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.

Camera Calibration

Camera is calibrated with help of snapshots of 9x6 chessboard patterns. The code for this step is contained in the third code cell of the IPython notebook located in "Lane finding.ipynb"

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, `pattern` is just a replicated array of coordinates, and `object_points` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgage_points` 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 `object_points` and `imgage_points` 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 in cell 4:

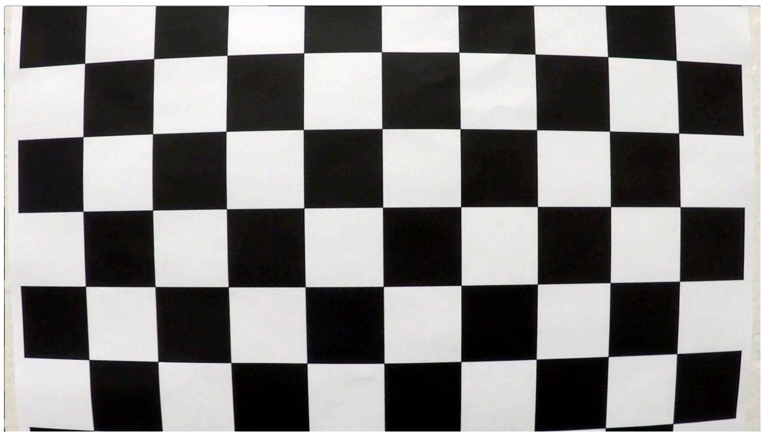

Figure 1: Original calibration pattern

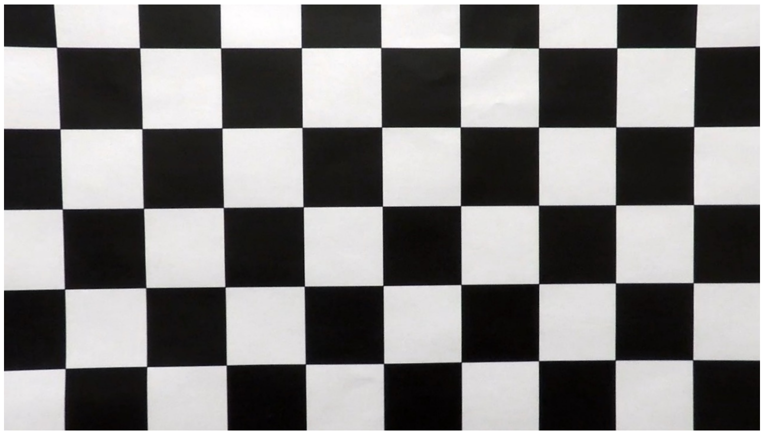

Figure 2: Undistorted calibration pattern

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 in cell 5:

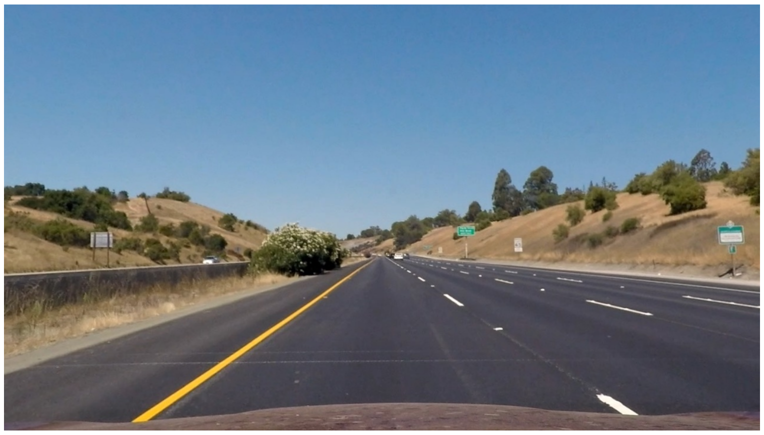

Figure 3: Original camera snapshot

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.

I used a combination of HSL and gradient thresholds to generate a binary image (thresholding steps in cell 7). Here's an example of my output for this step.

I used in final realization a saturation threshold `(80, 255)`, a luminosity threshold `(40, 255)` and a gradient magnitude threshold `(20, 255)`.

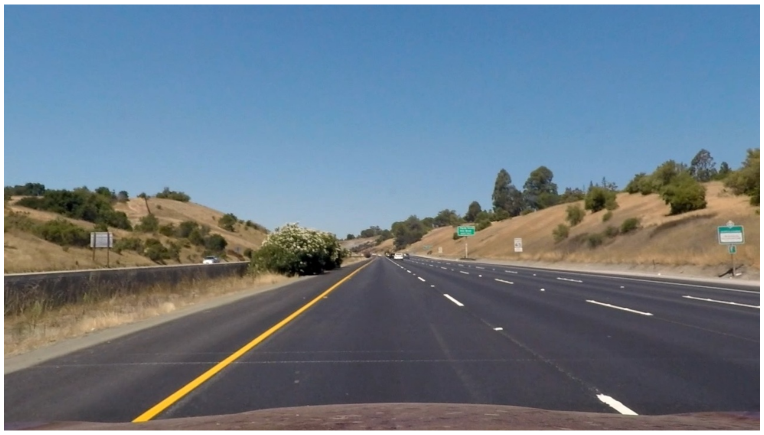
Figure 4: Undistorted camera snapshot

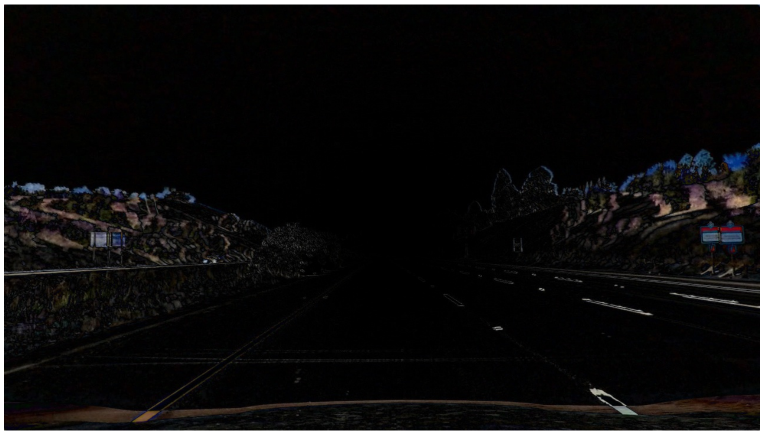
Figure 5: Difference between original and undistorted snapshots

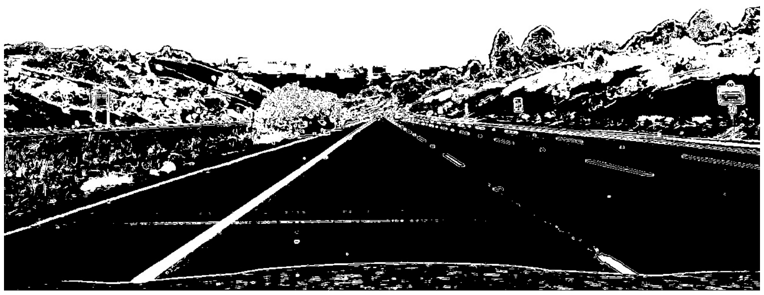
Figure 6: Binary output

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

The code for my perspective transform includes a function called `perspective_transforms()`, which appears in the cell 6 of the IPython notebook. The `perspective_transforms()` takes as input source (`src`) and destination (`dst`) points and returns two lambda functions `perspective_transform` and `perspective_inverse` that capture perspective transformations. Lambda functions take as inputs an image (`img`) and return respectively transformed images. I chose the hardcode the source and destination points in the following manner:

```
offset = 200
perspective_transform, perspective_inverse = perspective_transforms(
    np.float32([[315, 650], [554, 480], [736, 480], [1002,650]]),
    np.float32([[offset, 680], [offset, 0],  [1280 - offset, 0], [ 1280 - offset, 680]]))
```

This resulted in the following source and destination points:

Source	Destination
315, 650	200, 680
554, 480	200, 0
736, 480	1080, 0
1002,650	1080, 680

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

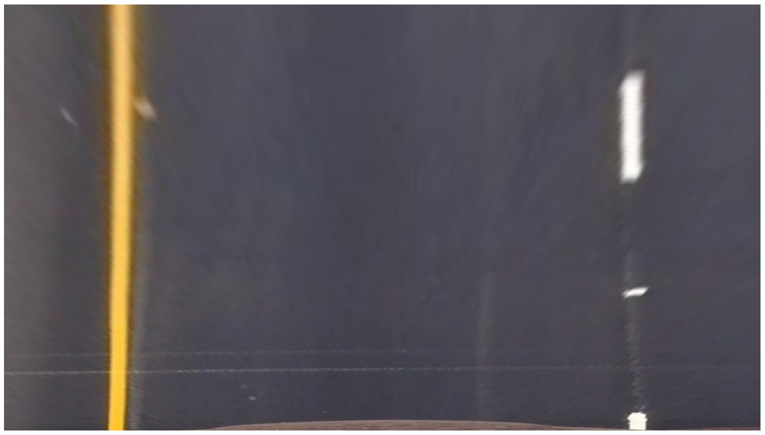

Figure 7: Warped image

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

I used truncated histogram of pixels in x-direction. To find pixels that correspond to lanes I sed window based search in the cell 8 I also added factor 1.75 to shift windows to capture lanes for challenge examples 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.

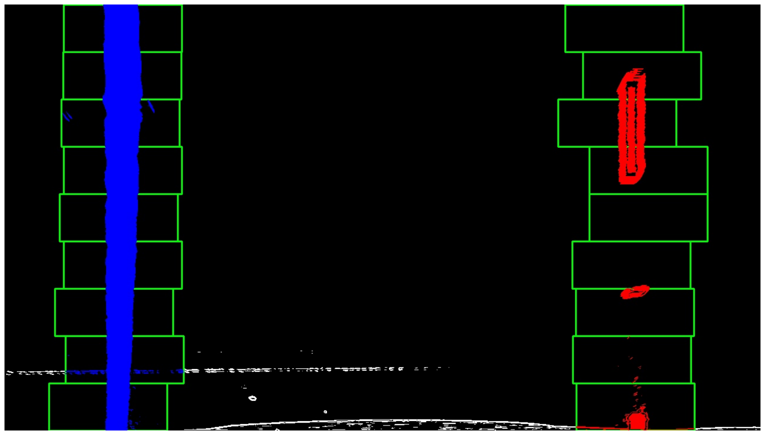

Figure 8: Windows search

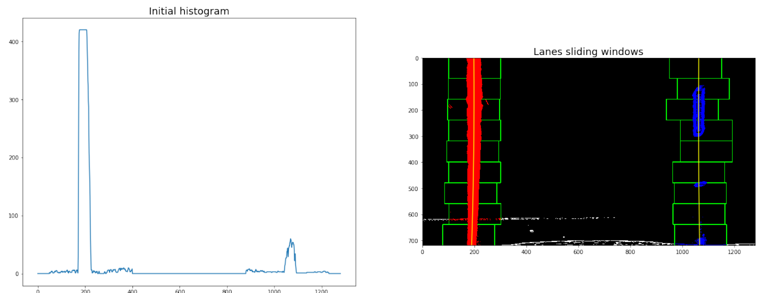

Figure 9: Lanes fitting

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 step in cell in the functions `plot_lane()` and `plot_overwview()`. Overview contains information about the radius of curvature, lane width and the offset from the center. The curvature is computed for with help of the linear transforms `x = kx X`, `y = ky Y` with `(X,Y)` in pixel units, and `(x,y)` in meters. For the polynomial fit `X = F(Y) = A Y^2 + B Y + C` or after transform `x/kx = A (y/ky)^2 + B y/ky + C` the curve in `(x,y)` coordinates is `(A kx/ky^2 y + B kx/ky y + C, y)` with derivatives `x'= 2 A kx/ky^2 y + B kx/ky`, `x''= 2 A kx/ky^2`, `y'=1`, `y''=0`, The curvature is `-2 A kx/ky^2 / ((2 A kx/ky^2 y + B kx/ky)^2 + 1)^1.5` or in pixel coordinates for `Y` is `k = -2 A kx/ky^2 / ((2 A kx/ky Y + B kx/ky)^2 + 1)^3/2` [1/m]. The radius of curvature is `1/|k|`, positive curvature corresponds to left turns, negative to right turns. Lane width is the value of `kx (right_fit(720) - left_fit(720))` and the offset is `kx (right_fit(720) + left_fit(720)) / 2`.

Also i added a debug window with the frames number, and numbers for accepted and rejected refits.

Here is an example of my result on a test image:

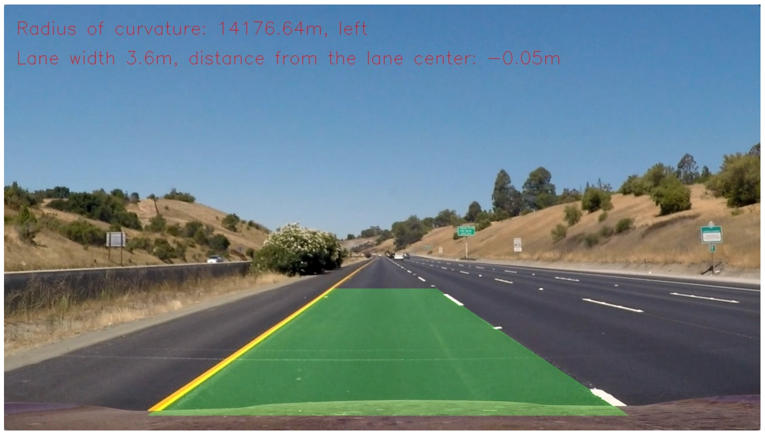
Radius of curvature: 14176.64m, left
Lane width 3.6m, distance from the lane center: −0.05m

Figure 10: Annotated test image

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?

The original project task is not very complicated, the main issue is to correctly find perspective transform and parameters for HSL and gradient filer.

Challenge task are more complicated. The first challenge task has very noisy video stream with many shadow regions where gradient and saturation thresholds pretty inefficient. I added a low-pass filter for polynomial coefficient and refitting of polynomial based on previous one to prevent jump lane oscillations **link to my video result**

The second challenge is even more complicated as video stream contains a lot of shadow and overexposed regions. Also quadratic polynomial is not a good choice for lane fits because in some places curve can have two turns. To avoid incorrect polynomial fitting a spline or piece-wise polynomial interpolation can be used. Also window search method is very inefficient on sharp turns, so to make windows more flexible in x-direction i added factor 1.75 for x updates in `find_lanes()` function **link to my video result**