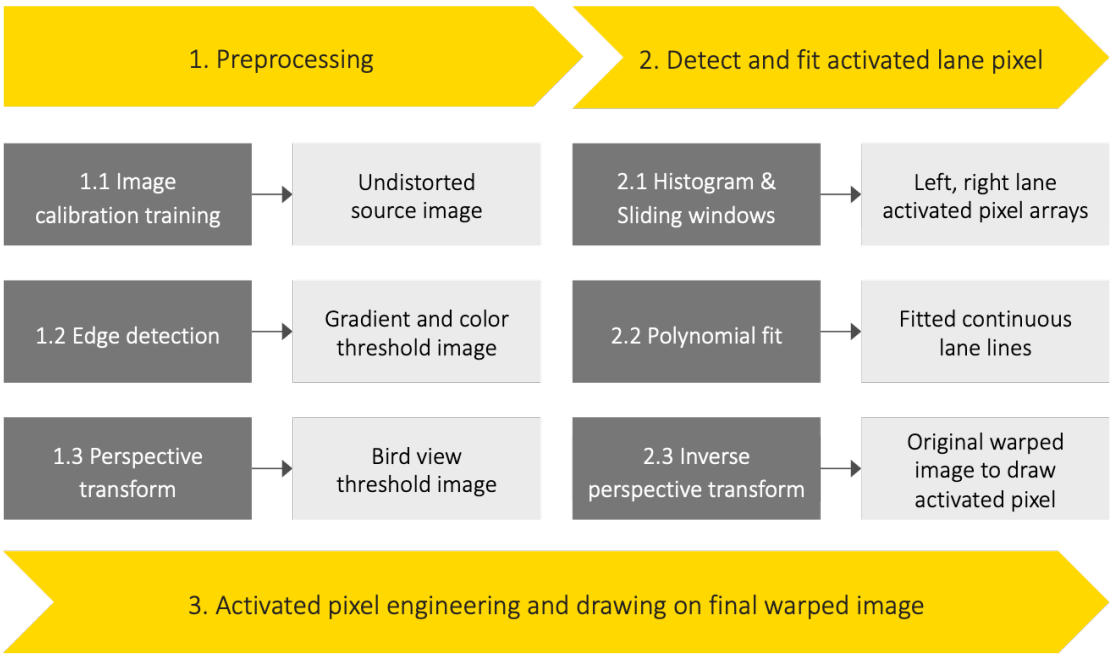
State of the Art Lane Line Detection

Author: Carolina Hoffmann-Becking
Github: carolina-github
Date: 24 Aug 2020

Goal is to write a software pipeline to identify the lane boundaries in a video from a front-facing camera on a car.

Approach

Libraries: OpenCV, matpotlib, numpy, moviepy, IPython



Result

Successfully detected lanelines over the full length of the project video. Reliable and consistant lane line detection across

- ✓ change of road conditions,
- ✓ change of lighting conditions,
- ✓ different coloured lane lines,
- ✓ surpassing vehicles

1. Preprocessing

1.1 Image calibration

Objective

Calibrate camera to remove inherent distortions that can affect its perception of the world.

“Image distortion occurs when a camera looks at 3D objects in the real world and transforms them into a 2D image; this transformation isn’t perfect. Distortion actually changes what the shape and size of these 3D objects appear to be. So, the first step in analyzing camera images, is to undo this distortion so that you can get correct and useful information out of them.” *(Source: Udacity)*

Approach

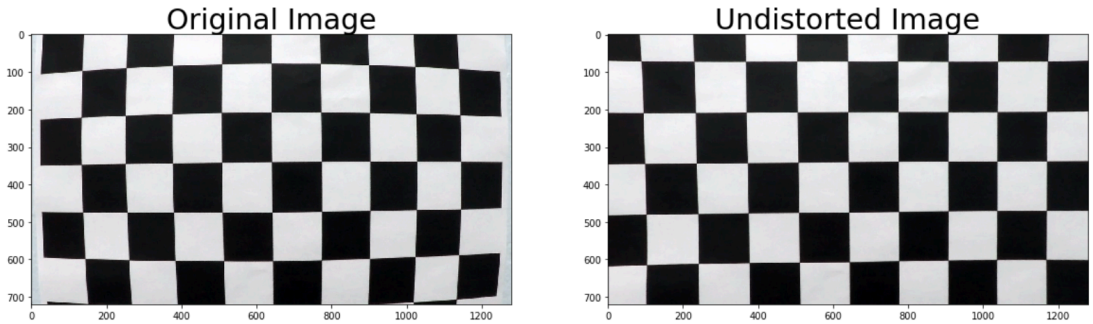
Extract object points and image points for camera calibration on at least 20 images to compute camera calibration matrix and calculate distortion coefficients

- ✓ *Initialize object points with numpy zero function and mgrid for x and y values*
- ✓ *Run corner detection function on images with OpenCV function cv2. findChessboardCorners*
- ✓ *Calibrate and calculate camera matrix, distortion coefficients and camera position vectors with OpenCV function cv2.calibrateCamera*
- ✓ *Test distortion correction on raw images with OpenCV function cv2.undistort*

Result

Calculated the following variables with cv2.calibrateCamera based on chessboard training images:

- dist - distortion coefficients
- mtx - camera calibration matrix
- rvecs - rotation vector camera position
- tvecs - translation vector camera position



This allows to correctly display objects in project images by undistorting any project image with cv2.undistort(image, mtx, dist, None, mtx).

1.2 Edge detection

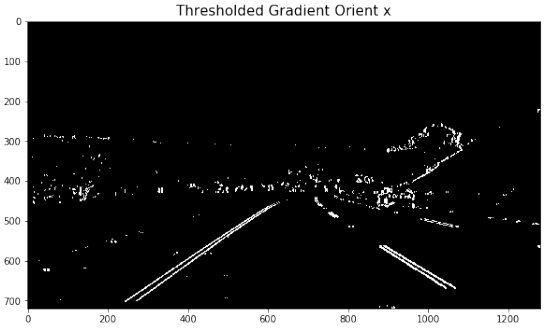
Objective

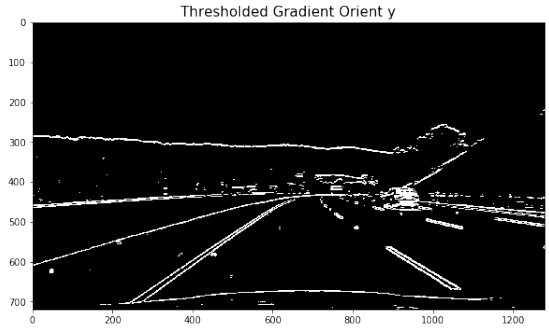
Use various aspects of gradient measurements to isolate lane-line pixels

1.2.1 X and Y gradients

Create a binary threshold to select pixels based on **gradient strength** for derivatives calculated in the x and y direction

- ✓ **sobelx** = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize = sobel_kernel)
- ✓ **sobely** = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize = sobel_kernel)
- ✓ Default **kernel size** of 3, taking the gradient over larger regions (>3) can smooth over noisy intensity fluctuations on small scales. I've selected a kernel size of 9 to allow lane line detection to be more robust to "noise" on the street.

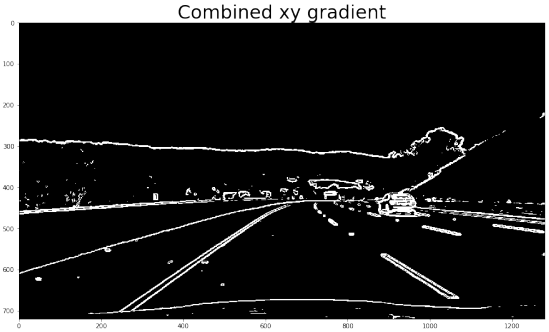




1.2.2 Overall gradient magnitude

Apply the combined magnitude of the x and y gradient. The **magnitude of the gradient** is the maximum rate of change at the point.

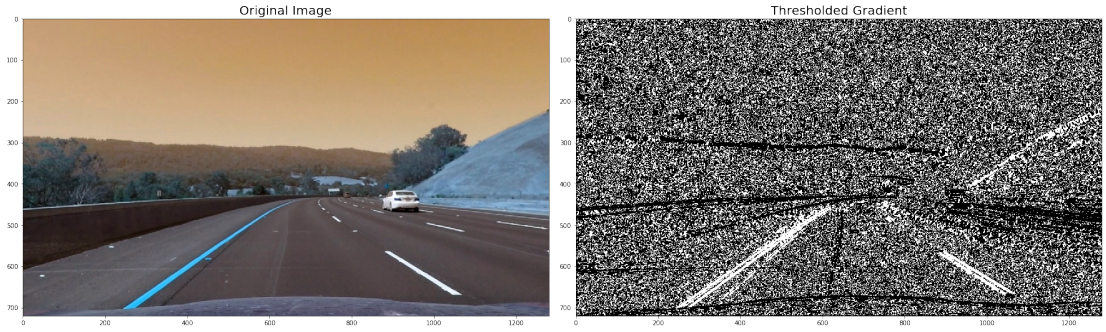
- ✓ **abs_sobelxy** = np.sqrt(sobelx**2 + sobely**2)



1.2.3 Gradient direction

The direction of the gradient is much noisier than the gradient magnitude, but you should find that you can pick out **particular features by orientation**.

- ✓ **gradient_direction** = np.arctan(abs_sobely/abs_sobelx)



1.2.4 Colour Gradient

Background

Conversion to gray scale images [RGB] works well alongside gradient detection, which relies on grayscale intensity measurements. However, analysis in alteranative colour space images are more reliable under different light conditions and when lanes are of different color. *(Source: Udacity)*

Alternative colour spaces such as HSV (hue, saturation and value) and HLS (hue, lightness and saturation) are most commonly used for image analysis

Approach

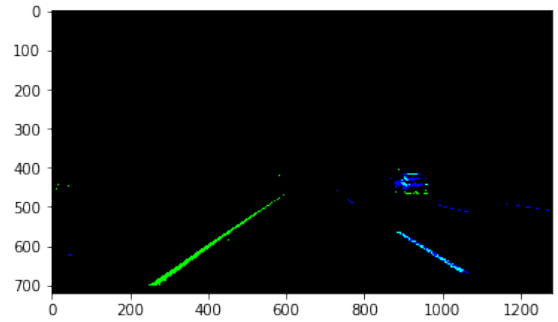
Conversion to HSL colour space:
hls = cv2.cvtColor(destination_image, cv2.COLOR_RGB2HLS)

- ✓ **S channel** picks up the lines under very different color and contrast conditions and is hence more robust to changing conditions
  S = hls[:,:,2]
  Define threshold for S channel: thresh_s = (170, 255)

- ✓ **L channel** to pick up lanes under different lighting conditions
  L = hls[:,:,1]
  Define threshold for L channel: thresh_l = (200, 255)

Result

Participation of different colour channels: L [Blue], S [Green]



1.2.5 Threshold summary

Implications

I have tested different combinations of gradient and colour thresholds. The final combination determines vastly the final success for accurate lane line detection among noise. [see variation testing at In[22]+ in the code]

There is a clear tradeoff between detecting as much as possible (including noise) and keeping detection to the basics to have a more robust model.

As an example, while the gradient in y direction detects additional edges in the picture that are not detected in x direction it also adds “additional noise” to the model. This can lead to detecting a passing car as a new edge or the shadows from sideroad blocks “falsly” as a lane.

Approach

In my model, I have chosen the following combination that activates image pixel if

- ✓ (sobel_x_binary == 1)
- ✓ **or** both (sobel_xy_combined_binary == 1) & (gradient_direction_binary_output == 1)
- ✓ **or** (binary_s == 1)
- ✓ **or** (binary_l == 1)

1.3 Perspective Transform

A perspective transform maps the points in a given image to different, desired, image points with a new perspective. The perspective transform most interesting for lane line detection in is a **bird's-eye view transform** which allows to view a lane from above; this will be useful for calculating the lane curvature as sanity check. *(Source: Udacity)*

Assumptions

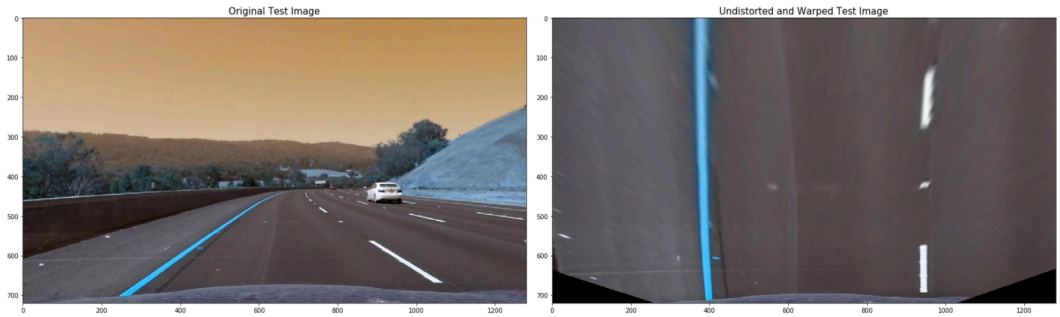
- ✓ Road is a flat plane as approximation
- ✓ Source points in a trapezoidal shape (similar to region masking) represent a rectangle in undistorted image (bird eye view)

Code

- ✓ src = np.float32([(150,imshape[0]),(600, 470), (750, 470), (1150,imshape[0])])
- ✓ dst = np.float32([(350,imshape[0]),(400, 0), (1000, 0), (950,imshape[0])])
- ✓ M = cv2.getPerspectiveTransform(src, dst)
- ✓ bird_view = cv2.warpPerspective(destination_image, M, img_size, flags=cv2.INTER_LINEAR)
- ✓ imshape = (test_img.shape[0], test_img.shape[1])
- ✓ img_size = (test_img.shape[1], test_img.shape[0])

Debugging: Note that imshape [y, x] and img_size [x, y] are not interchangeable. Imshape refers to defining the source points and destination points in the image [~region of interest], while img_size is required to warp the image using the opposite shape format.

Result



2. Detect and fit avtivated lane pixel

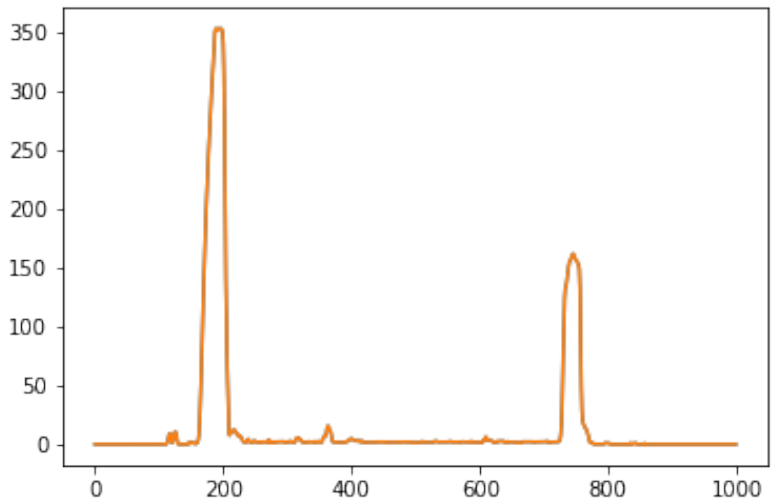
2.1 Histogram & Sliding windows

2.1.1 Histogram to define starting point of lane line detection

The histogram adds up the pixel values along each column in the image. In the thresholded 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. This can be used as a starting point for where to search for the lines. *(Source: Udacity)*

Code

- ✓ histogram = np.sum(bird_view_threshold[bird_view_threshold.shape[0]//2:,200:1200], axis=0)



Debugging: It is important to crop the image to a region of interest for the histogram function to avoid detection of non lane lines. Therefore the crop of 200 from the left of the image needs to be added in the following code to detect the accurate lane line positions

- ✓ midpoint = np.int(bird_view_threshold.shape[1]//2)
- ✓ left_lane_x_current = np.argmax(histogram[:midpoint]) + 200
- ✓ right_lane_x_current = np.argmax(histogram[midpoint:])+ 200 + midpoint

2.1.2 Sliding windows

From histogram detected starting 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.

Approach

2.1.2.1 Setting **hyperparameters**

- ✓ Choose the number of sliding windows [nwindows = 9]
- ✓ Set the width of the windows +/- margin [margin = 100]
- ✓ Set minimum number of pixels found to recenter window [minpix = 50]

2.1.2.2 Loop through each window in nwindows

- Find the **boundaries of the current window.** This is based on a combination of the current window's starting point (leftx_current and rightx_current), as well as the margin set in the hyperparameters.
  - ✓ # Identify window boundaries - Top and bottom
    win_y_low = bird_view_threshold.shape[0] - (i+1)*window_height
    win_y_high = bird_view_threshold.shape[0] - i*window_height
  - ✓ # Identify window boundaries - left lane window
    win_xleft_low = left_lane_x_current - margin
    win_xleft_high = left_lane_x_current + margin
  - ✓ # Identify window boundaries - right lane window
    win_xright_low = right_lane_x_current - margin
    win_xright_high = right_lane_x_current + margin

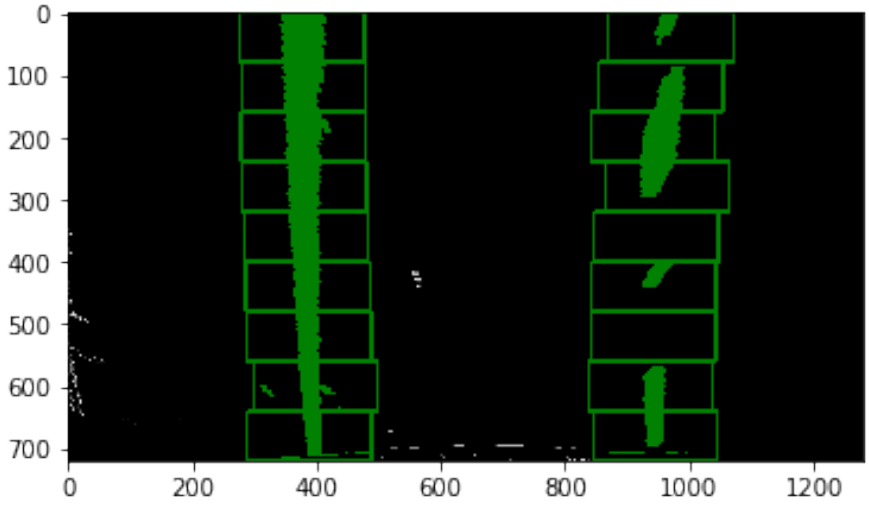
- Use cv2.rectangle to **draw window boundaries** onto the binary thresholded image
  - ✓ cv2.rectangle(bird_view_threshold,(win_xleft_low,win_y_low),(win_xleft_high,win_y_high),(0,127,255), 3)
  - ✓ cv2.rectangle(bird_view_threshold,(win_xright_low,win_y_low),(win_xright_high,win_y_high),(0,127,255), 3)

- Find relevant **activated pixels** from nonzeroy and nonzerox (project image) actually fall into the window and append these to the left_lane_inds and right_lane_inds lists
  - ✓ nonzero = bird_view_threshold.nonzero()
  - ✓ nonzeroy = np.array(nonzero[0])
  - ✓ nonzerox = np.array(nonzero[1])
  - ✓ 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]
  - ✓ left_lane_inds.append(good_left_inds)
  - ✓ right_lane_inds.append(good_right_inds)

- If the number of activated pixels are greater than the hyperparameter minpix, **re-center window** (i.e. leftx_current or rightx_current) based on the mean position of these pixels
  - ✓ if len(good_left_inds) > minpix:
    left_lane_x_current = np.int(np.mean(nonzerox[good_left_inds]))
  - ✓ if len(good_right_inds) > minpix:
    right_lane_x_current = np.int(np.mean(nonzerox[good_right_inds]))

Result

- Array of left lane pixels
- Array of right lane pixels
- Visualisation of sliding windows and activated pixels within windows:



2.2 Polynomial fit

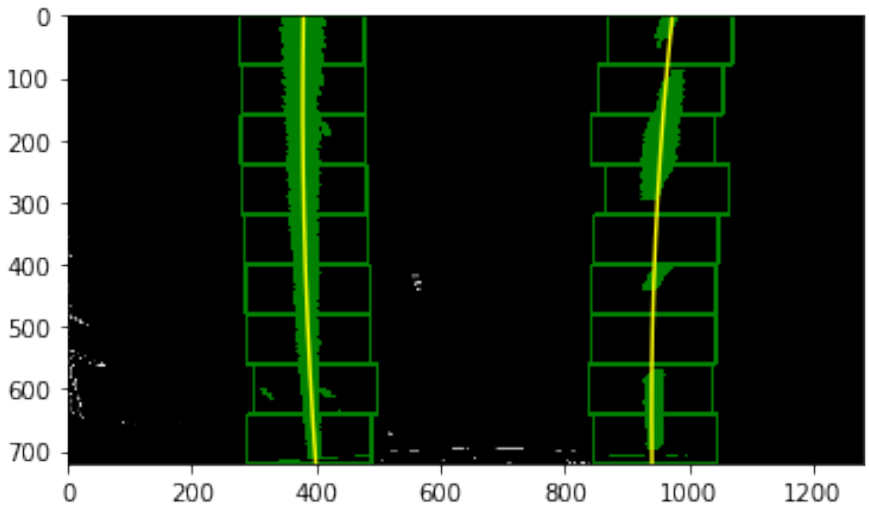
Polynomial is used in numerical analysis to approximate lane lines curvature. Degree 2 polynomial represents a quadratic function: f(x) = a*x2 + b*x + c

Approach

- Use **existing pixels** to derive variables a,b and c of quadratic function (2 degree polynomial)
  - ✓ left_fit = np.polyfit(lefty_poly, leftx_poly, 2)
  - ✓ right_fit = np.polyfit(righty_poly, rightx_poly, 2)
- Plot variables into polynomial to **generate additional y and x values** for plotting
  - ✓ ploty = np.linspace(0, bird_view_threshold.shape[0]-1, bird_view_threshold.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]

Debugging: While normally you calculate a y-value for a given x, here we do the opposite. Why? Because we expect our lane lines to be (mostly) vertically-oriented.

Result



2.3 Inverse perspective transform

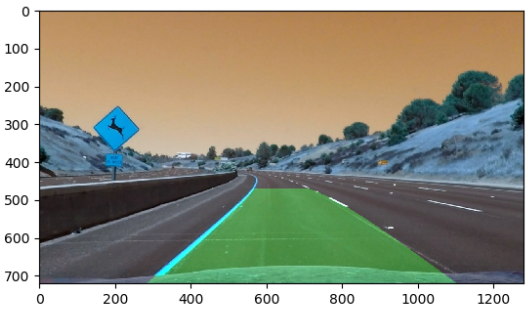
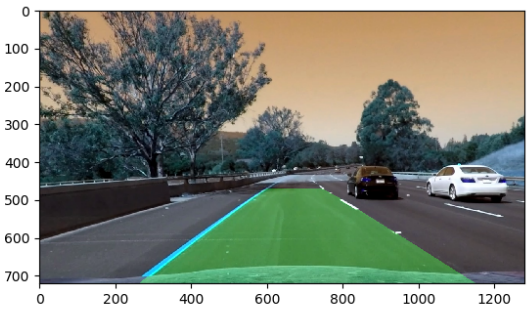
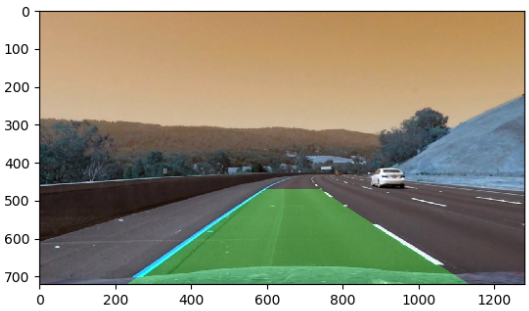
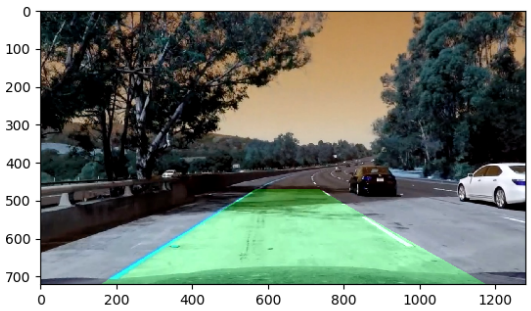
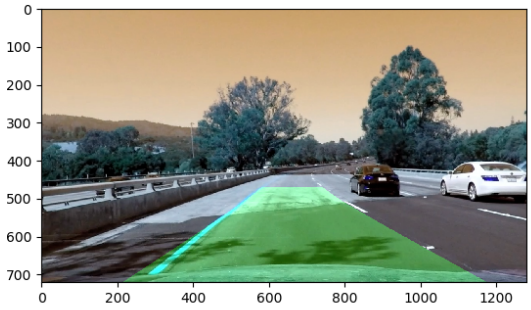
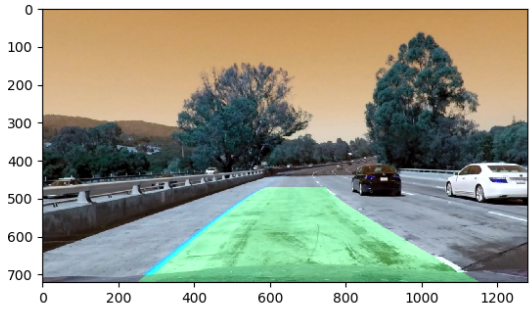
Warp the image with activated pixels back to original image space using inverse perspective matrix (Minv) to draw activated pixels onto image in next step.

- ✓ Minv = cv2.getPerspectiveTransform(dst, src)
- ✓ newwarp = cv2.warpPerspective(color_warp, Minv, (test_img.shape[1], test_img.shape[0]))

3. Activated pixel engineering and drawing on final warped image

- 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))
- Combine the result with the original image and inverse warped activated pixel image
  - ✓ cv2.addWeighted(destination_image, 1, newwarp, 0.3, 0)

Result

Final Result

- Annotated video

Successfully detected lanelines over the full length of the project video. Reliable and consistant lane line detection across

- ✓ change of road conditions,
- ✓ change of lighting conditions,
- ✓ different coloured lane lines,
- ✓ surpassing vehicles

Areas of improvement

1. Improve **Efficiency**

   Increase efficiency since once you know where the lines are in one frame of video, you can do a highly targeted search in the next frame

   Approach [code ln[56] ++]

   - ✓ Search within a margin (new hyperparameter) around the previous lane line position fitted using the polynomial function
   - ✓ If you lines are lost, go back to sliding windows search to redefine position

2. Adopt best practices for dode structure by **defining a class** to receive the characteristics of each line detection [code Appendix]

3. Embedd additional **sanity checks** to improve performance in challenge video:

   - ✓ Measure and track lane curvature measures to identify similar lane curvature as matching lane lines
   - ✓ Check if lane lines are separated by approximately the right distance horizontally and if they are roughly parallel
   - ✓ Determine vehicle position with respect to center