WRITE UP FOR 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.

Rubric Points - Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one.

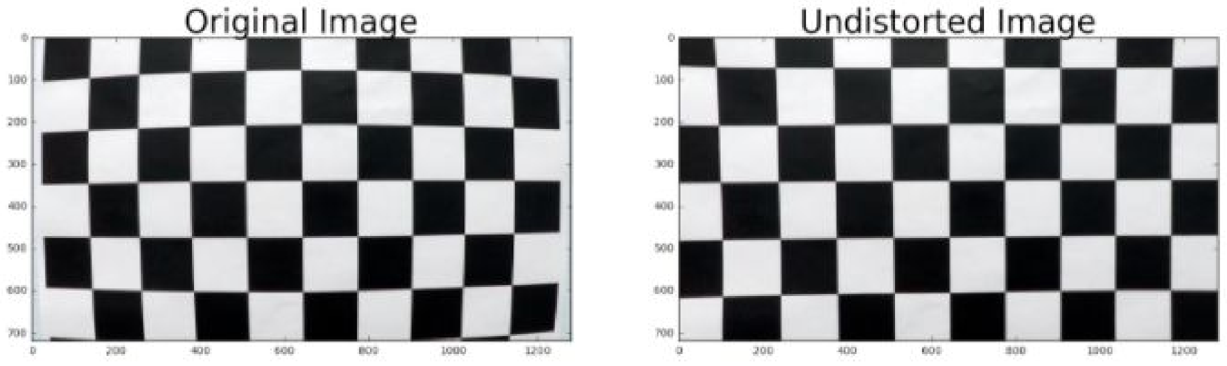
This is the writeup file.

Camera Calibration

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

The code for this step is contained in the third code cell of the IPython notebook. 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. The next step was to locate the "image points." I did this by graysca'ing the image and then using the cv2.findChessboardCorners() function. The `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

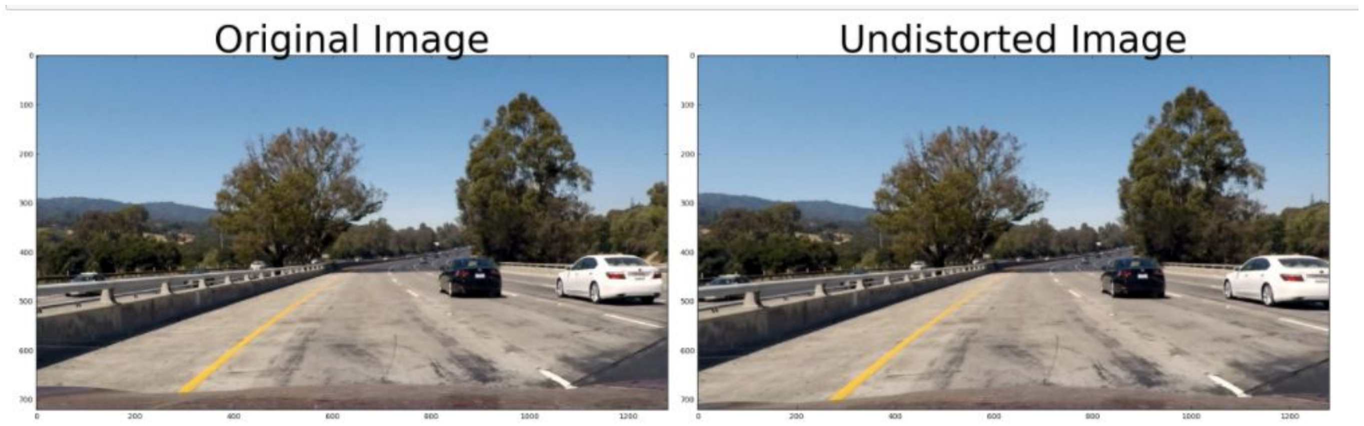
I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()`function and obtained this result:



Pipeline (single images)

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

Using the calibration_undistort() function, I was able to undistort images by first calibrating the camera with the objpoint and imprints, and then using the camera matrix and distortion coefficients to undistort. The result was an undistorted image like the one below.

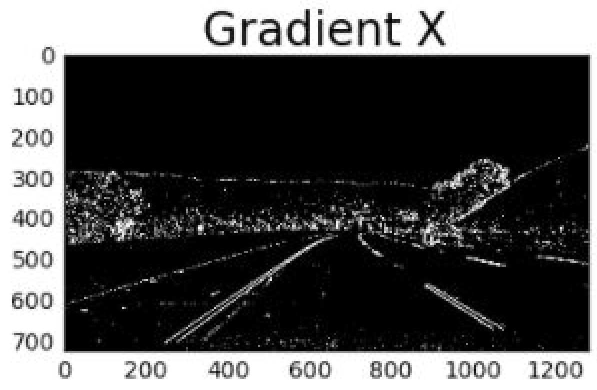


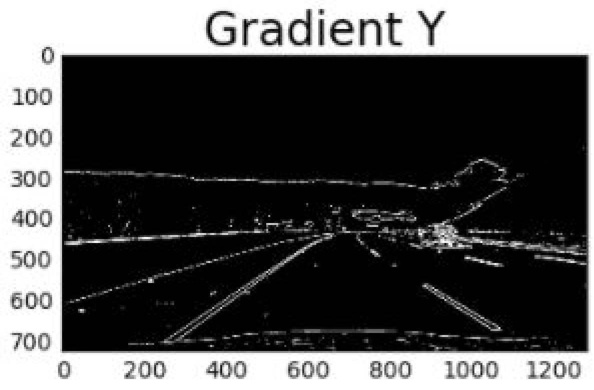
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

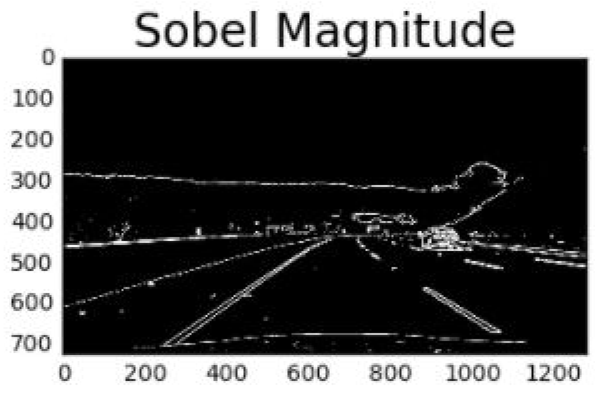
The first thing I did was design a function for each of the different tools described in the lectures. I created a threshold function for the HSL S-channel, the HLS L-channel, the absolute value of the sobel operator, the magnitude of the gradient, and lastly the direction of the gradient.

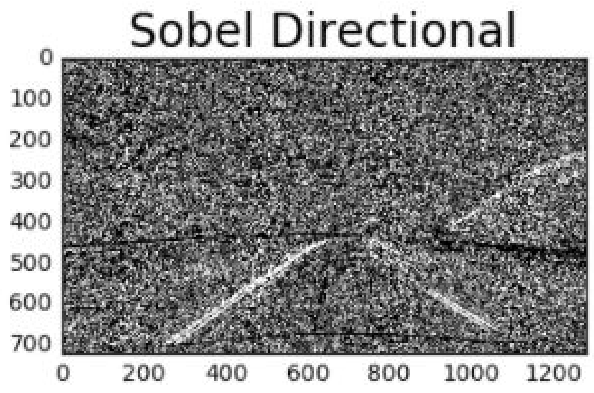
Then in cell 8 I experimented with the different thresholding options to see which ones output the best depiction of the lane lines. After a lot of trial and error, I realized that the L-channel and S-channel thresholds resulted in the best outputs, so I combined the two thresholds to create a binary image.

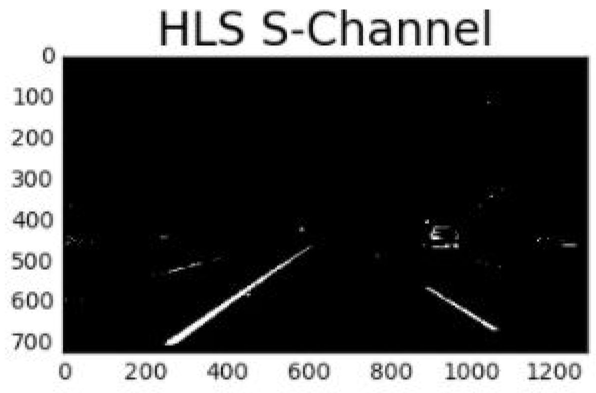
Here are a few examples of my outputs from thresholding:

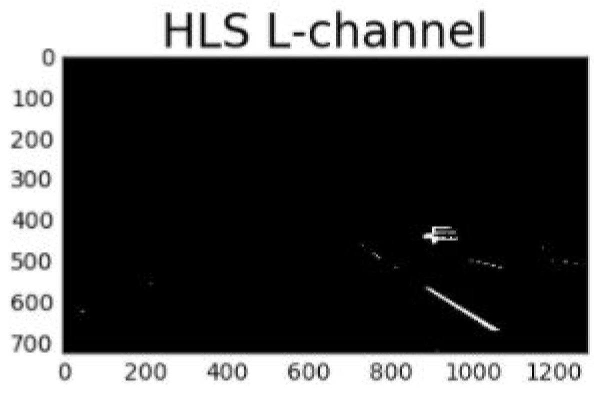












3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

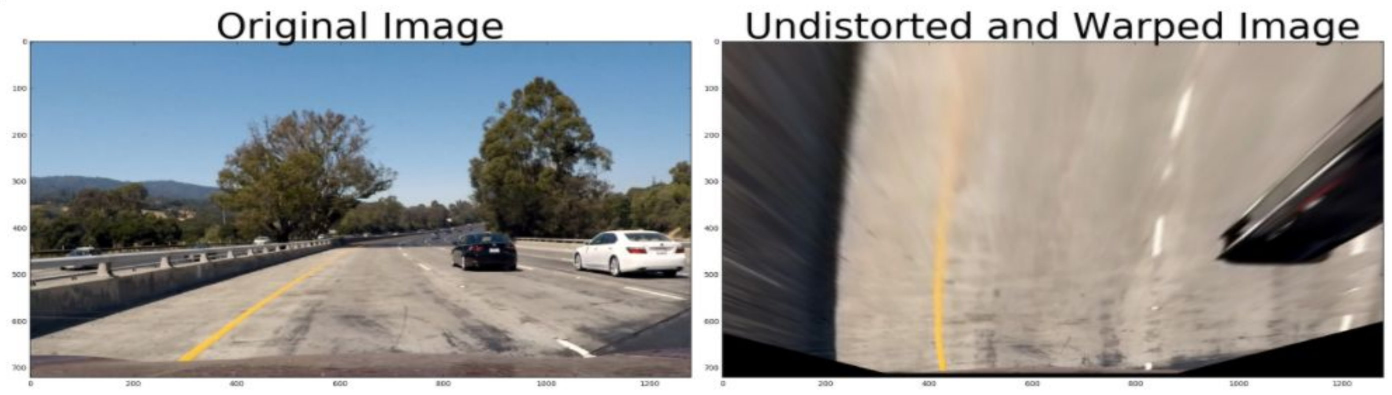
The next step was to apply an image transform to view the road from a top-down perspective. To do this, I created a function called perspective transform. The functions only input was an image, and it output the same image after transforming the perspective. I hard-coded all of the source and destination points by using an online image-mapping tool. In the future, I could use the image.shape[] to generalize for different sized images.

Here is a table with the coordinates for the destination and source points.

Source	Destination

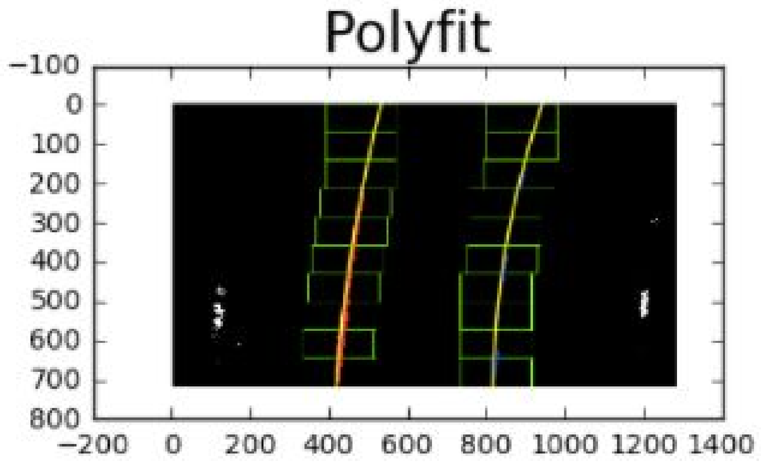
600, 450	400, 0
690, 450	800, 0
250, 680	400, 700
1040, 680	800, 700

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto an image and its warped counterpart to verify that the lines appear parallel in the warped image.



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

I then used a sliding window function to identify the lane lines. The function used a jistogram to determine where the peaks were in the lane lines. I did some other stuff and fit my lane lines with a 2nd order polynomial kinda like this:



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.

There were 3 main steps that I took to calculate the curvature of the lane and the position of the vehicle. The first step was focussed on the curvature. I used the pixel positions found in the sliding window search. Using those pixel positions and the polynomial fit, I converted everything to 'real world space' used those pixel positions and converted them to 'real world space'. I used the 3.7m and 30m measurements given in the lectures.

The next step was to determine the vehicle position in relation to the lane lines. I used the histogram to determine this. I found the midpoint of the histogram and then determined where the lane lines were in relation with that centerpoint. The final step was just drawing the curvature and lane position back onto the original image.

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

One really important thing to mention is that the first time I plotted the curvature back on the road, my code actually plotted across both lane lines (See image 1). After looking through the forums, I figured out that I needed to apply an image mask which solves the issue (see image 2).

Image 1 - No image mask:

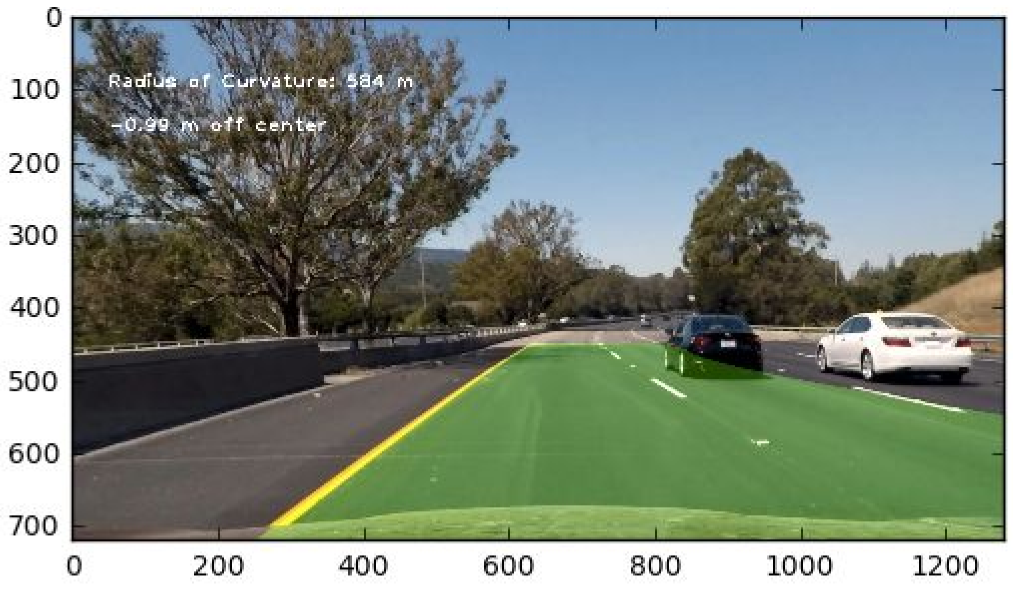
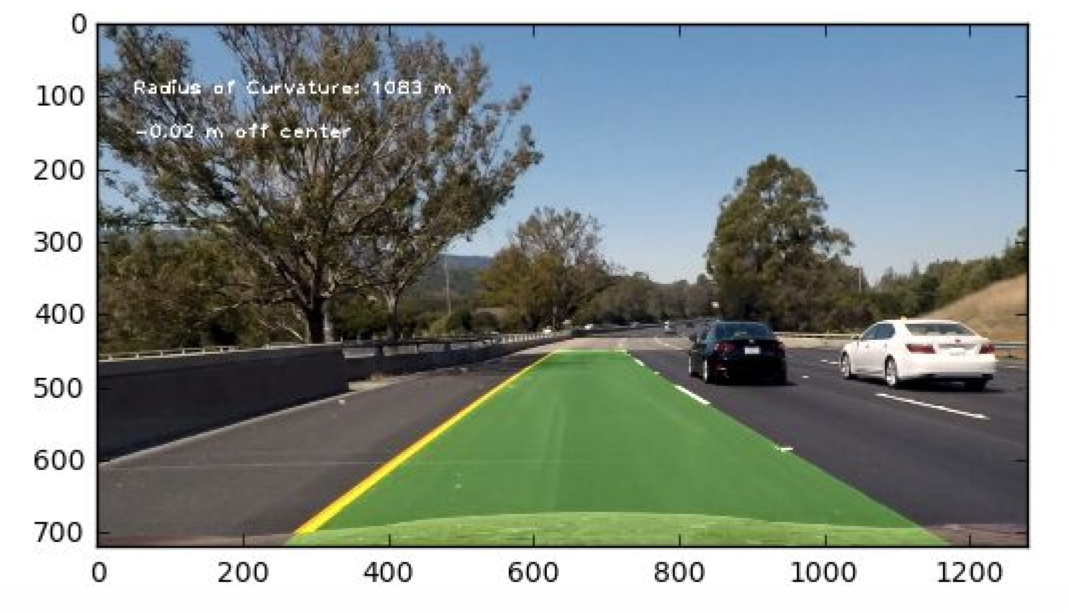


Image 2 - Image mask applied:



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!).

Here's a **link to my video result**

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?

What was the approach you took?

To create the final video out, my goal was to create a pipeline for each individual image in the video. The first step was to calibrate the camera. This gave me the camera matrix and the distortion coefficients which I then used to undistort the image. I then applied my thresholds to the image to identify the lane lines. Once properly thresholded, I transformed the perspective of the image to view it from the top down. With the top down image, I applied a sliding window search where I used a histogram to determine where the lane line pixels were. Once the pixels were properly identified, I drew lines on the pixels and fit the line with a polynomial. With the lines properly drawn, I then undid the perspective transform back to the original orientation I then calculated the curve of the radius and output it onto the image. This was the process that each image underwent.

What techniques did you use?

I used: camera calibration, distortion correction, color and gradient thresholding, perspective transform, sliding window search, polynomial fitting and radius curvature measurements.

What worked and why?

Specifically, for this video and the test images, the S-channel and L-channel thresholding worked very well. The pipeline was early able to detect the lanes also, applying an image mask significantly helped when determining which lane lines to identify. When I ran the pipeline without the mask, the pipeline drew the lane twice as wide into the other lane as well. You can imagine why this would be an issue.

Where would the pipeline fail?

Much of my pipeline is based upon the S-channel and L-channel thresholds. If something were to affect the colors of the image, such as extreme darkness, my pipeline would struggle. Another potential failure could happen with roadwork. There are times when the lane line in work zones are confusing, even for a human driver, so it would likely be difficult for a computer as well.

How might I improve it?

There are several actions I could take to improve this pipeline. The first would have been to implement transfer learning. With transfer learning, I could have allowed my pipeline to only search for a lane line close to where it identified the lane line in the previous image. Lane lines remain extremely close in orientation from one video frame to the next.

The next improvement I could have made was to implement a low pass filter for smoothing. By taking in the averages lane line pixels of the last *n* frames, I could have created a smoother output for the video, rather than the pipeline redrawing the lanes from frame to frame.

One last idea I had was to create an image classifier like the one in project 5. It may be more difficult to identify lane lines than cars, but still something to potentially consider about going forward.