Advanced_Lane_Lines

March 1, 2018

1 Advanced Lane Lines

From a car driving video, each frame can be processed to detect the lane lines in the video. The following techniques will be used:

1. Camera Calibration
2. Binary Threshold Images
3. Perspective Transform

With this, a bird's eye view of the road can be used to detect the lane lines. Information from past frames will also be used to make the lane line detection more robust.

```
In [345]: import numpy as np
          import pandas as pd
          import cv2
          import os
          from tqdm import tqdm
          import matplotlib.pyplot as plt
          from matplotlib.pyplot import imread
          import glob
          from scipy.misc import imsave
          import matplotlib.mlab as mlab
          from moviepy.editor import *
          from IPython.display import HTML
```

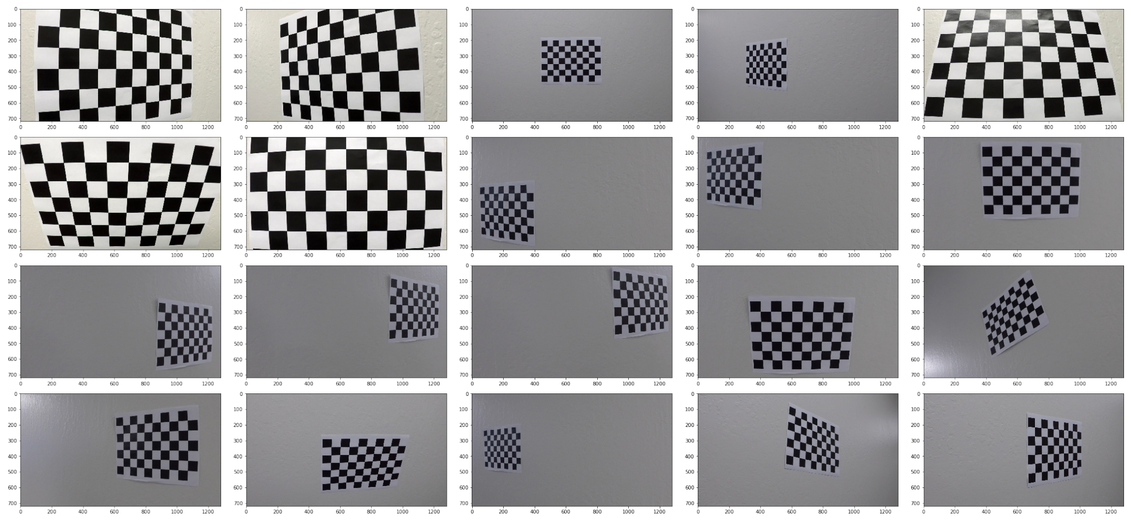
```
In [262]: from basic_lane_lines import weighted_img, find_lane3
          from util import *
```

```
In [263]: data_dir = 'data'
```

2 Camera Calibration

First, the camera needs to be calibrated to correct for distortions with the lens. Here, we will use some images of checkboards to calibrate the camera.

```
In [264]: camera_cal_dir = 'camera_cal'
In [265]: def load_camera_cal_images(camera_cal_dir):
              img_files = glob.glob('%s/*.jpg' % camera_cal_dir)
              return [imread(img_file) for img_file in img_files], img_files

In [266]: camera_cal_img_arr, camera_cal_img_files = load_camera_cal_images(camera_cal_dir)

In [267]: display_images(camera_cal_img_arr)
```



Save one of the image for testing later.

```
In [268]: test_img = camera_cal_img_arr[6]
          camera_cal_img_arr = [camera_cal_img_arr[i] for i in range(len(camera_cal_img_arr)) i
          camera_cal_img_files = [camera_cal_img_files [i] for i in range(len(camera_cal_img_fi
```

The objpoints and imgpoints will be used for unwarping video frames later .

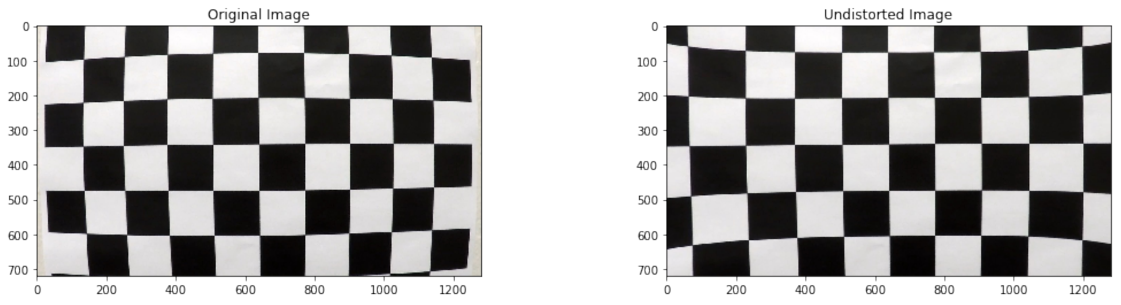
```
In [269]: objpoints = []
          imgpoints = []

          # termination criteria
          criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

          for img in tqdm(camera_cal_img_arr):
              x_cnt = 9
              y_cnt = 6

              objp = np.zeros((y_cnt*x_cnt,3), np.float32)
              objp[:,:2] = np.mgrid[0:x_cnt,0:y_cnt].T.reshape(-1,2)

              gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

```
            ret, corners = cv2.findChessboardCorners(gray, (y_cnt, x_cnt), None)

            if ret:
                objpoints.append(objp)
                cv2.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
                imgpoints.append(corners)

        ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[
```

```
100%|| 19/19 [00:04<00:00,  3.87it/s]
```

```
In [270]: def cal_undistort(img, objpoints, imgpoints):
              # Use cv2.calibrateCamera() and cv2.undistort()
              gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
              ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
                                                                  gray.shape[::-1],
                                                                  None, None)

              undist = cv2.undistort(img, mtx, dist, None, mtx)

              return undist
```

```
In [271]: undistorted_img = cal_undistort(test_img, objpoints, imgpoints)

          plt.figure(figsize=(18,4))

          ax = plt.subplot(1,2,1)
          ax.set_title('Original Image')
          plt.imshow(test_img)
          ax = plt.subplot(1,2,2)
          ax.set_title('Undistorted Image')
          plt.imshow(undistorted_img)
          plt.show()
```



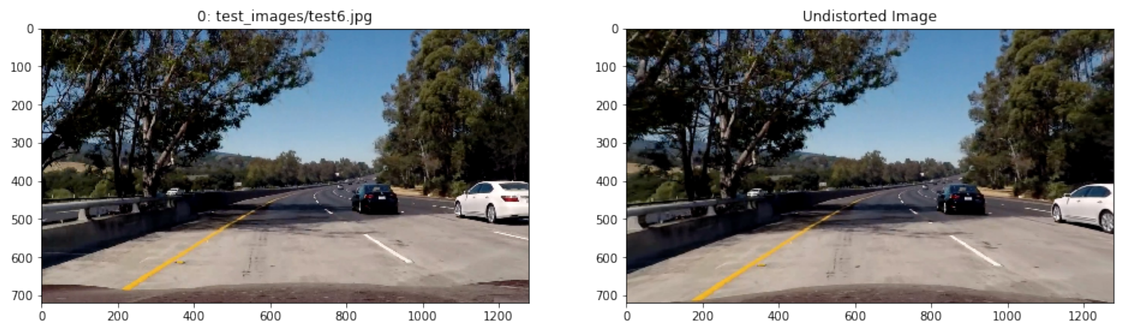
The original test image and the undistorted image using cal_undistort() are shown above.

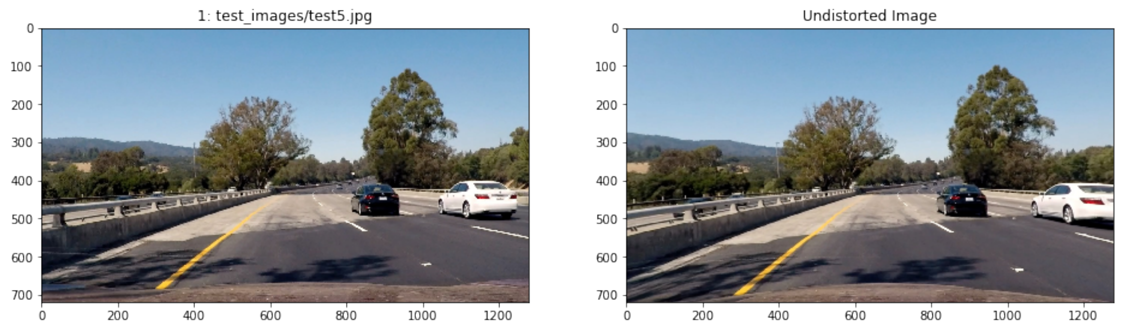
2.1 Undistorting Video Frames

```
In [272]: def load_test_images(test_img_dir):
              img_files = glob.glob('%s/*.jpg' % test_img_dir)
              return [imread(img_file) for img_file in img_files], img_files
```

```
In [273]: test_image_arr, test_img_files = load_test_images('test_images')
```

```
In [274]: for i,img in enumerate(test_image_arr[1:3]):
              undistorted_img = cal_undistort(img, objpoints, imgpoints)

              plt.figure(figsize=(16,16))

              ax = plt.subplot(1,2,1)
              ax.set_title("%d: %s" % (i, test_img_files[i]))
              plt.imshow(img)
              ax = plt.subplot(1,2,2)
              ax.set_title('Undistorted Image')
              plt.imshow(undistorted_img)
              plt.show()
```





Here are some test images with cal_undistort() applied to them. Two noticeable differences with the undistorted images are:

1. The dashboard on the bottom right has a similar curvature to the bottom left.
2. The white dashed lane lines on the right have a smoother curve.

3 Binary Thresholding

For finding the lane line, we will create a binary threshold image from the original image using various threshold methods. These images will be calculating the lane fits for the left and right lanes.

3.1 HLS Thresholds

The images will be converted from RGB to HLS. Only thresholds from the saturation and hue layers will be used.

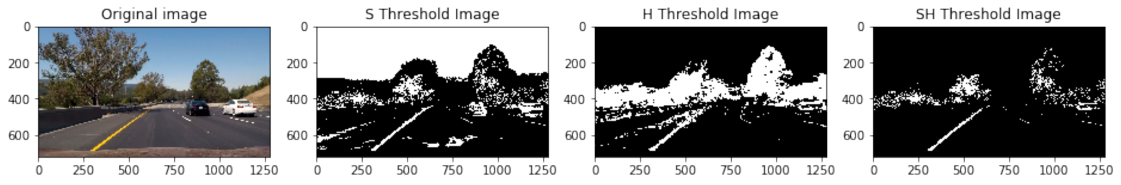
```
In [275]: def make_S_threshold_img(img, thresh = (90, 255)):
              hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
              S = hls[:,:,2]

              binary = np.zeros_like(S)
              binary[(S > thresh[0]) & (S <= thresh[1])] = 1

              return binary

          def make_H_threshold_img(img, thresh = (15, 100)):
              hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
              H = hls[:,:,0]

              binary = np.zeros_like(H)
              binary[(H > thresh[0]) & (H <= thresh[1])] = 1

              return binary

          def make_SH_threshold_img(img, S_thresh = (90, 255), H_thresh = (15, 100)):
              S_threshold_img = make_S_threshold_img(img, thresh = S_thresh)
              H_threshold_img = make_H_threshold_img(img, thresh = H_thresh)

              combined_thresholds_img = np.logical_and(S_threshold_img, H_threshold_img)

              return 255*combined_thresholds_img.astype(np.uint8)
```

```
In [276]: test_img = test_image_arr[0]
```

```
In [277]: test_S_threshold_img = make_S_threshold_img(img, thresh = (90, 255))
          test_H_threshold_img = make_H_threshold_img(img, thresh = (15, 100))
          test_SH_threshold_img = make_SH_threshold_img(img, S_thresh = (90, 255), H_thresh =
```

```
In [278]: plt.figure(figsize=(16,16))

          ax = plt.subplot(1,4,1)
          ax.set_title('Original image')
          plt.imshow(test_img)
          ax = plt.subplot(1,4,2)
```

```
        ax.set_title('S Threshold Image')
        plt.imshow(test_S_threshold_img, cmap='gray')
        ax = plt.subplot(1,4,3)
        ax.set_title('H Threshold Image')
        plt.imshow(test_H_threshold_img, cmap='gray')
        ax = plt.subplot(1,4,4)
        ax.set_title('SH Threshold Image')
        plt.imshow(test_SH_threshold_img, cmap='gray')
        plt.show()
```

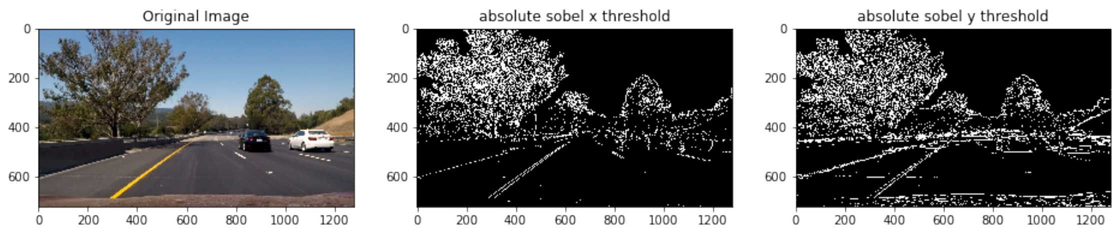


Above is a comparison of the original image, the saturation and hue threshold images and the saturation and hue combined (logical AND). The combination of saturation and hue has less noise compared to either of them separately.

3.2 Gradient Thresholds

```
In [279]: def abs_sobel_thresh(img, orient='x', thresh_min=0, thresh_max=255):
              # Convert to grayscale
              gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
              # Apply x or y gradient with the OpenCV Sobel() function
              # and take the absolute value
              if orient == 'x':
                  abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 1, 0))
              if orient == 'y':
                  abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 0, 1))
              # Rescale back to 8 bit integer
              scaled_sobel = np.uint8(255*abs_sobel/np.max(abs_sobel))
              # Create a copy and apply the threshold
              binary_output = np.zeros_like(scaled_sobel)
              # Here I'm using inclusive (>=, <=) thresholds, but exclusive is ok too
              binary_output[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

              # Return the result
              return binary_output

In [280]: abs_sobel_thresh_x_sample_video_img = abs_sobel_thresh(test_img, orient='x', thresh_
          abs_sobel_thresh_y_sample_video_img = abs_sobel_thresh(test_img, orient='y', thresh_

In [281]: i = 1
          cols = 3
```

```
        rows = 1

        plt.figure(figsize=(16,16))

        ax = plt.subplot(rows,cols,i)
        ax.set_title('Original Image')
        plt.imshow(test_img, cmap='gray')

        i += 1
        ax = plt.subplot(rows,cols,i)
        ax.set_title('absolute sobel x threshold')
        plt.imshow(abs_sobel_thresh_x_sample_video_img, cmap='gray')

        i += 1
        ax = plt.subplot(rows,cols,i)
        ax.set_title('absolute sobel y threshold')
        plt.imshow(abs_sobel_thresh_y_sample_video_img, cmap='gray')

        plt.show()
```

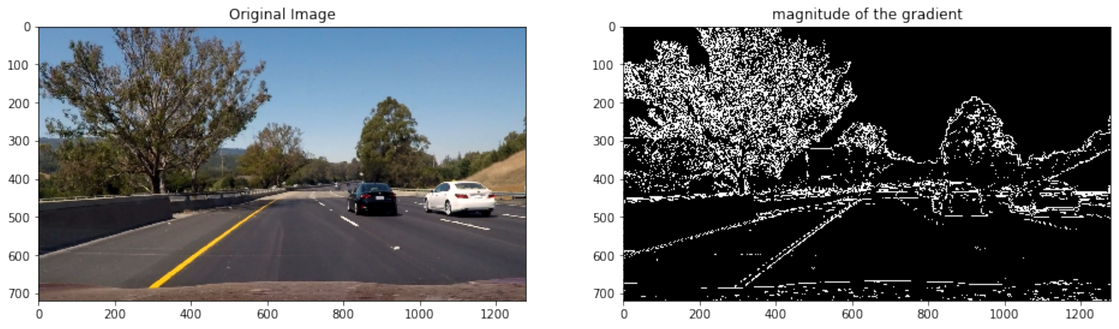


Here the Sobel filter is being used to find gradient thresholds in the x and y directions.

```
In [282]: def mag_thresh(img, sobel_kernel=3, mag_thresh=(0, 255)):
              # Convert to grayscale
              gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
              # Take both Sobel x and y gradients
              sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
              sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
              # Calculate the gradient magnitude
              gradmag = np.sqrt(sobelx**2 + sobely**2)
              # Rescale to 8 bit
              scale_factor = np.max(gradmag)/255
              gradmag = (gradmag/scale_factor).astype(np.uint8)
              # Create a binary image of ones where threshold is met, zeros otherwise
              binary_output = np.zeros_like(gradmag)
              binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])] = 1

              # Return the binary image
              return binary_output
```

```
In [283]: mag_thresh_sample_video_img = mag_thresh(test_img, mag_thresh=(30, 100))
```

```
In [284]: plt.figure(figsize=(16,16))

          ax = plt.subplot(1,2,1)
          ax.set_title('Original Image')
          plt.imshow(test_img, cmap='gray')

          ax = plt.subplot(1,2,2)
          ax.set_title('magnitude of the gradient')
          plt.imshow(mag_thresh_sample_video_img, cmap='gray')

          plt.show()
```



Using the x and Sobel gradients, we can also calculate the magnitude of the gradients.

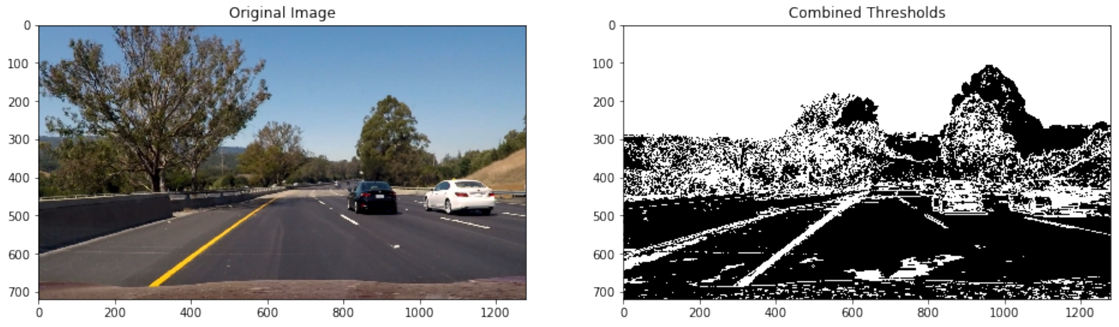
3.3 Combined Thresholds

```
In [285]: def combine_thresholds(abs_sobel_x_thresh_img, abs_sobel_y_thresh_img,
                                 mag_thresh_img, SH_threshold_img):
              combined_thresholds_img = np.logical_or(abs_sobel_x_thresh_img, abs_sobel_y_thres
              combined_thresholds_img = np.logical_or(combined_thresholds_img, mag_thresh_img)
              combined_thresholds_img = np.logical_or(combined_thresholds_img, SH_threshold_im

              return 255*combined_thresholds_img.astype(np.uint8)
```

```
In [286]: combine_thresholds_sample_video_img = combine_thresholds(abs_sobel_thresh_x_sample_vi
                                                                   abs_sobel_thresh_y_sample_vi
                                                                   mag_thresh_sample_video_img,
                                                                   test_S_threshold_img)
```

```
In [287]: plt.figure(figsize=(16,16))

          ax = plt.subplot(1,2,1)
          ax.set_title('Original Image')
          plt.imshow(test_img, cmap='gray')
```

```
ax = plt.subplot(1,2,2)
ax.set_title('Combined Thresholds')
plt.imshow(combine_thresholds_sample_video_img, cmap='gray')

plt.show()
```



For the combined binary threshold image, the following thresholds are logically OR-ed together:

1. Sobel x
2. Sobel y
3. Magnitude
4. Saturation and Hue

4 Perspective Transform

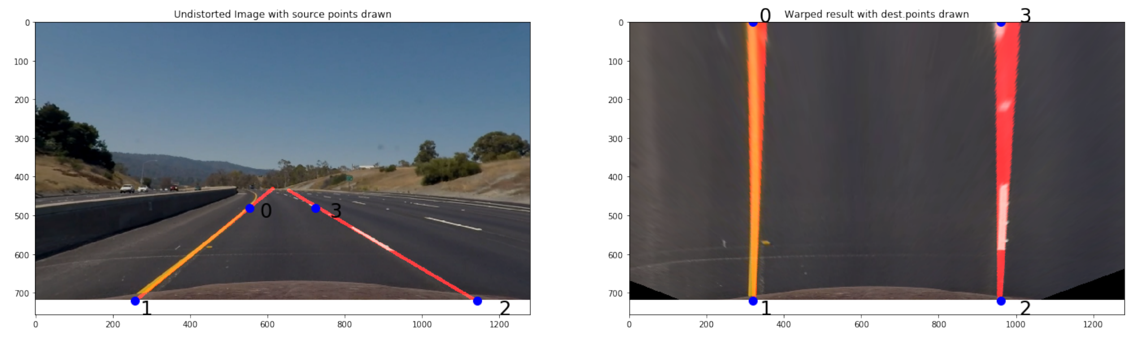
The original video frame will be transformed to a top down view. The top down view image will then be used to find the lane lines. Afterwards, the reverse tranformation will be applied to the lane lines and drawn back onto the original image.

```
In [288]: sample_video_img = imread('data/project_video/frame_0000100.jpg')
          undistorted_sample_video_img = cal_undistort(sample_video_img, objpoints, imgpoints)
          lane_line_sample_video_img = weighted_img(find_lane3(undistorted_sample_video_img), u
```

```
In [289]: def warp(img, src, dst):
              M = cv2.getPerspectiveTransform(src, dst)

              warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)

              Minv = cv2.getPerspectiveTransform(dst, src)

              return warped, Minv
```

The settings for src was set manually using the test image below. The test image was selected because it was relatively straight. The src settings were adjusted until the 4 points aligned with the tops and bottoms of the left and right lane lines in the warped image.

```
In [290]: img_size = (lane_line_sample_video_img.shape[1], lane_line_sample_video_img.shape[0]

          src = np.float32(
               [[(img_size[0] / 2) - 87, img_size[1] / 2 + 120],
               [((img_size[0] / 6) + 45), img_size[1]],
               [(img_size[0] * 5 / 6) + 75, img_size[1]],
               [(img_size[0] / 2 + 85), img_size[1] / 2 + 120]])
          dst = np.float32(
              [[(img_size[0] / 4), 0],
              [(img_size[0] / 4), img_size[1]],
              [(img_size[0] * 3 / 4), img_size[1]],
              [(img_size[0] * 3 / 4), 0]])

In [291]: warped_sample_video_img, Minv = warp(lane_line_sample_video_img, src, dst)

In [292]: plt.figure(figsize=(24,32))

          ax = plt.subplot(1,2,1)
          ax.set_title('Undistorted Image with source points drawn')
          plt.imshow(lane_line_sample_video_img)
          for i in range(src.shape[0]):
              x,y = src[i]
              plt.plot(x,y, 'bo', markersize=10)
              plt.text(x * (1.05), y * (1.05) , i, fontsize=24)

          ax = plt.subplot(1,2,2)
          ax.set_title('Warped result with dest.points drawn')
          plt.imshow(warped_sample_video_img)
          for i in range(dst.shape[0]):
              x,y = dst[i]
              plt.plot(x,y, 'bo', markersize=10)
              plt.text(x * (1.05), y * (1.05) , i, fontsize=24)

          plt.show()
```



```
In [293]: def apply_img_pipeline(undistorted_img, objpoints, imgpoints):
              img_size = (undistorted_img.shape[1], undistorted_img.shape[0])

              src = np.float32(
                   [[(img_size[0] / 2) - 87, img_size[1] / 2 + 120],
                   [((img_size[0] / 6) + 45), img_size[1]],
                   [(img_size[0] * 5 / 6) + 75, img_size[1]],
                   [(img_size[0] / 2 + 85), img_size[1] / 2 + 120]])
              dst = np.float32(
                  [[(img_size[0] / 4), 0],
                  [(img_size[0] / 4), img_size[1]],
                  [(img_size[0] * 3 / 4), img_size[1]],
                  [(img_size[0] * 3 / 4), 0]])

              warped_img, Minv = warp(undistorted_img, src, dst)

              abs_sobel_thresh_x_img = abs_sobel_thresh(warped_img, orient='x', thresh_min=20,
              abs_sobel_thresh_y_img = abs_sobel_thresh(warped_img, orient='y', thresh_min=20,
              mag_thresh_img = mag_thresh(warped_img, mag_thresh=(30, 100))
              SH_threshold_img = make_SH_threshold_img(warped_img, S_thresh = (90, 255), H_thre

              combine_thresholds_img = combine_thresholds(abs_sobel_thresh_x_img, abs_sobel_th
                                                          mag_thresh_img, SH_threshold_img)

              """
              plt.figure(figsize=(18,4))

              ax = plt.subplot(2,4,1)
              ax.set_title('undistorted')
              plt.imshow(undistorted_img)
              ax = plt.subplot(2,4,2)
              ax.set_title('warped')
              plt.imshow(warped_img, cmap='gray')

              ax = plt.subplot(2,4,3)
              ax.set_title('sobel x')
              plt.imshow(abs_sobel_thresh_x_img, cmap='gray')
              ax = plt.subplot(2,4,4)
              ax.set_title('sobel y')
              plt.imshow(abs_sobel_thresh_y_img, cmap='gray')
              ax = plt.subplot(2,4,5)
              ax.set_title('mag thresh')
              plt.imshow(mag_thresh_img, cmap='gray')
              ax = plt.subplot(2,4,6)
              ax.set_title('SH thresh')
              plt.imshow(SH_threshold_img, cmap='gray')

              ax = plt.subplot(2,4,7)
```

```
        ax.set_title('combine thresholds')
        plt.imshow(combine_thresholds_img, cmap='gray')
        plt.show()
        """

        return combine_thresholds_img, Minv
```

5 Lane Fitting

The two main functions for finding the lane fits are:

1. fit_initial_lanes()
2. fit_lanes()

They both take a preprocessed binary warped image as input. From this, a second order polynomial is detected for the left and right lanes. These are the lane fits. fit_initial_lanes() uses a sliding window to find the points for the new lane fits. fit_lanes() uses priors lane fits to points nearby them to find the new lane fits.

```
In [294]: def fit_initial_lanes(binary_warped):
              # Take a histogram of the bottom half of the image
              histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)

              # Create an output image to draw on and  visualize the result
              out_img = (np.dstack((binary_warped, binary_warped, binary_warped))*255).astype(

              # Find the peak of the left and right halves of the histogram
              # These will be the starting point for the left and right lines
              midpoint = np.int(histogram.shape[0]//2)
              leftx_base = np.argmax(histogram[:midpoint])
              rightx_base = np.argmax(histogram[midpoint:]) + midpoint

              # Choose the number of sliding windows
              nwindows = 9

              # Set height of windows
              window_height = np.int(binary_warped.shape[0]//nwindows)

              # Identify the x and y positions of all nonzero pixels in the image
              nonzero = binary_warped.nonzero()
              nonzeroy = np.array(nonzero[0])
              nonzerox = np.array(nonzero[1])

              # Current positions to be updated for each window
              leftx_current = leftx_base
              rightx_current = rightx_base

              # Set the width of the windows +/- margin
```

```
margin = 100

# Set minimum number of pixels found to recenter window
minpix = 50

# Create empty lists to receive left and right lane pixel indices
left_lane_inds = []
right_lane_inds = []

# Step through the windows one by one
for window in range(nwindows):
    # Identify window boundaries in x and y (and right and left)
    win_y_low = binary_warped.shape[0] - (window+1)*window_height
    win_y_high = binary_warped.shape[0] - window*window_height
    win_xleft_low = leftx_current - margin
    win_xleft_high = leftx_current + margin
    win_xright_low = rightx_current - margin
    win_xright_high = rightx_current + margin

    # Draw the windows on the visualization image
    cv2.rectangle(out_img,(win_xleft_low,win_y_low),(win_xleft_high,win_y_high),
    cv2.rectangle(out_img,(win_xright_low,win_y_low),(win_xright_high,win_y_high

    # Identify the nonzero pixels in x and y within the window
    good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
    (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_high)).nonzero()[0]
    good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
    (nonzerox >= win_xright_low) &  (nonzerox < win_xright_high)).nonzero()[0]

    # Append these indices to the lists
    left_lane_inds.append(good_left_inds)
    right_lane_inds.append(good_right_inds)

    # If you found > minpix pixels, recenter next window on their mean position
    if len(good_left_inds) > minpix:
        leftx_current = np.int(np.mean(nonzerox[good_left_inds]))

    if len(good_right_inds) > minpix:
        rightx_current = np.int(np.mean(nonzerox[good_right_inds]))

# Concatenate the arrays of indices
left_lane_inds = np.concatenate(left_lane_inds)
right_lane_inds = np.concatenate(right_lane_inds)

# Extract left and right line pixel positions
leftx = nonzerox[left_lane_inds]
lefty = nonzeroy[left_lane_inds]
rightx = nonzerox[right_lane_inds]
```

```
        righty = nonzeroy[right_lane_inds]

        # Fit a second order polynomial to each
        left_fit = np.polyfit(lefty, leftx, 2)
        right_fit = np.polyfit(righty, rightx, 2)

        return left_fit, right_fit, left_lane_inds, right_lane_inds, nonzeroy, nonzerox,
```

```
In [295]: def fit_lanes(binary_warped, left_fit, right_fit, margin = 100):
              # Assume you now have a new warped binary image
              # from the next frame of video (also called "binary_warped")
              # It's now much easier to find line pixels!
              nonzero = binary_warped.nonzero()
              nonzeroy = np.array(nonzero[0])
              nonzerox = np.array(nonzero[1])

              # Create an output image to draw on and  visualize the result
              out_img = (np.dstack((binary_warped, binary_warped, binary_warped))*255).astype(


              left_lane_inds = ((nonzerox > (left_fit[0]*(nonzeroy**2) + left_fit[1]*nonzeroy
              left_fit[2] - margin)) & (nonzerox < (left_fit[0]*(nonzeroy**2) +
              left_fit[1]*nonzeroy + left_fit[2] + margin)))

              right_lane_inds = ((nonzerox > (right_fit[0]*(nonzeroy**2) + right_fit[1]*nonzero
              right_fit[2] - margin)) & (nonzerox < (right_fit[0]*(nonzeroy**2) +
              right_fit[1]*nonzeroy + right_fit[2] + margin)))

              # Again, extract left and right line pixel positions
              leftx = nonzerox[left_lane_inds]
              lefty = nonzeroy[left_lane_inds]

              rightx = nonzerox[right_lane_inds]
              righty = nonzeroy[right_lane_inds]

              # Fit a second order polynomial to each
              left_fit = np.polyfit(lefty, leftx, 2)
              right_fit = np.polyfit(righty, rightx, 2)

              # Generate x and y values for plotting
              ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )
              left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
              right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

              nonzeroy = np.array(nonzero[0])
              nonzerox = np.array(nonzero[1])

              return left_fit, right_fit, left_lane_inds, right_lane_inds, nonzeroy, nonzerox,
In [296]: def plot_lanes(img, binary_warped, left_fit, right_fit, left_lane_inds, right_lane_i
              # Generate x and y values for plotting
              plt.figure(figsize=(18,4))

              ax = plt.subplot(1,3,1)
              ax.set_title('Original Image')
              plt.imshow(img)

              ax = plt.subplot(1,3,2)
              ax.set_title('Binary Warped Image')
              plt.imshow(binary_warped, cmap='gray')

              ax = plt.subplot(1,3,3)
              ax.set_title('Lane Lines')
              ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )
              left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
              right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

              out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
              out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]
              plt.imshow(out_img)
              plt.plot(left_fitx, ploty, color='yellow')
              plt.plot(right_fitx, ploty, color='yellow')
              plt.xlim(0, 1280)
              plt.ylim(720, 0)

              plt.show()

In [297]: video_image_files = sorted(glob.glob('%s/project_video/*.jpg' % data_dir))
          sample_video_imgs_np = read_imgs(video_image_files[100:110])
          sample_video_imgs_arr = [sample_video_imgs_np[i] for i in range(sample_video_imgs_np

In [298]: binary_warped_img_arr = []
          Minv_arr = []

          for sample_video_img in tqdm(sample_video_imgs_arr):
              undistorted_img = cal_undistort(sample_video_img, objpoints, imgpoints)
              binary_warped_img, Minv = apply_img_pipeline(undistorted_img, objpoints, imgpoin

              binary_warped_img_arr.append(binary_warped_img)
              Minv_arr.append(Minv)

100%|| 10/10 [00:11<00:00,  1.18s/it]
```

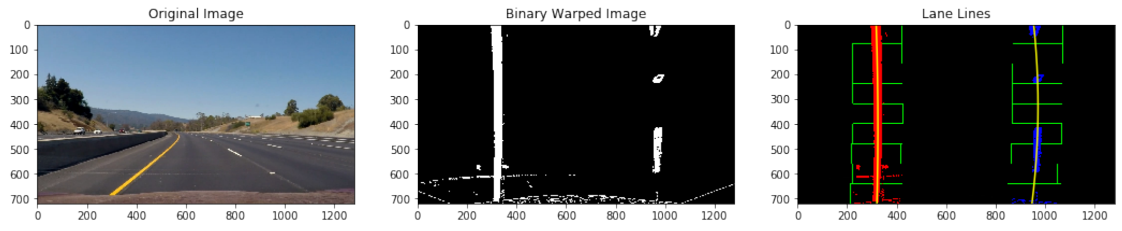
5.0.1 Fit Initial Lanes

This is using a sliding window method to find the lane lines.

```
In [299]: left_fit, right_fit, left_lane_inds, \
          right_lane_inds, nonzeroy, nonzerox, out_img = fit_initial_lanes(binary_warped_img_a
```

The Binary Warped Image is the bird's eye view of the road after camera calibration and thresholding. The image on the far right is a picture of the predicted lane lines. The left lane line is in red and the right lane line is in blue. The green boxes respresents the sliding windows used to find the pixels for the lane.

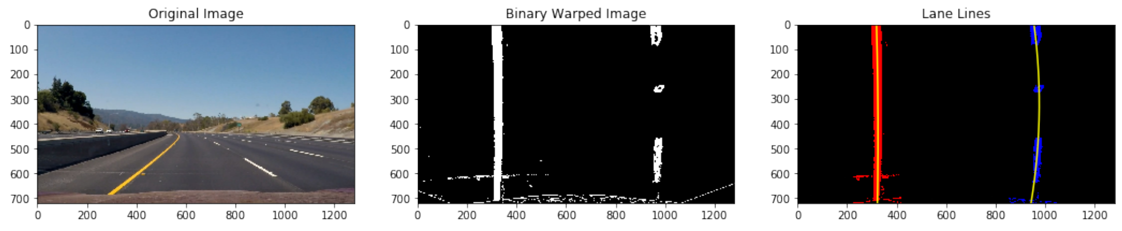
```
In [300]: plot_lanes(sample_video_img, binary_warped_img_arr[0], left_fit, right_fit, left_lane
```



5.0.2 Fit Lanes

Based on the prior left and right fits (equations of the lane lines), find new lane lines. Using the prior fits reduces the amount of the image that needs to be searched. The current lane lines should be close by to where the prior lane lines were. Problems occur when the prior fits are off. Periodically, Fit Initial Lanes with Basic Lane Lines will be done to make sure that fit_lanes() has good prior fits.

```
In [301]: left_fit, right_fit, left_lane_inds, right_lane_inds, \
          nonzeroy, nonzerox, out_img = fit_lanes(binary_warped_img_arr[1], left_fit, right_fi

In [302]: plot_lanes(sample_video_imgs_arr[1], binary_warped_img_arr[1], left_fit, right_fit,
                     left_lane_inds, right_lane_inds, nonzeroy,  nonzerox, out_img)
```



6 Processing Video Frames

All video frames will be preprocessed with the following steps:

1. Camera Calibration
    a. Fix distortions from camera with the raw images
2. Pipeline
    a. Warp image to a top down view of the road.
    b. Apply binary threshold filters.
        1. Sobel x Gradient
        2. Sobel y Gradient
        3. Gradient Magnitude
        4. Saturation and Hue

When new line fits are found for each frame, they will be stored in a Line object. Both the left and right lanes will have their own Line object. A max_n_fits number of line fits will be kept at one time. The mean of the stored line fits will be used to draw the current lane lines instead of the current fit. Having a larger value for max_n_fits makes the model more resilient handling one or more bad frames in close sucession. Usually, the lane fits don't change that dramatically from frame to frame so taking the mean of the fits smoothens out the errors. The disadvantage of having a large max_n_fits value is with curvy roads whose lane fits change a lot from frame to frame.

The first video frame will be processed with fit_initial_lanes(). This will use a sliding window to find the pixels to use for the polyfit to find the fit for the line. After the first frame, fit_lanes() will be used to find the fit for the line. This will use the mean line fit to help find the current line fit for the lane line.

There are two cases where fit_initial_lanes() will be called again after the first video frame:

1. If the width of the lane is outside the expected valid range for a lane.
2. After a specified number of frames has been processes.

The width of lane lines are usually around 3 m. If the lane line width at bottom of the image deviates from this by a large amount, the prediction for where the lane lines are are incorrect. fit_lanes() is probably having trouble finding the current lane line from the mean fits. Calling fit_initial_lanes() should fit this. fit_lanes() could still have trouble finding the current lane line but the bottom width of the lane line is within a valid range. Periodically calling fit_lanes() should help the model get back to a good state for fit_lanes()

```
In [303]: class Line(object):
              def __init__(self, img_height, max_n_fits):
                  self.img_height = img_height
                  self.max_n_fits = max_n_fits

                  self.fits = []

                  self._y_vals = None

                  self._last_mean_fit = None

              @property
              def mean_fit(self):
```

```
        if self.fits is None or len(self.fits) == 0:
            return None

        # Yah! Side effects! Cache the mean_fit value in last_mean_fit so you don't 
        self._last_mean_fit = np.mean(self.fits, axis=0)

        return self._last_mean_fit

    @property
    def last_mean_fit(self):
        if self._last_mean_fit is None:
            self._last_mean_fit = self.mean_fit

        return self._last_mean_fit

    @property
    def y_vals(self):
        if self._y_vals is None:
            self._y_vals = np.linspace(0, self.img_height-1, num=self.img_height)

        return self._y_vals

    @property
    def x_vals(self):
        if self.mean_fit is None:
            return None

        return self.mean_fit[0] * self.y_vals**2 + self.mean_fit[1] * self.y_vals + 

    @property
    def bottom_x(self):
        if self.mean_fit is None:
            return None

        return self.mean_fit[0] * self.img_height**2 + self.mean_fit[1] * self.img_he

    @property
    def radius_of_curvature(self):
        """
        radius of curvature of the line in meters
        """
        # Define conversions in x and y from pixels space to meters
        ym_per_pix = 30/720 # meters per pixel in y dimension
        xm_per_pix = 3.7/700 # meters per pixel in x dimension

        fit = np.polyfit(self.y_vals*ym_per_pix, self.x_vals*xm_per_pix, 2)

        # Calculate the new radii of curvature
```

```
            radius_of_curvature = ((1 + (2*fit[0]*self.img_height*ym_per_pix +
                                     fit[1])**2)**1.5) / np.absolute(2*fit[0])

            return radius_of_curvature

        def add_fit(self, fit):
            self.fits.append(fit)

            if len(self.fits) > self.max_n_fits:
                self.fits = self.fits[-self.max_n_fits:]
```

```
In [304]: def calc_lane_width(img_height, left_fit, right_fit):
              xm_per_pix = 3.7/700

              left_bottom_x = left_fit[0] * img_height**2 + left_fit[1] * img_height + left_fi
              right_bottom_x = right_fit[0] * img_height**2 + right_fit[1] * img_height + righ
              lane_width = (right_bottom_x - left_bottom_x) * xm_per_pix

              return lane_width
```

```
In [305]: def draw_lanes_and_annotations(image, warped, Minv, ploty, left_fitx, right_fitx,
                                         curverad, offset, lane_width, frame_id=0):
              # Create an image to draw the lines on
              warp_zero = np.zeros_like(warped).astype(np.uint8)
              color_warp = np.dstack((warp_zero, warp_zero, warp_zero))

              # Recast the x and y points into usable format for cv2.fillPoly()
              pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
              pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
              pts = np.hstack((pts_left, pts_right))

              # Draw the lane onto the warped blank image
              cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))

              # Warp the blank back to original image space using inverse perspective matrix (
              newwarp = cv2.warpPerspective(color_warp, Minv, (image.shape[1], image.shape[0])

              # Combine the result with the original image
              #result = cv2.addWeighted(undist, 1, newwarp, 0.3, 0)
              output_img = cv2.addWeighted(image, 1, newwarp, 0.3, 0)

              frame_id_txt = 'Frame #%d' % frame_id
              curvature_txt = 'Curvature: %.2f' % curverad
              offset_txt = 'Offset: %.2f' % offset
              lane_width_txt = 'Lane Width: %.2f' % lane_width

              font = cv2.FONT_HERSHEY_SIMPLEX
              text_color = (12, 12, 12)
```

```
        cv2.putText(output_img, frame_id_txt, (10,60), font, 2, text_color,2,cv2.LINE_AA
        cv2.putText(output_img, curvature_txt, (10,120), font, 2, text_color,2,cv2.LINE_
        cv2.putText(output_img, offset_txt, (10,180), font, 2, text_color,2,cv2.LINE_AA)
        cv2.putText(output_img, lane_width_txt, (10,240), font, 2, text_color,2,cv2.LINE_

        return output_img
```

```
In [306]: def make_lane_line_frames(img_files, output_dir, objpoints, imgpoints,
                                    window_width = 50, window_height = 80, margin = 100,
                                    max_n_fits=10, reinit_cnt=40, start_idx=0, min_lane_width=
              os.makedirs(output_dir, exist_ok=True)

              frame_files = []
              curvatures = []

              img_shape = imread(img_files[0]).shape
              img_height = img_shape[0]
              img_width = img_shape[1]

              left_line = Line(img_height, max_n_fits)
              right_line = Line(img_height, max_n_fits)
              ploty = left_line.y_vals

              init_fit_lanes = True
              xm_per_pix = 3.7/700
              offset = 0.0

              for i, img_file in enumerate(tqdm(img_files)):
                  img = imread(img_file)

                  # fix distortions
                  undistorted_img = cal_undistort(img, objpoints, imgpoints)

                  # apply pipeline
                  warped, Minv = apply_img_pipeline(undistorted_img, objpoints, imgpoints)

                  if init_fit_lanes:
                      left_fit, right_fit, _, _, _, _, _ = fit_initial_lanes(warped)
                      init_fit_lanes = False
                  else:
                      left_fit, right_fit, _, _, _, _, _ = fit_lanes(warped, left_line.last_mea
                                                                     right_line.last_mean_fit,

                  left_line.add_fit(left_fit)
                  right_line.add_fit(right_fit)

                  frame_idx = i+start_idx
```

```
                offset = (img.shape[1]/2.0)-(left_line.bottom_x+right_line.bottom_x)/2.0

                lane_width = (right_line.bottom_x - left_line.bottom_x) * xm_per_pix

                if ((min_lane_width is not None and lane_width < min_lane_width) or
                    (max_lane_width is not None and lane_width > max_lane_width)):
                    # the width of the lane seems too wide. reinit finding the lanes
                    init_fit_lanes = True

                curverad = np.min([left_line.radius_of_curvature, right_line.radius_of_curva
                curvatures.append(curverad)

                road_with_lanes = draw_lanes_and_annotations(undistorted_img, warped, Minv, 
                                                             left_line.x_vals, right_line.x_
                                                             curverad, offset, lane_width, f

                frame_file = "%s/frame_%s.jpg" % (output_dir, pad_zeros(frame_idx))
                imsave(frame_file, road_with_lanes)

                frame_files.append(frame_file)

                if i % reinit_cnt == 0:
                    # do an init after 10 times the number of cached fits
                    init_fit_lanes = True

            return frame_files, curvatures
```

```
In [307]: def make_video(img_dir, outfile):
              file_list = sorted(glob.glob('%s/*.jpg' % img_dir))
              img_clips = []
              for file in tqdm(file_list):
                  img_clips.append(ImageClip(file).set_duration(0.1))

              video = concatenate_videoclips(img_clips, method='compose')
              video.write_videofile(outfile, fps=24)
```

```
In [308]: def plot_curvature_histogram(curvatures, num_bins = 20):
              n, bins, patches = plt.hist(curvatures, num_bins, facecolor='blue', alpha=0.5)
              plt.xlabel('Radius of Curvature (meters)')
              plt.ylabel('Frequency')
              plt.title('Radius of Curvature Histogram')
              plt.show()
```

```
In [322]: output_video_dir = 'output_video'
          os.makedirs(output_video_dir, exist_ok=True)
```

6.1 Project Video

This is the main driving video for the project. It was taken somewhere in the Mountain View area.