Advanced Lane Detection

Camera Calibration:

I used the in-built opencv camera calibration toolbox based on yves-bouget's caltech camera calibration toolbox (http://www.vision.caltech.edu/bouguetj/calib_doc/). The procedure is shown in cell 2 of the ipython notebook. First, the corners of the chessboards were obtained from the calibration images using the findChessBoardCorners() function. The world co-ordinates for the same corners were assumed, with their z- coordinates being 0 as all of them lie on the same plane. The camera calibration is internally performed based on joint pose estimation using non-linear optimization. The returned outputs include the camera calibration matrix which are the intrinsic parameters and estimates poses of the different calibration images. The camera matrix is then used to correct the distortion introduced in the camera by the lens. The input image and the undistorted output is shown in the ipython notebook.

Image pre-processing:

First I obtain the gradients in the x and y directions using the sobel operators and then threshold them based on the magnitude and direction. The magnitude threshold is set at values of 0 and 255. The direction threshold is set between angles of 60 and 120 degrees. Then I combine all the thresholded images. This is followed by setting the region of interest based on manual observations in the images based on the rough locations of the lanes.

We then proceed to do color channel processing for the images to enhance the lanes. This is done by first converting the RGB (or BGR in opencv) to HSV color space. We then threshold the saturation channel based on brightness. This is further combined with output of the gradient processed image explained in previous paragraph. In the ipython notebook, the function gradient_processing() desvribes the above paragraph and the function color_processing() describes the current paragraph procedure. The corresponding outputs are also displayed in the ipython notebook.

Inverse Perspective Transformation:

The inverse perspective transformation is obtained using the getPerspectiveTransform() function. It internally uses the discrete linear transformation to obtain a homography between the planes. The perspective transformation retains straight lines and hence is helpful in obtaining a birds-eye view. Here, we are trying to obtain the transformation between road in the image plane and the road in real world. I chose 4 points corresponding to edges of some lanes, calculated their positions in real world wrt car based on highway codes and obtained a homography matrix. I then warped the thresholded image from the previous step to obtain a top view or a birds-eye view as shown in the ipython notebook. The points in image selected were ([284,682],[440,565],[860,565],[1039,682]) and the destination points are selected as ([400,600],[400,500],[800,500],[800,600]). This gives us a conversion value of 100 pixels for every 3 meters in the y direction and 400 pixels for every 3.7m (10 ft) in the x direction. This conversion factor is important while calculating the radius of curvature.

Detection of lane pixels:

I detected the lanes based on a weighting of occupied pixels in my binarized, thresholded IPM image. I consider only the bottom half of the image and calculate a histogram of the occupied pixels. Since, the lanes on the left and right have the highest concentration of white pixels, there are 2 distinct peaks in the

histogram. We assume these to be the starting points of the lane at the bottom of the IPM image. We then divide the image vertically into equidistant parts and assume a margin of 30 pixels on either side of the 2 histogram peaks in the horizontal direction. We store the indices of the occupied pixels in the base division window which are then updated based on the occupied pixels in upper windows. At the end we obtain a set of indices for the left and right lanes. We then use a polyfit() function to fit a second order non-linear curve to the lanes based on the selected indices. The output of the search windows and the set of good indices for the left and right lanes are shown in ipython notebook.

Calculating the radius of curvature:

I calculated the rough radius of curvature using 3 points of the curve from the tutorial described in http://www.intmath.com/applications-differentiation/8-radius-curvature.php . The 3 points I used were the min, mean and max of the vertical pixel values on the fit curve for each lane. In general, the *x*-value for the center of the circle passing through 3 points A (x_1, y_1), B (x_2, y_2), C (x_3, y_3),joined by lines with slopes m_1 and m_2 is given by $xc=(m1m2(y1-y3)+m2(x1+x2)-m1(x2+x3))/(2*(m2-m1))$. The Radius of curvature is now just the radius of the circle with the mentioned center. The pixel to real world distance conversion discussed in the previous segment are used here to obtain the ROC in meters based on detection in pixels. We calculate the radius of curvature for the both left and right lane separately and the take the mean as the ROC. We could have also considered the smaller of the ROC of the 2 lanes to be the resulting ROC. These are calculated and plotted on the image frame. The camera is assumed to be at the center, so we take the mean of the values of the left and right lane and their mean gives the offset from the center of the lane.

Re-projection onto the image plane:

The detected lanes are reprojected from the IPM image back on to the perspective image and the detected lane area in shaded as shown in the output figure in the ipython notebook. This re-projection is done just using the inverse of the transform used to get the IPM image.

Video Pipeline:

The pipeline is similar for each image. I didn't find the need to have temporal filtering for the project_video.mp4 as the outputs look acceptable. Hence, each frame just undergoes the above procedure sequentially for the detections to occur. The speed of processing could be increased by maintain a smaller search area based on the detection in the previous frame, but was not implemented in this project due to a time crunch. The output video is included in the project submission folder along with this write up.

Problems/issues/Failures:

Obtaining the right threshold for the gradient and color processing steps was the hardest to determine to give an acceptable output. I spent a lot of time trying to tune the same parameters. The rest of the pipeline procedure went on smoothly. The current pipeline will fail if there are other distinct edges due to shadows etc very close to the lane lines as in the challenge_video. This can be improved by having an adaptive search space around the peaks of the lane histograms. The pipeline also can't keep up with very small radius of curvature roads like the ones in harder_video, as we are fitting a simple curve and not a higher order curve which would require more processing.