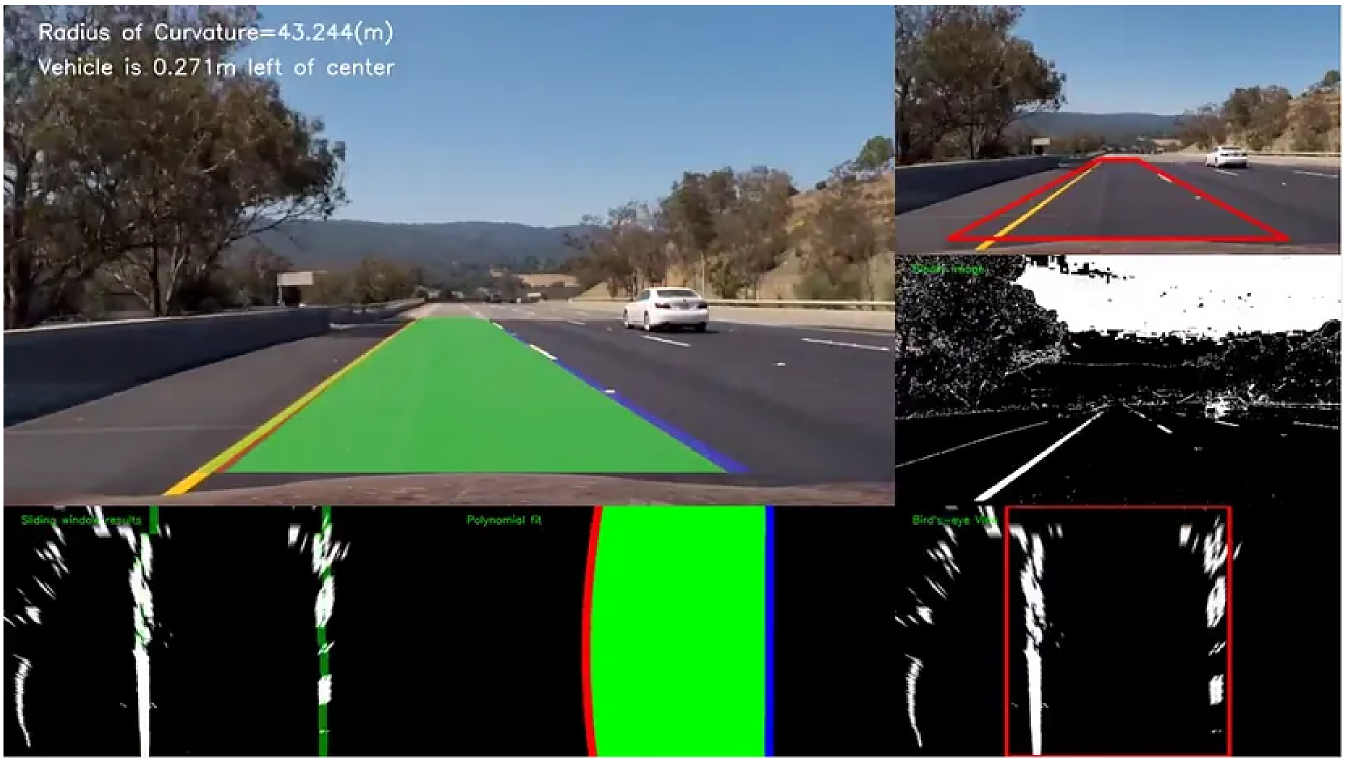
Advanced Lane Detection for Autonomous Vehicles using Computer Vision techniques

Raj Uppala · Follow
Published in Towards Data Science
7 min read · Aug 2, 2017


Radius of Curvature=43.244(m)
Vehicle is 0.271m left of center
Sliding window results
Polynomial fit
Bird's-eye View

In an earlier *project*, I used Canny and Hough transforms with gradients to detect changes in color intensity and confidence levels respectively, to detect lane lines. This project uses advanced techniques that builds on the earlier one by using thresholds for different color spaces and gradients, sliding window techniques, warped perspective transforms, and polynomial fits to detect lane lines. If you are interested in the end result alone, jump to the *video*. For a detailed discussion, read on.

The series of steps to accomplish the goal for this project are as follows:

1. Compute the camera calibration matrix and distortion coefficients.

2. Apply a distortion correction to raw images.

3. Use color transforms, gradients, etc., to create a thresholded binary image.

4. Apply a perspective transform to generate a "bird's-eye view" of the image.

5. Detect lane pixels and fit to find the lane boundary.

6. Determine the curvature of the lane and vehicle position with respect to center.

7. Warp the detected lane boundaries back onto the original image and display numerical estimation of lane curvature and vehicle position.

Let's delve deeper into each of the above steps to understand the process in detail.

1. **Compute the camera calibration matrix and distortion coefficients.**

All cameras use lenses and one of the problems with lenses is that they have some radial distortion. To remove this distortion, I used OpenCV functions on chessboard images to calculate the correct camera matrix and distortion coefficients. This can be achieved by finding the inside corners within an image and using that information to un-distort the image. Fig 1 shows the chessboard image on the left and the inside corners within this image detected on the right.

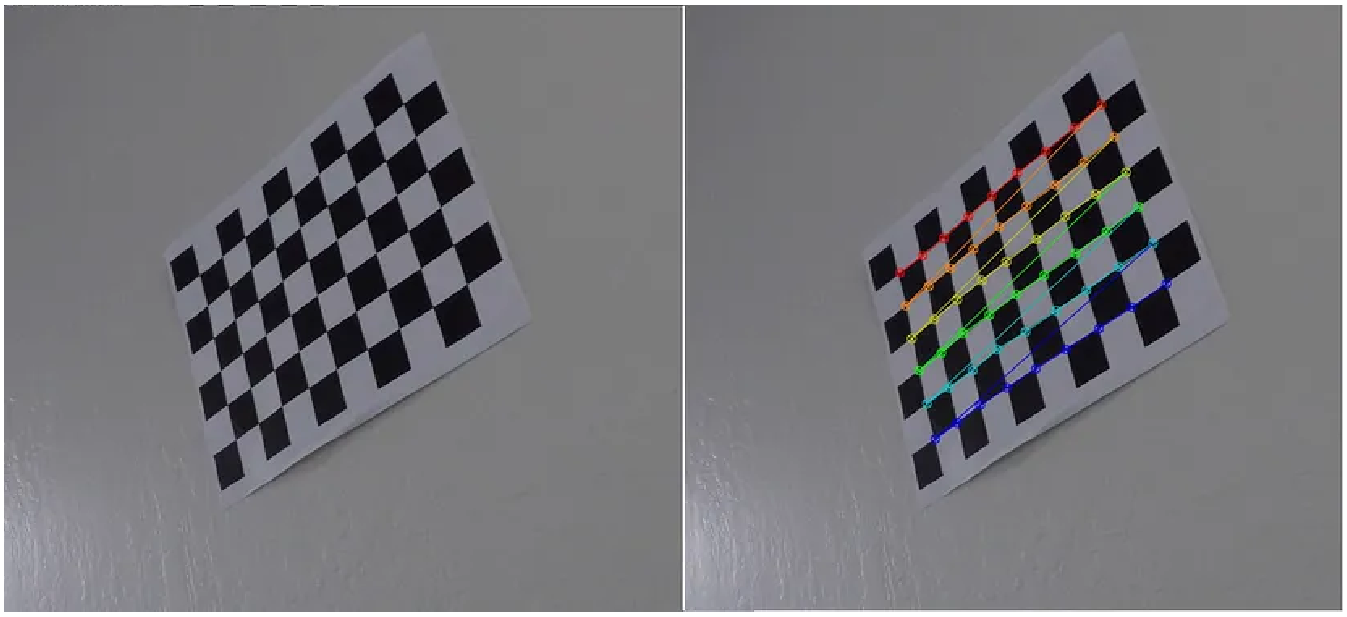


Fig 1. Calculating the camera matrix and distortion coefficients by detecting inside corners in a chessboard image (Source: Udacity)

The distortion matrix was used to un-distort a calibration image and provides a demonstration that the calibration is correct. An example shown here in Fig 2, shows the before/after results after applying calibration to un-distort the chessboard image.

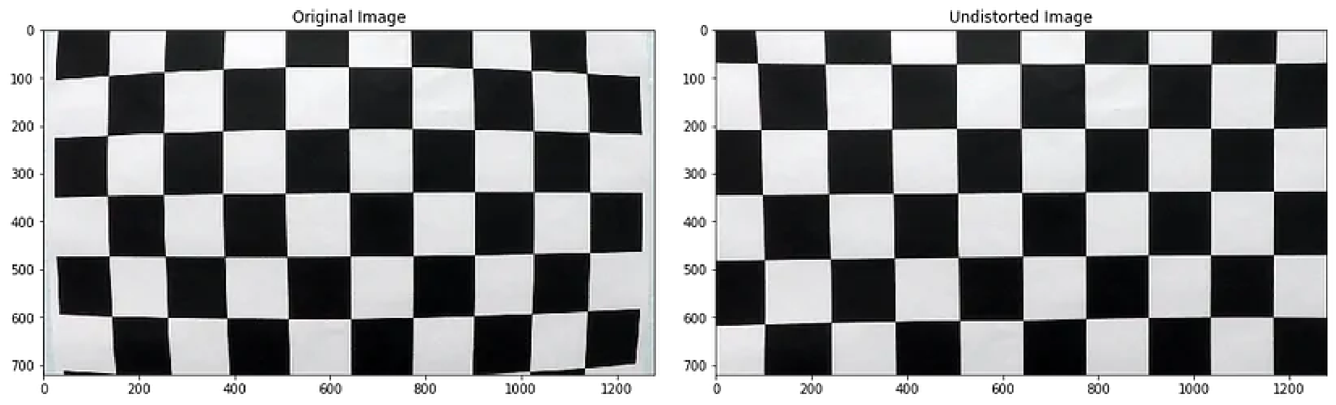


Fig 2. Before and after results of un-distorting a chessboard image (Source: Udacity)

2. Apply a distortion correction to raw images.

The calibration data for the camera that was collected in step 1 can be applied for raw images to apply distortion correction. An example image is shown here in Fig 3. It may be harder to see the effects of applying distortion correction on raw images compared to a chessboard image, but if you look closer at right of the image for comparison, this effect becomes more obvious when you look at the white car that has been slightly cropped along with the trees when the distortion correction was applied.

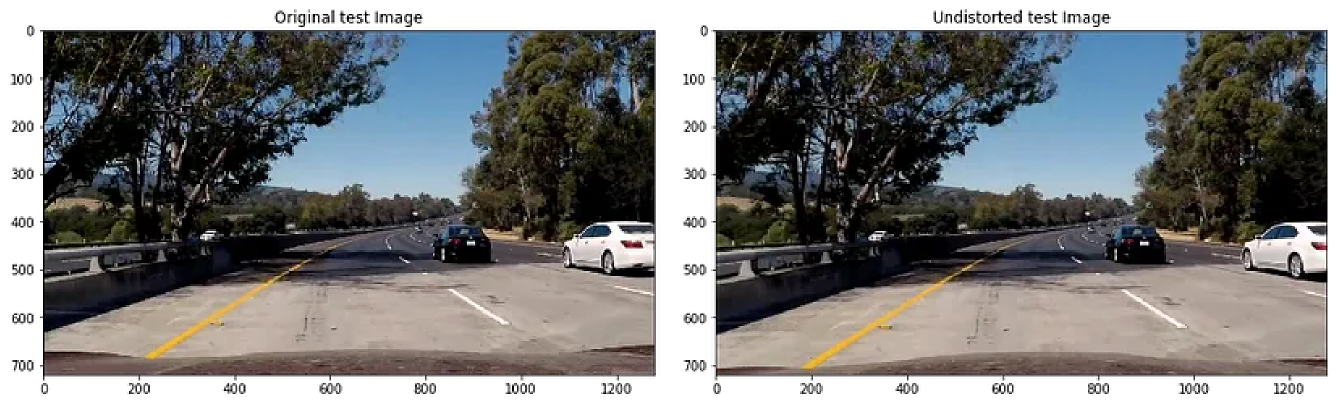


Fig 3. Before and after results of un-distorting an example image (Source: Udacity)

3. Use color transforms, gradients, etc., to create a thresholded binary image.

The idea behind this step is to create an image processing pipeline where the lane lines can be clearly identified by the algorithm. There are a number of different ways to get to the solution by playing around with different gradients, thresholds and color spaces. I experimented with a number of these techniques on several different images and used a combination of thresholds, color spaces, and gradients. I settled on the following combination to create my image processing pipeline: S channel thresholds in the HLS color space and V channel thresholds in the HSV color space, along with gradients to detect lane lines. An example of a final binary thresholded image is shown in Fig 4, where the lane lines are clearly visible.

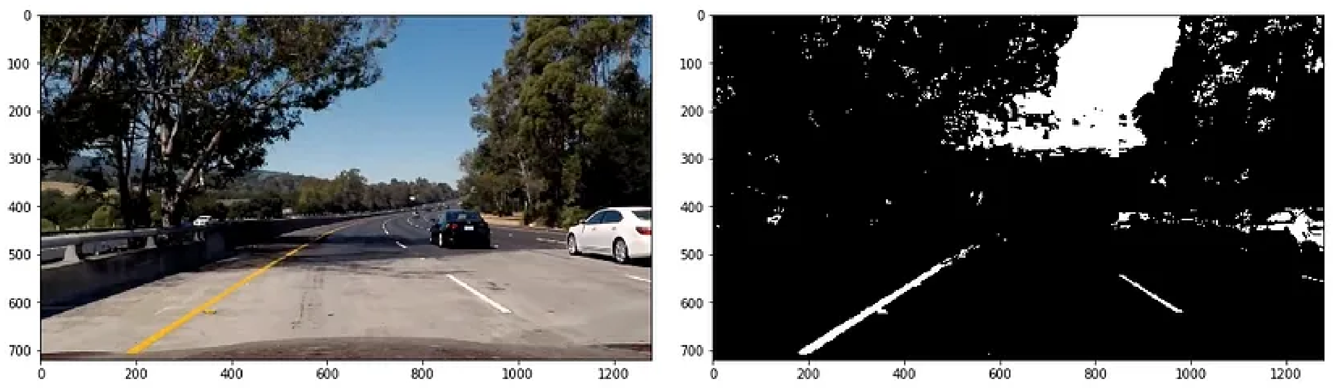


Fig 4. Before and after results of applying gradients and thresholds to generate a binary thresholded image (Source: Udacity)

4. Apply a perspective transform to generate a “bird’s-eye view” of the image.

Images have perspective which causes lanes lines in an image to appear like they are converging at a distance even though they are parallel to each other. It is easier to detect curvature of lane lines when this perspective is removed. This can be achieved by transforming the image to a 2D Bird’s eye view where the lane lines are always parallel to each other. Since we are only interested in the lane lines, I selected four points on the original un-distorted image and transformed the perspective to a Bird’s eye view as shown in Fig 5 below.

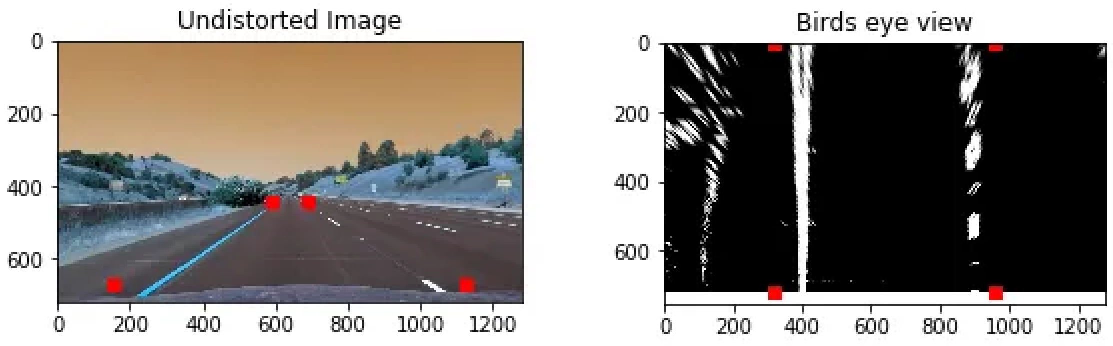


Fig 5. Region of interest perspective warped to generate a Bird’s-eye view (Source: Udacity)

5. Detect lane pixels and fit to find the lane boundary.

To detect the lane lines, there are a number of different approaches. I used convolution which is the sum of the product of two separate signals: the window template and the vertical slice of the pixel image. I used a sliding window method to apply the convolution, which will maximize the number of hot pixels in each window. The window template is slid across the image from left to right and any overlapping values are summed together, creating the convolved signal. The peak of the convolved signal is where the highest overlap of pixels are and it is the most likely position for the lane marker. Methods have been used to identify lane line pixels in the rectified binary image. The left and right lines have been identified and fit with a curved polynomial function. Example images with line pixels identified with the sliding window approach and a polynomial fit overlapped are shown in Fig 6.

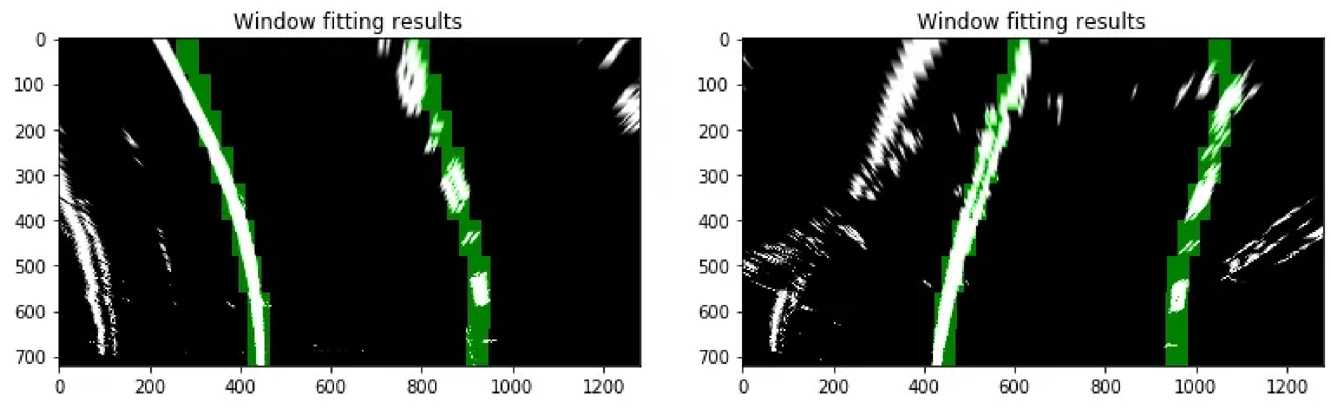


Fig 6. Sliding window fit results (Source: Udacity)

6. Determine the curvature of the lane and vehicle position with respect to the center of the car.

I took the measurements of where the lane lines are and estimated how much the road is curving, along with the vehicle position with respect to the center of the lane. I assumed that the camera is mounted at the center of the car.

7. Warp the detected lane boundaries back onto the original image and display numerical estimation of lane curvature and vehicle position.

The fit from the rectified image has been warped back onto the original image and plotted to identify the lane boundaries. Fig 7 demonstrates that the lane boundaries were correctly identified and warped back on to the original image. An example image with lanes, curvature, and position from center is shown in Fig 8.

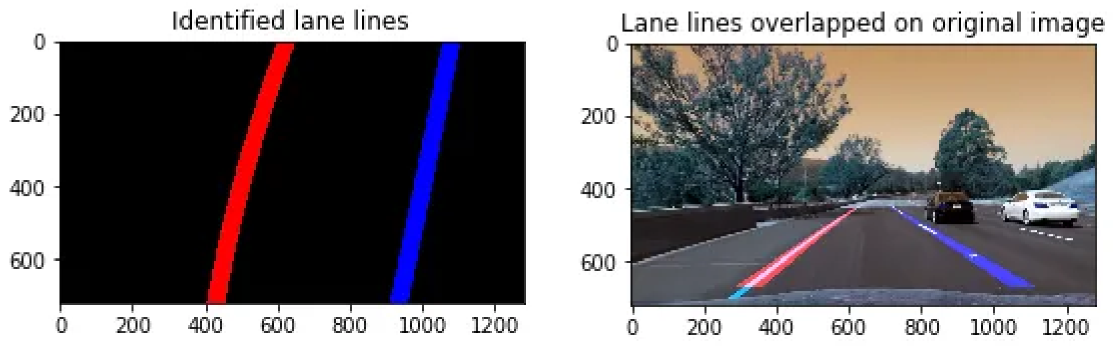


Fig 7. Lane line boundaries warped back onto original image (Source: Udacity)

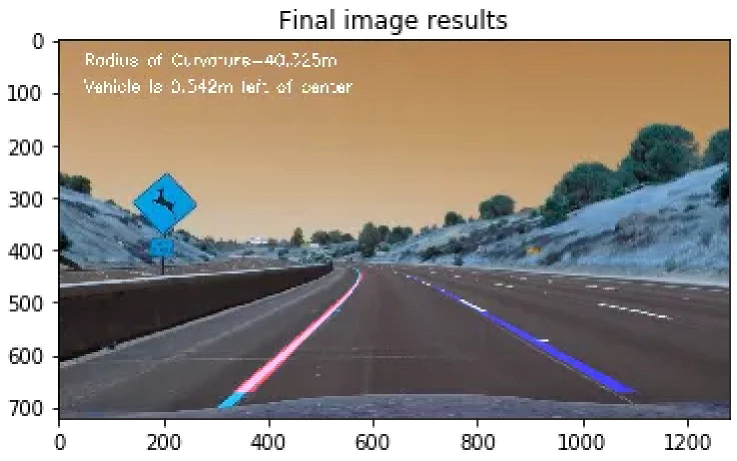


Fig 8. Detected lane lines overlapped on to the original image along with curvature radius and position of the car (Source: Udacity)

The above process was applied to each frame of a video and the end result with the key steps are shown in different frames here:

Summary:

This solution works well with normal lighting conditions. However, it needs to be improved to address different use cases. One example where it can be improved is for lanes where a portion of the lane is a freshly paved and is different in color with the other portion of the lane which is an older paved road. This algorithm also needs to be improved in cases where the camera has glare as a result of direct sunlight falling on it, and in other high contrast cases where the lane lines appear to be washed out making them harder to detect. These type of situations could be addressed by dynamically adjusting the contrast of the images to ensure the lane lines in the images are not washed out and to make sure the algorithm has a good dynamic range in all lighting conditions. In cases of roads which are curvy and have a slope, it would make it difficult to warp the images properly and it may cause problems for the algorithm. This issue can also be addressed by creating a dynamic region of interest for each image frame. These are some of the things that I need to explore at a later time, to build on the existing algorithm to make it robust for different use cases.

If you are interested in the technical details, you can review my Github repository.

References: All images & videos were sourced from Udacity, as part of their SD Car Nanodegree program. The augmented information on the original images were generated by me which is the output of the code that I developed.