Advanced Lane Finding Project

How To Run

Run on Images

By default, advanceLanes.py will run on the test images.

Run on Video

Edit line 14 to read “test_video1 = True” and run advanceLanes.py

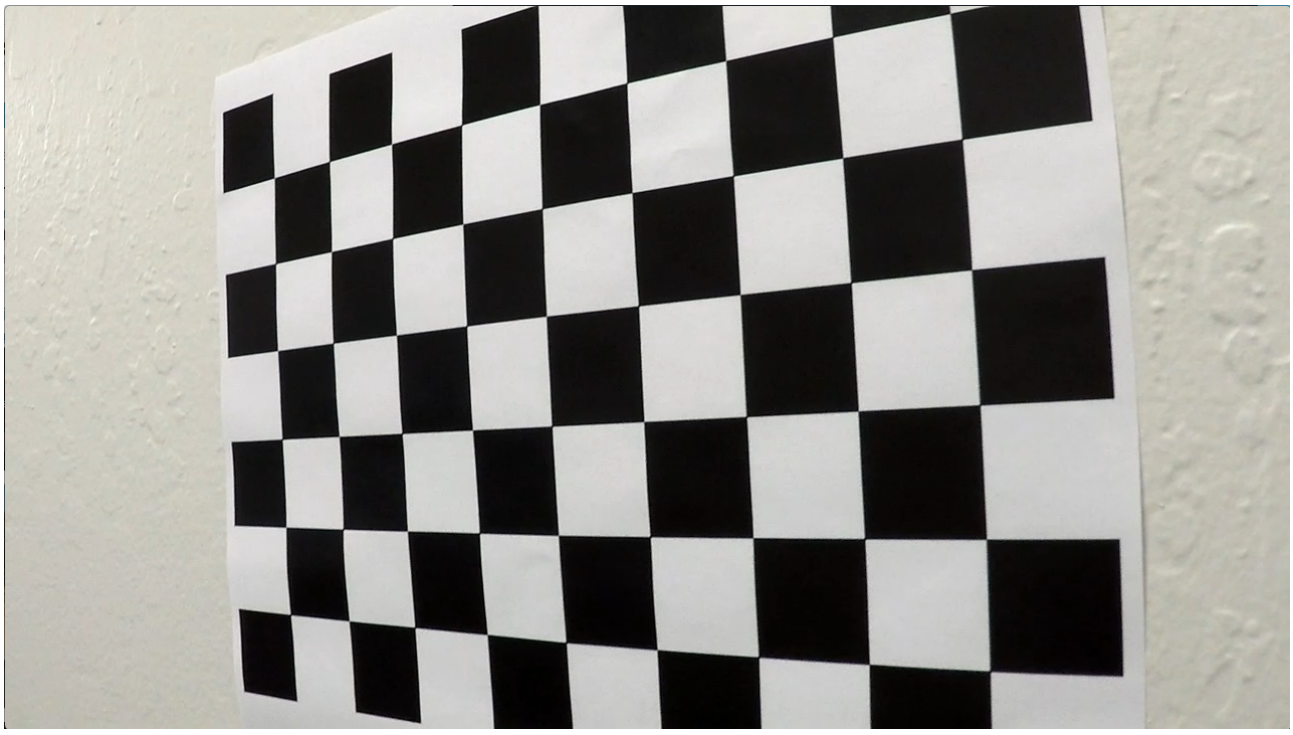
Steps Taken to Achieve Result & Techniques Used

Overall, my pipeline looks like the following:

Let’s look at each stage in some detail

Camera Calibration

In this stage, I read in a series of test images from the camera_cal directory. Each image is a view of a chessboard pattern at various angles, as shown below.



I use the OpenCV function findChessBoardCorners() to find the x,y coordinates of each corner on a given image. Once I have an array with all these corners, I use the OpenCV calibrateCamera() function to calculate the camera matrix, the vector of distortion coefficients, and the camera vectors of rotation and translation.

Undistort Image

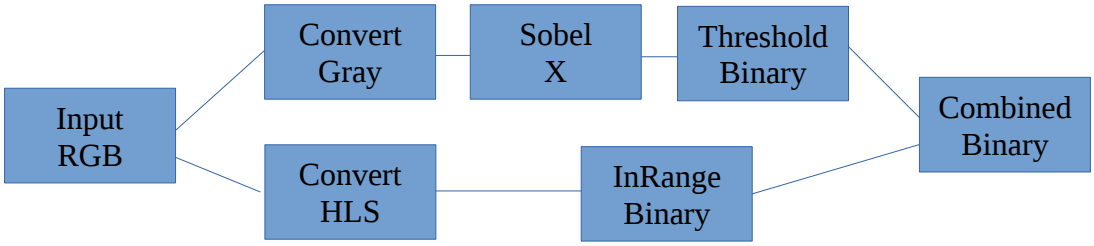
Once we have calibrated the camera as above, we can now apply the camera matrix and distortion coefficients to correct distortion effects on camera input images. This is done using the OpenCV undistort() function. For example, here are two images, one before undistortion correction, and one after. The effect is subtle, and most noticeable towards the edges of the image.

For example, look at the tree on the right side of the frame. You can see it's shape changes somewhat in the undistorted image.

Binary Thresholding

The Thresholding stage is where we process the undistorted image and attempt to mask out which pixels are part of the lanes and remove those that are not. This sounds far easier than it actually is. To achieve this, I use two methods and combine the outputs of both to generate my result.

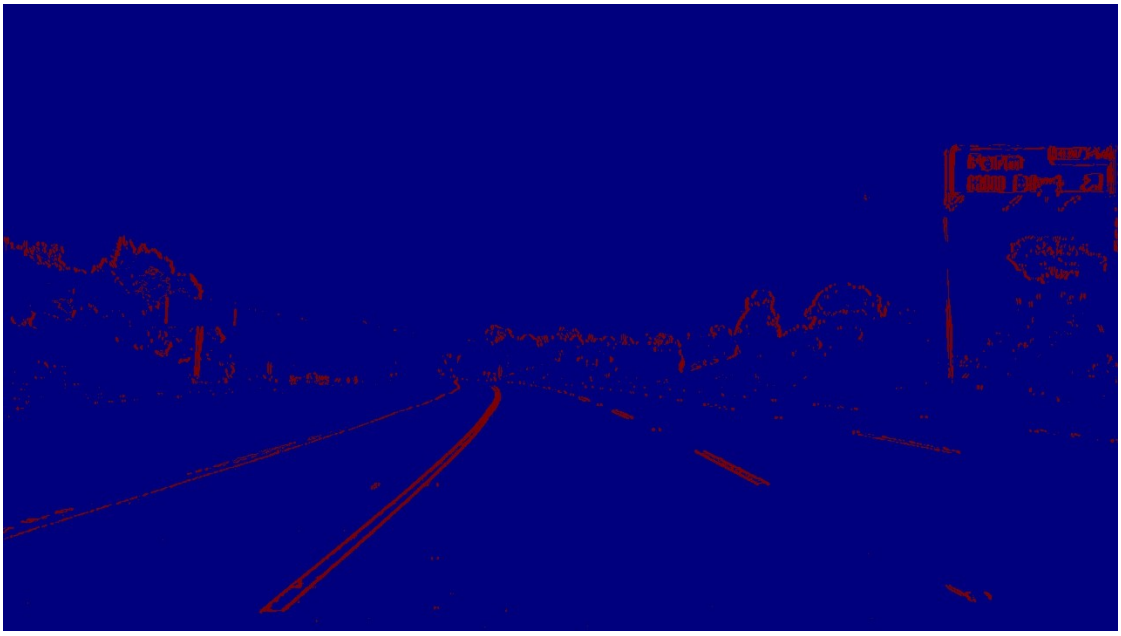
I have a mini-pipeline for this stage which looks like the following:



I take the input RGB image, and in the top path, I convert it to grayscale. I then apply a Sobel filter in the X direction to get image edges that match the direction of the lane lines. I then apply a threshold function on this to filter out out pixels that are not of interest. Through experiments, I found that min/max threshold values of 30 and 150 seem to work well. I use this output to generate a binary image of pixels of interest.

On the bottom path, I convert the RGB image to the HLS color space, and then use the S channel from that. The S saturation channel is useful for picking out lane lines under different color and contrast conditions, such as shadows. I then pass the S channel into an InRange function, again to filter out pixels that are not of interest. Through experiments, I found values of 175 and 250 to work best here. I also generate a binary image using this output.

Finally, I combine both the Threshold Binary and the InRange Binary to generate my final output, the Combined Binary. For reference, it looks like the following:

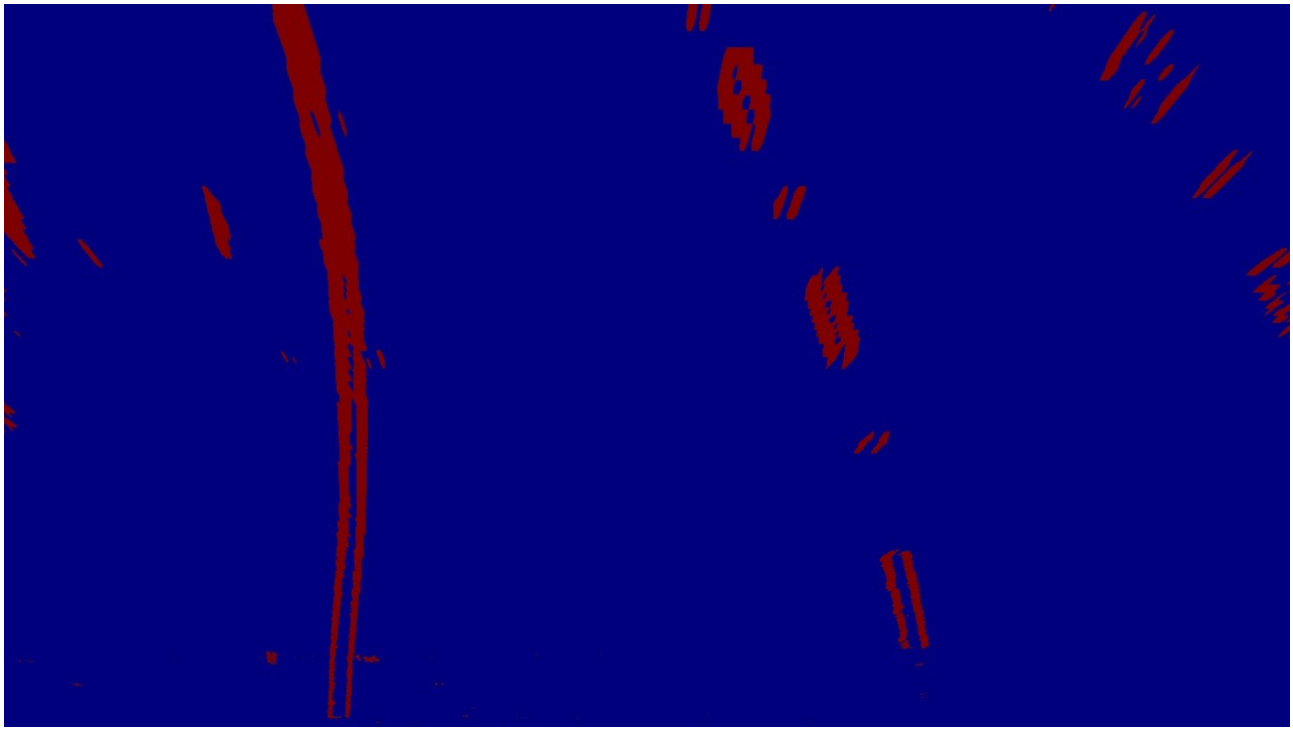


You can see that the lane lines are pretty well defined here.

Perspective Transform

Once I have the binary threshold image above, I apply a perspective transform on the image to generate an image with the effect of looking down on the road from above. I use the OpenCV warpPerspective() function to do this. I have defined a preset coordinate list to use for the perspective transformation, and the OpenCV getPerspectiveTransform() function generates a perspective matric using these.

After I have applied the perspective transform to the binary threshold image, I get an image that looks that lanes from above, similar to the below.



You can clearly see that left solid lane line, and the right dashed lane. There is still some noise in the image, which the next stage will remove.

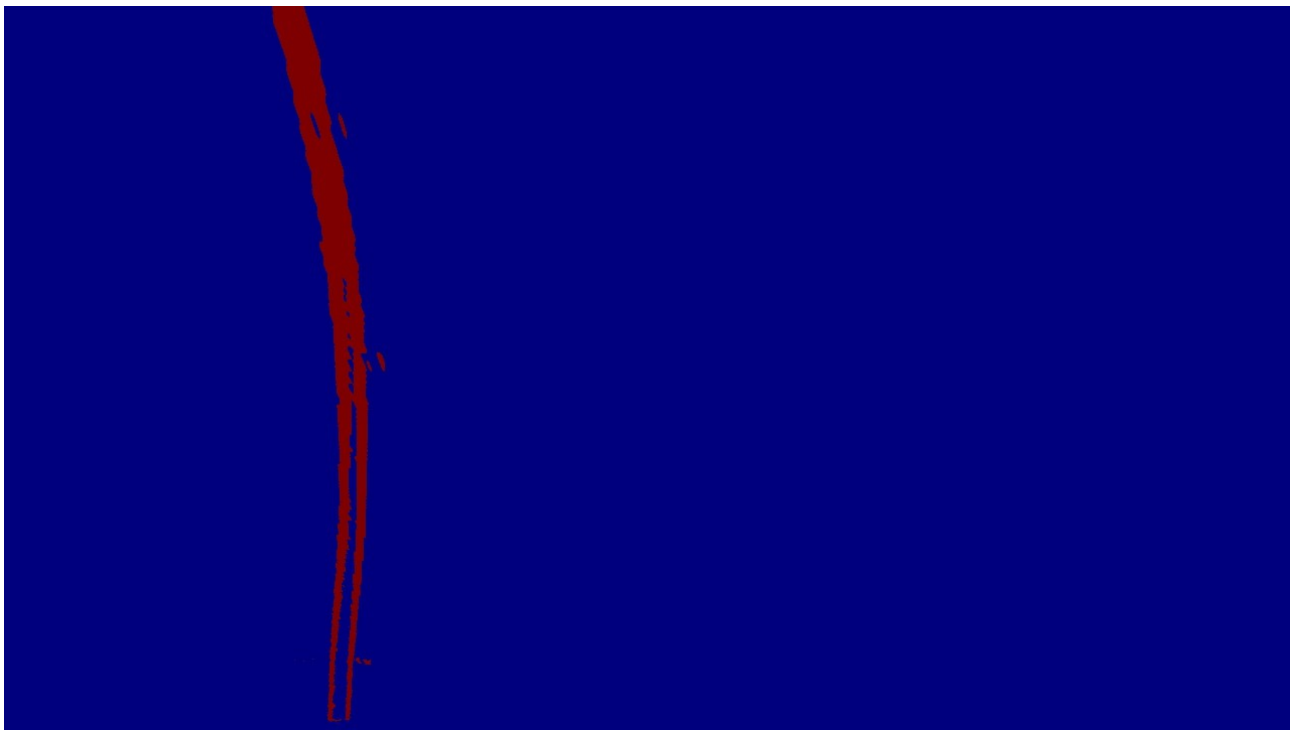
Locate Lanes

This stage is where we will try to extract the actual lane pixels for both the left and right lanes from the above image.

For the first frame from a camera, or for cases where we lanes are ‘lost’ (i.e. we have not been able to reliably detect a ‘good’ lane for a number of frames), I generate a histogram of the bottom half of the image. Then using the two peaks in this histogram, I determine a good starting point to start searching for image pixels at the bottom of the image. Let’s call these points x_left and x_right.

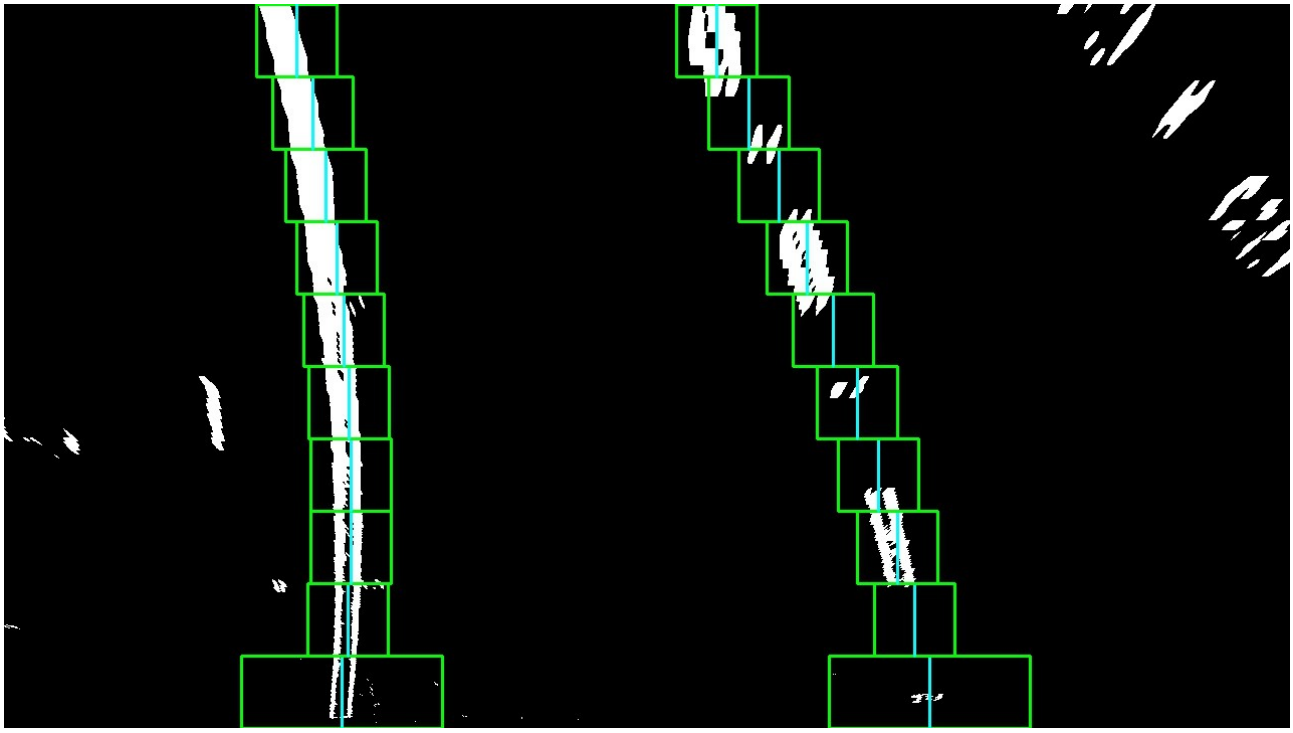
Once these points are calculated, I divide the image into 10 horizontal strips of equal size. For the bottom strip, I mask out everything outside of a small window around x_left and x_right in order to extract the pixels that belong to the lane, effectively discarded all other ‘noise’ pixels.

I repeat this process for each strip, using histograms on the strips to determine good x_left and x_right values. Once I have processed all strips, I then am left with images for both the left and right lanes. Here’s what the left lane image looks like:



As you can see, this is a pretty good representation of the left lane in perspective.

Since the above approach is quite slow, I only use this on the first frame or for when lanes get ‘lost’. For all other frames, I use the polynomial lane equation for the previous frame (caluculated in the stage below) to estimate the left_x and right_x values. I also track a lane x coordinate trend value to help better capture the lane lines. Here’s a picture of what my sliding window approach looks like for a typical frame:



Here you can see the left_x and right_x values, shown as blue lines for each horizontal strip. You can also see the windows I use around these values to determine which pixels are part of the lane, and thus masking all the other ‘noise’ out.

Fit Lanes

Once I have image for both the left and right lanes from above, then I use the Numpy polyfit() function to fit a polynomial equation to each lane. I also calculate the Radius of Curvature for each lane at this stage.

Once thee have been calculated, I do some checks to determine if the calculated lane if ‘good’ or not. First I check to see that the Radius of Curvature of the lane is above a minimum threshold. I selected this threshold by looking at the U.S. government specifications for highway curvature. Effectively, this checks to see that the calculated lane is not turning faster than we would expect.

The second check I do is to see that the left_x and right_x coordinates, essentially where the lane intersets the bottom of the image, have not changed too much from frame to frame. If you think about it, at 30 frames per second from a camera, the lanes should not change too much from frame to frame. I found that checking for a 15 pixel delta worked well for this check.

Finally, I check that each lanes Radius of Curvature is within 100 times (larger or smaller) than the previous frames values. As the RoC can be very large for vertical lines, I found that check for bounds of 100x seemed to work pretty well.

If any of the above checks fail for a lane, I consider the lane ‘lost’ or undetected for that frame, and I use the values from the previous frame. Again, at 30fps this should be ok for a short number of frames as the lanes will not change too much between sucessive frames. If a lane has not been successfully detected for 5 successive frames, then I trigger a full scan for detecting the lanes in the Locate Lanes section.

Draw Lanes

Finally, I draw the detected lanes back on to the undistorted image. We have to do an inverse perspective mapping to do this, as the lanes have been detected in the perspective view. I also annotate the left_x and right_x coordinates, as well as the Radius of Curvature for both lanes. Finally, I calculate a value for how far the car is from the center of the lane and annotate this too. The final output looks like this:

What Could Be Improved

Right now, I’m using a pre-defined set of coordinates to genreate a perspective transform matrix. I could look at some way of programatically generating these coordinates instead of using a fixed set.

While the above works well on the project video, the challenge video is not soo pretty. I would need to do more work on the Binary Thresholding stage to help with light variances and high contrast patterns in the road surfaces.