Advanced Lane Detecion Project

Mingrui Wu

1 Camera Calibration

The colde for this step is in py/camera_calibration.py. Camera calibration is done by following the materials taught in the class. Twenty chess board images are provided in the camera_cal folder.

1. For each image, find the inner corners by $cv2.findChessboardCorners$. For this function the number of corners in x and y direstions are 9 and 6 respectively.

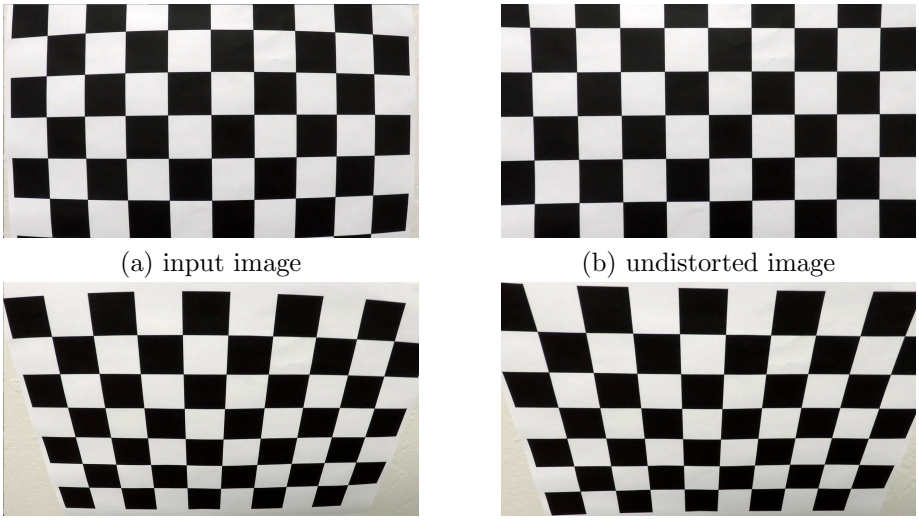
2. Append the corners found in the previous step to an array $imgpoints$, at the same time append pre-calculated mesh grid points to another array $objp$.

3. Having scanned all the twenty images, apply $cv2.calibrateCamera$ to calcuate the calibration paramters.

4. For each of the twenty images, apply $cv2.undistort$ to correction the distortion, where the calibration parameters needed by this function are computed in the last step.

An example of camera calibration is given in image 1. The two images on the left column are the inpute images, while the two on the right column are the images after distortion correction.

The undisorted images for all the twenty images provided in the camera_cal folder can be found in camera_cal/calib_result folder.



(a) input image (b) undistorted image

(c) input image (d) undistored image

Figure 1: Camera calibration

2 Pipline

In my code, LaneDetector is the class for performing lane detection. The code is in py/lane_detector.py. Within this class, the member function LaneDetector.lane_detect describes the whole pipeline.

Take the input image Fig 2a as an example, the whole piple line consists of the following steps:

1. **Loading pre-computed parameters.** First, an object of the LandDetectParam class is created. And this object contains all the parameters needed for the whole lane detection pipeline. In particular, it will load the pre-calculated paramters for camera calibration and perspective transform. The parameters are stored in the file lane_detect_param.pkl. The code for this part is in py/lane_detect_param.py

2. **Distortion correction.** Using the pre-calculated camera calibration parameters mentioned in the last section, the first step is to apply $cv2.undistort$ to the input image to get the undistorted image. The undistorted result for 2a is given in 2b

3. **Binary thresholding** BinaryThresholder is the class for this step. The code is in py/binar_thresholding.py. Following the class materials,

this done by combining the result from gradient and color transformation.

(a) **Sobel gradient** The input image is first transformed to a gray scale image. Then the Sobel operator is applied to the gray scale image to obtain the gradient image. Then based on the gradient strengh in x and y directions, the magnitude and the direction of the gradient at each pixel, binary thresholding is performed.

(b) **S space** The input image is transformed into HLS space first. Then binary thresholding is performed based on the S-space value.

(c) **Combination** The two binary images obtained from the last two steps are combined by OR operation. In addition, the pixels whose S-space value is lower than a threshold are filtered out.

(d) **RGB filtering** If the image obtained in the last step still have many pixels (more than 40%), then the pixels whose RGB value is lower than a threshold are further filtered out. This step is helpful to remove some edges caused by shadows.

The binarization example is given in 2c

4. **Perspective transform.**

   (a) **Pre-calculation.** Parameters for perspective transform are precalcuated. The code is in py/camera_calibration.py. Four source points and destinations points are selected manually. Then the matrices for both perspective and inverse perspective transform are obtained by applying the function $cv2.getPerspectiveTransform$. This is done by applying $cv2.warpPerspective$ function.

   (b) **Perspective transform.** The function $cv2.warpPerspective$ is applied to the binary image obtained in the binary thresholding step.

   The wapred image is given in 2d.

5. **Polynomial fit.** Having obtained the warped image in the last step, PolyFitter is the class for identifying lane pixels and curve fitting. The code is in py/poly_fitter.py. There are two sub steps:

(a) **Lane pixel identifcation** When processing an individual image or the first frame of a video, the approach based on histogram provided in the class material is applied. First the left and right starting points are selected based on the maximum column sum of the left half and right half of the binary image. Then for the left and the right lanes, we search along the y direction for the non-zero pixels around the current columns. And the staring points are updated based on the mean position of the non-zero pixels that have been found. This is coded in the function PolyFitter.find_lane_pixels. When processing videos, the lane pixels are searched along the polynomial curves obtained from the previous frame. The code for this in the function PolyFitter.search_around_poly.

(b) **Poly fit.** Having obtained the lane pixels, numpy.polyfit is applied to fit a quadratic curve. Here an important approach is to enforce the **consistency** between two consequtive frames. If the current fit is too different from the previous frame, then current fit is ignored. Otherwise, a weighted average of the previous and current fit is computed as the latest polynomial fit. The code for this in PolyFitter.update_fit.

The fitted polynomial curves are given in the image 2e.

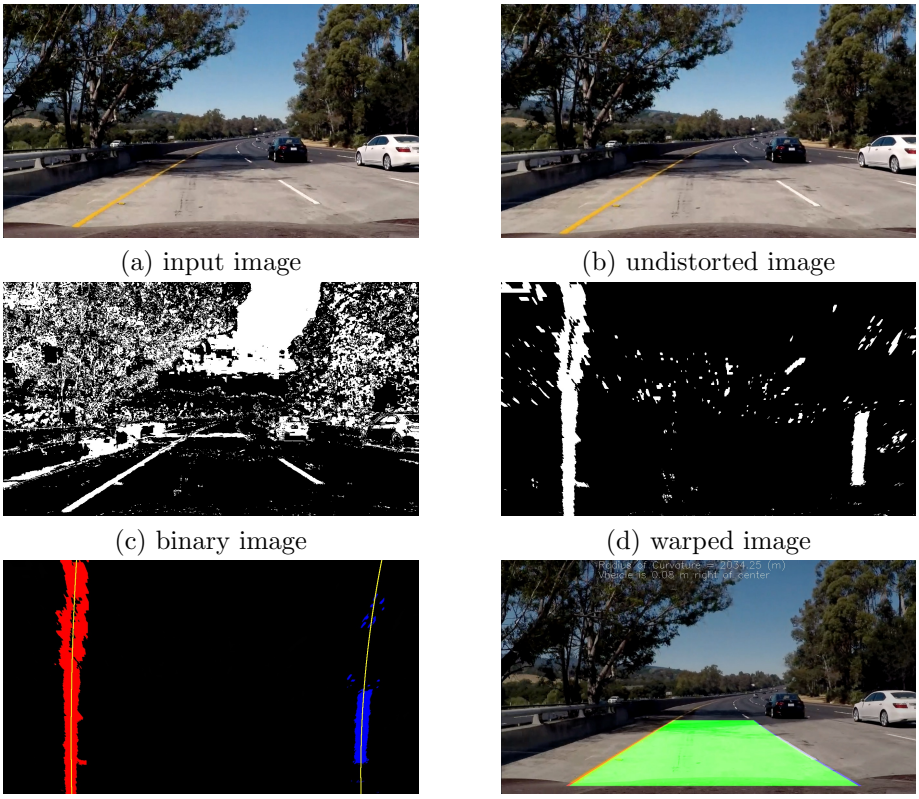
6. **Calculate radius of curvature and the position of the vehicle.**

(a) **Computing the radius of curature.** The radius of curvature of the left and the right lanes can be calculated once we have the parmaters of the fitted polynomial curves. But here the curves are fitted by the salced the coordinates of the lane pixels. The funtion is PolyFitter.meansure_curvature_real. The mean of these two radius of curvature is returned as the raidus of curvature of the road.

(b) **Computing the position of the vehicle.** Based on the two fitted polynomail curves, we can get the x coordinates of the two points corresponding to the maximum y value of the image. Then the mean value of these two x coordinates is the estimation of the vehicle's position. Note that here we need to scale the result to

reflect the real distance rather than the number of pixels. The code is given in LaneDetector.compute_vehicle_pos.

7. **Render the lane image**. Having obtained all the information in the previous steps, the lane image can be rendered by filling the regions surronded by the lane points. An example of the lane image is given in 2f.



(a) input image (b) undistorted image

(c) binary image (d) warped image

(e) poly image (f) lane image

Figure 2: Lane detection pipeline

3 Resuls

3.1 Results on test images

The lane detection results for the images in the folder test_images are given in the folder output_images. Each test image has 5 result images: undistortion, binary thresholding, warp, poly fit and lane detection. The title of the 5 images start with undist, bin, warp, fit and lane respectively.

3.2 Results on videos

The pipeline described above works well on project_video.mp4. The result is ld_project_video.mp4 in the github.

The pipeline is also tested on chanllenge_video.mp4. It also works although there are some small wobbles. The result is ld_challenge_video.mp4.

4 Discussion

The above pipeline works well on project_video.mp4 and chanllenge_video.mp4. In order to overcome the drastic changes in the curve fitting between two consecutive frames, which can lead to the failure of the lane detection, I enforce the consistency of the cure fit between two consecutive frames. This can smooth the estimation and prevent the fitted curve from changing drastically.

However, this cannot work in harder_challenge_video.mp4 because the lane curves change much more than the other two videos. So a better trade off and a better fusion of the vairous information (colore, gradient, cure fit, etc) to detect lane pixels are needed.