Writeup Template – Advanced Lane Finding

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.

Writeup / README

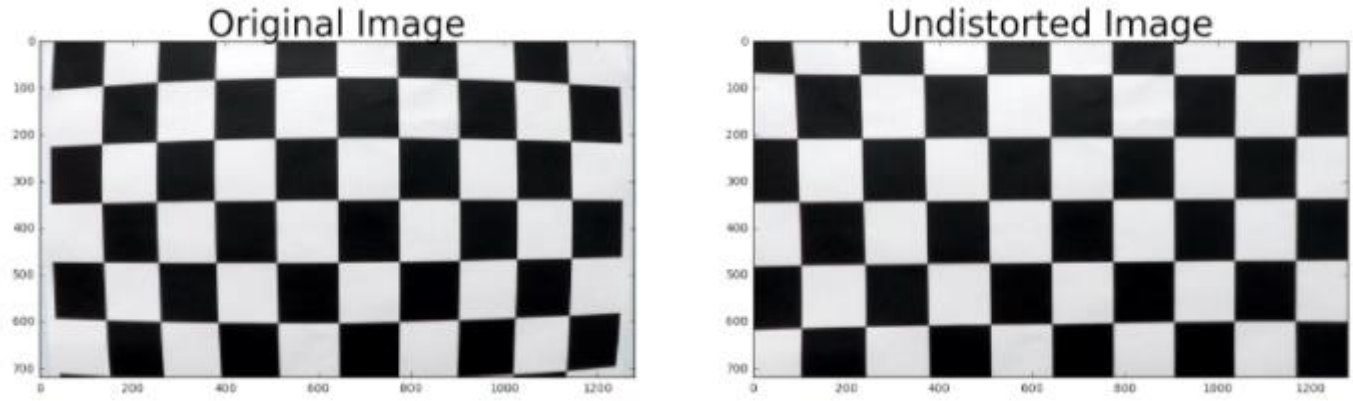
1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

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.

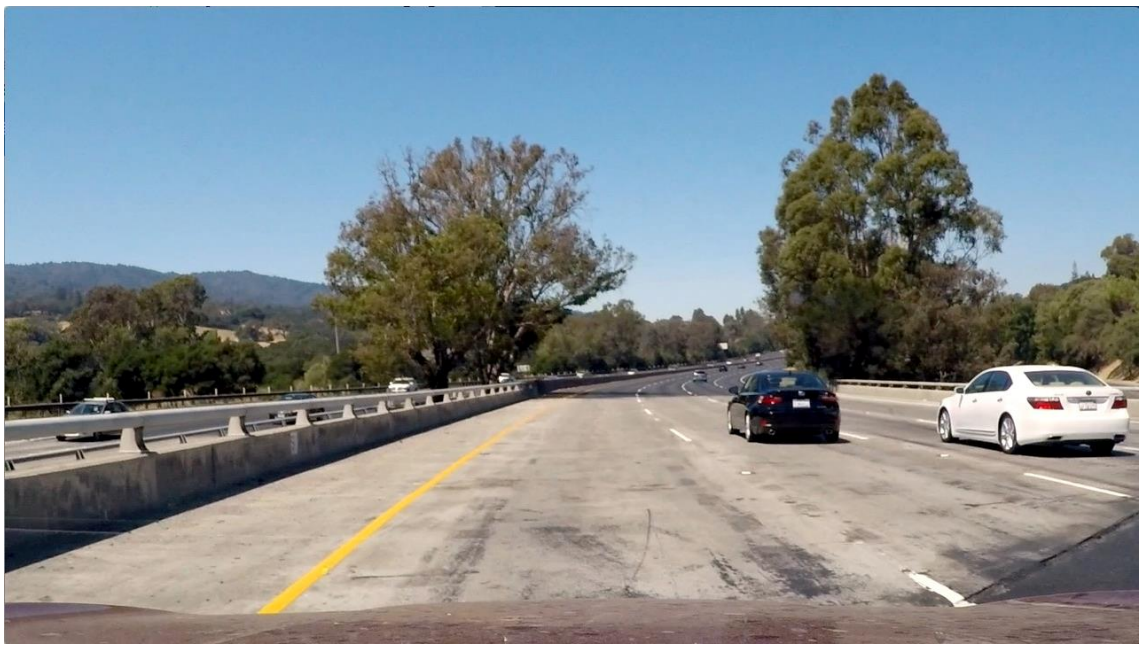
The code for this step is contained in the first code cell of the IPython notebook located in "./examples/example.ipynb" (or in lines # through # of the file called `some_file.py`).
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here 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. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



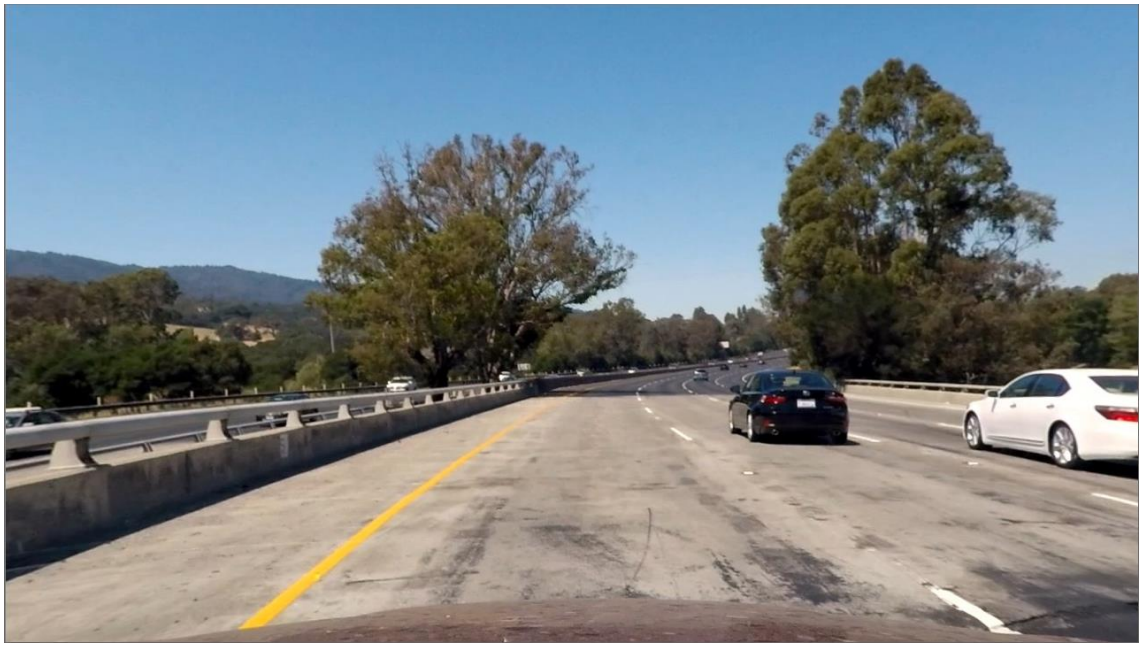
Pipeline (single images)

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:



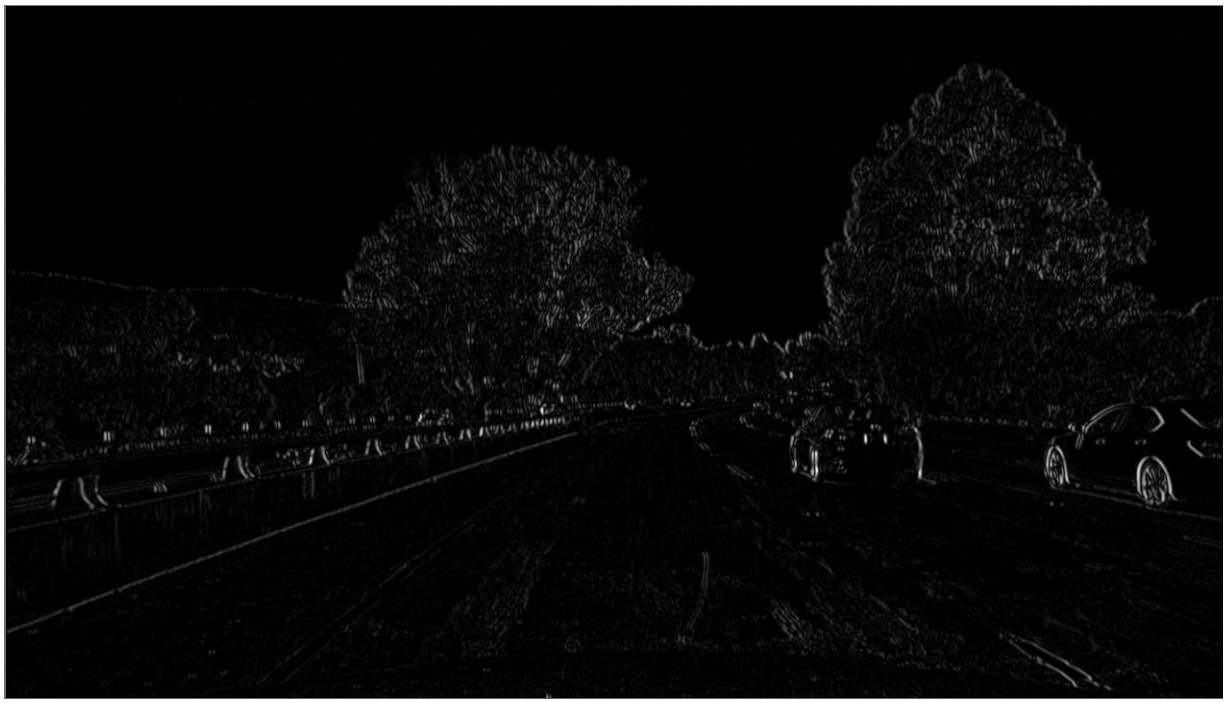
Without distortion



With distortion

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 color and gradient thresholds to generate a binary image (thresholding steps at lines # through # in `another_file.py`). Here's an example of my output for this step. (note: this is not actually from one of the test images)



Sobelx transformation

The same Sobelx image tresholded

```
gray_image = cv2.cvtColor(img_undist, cv2.COLOR_BGR2GRAY)

blur_gray = cv2.GaussianBlur(gray_image,(kernel_size, kernel_size), 0)

# Sobel x
sobelx = cv2.Sobel(blur_gray, cv2.CV_64F, 1, 0, ksize=3) # Take the derivative in x
abs_sobelx = np.absolute(sobelx) # Absolute x derivative to accentuate lines away from horizontal
scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))

# Threshold x gradient
thresh_min = 80
thresh_max = 250
sxbinary = np.zeros_like(scaled_sobel)
sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1
```

Here is an example of the code to do it so

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 `corners_unwrap()`. The `corners_unwrap ()` function takes as inputs an undistort image (`undist`).

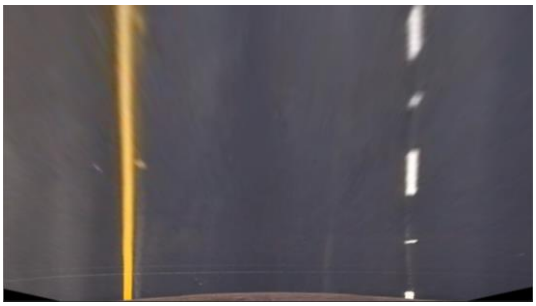
```
# Define a function that takes an image, number of x and y points,
# camera matrix and distortion coefficients
def corners_unwarp(nx, ny, undist):
    # Use the OpenCV undistort() function to remove distortion
#     undist = cv2.undistort(img, mtx, dist, None, mtx)

#     img = cv2.imread(image)
    # RGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # cv2.imshow('image',img)
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()

    offset = 300 # offset for dst points
    # Grab the image shape
    img_size = (undist.shape[1], undist.shape[0])

    # For source points I'm grabbing the outer four detected corners
    src = np.float32([(612, 440), (667, 440), (1100, 720), (210, 720)])
    # For destination points, I'm arbitrarily choosing some points to be
    # a nice fit for displaying our warped result
    # again, not exact, but close enough for our purposes
    dst = np.float32([[offset, -900], [img_size[0]-offset, -900],
                                      [img_size[0]-offset, img_size[1]],
                                      [offset, img_size[1]]])
    # Given src and dst points, calculate the perspective transform matrix
    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    # Warp the image using OpenCV warpPerspective()
    warped = cv2.warpPerspective(undist, M, img_size)

    # Return the resulting image and matrix
    return warped, Minv, undist
```

This resulted in the following source and destination points:

Source	Destination
612, 440	320, 0
667, 440	320, 720
1100, 720	960, 720
210, 720	960, 0

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.

Undistorted image with source points drawn

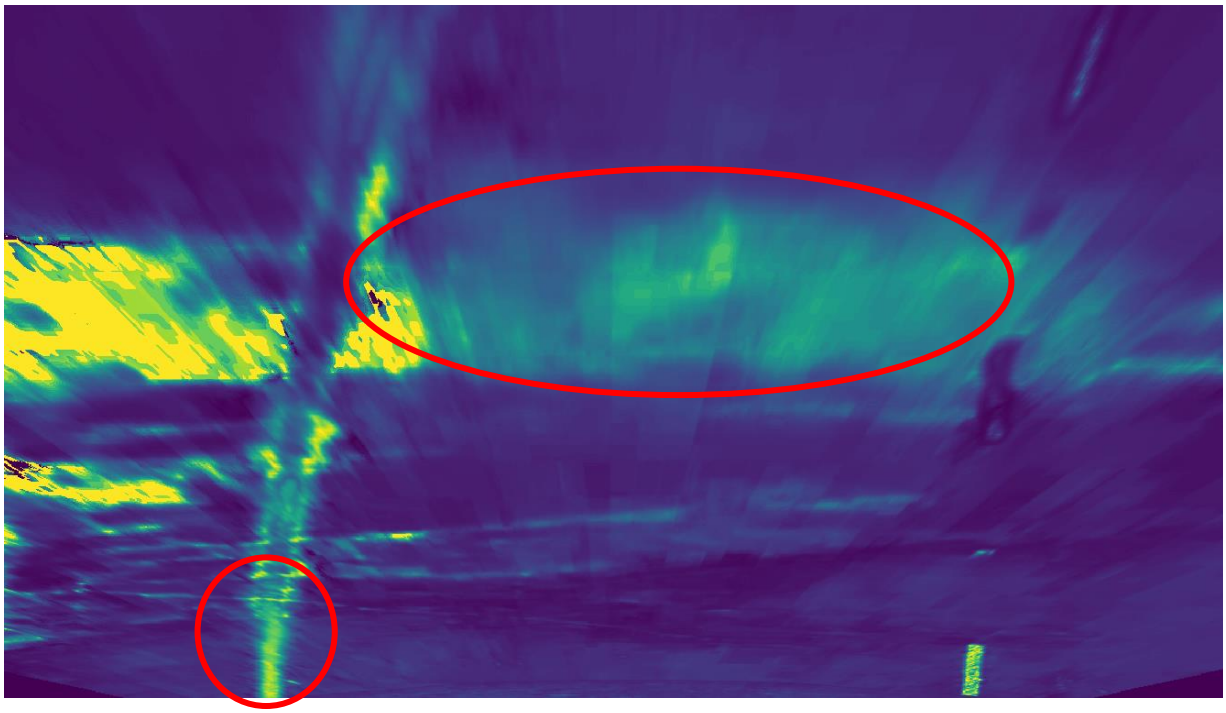


Warped result with dest. Points drawn

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

To identify lane line pixels, I had to pass the image through 3 different transformations and combine them. 2 transformations concerned colors and one concerned gradient transformation. First I had to find which color space was best representing the lane line for this video. I try some of them and finally chose the HSV for is good representative V channel. I combined HSV with a Sobelx transformation to clean to see all the interesting edge without the horizontal ones. From this point and by adjusting all the parameters (kernel_size, threshold, etc…) the result was quite satisfying but not perfect yet.

Indeed, some regions of the image were ambiguous to the V channel:



HSV, V_channel

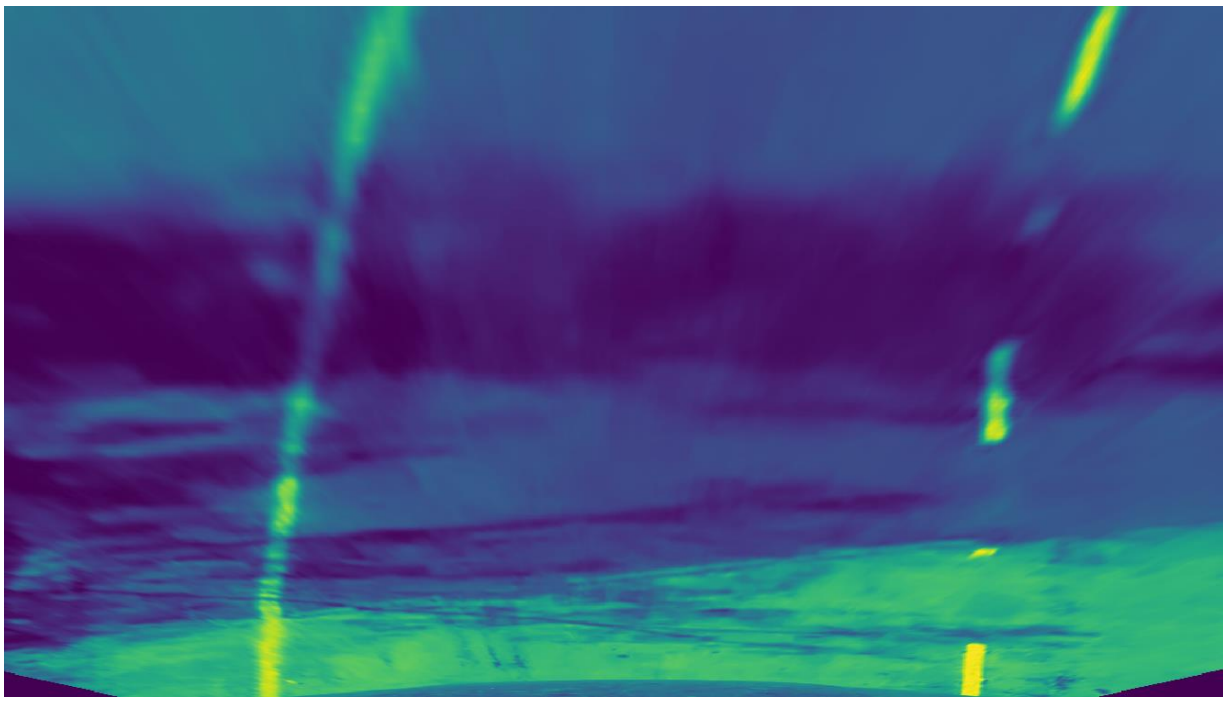
RGB color space

Disappointing binary result (not even good for Rorschach test)

As we can see on the image, yellow and black are sharing the same v_channel values whereas it's not the case in RGB. It was a huge problem because it was causing a lot of noise, and as a matter of fact it was difficult to identify the lane at this point.

My guess was to threshold the shadow with another channel from another color space. I took the r_channel of the RGB color space, picked the value of the shadows for this channel to create a threshold on it:

```
r_thresh_min = 0
r_thresh_max = 30
r_binary = np.ones_like(r_channel)
r_binary[(r_channel >= r_thresh_min) & (r_channel <= r_thresh_max)] = 0
r_binary2 = np.uint8(255*r_binary)

v_thresh_min = 110
v_thresh_max = 255
  v_binary = np.zeros_like(v_channel)
v_thresh= np.zeros_like(v_channel)
v_thresh[(v_channel >= v_thresh_min) & (v_channel <= v_thresh_max)] = 1
v_thresh2 = np.uint8(255*v_thresh)

v_sobelx = cv2.Sobel(v_thresh2, cv2.CV_64F, 1, 0, ksize=21) # Take the der
v_abs_sobelx = np.absolute(v_sobelx) # Absolute x derivative to accentuate
  abs_sobelx = np.absolute(abs_sobelx)
v_scaled_sobel = np.uint8(255*v_abs_sobelx/np.max(v_abs_sobelx))

combined_binary = np.zeros_like(sxbinary)
combined_binary[(v_thresh == 1) | (sxbinary == 1)] = 1
combined_binary[(r_binary == 0) & (combined_binary == 1)] = 0
combined_binary2 = np.uint8(255*combined_binary)
```

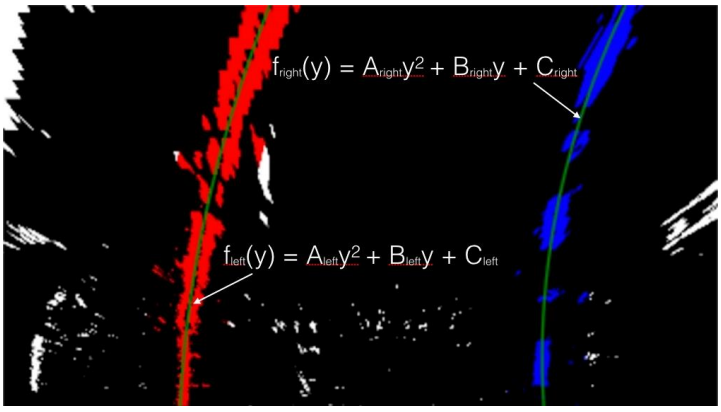


R_channel from RGB color space

R_channel binary thresolded range 0, 30

Final binary result

I took this red_channel thresholded binary and subtracted it to the combined sobelx and v_channel thresholded binaries and obtain what we can see above.



Fit principle

```
# Extract left and right line pixel positions
leftx = nonzerox[left_lane_inds]
lefty = nonzeroy[left_lane_inds]
rightx = nonzerox[right_lane_inds]
righty = nonzeroy[right_lane_inds]

# Fit a second order polynomial to each
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

We fit the x and y datas using the numpy function "polyfit"

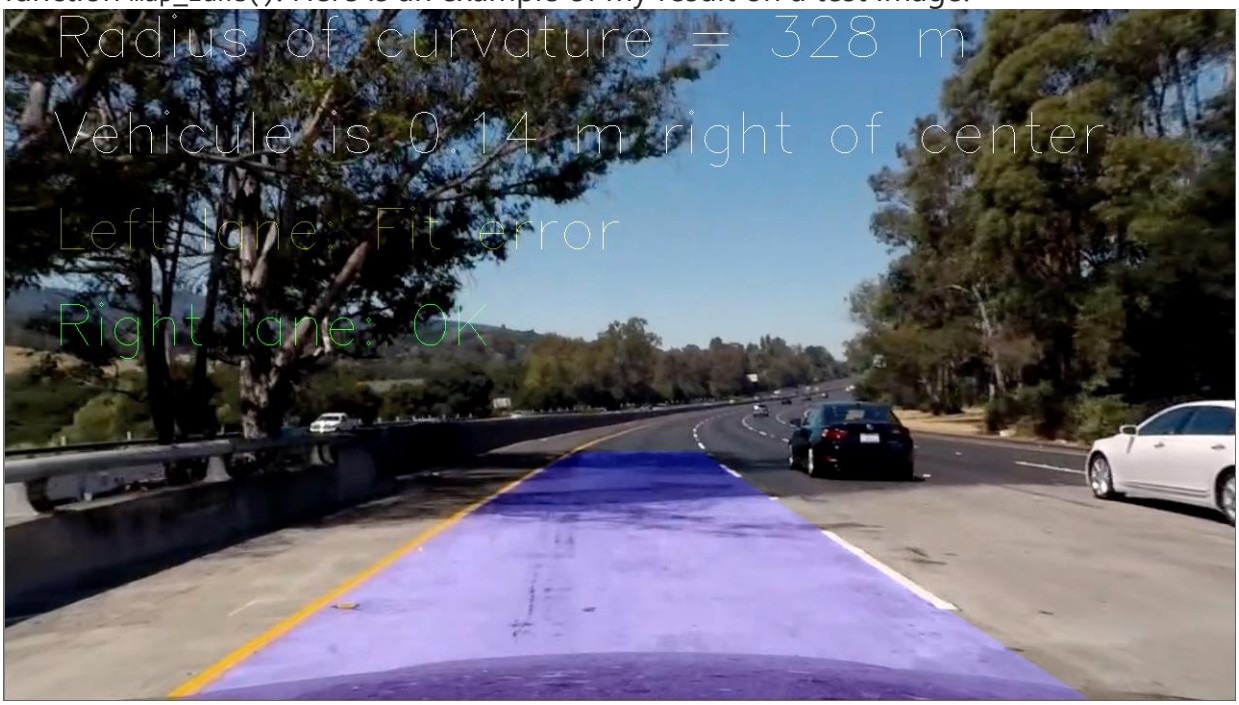
5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

```
def calculate_radius_distance(img_width, ploty, leftx, rightx, lefty, righty):
    y_eval = np.max(ploty)

    # Define conversions in x and y from pixels space to meters
    ym_per_pix = 30/720 # meters per pixel in y dimension
    xm_per_pix = 3.7/730 # meters per pixel in x dimension

    # Fit new polynomials to x,y in world space
    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)
    # Calculate the new radii of curvature
    left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])
    right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])
    # Now our radius of curvature is in meters

    # Calculate the distance between the center of the vehicule and the lane lines
    distance = img_width - rightx[0] - leftx[0]
    text_right = 'Vehicule is {:.2f} m right of center'.format(float(abs(distance)*xm_per_pix))
    text_left = 'Vehicule is {:.2f} m left of center'.format(float(abs(distance)*xm_per_pix))
    text_distance = text_right if distance >= 0 else text_left

    # Save the datas for both lane lines
    left_lane.radius_of_curvature.append(left_curverad)
    right_lane.radius_of_curvature.append(right_curverad)
    left_lane.line_base_pos = 3.7/2 + distance*xm_per_pix
    right_lane.line_base_pos = 3.7/2 + distance*xm_per_pix

    return left_curverad, right_curverad, text_distance
```

A formula was provided to transform the fit curve parameters to the radius of curvature. Then it was important to care about the conversion between pixels and meters to output the result in the good real world dimensions.

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 lines # through # in my code in `yet_another_file.py` in the function `map_lane()`. Here is an example of my result on a 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?

My pipeline first failed on the shadows. But for me the most important and challenging task is to find a solution that fits not only this video but the challenge and harder challenge video.

In the project video, I didn’t take advantages of the Hough transform neither the canny edges (not directly). However, I think that a we could use Hough transform not only to detect straight lines but to also detect curve lines with not a 2 dimensions’ space but maybe a 3 or more as we need more coefficient to characterize the curve. Unfortunately, such a function was not implemented in OpenCV and I didn’t have the time to look at it on the web.