Writeup

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. The complete pipline can be found in `P4_pipline.ipynb` for process video while `P4.ipynb` shows the progress of how did I figure out the pipline for single image.

Camera Calibration

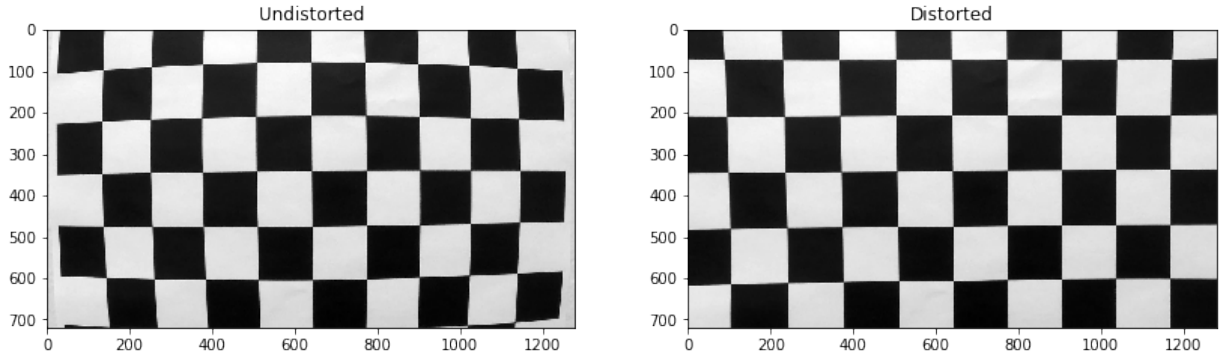
1. Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as a test?

The code for this step is contained in the "Camera Calibration" block of the IPython notebook located in `./P4_pipeline.ipynb`.

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.

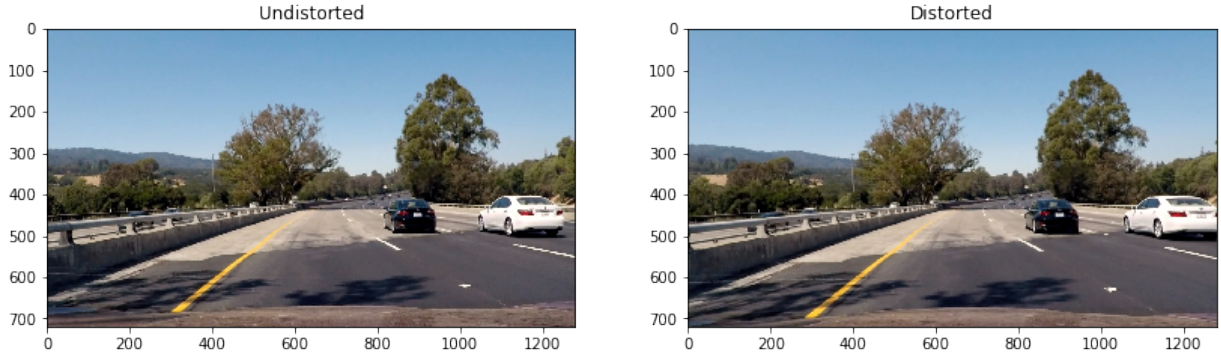
I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



Pipeline (single images)

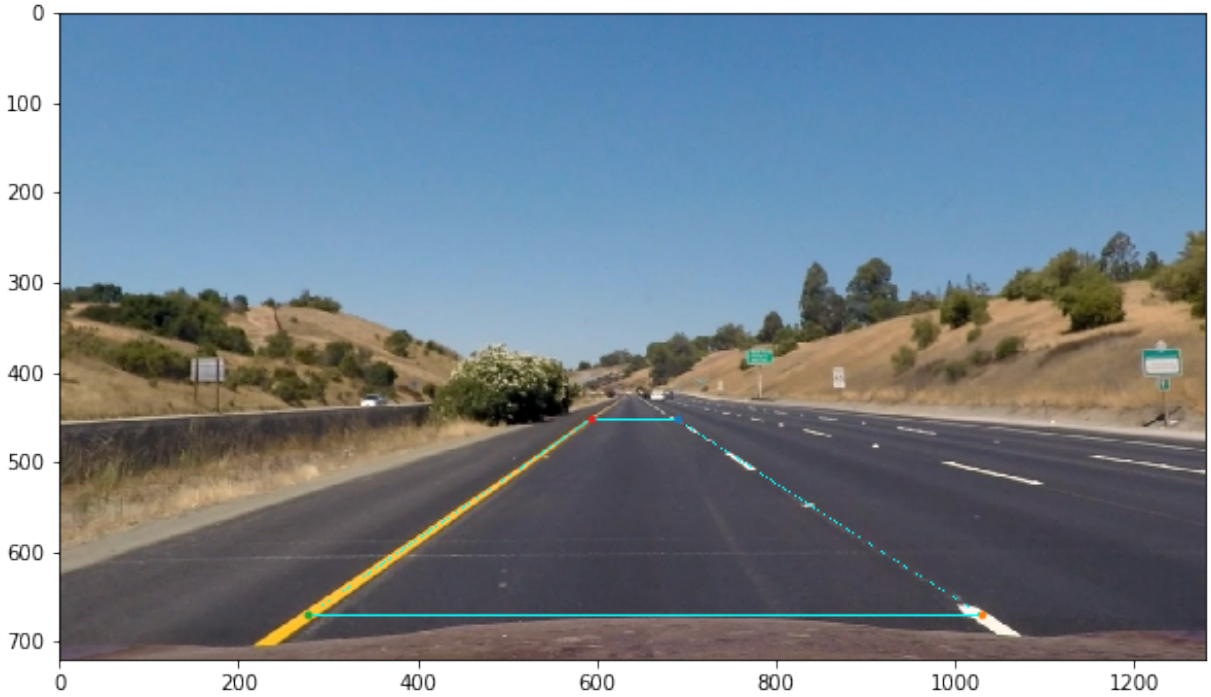
1. Undistorted image

I can do the same operation to undistort a test image just like this one. It looks that the test image has been undistorted successfully.

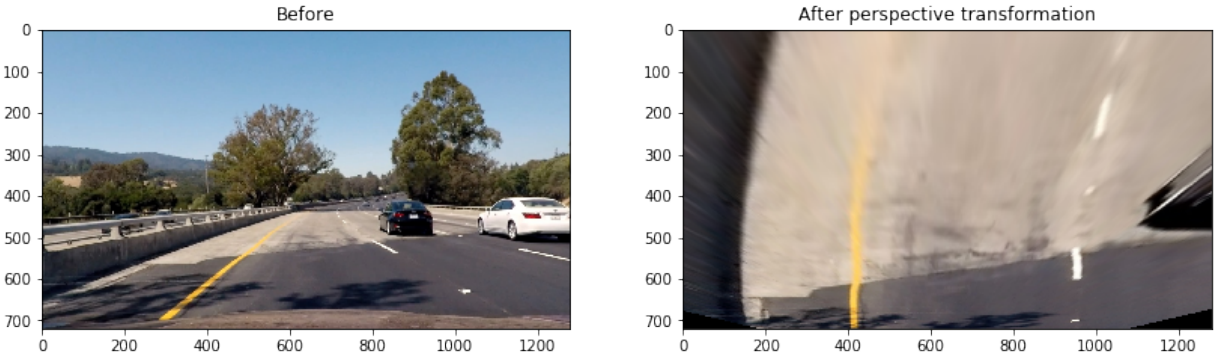


2. Perspective transformation

To perform perspective transformation, first I selected a image that contains straight line, and marked four points on the lane lines, which shoud be four corners of a rectangle in birdview.



With these points, I can use `cv2.getPerspectiveTransform` to get transformation matrix and `cv2.warpPerspective` to get warpped image.



The code for my perspective transform includes a function called `warper()`, which appears in `Perspective transformation` in the file `P4_pipline.ipynb`. The `(perspectiveTrans)` function takes as inputs an image (`img`), as well as transformation matrix `M`. `M` is generated by source points `src_p` and destination points `dst_p`, and I chose the hardcode the source and destination points in the following manner:

```
src_p = np.float32(
        [
            [688, 452],
            [1029, 670],
            [278, 670],
            [593, 452]
        ]
    )

    dst_p = np.float32(
```

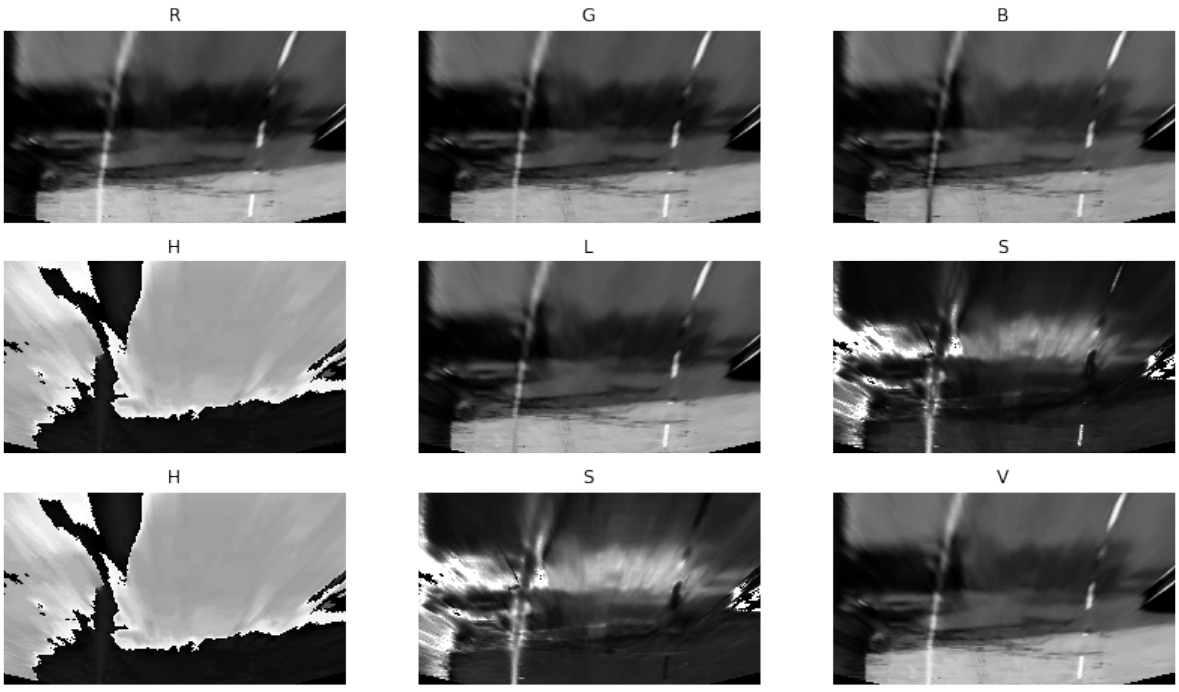
```
        [
            [w-380, 0],
            [w-380, h],
            [380, h],
            [380, 0]
        ]
    )
```

This resulted in the following source and destination points:

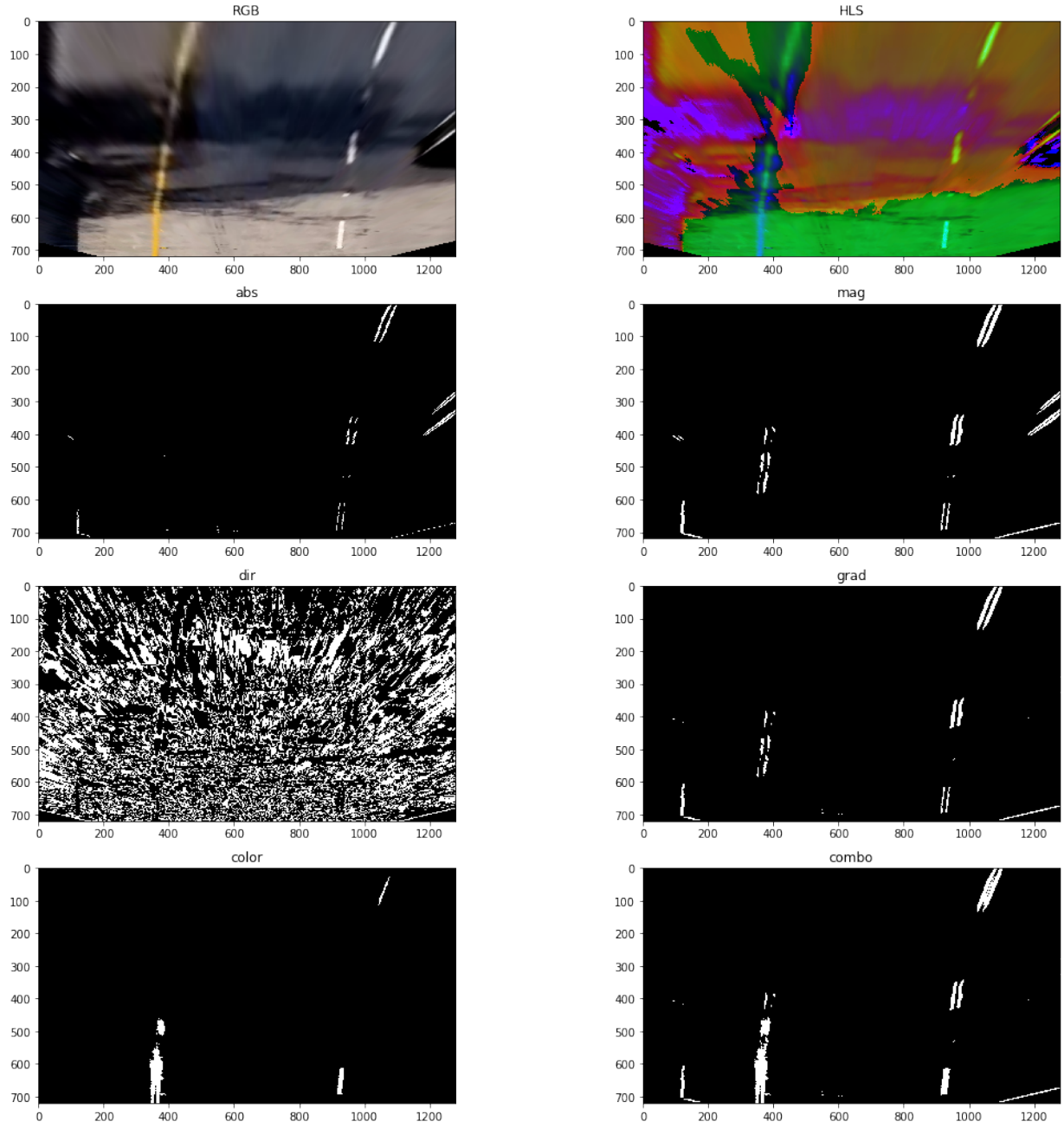
Source	Destination
688, 452	900, 0
278, 670	380, 720
1029, 670	900, 720
593, 452	960, 0

3. Color space and gradient thresholds

I convertedted a RGB test image to HLS and HSV, and plotted them in seprate channels to find which is better for finding the lane line. As shown below, I think R, L, S and V is pretty good. The code of thresholds can be find in `Color space and gradient thresholds` block.

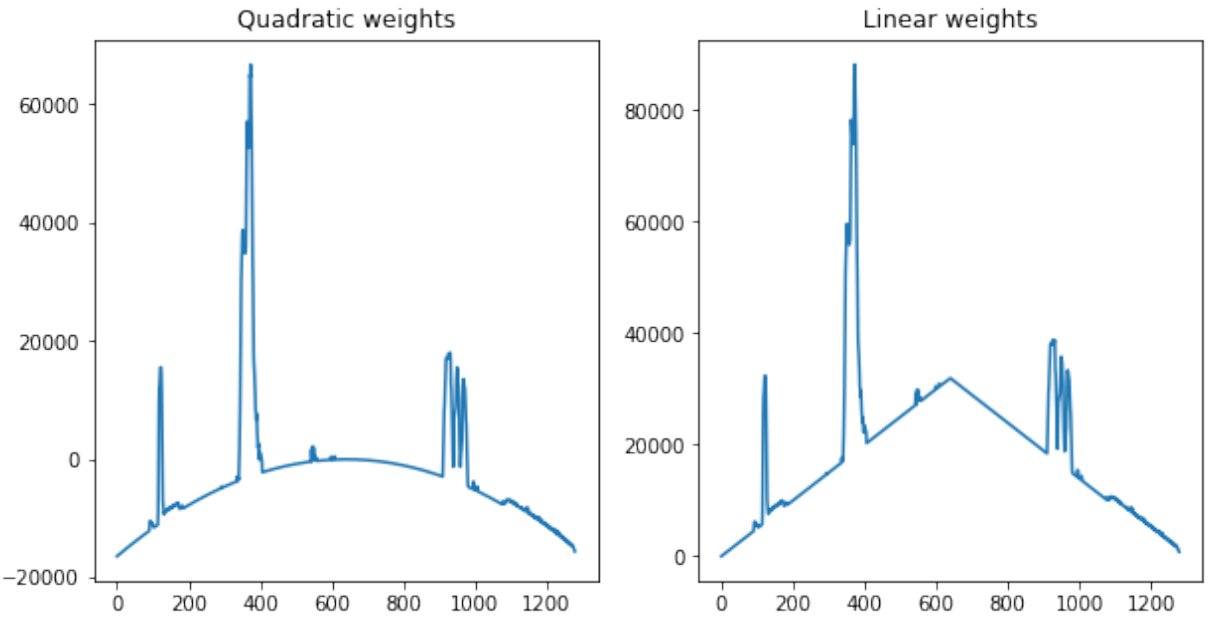


Here's how I apply various of sobel methods to a test image and how I combine the to one single bianry image that show where the lane line is.

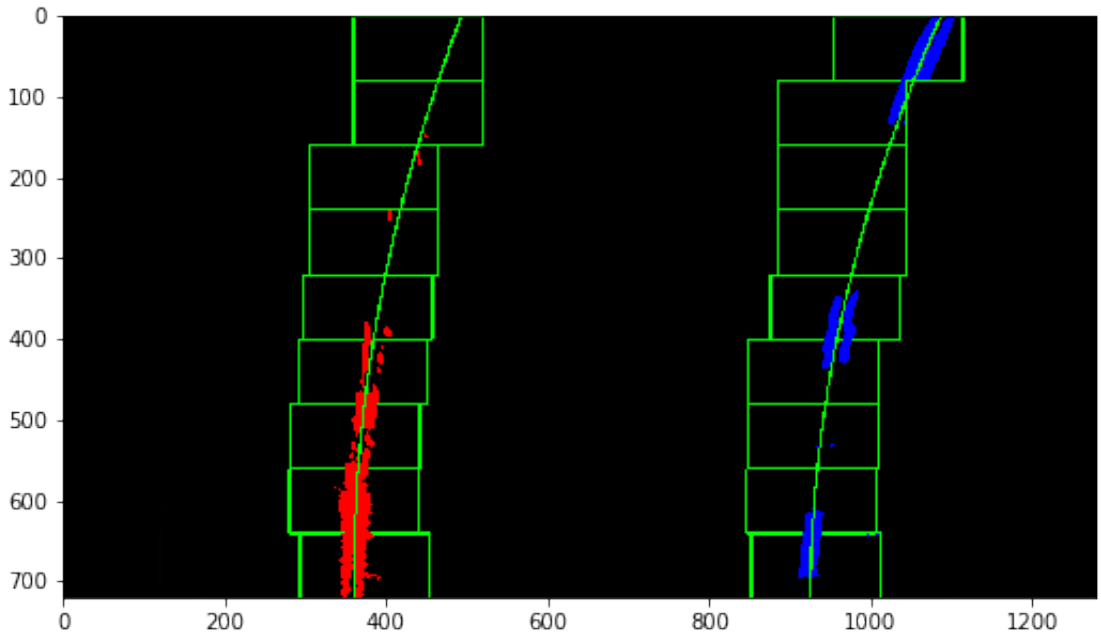


4. Search lane line and polyfitting

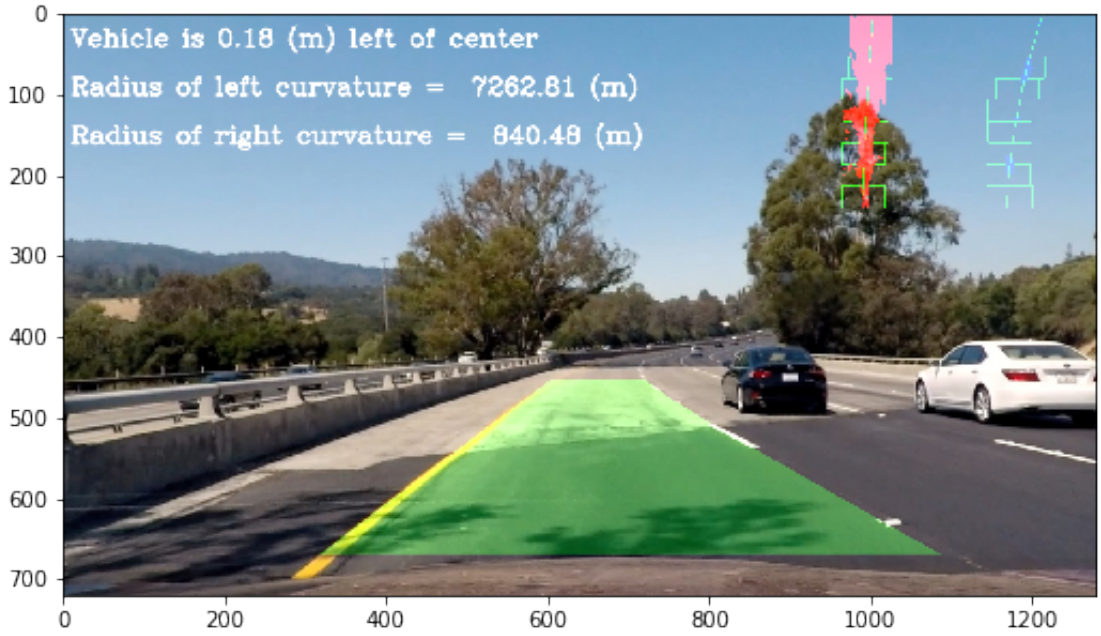
I search the left and right using slideing windows on the left half and right half of the image resprectively. But before I start searching, I must decide two base points at the bottom of the image by histogram, and I will start searching from them. The base points must be on the lane line and I must figure out some tricks to get rid of the noise the looks like a maxium but not a lane line. After manys time of video processing, I find out that I must give the histogram value in the center of x-axis more weights. So I apply following weights to the histogram:



Quadratic weights is better in most of situations. Lane line pixels with polyfit curve(by `np.polyfit`) is shown below:



And I can calculate the curvature of the road using the coefficients returned by `np.polyfit`:



All the code can be found in `Search lane line and polyfitting` block of `P4.ipynb`.

Pipeline (video)

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?

The biggest problem I faced was thresholds. The image color varies in different type of road and different light conditions. Moreover, weather and the time of driving can also have huge impacts on threshold results. I must test millions of times to find a good threshold values for one situation, and they are very likely to have catastrophic performance in other situations. I think just simply use the gradient methods to find lane line pixels is the biggest enemy to make a lane line finding algorithm robust.