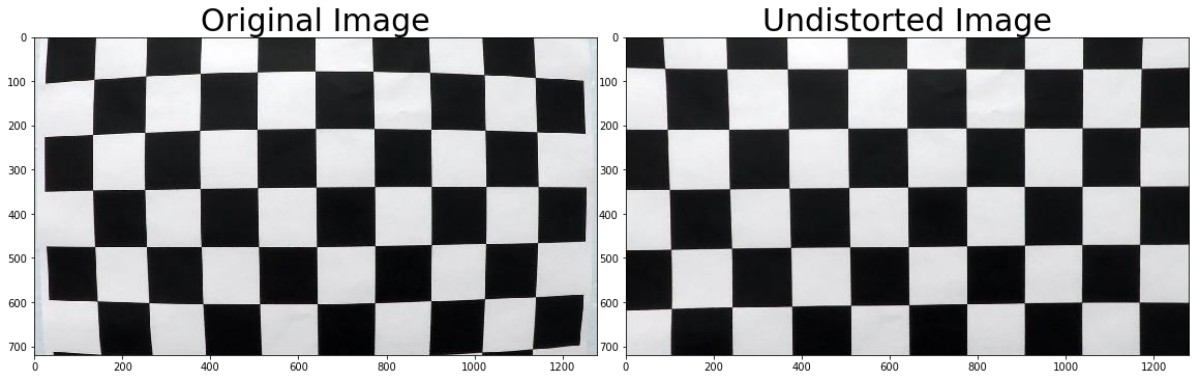
**Camera Calibration**

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

The first step is to take 20 images of the same chessboard from different angles and use *findChessboardCorners* from OpenCV to find all image chessboard corners of those images. The second step is to use *calibrateCamera* from OpenCV and take the image chessboard corners array and object points array to form the camera calibration parameters. The object points are essentially the 3D location (indices) of those chessboard corners. The third step is to feed all the camera calibration parameters into *undistort* from OpenCV to perform correction on each distorted image. The camera calibration will remind the same for a given camera and can perform correction for all the images taken from this camera.

The images below depict the original image and corrected image of the road.



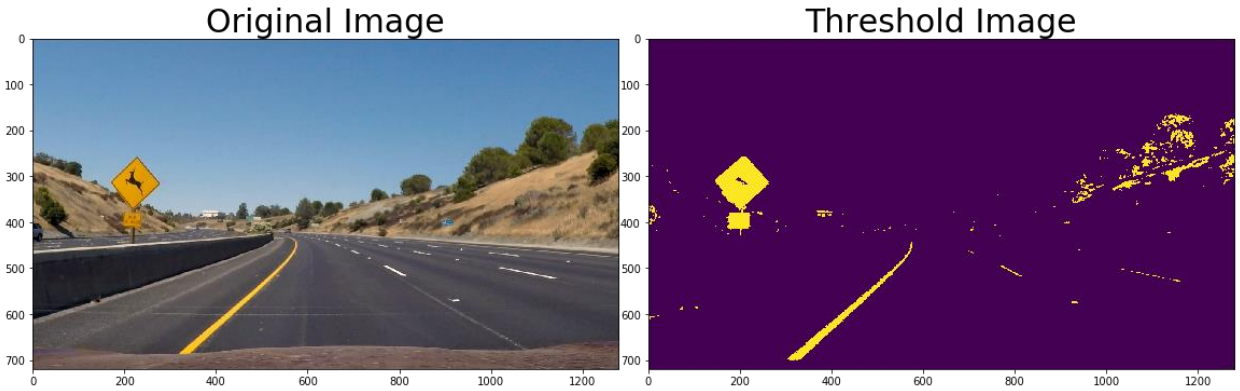

**Pipeline (single images)**

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

It is hard to spot the difference between these two images, but we can easily notice the subtle difference in the shape of the car hood at bottom.

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

   I used a combination of color threshold and gradient threshold methods to find the lane edges along the road. I applied the saturation threshold, lightness threshold, gradient magnitude threshold and gradient direction threshold to the images with specific threshold ranges. The areas that are beyond the threshold ranges are set to zero in the output binary image. The comparison of original image and thresholded image is depicted below. The lane marks are clearly highlighted in the binary image. Beyond the lane marks, the trees and traffic sign are also highlighted. But they will be filtered out after perspective transformation.

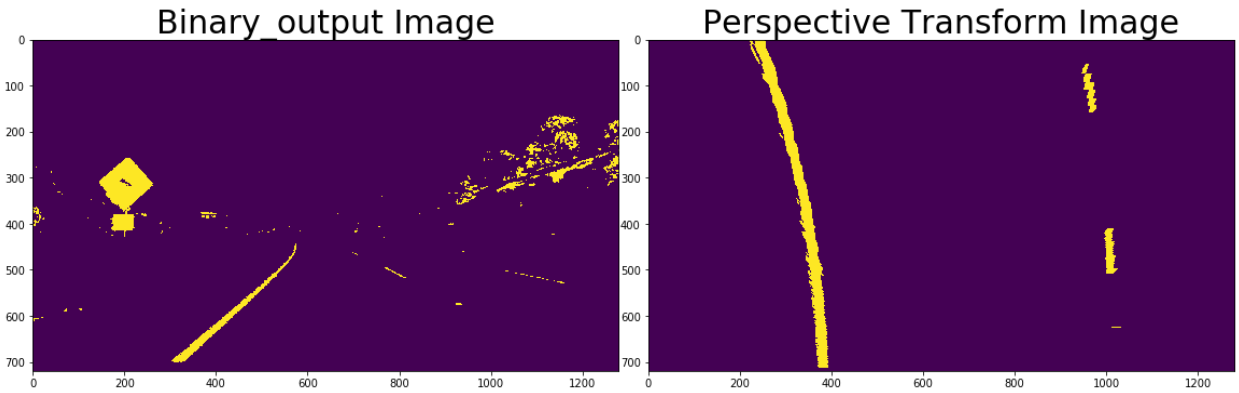



3. **Describe how (and identify where in your code) you performed a perspective transform provide an example of a transformed image.**

   The first step is to find the source points and destination points from the image to calculate the transformation matrix. Those points are listed as below:

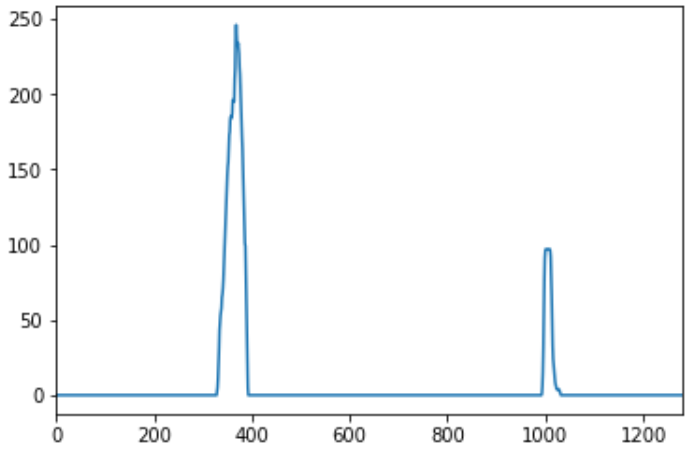
| Source | Destination |
|---|---|
| 585, 460 | 320, 0 |
| 203, 720 | 320, 720 |
| 1127, 720 | 960, 720 |
| 695, 460 | 960, 0 |

   The second step is to use the *getPerspectiveTransform* from OpenCV and take the above source points and destination points to calculate the transformation matrix. The third step is to use *warpPerspective* from OpenCV to transform the original image to bird view image by feeding in the transformation matrix. Below is the comparison of original image and bird view image.

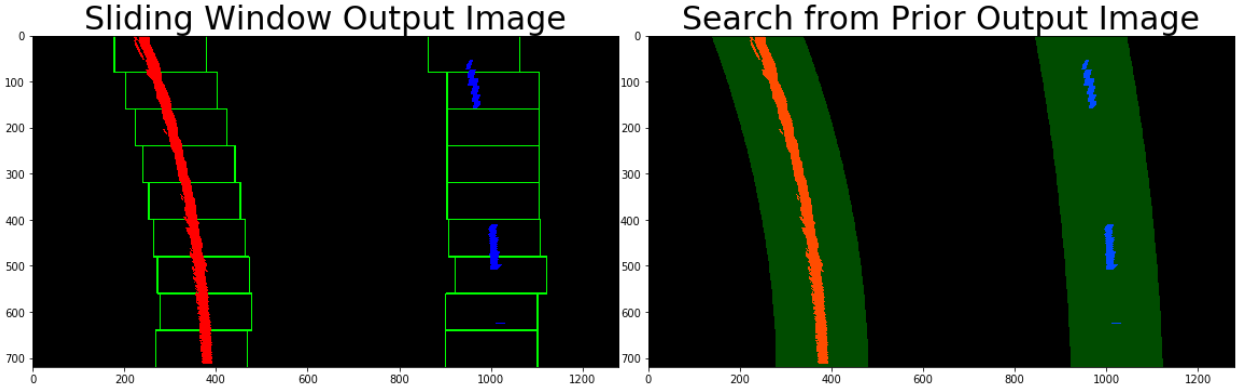



**4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?**

Initially, the algorithm will find the histogram of image along the horizontal direction. The two peaks of the below histogram corresponds to the two lanes.



Once the two lanes starting positions are detected, the second algorithm defines a rectangle sliding window and slides upward to find the rest of the lanes points.



The left image above draws out the sliding windows and the trajectory of sliding window. The third step is to highlight all the nonzero points inside each sliding window and feed them into polynomial line fitting. Since the lanes shape will not change abruptly in adjacent frames, it is not necessary to apply sliding window algorithm to each frame. Instead, we can utilize all the polynomial lines from previous frame and highlight all the nonzero points near those two polynomial lines. We fit two new polynomial lines along those new highlighted nonzero points. In the right image above, we can see that the nonzero points inside the green zone are highlighted.

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 radius of curvature of the left and right lanes are calculated in terms of the following equations

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

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

The curvature function takes the two polynomial parameters A and B as well as the y value. Since we only focus on the lanes curvature near the car, we feed the biggest y value to the function, which is *y_eval=undist.shape[0]*.

The car center relative to the lanes center is calculated in terms of the following code.

```
xm_per_pix = 3.7/378
line.line_base_pos = ((left_fitx[-1] + right_fitx[-1])/2.0-undist.shape[1]/2.0)*xm_per_pix
```

The car center is the image center given the assumption that the camera is mounted at the center of the car. The lanes center is calculated by taking the average of the left and right lanes x position. The ratios between the real world coordinate and pixel coordinate are 3.7/378 and 3.048/100 for x and y directions respectively. These two ratios can convert the pixel coordinate to real world coordinate.

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

Below is an example image that highlights the lane area and indicates the road curvature and car position relative to the lane center. The green area is drawn between the two fitted polynomial lines from the bird view. It is then transformed back to the original front view through the perspective transform. The curvature for the image is 3.70 km and the car center relative to the lane center is 0.59 m


**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!)**
   Link to the Video

**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?**

The first issue I encountered is the improper calibration of camera. This could cause the two fitted lines change abruptly when the car is switching from pitch road to cement road. The second issue I faced is the improper use of threshold. The function will output an all zeros binary image if the all the thresholds are combined in terms of *and* logic. To resolve it, we need to use *or* logic to combine all the thresholds.

The third issue I faced is the improper definition of sanity check tolerance. If the allowable tolerance of curvature deviation is too small, the algorithm will keep dropping the newest measurement. In the video, we can notice that the two fitted polynomial lines will be changing slowly. To resolve this problem, we can tune the tolerance and reapply sliding window algorithm to search for the lanes if the newest measurement is dropped.