Advanced Lane Finding

Overview

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 ("bird's-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

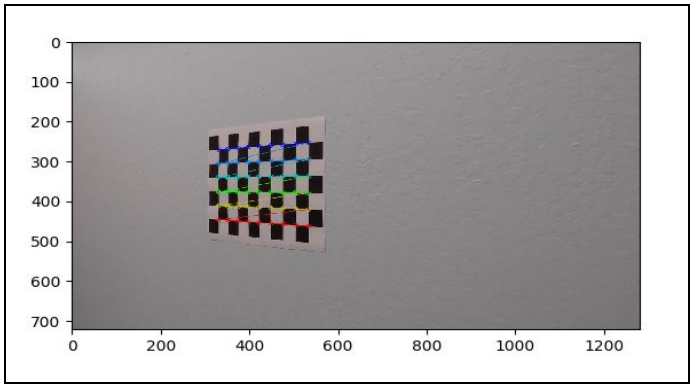
Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the file called `cam_calibration.py`.

The first step was to read in and display 20 calibration images of a 9x6(corners) chessboard. The images are located in the `camera_cal` folder. My goal is to map the coordinates of the corners (`imgpoints`) in these 2D images to the 3D coordinates of the real undistorted chessboard corners (`objpoints`). I prepared the `objpoints` array by initializing it with `np.zeros` and then generated the x,y coordinates that I wanted (z = 0, so I don't need it) using `np.mgrid`.

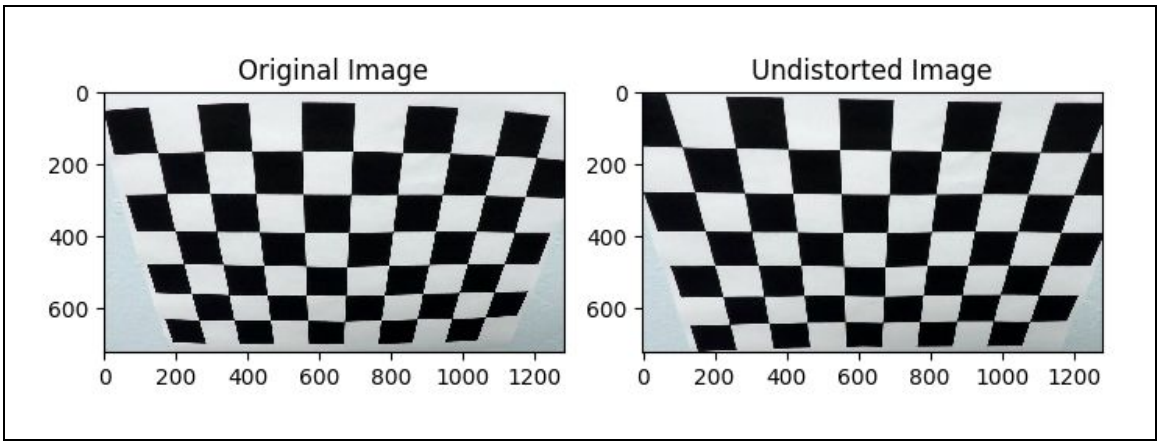
To create the `imgpoints`, I had to detect the corners of the board in the calibration image. I did that with the help of the OpenCV function `cv2.findChessboardCorners()`.

The image input of this function has to be grayscale. Then, I drew the detected corners on the images with `cv2.drawChessboardCorners()`.



Lastly, I used the extracted `objpoints` and `imgpoints` to compute the camera calibration matrix and the distortion coefficients using

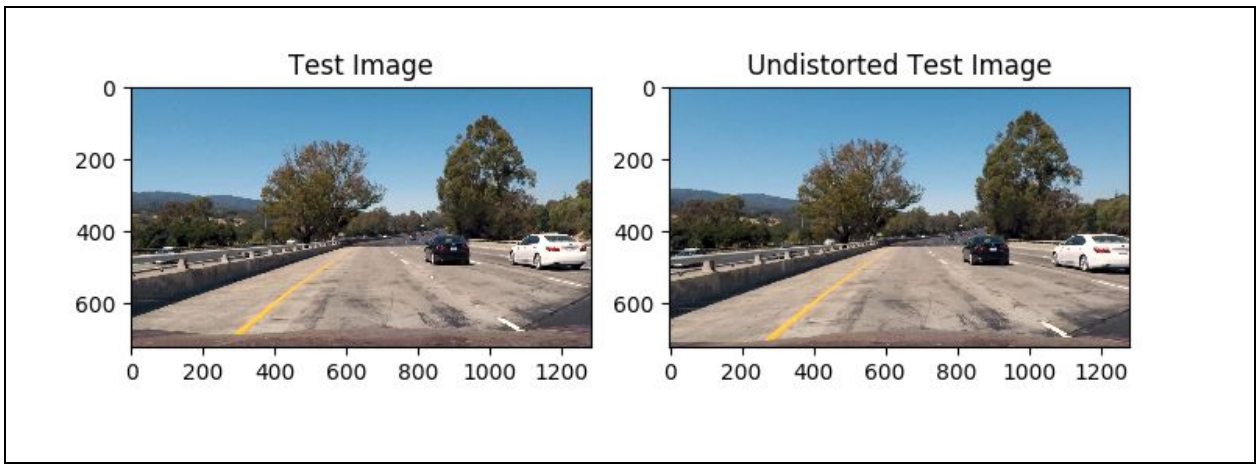
`cv2.calibrateCamera()`. I applied this distortion correction to all my images using the function `cv2.undistort()`. This is an indicative example of the results I got:



Pipeline (test images)

Provide an example of a distortion-corrected image.

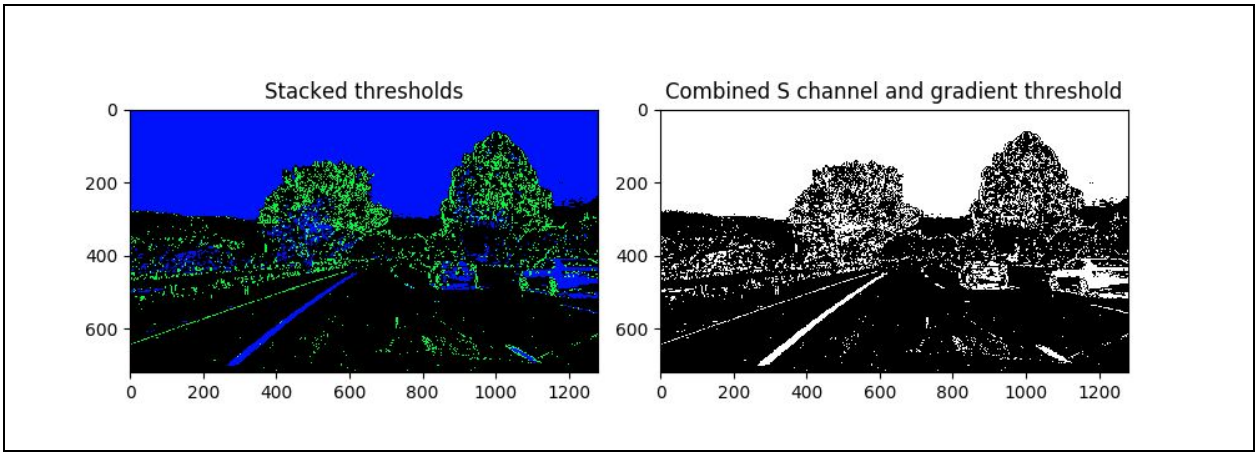
The code for this step is contained in the lines 9 through 40 of the file called `undist.py`. I applied distortion correction to each image contained in the `test_images` folder, by first loading the camera matrix and distortion coefficients which I calculated before, then creating a function that uses `cv2.undistort()`, and lastly applying this function while iterating over all images contained in the `test_images` folder. This is an indicative result I obtained:



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 code for this step is contained in the lines 45 through 102 of the file called `undist.py`. I combined color and gradient thresholds to obtain binary images

containing likely lane lines. Specifically, I created the `abs_sobel_thresh()` function that takes the derivative of the images, by applying the Sobel operator in the x direction. In addition, I created a function called `hls_select()` that converts my images to HLS color space and applies a threshold to the S-channel, which is the one that picks up the lines fairly good in the given images. You can see in my output images contained in the `thresholded_images` folder, that for each test image I plotted a stack of two binary images (Stacked thresholds), showing which parts of the lane lines were detected by the gradient threshold (green color) and which ones were detected by the color threshold (blue color), as well as a binary image (Combined S channel and gradient threshold) combining the two thresholds. This is an example:

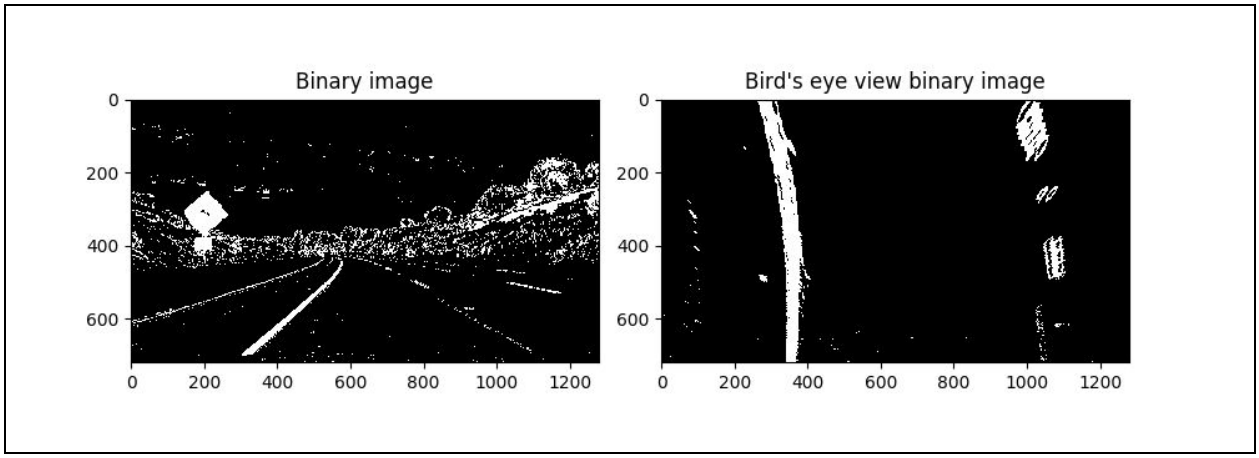


Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for the step of perspective transform is contained in the lines 107 through 138 of the file called `unsdist.py`. I created the function `warper()` which takes a thresholded image and warps it using the perspective transform M, which is computed using the following OpenCV function `cv2.getPerspectiveTransform(src, dst)`. I hardcoded the source and destination points, as seen in the table below.

Source Points	Destination Points
570, 460	280, 0
230, 700	280, 720
1035, 700	980, 720
690, 460	980, 0

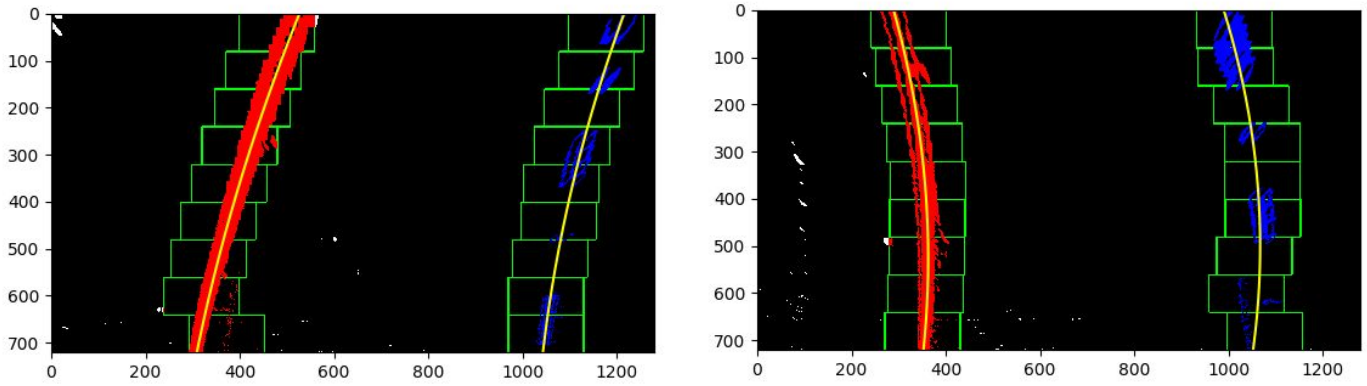
I iterated over the test images and got a bird's eye view of the lanes. The transformed images are located in the `output_images` folder. Below you can see one of these output warped images and that the lanes appear parallel in it.



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

The code for this step is contained in the lines 143 through 232 of the file called `unsdist.py`. After extracting the binary images, where the lanes stand out pretty clearly, I took a histogram along all the columns in the bottom half of each transformed image, in order to classify the pixels that belong to the left line and the pixels that belong in the right line. I did it like this; `histogram=np.sum(transformed_image[transformed_image.shape[0]//2:,:], axis=0)`.

Then, I used a sliding window, placed around the line centers, to find and follow the lines up to the top, and afterwards fit a second order polynomial to each one, using `np.polyfit()` function. This was done for each transformed image of all test images. I used the code provided in the course, and I tuned the margin of my windows, so as this implementation to work robustly for all the test images given. You can find the results of this implementation in the `polynomial_fit` folder. These are some indicative examples:



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 code for this step is contained in the lines 234 through 252 of the file called `unsdist.py`. At first I converted my x and y pixel values to real world space. As suggested in the course, I assumed the lane is about 30 meters long and 3.7 meters wide. Then, I fit new polynomials to x,y in world space like this:

```
left_fit_cr = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
right_fit_cr = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix, 2)
```

Finally, I calculated the new radii of the right and left curvatures, as well as their average value.

In the lines 248-253, you can find the calculations of the vehicle position. I assumed the camera is mounted at the center of the car and the `car_offset` value is the deviation of the midpoint of the lane from the center of the image.

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

The code for this step is contained in the lines 255 through 278 of the file called `unsdist.py`. In this step, I projected my results back down onto the undistorted road images (undistorted test images). You can find the output images in the `output_images/road` folder. Specifically, I created an image called `color_warp` to draw the lines on and then used the inverse perspective matrix `Minv`, which was calculated earlier in the `warper()` function to warp the blank image back to the original image space. Lastly, I combined the result with the original undistorted test images and draw on them as well text that indicates the curvature and the offset.

Pipeline (video)

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 image processing pipeline is located in the file called `lane_finding_video_gen.py`. I adjusted my previously developed pipeline for the test images in the `pipeline_final()` function, so as to generate a video, where the lanes are identified in every frame, and outputs are generated regarding the radius of curvature of the lane and vehicle position within the lane.

After extracting a preliminary result, I noticed that my pipeline failed when the color of the pavement changed or when shadows are present. So, I tried a different threshold approach; This time I converted images to HSV color space, and I applied white and yellow color masks, as well as the sobel operator in the x direction. Although, I obtained a smoother result, the problem still remained.

At last, I managed to achieve a far better result, by removing completely the sobel x operator mask. It seems that applying a yellow and white mask on the HSV color space is very robust to different lighting conditions.

Here is a link to the video: https://youtu.be/7CLFQJl2bog

Discussion

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?

I noticed that my pipeline would likely fail when the color of the pavement changes, or when there are shadows on the pavement, situations that generate more 'hot' pixels in my searching windows (see pictures below). Taking a better look at tuning the relevant parameters like the threshold margins, and improving the sliding windows search could make my pipeline more robust.