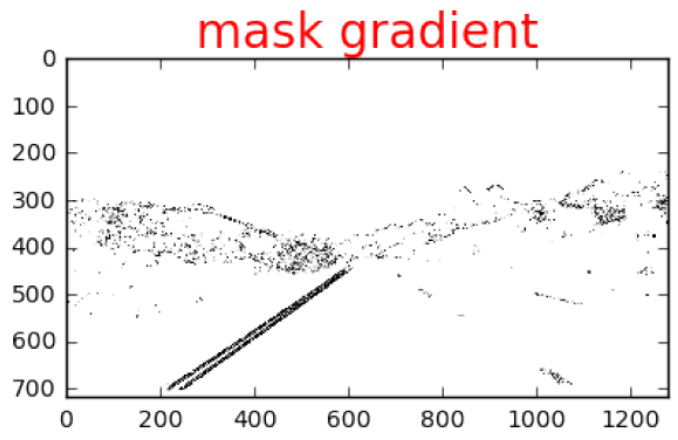
B

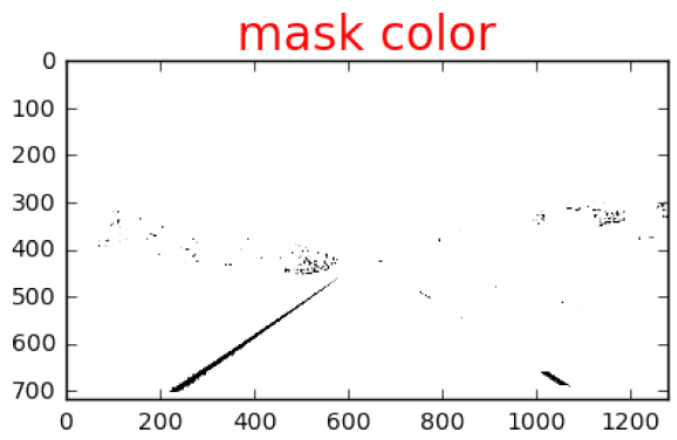
And below is an example of after Scharr kernels applied to the Saturation channel.

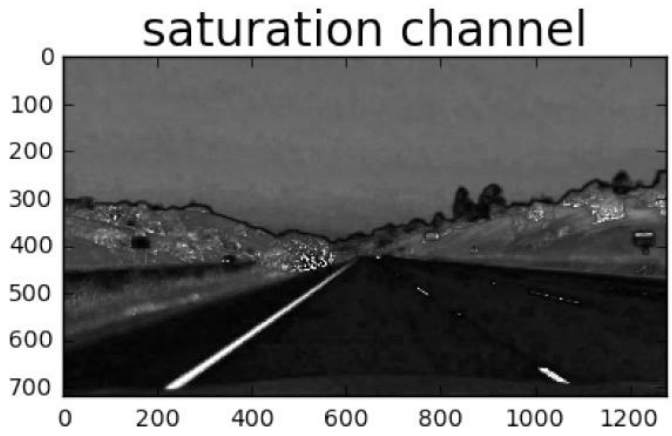


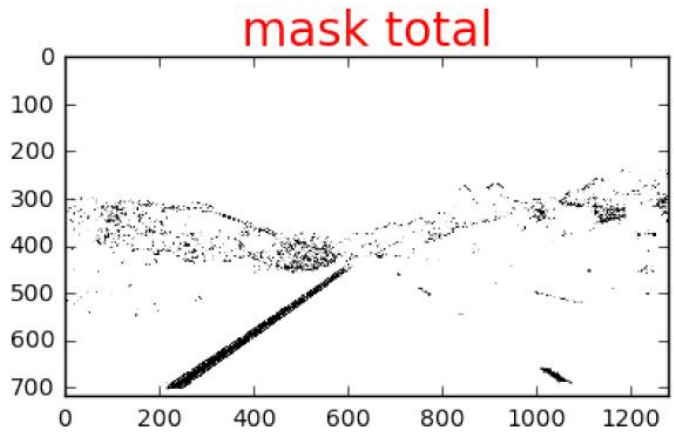


I combine above four masks to get “mask gradient”. And for “mask color”, I use a threshold band [177, 255] on the Saturation channel. Finally, I combine both gradient and color masks to produce a total binary mask for detecting edges. An example of these masks is shown below.









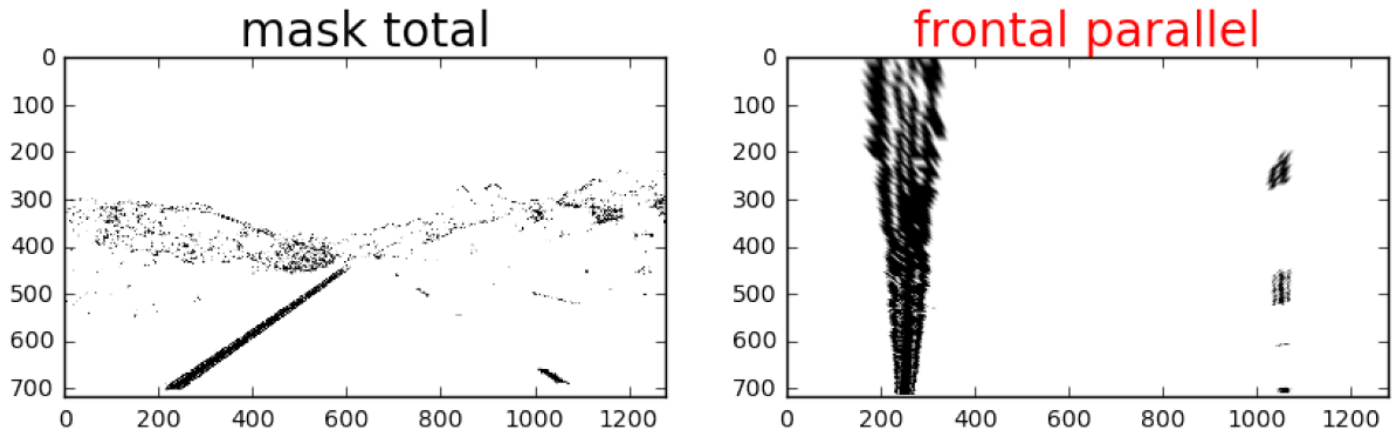
For source codes, please check out the github file “gradients_and_color_mask.py”.

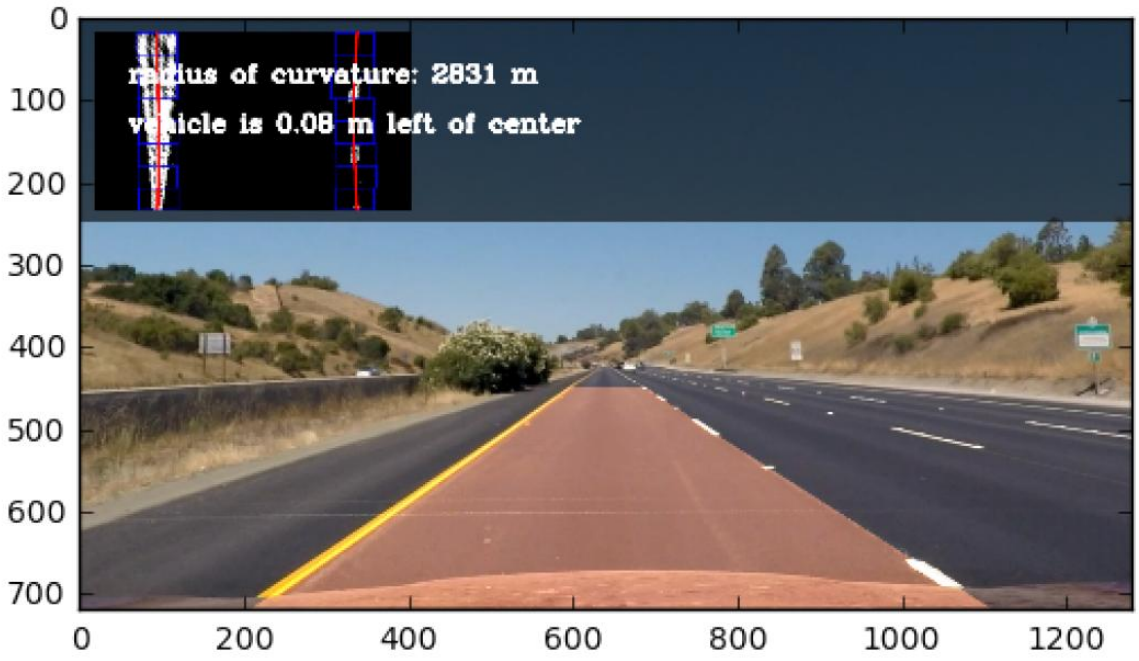
III. Bird’s-Eye View

A bird’s-eye view, aka frontal parallel, is a perspective transformation between two 2D-views. After the transformation, a pair of nonparallel straight lines will be converted back to parallel as in the original 3D physical world. To get a bird’s-eye view, I use OpenCV’s getPerspectiveTransform function. The function has two parameters, the source src and the destination dst. The source, specified by four pairs of (x, y) coordinates, is the area of interest in the original image. The destination, also four pairs of (x, y) coordinates, is a rectangular screen that fits into the size of the original image. Combining the source and the destination, I have four pairs of point correspondences, which are used to calculate the homography matrix and its inverse. Please keep in mind that the order of point correspondences is important in using getPerspectiveTransform function. To simplify the transformation process, I design and use *only one parameter* for tuning. Here are its codes:

```
# provide four point correspondences
knob = 65   # a parameter to make lane marks parallel in dst image
y_top = 450   # kind of at where the horizon is

# four points from the source
# start w/ top left, then clockwise
src_pt_top_left = (w//2-knob, y_top)
src_pt_top_right = (w//2+knob, y_top)
src_pt_bot_right = (w, h)
src_pt_bot_left = (0,h)

src = np.float32([src_pt_top_left, src_pt_top_right,
                  src_pt_bot_right, src_pt_bot_left])

# four poinst from the destination
# start w/ top left, then clockwise
dst_pt_top_left = (knob,0)
dst_pt_top_right = (w-knob,0)
dst_pt_bot_right = (w-knob,h)
dst_pt_bot_left = (knob,h)

dst = np.float32([dst_pt_top_left, dst_pt_top_right,
                  dst_pt_bot_right, dst_pt_bot_left])
```

And below is a sample result.





Getting a bird's-eye view on camera images is required in later data fusion stage with measurements from other sensing devices, such as Lidar. For source codes, please check out the github file "perspective_transform.py".

IV. Lane Finding and Calculations of Curvature and Center Deviation

Lane finding and calculations of curvature and center deviation are based on two python classes: Sliding_Window and Lane_Marks. (Their codes are in github python notebook respective sections.)

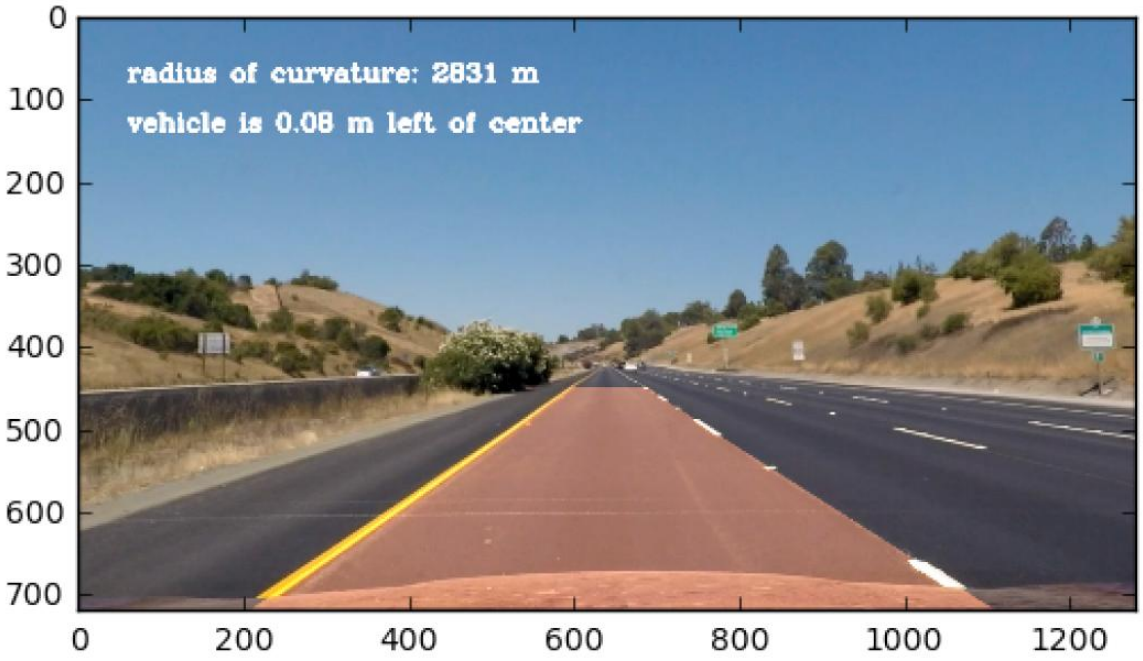
The idea of using sliding windows comes from Page 5 of Lee-Wong-Xiao[2015]. "With the information of the near lanes color and location, we can gradually move the window upward and detect the lane marking within the local window." A sliding window instance contains the window's coordinate information and specifies which edge pixels are inside the sliding window. Sliding windows are stacking up from bottom to the top. The next adjacent sliding window can be shift left or right depends on the count of edge pixels inside. If the count is greater than a certain threshold, the x position of next sliding window is the mean of those internal edge pixels' x position.

Lane_Marks class is the core of the pipeline. It processes the raw image with the total mask then transforms the result into bird's-eye view. It initializes the sliding windows, find the coefficients for left and

right curves, and calculate the curvature and the deviation from the center. It also marks and overlays the curves back to the raw image for display. The coefficients for the curves come from the assumption of 2nd degree polynomials, $C_1 y^2 + C_2 y + C_3$. The radius of curvature R_{curve} is calculated through $R_{curve} = (1 + (2C_1y + C_2)^2)^{3/2} / |2C_1|$, with the assumptions of 27 meters per 720 pixels in the y direction and 3.7 meters per 700 pixels in the x direction. The deviation from center is calculated from finding the offset from the image center with the assumptions that the camera is mounted at the center of the vehicle and 3.7 meters per 700 pixels in the x direction. Here are the codes cut and pasted from the python notebook.

```
def radius_of_curvature(self,coefficients):
    # Define conversions in x and y from pixels space to meters
    ym_per_pix = 27 / 720  # meters per pixel in y dimension
    xm_per_pix = 3.7 / 700  # meters per pixel in x dimension
    # Fit new polynomials to x,y in world space
    points = self.get_points(coefficients)
    y = points[:, 1]
    x = points[:, 0]
    fit_cr = np.polyfit(y * ym_per_pix, x * xm_per_pix, 2)
    return int(((1 + (2 * fit_cr[0] * 720 * ym_per_pix + fit_cr[1])
                ** 2) ** 1.5) / np.absolute(2 * fit_cr[0]))

def camera_distance(self, coefficients):
    points = self.get_points(coefficients)
    xm_per_pix = 3.7 / 700  # meters per pixel in x dimension
    x = points[np.max(points[:, 1])][0]
    return np.absolute((self.w // 2 - x) * xm_per_pix)
```

Below is an example of the result with lane curves, curvature, and center deviation shown.



For the rest of test images output, the video output, and their codes, please check the github python notebook.

V. Discussion

Sliding window is useful but relatively slow. The current pipeline cannot meet the challenge of faster vehicle speed and sharper curves. Most parameters are manually set. At night, a different set of parameters are required.