In this project, our goal is to write a software pipeline to identify the lane boundaries in a video from a front-facing camera on a car. This project consists of 6 stages

- 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 threshold 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.

Below is the code structure

a) CameraCalibration.ipynb:- Camera calibration code using chessboard image and distortion correction on test images
b) LaneDetection.ipynb:- Lane detection algorithm
c) pipeline.py:- Pipeline used for processing the Video and test images

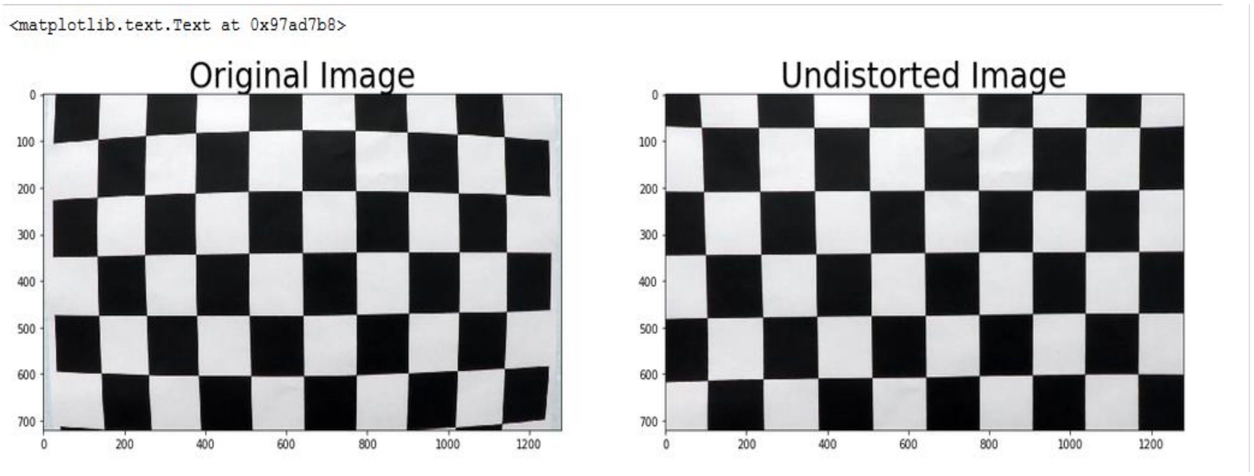
<u>Camera calibration matrix and distortion coefficients</u>

Distortion changes shape and size of these 3D objects in 2D image. Real cameras use curved lenses to form an image and light rays bend a little too much at the edges of the images. This is called radial distortion. Another type of distortion, is tangential distortion. This occurs when a camera's lens is not aligned perfectly parallel to the imaging plane, where the camera film or sensor is. This makes an image look tilted so that some objects appear farther away or closer than they are.

So, the first step in analyzing camera images, is to undo this distortion. This is done using OpenCV library APIs. Test chessboard images are used for calibrating and computing the distortion coefficients. Code for camera calibration is available in **CameraCalibration.ipynb**. We start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners. We assume the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. **objp** is just a replicated array of coordinates, and **objpoints** will be appended with a copy of it every time we 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.

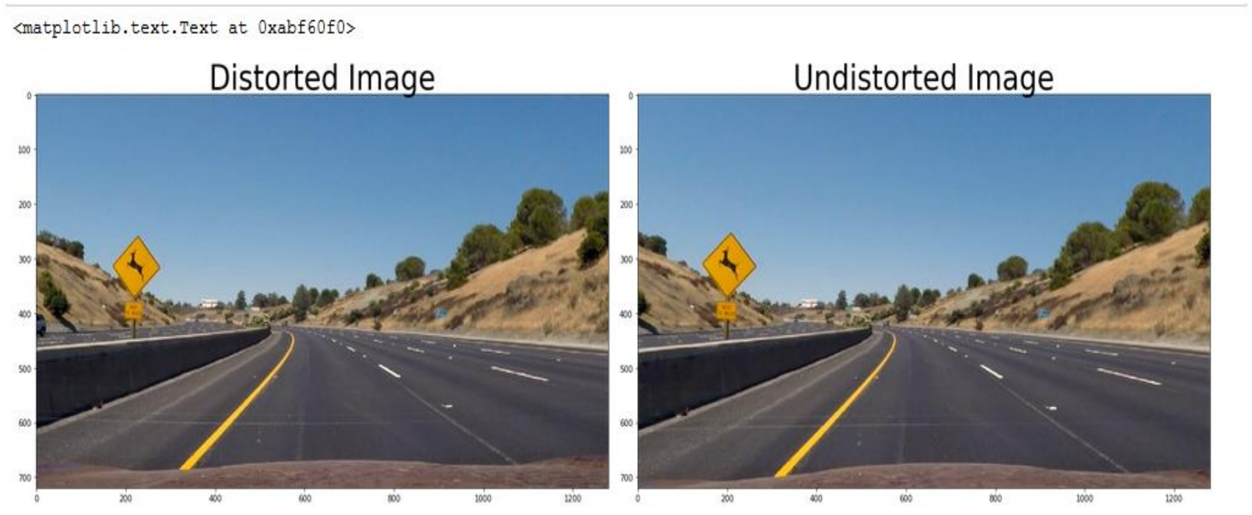
20 test images are provided in **camera_cal** directory. In Algorithm, **ny** refers to number of inner points along column and **nx** refers to number of inner points along row. Some of the test images contain less than 9*6 inner points. It searches for **ny** in range of [5,6] and **nx** in [6,7,9]

objpoints and **imgpoints** arrays are fed into cv2.calibrateCamera() function to compute the camera calibration and distortion coefficients. Next, distortion correction is done on the test image using cv2.undistort() function and obtained this result. Below is sample output.



```
<matplotlib.text.Text at 0x97ad7b8>
```

Calibration matrix and vector matrix are saved in pickle file so that they can be reused in later stages of pipeline. In **LaneDetection.ipynb** notebook, distortion correction is performed on all test images and output is saved in output_image/undistorted folder. Below is one sample output image.



```
<matplotlib.text.Text at 0xabf60f0>
```

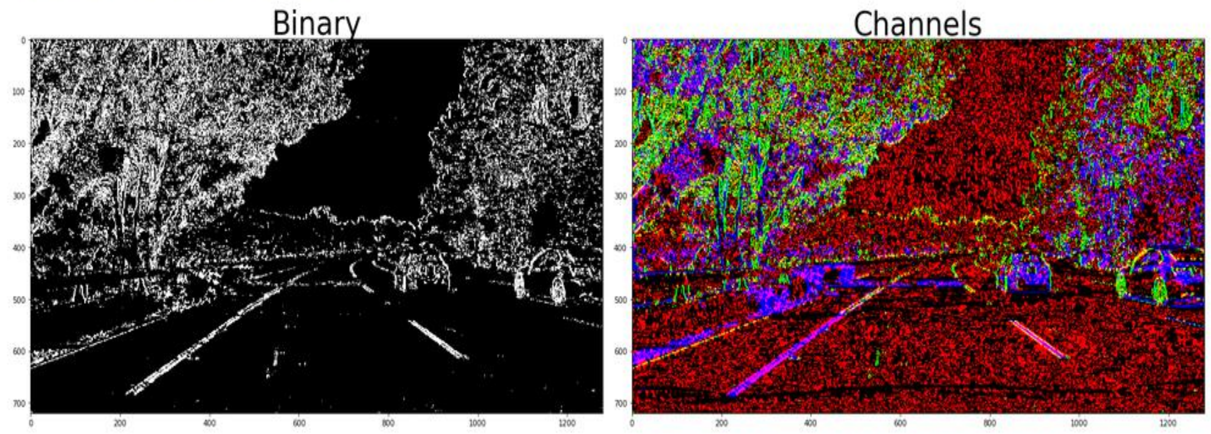
Use color transforms, gradients to create a binary image for lane detection.

To correctly detect the lane lines, below 3 techniques were used to create a binary image.

- Color space mapping - RGB to HLS and S-Channel thresholding (120-255)
- Sobel gradients along X axis - (kernel size: 9, sx_threshold=(20, 255)).
- Gradients direction thresholding – (dir_thresh=(0.7,1.3).

These 3 techniques are defined as helper functions in cell5 of **LaneDetection.ipynb**. binarize() function calls into these 3 helper functions to generate a binary image where lanes are more clearly visible. Code is available in cell3 of the pipeline2.ppyd notebook.Below is sample output of binarize () function on test image.

```
<matplotlib.text.Text at 0x13a766d8>
```

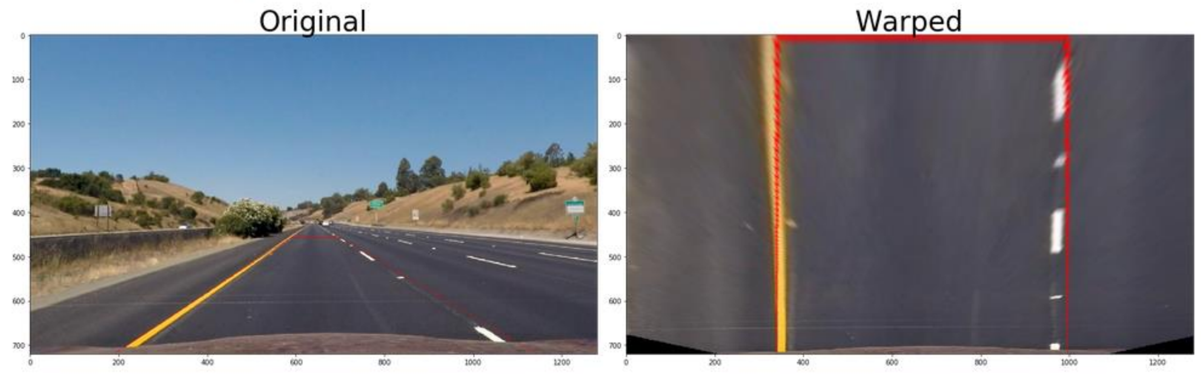


Apply a perspective transform to rectify binary image (“birds-eye view”).

Next, perspective transform is applied on the binary threshold image to transform from real world image space to the bird's eye space. The bird's eye space is useful as it allows us to treat the binary image pixels as dots and fit them with a particular 2nd order polynomial. In this case, we assume the road is a flat plane. We pick four points in a trapezoidal shape (similar to region masking) that would represent a rectangle when looking from a bird’s eye space. We choose 4 points in the original image (two points at the bottom of the image lane lines, two at the top of the lane lines) and map them to a square in the bird's eye space. Next, cv2.warpPerspective() function is used to transform the original image and generate the bird's eye perspective.

Cell 7,8 of LaneDetection.ipynb notebook implements the bird's eye transformation. In pipeline.py, hardcoded points are used for perspective transform. The figure below depicts the output of the bird's eye transformation.

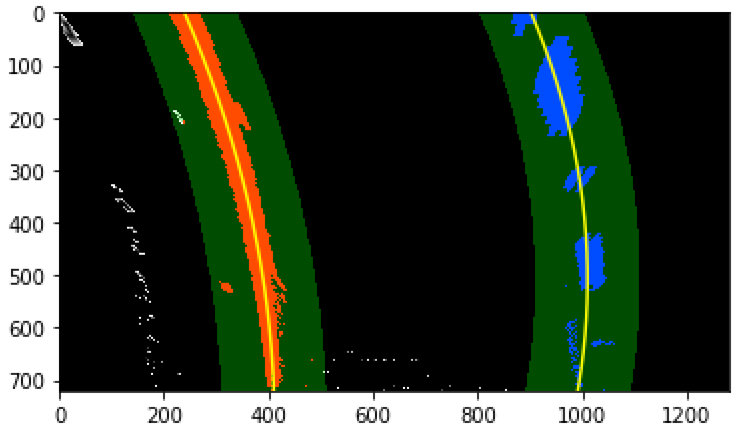
```
<matplotlib.text.Text at 0x16e42668>
```



Detect lane pixels and fit to find the lane boundary.

After applying calibration, thresholding, and a perspective transform to a road image, we should have a binary image where the lane lines stand out clearly. Now, we will use histogram peak detection method to identify lanes in the image. With this histogram, we are adding up the pixel values along each column in the image. In the binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. We can use that as a starting point for where to search for the lines. From that point, we can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

Once the lane lines are identified with two sets of (x,y) pairs, we can fit them using a 2nd order polynomial in the form f(y) = A*y^2 + B*y + C. This mathematical form will allow us to better identify the curve radius. Cell 13,14 of LaneDetection.ipynb contains the implementation details of this algorithm. Below is the sample output



Determine the curvature of the lane and vehicle position with respect to center.

Self-driving cars need to compute correct steering angle to turn, left or right. To calculate the radius of the curvature, we use a simple technique that computes analytically the radius of a circle that is tangent to a function in a given point. The method relies on the first and second order derivative of the function and defines the curve radius as follows:

$R = (1/|2A|) * (1 + (2Ay + B)^2)^3/2$

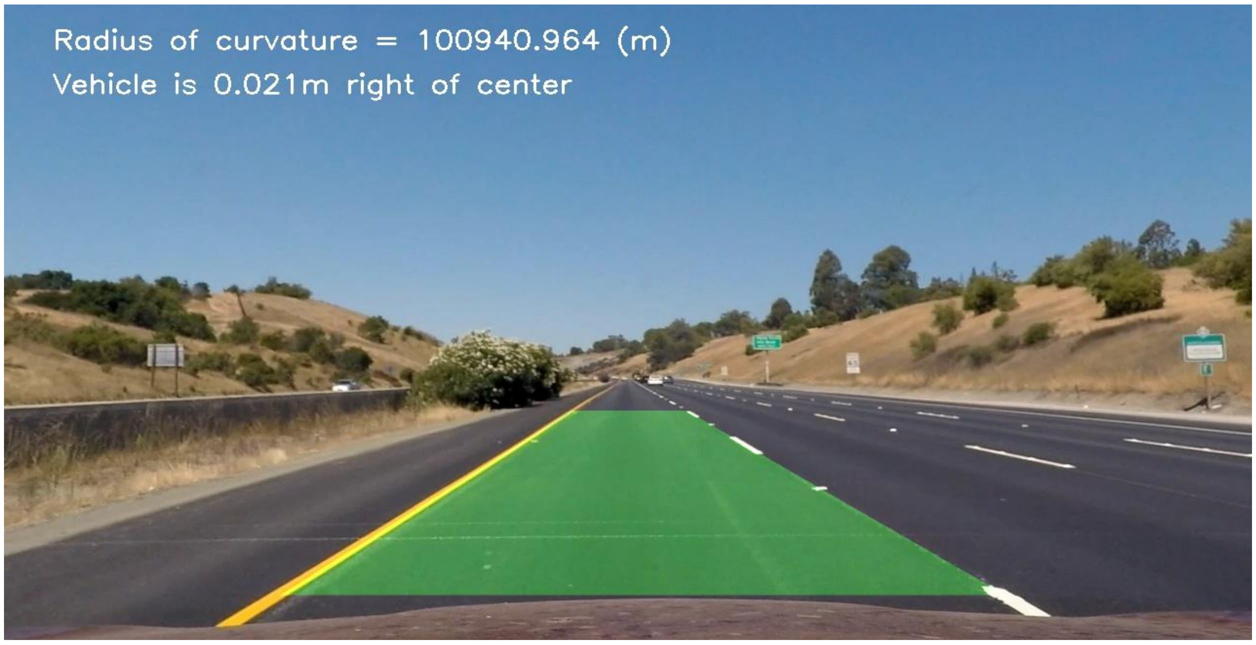
In addition, we calculate an estimation of the car position on the lane. First, we determine the lane center by calculating the difference between the position of the right and the left lines. Next, we determine the center of the car which is the midpoint of the image (because the camera in mounted on the car). The position of the car on the lane is given by the difference between the center of the lane and the center of the car.

All these calculations are made in a *pixel space*. Therefore, to give a meaning to the values we need to transform them to the real-world space. we can use the following constants:

- ym_per_pix = 30/720 # meters per pixel in y dimension
- xm_per_pix = 3.7/700 # meters per pixel in x dimension

which maps the number of meters per pixel in the x and y dimensions.

Cell14,15 of the LaneDetection.ipynb notebook has the implementation details of calculating the curvature of lane. Detected lanes are warped back onto original image and curvature, position of car are printed on top of the image. Pipeline_test() function execute the lane detection pipeline on test images. Below is one sample output.



```
Radius of curvature = 100940.964 (m)
Vehicle is 0.021m right of center
```

Discussion

This project was a bit challenging when it comes to perspective transform and making it work on video stream. It required lot of trials as multiple parameters have to be modified. (gradient thresholds, corners for perspective transform). Pipeline worked fine on project video.

Below are some possible improvements

- Use some CNN or Deep neural network to calculate the position of the car.
- Use higher order polynomial for fitting the lane
- Use dynamic selection of masks (gradient thresholds, color space transformations)