Writeup Template

Rubric Points

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

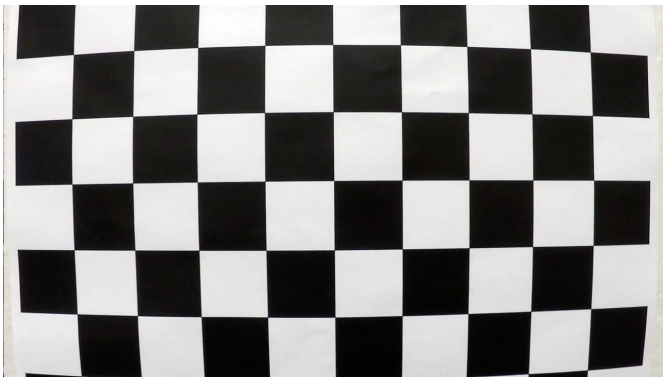
Camera Calibration

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

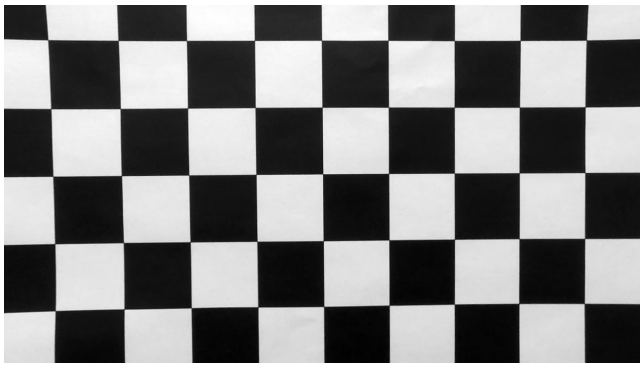
The code for this step is contained inside `src/camera_calibration.py`.

I start by preparing "object points" inside object_points(). These represent the (x, y, z) coordinates of the chessboard corners in the world. The coordinates are constant since we the object is not moving. Rather the camera is moving.

Next, for each image I find the chess board corners using findChessboardCorners(). OpenCV's built in calibrateCamera() function is then able to determine the calibration coefficients given the list of chess board corners and matching object points.



Original Image



Undistorted image

As a minor optimization, I load and process each image in parallel. After all steps are done, I cache the calculations to camera_cache.pickle.

Pipeline (single images)

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

To demonstrate this step I execute `self.undistort(test1)` on the following distorted test image.

Distorted image: test 1

The resulting undistorted image (below) looks similar the distorted image. Without the checkerboard pattern it is difficult to detect the difference.

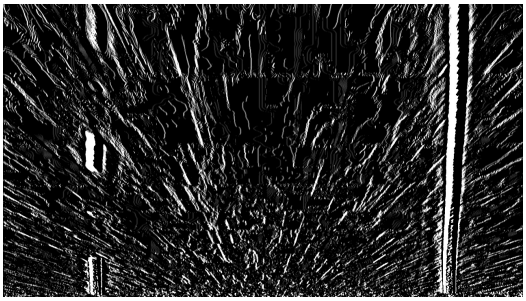
Undistorted test 1

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 created the binary image inside process_frame() after performing a perspective transform. I attempted dozens of different transformation types and toyed with using CLAHE to adjust for local conditions. Ultimately, I settled on a combination of Sobel transforms on the saturation channel and grayscale channel.

I started by extracting the saturation channel and then calculating various horizontal sobel transforms on it. The sobel kernel size of 3 contained very little noise. However, if wasn't effective for detecting edges towards the top of the image. Towards the top of the image I instead used a sobel kernel of 5.

Test image 2 **Saturation channel**

Horizontal sobel, filter of 3 **Horizontal sobel, filter of 5**

The saturation channel contained very little noise and was excellent at highlighting yellow lane lines. However, it had sometimes had difficulty detecting white lines. To overcome this issue, I also performed some horizontal sobel transforms on the grayscale image. These grayscale sobels performed poorly on yellow lanes. They were a good complement to the saturation images.


Test image 2 **Grayscale**


Horizontal sobel, filter of 3 **Horizontal sobel, filter of 5**

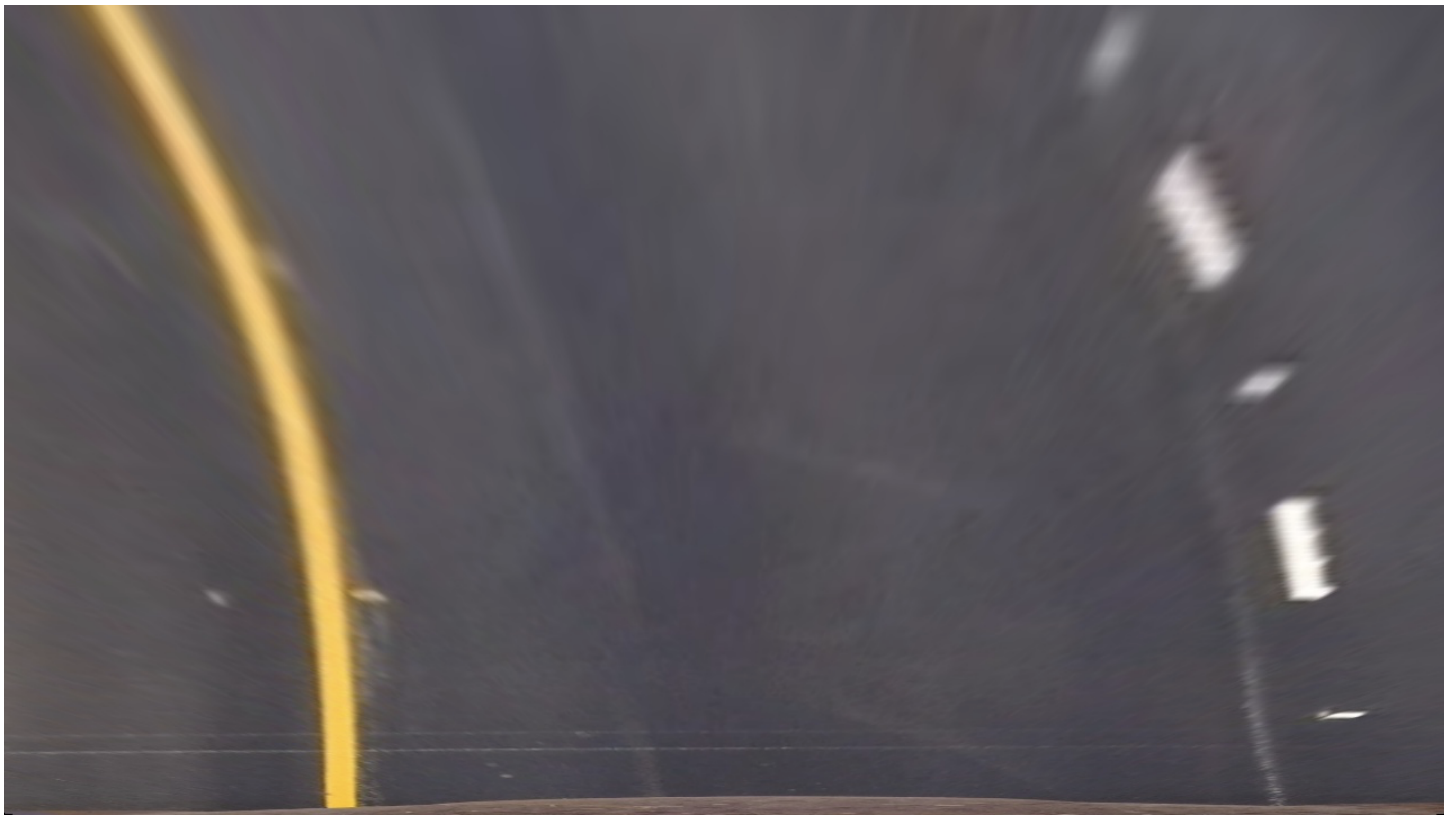
Finally, I combined the saturation_sobel3, saturation_sobel5, grayscale_sobel3, grayscale_sobel5 into a single binary image by applying different threshold to each input image. The threshold values varied depending on how high up the pixel was in the image. The final output looked like the following:

Binary image

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 is invoked at the beginning of my pipeline.process_frame() method. First, get_image_points() calculates the polygon inside the undistorted source image which would correspond to an overhead view. This calculation is based off the assumption that all objects are infinitely small at the horizon. I used straight_line1.jpg to determine where the horizon should appear in the image and where to offset from the side of the screen. The following is an example of such a polygon:

Next, inside get_destination_points() I calculate where the above polygon should be transformed into. I use CV2's getPerspectiveTransform() and warpPerspective() functions to perform this warping. The resulting output images look like the following:



I verified that my perspective transformation was working correctly by transforming straight1.jpg and straight2.jpg. When transformed into this new perspective the lines were correctly parallel to the side of the image.

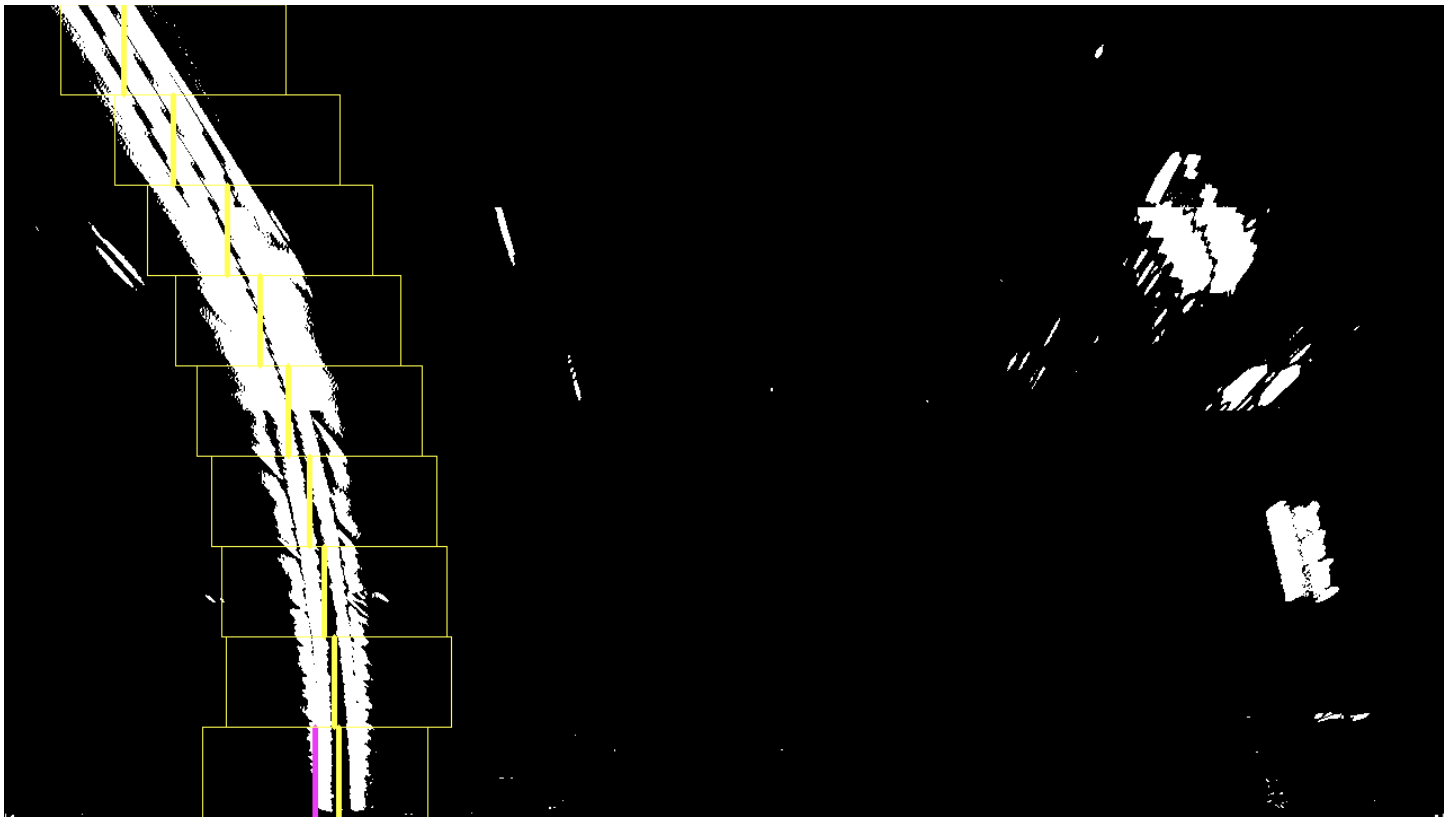
Warped straight1.jpg

Warped straight2.jpg

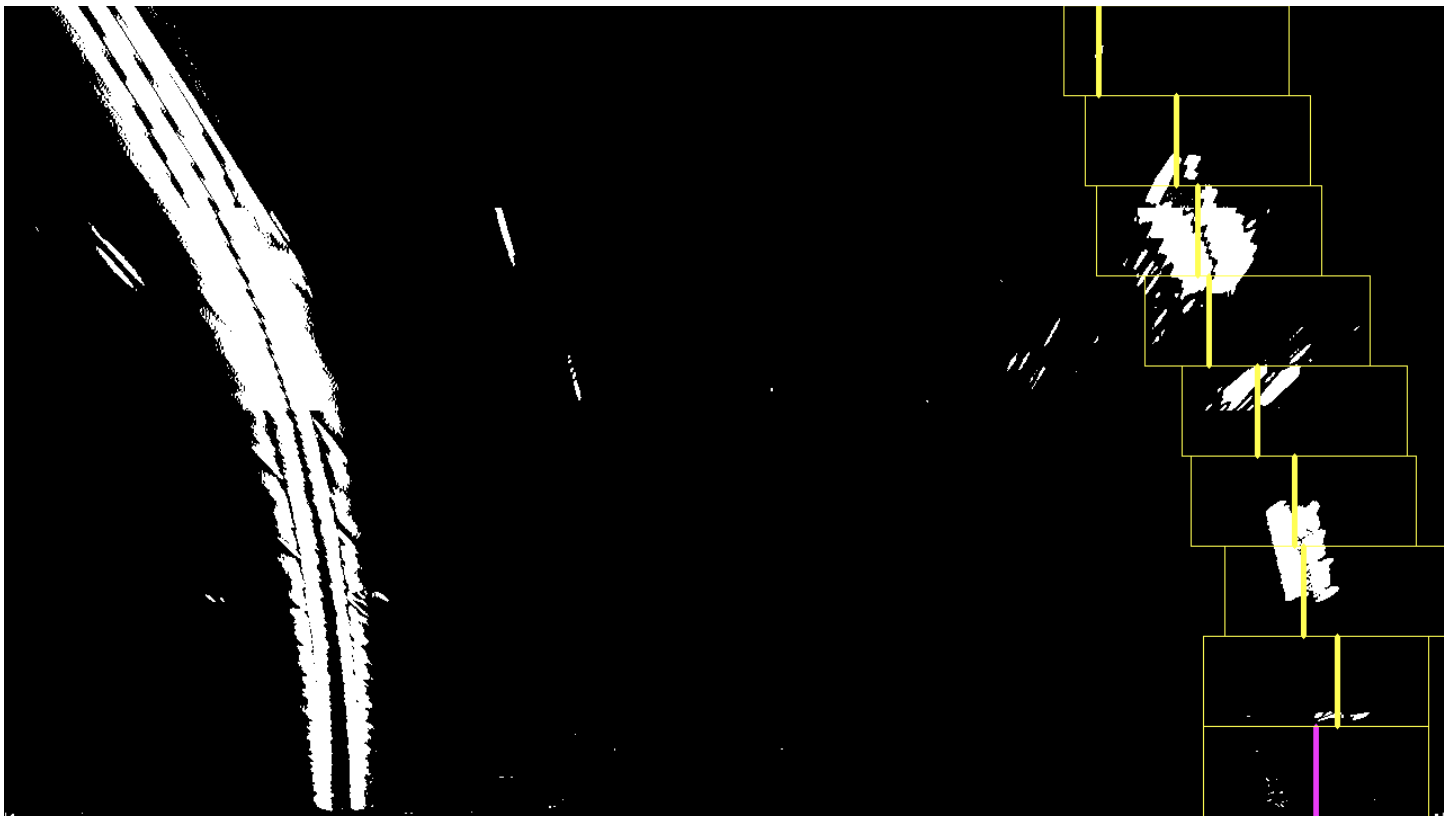
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

My lane finding code was executed independently for the right and left lanes. The code that detected lane points is inside pipeline.find_lines(). I started by finding the center of mass for the bottom corner of the screen. Next, I used the x coordinate for this center of mass as the starting point for a progressive upwards scan of the lane.

In the following images the yellow boxes represent a section of the image where I calculated a center of mass (thick yellow vertical line). After calculating center of mass, I started the next box centered on this location. The thick purple line shows the starting point.

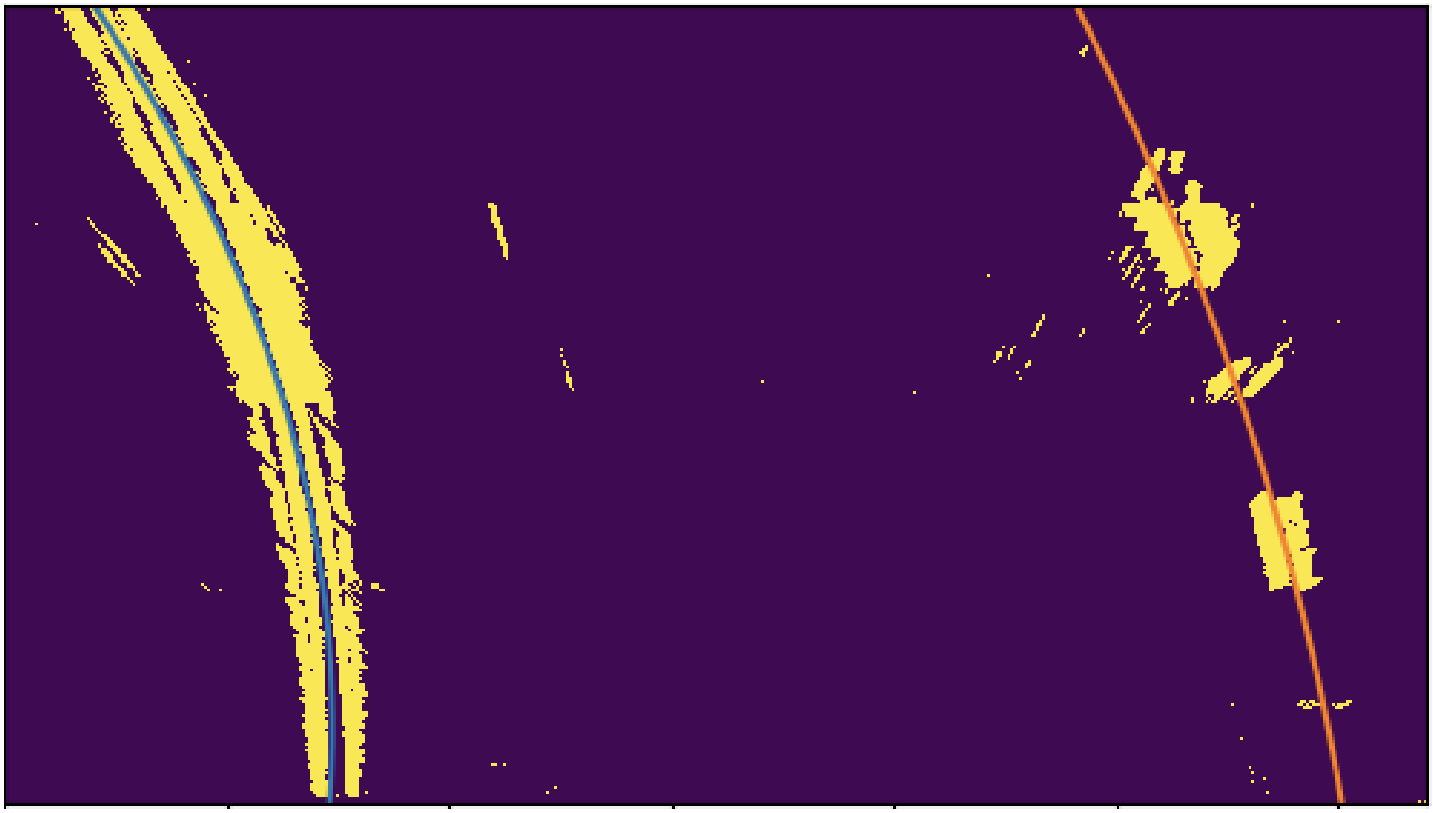


Progressive lane finding, for test image 2's left lane



Progressive lane finding, for test image 2's right lane

Next, I calculated these the polylines for these points using numpy's polyfit function.



Polyline's plotted using matplotlib

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 calculated this in calculate_curvature_meters() and calculate_offset().

To calculate the curvature of the road I transformed my line-paths from pixel coordinates to meters, polyfit that path to find a curve equation and then used that equation to calculated the curve.

For the offset, I calculate how far apart the center of the screen is from the center of the detected lines.

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

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?

When working on this pipeline the majority of my time was spent

- Futzing with the correct index ordering and building. Getting this right required getting better at persisting intermediate debug images
- Trying to get a perfect binary image. I eventually decided that it didn't need to be perfect. I proceeded with an imperfect binary image. For example, if I needed further improvement in right lane detection I could get additional accuracy by blending information from the left and previous frames. In the end this didn't prove to be necessary.

The pipeline is forgiving of most shadows and changes to road coloring. However, there are some conditions that can trip it up:

1. If a large portion of the lane line is occluding by another car while a car is changing lanes. The pipeline does not know to filter out the car.
2. Road patchwork with a vertical edge

Fixing (1) requires building awareness of occlusions in the pipeline. To fully address occlusion, memory of recent lane positions could be used to supplement the lane's we can currently see.