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.

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!

Camera Calibration

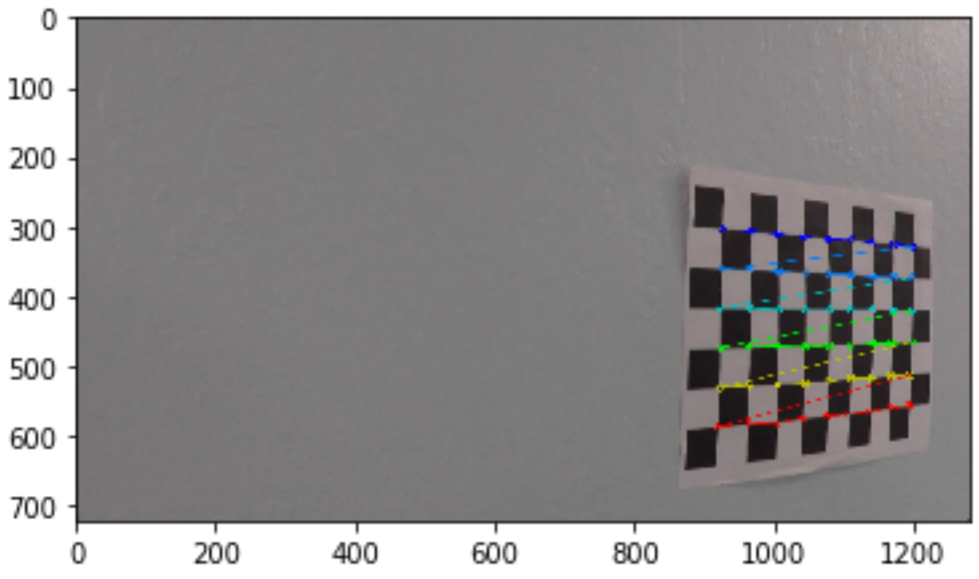
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this can be found in calibrate_camera() in calibrate.py, and is used in project.ipynb: "SECTION A".

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, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` 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 `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function.

To see that the calibration was done correctly, Section A shows a run of all the chessboard images with lines drawn traversing the diagonals of the internal rows of the chessboard. One such image is shown below as an example:



I applied this distortion correction to the test image using the `cv2.undistort()` function to all the images in the test images folder.

An example of before/after (see Section B in notebook for more) is shown here:

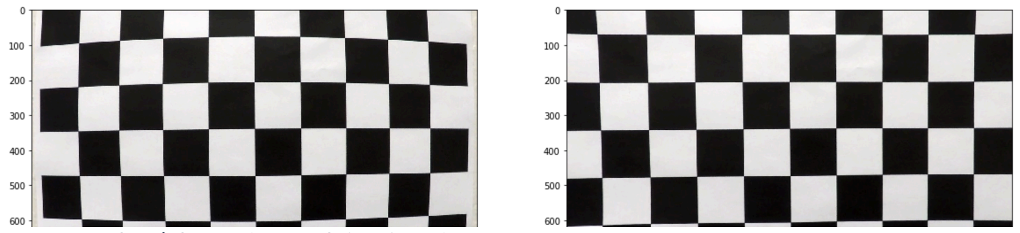


Figure 1 - Before/after undistort of chessboard

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:

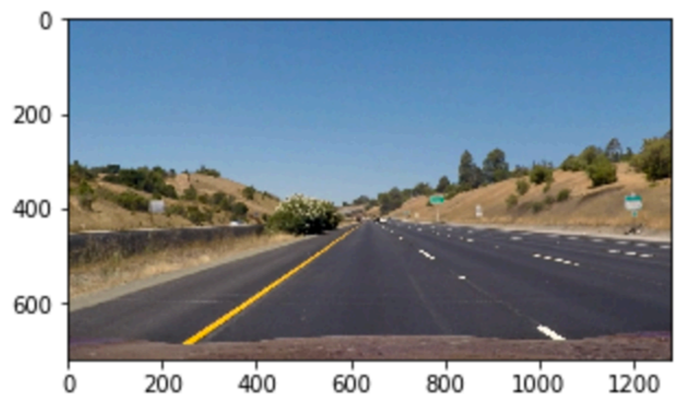


Figure 2A - Before Undistort

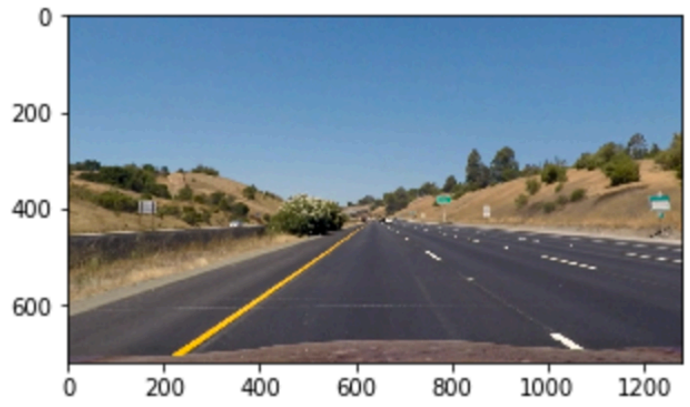


Figure 3B After undistort

This was done in Section C of the project.ipynb file. Essentially, for all images read in with cv2.imread(), cv2.undistort() was called using the mtx, distCoefs provided by the camera calibration step. The camera matrix was not altered for the output image from the input image.

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.

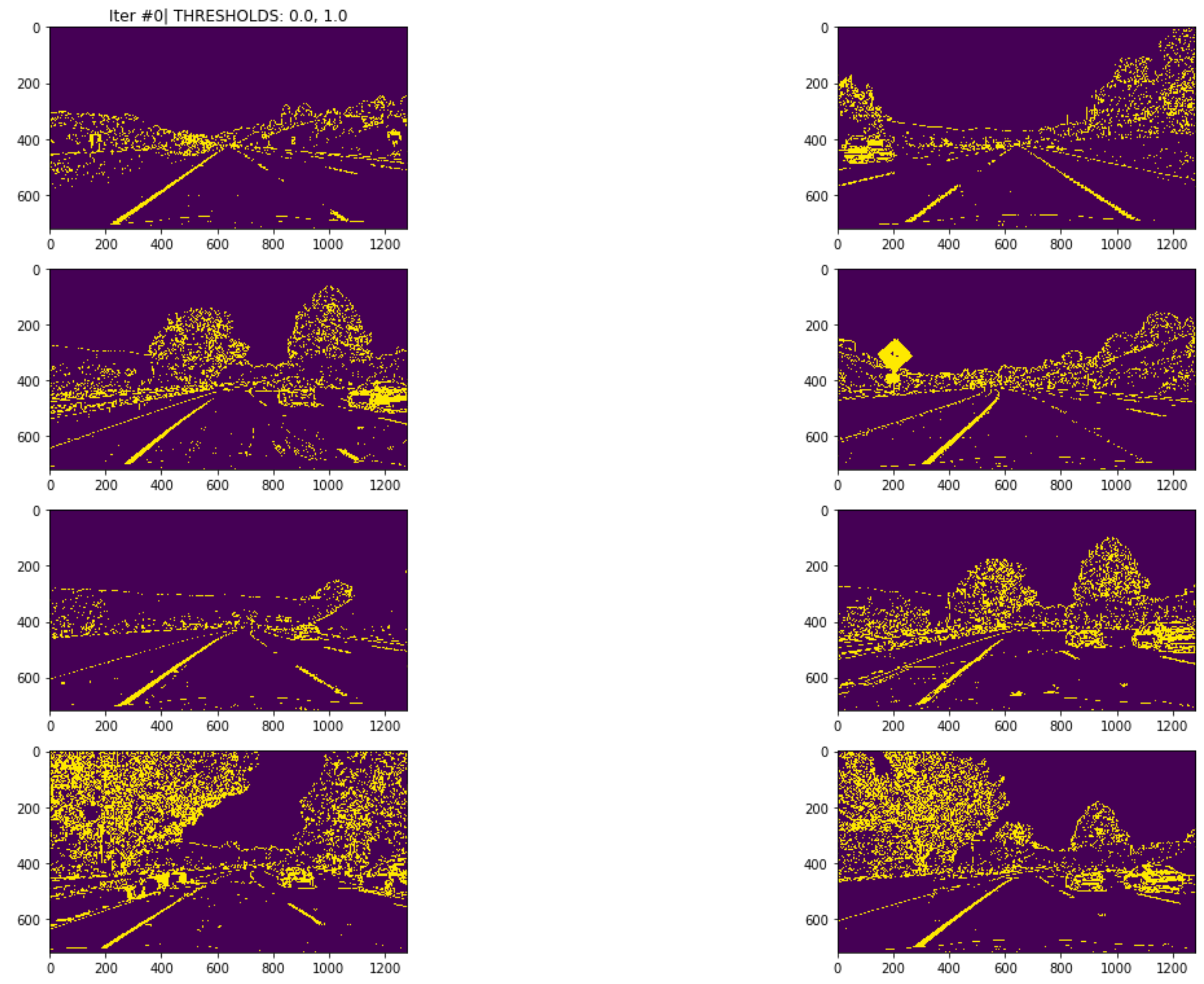
In "Section D" of project.ipynb, I perform an investigation of each of the individual thresholding transformation methods across a wide range of thresholds:

- Abs_sobel_thresh_x & y:
    - Plot all the test images with threshold ranges with a span of 30 in 30 point increments (i.e. 0-30, 30-60, etc.)
    - We find the optimum individual is:
        - Abs_sobel_thresh_x=(5,90)
        - Abs_sobel_thresh_x=(50,120)
- Similar for mag_thresh, except:
    - We tweak with parameters sobel_kernel = 3 & 5 as well
    - Optimal is mag_thresh=(60,120)
- For dir_threshold, we sample 20 slots across 0->pi/2
    - Optimal individual = 0.3*pi to 0.325*pi (i.e. 0.94 to 1.02)

- For extracting S channel, increments spanning 30 units also sweeped:
    - Optimal was 150-250 window

After this, we combined the pipeline together and perform a logical OR for all the bits: this process caused a further refining of the thresholds and can be seen in transforms.py:71

NOTE: dir_threshold was removed as the noise

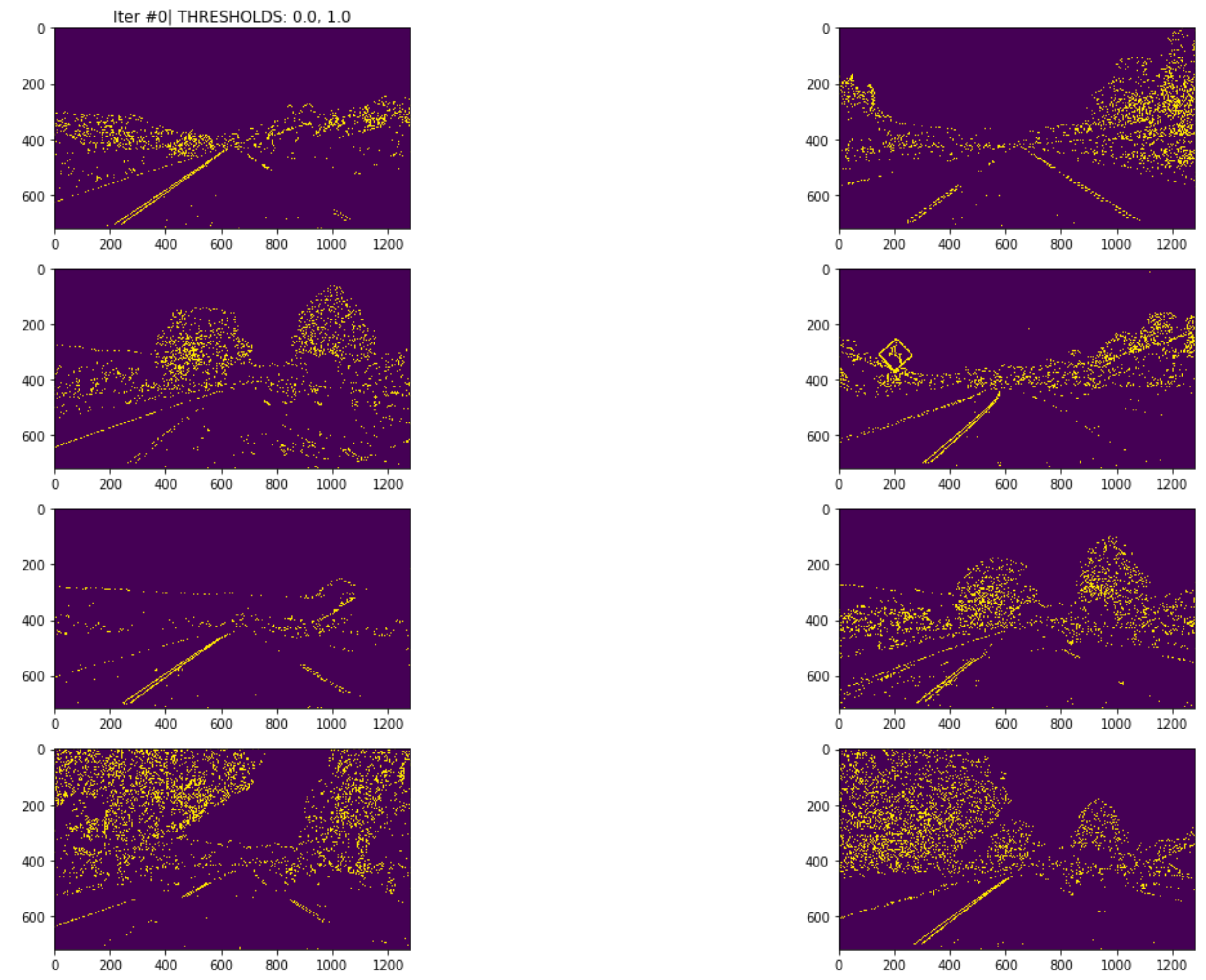


With this combination, it was found that detecting the lane lines and curvature of the road worked really well; however, occasionally it was sensitive to other lines on the road. This could have been salvaged with smoothing processing, though in the interests of time this was not performed.

Using some trial and error techniques, we found that a variation of the thresholds which was more resistant to other road lines (but had an inferior detection of curvature) was the following:

```
    gray = cv2.cvtColor(warped, cv2.COLOR_BGR2GRAY)
    gradx = abs_sobel_thresh(warped, orient='x', thresh_min=10, thresh_max=230)
    grady = abs_sobel_thresh(warped, orient='y', thresh_min=10, thresh_max=230)
    mag_binary = mag_thresh(warped, sobel_kernel=3, thresh_min=30, thresh_max=150)
    dir_binary = dir_threshold(warped, sobel_kernel=3, thresh_min=0.7, thresh_max=1.3)
    hls_binary = extract_s_channel(warped, thresh_min=80, thresh_max=255)
    combined = np.zeros_like(dir_binary)
    combined[((gradx == 1) & (hls_binary == 1)) | ((mag_binary == 1) & (dir_binary == 1))]
  = 1
```

This was used to create the output videos and showing the output of this pipeline are some sample images below:



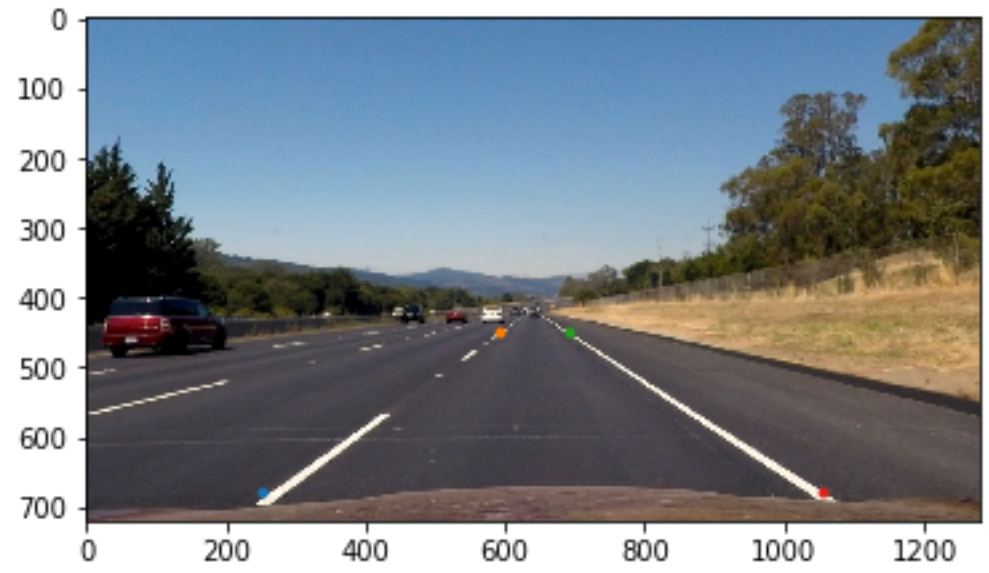
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The perspective transform is integrated as part of the video processing pipeline; in video.py, there are 2 calls to warpPerspective:

- One of the first few lines of def pipeline(img)
    - warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0]), flags=cv2.INTER_LINEAR)
- One of the final lines of the same function
    - unwarp = cv2.warpPerspective(warp_zero, Mi, (warp_zero.shape[1], warp_zero.shape[0]), flags=cv2.INTER_LINEAR)

These warp and unwarp the perspective to and from birdseye view respectively. We need to warp it to perspective view so that we can perform some polynomial fits and ascertain/visualize the radius of curvature; warping back provides the original image with our included annotations/processing enhancements.

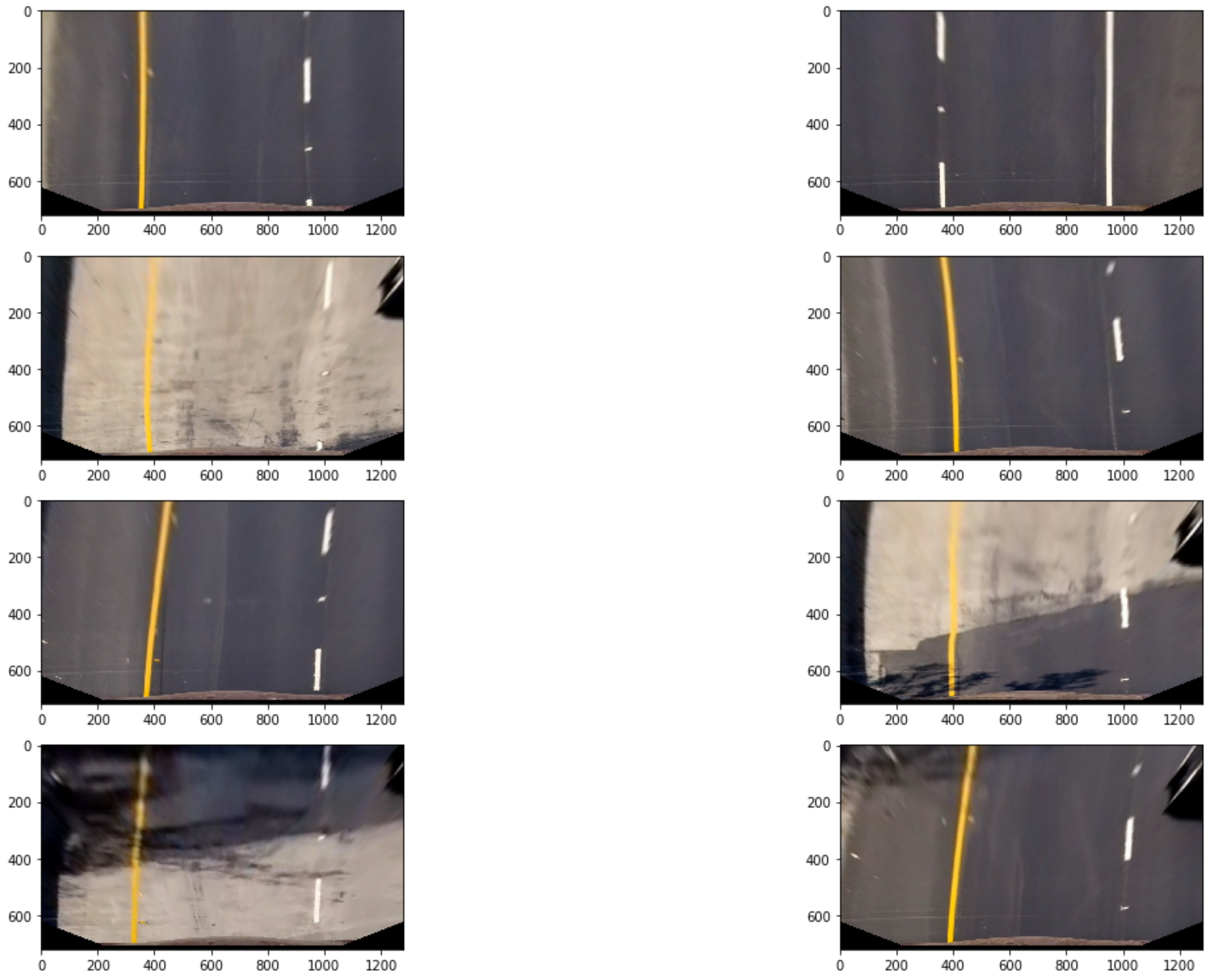
To calibrate the perspective transform, this was done in project.ipynb – Section E. Four points corresponding to the left and right road lanes of a sample image were plotted:



A rectangle was then selected mapping into the destination matrix:

Source	Destination
250, 680	320, 680
590, 450	320, -700
690, 450	955, -700
1055, 680	955, 680

This produced the following outcome:

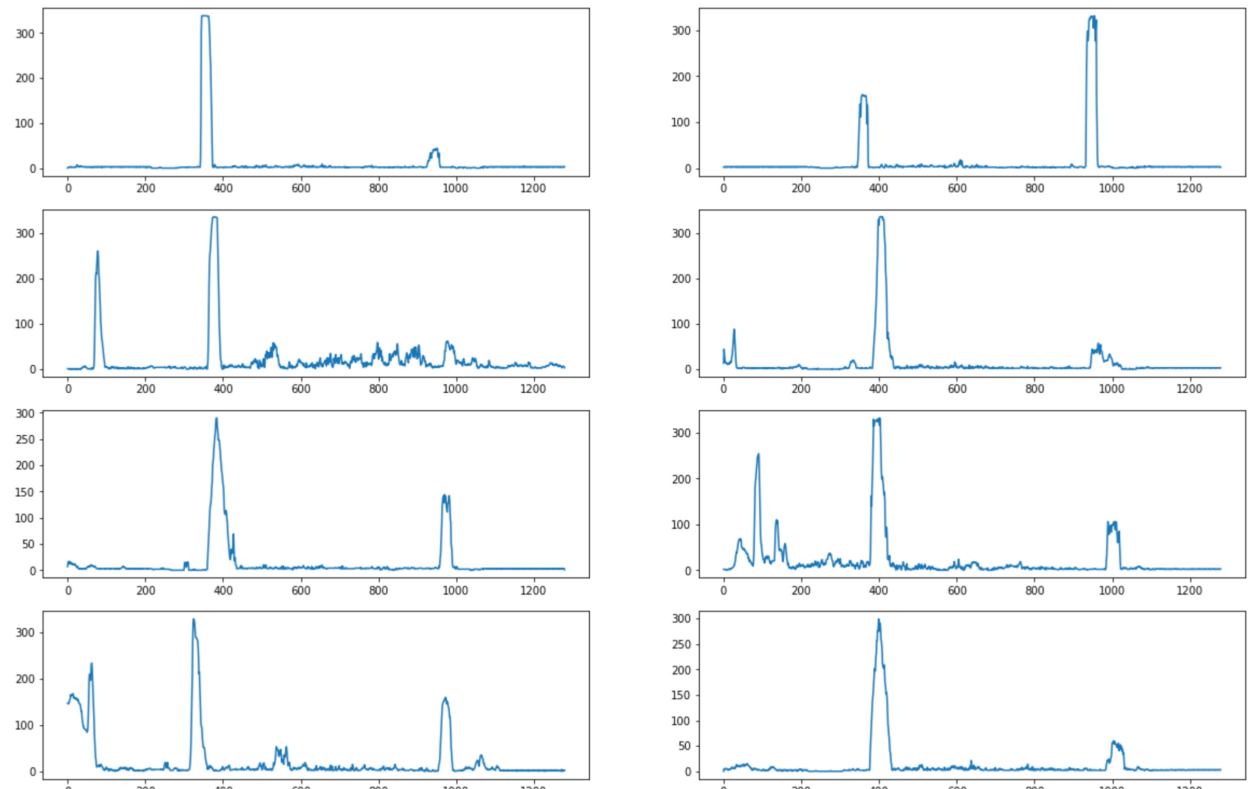


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

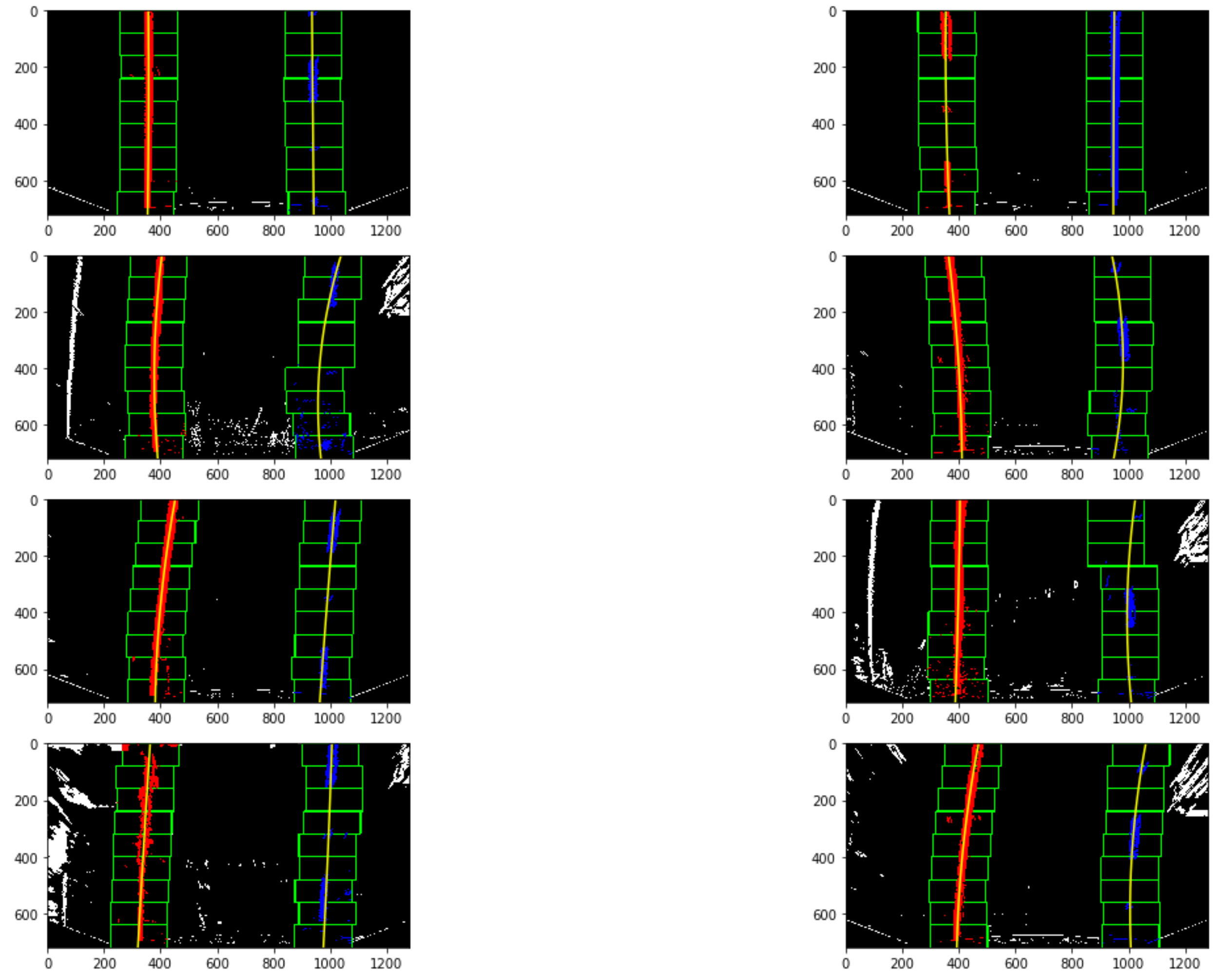
In Section F of project.ipynb, a call is made that tests the polynomial fitting function ('fit_poly'()). fit_poly can be found in findlines.py:53

The code (when an axis is provided for plotting) colors the left and right lane pixels red and blue respectively, draws the sliding windows (default=9) that aggregates the points that fit in a tolerable bounding box, and then fits a polynomial of best fit (2nd order) of these selected points.

The first bounding box is first initialized about the maximums (left and right of center) of the bottom half of the image (the following are plots based off the test image set after a set of gradient, color and perspective transformations):



The polynomial fit then can be visualized as such:



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.

In video.py:75, the polynomial (left & right) is evaluated across 3 points along the y-axis (spread across the entire image). Upon these evaluations, they are fed into a function called leastsq_circle from least_squares_circle.py, which is a scipy-based implementation of extracting the least squares fit for a circle given some input data. 3 points were selected rather than the full data set because we wanted to capture the full range of curvature of the lane and not find a circle fit that is biased because a polynomial has a non-constant radius of curvature along it. Fitting more than 3 points is likely to produce an underestimate.

The output of leastsq_circle gives the centerx, centery and radius values. Because curvature is so large, radius is approximately the same as centerx – so we use the centerx for finding steering angle as it also contains directional information.

Determining an accurate radius of curvature and position of vehicle requires that image scale is preserved throughout a number of transformations and image-to-pixels mapping is known.

Error could have been introduced easily in the perspective transform stage – ideally, we would have drawn a rectangle on the road of which we knew the exact aspect ratio, but instead we had to estimate this value.

Assuming the perspective transform is correct, then the meters to pixels ratio needs to be known. Because our perspective transform does NOT keep the x and y scaling the same, technically pixels-to-meters ratios for both dimensions.

The U.S. Interstate Highway System uses a 3.7m standard for lane width, so from inspecting the histograms above – Approx 620 pixels maps to 3.7 meters of width maps to around. This lends itself to the line on
video.py:107 ::

```
xm_per_pix = 3.7/620 # meteres per pixel in x dimension
```

For the y scaling, we have to rely on the visual integrity of the perspective transform; hence, ym_per_pix is set to be the same. In reality, this is highly unlikely to be correct, though.

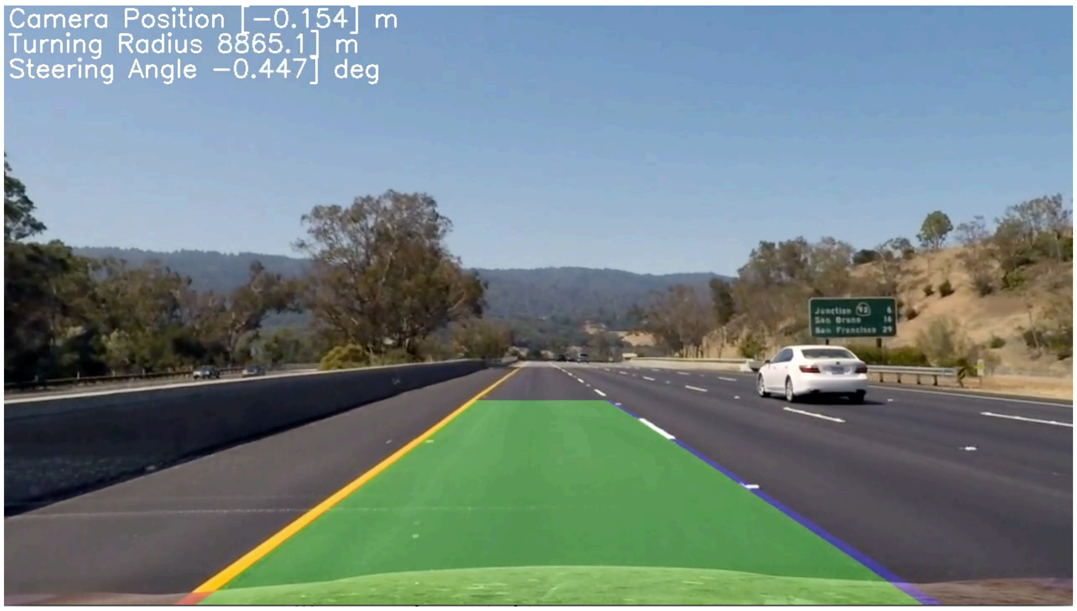
For the camera position: see video.py:100 (repeated below):

```
left_mean = np.mean(leftx)
right_mean = np.mean(rightx)
camera_pos = (combined.shape[1]/2)-np.mean([left_mean, right_mean])
```

camera_pos hence gets set as the number of pixels offset from the mean of the pixels defining the left and right lane lines. This gets scaled later into meters and output to the screen.

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

This is a screenshot of the advanced lane detector in operation:



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 with steering angle and turning radius

Here's a link to my video result for challenge video

Here's a link to my video result for harder challenge video

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?

Finding the lane lines in a consistent and smooth fashion was always going to be a difficult objective to implement, and the current implementation still leaves plenty of room of improvement to achieving this target. The groundwork was set for the thresholding transformations to clearly identify the lane lines, though smoothing techniques would have helped greatly such that other line markings on the ground or road dividers would not be mistaken for lane lines.

The pipeline also was slow to process – the videos taking approximately double the length of time to process than their runtime. This could be improved through reducing the image size and exploiting the fact that once the sliding window search for polynomial fitting was performed, future line fittings need only search within the vicinity of the previous fitted line. That being said, occasionally when the line search failed, the sliding window approach would still need to be invoked from time to time.

Also, the experimentation with thresholds could have been more systematic. If we knew the exact radius of curvature of road, for example, we could process the video amidst different combinations of curvature systematically and then compare the radius of curvature differences between predicted and actual.

Also, uphill/downhill videos will cause failure because the perspective transform assumes flat ground. There is potential to do different lane findings based on the known slope of the road, perhaps augmented with a gyroscope installed into the car.

2. Future Work

- Add smoothing techniques which can affect:
    - Sliding window search
    - Polynomial fitting – incorporating information from the previous x number of polynomial fits
    - Curvature and vehicle position estimates (moving average)
- Systematic/automatic tweaking of thresholds and processing pipeline
- More accurate calibration of perspective transform with properly annotated/acquired images for calibration
- Speed enhancements (e.g. resizing image size before processing)

- Experiment with doing radius of curvature fits across the entire set of data points instead of just extracting 3 points from a polynomial fit – and examining the bias it introduces and correcting for this bias.