# Project 2: Advanced Lane Lines

The goal of this project is to write a software pipeline, that will identify positions of the left and right lane lines on the road, in the images and video stream.

This is achieved in following steps:

1. Camera Calibration – Calculate calibration matrix and distortion coefficient for the installed camera from a set of chessboard images
2. Undistortion – Apply this distortion correction to raw road images
3. Color and gradient thresholds – To obtain a binary image containing maximum pixels belonging to lane lines
4. Perspective Transform – obtain birds eye view of each road image
5. Fit polynomials to lane lines
6. Calculate radius of curvature and offset from center for each lane line

### Camera Calibration

A camera mounted on the car (used to capture test images and videos) converts 3D image of the road into 2D. This conversion causes the generated images to become distorted – bent at the edges and lines may appear incorrectly skewed.

The distortion can be either radial or tangential and it needs to be corrected for an accurate lane lines detection in self driving cars

Radial distortion correction –

Tangential distortion correction –

Camera calibration for distortion correction can be performed using multiple images of a chessboard taken from different angles and applying functions from OpenCV library in python.

As seen in the attached snippet of the code (figure 3), I followed following steps for camera calibration –

1. Create empty arrays of object points (3D coordinates of undistorted chessboard image) and image points (2D coordinates of distorted chessboard image)
2. Object and image point here correspond to 9X6 corners visible in the chessboard images
3. A for loop is created to load the images of the chessboard one by one and convert them to grayscale
4. Cv2.findChessboardCorners() function is used to find chessboard corners in the grayscaled image
5. Once the 9X6 corner points are detected at the end of one for loop, their coordinates are added to the object and image points array.
6. Copy of the original image – img is used to overlay these corner points using cv2.drawChessboardCorners() function

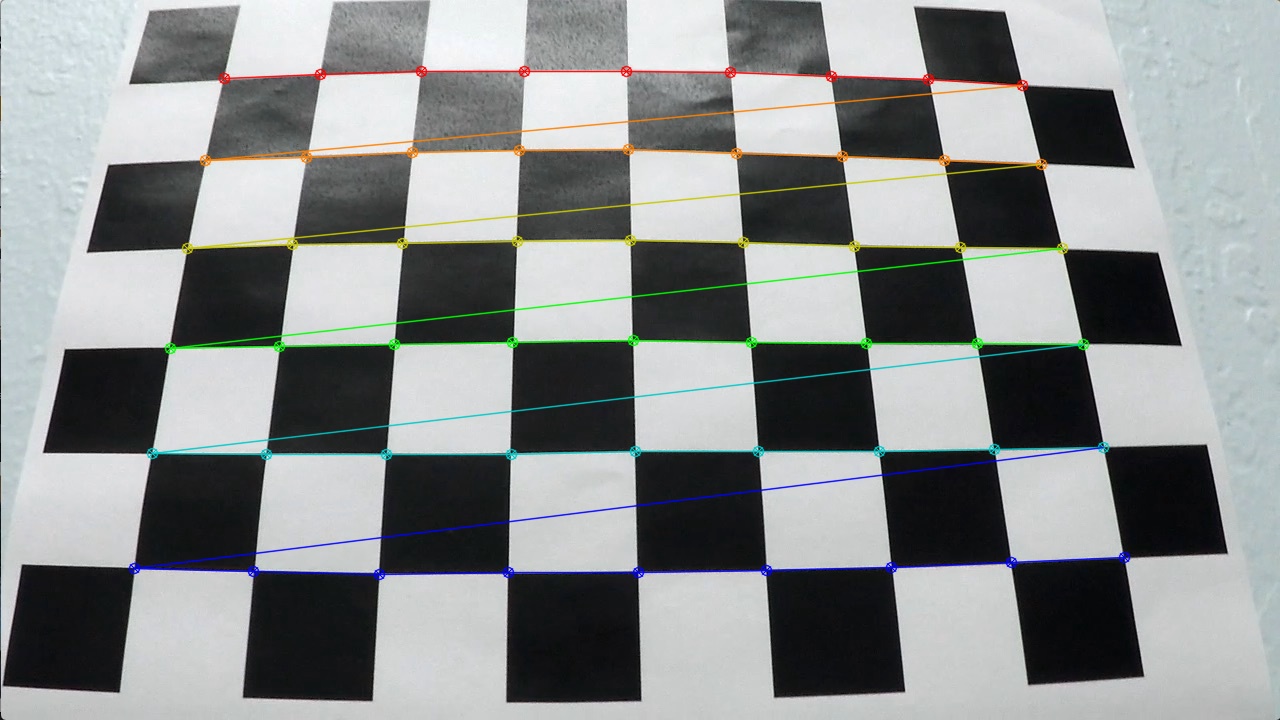


Figure :draw chessboard corners

1. Cv2.CalibrateCamera() function is used to compute the calibration coefficients. This function returns camera matrix (mtx), distortion coefficients (dist) along with rotation and translation vectors. Mtx and dist from this function are used later in the software pipeline to undistort test images using cv2.undistort()

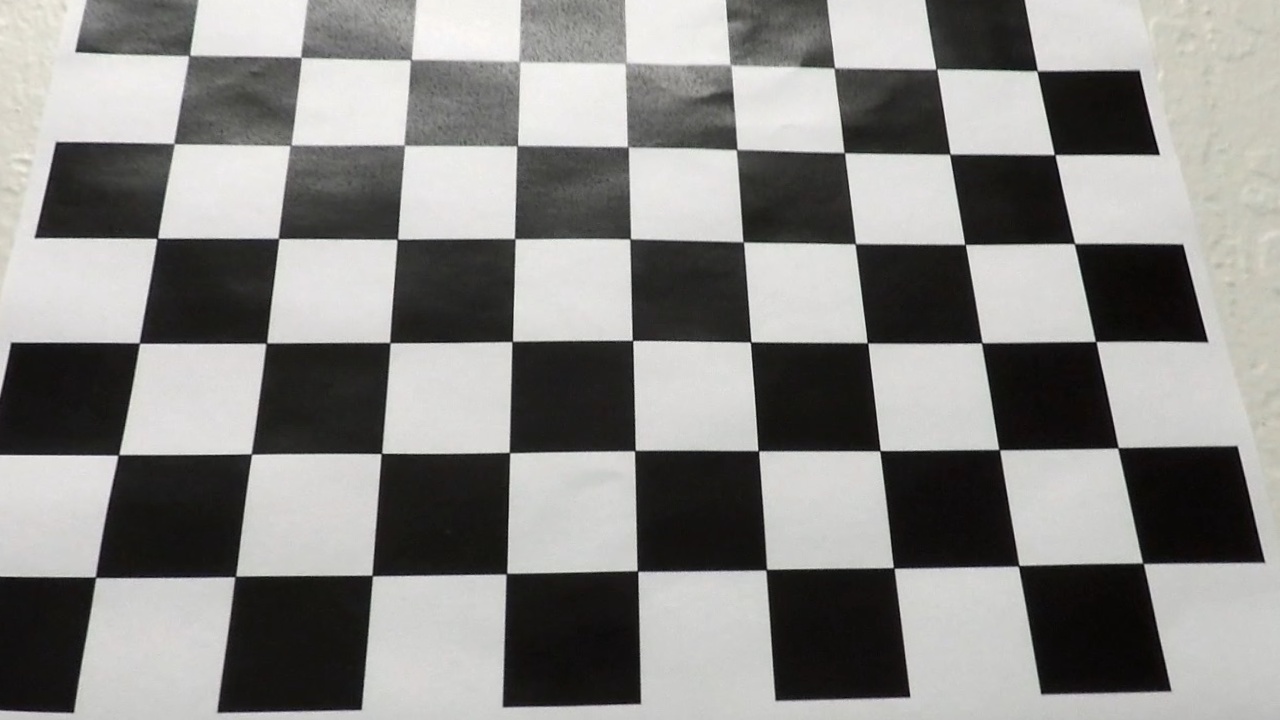
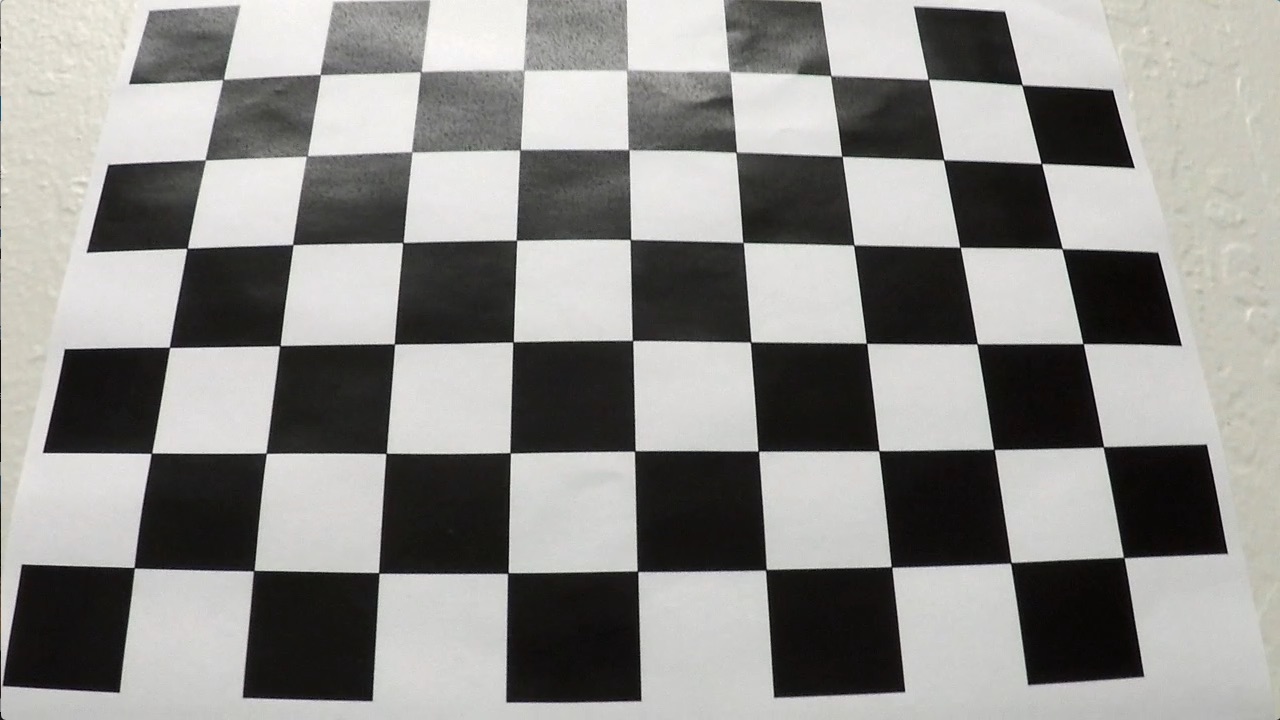


Figure : Image undistortion

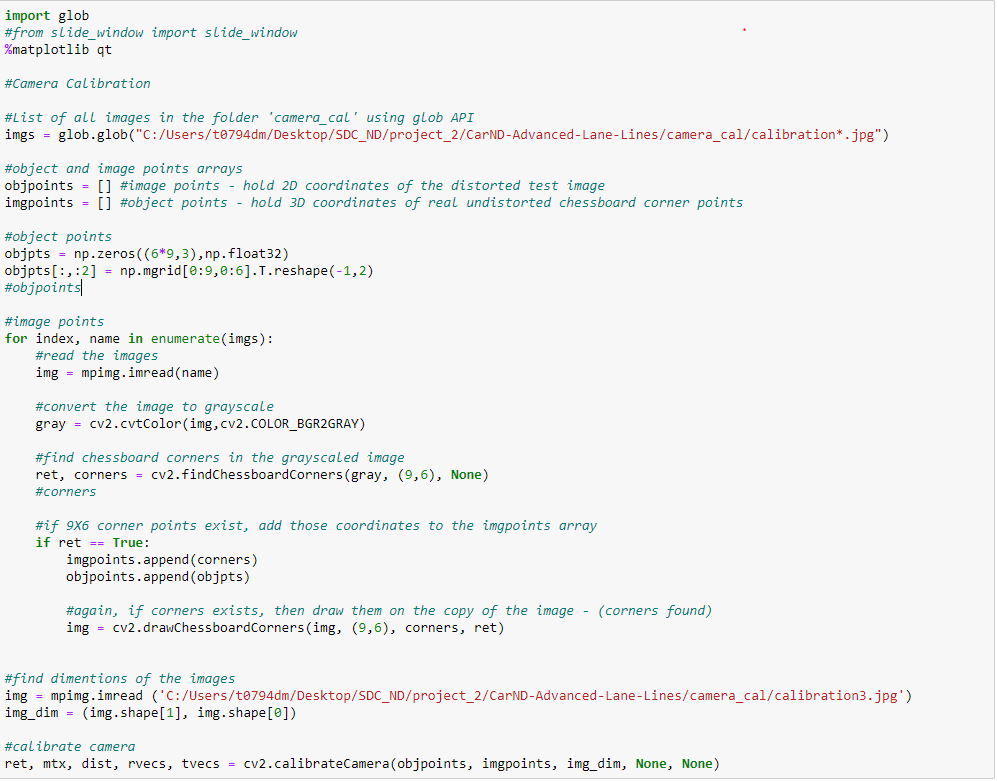


Figure : Code for Camera Calibration

### Software Pipeline (for images)

1. **Distortion Correction** –

Distortion correction coefficients and camera matrix calculated via cv2.calibrateCamera() function are applied to each image in the test image folder to undistort them, using cv2.undistort()





Figure :image undistortion

1. **Gradient Threshold (Sobel Operator) –**

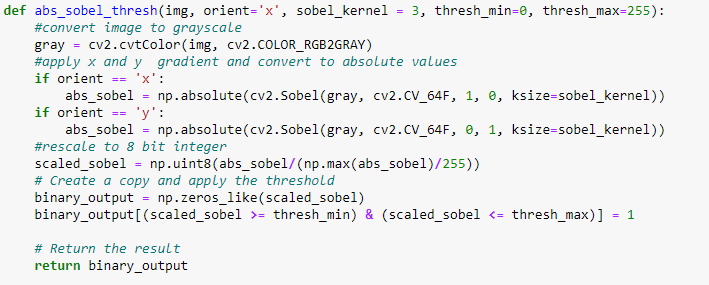
After images are undistorted, we convert it into a binary image in order to perform further filtering to highlight lane lines in the image

First step towards achieving that is applying Sobel operator. I defined a function abs\_sobel\_thresh() to apply sobel operator to each test image and take derivative in the x and y direction.

Keeping the kernel size at 3, threshold windows are chosen such that maximum pixels from both lane lines are highlighted with minimum noise from other objects

X gradient threshold – [15,255]

X gradient threshold – [25,255]



1. **HLS and HSV Threshold –**

In addition to gradient threshold, I have applied saturation threshold (HLS) to capture lower saturation lane lines and value saturation to reduce the effect of right light and shadows from the image

S threshold – [110,255]

V threshold – [60,255]

In the final software pipeline, gradient and SV threshold is applied with OR condition so that lane line detection failure with gradient threshold because of lighting is captured by SV thresholds

Example of final binary image after color and gradient threshold is shown below

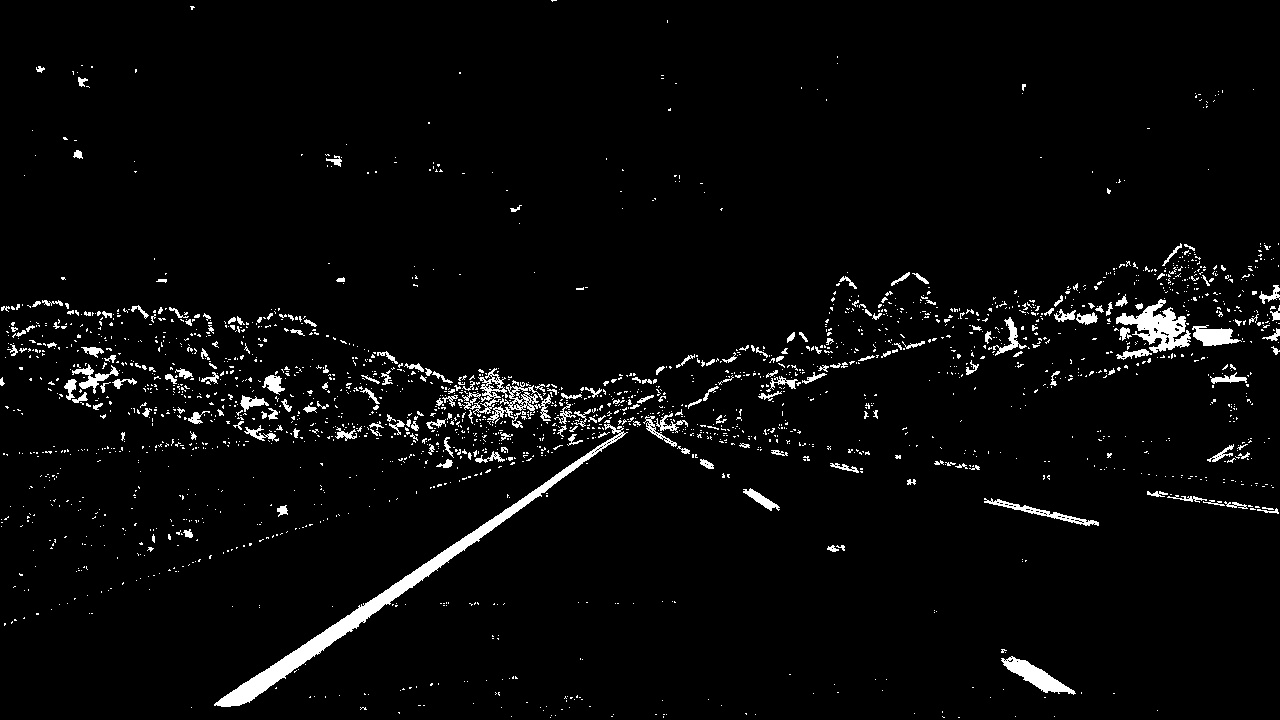


Figure – Binary image after thresholds

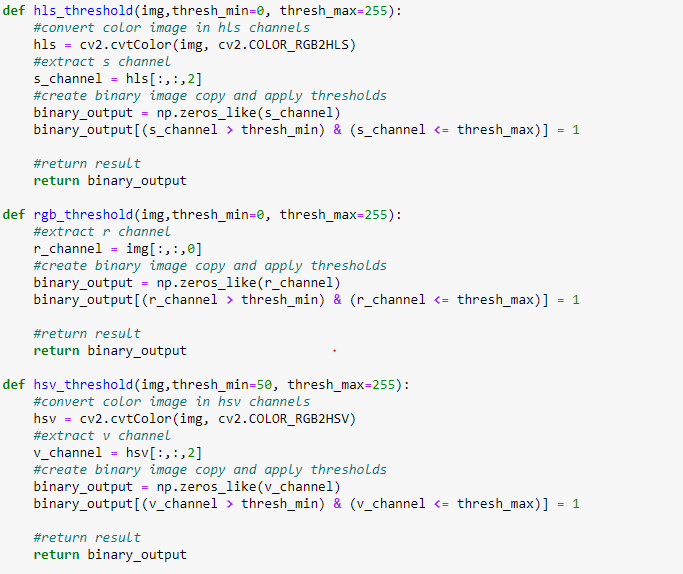


Figure : code for hls, hsv, rgb threshold function

1. **Perspective Transform** –

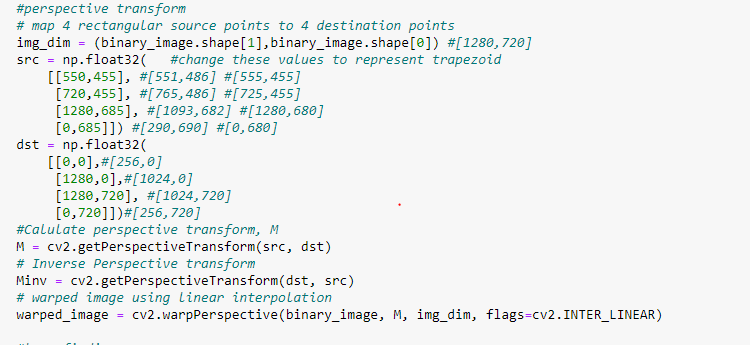
Once the binary image is created that highlights maximum pixels from lane lines, the next step is to perform perspective transform to generate a birds’ eye view of the road image. This is done because ultimately we want to measure curvature of the lines. To do this we first define source and destination points. Source points are coordinates of 4 points on the test image that are approximately in the same plane and adequately cover both lane lines. I have selected source points such that perspective transform only captures lane lines while avoiding any noise from surrounding objects. I used an image Interactive window to detect x and y coordinates of source points. These source points are mapped to destination points that represent a rectangle of the same size as test image

Cv2.getPerspectiveTransform() function is used to generate a transform, M and cv2.warpedPerspective() function is used to create a warped image with bird’s eye view of the lanes.

Figure 7 shows an example of the road image after perspective transform. One of my criteria to define source points was that both lane lines should appear approximately parallel to each other after perspective transform



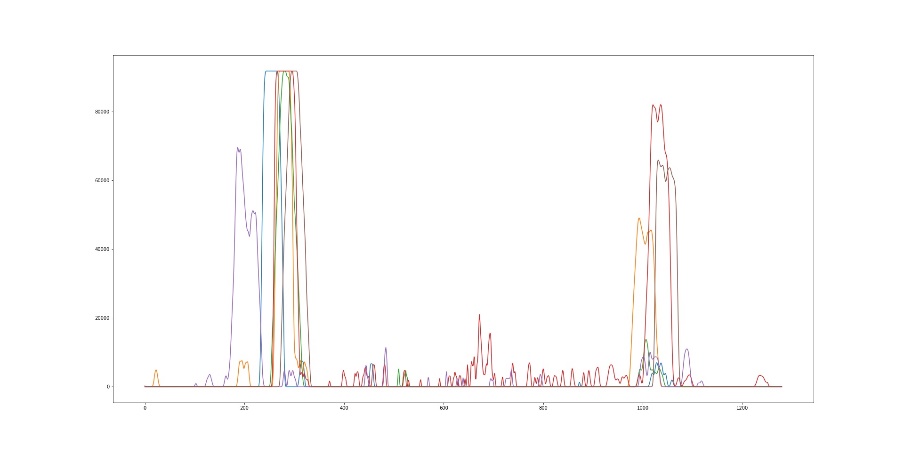
Figure : warped image with bird's eye view of the road



1. Finding Lane Lines (polynomial fit)

Once the binary warped image of the lane lines is generated, the next step is to locate pixels belonging to left and right lane lines. For this I followed histogram technique as described in the course.

* First, I took a histogram of the bottom half of the image (bottom half because we want to locate left and right lane pixels for the first frame starting at the bottom). Following plot shows histogram for all test images.



* Start point of the left and right lane is easily noticed in the above image and can be calculated from x and y coordinates of the image
* Using two highest peaks of the histogram as starting point, I used sliding window technique to determine location of frames moving further along the road
* For this project I selected 9 frames in one image, with a margin of 100 and minimum pixel concentration is kept at 50.
* Looping through each window, I found boundaries of current window and number of activated pixels in it. If the number of pixels in the subsequent window is more than 50, I reset the window location to the new mean position. This is performed in the function find\_lane\_pixels() in my code. A n example of test image 1 shown below.

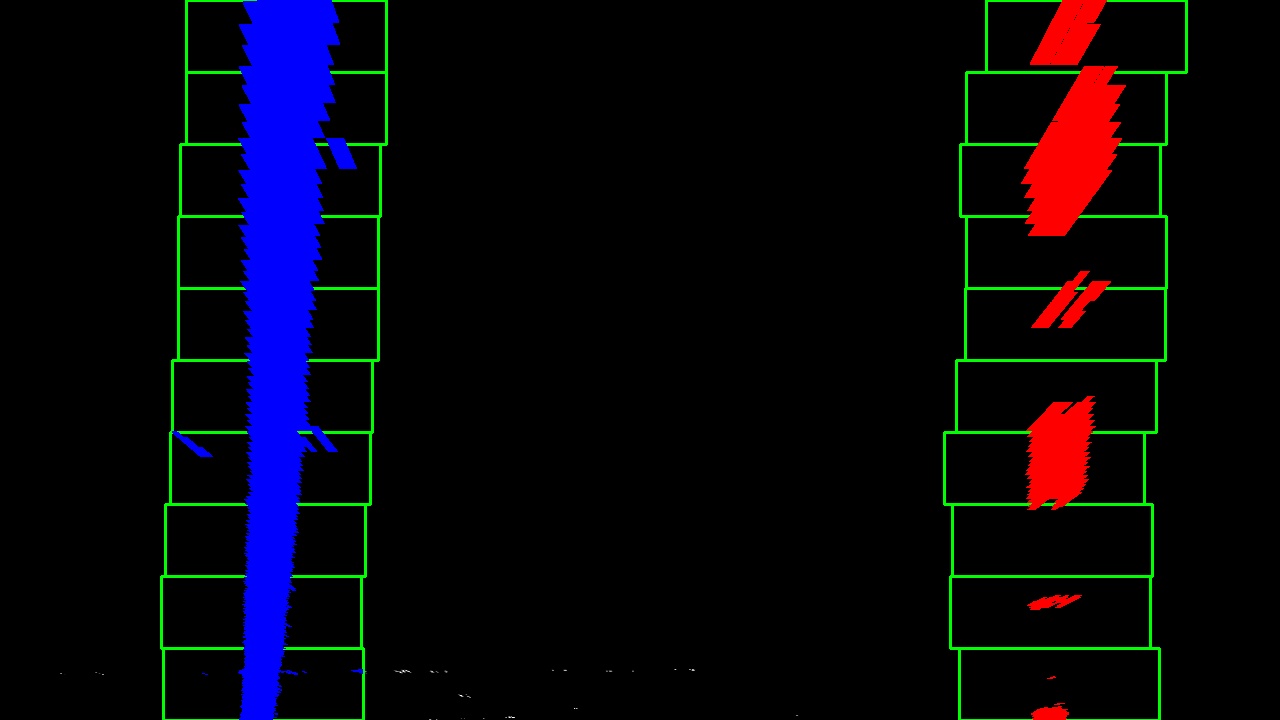


Figure 8: lane lines tracked

Once lane lines are tracked in the binary warped image, next step is to fit a polynomial through it to define left and right lanes.

* Using left and right pixel coordinates, I used np.polyfit to generate a polynomial for left and right lane
* To plot these curves on the top of the test image, I created a different array of x and y coordinates of lanes, using polynomial equation and an array of y coordinates in ploty (starting from the bottom of the image in an interval of 1)
* To visualized this on the binary image, left lane pixels are shown in blue and right lane in red. The polynomial fit is shown in green on the top of the lane lines

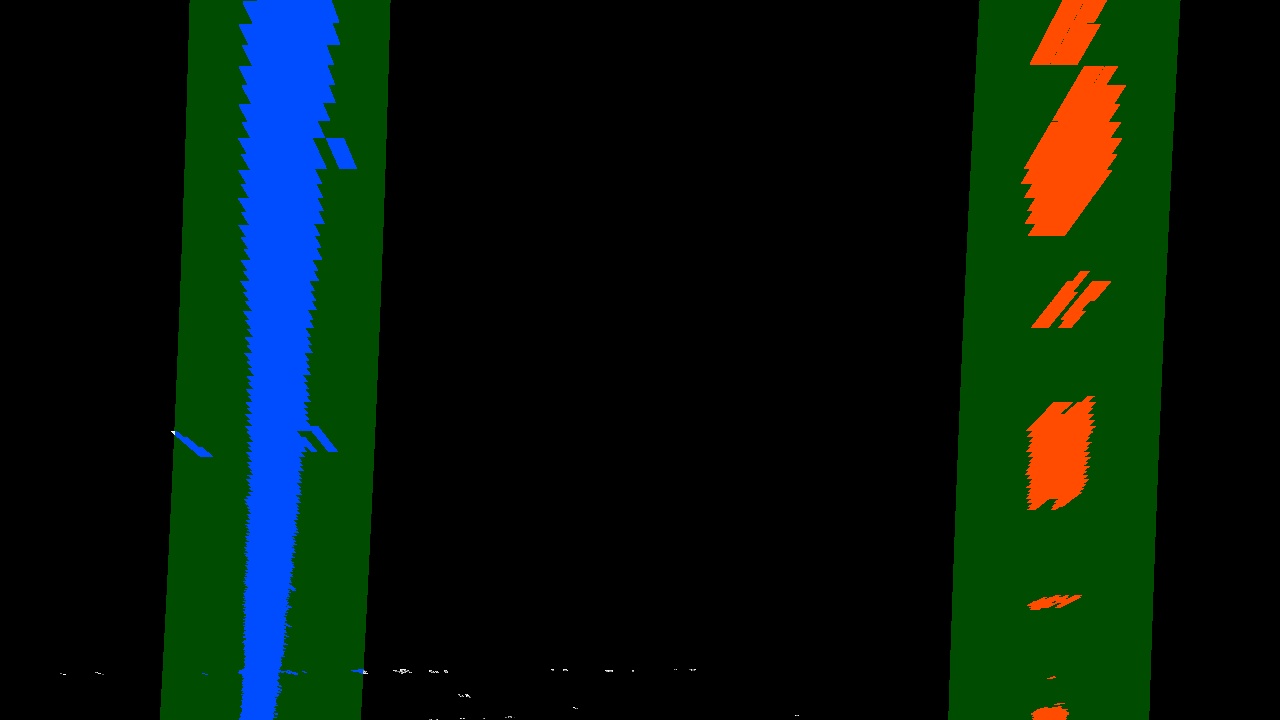


Figure 9: Lane lines visualization

For running this pipeline on videos, we don’t need to do this process from the beginning for every frame, we can just search in the margin of previous lane lines position. For this I grabbed x and y pixel position from earlier frame along with the polynomial fit, and calculated location of pixels that fall into green shaded area shown above.

This way all the relevant lane pixels are found and we fit a second order polynomial, again for both lanes, repeating the step above, I generated left and right lanes [x,y] coordinates to plot on the image.

As a final step, lane lines are plotted on the original image using cv2.addWeighted().



Figure 10: lane lines on the original image



Figure 11: function to find lane pixels in a binary image

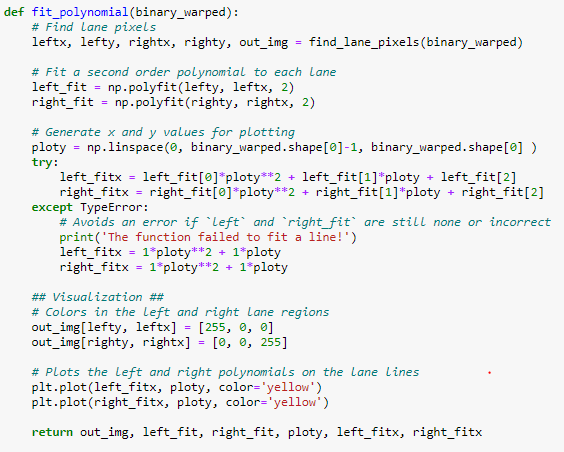


Figure 12: Function to fit polynomial both lanes

1. Radius of Curvature and Offset Calculation

* For additional information, I calculated radius of curvature for both left and right lane lines and offset from center (assuming camera I mounted at the center of the image/car)
* Lane is assumed to be 30 m long and 3.7 m wide. This information is used ot convert pixels into meters. using np. Polyfit(), I fit a polynomial for lanes (this time in units of meters)
* Offset is distance by which center ofteh lane lines is off from center of the car/image. Center of the lane line is average of x coordinate of left and right lanes and center of the car is image width/2. Difference, offset is printed on the final output image, as shown below



Figure 13: final output image

### Issues and possible improvements to the pipeline

Possible issues and improvements that can be implemented to my current pipeline

* Four coordinates of the rectangle selected for perspective transform works well with project video. I have eyeballed them for project video, but it captures a lot of noise other than lane line pixels in case of challenge videos. More concise region will increase chances of capturing only lane lines
* Additional thresholds need to be added to neutralize effect of bright light, tree shadow, cracks in the road and color change of the road.
* Crack in the road and dark marks beside right lane in the challenge video can be eliminated by taking only x gradient
* Kernel size can be increased to avoid spurring gradients from tree shadow road shade
* Perspective transform does not work well for challenge video where car is going down a curvy road. This effect can be minimized by reducing height of image during perspective transform, so as to only capture plane closer to the car