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.

Camera Calibration

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

The Jupyter Notebook named `./Lane-Detection-System.ipynb` contains the code for this step. You can find it in the cell with the header "Camera Calibration".

There are three main functions OpenCV provides that I use to calibrate the camera, `findChessboardCorners()`, `calibrateCamera()`, and `undistort()`.

`findChessboardCorners()` finds a chessboard in an image and returns the corners of the squares' locations on the board. It takes the chessboard size, I pass in (9, 6) because that is the size of the chessboard in the images given to me. I loop through the calibration images with chessboard taken from different angles and save the corners' locations in the variable `imgpoints` to use later.

I also save object points which represent the coordinates of an undistorted chessboard for every image. I store these in the variable `objpoints`.

`calibrateCamera()` uses the image points and object points I found earlier to compute the camera calibration and distortion coefficients.

Finally, the `undistort()` function uses what was returned from `calibrateCamera()` to undistort an image.

I created the function `undistort_image()` to save my parameters to use later in the project.

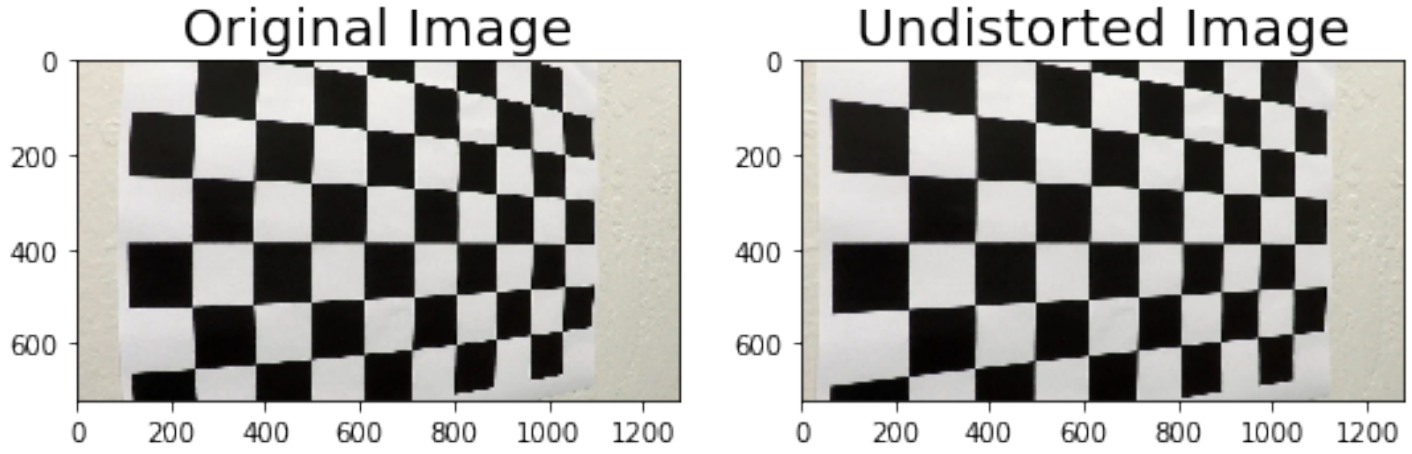

Figure 1: alt text

Pipeline (single images)

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

Figure 2: alt text

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 Jupyter Notebook named `./Lane-Detection-System.ipynb` contains the code for this step. You can find it in the cell with the header "Thresholds".

The main algorithm can be found in the function `create_binary_threashold_image()`. The algorithm is separated into 5 functions: `abs_sobel_thresh()`, `dir_threshold()`, `mag_thresh()`, `s_select()`, and `l_select()`.

`abs_sobel_thresh()` calculates the absolute gradient and applies a threshold.

`dir_threshold()` applies the sobel operator in both the x and y direction, computes the direction of the gradient, and then applies a threshold.

`mag_thresh()` applies the sobel operator in both the x and y direction, computes the magnitude of the gradient, and then applies a threshold.

`s_select()` computes the saturation level and applies a threshold.

`l_select()` computes the lightness level and applies a threshold.

The outputs of these methods are then combined into a single threshold image:

```
combined[(s_binary == 1) | (l_binary == 1) & ((gradx == 1) | (mag_binary == 1)) & (dir_binary == 1)] = 1
```

Here is an example output of this step (More examples can be found in the cell titled "Print Thresholds images"):

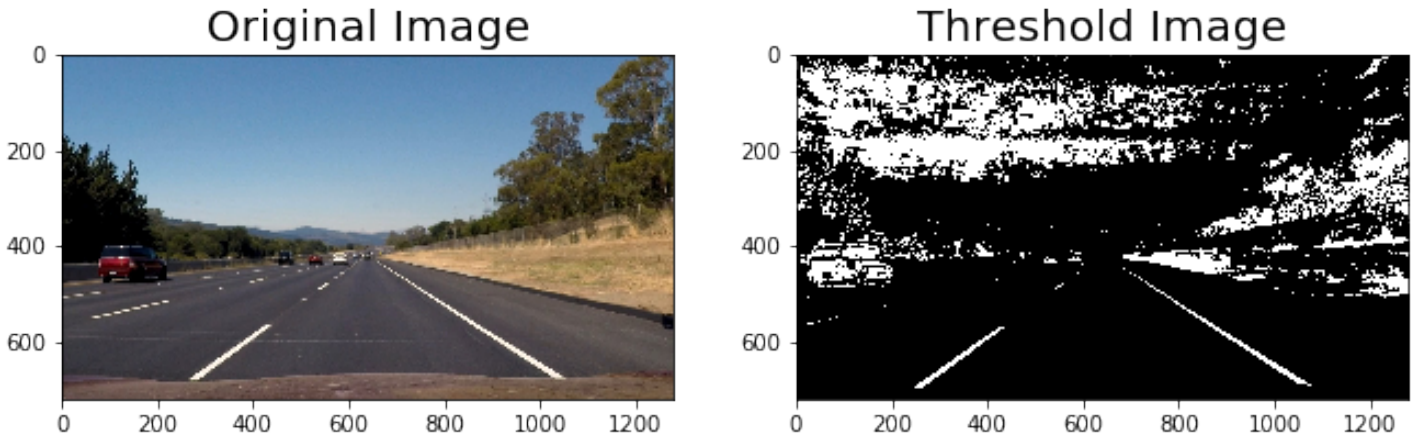

Figure 3: alt text

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

The Jupyter Notebook named `./Lane-Detection-System.ipynb` contains the code for this step. You can find it in the cell with the header "Perspective Transform".

The function `transform_to_birds_eye_view()` takes the image a returns it from a birds eye view. It's hardcoded with the following source and destination points:

```
src_points = np.array([[
    ((width/2)-50, (height/2)+90), # Top Left
    ((width/2)+50, (height/2)+90), # Top Right
    (200, height),                 # Bottom Left
    (width-200, height),           # Bottom Right

]], dtype=np.float32)

dst_points = np.array([[
    ((width/4), 0),         # Top Left
    ((width/4)*3, 0),       # Top Right
    ((width/4), height),    # Bottom Left
    ((width/4)*3, height), # Bottom Right
]], dtype=np.float32)
```

I found these points by guessing and checking. I used OpenCV's `getPerspectiveTransform()` to get the transformation matrix, and then used the `warpPerspective()` function I was able to produce the image.

Here is an example image I got from using this method (More examples can be found in the cell titled "Print birds eye images"):

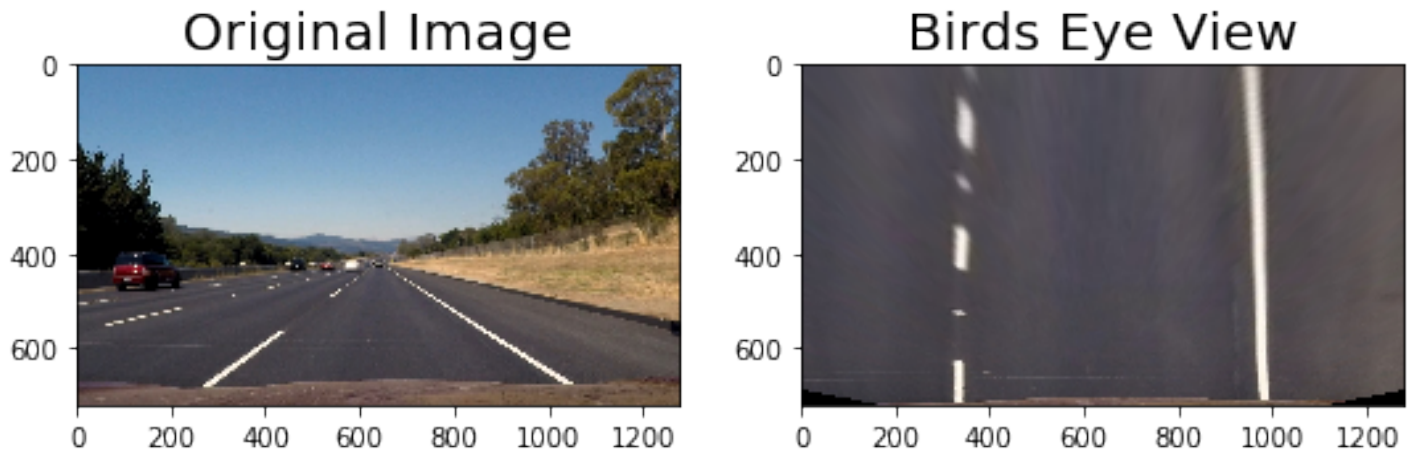

Figure 4: alt text

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

The Jupyter Notebook named `./Lane-Detection-System.ipynb` contains the code for this step. You can find it in the cell with the header "Search around poly".

Assuming we already have detected the lanes in the previous frame, we can then search around the previously detected polynomial for the lanes in the next frame.

In the function `search_around_poly()` I first find all the activated pixels that are around the margins of the previously detected lanes and then fit a polynomial to those points. The `fit_in_left_line` and `fit_in_right_line` functions take a position and tell you if they are within the margins, I then run them on every activated pixel and store the results in `left_lane_inds` and `right_lane_inds`. I then run those indices through the `fit_poly()` method which fits pixels to a second order polynomial (The `fit_poly()` method can be found in a previous cell in `./Lane-Detection-System.ipynb`).

Here is an example image I got from using this method (More examples can be found in the cell titled "Print detected lanes"):

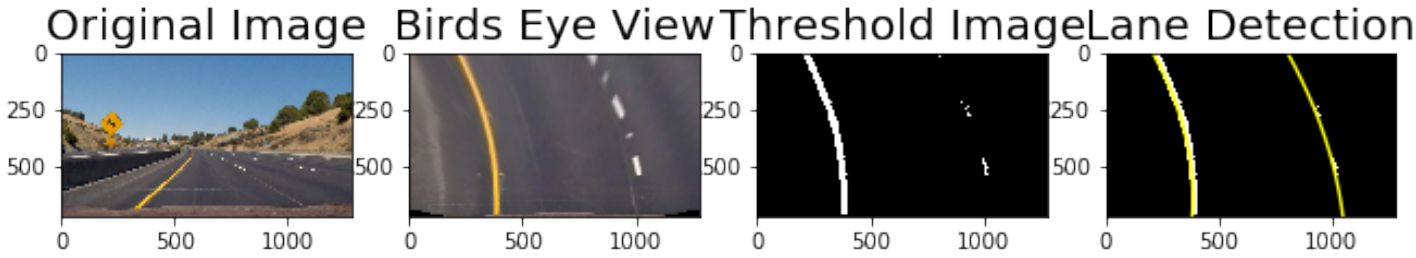

Figure 5: alt text

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.

The Jupyter Notebook named `./Lane-Detection-System.ipynb` contains the code for this step. You can find it in the cell with the header "Final image process".

The function `measure_curvature()` takes the polynomials found from the detected lanes and returns the radius of the curve. The equations for doing this can be found here: https://www.intmath.com/applications-differentiation/8-radius-curvature.php. For the final output I take the average radius from the left and right names.

The function `measure_vehicle_position()` returns how many meters the vehicle is from the center of the lane. I do this by finding by first finding the position of the lanes by taking the equations for the lanes and inputting the height of the image (which is the bottom of the image) to find the X coordinates. Once I have the positions of the lanes I can find the center of the lane, and then take the difference between that and the center of the image. I then have to convert that into meters before returning the value for the final output.

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

The Jupyter Notebook named `./Lane-Detection-System.ipynb` contains the code for this step. You can find it in the cell with the header "Warp lane back to undistored image".

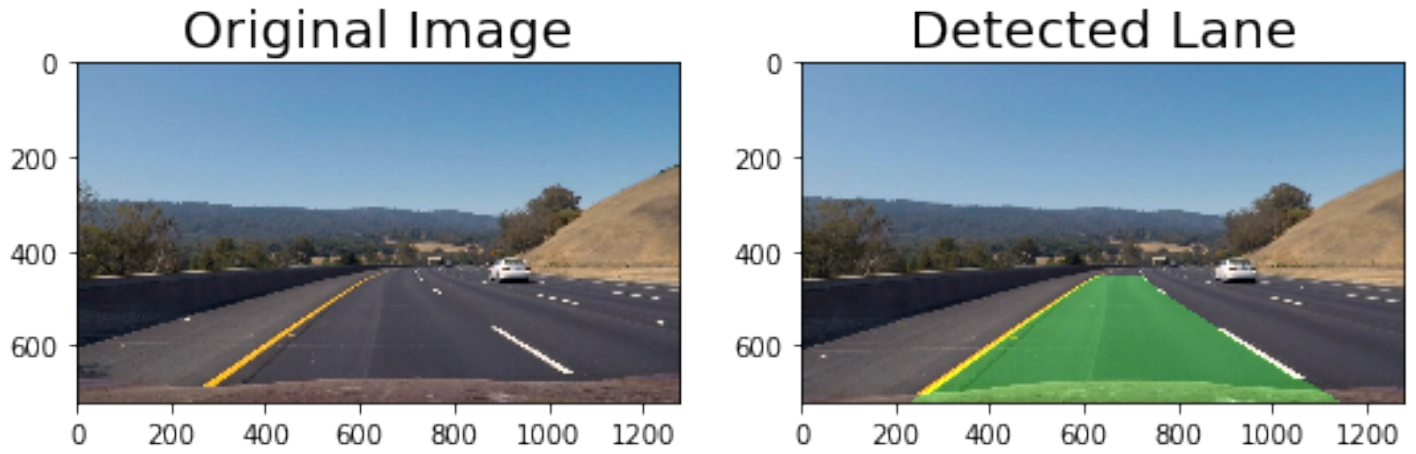

Figure 6: alt text

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!).

The Jupyter Notebook named `./Lane-Detection-System.ipynb` contains the code for this step. You can find it in the cell with the header "Final image process" in the function `process_image()`. The final output is in the final cell of the notebook.

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 most challenging part of this process is tuning the threshold image. It's not always clear what distinguishes a lane from other lines on the road, like shadows or cracks. My algorithm fails on roads with more with imperfections. I tried the challenge_video for this project, but it fails because there is a left shadow from a highway divider that my algorithm thinks is a lane. Also, the lane lines have less color saturation; I think it's an overcast day, which also makes it difficult.

Another issue is that this lane detection only works on highway clear highway roads. Even something as simple as an off-ramp would cause it to fail.

To improve this project I would first find ways to improve the threshold detection to detect lanes with lower saturation better and be better at ignoring shadows. I then might look at trying to detect different road features like off-ramps so my algorithm wouldn't fail if it came across one.