Advanced Lane Finding Project (SDC Project 4)

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.

Software Modules

The major implementation of this project is alld module which is locale under alld folder:

1. alld

alld is advanced lane line detection implementation for this project

1.1 alld.cam.CameraAuxiliary

Camera calibration relative function's implementation

1.2 alld.image_filter.EdgeDetector

High level edge detection algorithm implementation

1.3 alld.image_filter.GradientFilter

Low level edge detection algorithm

1.4 alld.trans.PerspectiveTrans

Implement the perspective transformation

1.5 alld.road_mgr.LaneLine

Imp#lement lane line fitting algorithm and information

1.6 alld.road_mgr.RoadManager

Management road information and the hight level lane lines detection function.

1.7 alld.viz_util.VisualUtil

All the required visualization functions are implemented in this class

2. sdcp4

Except alld, there is a file sdcp4.py implements the pipeline to generate final output image

```
usage: python sdcp4.py [-f filename] [-d]

optional arguments:

-f  input video file name, defult is project_video.mp4

-d  use diagnostic pipeline
```

2.1 sdcp4.pipeline

Providing a pipeline to handle whole process from camera data to lane detected image.

2.2 sdcp4.pipeline_diag

Generating images which combine more images in pipeline stage for diagnostic.

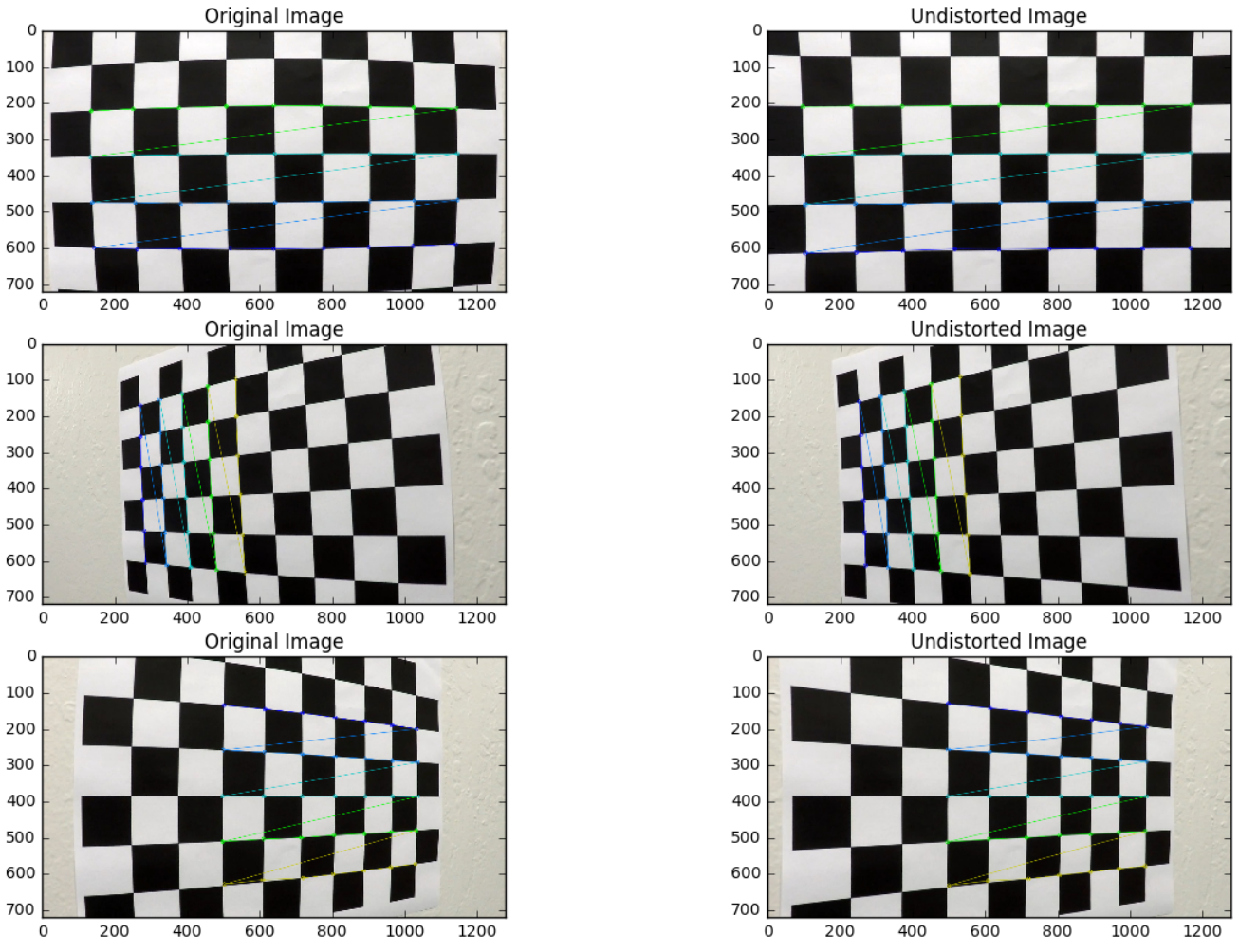
3. misc

3.1 adv_lane_line_det.ipynb

This is the major file to generate all images which are used in README.md

Camera Calibration

Because camera captured images may have lens distortion. Before going to process image, the first step is camera calibration. Camera calibration is required many chessboard images to calculate transform matrix and distortion by image points and object points.Once the image points of chessboard corners have been detected, we could generate object points. After all the image points and object points have been collected, it is ready to use `cv2.calibrateCamera()` to get transform matrix and distortion. In the project, there are 20 chessboard images locale at `camera_cal` folder. Almost images could be fitted by object points with corner counts: (9,6). There three images: calibration4.jpg (6,5), calibration5.jpg (7,5), calibration1.jpg(9,4) have different corner counts. I implemented a simple search procedure at `alld.cam.CameraAuxiliary::_gen_cb_corners()` The major code is implemented at `alld.cam.CameraAuxiliary::fit()` The code `sdcp4.py calbrate_camera()` presents how to use `fit()` function. For further using, the calibration data could be stored by `save()` with specific file name. (default is cam_cal.p). The calibrated sample images can be generate by `sdcp4.py draw_image_calibration()` :



Pipeline (single images)

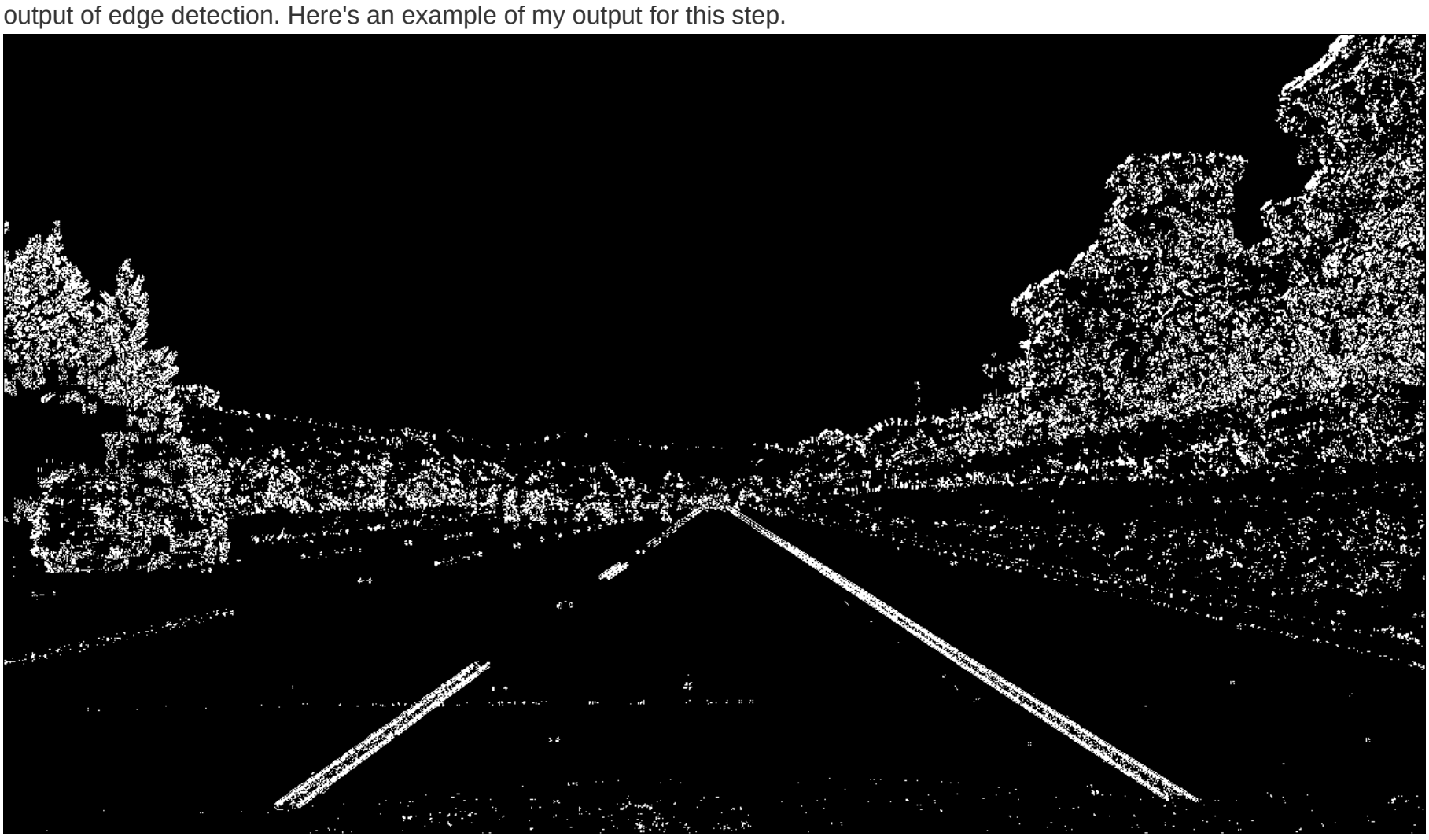
The pipeline is consist of 5 steps: 1. distortion correction, 2. edge detection, 3. perspective transform, 4. lane area detection, 5. final output. `sdcp4.py draw_pipeline()` is presenting the whole procedure. The details are listed as the following: (original image is './test_images/test4.jpg')

1. Distortion Correction

Once we have transform matrix and distortion of camera, we could apply it on new images by `cv2.undistort()` to calibrate images. In previous stage, we saved the matrix and distortion into file. Now, we can use the data to undistort image through `alld.cam.CameraAuxiliary::calibrate()` . The following is undistorted image:

2. Edge Detection

Edge detection is not a trivial task. There are 4 kinds of gradient of sobel filter could be used, threshold hold is required to be decided and there are many color space are available to be considered. I implemented a warper class for sobel edge detector: `alld.image_filter.GradientFilter` and the major edge detection logics are implemented at `alld.image_filter.EdgeDetector`. After a lot of trials on combination, I used a combination of color and gradient thresholds to generate a binary image. The function is `alld.image_filter.EdgeDetector::detect_edge_complex_3()` that uses two channels to get the final combination. The first one is gray channel for the (magnitude & directional) gradient. The second one is saturation channel of HLS, I used the ((x | y | magnitude) & directional) combination. Then OR the two channels as the final output of edge detection. Here's an example of my output for this step.



3. Perspective Transform

The code for my perspective transform is available at from `alld.trans.PerspectiveTrans` class that includes a function named `warper()`, which appears in the file `adv_lane_line_det.ipynb` cell 7 function `demo_perspective_trans()`. The `warper()` function takes as inputs an image (`img`). I chose the hardcode the source and destination points in the following manner:

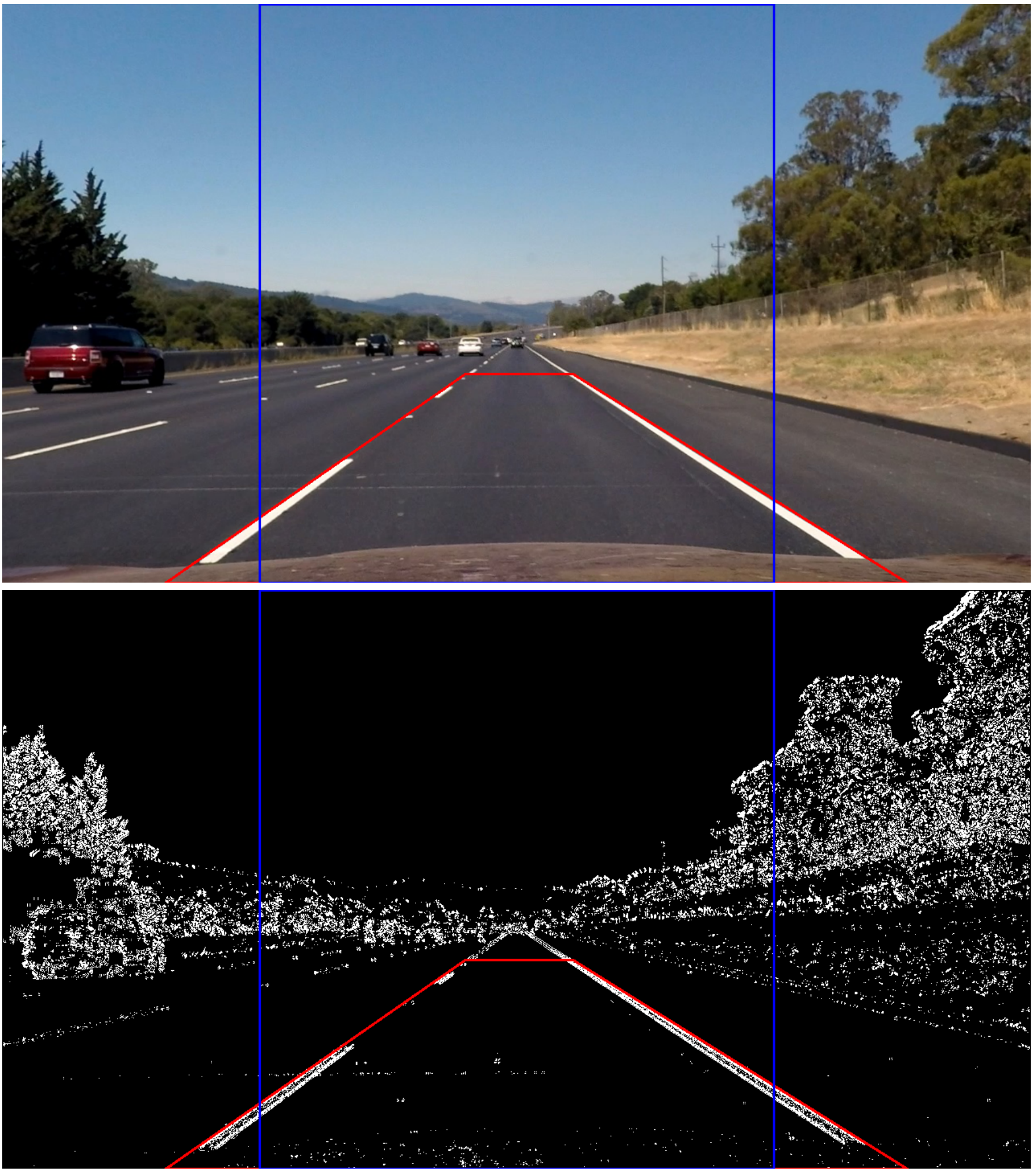
```
src = np.float32(
        [[(w / 2) - 64, h / 2 + 100],
```

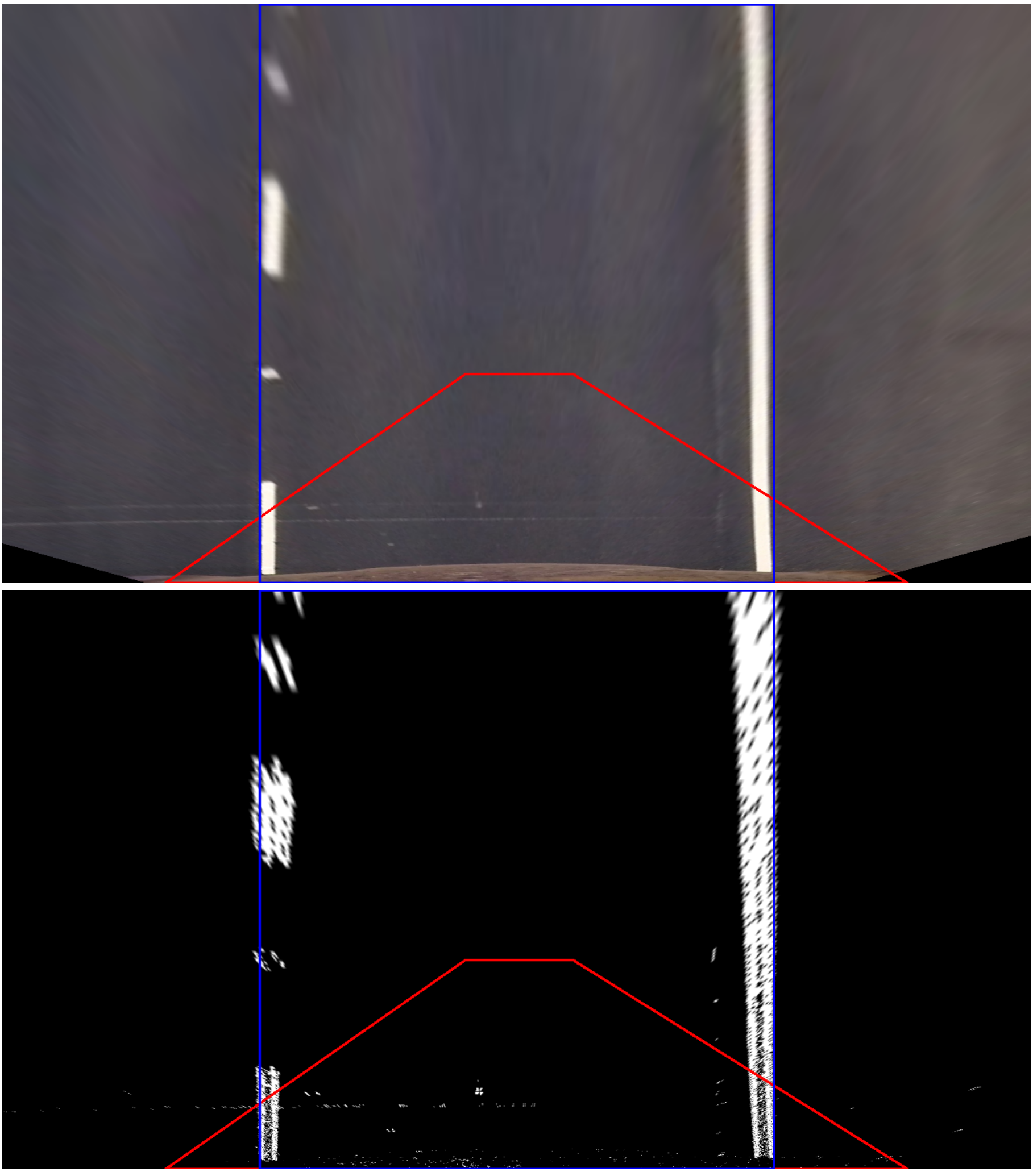
```
        [((w / 6) - 10), h],
        [(w * 5 / 6) + 60, h],
        [(w / 2 + 70), h / 2 + 100]])

dst = np.float32(
        [[(w / 4), 0],
        [(w / 4), h],
        [(w * 3 / 4), h],
        [(w * 3 / 4), 0]])
```

This resulted in the following source and destination points:

Source	Destination
576, 460	320, 0
203, 720	320, 720
1127, 720	960, 720
710, 460	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. I show 4 images: 'original view-normal', 'original view-edge', 'bird-eyes view-normal' and 'bird-eyes view-edge':





4. Lane Area Detection

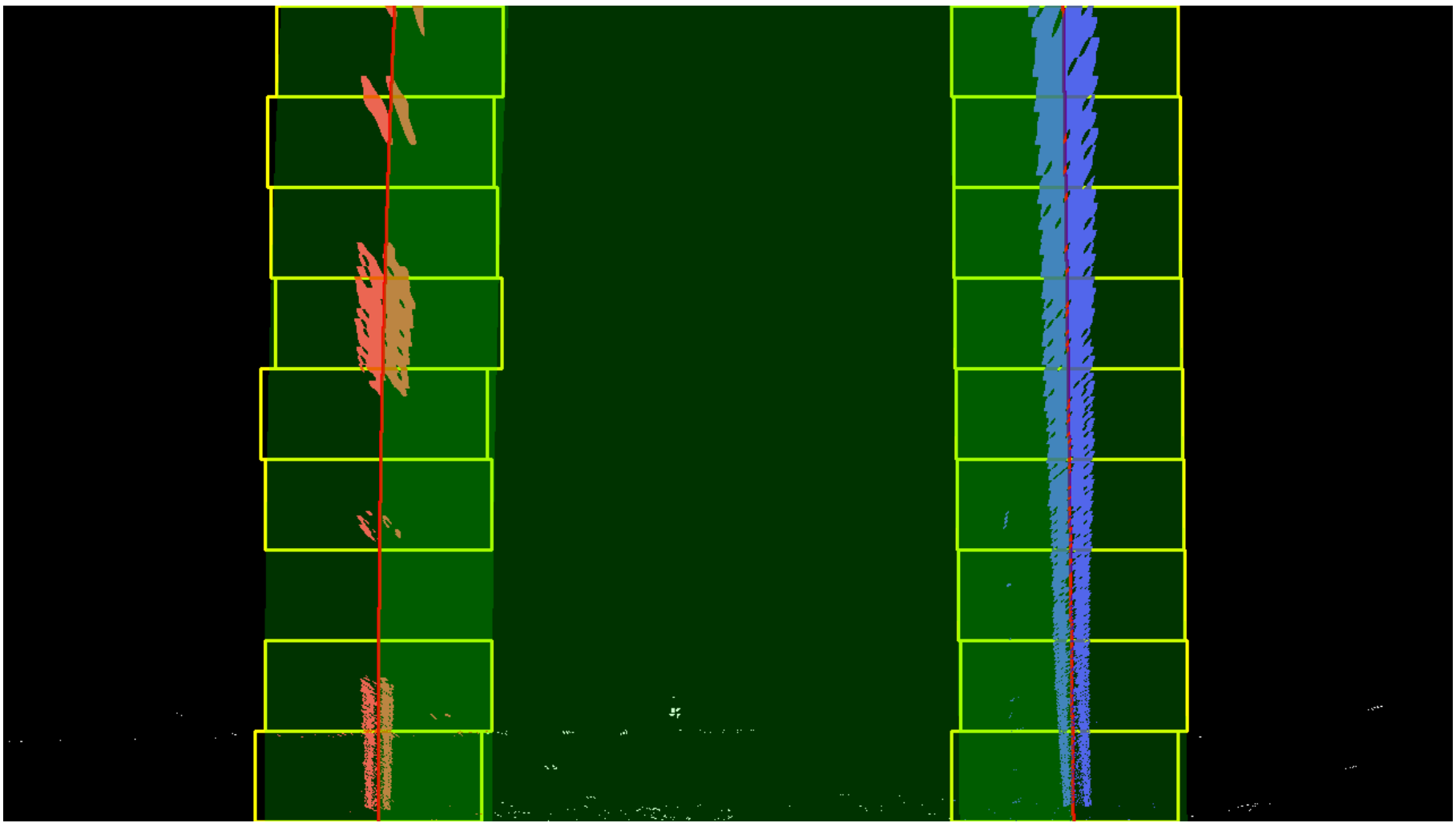
Once the bird-eyes view edge had been prepared, the next step is to detect the lane lines between that. The lane line detection algorithm assumes there are two lane lines in a road, split the two lane lines by mid point. The algorithm will have specified scanning windows (default is 5) per each lane line. Those window will scan form bottom to up. The start point of scanning in two lines will be decided by the histogram of the bottom portion. The x has most points in average will be the start points of next scan window. After the window scanning, we will get a sorts of points in two sides (left lane and right lane). We can use polynomial curve (3 degree) to fit the lane line. The code is implemented at `alld.road_mgr.LaneLine.fit()` and `alld.road_mgr.RoadManager::scan()` . After curve be found, we can fill the are within the two lane lines. In general case, users only need to call `alld.road_mgr.RoadManager::detect()` to detect the lane area as the code in `adv_lane_line_det.ipynb` .

4.1 Confidence

I developed a index to evaluate the fitting performance named as confidence. That considers two portions. The first portion is: how good is the curved fitting? We use RMSE to represent that. The second portion is: doest the lane lines reasonable? I measure the distance between the two lines and subtract the width of lane (hard code to 600 pixel). The code of confidence could be found at `alld.road_mgr.RoadManager::confidence()` If the curve can't be fitted properly, the confidence may be 0. It may not a very precise value, but is good enough for my evaluation.

4.2 Fast scanning

Video is consists of frames, the curve between consequent frames may very similar with each other. We many not required to scan by window again. Just apply the exists curve, use it to predict y-value by new x-value then fit it again. The procedure is names as fast `alld.road_mgr.RoadManager::_scan_fast()` . The will be used in most case. If the confidence of fast scanning is lower than threshold (default is 0.8), that will trigger a full scan. The result image of lane area detection is the following:



5. Final Image

5.1 Final Result

After all the steps be applied, we can get a final result now. The code is `adv_lane_line_det.ipynb` cell 7 `demo_final_result()` , it use the `sdcp4.pipeline()` to generate final image, the implementation is locale `sdcp4.py` Here is an example of my result on a test image:

5.2 Radius of Curvature and Position of Vehicle w.r.t Center

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

I applied the formula to calculate radius of curvature in meter:

The code can be found at radius_in_meter `alld.road_mgr.RoadManager::radius_in_meter()` . That is a simple wrap to average the radius of curvature of two lane lines. The code is `alld.road_mgr.LaneLine::radius_in_meter()` .

Another required information is the position of vehicle w.r.t center. It is a naive implementation to calculate the difference between center of screen and middle point of detected lane line then convert pixel to meter. Pixel per meter is a hard coding:

```
Pixel per meter in X: 3.7/height
Pixel per meter in Y: 30.0/height
```

One more detail of implementation is that the code will also detect that are the lanes straight. If the slope of both lane lines are smaller than a threshold (0.05), it will be considered as straight lane lines.

Pipeline (video)

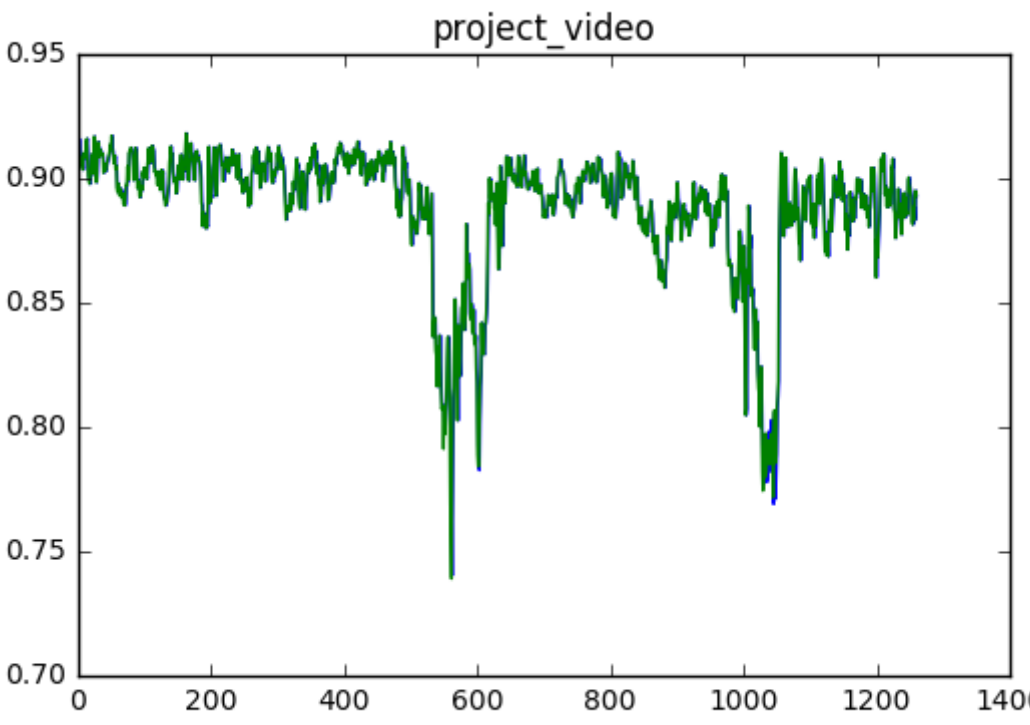
1. Final Video

The pipeline code can be found at `sdcp4.py` , except run by command line, the code of video file generation is also implmented in `adv_lane_line_det.ipynb` cell 8.

Here's a link to my video result

2 Confidence of Lane Line Fitting

We can plot the confidence after run pipeline for all frames. The plot for project_video confidence is:



Discussion

1. Challenge

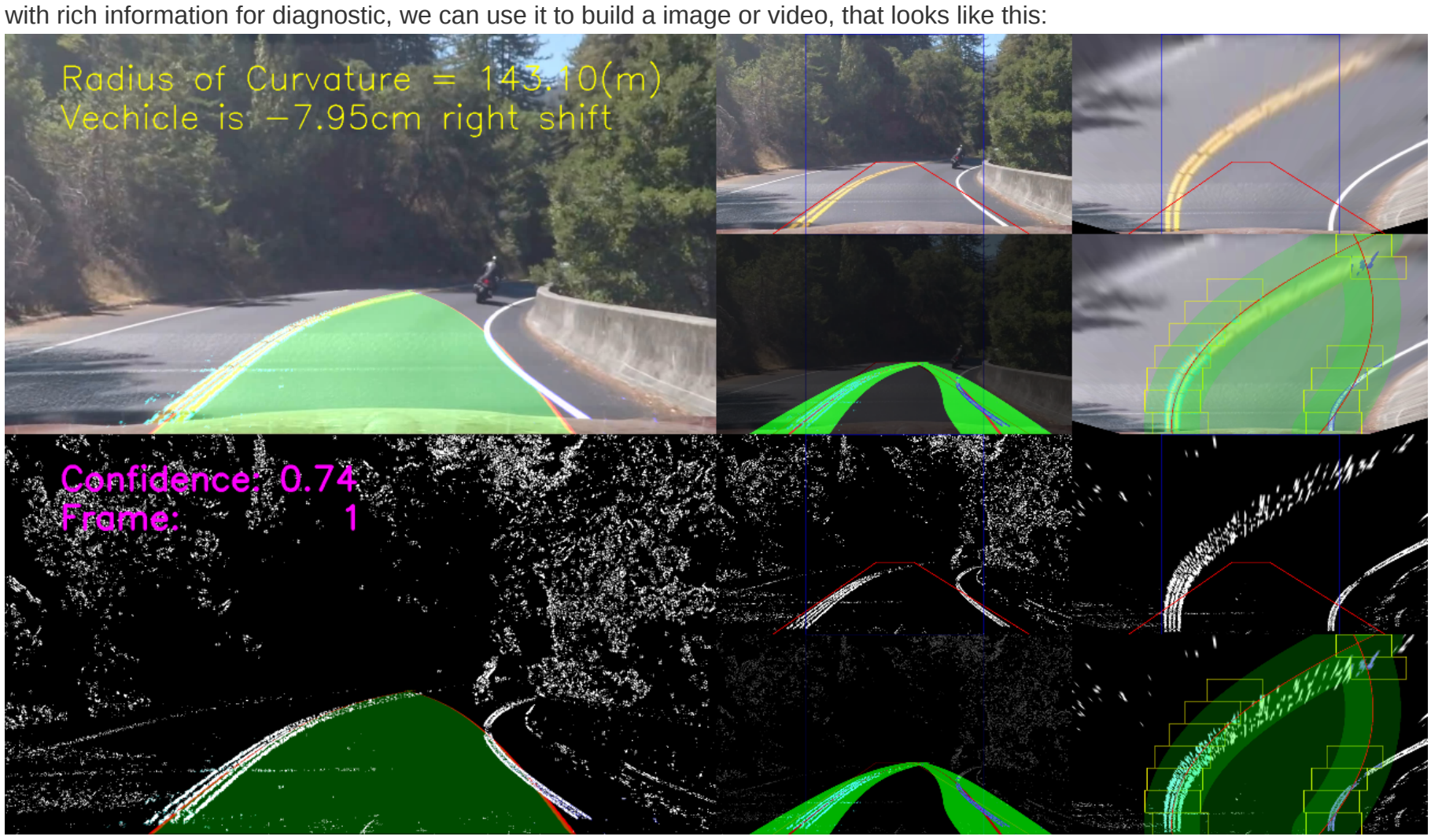
There is a lot of challenge when I building the pipeline. Except there are a lot of decisions need to be decided and they may be required a lot trade off for different scenarios.

Ex. Edge detection method: What channel of color space should I used? What edge detection method is better? What should be the threshold for edge detection?

Except the parameter selection, one more challenge is the scan algorithm. In current design, the scan algorithm is good for the lane lines are nearly to straight. But that will get problem at the hair-pin curve. Because the right lane detector will collect the left lane when the right hair-pin curve like the following image (./frames/harder_challenge_video/frame0140.jpg):

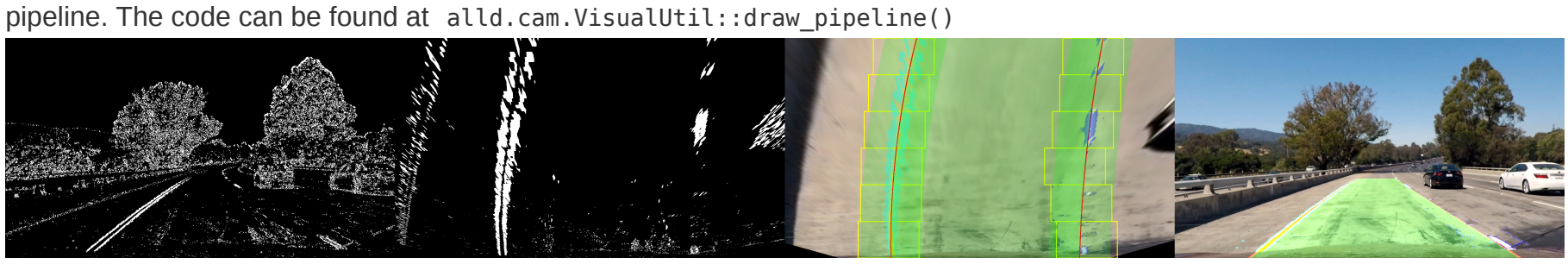
2. Diagnostic & visualization

Visualization will be very useful when tunning the pipeline. I implemented many functions to help me for getting better result. All the function can be find at `/alld/viz_util.py` . There is a class to handle the visualization task: `alld.cam.VisualUtil` Beside the class, there is a diagnostic purpose pipeline function `pipeline_diag()` in `sdcp4.py` that can generate a image with rich information for diagnostic, we can use it to build a image or video, that looks like this:



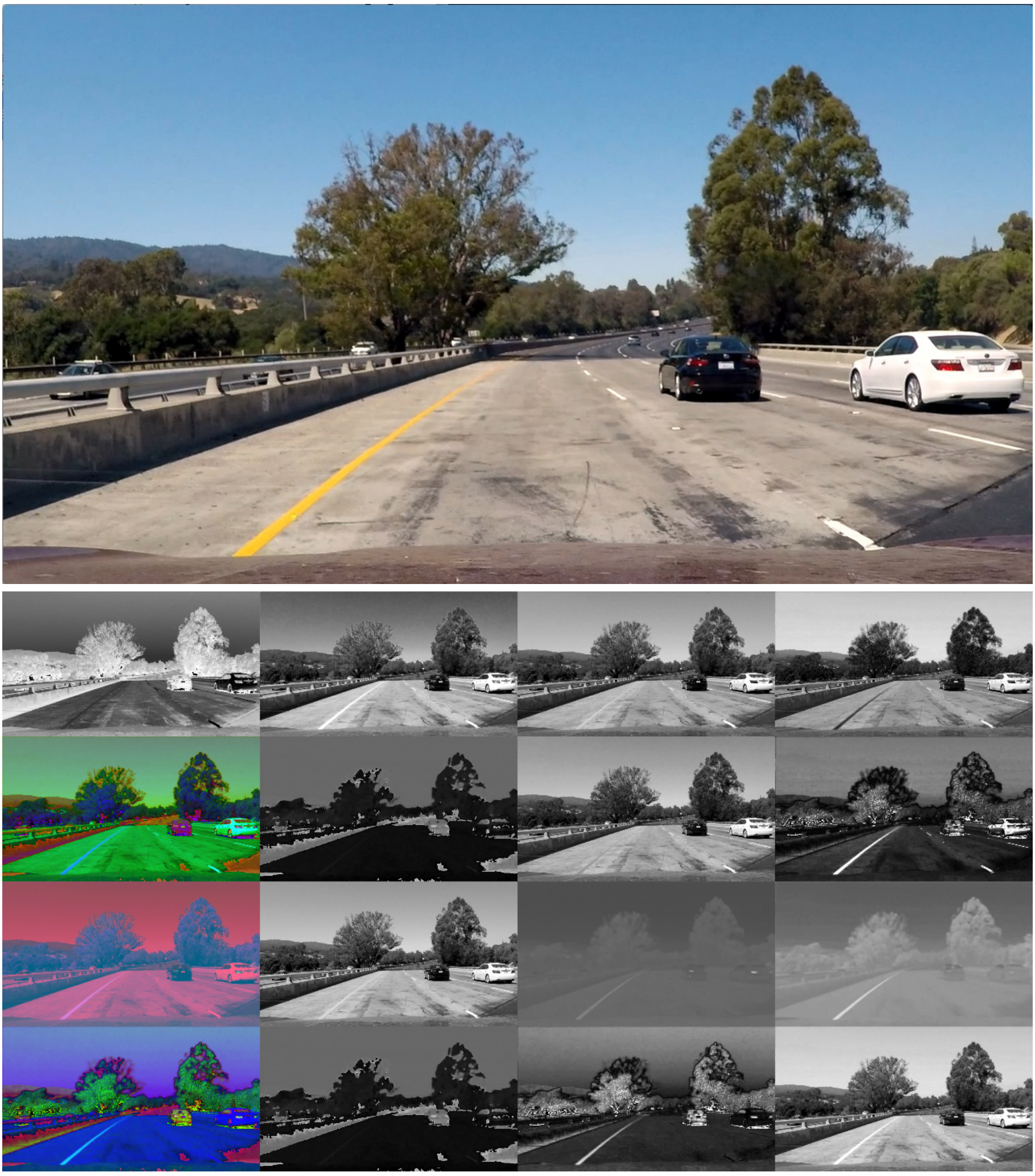
2.1 Pipeline for Diagnostic

When tuning edge detection, we will try the different combinations and threshold. If the major steps in pipeline could be visualized in place, that will be very helpful on this task. I implemented a function to draw a figure which combines major state in pipeline. The code can be found at `alld.cam.VisualUtil::draw_pipeline()`



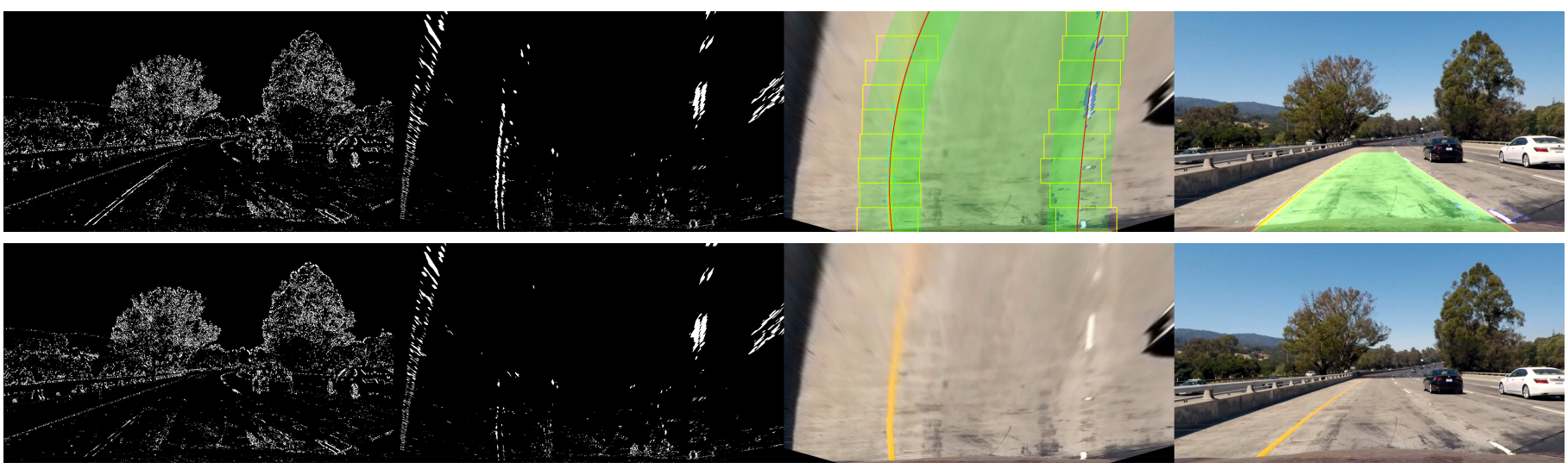
2.2 Draw all channels

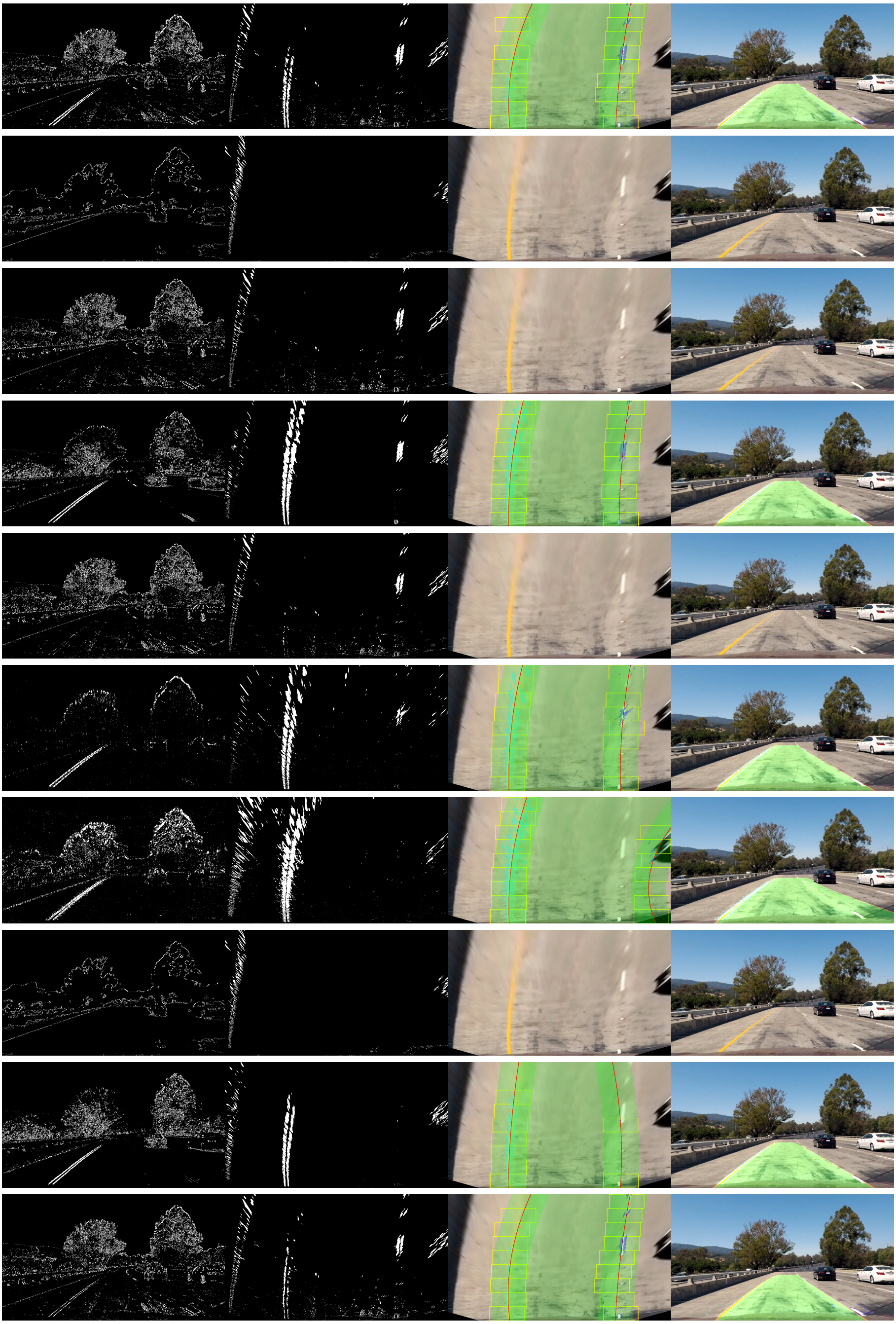
The different scenarios may have different characters in color space, if we want to see what's different between in all color space, there is a function implemented for this purpose, `alld.cam.VisualUtil::draw_all_channels()` , we can generate all channels easily like the following:



2.3 Draw Pipeline for all channels

Once we know the characters within channels for specific scenario, we may draw the pipeline for all channels. If we change edge detection threshold or combinations, we can see the effect on difference channels instantly. The drawing is like the following:



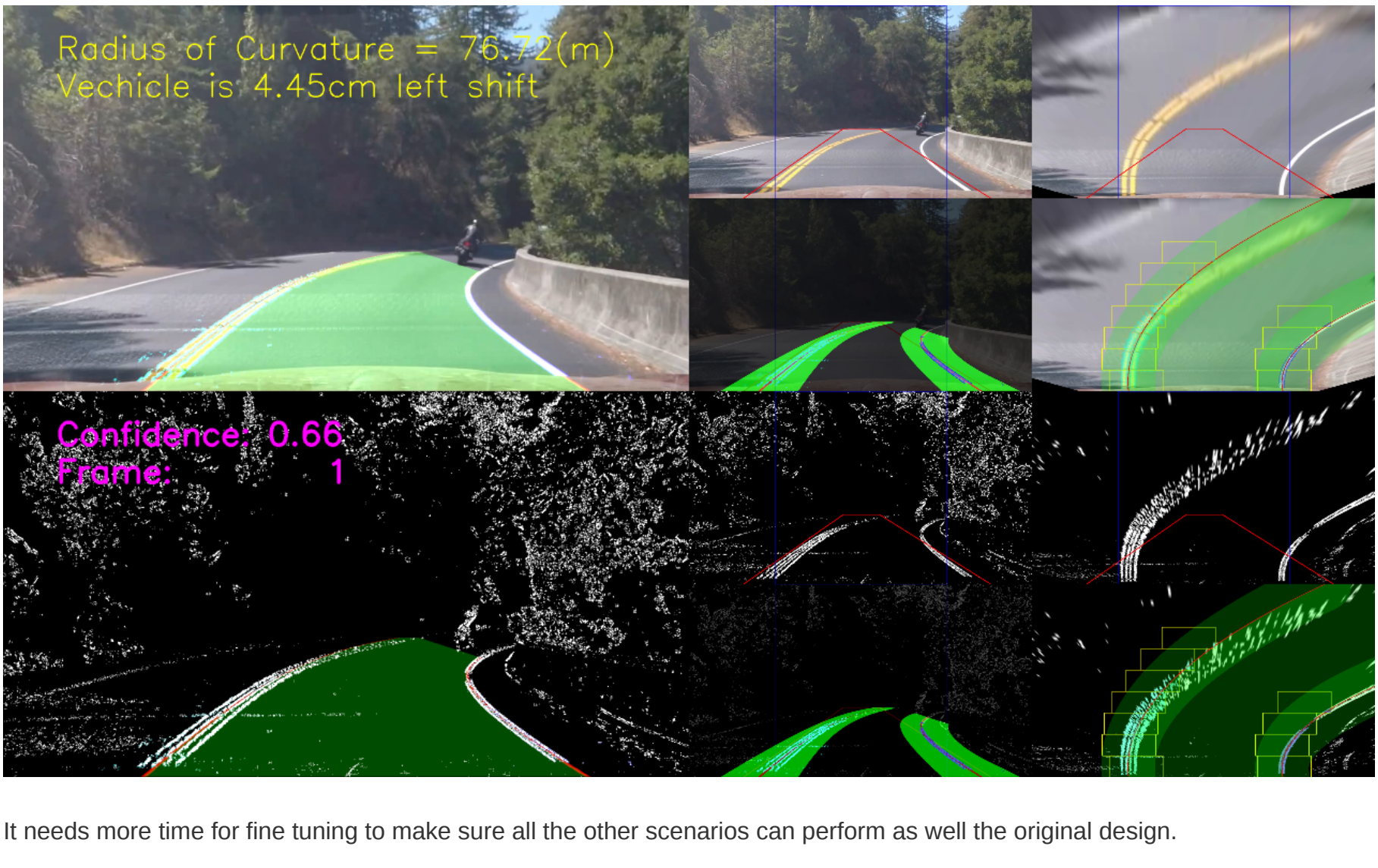


3. Future Improvement

3.1 Scanning algorithm

Scanning algorithm is the most critical part. I think I will consider to improvement in the future. 'Harder challenge video' needs more sophisticate mechanism to detect lane lines as hair pin curve. For example, the './frames/harder_challenge_video/frame0140.jpg' can be detect correct if we skip the scanning procedure when no good pixels

in scanning windows in line. The result is:



It needs more time for fine tuning to make sure all the other scenarios can perform as well the original design.

3.2 Edge detection

Edge detection is very important, if there is a way to let it learning from data or more adaptive, the performance may be better in different scenarios. I will try to find is there any better way to do the detection more adaptive in the future.