writeup.md

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. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

Code

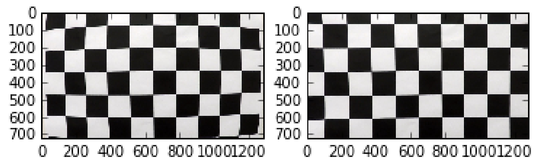
I have divided the code in two files: `Demo_LineDetection.py` is the execution file and `lineDetection.py` contains helper functions.

Camera Calibration

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

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, `objPoints` is just a replicated array of coordinates, and `checkboardObjCorners` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `checkboardCorners` 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 `checkboardObjCorners` and `checkboardCorners` 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. The code for calibrating the camera is contained in the function `myCameraCalibration()` in `lineDetection.py`. Below figure presents a calibration image before and after rectification.



Pipeline (single images)

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

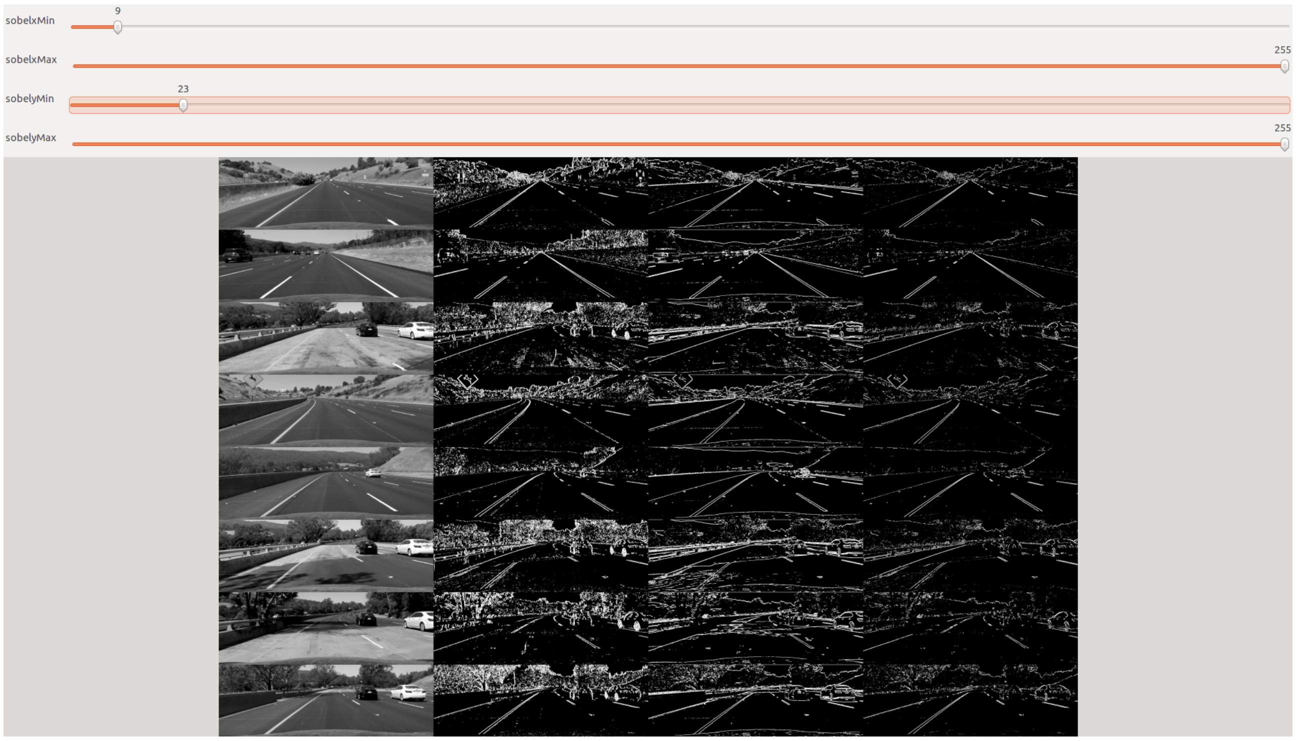
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

Similar to the checkerboard example, 'cv2.undistort()' was applied on the input image using the camera calibration matrix and distortion parameters.

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.

To detect lines, I used a combination of features: sobelx, sobely, magnitude, direction and the hls color space. A binary image - e.g. `bwSobelx` - was generated for each "feature" using an upper and lower threshold. Using the or-bitwise operation `|` the binary images were combined in pairs and later joined using the and-bitwise operation `&` . This is more easily expressed using code: `(bwSobelx & bwSobely) | (bwMag & bwDirection) | (bwL & bwS)` . To select appropriate thresholds a ligth-weight opencv gui was created to select appropriate thresholds. The GUI and the selected thresholds are presented below. Each row presents stacked test image and each column demonstrats the processing steps.

Column 1: Show input images. Column 2: Result after thresholding with min and max for the first feature. Column 3: Result after thresholding with min and max for the second feature. Column 4: Result after the two previous columns have been `and` 'ed together.

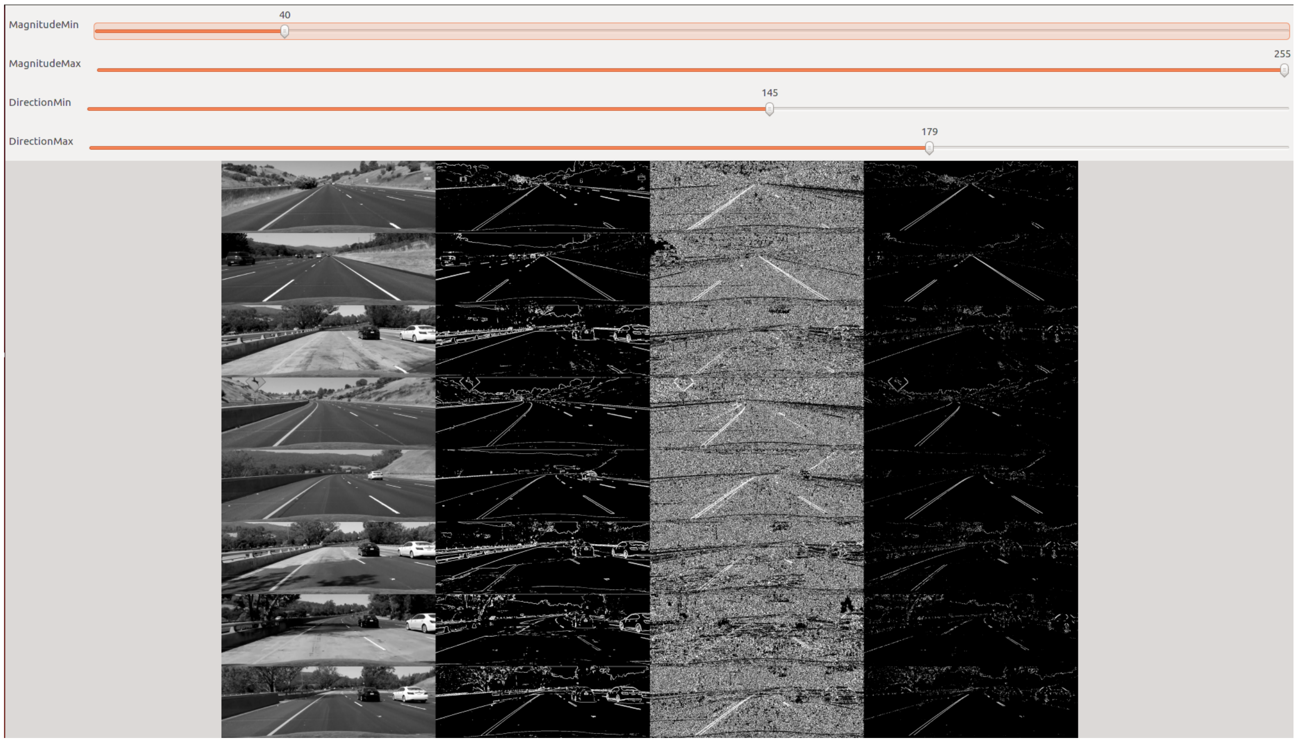


sobelxMin 9

sobelxMax 255

sobelyMin 23

sobelyMax 255

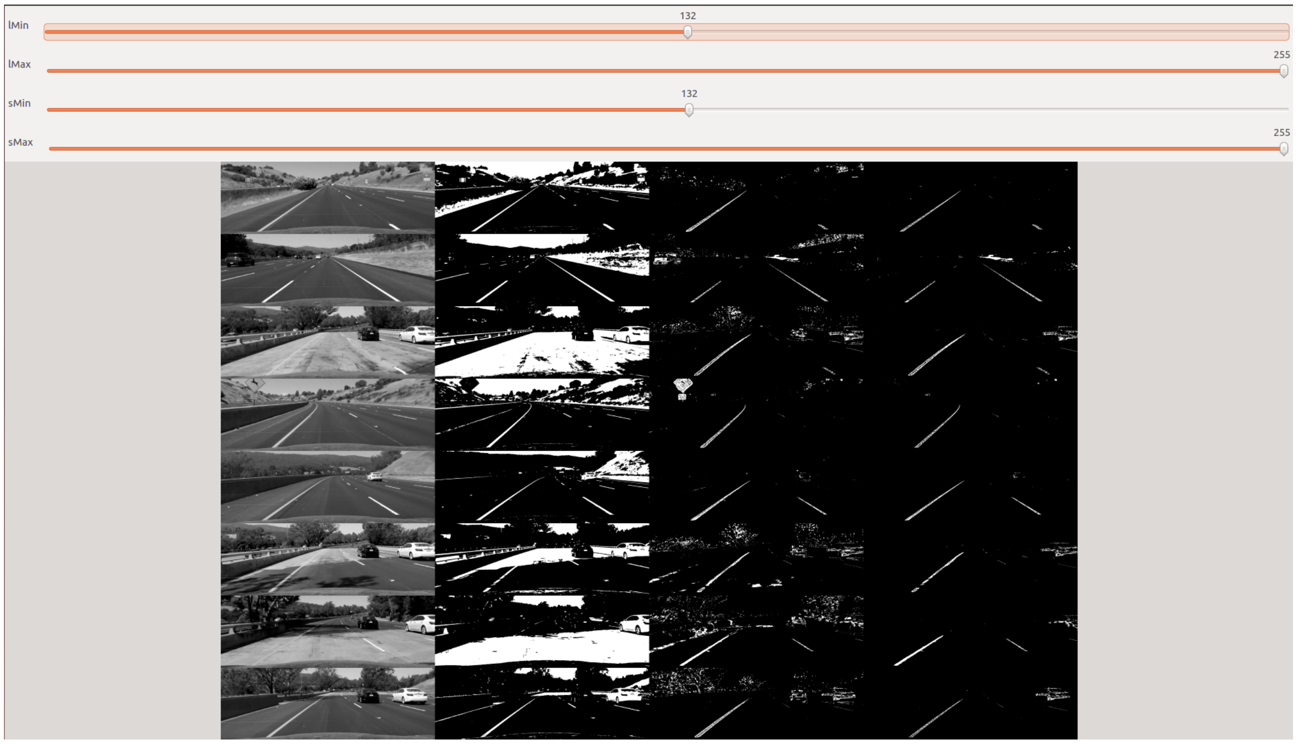


MagnitudeMin 40

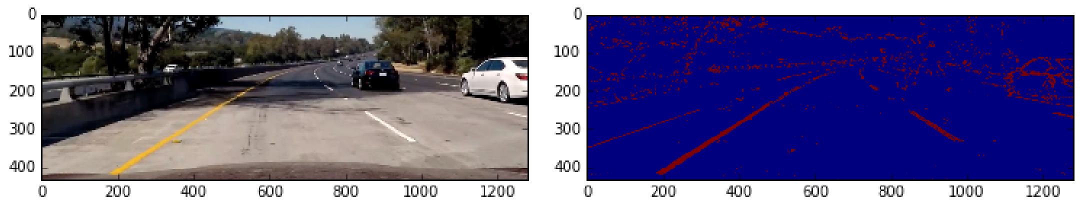
MagnitudeMax 255

DirectionMin 145

DirectionMax 179



The combined result.

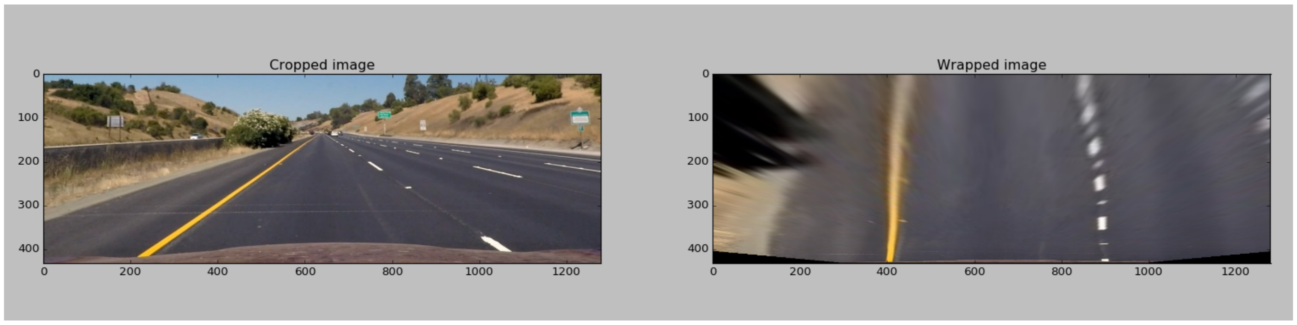


The code is devided in many sub functions in the `lineDetection.py` . The GUI is implmented in `visualizeAndSelectThreshold()` using other functions `applyThreshold(), performSobel(), abs_sobel(), magnitude(), sobelDirections(), hls()`

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

In `Demo_LineDetection.py` line 158-159 the source (`pSrc`) and destination (`pDist`) points have manually been selected and written to the two variables. The perspective projection (and its inverse) is specified in line 162-163.

Figure shows result.



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

Identifying and fitting the lane-line pixels to a polynomial required multiple steps.

1) A histogram of the bottom wrapped binary lane image is generated. The histogram is split into two and the peak value of each half was identified as the beginning position of a lane.

2) Based on the beginning position a sliding window approach is used to follow the lanes. A description of this is provided by udacity prior to the exercise.

3) The sliding window extract lane pixels that can be fitted using a 2. order polynomial function.

4) This polynomial function can be used to more easily detect lanes of the next image frame by only using lane pixel in a margin around the polynomial function.

5) However, to be less sensitive to errors in step 4. A sanity check is performed on the detected lanes by measuring the difference of the 2. order term for the left and right lane. A to big difference means that the lines are not bending similar and the algorithm is restarted to step 1.

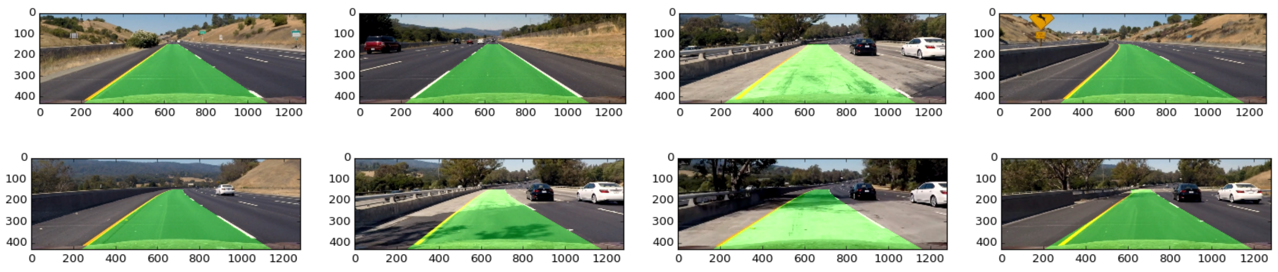
These steps uses helper functions in `lineDetection.py` : `fitLanesSlidingWindow()` and `fitLanesPoly()`

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.

I used the functions and code described in the udacity course material. Calculations of "radius of curvature" is located in the function `fitLanesPoly()` in `lineDetection.py` .

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

I implemented this using the functions and code described in the udacity course material. The visualization is created using `visualizeLaneDetection()` and `putTextCruveShift()` in `lineDetection.py` . Image examples of all test images is presented below:



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?

Tricks: Not everything worked from the start, so I had to add a few tricks in the process.

1) Creating a light-weight GUI for selecting optimal threshold-values - I don't know if this actually was faster than just trying out a few different thresholds :D

2) The binary image is remembered by a factor to use information from previous frames.

3) When fitting the polynomial function, I have weighted each lane-pixel by its row position. This ensures that lane-markings close to the vehicle are more heavily weighted and distant lanes are weighted less. This is reseanoable as distant lanes are detected less accurately and are of less important for controlling the vehicle.

4) After succesfully detecting the test images, I ended up having struggling with the video. Initially, I used the sliding window approach only in the first frame of the video and a margin around the polynomial function for all the other frames. The problem is that a single mistake will mess up lane-detections for the rest of the video. To fix this, I made a sanity check on the polynomial values for the left and right lane. A big difference in the 2. polynomial value - between left and right lane - indicates that the lanes are bending in different direction. A to big difference (>0.001) will reset and run the sliding window approach again.

What could you do to make it more robust?

1) Better detection of lane mark pixels. Add a bit of machine learning by training a classifier to detect lane markings... It might be fun running a small convolutional neural network (semantic segmentation) for detecting lines.

2) Better use of tracking and previous frames. Tracking of polynomial values. The algorithm remembered previous binary image lane detection by a factor, but more refined methods should be used.