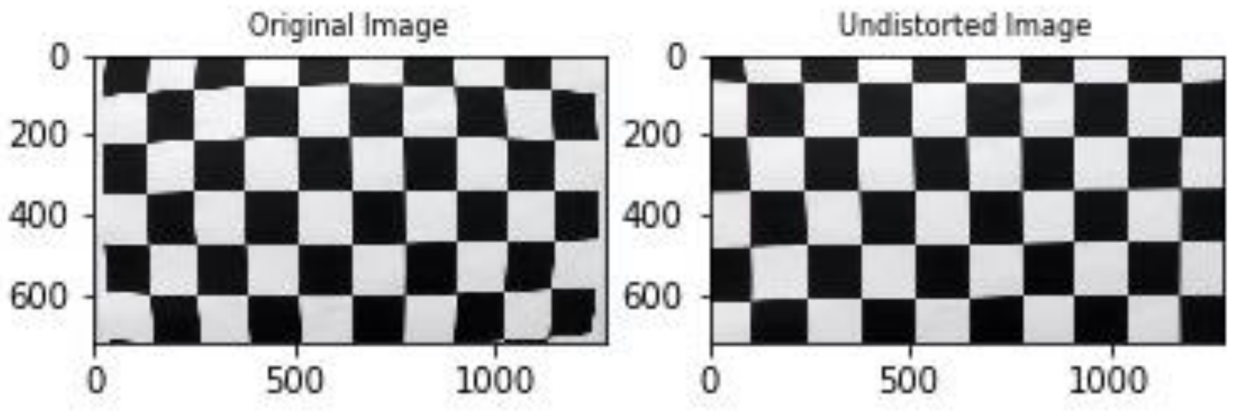
Advanced lane Finding Project

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.

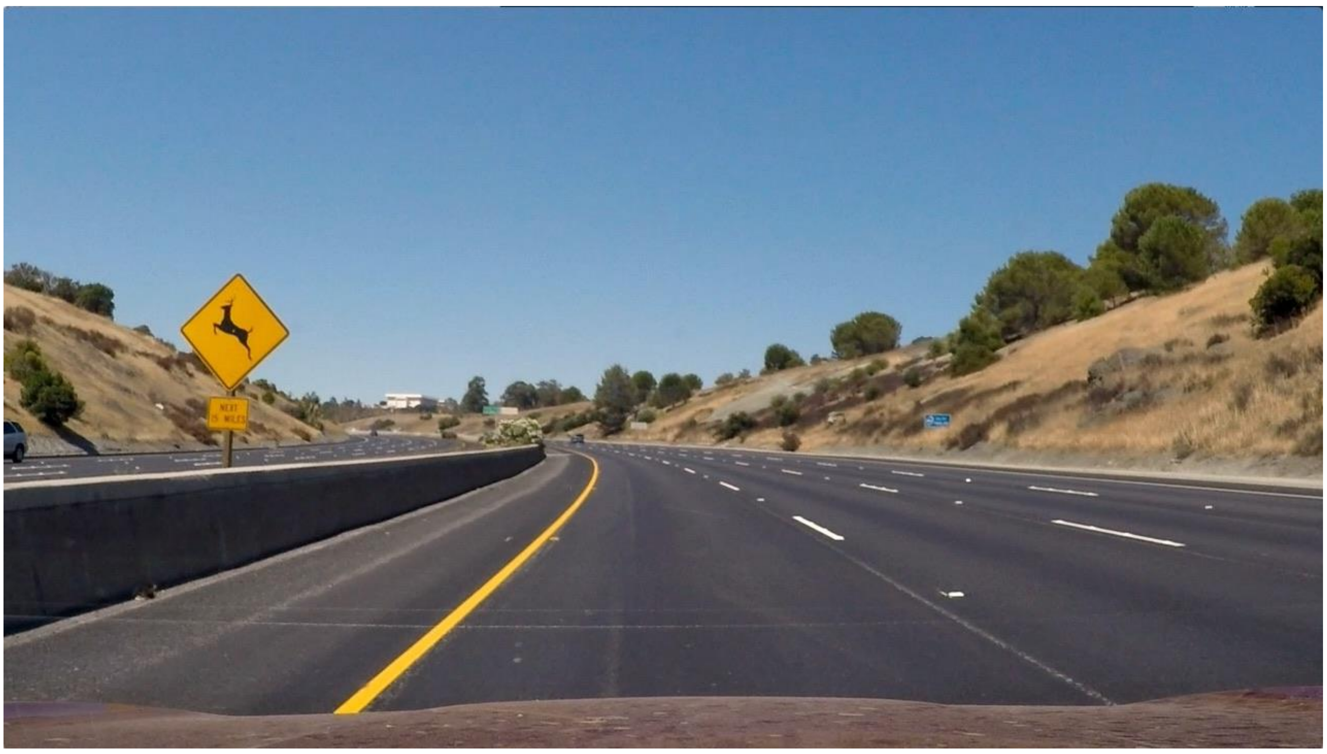
Camera Calibration

- The code for this step is contained in lines 24 through 55 of the file called `final_submission.py`
- 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 both the calibration and test images using the `cv2.undistort()` function and obtained this result:

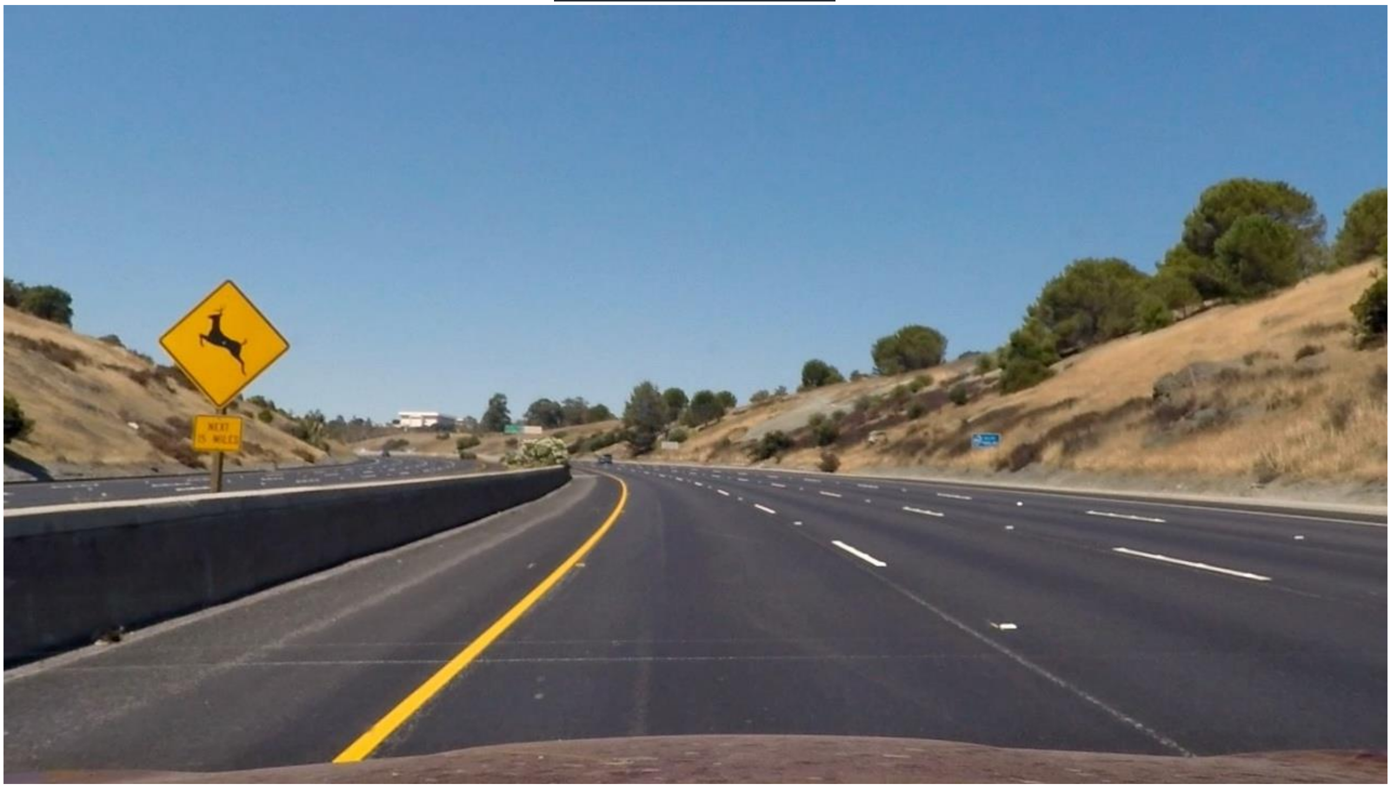


The test Images are read in a sequence and the function cv2.undistort is used for distortion correction. All the images are saved in the output_images folder. A sample of distortion correction on test image is as below:-

Original Image



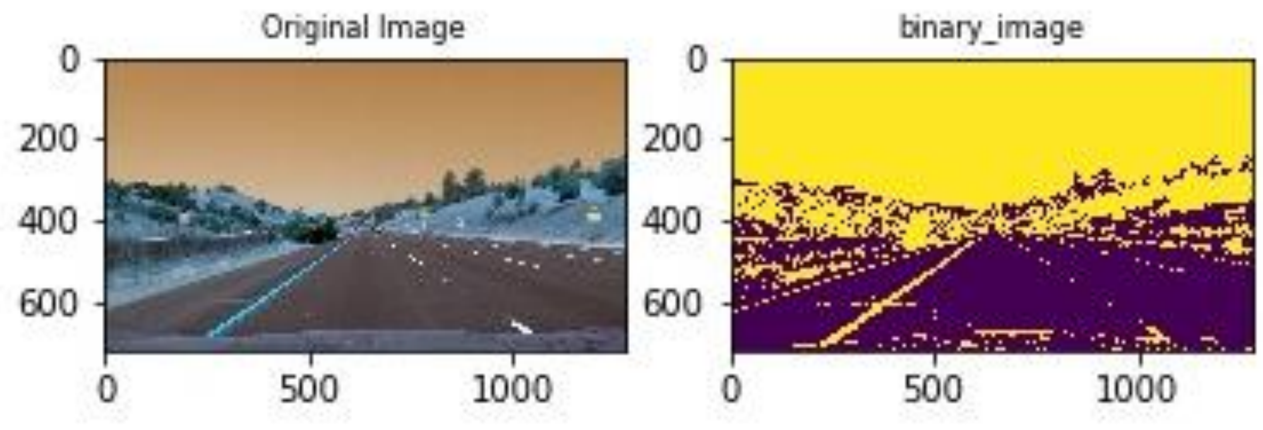
Undistorted Image

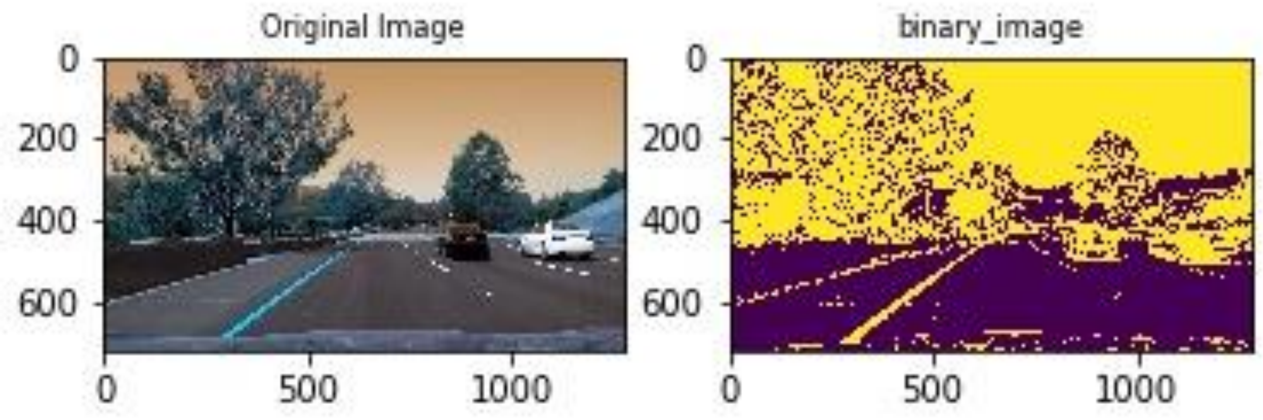


Color and Gradient Thresholding

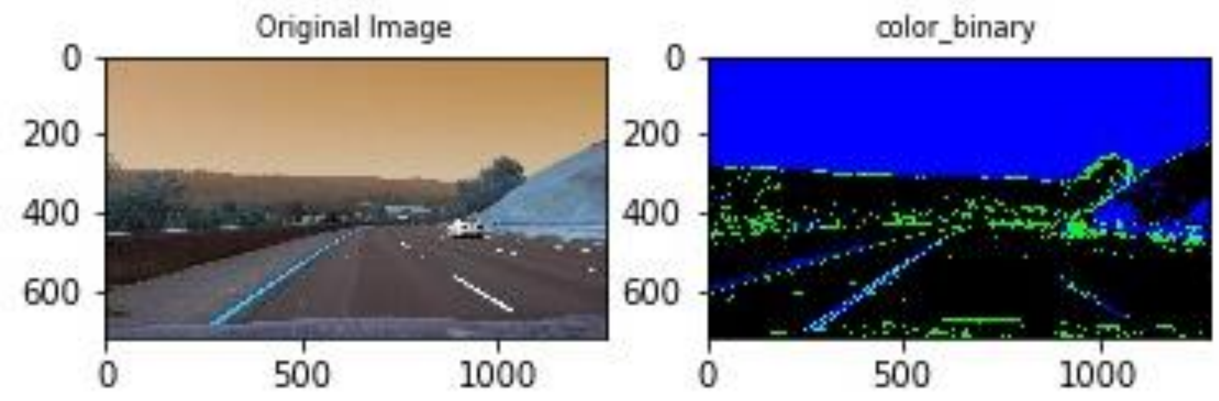
I created a function for both color and gradient thresholding. The code for the same is contained in the lines between 100 and 119. I used S_channel for binary thresholding as it was giving better results. For gradient I applied sobel for both x and y direction.

Took square and added both the sobels. I took square root and scaled the sobel for standardization. Finally, I created a binary Image with a combination of color and gradient thresholding. This function was applied on all test images and the output was saved to the folder. An image on which this function is applied is as follows: -





The effect of color and gradient thresholding on a test image is as follows: -



Perspective Transform

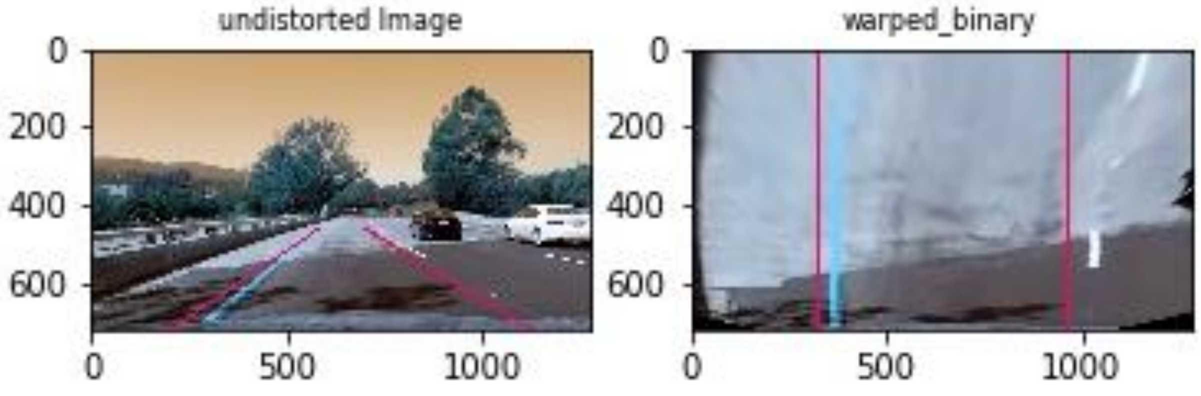
The code for selecting source points, destination points, applying perspective transform and getting a warped image is from lines 178 to 191 in the final_submission.py file. The source and destination points I selected are as follows: -

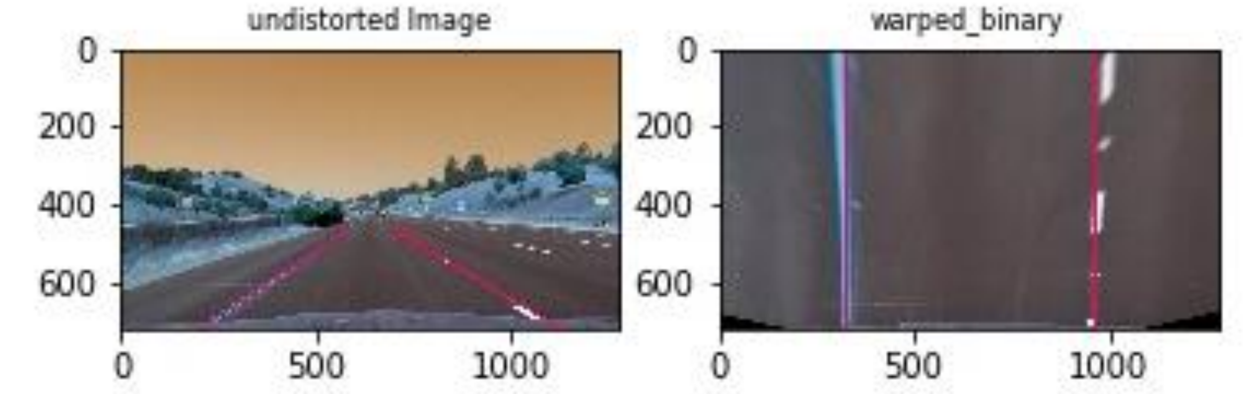
```
src = np.float32(
    [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
    [((img_size[0] / 6) - 10), img_size[1]],
    [(img_size[0] * 5 / 6) + 60, img_size[1]],
    [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
dst = np.float32(
    [[(img_size[0] / 4), 0],
    [(img_size[0] / 4), img_size[1]],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

This will be translated into the following numeric source and destination points: -

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

Using this source and destination points the perspective transform, M was calculated using the function cv2.getPerspectiveTransform ( ). Using M, an image is transformed using the function cv2.WarpPerspective ( ). This was applied on test images and saved to the folder. The application of perspective transform on a test image is as follows: -

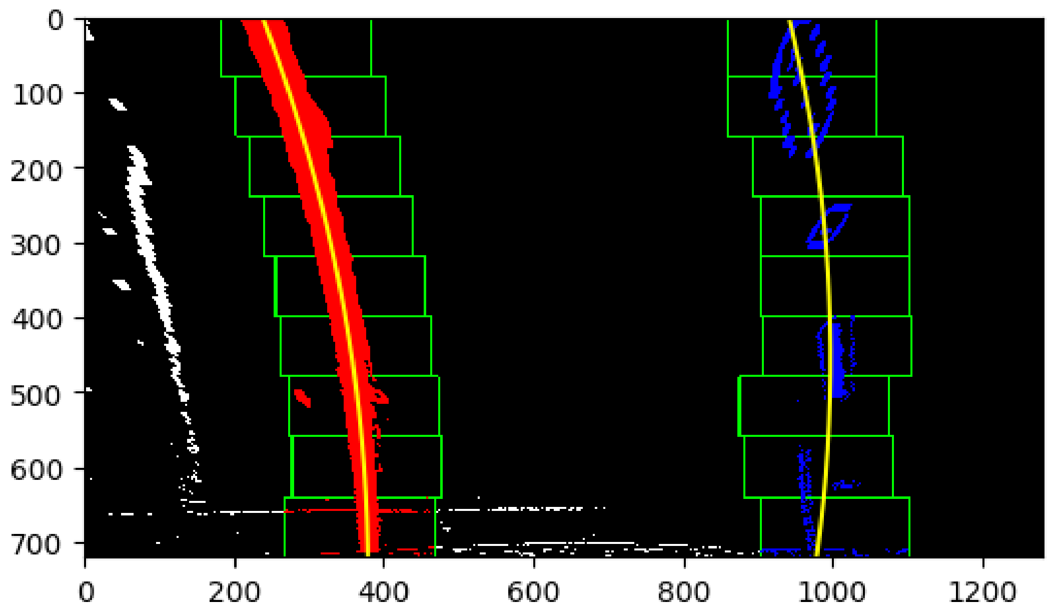




Lane line Identification and polynomial fitment

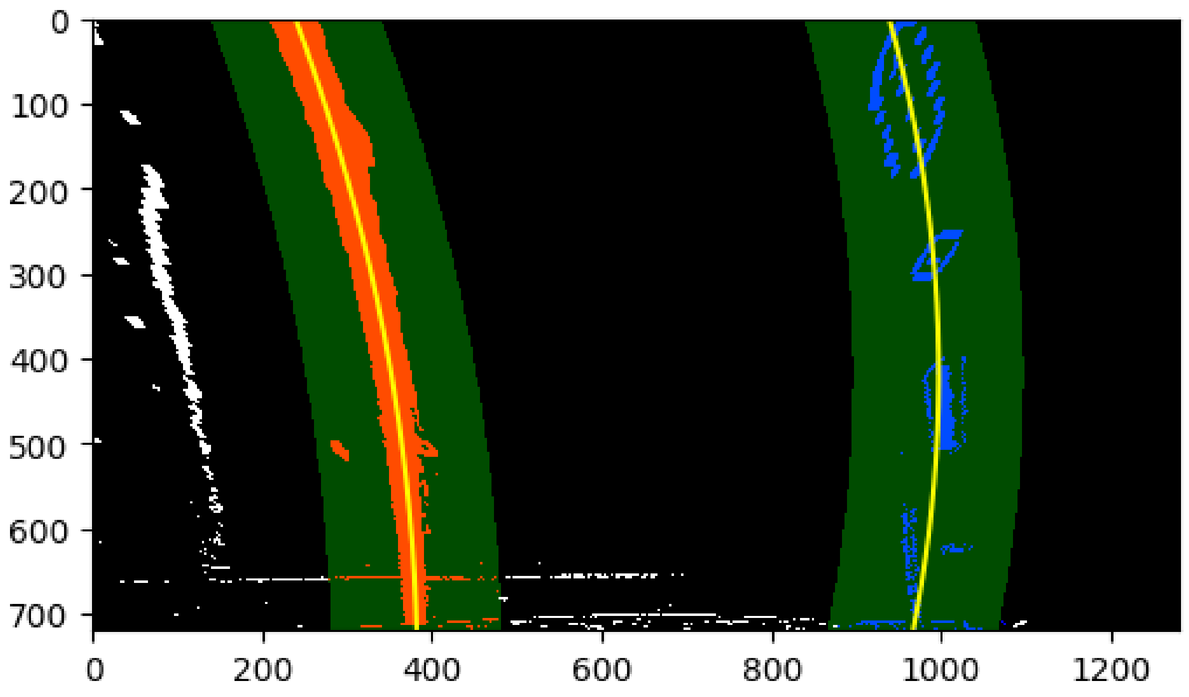
A function was written to identify and fit the polynomial to the lane lines. The function is from lines 246 to 369 in the final_submission.py file. This function will take as input the thresholded binary image with perspective warping. Sliding window method is used for finding the lane lines. Second image onwards the search area for finding the lane lines is narrowed down to a margin surrounding the polynomial fit of the first image.

A histogram is plotted for the bottom half of the image. The two highest peaks of the histogram will help identify the left and right lane line positions on the x-axis. A sliding window is created with a margin which will be utilized for finding the lane lines within itself. The position of the window will be adjusted by shifting left or right according to the change in lane line position. A polynomial of second order was fit to the good nonzero values found inside the sliding windows on either side of the image. This method will be utilized only for the first image in a series. (The code for this is included in a lane_plot.py file. As I dint want to include plotting within the function itself, a separate file was created to generate this plot for inclusion in the writeup).



Second image onwards the search for lane lines will be narrowed down to a margin around the polynomial fit of the first Image. A polynomial of second order was fit to the good nonzero values found within a margin of the polynomial fit of the first image. This method is utilized for finding the lane lines from the second image onwards. Finally, the lane lines found on the perspective warped image is mapped back to the original image using inverse of the perspective transform M.

An image depicting the search area (green region in the below image) which is narrowed down to a margin around the polynomial fit of the previous image is as follows (The code for this is included in a lane_plot.py file. As I dint want to include plotting within the function itself, a separate file was created to generate plots for inclusion in the writeup): -

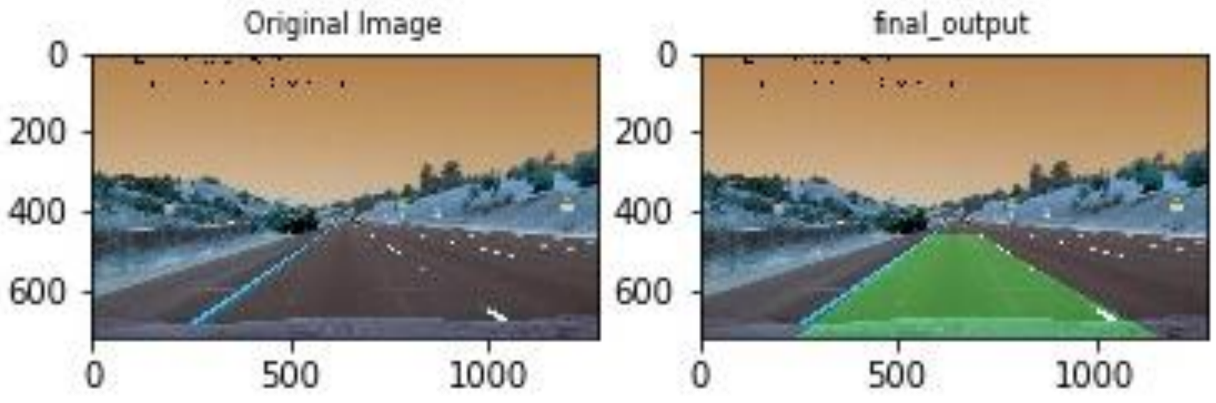


Radius of Curvature and Offset

The radius of curvature and offset was calculated within the drawlanelines function itself from lines 344 to 367 of the final_submission.py file. The pixel values are converted to meters and overlay on top of the images. For calculating the radius of curvature, the lane is considered as 30 meters long and the 3.7 meters wide. For calculating the offset the distance from center of the lane to the image center is calculated and converted into meters and overlay on the image using Cv2.putText function.

Final Result

A function (final_pipeline) was created which will perform undistortion, make a thresholded binary image, warp perspective, find the lane lines, fit the polynomial, calculate the radius of curvature including offset and map the findings back to the original image. The code for this is included from lines 374 to 379 in final_submission.py file. All the test images are read sequentially and the final_pipeline function is applied on them. All images resulted from the function are saved to the output_images folder. A sample image is as follows: -



Video
The project video was processed using the final_pipeline. The code for the same is included from lines 396 to 397 of the final_submission.py file. The video file named submission is included in the zip file submitted.

Challenges

- This code narrows down the search for lane lines to the polynomial fit of the previous image. If the lane lines were not found than it should go back to the sliding window search. This will improve the performance in unforeseen conditions also. This can be implemented in future versions.

- Hit and trail was carried out for gradient and color thresholding. There could be a better combination which can be further explored to create better pipeline.

- Presently, this code is doing well with a set lane length and wide. If these parameters change the pipeline may not perform well for example, Indian roads.

- If the environment conditions change (lighting, weather) or the surroundings in which car is being driven change (bridge) or the presence of other nearby vehicles may confuse the model. These are some of the challenges which can be tackled in future versions.