Advanced Lane Finding

Advanced Lane Finding Project

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

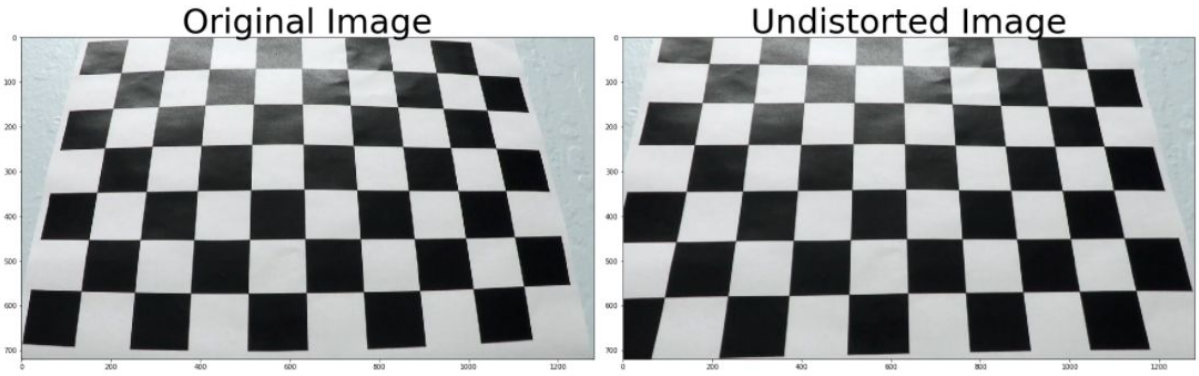
Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the 2nd, 3rd, 4th and 5th code cells of the IPython notebook located in "./examples/Advanced Lane Detection - Final Version Edited.ipynb".

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. The `cv2.calibrateCamera()` function returns the camera matrix, distortion coefficients, rotation and translation vectors. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



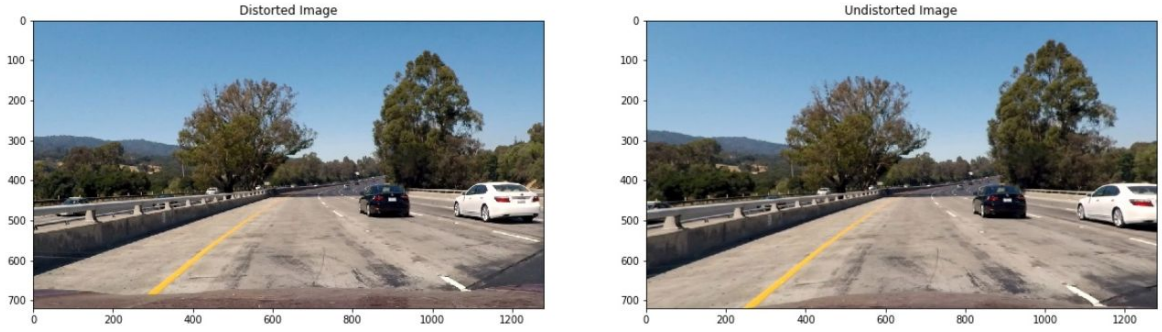
Pipeline (single images)

My pipeline is laid out as follows:

1. Define a class in order to store information from previous image frames.
2. Undistort the camera image.
3. Warp the image to a birdseye view in order to measure the curvature of the road.
4. Apply colour and gradient thresholding to the warped image in order to find the lane marker pixels.
5. Use a histogram to find the start of the lane markers.
6. Use search windows to identify pixels which are most like lane markers.
7. Fit a 2nd order polynomial to the identified pixels.
8. Find the curvature of the road using the 2nd order polynomial.
9. Mark the identified lane on the warped image and then un-warp the identified lane and draw this on the undistorted image.

1. Provide an example of a distortion-corrected image.

To demonstrate this step, I will describe how I apply the distortion to undistort the image below.



I used the cv2.undistort function in the 7th cell inside the MyVideoProcessor() class . This takes in the following arguments; image, mtx, dist, and mtx and returns the undistorted image. The image is the radially distorted image captured by the camera. The mtx is the camera matrix. The dist argument is the distortion coefficients required to correct for distortion. The final mtx (camera matrix) argument can be used to undistort the image more accurately.

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

The information for this section is included in section 3 as I performed a perspective transform first, and then used gradient and colour thresholding to identify the lane markers.

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The first step I took was to warp the image. This is carried out inside the warp(img) function. I defined the source and destination points myself as follows:

top_left_src = [588, 454]

top_right_src = [695, 454]

bottom_right_src = [1135,bottom]

bottom_left_src = [185,bottom]

top_left_dst = [320, 1]

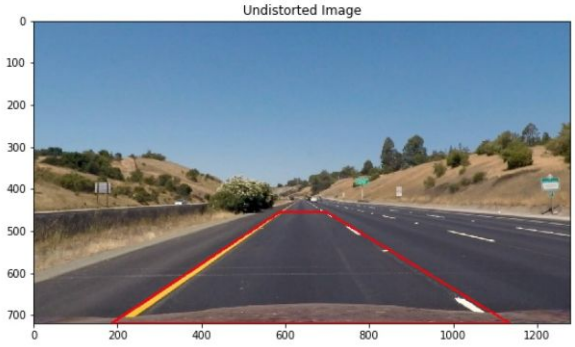
top_right_dst = [920, 1]

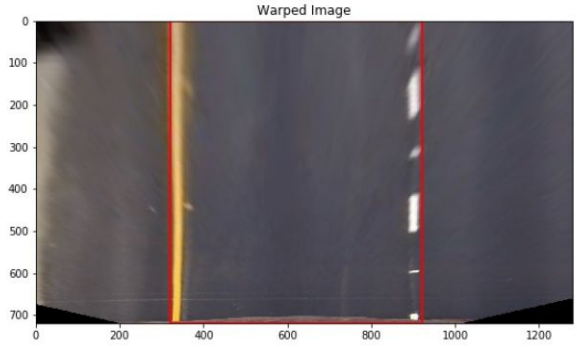
bottom_right_dst = [920, 720]

bottom_left_dst = [320,720]

The perspective transform matrix M is got by feeding the source and destination points to the cv2. getPerspectiveTransform() function. The warped image is then found by feeding the image, perspective transform matrix M, image size, and the interpolation method (linear in this case) to the cv2.warpPerspective() function.

I verified my source and destination points by drawing them on the image as can be seen below. The original image is the undistorted camera image.





I found that tweaking my source point to match up exactly with the point where the lines intersect with the x axis in the undistorted image improved my lane detection. The source points which I transformed originally made the right lane marker look like it was curved.

I then used a combination of color and gradient thresholds to generate a binary image from the warped image.

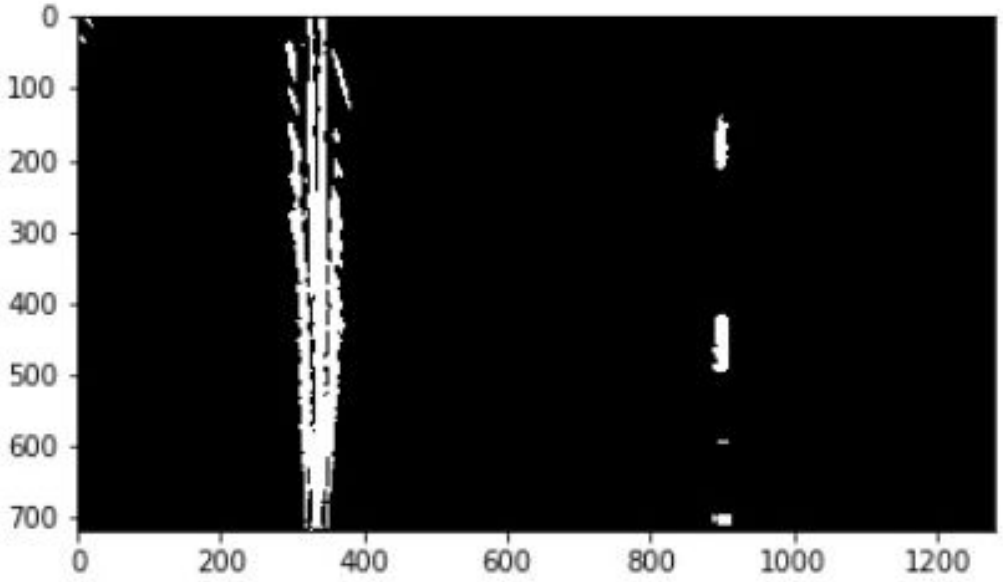
I used colour thresholding initially to try and detect the lane markers. I converted the image from RGB colour space to HLS colour space and also to HSV colour space. I used the Saturation (S) and Lightness (L) channels initially. The S channel was good at detecting the yellow lane boundary as saturation is a measure of colourfulness. The

yellow line had a high saturation value. I initially used the L channel but I then changed to the V channel as it was more robust to bright sunlight. The threshold values can be seen in the code below.

For gradient thresholding, I applied a sobel filter in the x direction to the S channel. I initially tried this on the gray scaled image but there was a lot of noise coming from changes in colour of the tarmac and also marking in the tar in the vertical direction.

```
# Threshold x gradient
thresh_min = 20
thresh_max = 100
sxbinary = np.zeros_like(scaled_sobel)
sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

# Threshold color channel
s_thresh_min = 190
s_thresh_max = 255
s_binary = np.zeros_like(s_channel)
s_binary[(s_channel >= s_thresh_min) & (s_channel <= s_thresh_max)] = 1

# Threshold l color channel
  l_thresh_min = 200
  l_thresh_max = 255
  l_binary = np.zeros_like(l_channel)
  l_binary[(l_channel >= l_thresh_min) & (l_channel <= l_thresh_max)] = 1

# Threshold v color channel
v_thresh_min = 230
v_thresh_max = 255
v_binary = np.zeros_like(v_channel_hsv)
v_binary[(v_channel_hsv >= v_thresh_min) & (v_channel_hsv <= v_thresh_max)] = 1
```

Here's an example of my output for this step:



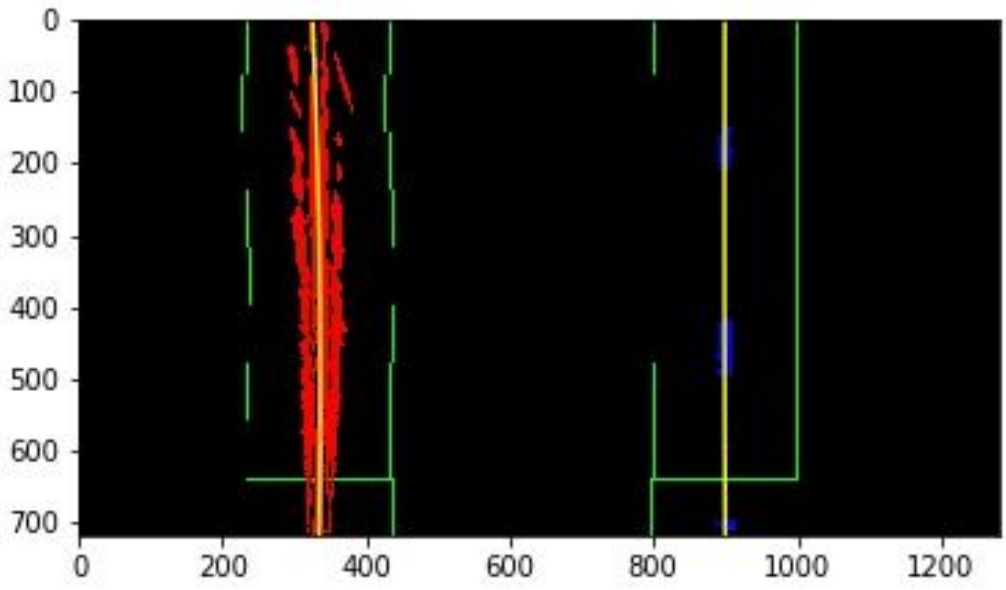
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

I then applied a region of interest mask to the image to only search for lane markings in the current lane. This was carried out in the function called region_of_interest().

I then took a histogram of the bottom half of the image and found the peaks of the left and right side of the histogram in order to find the starting points of the left and right lines. The indices of the highest peaks on the left and right side of the histograms indicate the starting points of the left and right lane lines in the x direction.

I then draw a search window around the identified x coordinates from the histograms. I search for any non-zero pixels inside these windows. I store the indices of these pixels in an array. If I find the minimum amount of lane pixels I use these indices to centre my next search window in the mean position of these indices.

I then concatenate the left and right lane indices found in the current frame and fit a second order polynomial to them. This section gave the following output:



For the first frame in the video I use the search windows to find the pixels inside the search windows. Once I have found the lane markers in the first fram, I search for lane pixels inside the next frame around the 2nd order polynomial from the previous frame. If the slope of the line increases beyond a certain value then I revert to searching for lane pixels using the search windows.

To try and smooth the output of my lane detection algorithm I store the polynomial coefficients from the previous 10 frames. Once I have processed the first ten frames in the video feed, I take the average of the polynomial coefficients of the current frame and the previous ten frames and use these to plot the 2nd order polynomials in the current frame as can be seen below:

```
# If there are sufficient coefficients from the previous frames,
# use the average of these coefficients to smooth the lane detection output.
if len(self.recent_left_fits) >= self.smooth_factor:

    left_fit = np.average(self.recent_left_fits[-self.smooth_factor:], axis = 0)
    right_fit = np.average(self.recent_right_fits[-self.smooth_factor:], axis = 0)

    self.recent_left_fits.append(left_fit)
    self.recent_right_fits.append(right_fit)

# Generate x and y values for plotting
ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )

# Find the slope of the identified lane markers at every point
# along the lane markers
left_slope = 2*left_fit[0]*ploty + left_fit[1]
right_slope = 2*right_fit[0]*ploty + right_fit[1]

# Find the difference in slopes between the left and right lines.
# Lines with similar slopes are approximately parallel.
slope_diff = (left_slope) - (right_slope)
slope_diff = abs(slope_diff)

# Write the second order polynomials for the right and left lane markers. Ax^2 + By + C.
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]
```

5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I calculated the curve radius using the following formula presented in class:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

I used the code presented in the class to implement this:

```
# Define y-value where we want radius of curvature
# I'll choose the maximum y-value, corresponding to the bottom of the image
y_eval = np.max(ploty)
left_curverad = ((1 + (2*left_fit[0]*y_eval + left_fit[1])**2)**1.5) / np.absolute(2*left_fit[0])
right_curverad = ((1 + (2*right_fit[0]*y_eval + right_fit[1])**2)**1.5) / np.absolute(2*right_fit[0])
```

I calculated the offset from the centre using the following code:

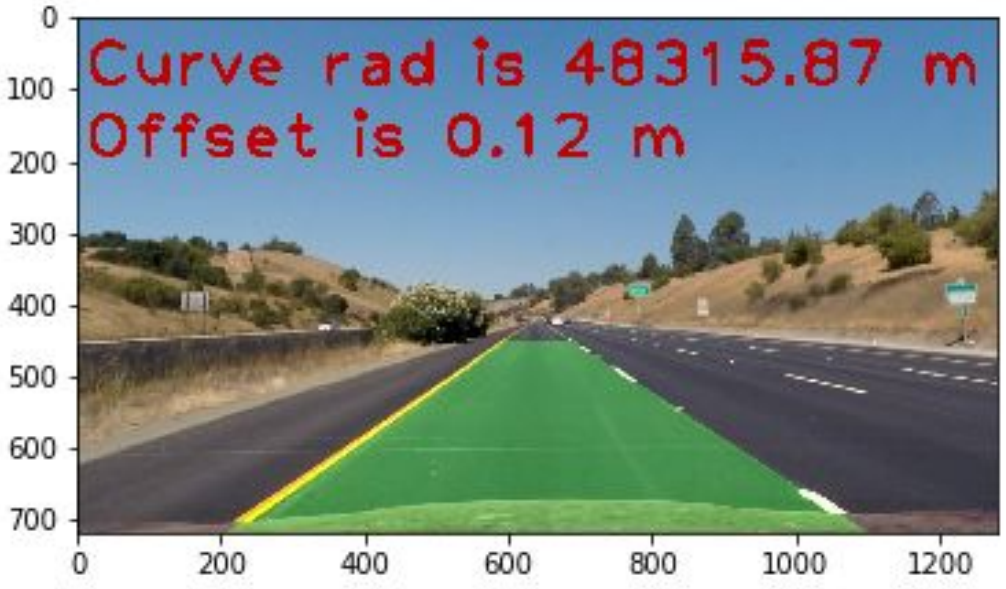
```
camera_pos = img.shape[1]/2
lane_center = (left_fitx[719] + right_fitx[719])/2

center_offset_pixels = abs(camera_pos - lane_center)
center_offset_m = center_offset_pixels*xm_per_pix
center_offset_m = round(center_offset_m,2)
```

right _fitx and left_fitx are the starting positions of the right and left lane markers respectively. xm_per_pix is the the number of metres in a pixels.I firstly calculate the position of the centre of the lane in the current frame. The vehicle's offset from the centre of the lane is the difference between the current camera position and the lane centre.

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

I implemented this step using the code from the classes as follows:

```
# Create an image to draw the lines on
warp_zero = np.zeros_like(binary_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 (Minv)
newwarp = cv2.warpPerspective(color_warp, Minv, (img.shape[1], img.shape[0]))

# Combine the result with the original image
result = cv2.addWeighted(undist, 1, newwarp, 0.3, 0)

cv2.putText(result, 'Curve rad is {0} m'.format(avg_curverad), (10, 100), cv2.FONT_HERSHEY_PLAIN, 6, 200, 8)
cv2.putText(result, 'Offset is {0} m'.format(center_offset_m), (10, 200), cv2.FONT_HERSHEY_PLAIN, 6, 200, 8)
```



Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

My video is in the project submission folder: Submission/project_video_part_4Mar18_1

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

I performed the lane detection in a slightly different order to the one suggested in the lectures. Firstly I warped the image to get a bird's eye view of the road. I then performed the gradient and colour thresholding on that image. This may not be the best approach as the top of the warped image does not always give a clear view of the lane markings.

In order to make the output more smooth, I used the average of the polynomial coefficients of the previous ten frames. This may not work as well on roads with sharper turns.

In the event where the algorithm detects that both lane markers are not parallel (indicating that one line is not actually a line), I use the polynomial coefficients from the previous frames to mark the lane in the current frame. When the lane markers are not

detected as parallel this usually means that the algorithm is incorrectly detecting shadows on the road or other changes in the road appearance as lane markers.

I detect whether lines are parallel by finding the slope at every point along the line and then finding the difference between the slopes. If the difference is too large, then this indicates that one of the lines is not being correctly detected. I originally attempted to use the lane width instead of the road slope but I did not find it very robust. Also this would struggle on roads of varying widths.

Warping the image from the undistorted camera image to the birdseye view also had an effect on the performance of the algorithm. I had to calibrate my source points and destination points to ensure that the straight road in the test image appeared straight in the transformed image. There was a slight curve in the transformed image originally and this affected the performance of the lane detection algorithm.

In order to make my algorithm more robust, I would try to get more useful information from the H channel, and try to rely less on the V channel as the information in this channel can change a lot under varying light conditions.