Project: Advanced Lane Finding

Patrick Cleeve

Objective

The objective of the project is to develop a lane finding pipeline for autonomous driving using computer vision techniques, using only camera images. The pipeline projects the calculated lane area onto the image taken from a camera mounted on the vehicle.

The project code is available here:
https://github.com/patrickcleeve/CarND-Term1-AdvancedLaneFinding-P4

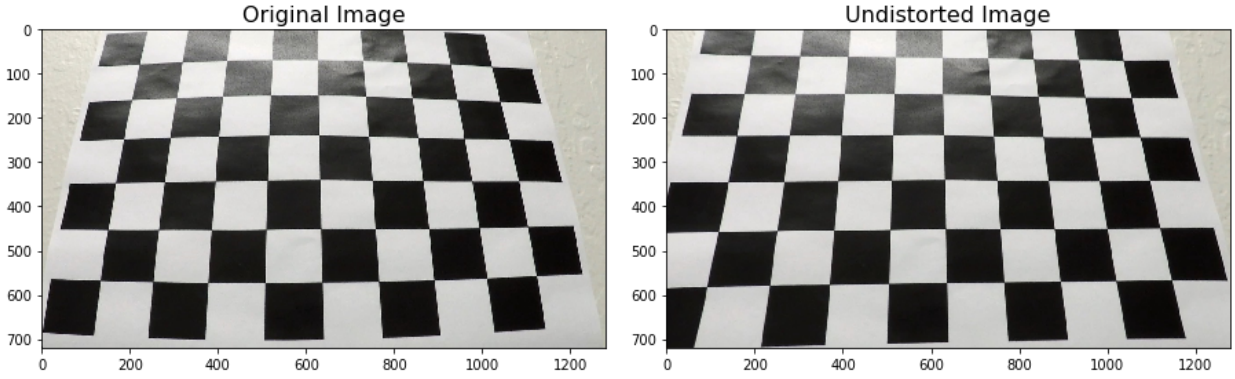
Camera Calibration

Images taken with cameras are distorted because the 3D objects are represented in the 2D image. This causes objects, particularly near the edges of the image to become distorted. Each camera and lens has a different distortion of the object and this distortion are used stylistically, such as fish eye cameras.However, when trying to detect objects or perform measurements in an image, the distortion must be corrected so that all objects regardless of their position in the image can be correctly measured.

The distortion of a camera is corrected using a transform. To calibrate the distortion we use objects of known shapes and size to measure the distortion of the image. For the project we use a chessboard to calibrate the camera, as it has distinct, high contrast squares which can be easily identified in an image. Multiple images (of the chessboard) taken from different perspectives to measure the apparent size of the square in each of these images.

This calibration is done using the following OpenCV functions:

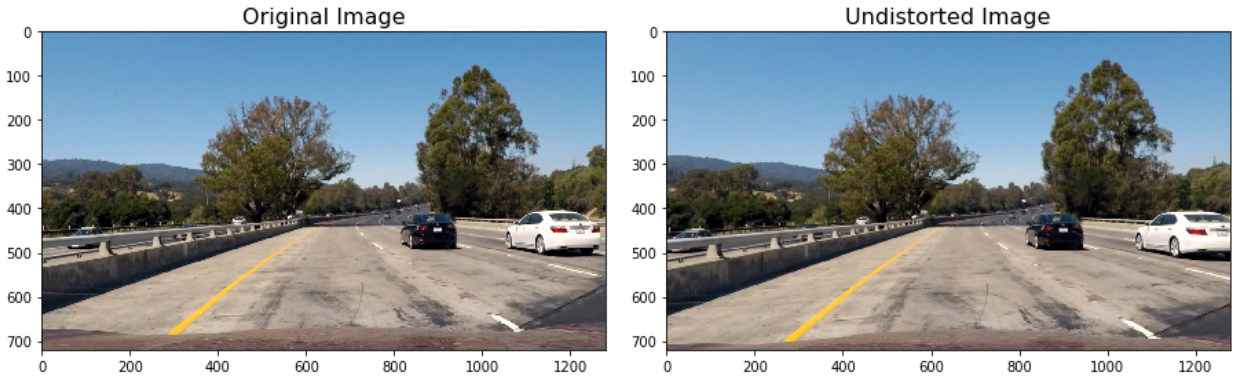
cv2.findChessboardCorners: Identify the chessboard corners in each of the calibration images to map back to the object points.
cv2.calibrateCamera: Use the identified chessboard corners and object points to calculate the camera (transformation) matrix, and distortion coefficients used to undistort the images.
cv2.undistort: Use the camera matrix, and distortion coefficients to undistort the camera images.



Distortion Corrected Image (Chessboard)

Distortion Correction

Now that the camera has been calibrated, we can use the camera matrix and distortion coefficients previously calculated to undistort the images from the road. The image below shows the undistorted road image.



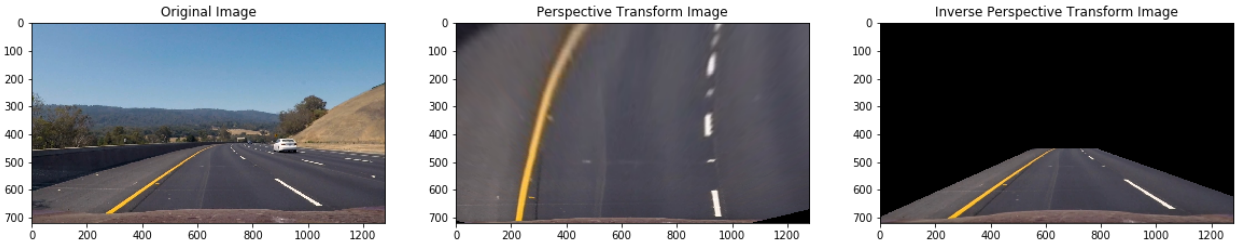
Distortion Corrected Image (Road)

Perspective Transform

Objects further away in images appear smaller, due to perspective. This makes it difficult to measure objects (lane lines) accurately. For example, lane lines appear to converge at the horizon, when in reality the road is straight and lanes are constant distance apart. Mathematically, the greater an object's z-coordinate (distance from camera), the smaller it will appear in an image.

A perspective transform warps an image, and drags points further away towards the camera (and vice versa). For this project we use a bird’s eye perspective transform to view the lane lines from top down, making them easier to detect and measure lane curvature.

To conduct the perspective transform we map source points in the original image to a new top down perspective destination. We use the OpenCV function cv2.getPerspectiveTransform to map the transform matrix between these two perspectives. We can use the same function with reversed arguments (source is now destination, and destination is now source) to compute the inverse perspective transform used to project the lane lines back down to the road. The image below shows both perspective transforms on one of the test images.



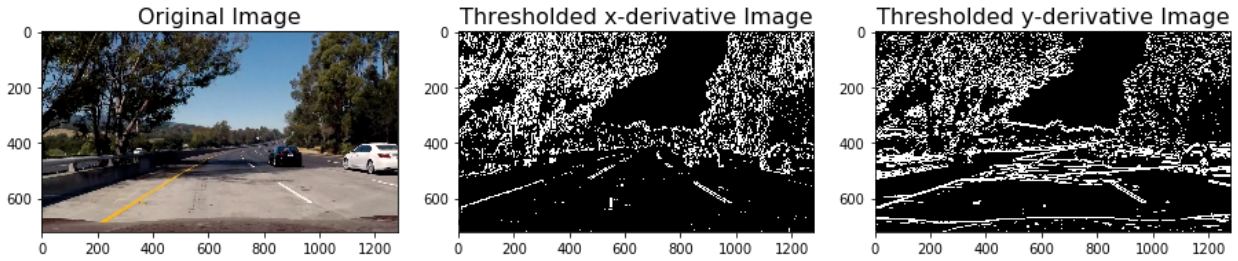
Perspective Transform Images (Road)

Gradient Threshold

We can use a number of different measures of gradient to detect areas of the image corresponding to lines and thresholds to only keep image pixels that potentially correspond to lane lines. The pixels retained show up as white in the binary images, as shown throughout the section.

Gradient (X and Y)

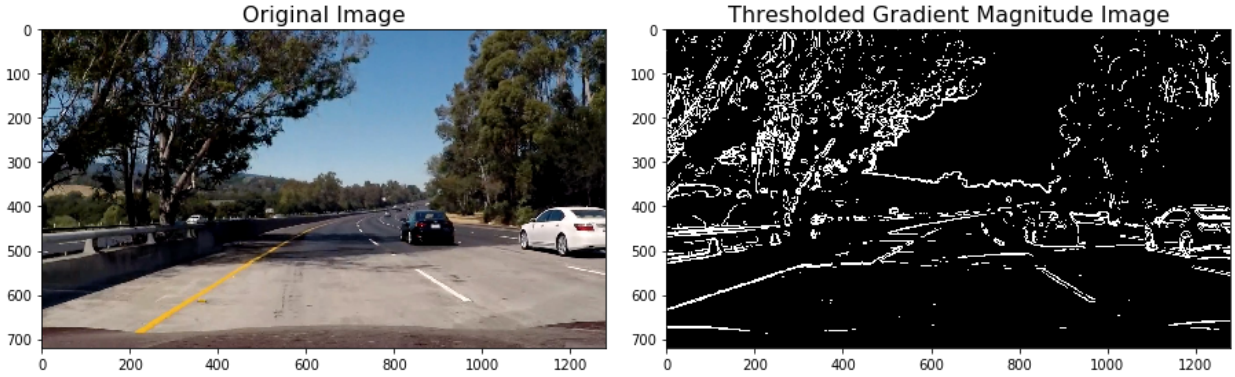
In the introduction lane finding project, we used Canny edge detection to find lane lines. Lines in images can be identified where the gradient changes over a small region. We can use the Sobel operator to detect the image gradient in both the x and y direction. Once we have calculated the gradient in each direction we can then use a threshold to limit the gradients to the most significant lines. The image below shows thresholded x and y gradient binary images.



Thresholded Gradient Images (Road)

Gradient Magnitude

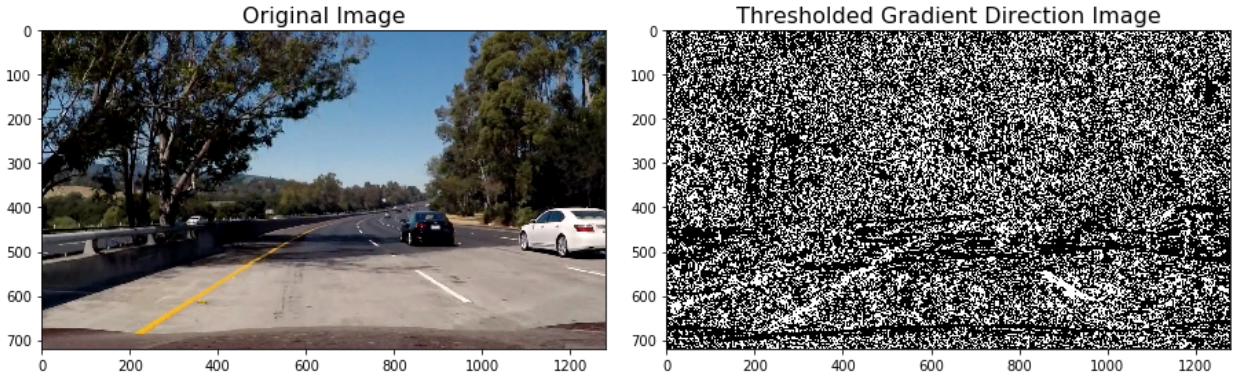
Instead of using the x and y gradients individually, we can combine both to threshold the magnitude of the gradient in both x and y directions. The magnitude is square root of the sum of squares of individual x and y gradients. The image below shows the thresholded gradient magnitude binary image.



Thresholded Gradient Magnitude Image (Road)

Gradient Direction

Because we are only interested in lane lines, which are primarily close to vertical, we can use the direction of the gradient to limit the pixels returned. This measure is the arctangent of the y gradient and x gradient. We can threshold the direction so only lines near vertical are returned, however, this measure produces a large amount of noise, as shown below.



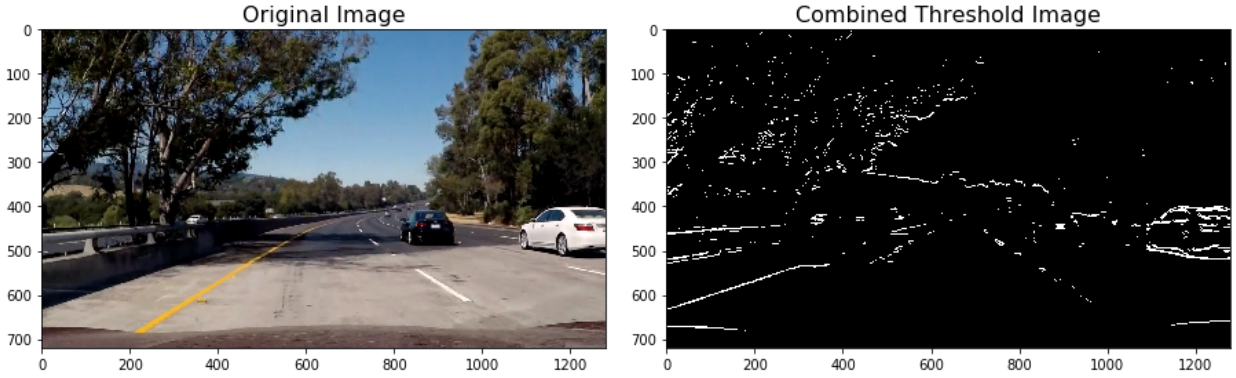
Thresholded Gradient Direction Image (Road)

Combined Gradient Threshold

We can combine these different gradient threshold techniques to create a more robust method of detecting lane lines. The kernel size can be adjusted to calculate the gradient over a larger area, with larger kernel sizes reducing image noise. The table below shows the different gradient measures and thresholds used for the project.

Measure	**Kernel Size**	**Threshold**
Gradient (X)	15	(50, 150)

Gradient (Y)	15	(50, 150)
Gradient Magnitude	15	(80, 200)
Gradient Direction	15	(pi/4, pi/2)

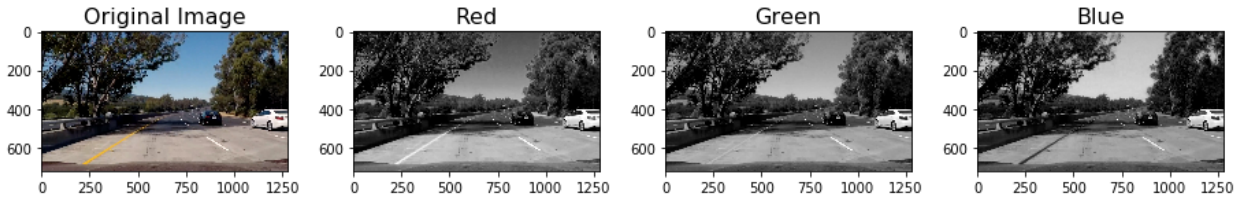


Combined Gradient Threshold Image (Road)

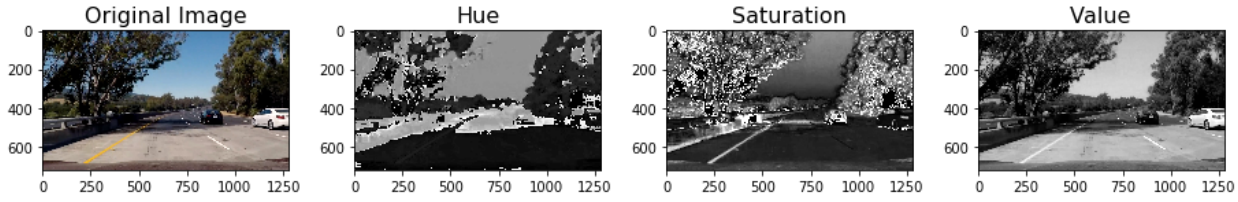
Colour Threshold

Just as we identified lane pixels using the gradient of the image we can use the colour of the lane markings to detect lane pixels in the image. There are a number of different colour spaces that we can use to try to robustly detect lane lines in a variety of conditions.

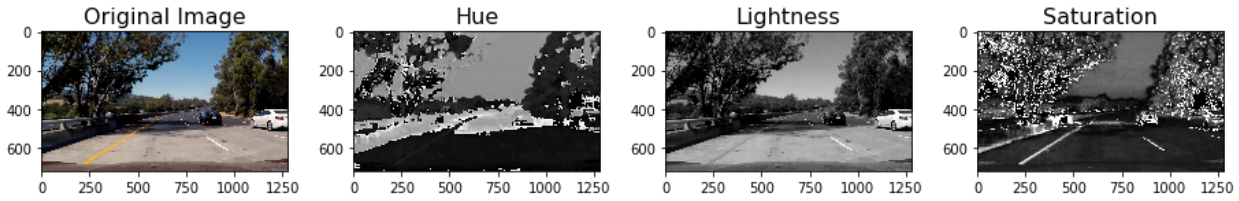
Red Green Blue (RGB)



Hue Saturation Value (HSV)



Hue Lightness Saturation (HLS)



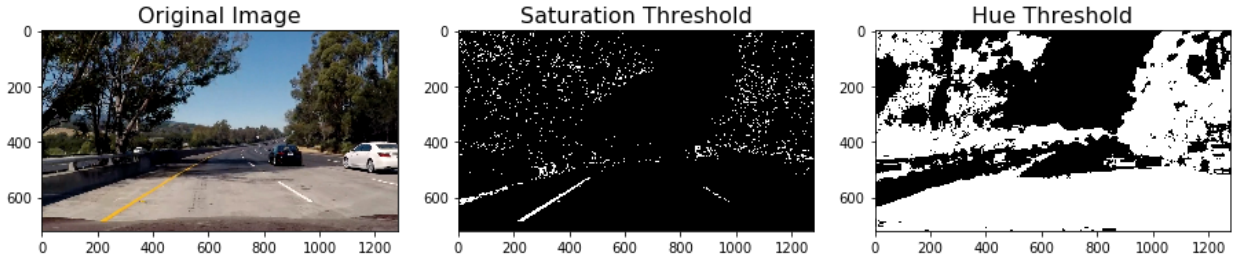
The HLS colour space was chosen for the following reasons:

Saturation:

- The saturation of the lane lines provides the most consistent marker of lane lines, even with different yellow and white lines.
- Other channels such as Red, or just Grayscale lose a significant amount of information.

Hue:

- Saturation threshold struggled in shadowed areas
- Hue channel was constant even in shadow areas, however it had a lot of excess information. Needed to be combined with saturated areas to only isolate lane lines.

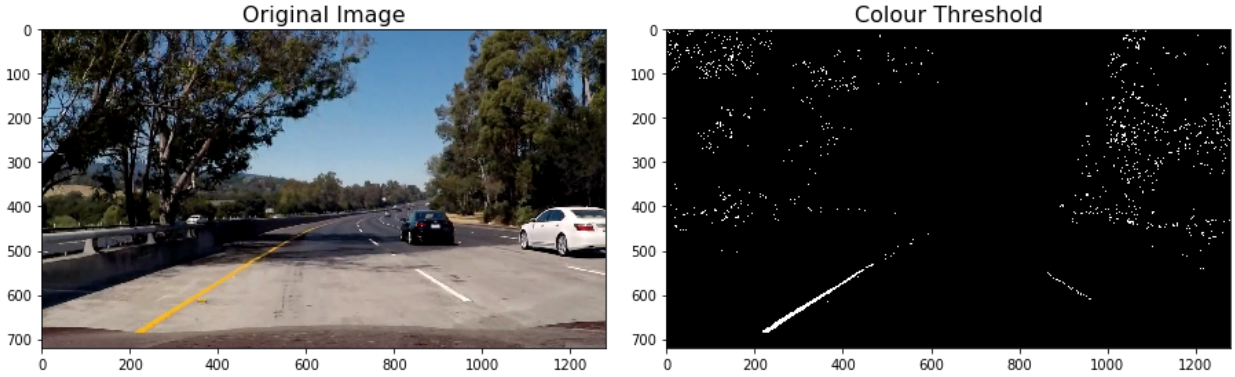


Saturation and Hue Threshold Images (Road)

Combined Colour Threshold

We combine both of the saturation and hue thresholds to produce a more robust detection. We can therefore loosen constraints on saturation thresholds to pick up more of the lane lines, then use the hue channel to filter to only include lane, and not pick up excess information.

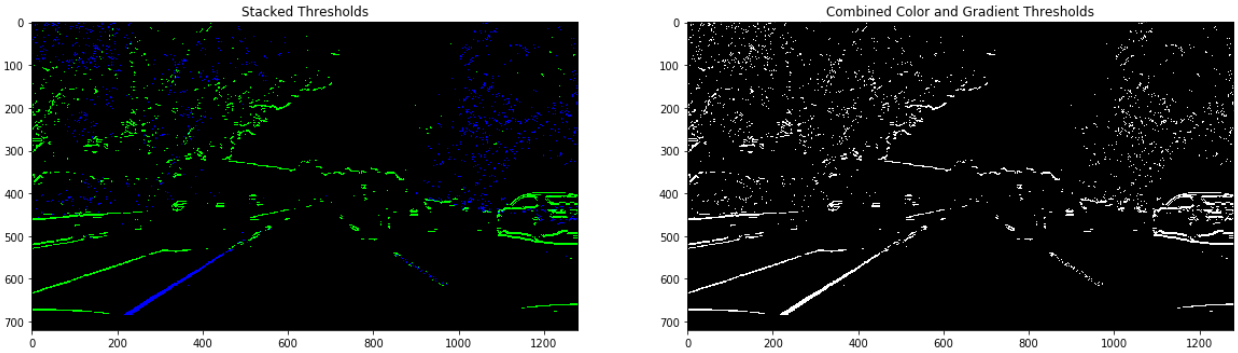
Measure	**Threshold**
Saturation	(180, 250)
Hue	(0, 50)



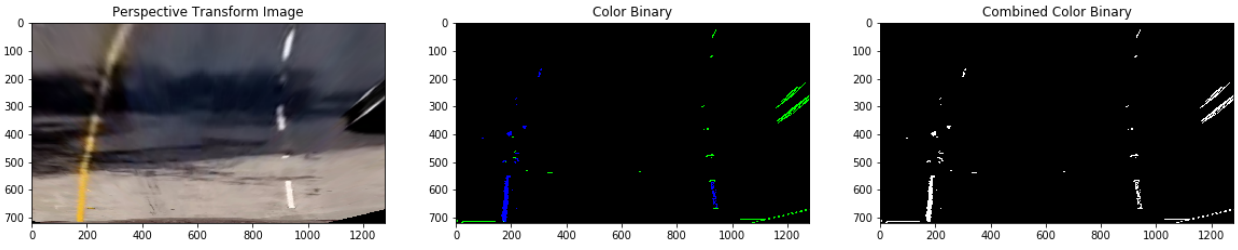
Combined Colour Threshold Image (Road)

Combined Threshold

We combine and stack both gradient and colour threshold techniques to create a more robust method. This allows the pipeline to perform in different conditions, by relying on different techniques in different conditions. These stacked threshold filters are shown below:



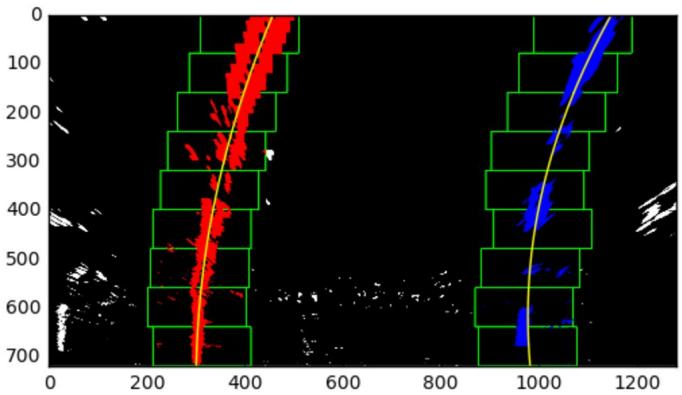
Gradient (Green) and Colour (Blue) Threshold Image



Combined Threshold Images (Perspective Transform)

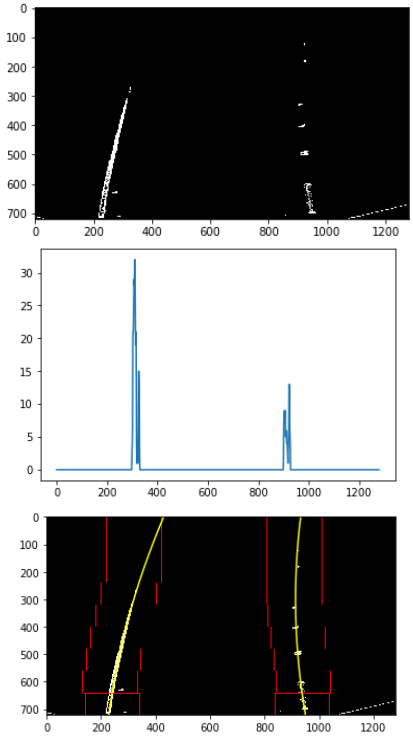
Lane Line Identification

Having isolated potential lane pixels using gradient and thresholding techniques, we now must identify the left and right lane lines in the image. We use the sliding window technique to identify each lane's pixel, then fit a second order polynomial through the identified pixels to create a lane line.



Sliding Window Example (Udacity Lecture)

The algorithm uses the following steps to identify lane lines:



- Identify potential lane pixels using colour and gradient thresholds.
- Identify starting point of potential lane line using histogram of pixel values (bottom half of image only).
- Divide image vertically into a set of rectangular windows, and iterate through each window (Number of windows = 9).
    - Check to see if the minimum required pixels lie within the current window.
    - If not, re-centre the window on the mean pixel position.
    - Otherwise, append the list of pixel values to left and right lane line arrays.
- Once all windows have been evaluated, fit a second order polynomial to each of the points found for each left and right lane lines using np.polyfits.
- Plot the calculated polynomial lane lines onto the image (see yellow lines).

Lane Curvature and Position

Lane Position:

We can calculate the position or offset of the car in the lane, using the assumption that the centre of the image is the centre of the car. The position of the centre of the lane is the midpoint between lane lines, measure at the vehicle. We calculate the lane offset as the distance between centre of image and centre of the lane lines (Converted to metres).

Lane Curvature:

We calculate the lane curvature using the function measure_curvature_real(). This function takes the lane points, and calculates the lane polynomials in real space (i.e. in metres instead of pixels). Then using the polynomial, we can calculate the curvature of each line using the following equations:

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

The curvature is then averaged across both lines. Both lane curvature and offset are displayed on the image.

Lane Offset and Curvature (Road)

Lane Visualisation

We can now plot the identified lane lines and attributes (curvature, position) back onto the original image. The area between the lane lines is filled to show the identified lane. The inverse perspective transform is used to convert from the top down perspective used for identification, back to the original driving perspective. The image below shows the original road image, and the final processed image, identifying lane area, and lane attributes.

Lane Line Identification (Road)

Pipeline

We combine these each of these processing steps into a pipeline to identify lane lines. The pipeline is as outlined (Note: camera calibration separate):

process_image(image):

- Load Image
- Undistort Image
- Perspective Transform Image
- Compute Gradient Threshold Image
- Computer Colour Threshold Image
- Combined Gradient and Colour Threshold Binary
- Sliding Window Algorithm
- Polynomial Line Fitting and Plotting
- Inverse Perspective Transform
- Calculate Lane Curvature and Position (Offset)
- Return Processed Image

Discussion

The pipeline uses computer vision techniques to detect line pixels in the image, then fits a polynomial line through these points to determine each left and right lane line.

The set up and detection of lines was very manual, and iteration intensive. It was difficult to determine which threshold techniques (gradient and/or colour) and the numbers which would produce a good result. This was increased, when combining multiple thresholds together and producing an adequate outcome. It was effective to combine multiple filtering technique together, each with different characteristics to create a relatively robust detection in all conditions (such as changing road colour, or shadowed areas).

The perspective transformation was a powerful technique that made lane line detection more straightforward from a birds eye view. A number of different source points were evaluated, and transformed. When the source shape was more square, the pixels towards the top of the image were less stretched out resulting in an easier lane pixel detection. However, as the road curves it was important to ensure that it didn't pick up objects (such as other vehicles in the next lane) interfering with the lane prediction. Additionally, if the lane is highly curved (such as challenge videos), the current algorithm will just continue vertically upward resulting in an incorrect prediction. This will need to be resolved for later versions.

The current lane detection algorithm is very inefficient, it does not utilise the previous frame's lane lines and restarts the search blindly in each frame. This result in reduced performance, and can cause jitter in the lane projection. To resolve these issues, the algorithm should utilise the previous lane line, perform a search only around a margin and check if the required number of pixels are found. In addition, outlier rejection needs to be added, such as building lane lines as a weighted composite of previous lane estimates, rather than calculating the entire line again. These techniques would likely improve performance, accuracy and reduce jitter as the next line segment is logically a continuation of the previous segment. This would also improve performance in sections were multiple lines (such as painted over old lines) were present on the road, as the weighted average would increase the likelihood of the actually continuing road lines (from previous segments) continuing rather than the incorrect lines being detected.

Reflection

The project demonstrated the power and relative simplicity (and interpretability) of a pure computer vision approach to the lane detection problem. It was a very manual process to set up each of the thresholding techniques and perspective transform, which made the pipeline feel specific to the individual road segment.

To improve the pipeline:

- Implement a lane object to store previous frame prediction, and use it to reduce the search space for the subsequent frames.
- Implement an outlier detection to build up the lane as a weighted sum of the previous segments, to reduce the overall jitter of the projection.
- Test and try challenge videos.