Advanced Lane Finding

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.

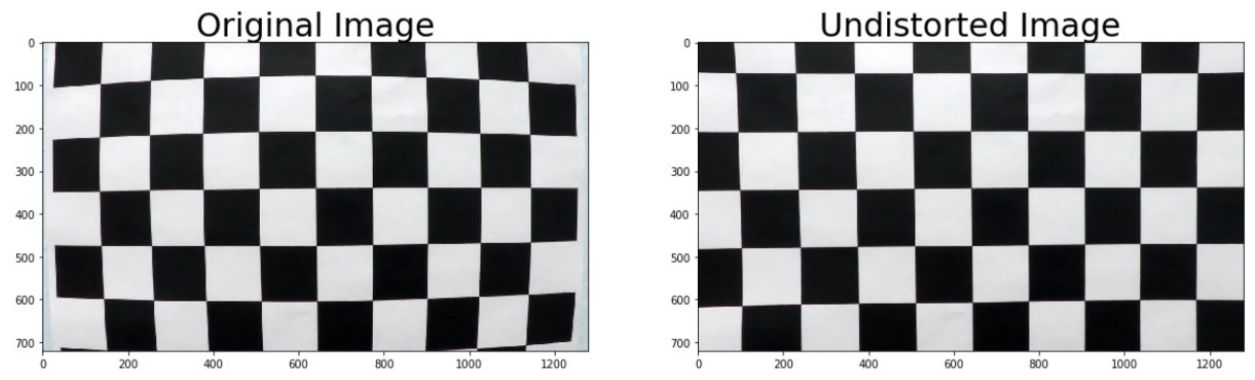
Camera Calibration

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 code cell ([2], [3], [4], [5]) of the 'advanced_lane_detection.ipynb ' ipython notebook.

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. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

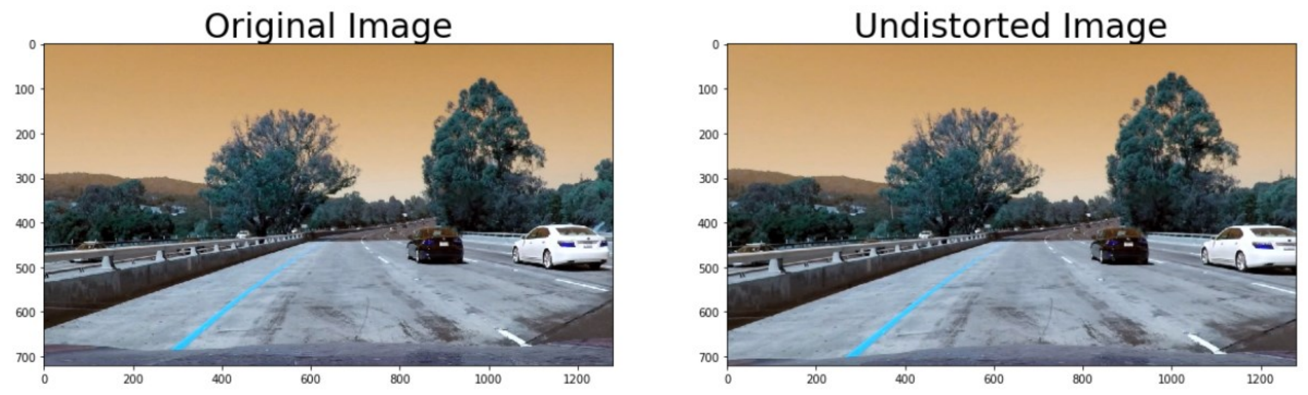


Pipeline (single images)

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

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

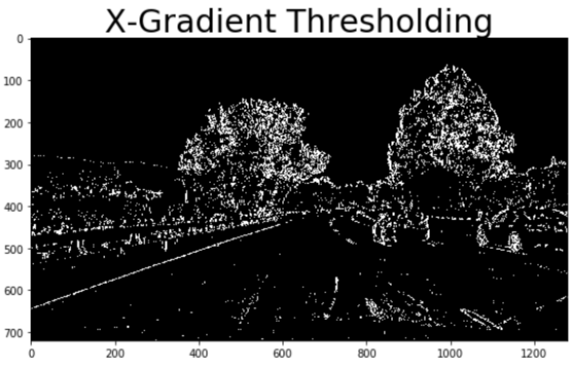
The camera calibration results saved in a pickle file to undistort the original distorted images were used to correct the distorted images. 'undistort(image)' function is used to load the pickle file, read the calibration and distortion co-efficients from the pickle file and to undistort the image.

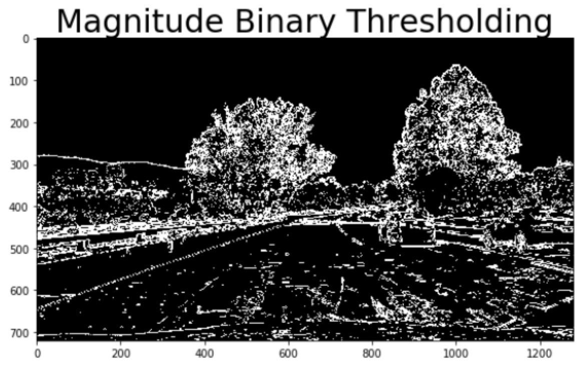


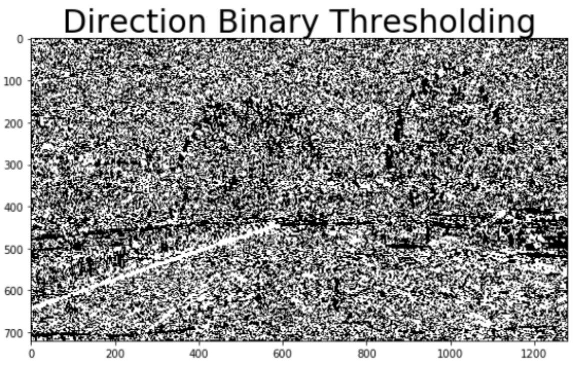
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

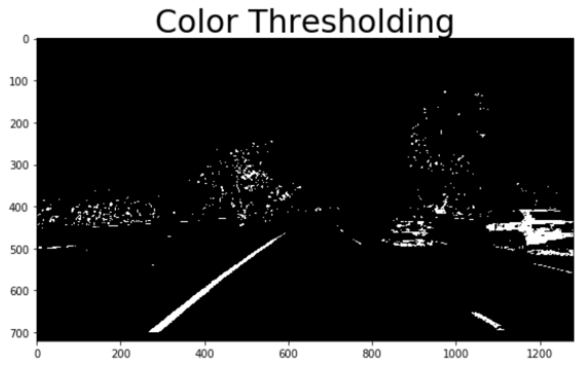
I have used both gradient and color thresholding to create thresholded binary image. The get_thresholded_binary_image () function (code [7]) can be found in the ' advanced_lane_detection.ipynb ' ipython notebook.

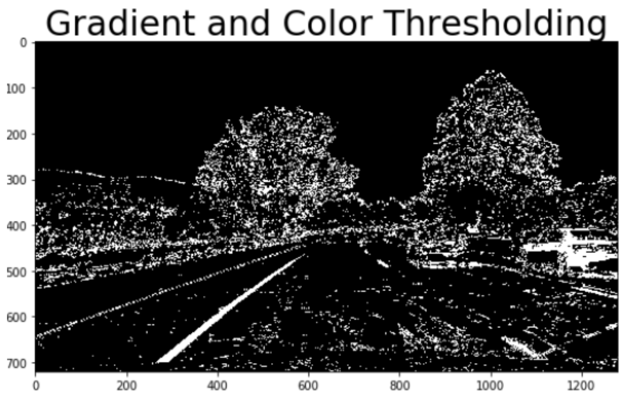
Initially I undistorted the image. In the second step, I calculated the gradient binary threshold of undistorted image in both x and y orientations. Gradient threshold were calculated by using Sobel operator. In the next step, I calculate the magnitude binary threshold, direction binary threshold and color threshold. Combinations of these thresholds were calculated to obtain the thresholded binary image.











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

Perspective Transform is implemented by using a function **get_perspectiveTransform**() which can be found in the ' advanced_lane_detection.ipynb ' ipython notebook.

Perspective Transform of an image can be obtained by using an OpenCV function cv2.warpPerspective(). We need to pass image, size of the image and source and destination points of an image for which we need birds eyes' perspective. It is an important step to identify the curvature of the lane lines.

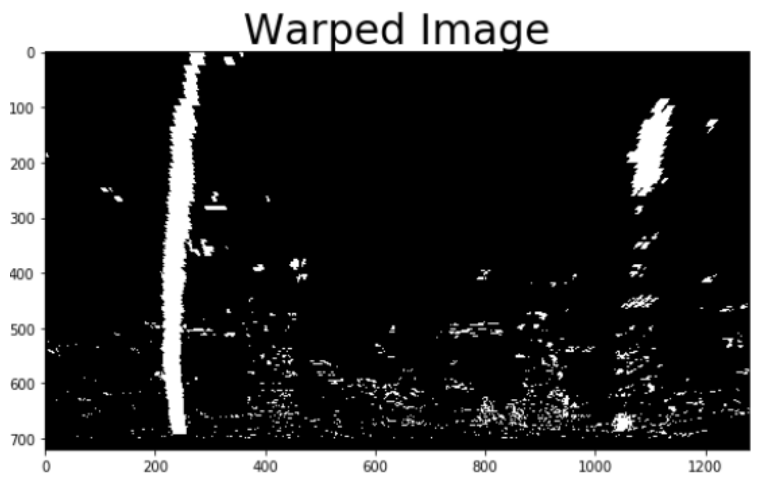
The following source and destination points were selected to obtain the perspective transform.

I chose the hardcode the source and destination points in the following manner:

~~~
```python
src = np.float32(
    [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
    [((img_size[0] / 6) - 10), img_size[1]],
    [(img_size[0] * 5 / 6) + 60, img_size[1]],
    [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
dst = np.float32(
    [[(img_size[0] / 4), 0],
    [(img_size[0] / 4), img_size[1]],
    [(img_size[0] * 3 / 4), img_size[1]],
    [(img_size[0] * 3 / 4), 0]])
```
~~~

Source Points	Destination Points
585, 460	320, 0
203, 720	320, 720
1127, 720	960, 720
695, 460	960, 0

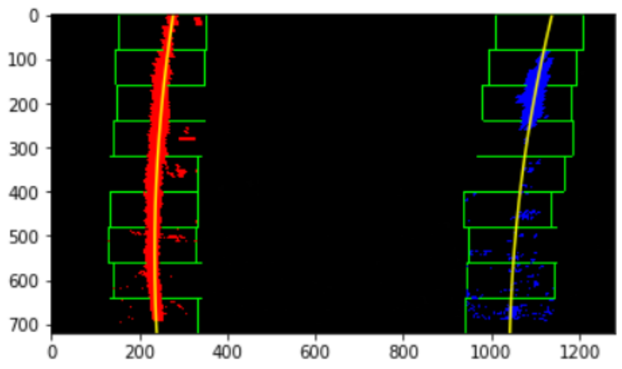
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



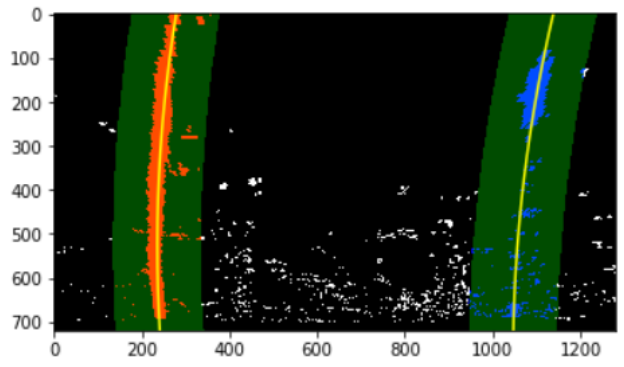
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The code for this step is contained in the code cell [31] of the 'advanced_lane_detection.ipynb ' ipython notebook. This is implemented by using the function 'find_lane_pixels()' and 'fit_polynomial()'.

I used Sliding Window Technique and fit my lane lines with a 2nd order polynomial kinda like this:



Lane lines don't necessarily move a lot from frame to frame. Hence, we do not need to apply Sliding Window Technique to each frame. Instead we can just search in a margin around the previous line position. This is implemented by using a function search_around_poly() in the python notebook[code 41].

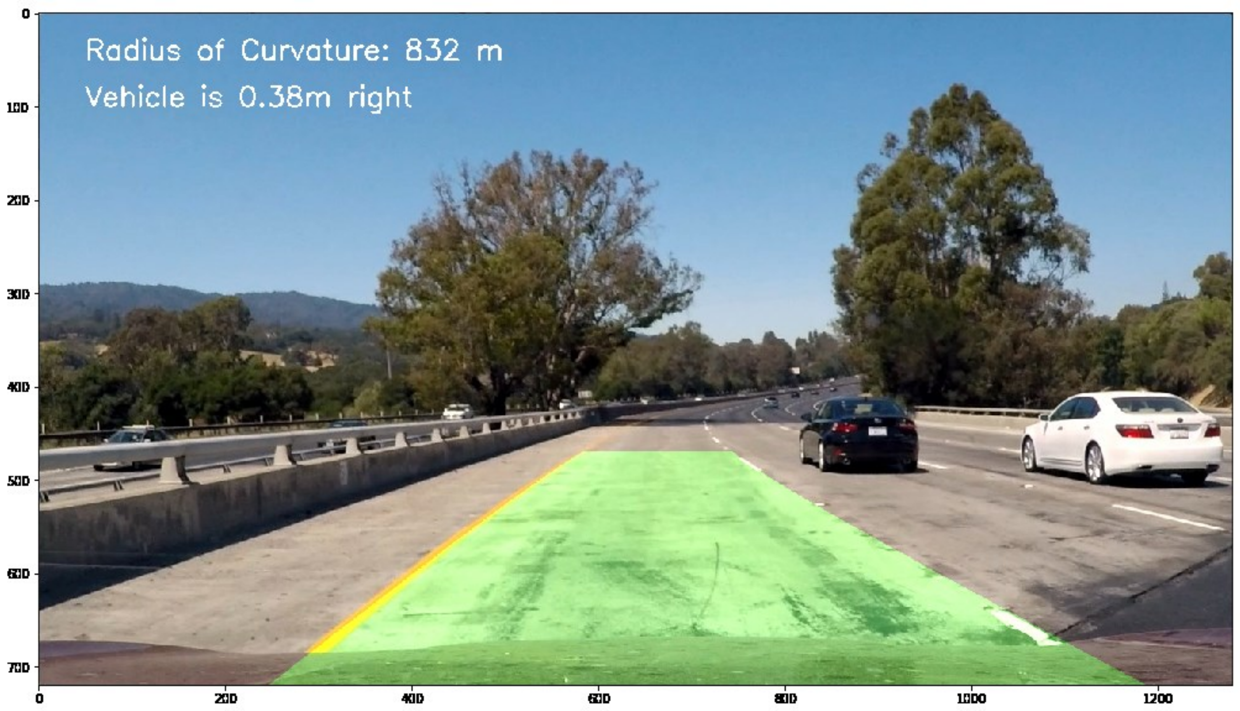


5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

The code for this step is contained in the code cell [34] of the 'advanced_lane_detection.ipynb ' ipython notebook. This is implemented by using the function 'measure_radius_of_curvature ()'

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

The code for this step is contained in the code cell [36, 37, 38] of the 'advanced_lane_detection.ipynb ' ipython notebook.



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!).

Here's a [link to my video result](./output_video/project_video.mp4)

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?

Pipeline likely fail in the below scenarios:

1. For video 'challenge_video.mp4', pipeline is not working.

2. For 'harder_challenge_video.mp4', lane lines are not detected properly. This video is not a highway video. This video consists of more number of curves and radius of curvature of lane lines vary abruptly. Pipeline is not robust to adapt sudden changes in radii of curvature.

Challenges faced:

1. In the current pipeline, Perspective transform is applied after binary thresholding. If the pipeline is going to get implemented in hardware, it is better to apply binary thresholding after perspective transform. I tried to implement the same, but facing challenge in implementing the same.

2. Faced challenges in smoothening the curves for lane lines in case of non-highway videos.

Methods to improve:

1. The src and dst points for perspective transform are hardcoded. Dynamic calculation of src and dst points would make the pipeline more robust. To dynamically calculate the src and destination points for perspective transform, we can use Hough Transform to initially identify the lane segment for each lane line and by using the end points of the lines as co-ordinates for src and dst points.

2. We can make the pipeline more robust by taking weighted average of previous detections to smoothen the lane lines.

3. Shadowy images seems like more noisy. Need to implement filters or some other methods to reduce noise.