## Writeup Template

### You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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.

[//]: # (Image References)

## [Rubric](https://review.udacity.com/#!/rubrics/571/view) Points

### Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

### Writeup / README

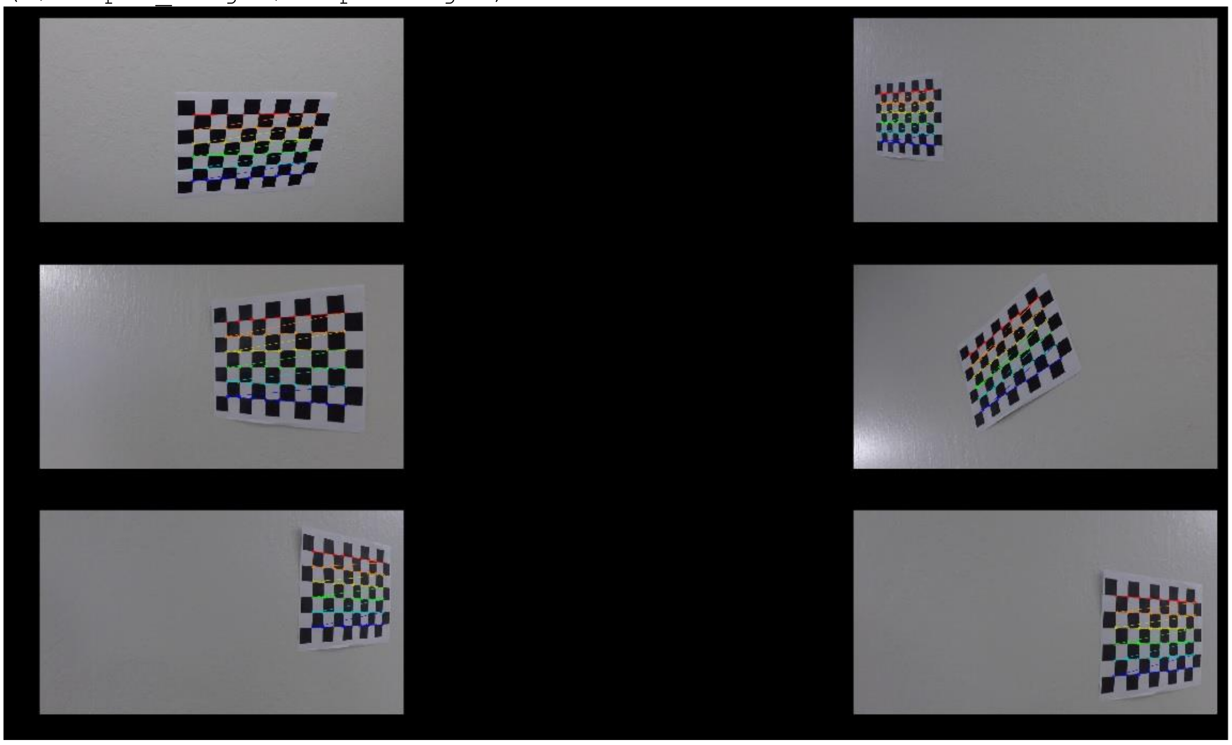
#### 1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. [Here](https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

### 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 CameraCalib.py, the function FindChessCorners(). As explained in the lesson, I prepare the ‘object points’ which will be (x,y,z) coordinates of the chessboard. When we assume, the chessboard to be placed in (x,y) plane at z equals to 0, such that the object points are same for each calibration. objp in FindChessCorners() is the replicated array of coordinates and objpoints will be appended with a copy of it everytime successful detections of chessboard corners in test images happens. Imgpoints will be appended with x & y pixel position of each of the corners of the image plane with each successful chessboard detection. This is illustrated in below image (./output_images/OutputImage1)

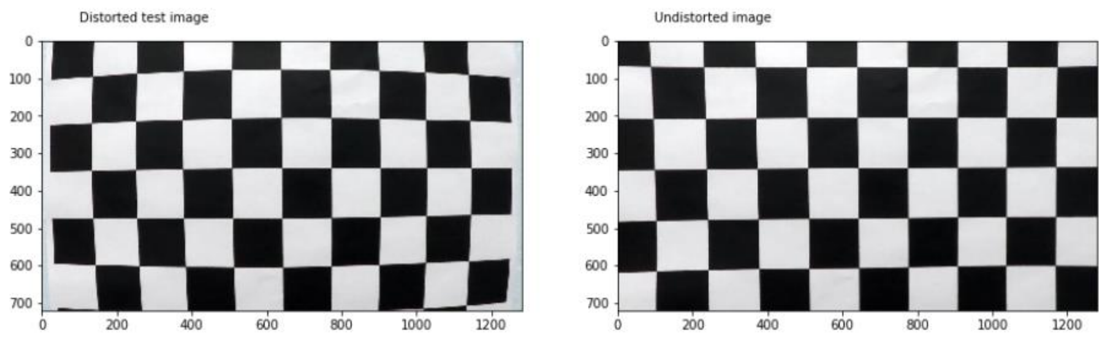


### Pipeline (single images)

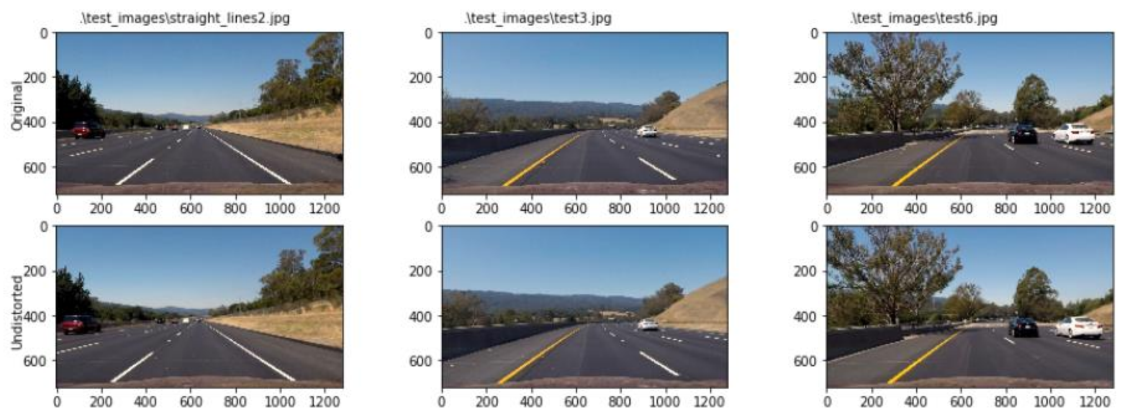
#### 1. Provide an example of a distortion-corrected image.

Camera calibration : objpoints and imgpoints are used to compute the camera calibration and distortion coefficients using cv2 function cv2.calibrateCamera(). This distortion correction is applied to the test image using cv2.undistort() function (UndistImg() function in CameraCalib.py). This distortion and undistortion is illustrated in below images.
To the left side, is the distorted image, to the right is the undistorted image.

```
Out[87]: <matplotlib.image.AxesImage at 0xe9577f0>
```

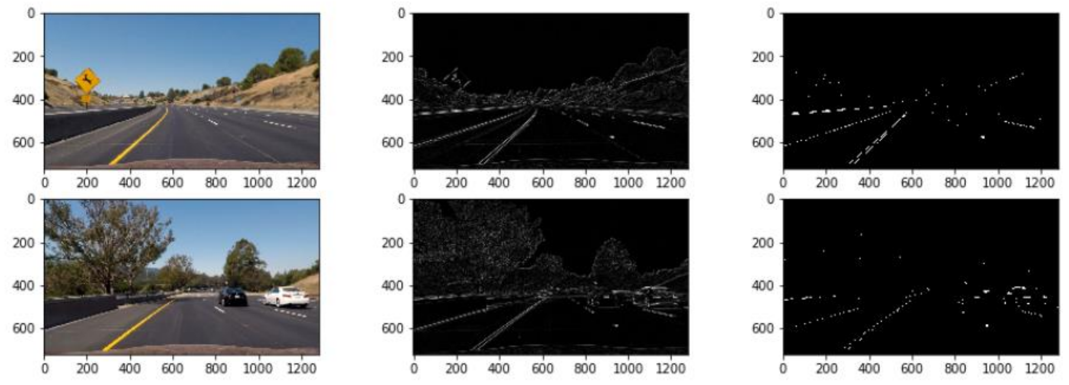


Distortion correction is applied on the test images : Straight_lines2, teset3, test6. This is illustrated below



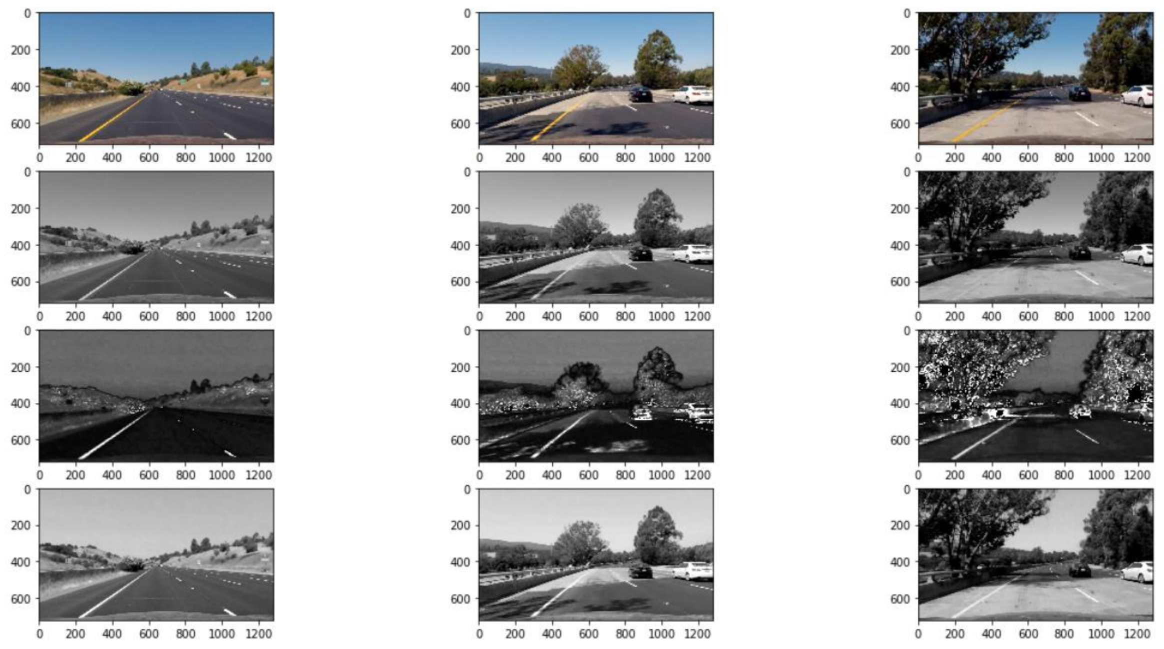
#### 2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.
There are various transformations that can be applied as in lecture. But I tried to apply the direction gradient. Just by applying the gradient in x or y direction doesn't account the lane directions. The direction gradient computes the gradient in direction of alpha angle (typically lane angles as 45 deg to 55 deg. I consider arctan(4/3). This is illustrated below. In the second set, there is not so much performance. It could be due to contrast. I try out color space transforms.

```
In [126]: from CameraCalib import bgr2rgb,bgr2gray
          #alpha is the angle of lane lines which is kept at almost more than 50 degrees since gradient in x or y do
          esn't
          #consider lane angles atleast.
          index=(3, 7)
          plt.figure(figsize=(15,5))
          for i in (0,1):
              plt.subplot(2, 3, 1+i*3)
              plt.imshow(bgr2rgb(undistTestImgs[index[i]]))
              b, s= Sobel(bgr2gray(undistTestImgs[index[i]]), sobel_kernel=7, alpha=np.arctan(1.333), thresh=(125, 2
          55))
              plt.subplot(2, 3, 2+i*3)
              plt.imshow(s, cmap='gray')
              plt.subplot(2, 3, 3+i*3)
              plt.imshow(b, cmap='gray')
```



As in the course, by just transforming to gray scale, much information may be lost. Using color space transformation ,valuable information could be preserved such as value, saturation or lightness.
The test images are undistorted using the above step and are considered for processing here. Images used here : Straightlines1, test4 and test5

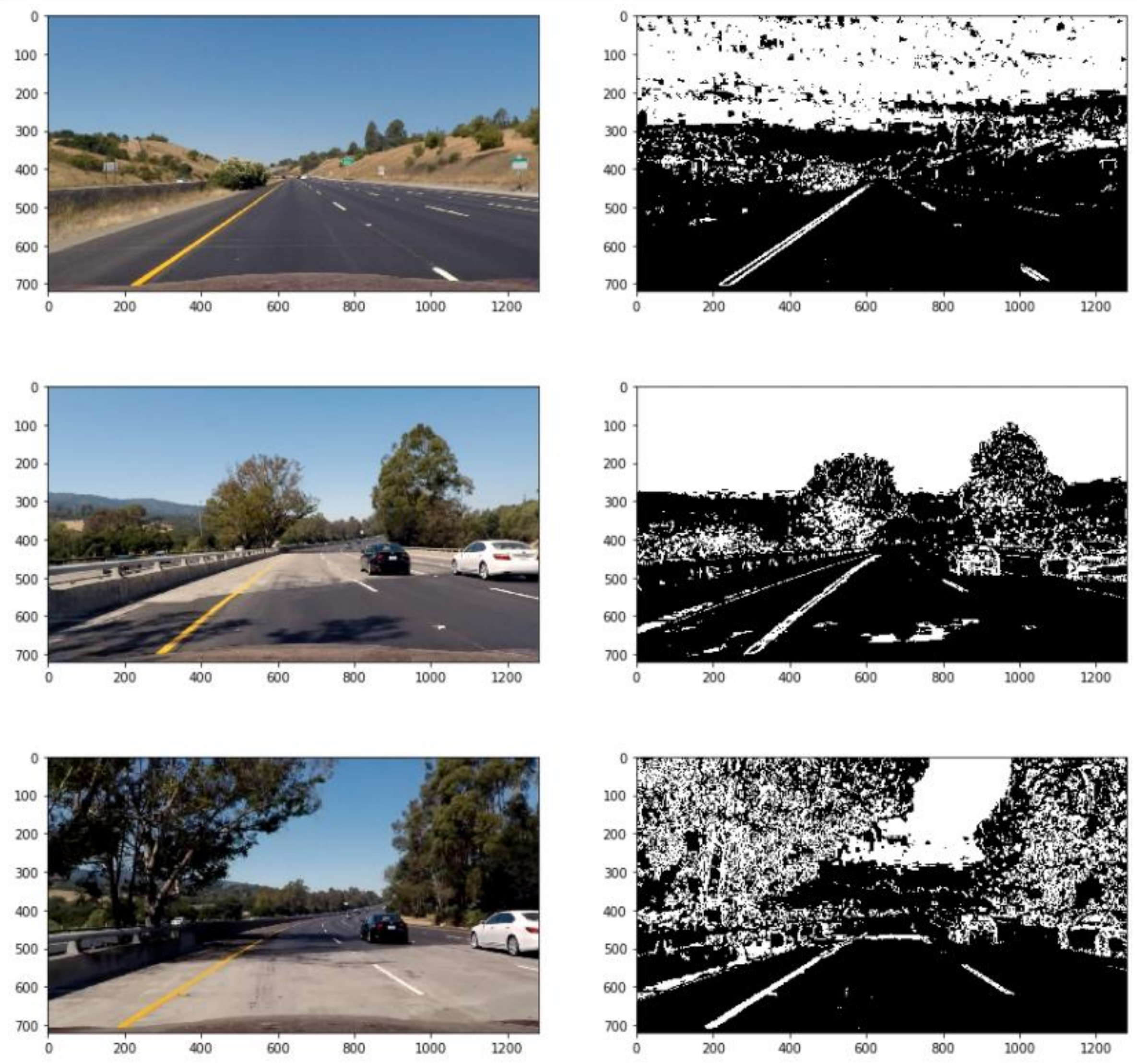
- Original image is converted to gray scale using bgr2gray()
- HLS transformation analysis
- HSV transformation analysis



The direction gradient is also applied on these transformations basically and finally a pipeline is defined called as BinaryPipeline()

```
#Step 3 : Use color transforms, gradients, etc., to create a thresholded binary image.
#From the above image transforms analysis the pipeline is as followed with thresholds for different transf
orms

from CameraCalib import *

def BinaryPipeline(img):
    hlsLowThr = 40
    hlsUpThr  = 255
    hsvLowThr = 40
    hsvUpThr  = 255
    sThrUp = 240
    sThrLw = 100
    sobelKernel = 9
    hls=bgr2hls(img) # HLS transformation
    hsv=bgr_hsv(img) # HSV transformation

    #compute binary image based on color threshold on S channel
    hlsBin = np.zeros_like(hls[:,:,2])
    #hlsBin[(normstreet(hls[:,:,2]) >= 110) & (normstreet(hls[:,:,2]) <= 240)] = 1
    hlsBin[((hls[:,:,2]) >= sThrLw) & ((hls[:,:,2]) <= sThrUp)] = 1
    binary = np.zeros_like(hlsBin)

    bHLS, sHLS=Sobel(hls[:,:,2], sobel_kernel=sobelKernel, alpha=0, thresh=(hlsLowThr, hlsUpThr))
    bHSV, sHSV=Sobel(hsv[:,:,2], sobel_kernel=sobelKernel, alpha=0, thresh=(hsvLowThr, hsvUpThr))
    binary[(binary==1) | (bHLS==1) | (bHSV==1)] = 1
    binary[(binary==1) | (hlsBin==1)] = 1
    return binary
```

The thresholds are manually adjusted by trial and error and checking which could suit the considerable values.
This BinaryPipeline is applied on set of test images and illustrated below (also in my jupyter notebook)



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

I tried to have two possibilities for src and dst points. One was manually hardcoded and other was based on height and width of image. Both can be found in my notebook.

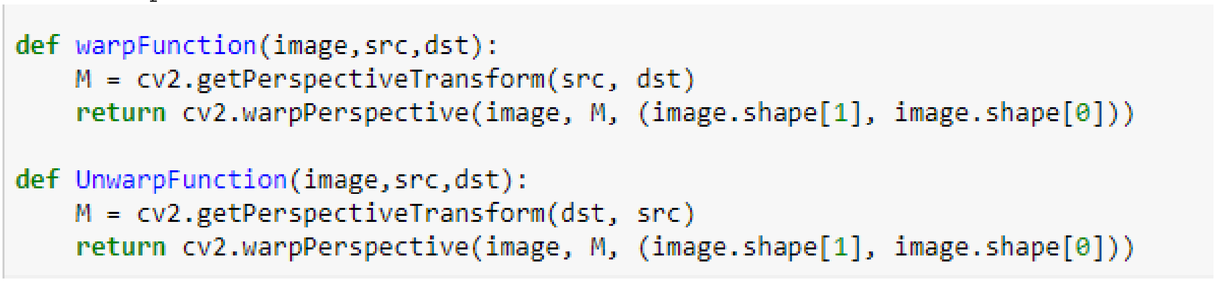
```
def srcdst():
    height =720
    src=np.float32([[596, 450], [686, 450], [1027, height-50], [276, height-50]])
    offset=100
    dst = np.float32([[305, offset], [1005, offset], [1005, height-1],[305, height-1]])
    return src,dst

def SrcDst(height,width):
    #height, width = image.shape[0], image.shape[1]
    output_size = height/2
    top=70
    bottom=370
    src = np.float32([[(width/2) - top, height*0.65], [(width/2) + top, height*0.65], [(width/2) + bottom,
 height-50], [(width/2) - bottom, height-50]])
    dst = np.float32([[(width/2) - output_size, (height/2) - output_size], [(width/2) + output_size, (heig
ht/2) - output_size], [(width/2) + output_size, (height/2) + output_size], [(width/2) - output_size, (heig
ht/2) + output_size]])
    return src,dst
```

This resulted in the following source and destination points:

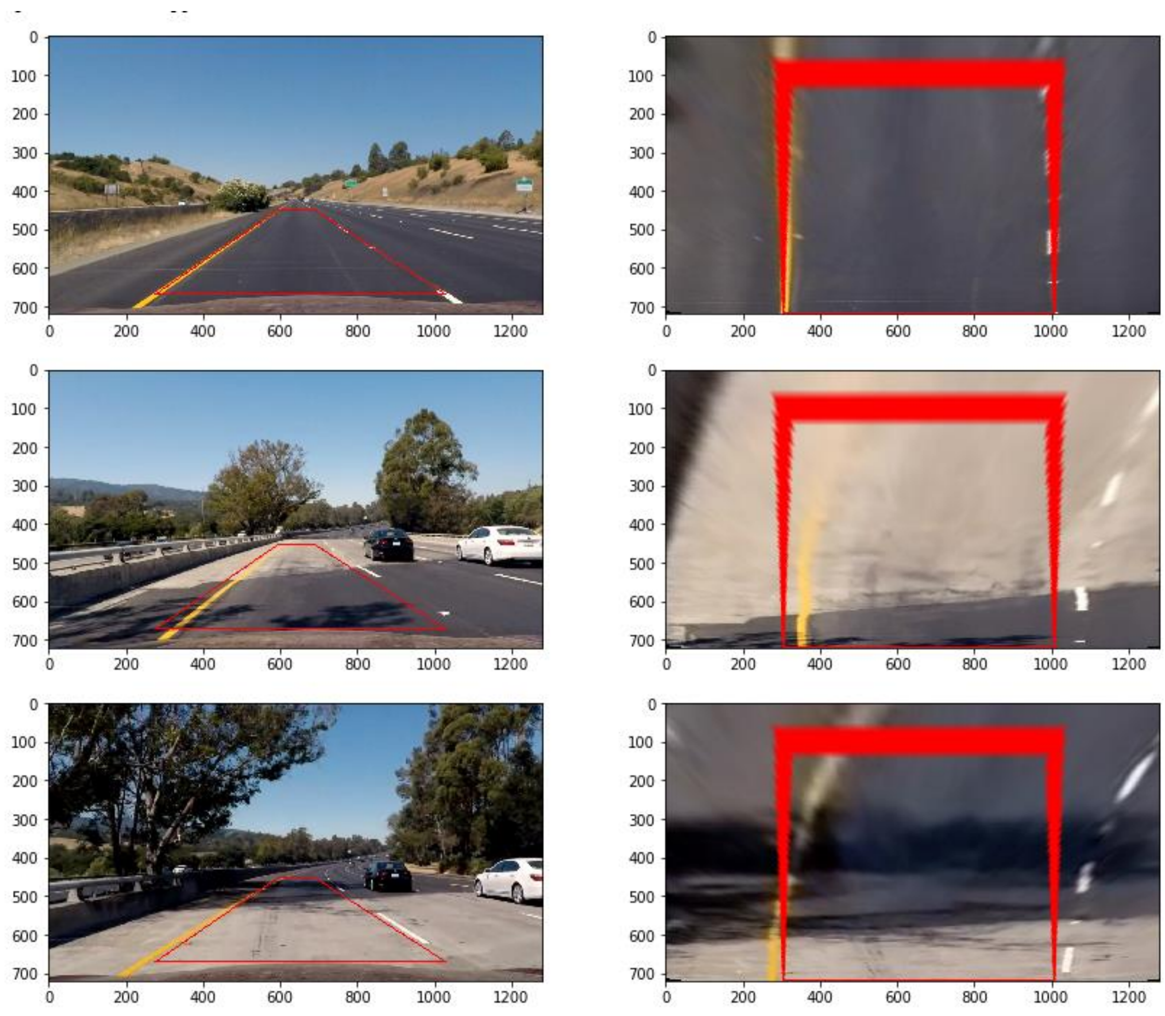
Source	Destination
596, 450	305, 100
686, 450	1005,100
1027,670	1005, 719
276, 670	305, 719

This is applied on the test images and is verified as follows with Perspective transformation.

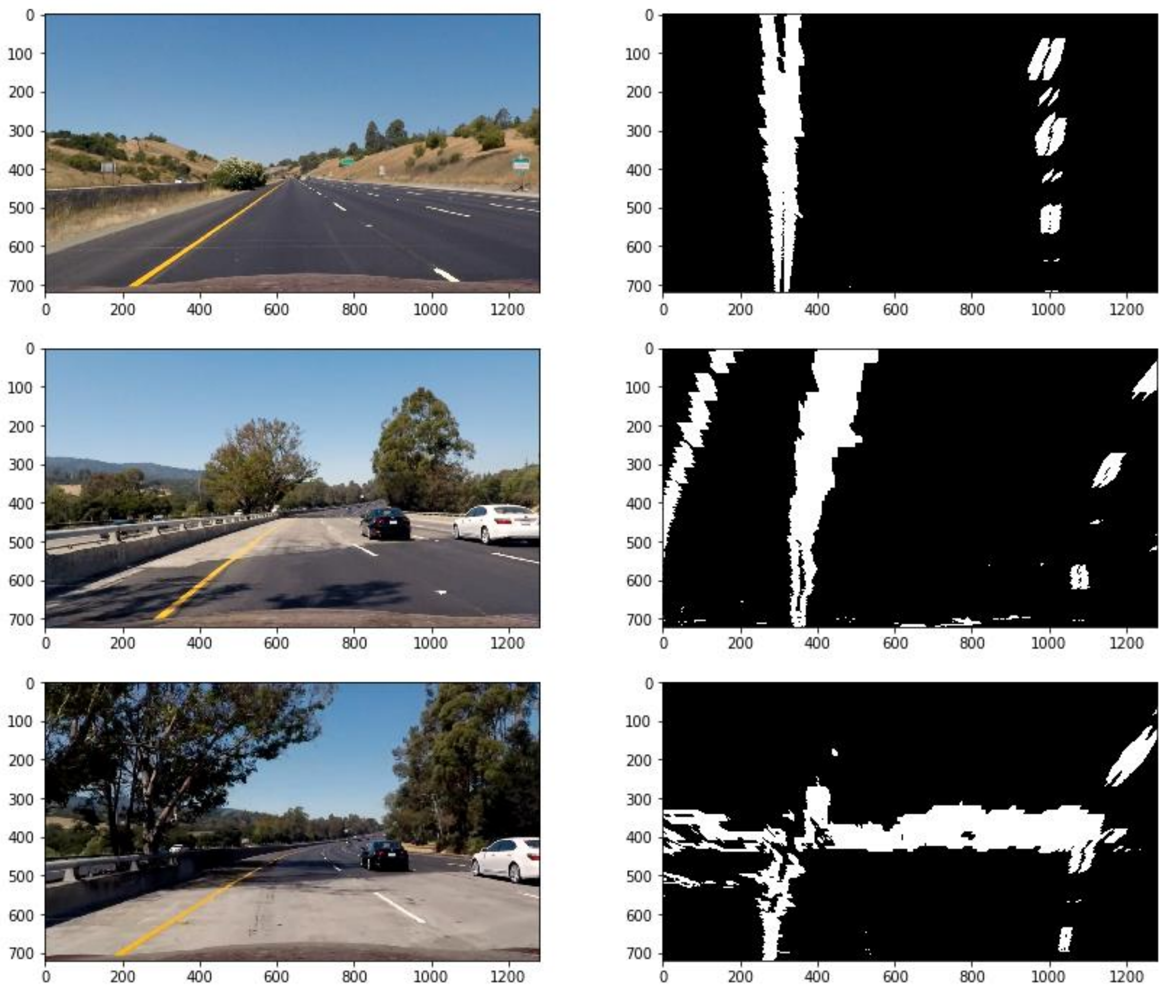
- Trapezoidal shape is chosen with src points
- Perspective transformation is applied using function warpFunction, also UnwarpFunction()



```python
def warpFunction(image,src,dst):
    M = cv2.getPerspectiveTransform(src, dst)
    return cv2.warpPerspective(image, M, (image.shape[1], image.shape[0]))

def UnwarpFunction(image,src,dst):
    M = cv2.getPerspectiveTransform(dst, src)
    return cv2.warpPerspective(image, M, (image.shape[1], image.shape[0]))
```



A test on binary images was also carried out how warpFunction() would work or the perspective transformation would look. This is illustrated as below :

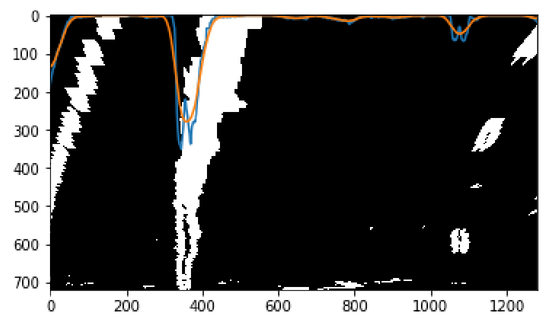


#### 4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

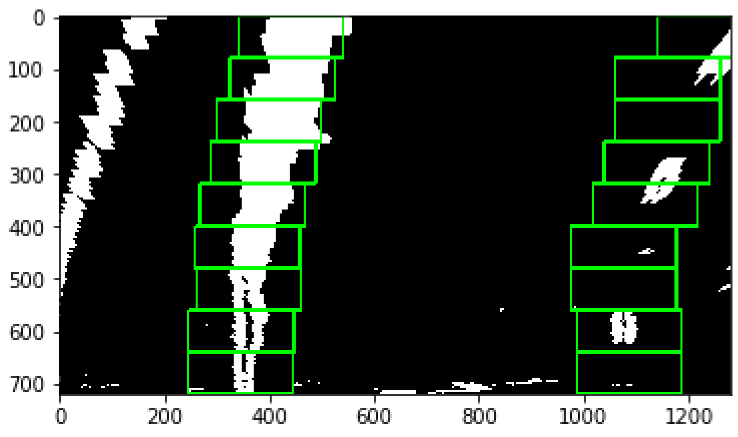
The lane lines are identified with the threshold on the histogram of binary image (perspective transformation applied). This is illustrated below:
I applied histogram on half image basically, but one could also apply on full image.

```
: #Step 5: Detect lane pixels and fit to find the lane boundary.
  #Apply the histogram as described in the lesson.
  #A part of the code is taken from the lecture.
  import scipy.ndimage as ndimage

  #TestImages : Straight_lines1, test4, test5 are used here
  index = (0,5,6)
  src,dst = srcdst()
  IMG1 = undistTestImgs[index[1]]
  binaryIMG1 = BinaryPipeline(IMG1)
  warpedIMG1 = warpFunction(binaryIMG1,src,dst)
  plt.imshow(warpedIMG1, cmap='gray')

  #One can take the histogram at lower part of image only the half part.
  histogramIMG1 = np.sum(warpedIMG1[360:,:], axis=0)
  #histogramIMG1 = np.sum(warpedIMG1, axis=0)
  plt.plot(histogramIMG1)

  #Gaussian filter for smoothing.
  histogramIMG1Filt = ndimage.gaussian_filter(histogramIMG1, sigma=14, order=0)
  plt.plot(histogramIMG1Filt)
```

```
: [<matplotlib.lines.Line2D at 0x1a441b70>]
```

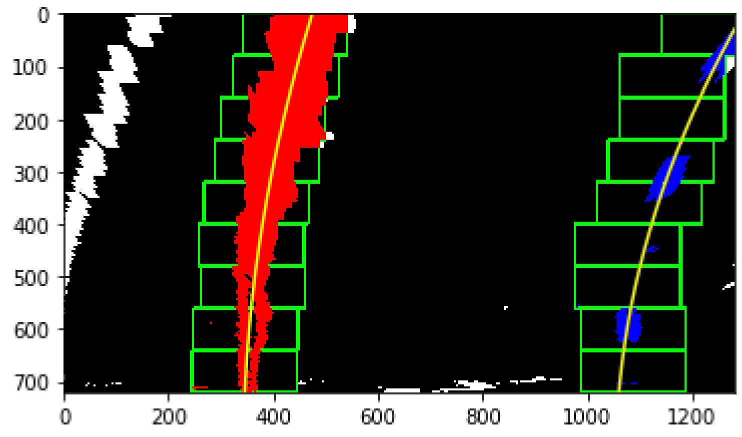


After identification, the iteration process to draw the windows on the lanes is present in findLanes() function basically but doesn't draw on the image. The loop iteratively recenters the window on the lanes and pixels are identified which are present inside the window. Most part is taken from the lecture notes.
The output :



Identifying lane lines pipeline :

- Lanes are detected using histogram
- Applying window to distinguish non zero lane pixels from other non-zero pixels. Pixel positions are added into the list.
- Second order polynomial fit (numpy.polyfit) is applied to get lane slope and curvature



#### 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.

findLanes() gives the coefficients A,B,C. position of the lane is given by

$$f(y) = Ay^2 + By + C$$

This allows direct computation of lane curvature radius 'R' in meters. Curvature is evaluated at the postion of the car. Y_eval is taken 720.

The values for the unit conversion are chosen as

- 3.7m per 700 pixels in x direction (default value from the lesson, matches very nicely with the straight lanes image)
- 3m per 50 pixels in y direction (estimated from looking at the warped color image of the straight lane, the dashed lane lines are assumed to be 3 meters long).

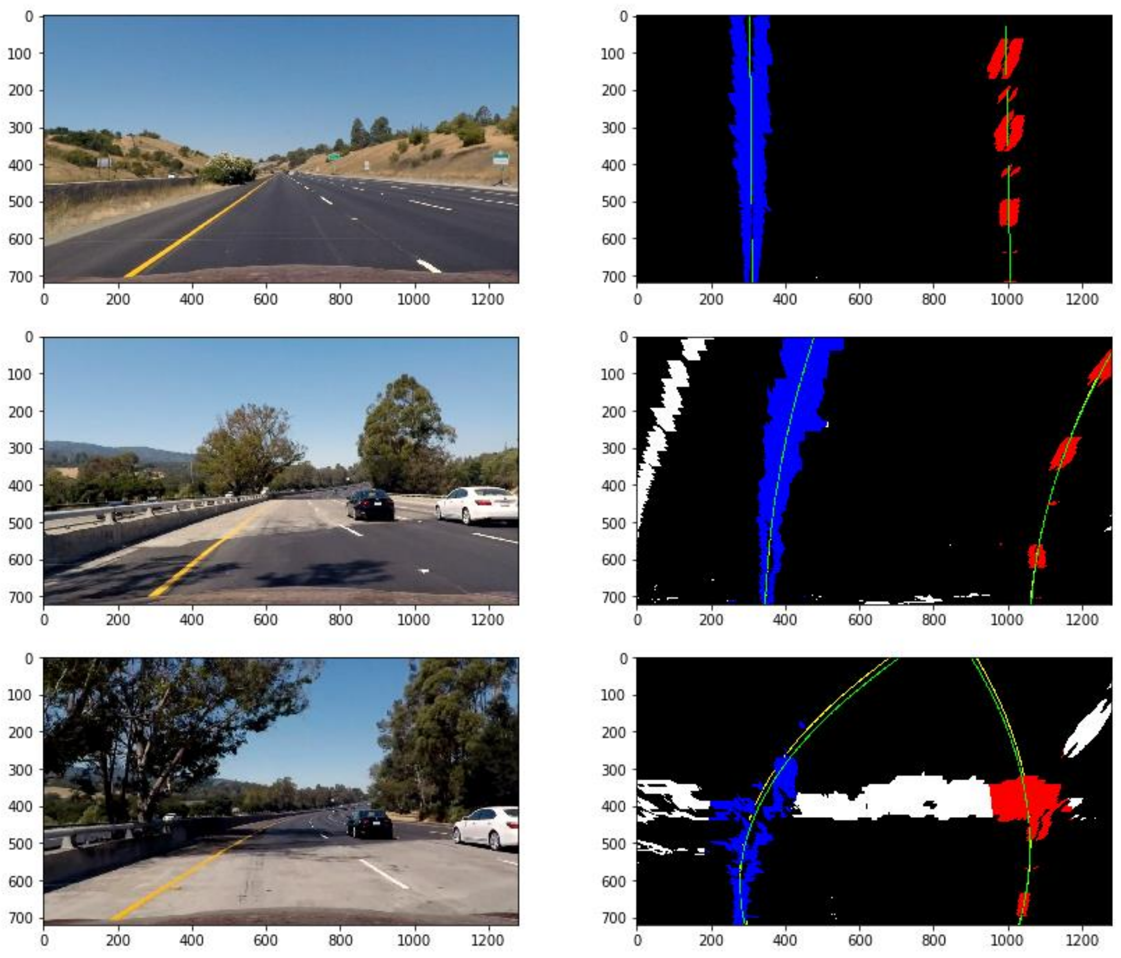
The position of the vehicle in meters, relative to the center is also computed. I have to look for the difference of the midpoint of the lane from the center of the image.

In the end, I have lane distance, car position relative to the midpoint of lane, left and right lane curvature, left and right angles. These are enough to determine the two 2nd order polynomials describing the lanes.

As explained in previous step. From notebook

findLanes() -- Finds the lanes using windows drawLanes() -- Lane finding with windows drawn drawLanesW()-- Explained in later part. Illustration of the lane finding (without the windows being drawn). findLanesReuse() -- uses polynomial coefficients from the previous step, as described in the lecture. The coefficients for the second order polynomial are computed here with a unit conversion from pixels to meters, which is required for the computation of curvature in meters.

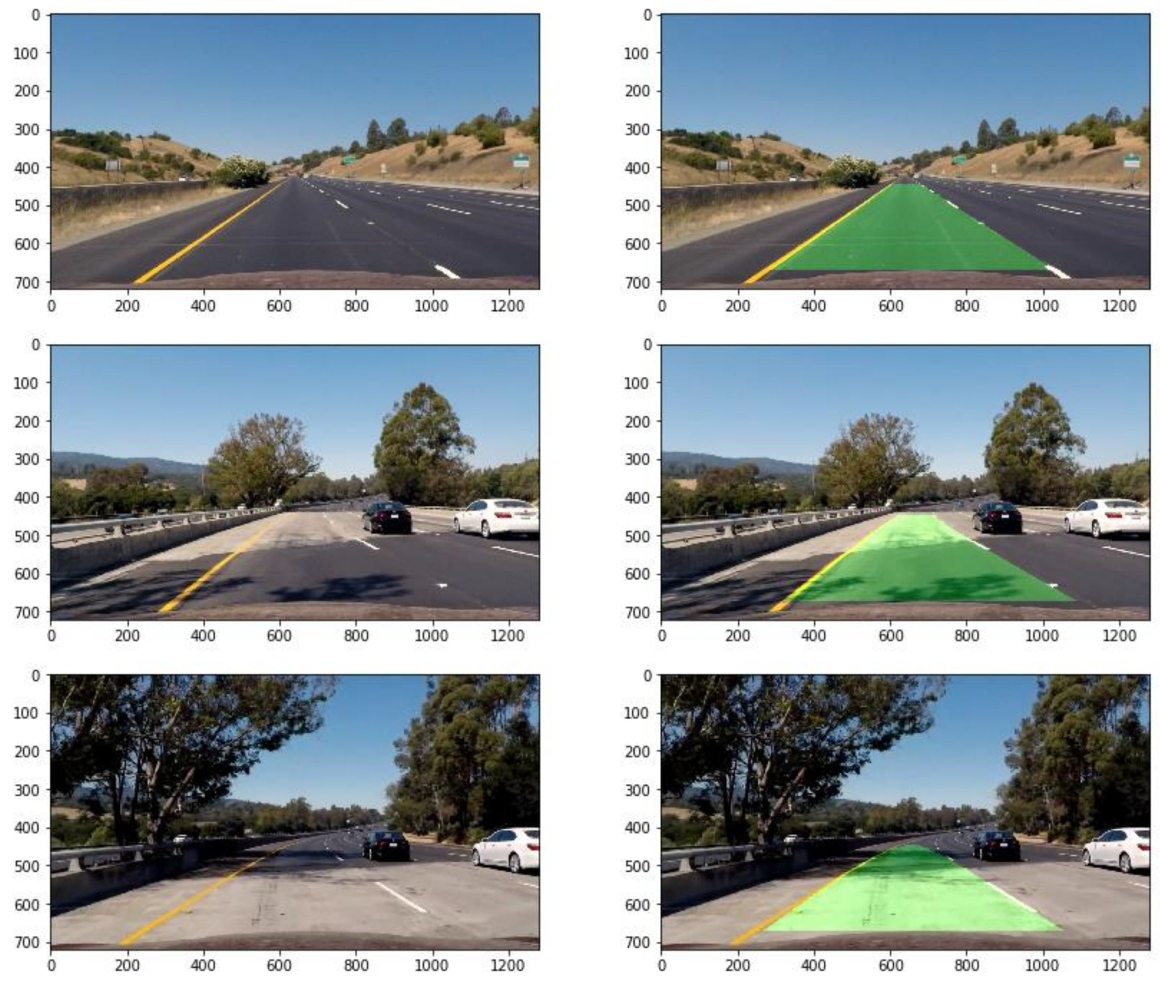
Pipeline applied on sample test images:



The third image looks bad, but can be improved with previous frame information.

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

The pipeline is applied for visualization and part of the code for visualization is written in drawLanes() function.



### Pipeline (video)

#### 1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

It is present in the github repository TestVideoOutput.mp4

I do not know why the 'TAB' display of %video processing ran downwards before submission, but the %video processing showed horizontal when I ran for the first time! Hence you see a long list of % display in the notebook.

### 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?

Before processing, the properties of the lane lines are stored in a class. The properties ie. Lane distance, position of car with respect to center, left and right curvature radius, left and righ angles. There is also transformations(toParam, toCoeff, toCoeff2 [which transforms to polynomial coefficients of a polynomial with reversed y coordinate]) used. These were discussed with fellow Udacity student Mr.Alexander Braun. I found that quite useful and stated in notebook.

findLanesReuse() function looks for lane line close to polynomial from previous step which makes detection stable in situations where there are shadows, dirt on raod. There is also weighted filtering of new and old coefficients which will definitely help. I find hardcoding of minpix parameter would also increase robustness against noise. There is also re-initialize step by reverting back to findLanes() in case the highest polynomial order coefficient exceeds value 0.001. In the end, the reduction of lower threshold for saturation component which made lane lines prominent under bad lighting conditions.

Improvisation/Problems :
Vertical gradients as in challenge video, lanes with small radius would be a slight problem for this pipeline. The scope of improvement is in Image processing and in turn identification of lane lines properly.