AdvancedLaneFind_Submission

1. Submission packages

Code folder:

Lane_find_img.py: script to process sample images for lane finding, which automatically save processed image to specified folder.

pipe_line.py - pipe line to process video for lane finding

process.py - Defines the core functions to process images for lane finding

visualization_save.py - visualize and save image

configure.py- define basic input and output source/address of image process. It also includes the definition of perspective transformation source and destination points.

writeup_report - Explain what is included for the submission and how it is done.

output_images folder

bird_eye_view: bird eye view (top to down) view of sample images

camera_cal_output: camera calibration matrix and undistorted image sample

Feature_Selected_Image: images after features selected. The features include: s channel in HLS color space, gradient of image x, y direction, magnitude and direction of gradient

Poly_Fit: plot of polynomial fit of those points which are identified as lane points

Show_Lane_In_Image: overlay identified lane in original image (camera view)

Undistorted_Image: undistorted image for image samples

project_video_DetLane.mp4

project submission video which overlays identified lane with the original video

2. Go through rubric score

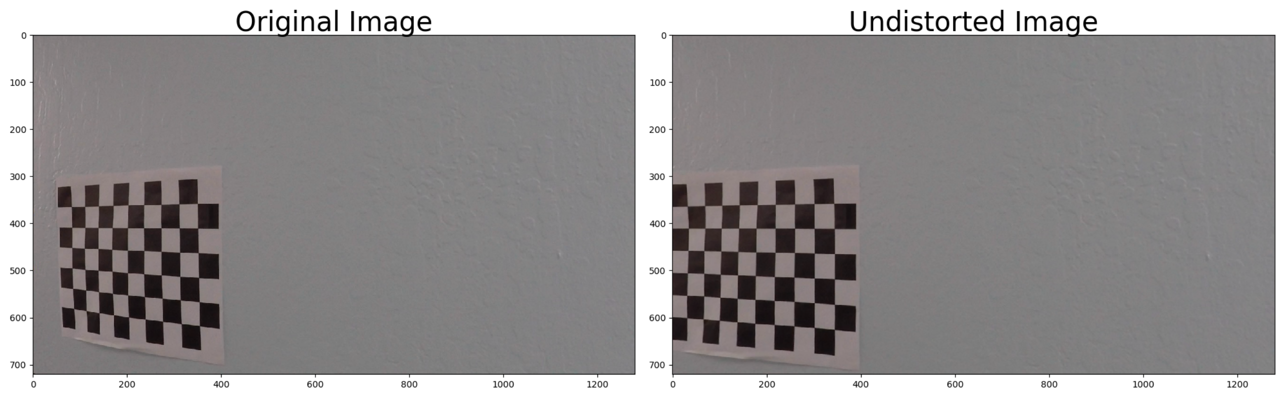
Camera Calibration

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 function of camera_cal() in the alt txt

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the camera_cal folder. Assume each chessboard is in a flat plane therefore z is set as 0. Then 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.

After obtaining the camera matrix from previous step, I then applied undistortion function cv2.undistort to undistort the raw image. Here is an example of before and after image undistortion operation:

For more samples, please find through Here

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.

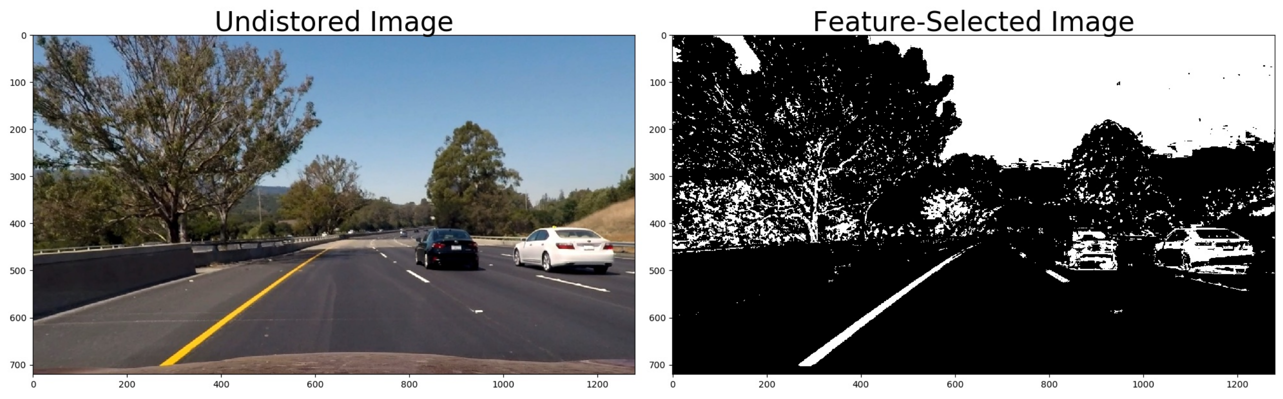
For color selection (see function 'hls_select' in process.py , I used the s channel of HLS color space by specifying threshold value thresh=(80, 255)

```
s_binary = hls_select(image_color, thresh=(80, 255))
```

For gradient selection, I used the x, y direction gradient (see function "abs_sobel_thresh" in process.py), magnitude of gradient (see "mag_thresh" in process.py), and direction of gradient (see function dir_threshold in process.py). Their threshold are shown as follows:

```
gradx = abs_sobel_thresh(image_color, orient='x', sobel_kernel=ksize, thresh=(50, 200))
grady = abs_sobel_thresh(image_color, orient='y', sobel_kernel=ksize, thresh=(50, 200))
mag_binary = mag_thresh(image_color, sobel_kernel=ksize, mag_thresh=(100, 255))
dir_binary = dir_threshold(image_color, sobel_kernel=ksize, thresh=(0.7, 1.2))
```

Here's an example of my output for this step.



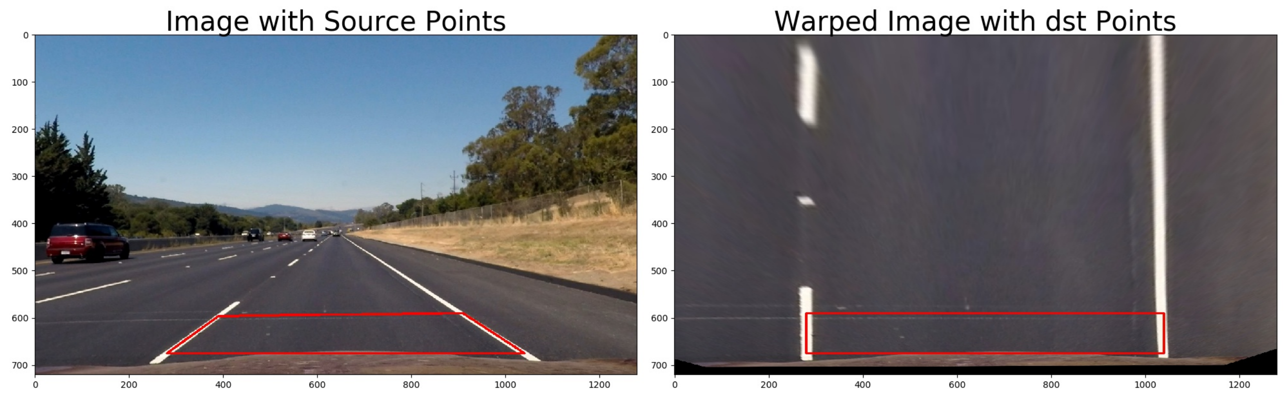
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform includes a function called `perspective_trans` defined in process.py. The `perspective_trans` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points by my visual check of the straight line shown in the test images:

This resulted in the following source and destination points:

Source	Destination
280, 675	280, 675
1040, 675	1040, 675
909, 590	1040, 590
390, 596	280, 590

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.

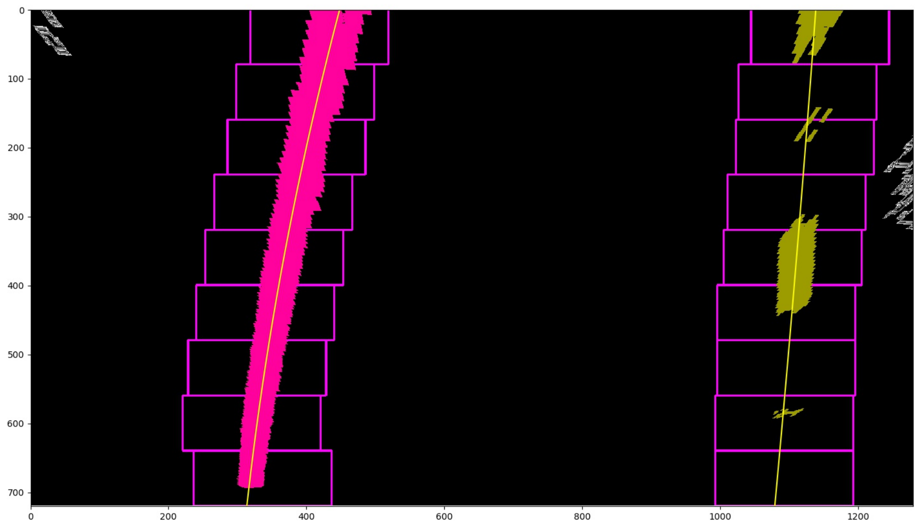


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

I used the method called 'sliding_window' which was introduced in the course material to identify the lane-line pixel. The function is included in process.py). Afer the lane points are available, they are used to fit a second order of polynomial:

```
left_fit = np.polyfit(lefty, leftx, 2) # polyfit coefficients of lane left edge
right_fit = np.polyfit(righty, rightx, 2) # polyfit coefficients of lane right edge
```

An sample image of lane points indentification and polynominal fit is shown:



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 radius of curvature of the lane is calcuated through function 'curvarad' in process.py. the position of the vehicle with respect to center is calcuated through function 'lane_center_offset' in process.py. The assumption here is the camera is mounted in the center fron vehicle. The lane center is identified as the middle of left and right line of a lane.

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

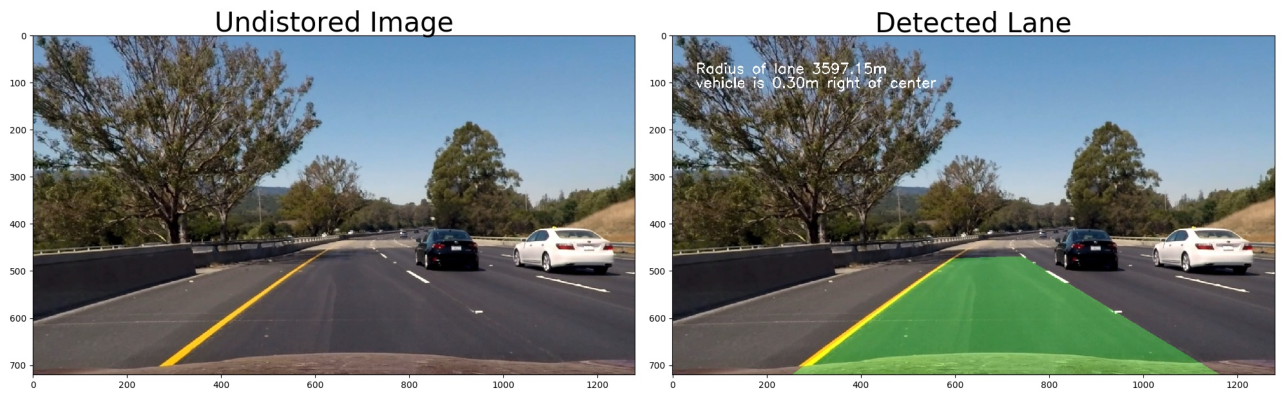
I implemented image overlay by mapping the lane identified from bird eye view to the original image through inverse perspective transformation. I also added radius of curvature and distance of vehicle off to lane center into orginal image through text overlay. The function is read as:

```
result = image_overlay(image_color, image_warped, left_fitx, right_fitx) # image_warped

# text overlay
result = txt_overlay(result, left_curverad, right_curverad,middle_curverad, vehicle_offcenter)
```

The functions of "image_overlay" and "txt_overlay" can be found in alt txt.

Here is an example:



For more image, please go to Here

6 Pipeline (video)

Here's a link to my video result: link

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?

The pipeline works well for most of images in the vedio except a few of them. There are several causes for the error detection

```
1. background noises: a. uneven shadow area where the tree shadow to the ground is uneven and changes dynam

2. insufficient or missing lane points which causes the polynomial fit incorrect

3. curvature change: this particularly is an issue for the challenge video where the curvature of lane chan
```

To isolate error detections, i added sanity check function (please refer to the sanity_check in alt txt) to check if the polynomial fit makes sense. Several criteria were considered:

```
1. radius of curvature: compare the radius of recently identified lane lines with the one in previous step.

2. Absolute radius of curvature: this is a little tricky. In general situation, we do not know what's the r

3. compare the shift between the new identified lane lines with the lane center identified previously. The
```

Sanity check function returns the True or False of left and right line detection.

In pipe_line.py alt txt, I defines handling methods for different types of line detection errors. If both lines are faulted, keep previous lines; if one of lines are faulted, use another line plus a lane width shift. Here is the code extracted from pipe_line.py:

```
    if (san_check_left & (not san_check_right)):
        right_fit = 0*right_line.current_fit + 1*(left_fit + np.array([0, 0, fit_offset]))

    elif (not san_check_left) and san_check_right:
        left_fit = 0*left_line.current_fit + 1*(right_fit - np.array([0, 0, fit_offset]))

    elif (not san_check_left) and (not san_check_right):
        right_fit = right_line.current_fit
        left_fit = left_line.current_fit
    else:
        pass
```

Even though i spent a lot of time to design and debug the strategy for robustness, in general, more work is needed to improve the detection quality, which is out of the scope of this submission.