Advanced Lane Lines Detection

Dan Bergeland, Feb 6, 2017

Udacity Self Driving Car Nano-Degree Term 1, Project 4

Overview:

In this assignment, the desired result is to take a recorded video and accurately draw lane lines and the desired path forward in front of the vehicle. This is accomplished by using the OpenCV library to adjust for camera distortion, apply lane detection thresholding and add graphical overlays to the original stream.

Camera Calibration:

Camera distortion is a well known issue with inexpensive cameras that are widely used for phones, web cams and dash cams. Since this is such a common issue, OpenCV has built in functionality to create calibration matrices that adjust for distortion caused by a camera. One of the calibration techniques is to use chessboard patterns with a known number of black/white corner intersections. For this class, the images of the chessboards were provided in the course materials.

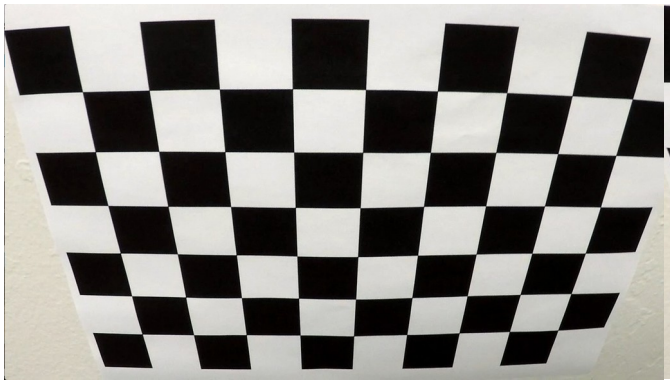

Illustration 1: Distorted Calibration Image

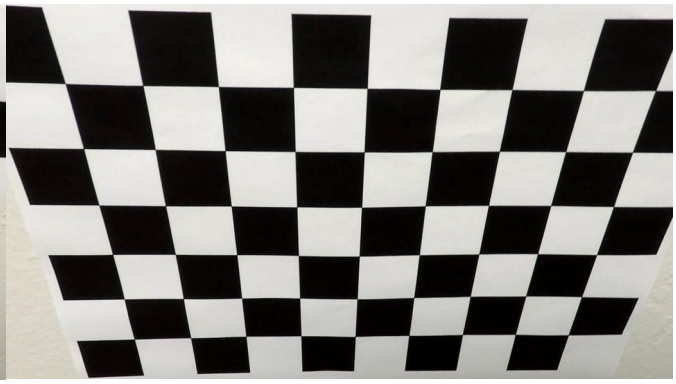

Illustration 2: Calibration Image with transform

The important thing to note for the calibration is that there are 6 rows and 9 columns of intersections. If these values are incorrect, the findChessboardCorners will not detect the chessboard. OpenCV's 'findChessboardCorners' function was applied to a set of 20 provided calibration images similar to Illustration 1 and 2. This function returns a transform matrix that describes how each point in the image needs to be modified to get an undistorted image. This matrix is used with the 'undistort' function in OpenCV.

Looking closely at the image, it's apparent that Illustration 1 has noticeably bending lines in both the horizontal and vertical directions. This minor 'fish-eye' lens effect is a result of inexpensive optics that cause distortion. After the calibration is applied, as shown in Illustration 2, the lines are flat in vertical

and horizontal directions. This is the expected result because we know the real chessboard has parallel lines, so the undistort function worked.
For use in the pipeline, the calibration program was used only once. The resulting calibration matrix was saved in a python pickle file. The undistort function was abstracted to its own python file to keep the matrix in memory for repeated access while the pipeline is running. In the file package, the camera calibration is “calibrate_camera.py”.

Pipeline:

The actual code to encode the video files is only a few lines long. It goes frame-by-frame and calls ‘find_lane_lines’ on each frame, and saves the result to a new file. Here’s the entire video encoding code in the file ‘lane_find_video.py’; it takes the file name of the source video as a command line argument.

```
import imageio
import sys
from moviepy.editor import VideoFileClip
from lanefinding import find_lane_lines
if __name__ == "__main__":
    vidPath = sys.argv[1]
    clip1 = VideoFileClip(vidPath)
    newClip = clip1.fl_image(find_lane_lines) #NOTE: this function expects color images!!
    dstName = 'outputs/laneFind'+vidPath
    newClip.write_videofile(dstName, audio=False)
```

The file ‘lanefinding.py’ contains the function ‘find_lane_lines’, which runs the actual frame-by-frame pipeline. As it will be shown, this pipeline makes use of more class-heavy object oriented programming, so it’s also a relatively short 105 total lines of code.

Open the image and handle color space:

The pipeline uses native OpenCV BGR color space, but the video encoder which is a moviepy.editor VideoFileClip, uses RGB color space. The very first thing the pipeline does is convert the RGB images to BGR images. This is to accommodate the use of other OpenCV based functions that were developed in labs during the lectures.

Apply calibration / Undistort:

The undistort function is not affected by the color space transform in the previous step. The undistort function is in the file “undistort.py”. That file unpickles the camera calibration transform matrix and keeps it as a global variable in memory, so it doesn’t need to be reloaded each time the undistort function is called. Once the correction matrices (shown as ‘mtx’ and ‘dist’ below) are loaded from the pickle file, the following code corrects the image:

```
def get_undistort(img):
    return cv2.undistort(img, mtx, dist, None, mtx)
```

Pre-process the image:

As with previous labs, there's Gaussian blur applied to the image. This makes use of the OpenCV GaussianBlur function with a 3x3 kernel. This doesn't have a dramatic affect on the image, but does somewhat reduce the noise in detecting the road surface.

Apply thresholds to generate bit-mask:

After some experimentation, a simple combination of 2 threshold functions were used to create binary masks for lane detection. As mentioned above, thresholds were created based on BGR color space. All of the functions are in 'thresholds.py'. For this pipeline, one filter is setup to find yellow lines in HLS color space and the other is setup to find white lines in RGB. The result of these 2 filters is combined with bit-wise OR to make the binary mask. The following result shows test image 3 with the generated bitmask applied.

Illustration 3: Test3.jpg image

Illustration 4: Test3 BitMask

The filter result is, for this roadway, a clean feature extraction. I used the following threshold values for this pipeline:

```
rgbThresh = rgb_color_thresh(img,r_thresh=(185,255), g_thresh=(185,255), b_thresh=(185,255))
hlsThresh = hls_color_thresh(img,h_thresh=(18,35), l_thresh=(0,255), s_thresh=(40,255))
bMask = np.zeros_like(img[:,:,0])
bMask[(rgbThresh==1)|(hlsThresh==1)]=1
```

Post-process the mask:

To further improve my detection result, I added a morphology ‘close’ function on the bit mask. This has the effect of dilating pixels to fill in empty spaces, then erodes the edge pixels an equal amount so it doesn’t appear any larger. Basically, this will fill in areas with empty pixels and enhance the consistency of the filter results from the previous step.

Illustration 5: Bitmask from above with Close function applied

The result is more noticeable on the left line near the end, where it starts to disappear. On the above image, there’s pixelation as it fades, but this fills those spaces in. This function is the OpenCV function morphologyEx, with parameter cv2.MORPH_CLOSE. The kernel size is 11x11.

```
kernel = np.ones((11,11),np.uint8)
close = cv2.morphologyEx(img,cv2.MORPH_CLOSE, kernel)
```

Perspective Transform:

Now that the image has detection of features, it’s time to transform it to an overhead view to determine the curvature and other information. An alternative pipeline was attempted, where the perspective was transformed before applying thresholds, but it appeared that the transform added noise and artifacts to the image, which made lane line detection less consistent.

The transform uses the OpenCV function warpPerspective. As with the undistort function from above, the transform matrix is calculated once and kept as a global variable in the file containing the function.

This keeps the transform matrix in memory and saves execution time by not recalculating it each iteration.

```
def overhead_perspective(img,M=defM,img_size=defOverheadSize):
    return cv2.warpPerspective(img, M, img_size)
```

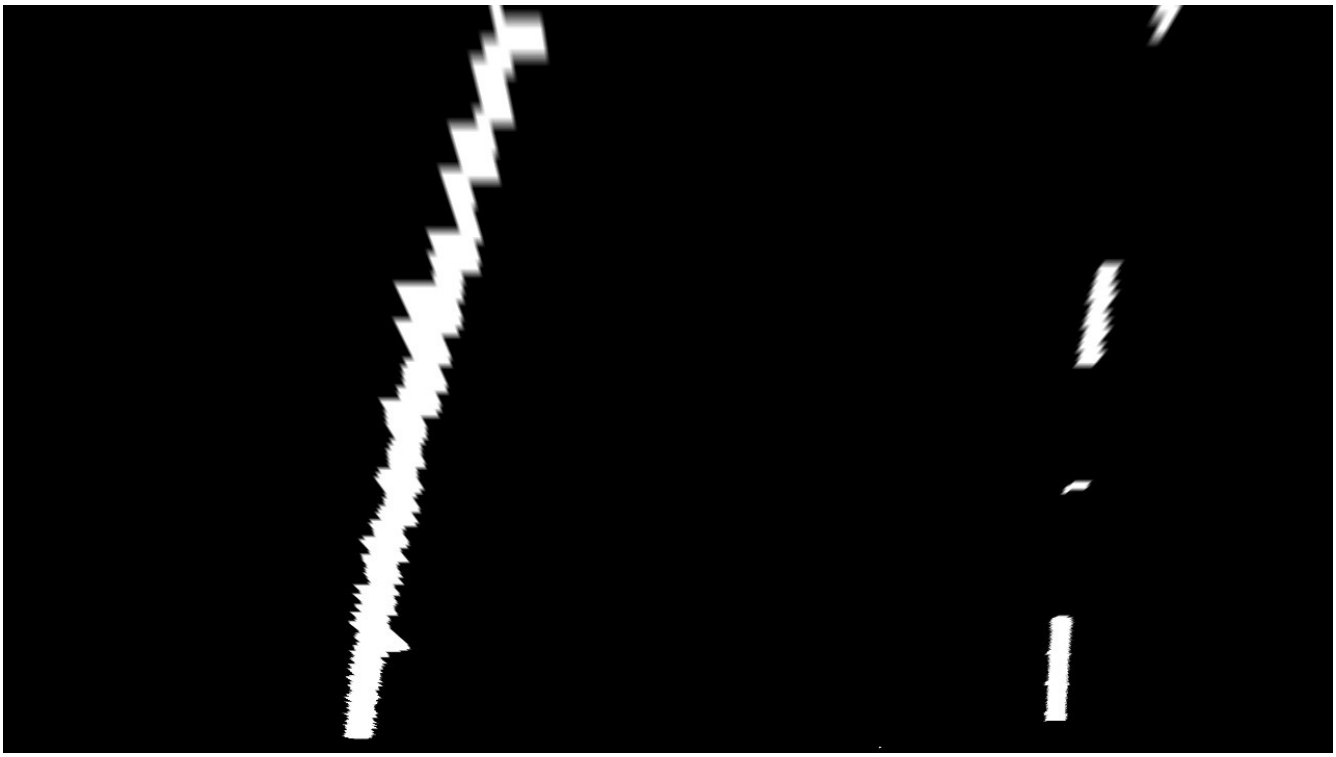

Illustration 6: Bitmask in overhead perspective

The transform assumes a 1280x720 image, as that is the image size coming from the camera in this project. The corner points were defined as ratios of the picture size on the non-transformed image. The top points are at 64% down from the top of the image. The bottom points are at the bottom of the image, or 100% of the way down. Another way to say this, as the bottom 36% of the image is used in the transform. The top left and right are at the center of the image +/- 6.8 % of the width. The bottom points are at 1% and 99% of the width. These were defined mathematically because they had to keep symmetry to correctly sample the starting image.
The transformed points are at the top and bottom of the image, and set towards the middle by 180 pixels.
The transform code is in the ‘undistort.py’ file, and the following code shows the points defined in the image:

```
defImgSize = (1280,720)
height = defImgSize[1]
width = defImgSize[0]
```

```
split = .068
#horizon from top (higher shows less)
redHorz = .64

roadPts = np.float32([[width*(.5-split),height*redHorz],[width*(.5+split),height*redHorz],
[width*.99,height],[width*.01,height]])

defOverheadSize = (1280,720)
offset = 180
left = offset
right = defOverheadSize[0]-offset
top =  0
bot = defOverheadSize[1]

showPts =np.float32([[left,top],[right,top],[right,bot],[left,bot]])
defM = cv2.getPerspectiveTransform(roadPts, showPts)
invM = cv2.getPerspectiveTransform(showPts, roadPts)
```

Lane Line identification:

The method described in class lectures was applied to the overhead transform to detect lane lines. This method is to use a set of sliding windows, stacked from the bottom of the image towards the top, to identify and follow the points in the line. The windows are re-centered at each level to the center of mass for the pixels inside of the previous window. By doing this, the windows can slide to adjust to curving lines.

A histogram of the points in the bottom half of the overhead image is used to identify the starting points for the first of the sliding windows. The peak to the left of the center is used for the left line starting box, the peak to the right of the center line is used for the right line starting box.

For each step, the points inside the box are added to a collection of points that represent each line. The center of mass is then calculated for each window, and the center for the box to be stacked on top is updated to this new center point.

This method deviates from the course method, because additional windows (used 12) were added to allow more flexibility in tracking. This method also adds a minimum number of total pixels in the line to be considered a valid detection (700 pixel threshold was used). With a judicious binary mask, the calculation images can become sparse enough that there is no valid line present. By requiring a minimum number of pixels, it prevents calculations of lines based on a small number of pixels that look more like a spot than a line.

As in the lecture, the pixels found in the left line windows are painted red and the right line is painted blue. The green outlines are the windows that were used to track the line pixels.

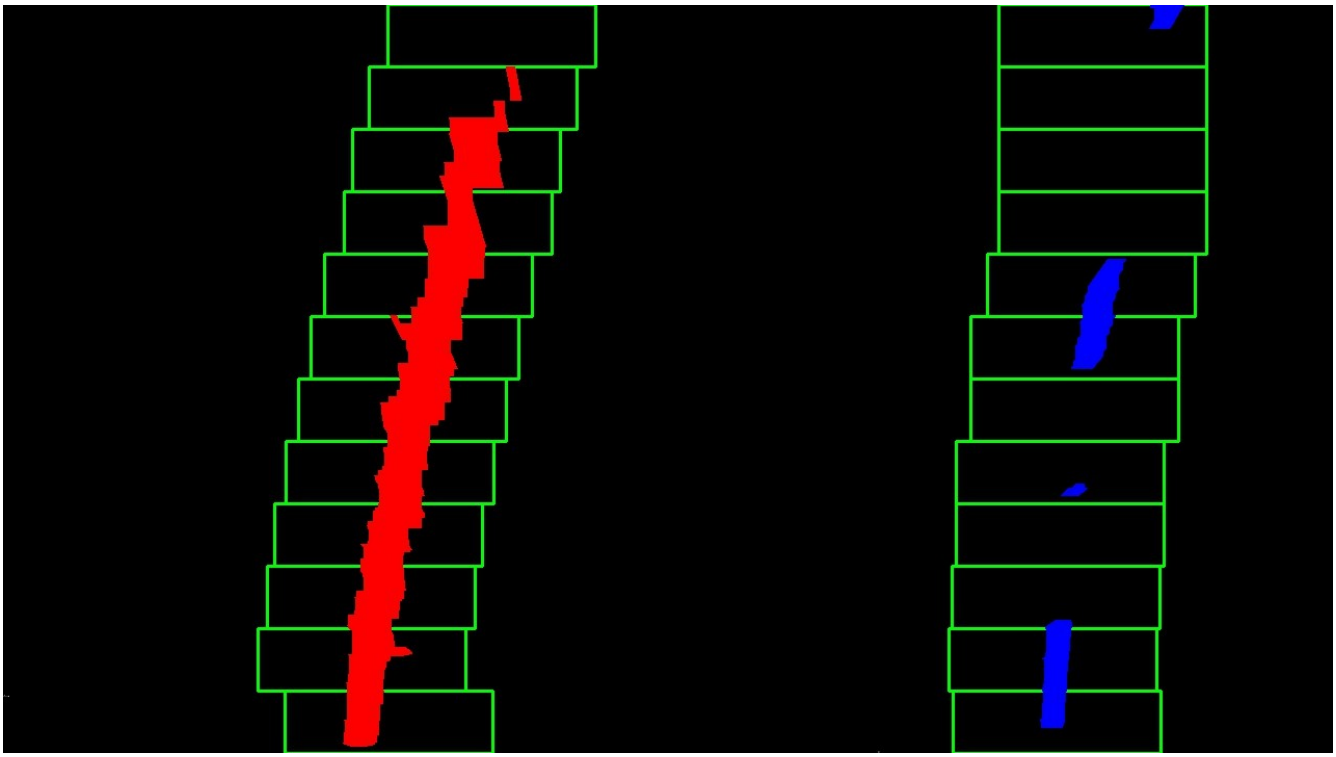

Illustration 7: Test Image 3 with sliding windows applied

Also within this function, the points are used with numpy.polyfit to return the polynomial coefficients for a second order line equation. These polyfit results were used to calculate and draw the lane lines for the project.

The code for this is found in Appendix A.

Data structures and calculations:

Object oriented design was used to handle all the different frames and to allow for averaging and batch handling. Three classes were created, Line, Lane and LaneHistory. The classes are defined in the file lane_data_types.py.

The Line class holds information for each line. This information is the polynomials for the fit run on each line, a boolean value for whether it was detected successfully, and x,y points that make up the line. The only class method for line is to calculate curvature, which is also stored in the line object.

The Lane class holds 2 line objects, one for left and one for right, the lane position, the lane curvature, the calculation image (which shows the sliding windows and left/right pixels). It also contains some meta information, like the camera offset from center and the lane width in meters. The camera offset is set to 0, but it provides generalization so if another vehicle had the camera off to the side, it could still determine lane position. The class methods for lane are to find the lines, calculate lane position and calculate curvature.

The third class is a LaneHistory. This object contains a first in first out buffer by using the python collections.deque object. This buffer contains a set of lane objects that it uses to average all the

calculations. The LaneHistory object holds the last 10 (or whatever size buffer) lanes, the average curve, the average lane position, the average left and right polyfit lines, as well as the overlay image to draw on to the main image. By using the deque type, it handles continuously adding new Lane objects and removing the oldest one from the buffer.
It has 2 class methods, one to average the statistics for all the lanes in the buffer, and another to draw the lane overlay based on the result of the averaged polyfit lines.

Calculating Curvature and Lane Position:

The curvature is calculated using the equation presented in the lecture shown as follows:

```
def calc_curvature(self):
    ym_per_pix = 30/720 #meters per pixel y dim
    xm_per_pix = 3.7/700 #meters per pixel x dim
    y_eval = np.max(self.ally)
    line_fit = np.polyfit(self.ally*ym_per_pix, self.allx*xm_per_pix, 2)
    if line_fit[0]!=0:
        self.radius_of_curvature = ((1 + (2*line_fit[0]*y_eval*ym_per_pix + line_fit[1])**2)**1.5) /
np.absolute(2*line_fit[0])
```

The lecture pixel scaling was also used, because the transform presented here resulted in a similar image. There's a check to make sure that line_fit isn't 0 because that would cause a divide by 0 error, or nan to occur. This calculation is stored for each line.
The lane object takes the average of the left and right line curves, to store a lane curvature estimate. The average is taken because the inside and outside lines have different curves by necessity; if the outside curve had the same radius as the inside curve, it would eventually intersect. The logic is that the center of the lane likely is half way between these two curve values.
The actual output from the program, as displayed on the video, comes from the LaneHistory object. Each lane contains an average between left and right lines, so the LaneHistory will use the average of all ten lanes in its buffer.

Lane position is calculated based on the intercepts of where each lane line intercepts the bottom of the image. Each line has a member data that saves this intercept pixel. The lane object calculates by getting the ratio of left pixels to total lane pixels. This ratio is then multiplied by the lane width to get a position.

For example, the center pixel is at 640, if the left intercept is at 440 and the right pixel is at 1040, the car's position would be 200/600 from the left side. Knowing that the camera, and the middle of the car, is at 33% from the left line, allows the position to be calculated by the known lane width of 3.7m. This is easily converted to a position relative to the center of the lane by subtracting .5 from the ratio based position. In this example .5-.33=-.16, so the lane position from center is 3.7m * -.16 = .59 m. As with the curve above, the average is taken for all ten lanes in the buffer, and the result is used for the lane position. A camera offset was added for future implementations or tuning, where the camera may not

be perfectly centered in the car. For this project, it was set to 0. As shown in code below for the Lane class:

```
    def calc_lane_position(self):
        #must run 'find_lines'
        #get the center value
        if self.leftLine.detected ==True and self.rightLine.detected==True:
            center = int(self.calcimg.shape[1]/2)
            left = abs(self.leftLine.line_base_pos - center)
            right = abs(self.rightLine.line_base_pos - center)
            #add camera offset.  Take ratio of left/left+right-.5 to get center lane = 0.  Multiply by lane
width.
            self.lanePosition = self.cameraOffset + (left/(left+right)-.5)*self.laneWidthMeters
```

Creating the overlay and drawing the output:

As with the other output values, the polyfit lines are an average of the coefficients of the lane lines held in the buffer. That means that the left and right lines drawn at the LaneHistory level object are each the average of 10 previous lanes. The points are generated for all the values in the bounds of the image, since it's using polynomial coefficients to calculate the line. These points, for both the left and right lines, are transposed and built into a single list that's used with the OpenCV fillPoly function. This creates an overhead view of the space between the two lines.
This image is transformed back to the original image by using the OpenCV functions getPerspectiveTranform and warpPerspective. The new image is generated with the following code:

```
#define road to overhead transform matrix
defM = cv2.getPerspectiveTransform(roadPts, showPts)
#define reverse transform from overhead to road view
invM = cv2.getPerspectiveTransform(showPts, roadPts)

def overhead_perspective(img,M=defM,img_size=defOverheadSize):
    return cv2.warpPerspective(img, M, img_size)

def revert_from_overhead(img,M=invM, img_size=defImgSize):
    return cv2.warpPerspective(img,M, img_size)
```

After the overlay is back in the same perspective as the original image, the overlay is given a transparent effect by using the OpenCV function addWeighted:

```
outImg = cv2.addWeighted(img, 1, newwarp, 0.3, 0)
```

I added a white box for the text to be written on in the upper left corner because sometimes the text gets hard to see when it's just written in the sky. I added text for lane position and curve radius, which is the value held in the LaneHistory object.
I also took an extra step and took the individual calculation image for the current lane and inserted it in the upper right corner. I found this to be useful for debugging, and overall interesting to understand what the binary mask would pick up and which pixels it considered to be part of a lane line.

Illustration 8: Processed image, with detected lane, calculation image and position statistics

Link to Output for project video:

Video is on my youtube channel:
Project Video: https://youtu.be/g0rSoJCEcn8
Challenge Video: https://www.youtube.com/watch?v=b6jHO7arALY

Appendix A: Find Lane Line Code

```
def find_lines(self):
    img = self.ohroad
    height = img.shape[0]
    width = img.shape[1]
    # Take a histogram of the bottom half of the image
    histogram = np.sum(img[int(height/2):,:],axis=0)
    # Create an output image to draw on and  visualize the result
    out_img = np.dstack((img, img, img))*255
    # 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 = 12
    # Set height of windows
    window_height = np.int(height/nwindows)
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = img.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 = height - (window+1)*window_height
        win_y_high = height - 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),(0,255,0), 2)
            cv2.rectangle(out_img,(win_xright_low,win_y_low),(win_xright_high,win_y_high),(0,255,0),
2)
            # 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)

        fity = np.linspace(0, height-1, height )

        #Reject frames that don't have very many pixels.  By failing this test, "detected" is false for the
line.
        lane_inds_threshold = 700

        # Extract left and right line pixel positions
        if len(left_lane_inds)>lane_inds_threshold:
            leftx = nonzerox[left_lane_inds]
            lefty = nonzeroy[left_lane_inds]
            left_fit = np.polyfit(lefty, leftx, 2)
            self.leftLine.current_fit = left_fit
            self.leftLine.ally = fity
            fit_leftx = left_fit[0]*fity**2 + left_fit[1]*fity + left_fit[2]
            self.leftLine.allx = fit_leftx
```

```
    self.leftLine.line_base_pos = fit_leftx[-1]
    out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
    self.leftLine.detected = True


if len(right_lane_inds)>lane_inds_threshold:
    rightx = nonzerox[right_lane_inds]
    righty = nonzeroy[right_lane_inds]
    right_fit = np.polyfit(righty, rightx, 2)
    self.rightLine.current_fit = right_fit
    self.rightLine.ally = fity
    fit_rightx = right_fit[0]*fity**2 + right_fit[1]*fity + right_fit[2]
    self.rightLine.allx = fit_rightx
    self.rightLine.line_base_pos = fit_rightx[-1]
    out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]
    self.rightLine.detected = True
self.calcimg = out_img
```