Advanced Lane Finding

September 30, 2019

1 Project: Advanced Lane Finding

This project demonstrates how to use computer vision tools to find lanes. I'll be using the following steps:

1. Camera Calibration
2. Color and Gradient Thresholds
3. Perspective Transform
4. Lane Detection and Lane Curvature

1.0.1 Libraries

We'll use OpenCV and numpy throughout, and a few other libraries for displaying images and plotting data.

```
In [1]: import cv2
        import glob
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
```

2 Camera Calibration

Because camera lenses distort the light that passes through them (especially around the edges), calibration is needed to adjust for this distortion. OpenCV has several functions to make this easy, and it really only needs to be done once. First, we need multiple images of a chessboard pattern from several different angles. Then we can run them through the following procedure.

```
In [2]: objpoints = []
        imgpoints = []

        nx,ny = (9,6)

        # initialize three columns (x,y,z) with zeros
        # (0,0,0), (1,0,0), (2,0,0) ...
        objp = np.zeros((ny*nx,3), np.float32)
```

```
        # Use mgrid to generate coordinates
        # reshape back into two columns
        objp[:,:2] = np.mgrid[0:nx,0:ny].T.reshape(-1,2)
```

2.1 Corner Detection

Convert to grayscale and run `cv2.findChessboardCorners` on all of the chessboard images in the calibration folder, saving the object points and image points. The Object Points (`objpoints`) we created earlier, represent the 9x6 corners of the chessboard. These are appended each time we add image points. The image points are the image x,y pixel locations of each corner.

```
In [3]: calibration_images = glob.glob('camera_cal/calibration*.jpg')

        for fname in calibration_images:
            img = mpimg.imread(fname)

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

            # Find the chessboard corners
            ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)

            # If found, draw corners
            if ret == True:
                imgpoints.append(corners)
                objpoints.append(objp)
```

2.2 Undistort

Convert to grayscale and run cv2.calibrateCamera using the discovered object points and image points. Finally, undistort the image using cv2.undistort.

```
In [4]: def cal_undistort(img, objpoints, imgpoints):
            gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
            ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[
            undist = cv2.undistort(img, mtx, dist, None, mtx)

            return undist
```

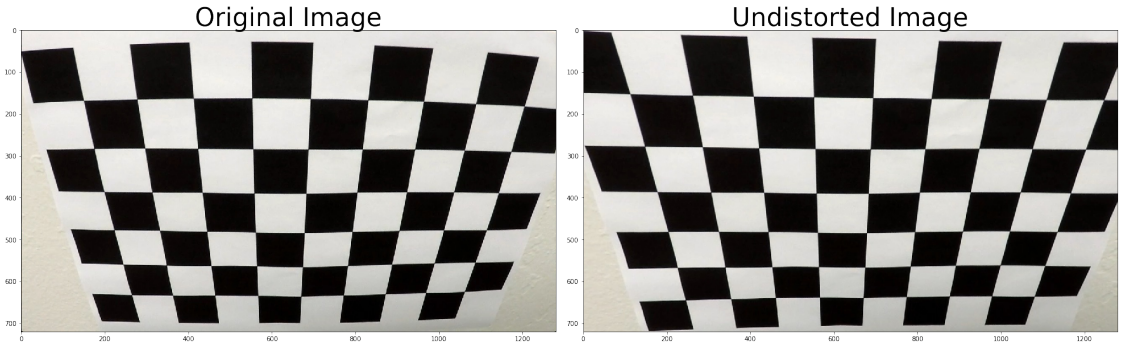
2.2.1 Example of Removing Distortion from an Image

Here I take one example chessboard image and apply the undistort function on it.

```
In [5]: img = mpimg.imread(calibration_images[1])
        undistorted = cal_undistort(img, objpoints, imgpoints)
        mpimg.imsave("output_images/undistorted.jpg", undistorted)
```

```
In [9]: f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()
        ax1.imshow(img)
```

```
    ax1.set_title('Original Image', fontsize=40)
    ax2.imshow(undistorted)
    ax2.set_title('Undistorted Image', fontsize=40)
    plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
    plt.show()
```



3 Color and Gradient Thresholds

There are numerous ways to use color transforms and gradient thesholds to highlight lane lines. I'll show examples after defining each function.

3.0.1 HLS Color Space and Threshold

By exploring different color spaces, like HLS, we can more accurately find lane lines in a variety of conditions. The Saturation channel in HLS does well to highlight lane lines, even in shadow. With a well-chosen threshold, we can adeptly pick out lane lines.

```
In [10]: def hls_select(img, thresh=(0, 255)):
             # Convert to HLS color space
             hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
             # Apply a threshold to the S channel
             S = hls[:,:,2]
             # Return a binary image of threshold result
             binary_output = np.zeros_like(S) # placeholder line
             binary_output[(S > thresh[0]) & (S <= thresh[1])] = 1
             return binary_output
```

3.0.2 Sobel Filter

The Sobel filter works very well to detect and emphasize edges in the image. It can operate in either the x or the y direction.

```
In [11]: def abs_sobel_threshold(img, orient='x', sobel_kernel=3, thresh=(0, 255)):
             o = {"x" : 0, "y" : 0}
             o[orient] += 1
```

```
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    sobel = cv2.Sobel(gray, cv2.CV_64F, o['x'], o['y'], ksize=sobel_kernel)
    abs_sobel = np.absolute(sobel)
    scaled_sobel = np.uint8(255*abs_sobel/np.max(abs_sobel))
    binary_output = np.zeros_like(scaled_sobel)
    binary_output[(scaled_sobel >= thresh[0]) & (scaled_sobel <= thresh[1])] = 1
    return binary_output
```

3.0.3 The Overall Magnitude of the Gradient

Although the x-gradient clearly shows the lane lines, we can detect the lines in the y-gradient as well, so we can apply a threshold to the overall magnitude of the gradient, in both x and y.

```
In [12]: def mag_threshold(img, sobel_kernel=3, mag_thresh=(0, 255)):
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    sobel = {"x" : cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel), "y" : cv2.Sob
    mag = np.sqrt(np.power(sobel["x"],2) + np.power(sobel["y"],2))
    scaled_sobel = np.uint8(255*mag/np.max(mag))
    binary_output = np.zeros_like(scaled_sobel)
    binary_output[(scaled_sobel >= mag_thresh[0]) & (scaled_sobel <= mag_thresh[1])] =
    return binary_output
```

3.0.4 Directional Threshold

Since lane lines are basically vertical, a directional gradient has the potential to isolate them even further, as taking the gradient in the x direction emphasizes edges that are near vertical.

```
In [13]: def dir_threshold(img, sobel_kernel=3, thresh=(0, np.pi/2)):
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    sobel = {"x" : cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel), "y" : cv2.Sob
    direction = np.arctan2(np.absolute(sobel['y']), np.absolute(sobel['x']))
    binary_output = np.zeros_like(direction)
    binary_output[(direction >= thresh[0]) & (direction <= thresh[1])] = 1
    return binary_output
```

3.0.5 Examples of Various Color and Gradient Thresholds

```
In [17]: test_images = glob.glob('test_images/*.jpg')
         test_im = mpimg.imread(test_images[5])

In [18]: # Correcting camera lens distortion for the sample image:
         image = cal_undistort(test_im, objpoints, imgpoints)
         mpimg.imsave("output_images/undistorted_lanes.jpg", image)

         saturation = hls_select(image, thresh=(170, 255))
         mpimg.imsave("output_images/saturation.jpg", saturation)

         x_gradient = abs_sobel_threshold(image, orient='x', sobel_kernel=7, thresh=(50, 200))
         mpimg.imsave("output_images/x_gradient.jpg", x_gradient)
```

```
y_gradient = abs_sobel_threshold(image, orient='y', sobel_kernel=7, thresh=(50, 200))
mpimg.imsave("output_images/y_gradient.jpg", y_gradient)

magnitude = mag_threshold(image, sobel_kernel=9, mag_thresh=(30,100))
mpimg.imsave("output_images/magnitude.jpg", magnitude)

direction = dir_threshold(image, sobel_kernel=15, thresh=(0.7, 1.3))
mpimg.imsave("output_images/direction.jpg", direction)
```

In [19]:
```
# Plot several options
f, (ax1, ax2, ax3, ax4, ax5) = plt.subplots(1, 5, figsize=(24, 9))
f.tight_layout()
ax1.imshow(image)
ax1.set_title('Original Image', fontsize=30)
ax2.imshow(saturation, cmap='gray')
ax2.set_title('Thresholded S', fontsize=30)
ax3.imshow(x_gradient, cmap='gray')
ax3.set_title('Abs Sobel Threshold', fontsize=30)
ax4.imshow(magnitude, cmap='gray')
ax4.set_title('Mag Threshold', fontsize=30)
ax5.imshow(direction, cmap='gray')
ax5.set_title('Directional Threshold', fontsize=30)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```

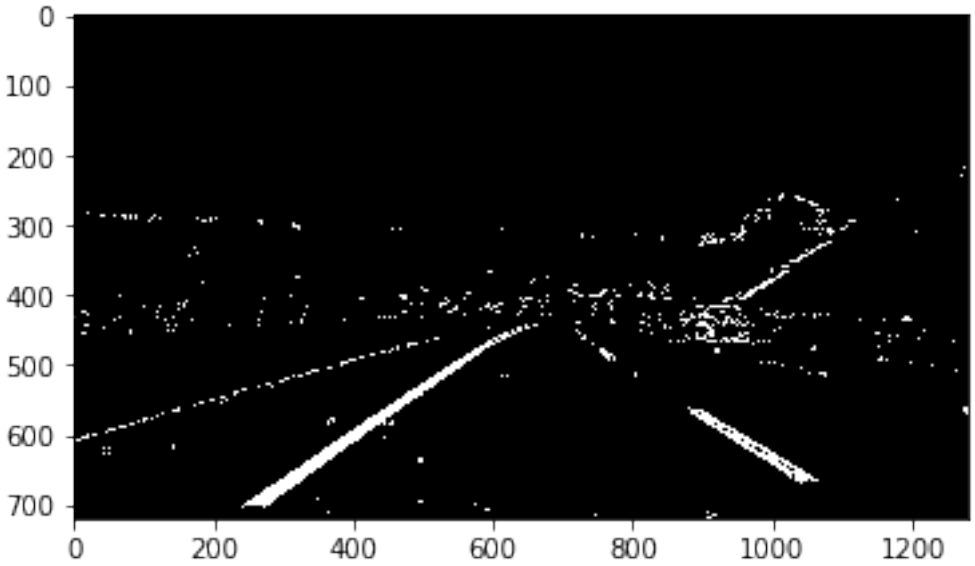
3.1 Color and Gradient Combinations

A combination of absolute Sobel and the saturation channel seem to bring out the lane lines reasonably well. I'm using `sobel_kernel=3, thresh=(50, 200)` for the absolute Sobel, and `thresh=(90, 255)` for the S channel.

In [20]:
```
def color_gradient(img):
    saturation = hls_select(image, thresh=(170, 255))
    x_gradient = abs_sobel_threshold(img, orient='x', sobel_kernel=7, thresh=(50, 200))
    y_gradient = abs_sobel_threshold(img, orient='y', sobel_kernel=7, thresh=(50, 200))
    magnitude = mag_threshold(img, sobel_kernel=9, mag_thresh=(30,100))
    direction = dir_threshold(img, sobel_kernel=15, thresh=(0.7, 1.3))

    combined = np.zeros_like(direction)
    combined[((x_gradient == 1) & (y_gradient == 1)) | ((magnitude == 1) & (direction =

    return combined
```

3.1.1 Color and Gradient Combination Example

```
In [21]: cb = color_gradient(image)
         plt.imshow(cb, cmap='gray')
         mpimg.imsave("output_images/combined_color_gradient.jpg", cb, cmap='gray')
```



4 Perspective Transform

To help identify lane lines with more accuracy, including when there are curves in the road, we can warp the image to have a top-down perspective. I examined a source image and chose four points along the lane lines. I then chose four rectangular points for the transformation.

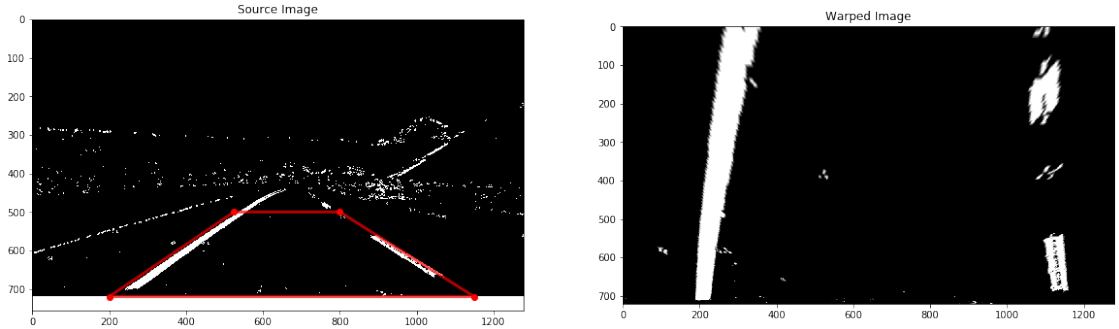
```
In [22]: def warp(img):
             img_size = (img.shape[1], img.shape[0])
             src = np.float32(
                 [
                     [800,500], # top r
                     [1150,720], # bot r
                     [200,720], # bot l
                     [525,500] # top l
                 ]
             )

             dst = np.float32(
                 [
                     [1150, 300], # top r
                     [1150,720], # bot r
                     [175,720], # bot l
```

```
            [175,300] # top l
        ]
    )

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

    return warped, M
```

4.0.1 Warped Image Example

This example shows how, by warping the image, we can show parallel lane markings from a top-down perspective.

```
In [23]: warped_im, M = warp(cb)
         mpimg.imsave('output_images/warped.jpg', warped_im)

         f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
         ax1.set_title('Source Image')
         ax1.imshow(cb, cmap='gray')
         points =          [
                    [800,500], # top r
                    [1150,720], # bot r
                    [200,720], # bot l
                    [525,500] # top l
                ]
         from shapely.geometry.polygon import LinearRing, Polygon
         poly = Polygon(points)
         x,y = poly.exterior.xy
         ax1.plot(x, y, color='#FF0000', alpha=0.7,
             linewidth=3, solid_capstyle='round', zorder=2)

         for p in points:
             ax1.plot(p[0], p[1], '-ro')
         ax2.set_title('Warped Image')
         ax2.imshow(warped_im, cmap='gray')

Out[23]: <matplotlib.image.AxesImage at 0x7fcfe745fc50>
```



5 Detect Lane Lines

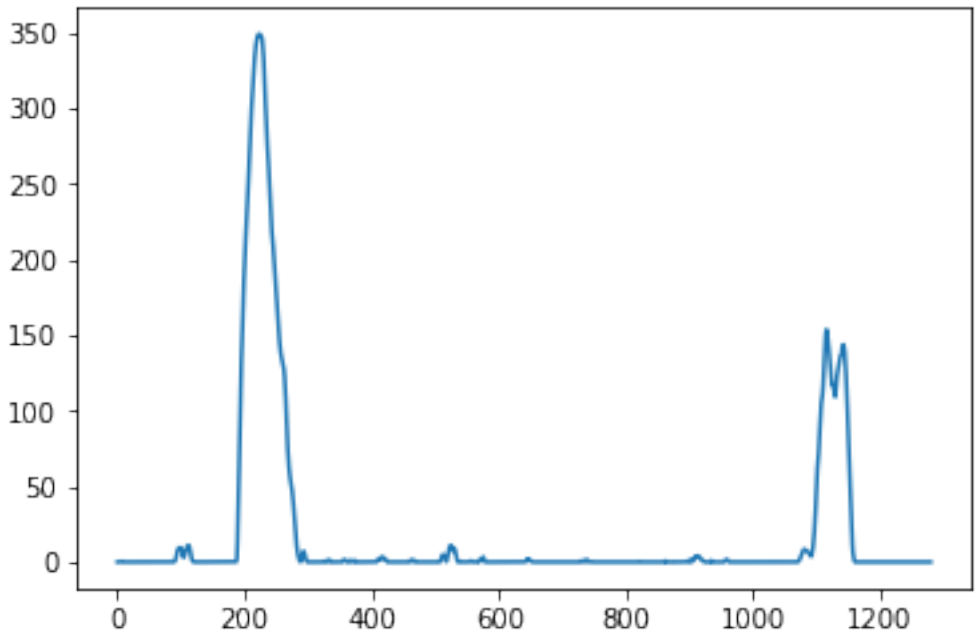
Since lane lines are likely to be mostly vertical nearest to the car, we can just focus on the bottom half of the perspective-warped image and observe the histogram.

```
In [24]: def hist(img):
             # Sum across the bottom half image pixels vertically, (axis=0).
             return np.sum(img[img.shape[0]//2:,:], axis=0)
```

5.0.1 Histogram example

In the example below, the histogram of our warped image very clearly shows the location of the two lane lines.

```
In [25]: histogram = hist(warped_im)

         plt.plot(histogram)
         plt.show()
```




```
In [26]: def find_lane_pixels(img):
             # Take a histogram of the bottom half of the image
             histogram = hist(img)
```

```
# Create an output image to draw on and visualize the result
out_img = np.dstack((img, img, img))
# 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

# 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

# Set height of the windows - based on the number of windows and the image height
window_height = np.int(img.shape[0]//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 later for each window in nwindows
leftx_current = leftx_base
rightx_current = rightx_base

# 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 = img.shape[0] - (window+1)*window_height
    win_y_high = img.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),(0,255,0), 2)
    cv2.rectangle(out_img,(win_xright_low,win_y_low),
    (win_xright_high,win_y_high),(0,255,0), 2)
```

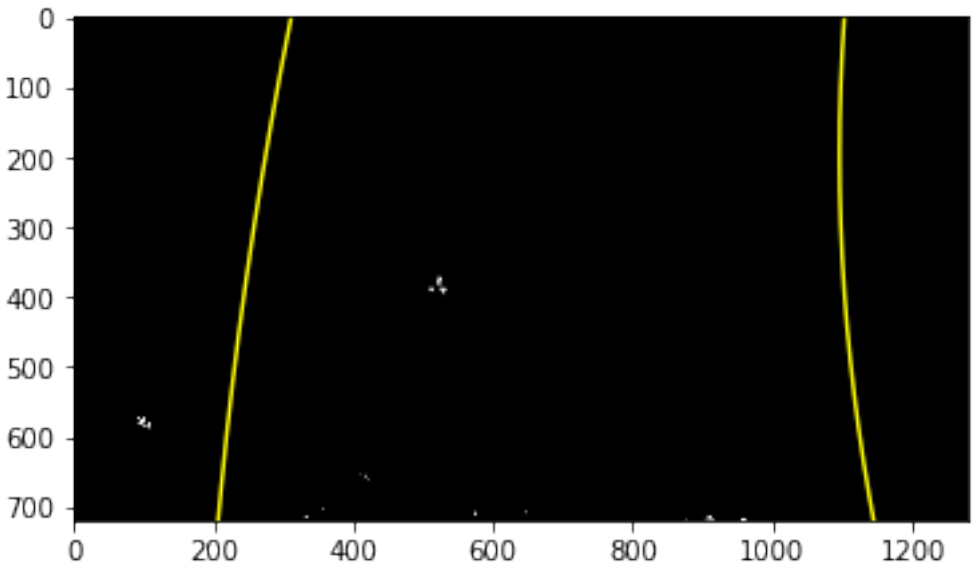
```
            # Identify the nonzero pixels in x and y within the window #
            good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
            (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_high)).nonzero()[0]

            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 (previously was a list of lists of pixels)
        try:
            left_lane_inds = np.concatenate(left_lane_inds)
            right_lane_inds = np.concatenate(right_lane_inds)
        except ValueError:
            # Avoids an error if the above is not implemented fully
            pass

        # 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]

        return leftx, lefty, rightx, righty, out_img
```

In [27]:
```
def fit_polynomial(img):
    # Find our lane pixels first
    leftx, lefty, rightx, righty, out_img = find_lane_pixels(img)

    # Fit a second order polynomial to each using `np.polyfit`
    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, img.shape[0]-1, img.shape[0] )
    try:
        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]
    except TypeError:
        # Avoids an error if `left` and `right_fit` are still none or incorrect
```

```
            print('The function failed to fit a line!')
            left_fitx = 1*ploty**2 + 1*ploty
            right_fitx = 1*ploty**2 + 1*ploty

        ## Visualization ##
        # Colors in the left and right lane regions
        out_img[lefty, leftx] = [255, 0, 0]
        out_img[righty, rightx] = [0, 0, 255]

        # Plots the left and right polynomials on the lane lines
        plt.plot(left_fitx, ploty, color='yellow')
        plt.plot(right_fitx, ploty, color='yellow')

        return left_fit, right_fit, ploty, out_img
```

In [28]:
```
left_fit, right_fit, ploty, out_img = fit_polynomial(warped_im)

plt.imshow(out_img)
```

Out[28]: <matplotlib.image.AxesImage at 0x7fcfe7165d68>



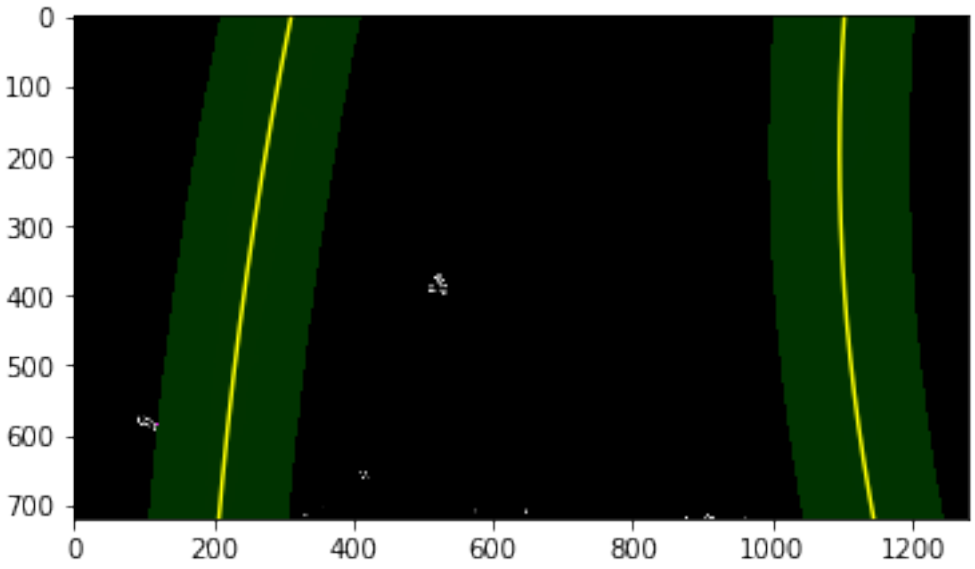
In [29]:
```
def fit_poly(img_shape, leftx, lefty, rightx, righty):
     ### Fit a second order polynomial to each with np.polyfit() ###
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)
    # Generate x and y values for plotting
    ploty = np.linspace(0, img_shape[0]-1, img_shape[0])
    ### TO-DO: Calc both polynomials using ploty, left_fit and right_fit ###
```

```
    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]

    return left_fitx, right_fitx, ploty

def search_around_poly(img, left_fit, right_fit):
    # HYPERPARAMETER
    # Choose the width of the margin around the previous polynomial to search
    margin = 100

    # Grab activated pixels
    nonzero = img.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    ### Set the area of search based on activated x-values ###
    ### within the +/- margin of our polynomial function ###
    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]*nonzeroy
                    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 new polynomials
    left_fitx, right_fitx, ploty = fit_poly(img.shape, leftx, lefty, rightx, righty)

    ## Visualization ##
    # Create an image to draw on and an image to show the selection window
    out_img = np.dstack((img, img, img))*255
    window_img = np.zeros_like(out_img)
    # Color in left and right line pixels
    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]

    # Generate a polygon to illustrate the search window area
    # And recast the x and y points into usable format for cv2.fillPoly()
    left_line_window1 = np.array([np.transpose(np.vstack([left_fitx-margin, ploty]))])
    left_line_window2 = np.array([np.flipud(np.transpose(np.vstack([left_fitx+margin,
                                ploty])))])
    left_line_pts = np.hstack((left_line_window1, left_line_window2))
    right_line_window1 = np.array([np.transpose(np.vstack([right_fitx-margin, ploty]))]
```

```
            right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx+margin,
                                    ploty])))])
            right_line_pts = np.hstack((right_line_window1, right_line_window2))

            # Draw the lane onto the warped blank image
            cv2.fillPoly(window_img, np.int_([left_line_pts]), (0,255, 0))
            cv2.fillPoly(window_img, np.int_([right_line_pts]), (0,255, 0))
            result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)

            # Plot the polynomial lines onto the image
            plt.plot(left_fitx, ploty, color='yellow')
            plt.plot(right_fitx, ploty, color='yellow')
            ## End visualization steps ##

            return left_fitx, right_fitx, result
```

In [30]:
```
plt.imshow( search_around_poly(warped_im, left_fit, right_fit)[2] )
```

Out[30]: <matplotlib.image.AxesImage at 0x7fcfe7b8dda0>



6 Measure Lane Curvature

In [31]:
```
def curvature(img, left_fit, right_fit):
    ploty = np.linspace(0, img.shape[0]-1, img.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]
```

```
            # 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
            y_eval = np.max(ploty)

            # Fit new polynomials to x,y in world space
            left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)
            right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*xm_per_pix, 2)

            # Calculation of R_curve (radius of curvature)
            left_curvature =  ((1 + (2*left_fit_cr[0] *y_eval*ym_per_pix + left_fit_cr[1])**2)
            right_curvature = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)

            # Calculate vehicle center
            left_lane_bottom = (left_fit[0]*y_eval)**2 + left_fit[0]*y_eval + left_fit[2]
            right_lane_bottom = (right_fit[0]*y_eval)**2 + right_fit[0]*y_eval + right_fit[2]
            lane_center = (left_lane_bottom + right_lane_bottom)/2.
            center = (lane_center - img.shape[1]//2) * xm_per_pix

            return left_curvature, right_curvature, center
```

```
In [32]: # Calculate the radius of curvature in meters for both lane lines
         left_curvature, right_curvature, center = curvature(warped_im, left_fit, right_fit)

         print('left', left_curvature, 'm\nright', right_curvature, 'm\ncenter', center, 'm')
```

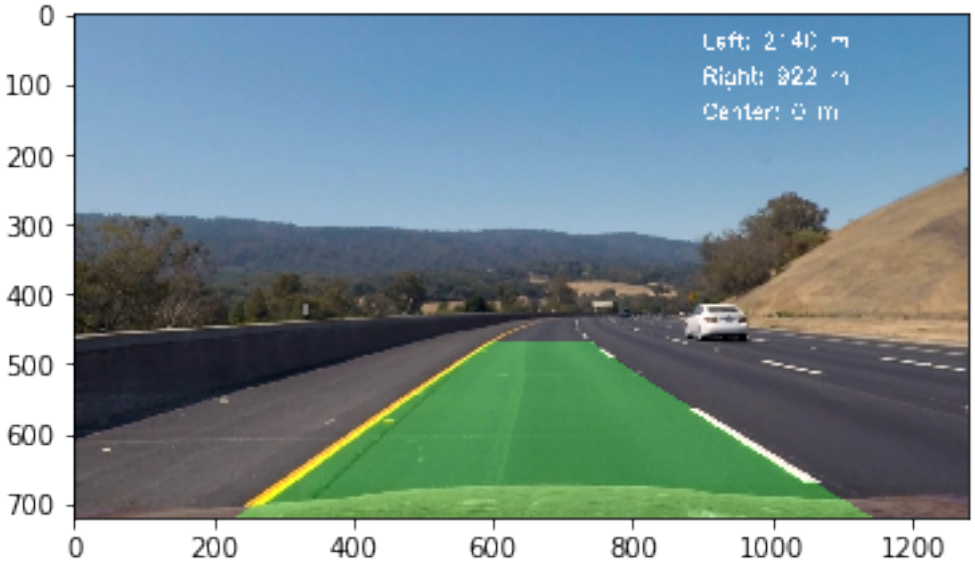
```
left 2140.48601412 m
right 922.41246691 m
center 0.35369111021 m
```

6.1 Display the Result on the Frame

```
In [33]: def lane_display(undist, img, left_fit, right_fit, M, left_curvature, right_curvature,
            ploty = np.linspace(0, img.shape[0]-1, img.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]

            # Create an image to draw the lines on
            warp_zero = np.zeros_like(img).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
```

```
        # cv.FillPoly(img, polys, color, lineType=8, shift=0)
        cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))

        # Warp the blank back to original image space using inverse perspective matrix (Min
        newwarp = cv2.warpPerspective(color_warp, np.linalg.inv(M), (undist.shape[1], img.s

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

        font_family = cv2.FONT_HERSHEY_SIMPLEX

        # cv2.putText(img, text, org, fontFace, fontScale, color[, thickness[, lineType[, b
        cv2.putText(result, 'Left: {:.0f} m'.format(left_curvature), (900, 50), font_family
        cv2.putText(result, 'Right: {:.0f} m'.format(right_curvature), (900, 100), font_fam
        cv2.putText(result, 'Center: {:.0f} m'.format(center), (900, 150), font_family, 1,

        return result
```

In [35]:
```
output_img = lane_display(image, warped_im, left_fit, right_fit, M, left_curvature, rig
plt.imshow(output_img)
mpimg.imsave('output_images/lane_drawn.jpg', output_img)
```



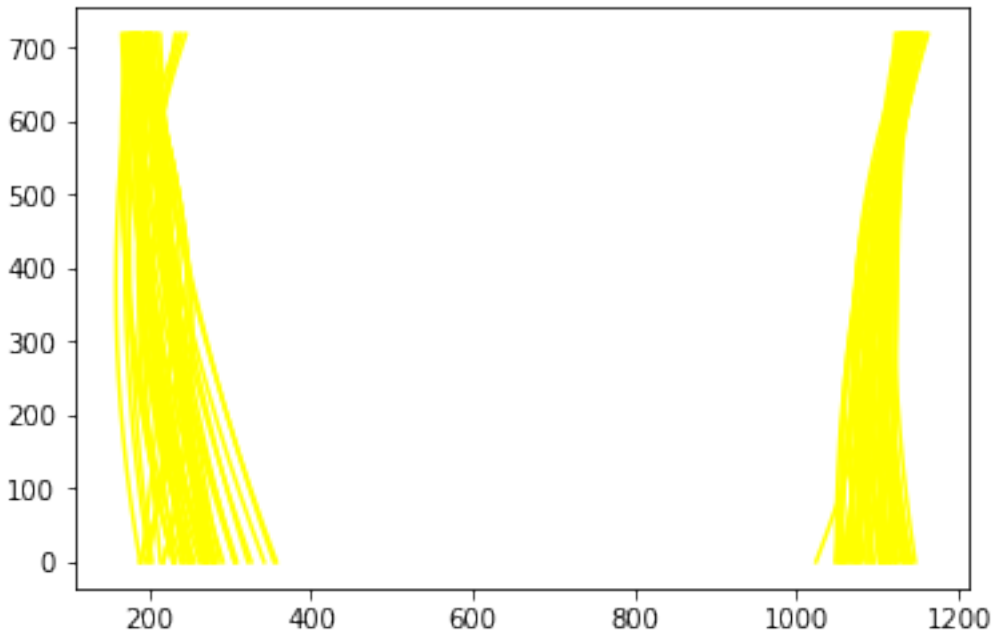
In [54]:
```
def pipeline(frame):
    undistorted = cal_undistort(frame, objpoints, imgpoints)
    binary_threshold = color_gradient(undistorted)
    warped, M = warp(binary_threshold)
    l_fit, r_fit, ploty, fitted = fit_polynomial(warped)
    left_curvature, right_curvature, center = curvature(warped, l_fit, r_fit)
```

```
            output_frame = lane_display(frame, warped, l_fit, r_fit, M, left_curvature, right_c

            return output_frame
```

In [55]:
```
# Import everything needed to edit/save/watch video clips
from moviepy.editor import VideoFileClip
from IPython.display import HTML
```

In [56]:
```
white_output = 'output_images/project_video_solution.mp4'
## To speed up the testing process you may want to try your pipeline on a shorter subcl
## To do so add .subclip(start_second,end_second) to the end of the line below
## Where start_second and end_second are integer values representing the start and end
## You may also uncomment the following line for a subclip of the first 5 seconds
clip1 = VideoFileClip("project_video.mp4").subclip(41,43)
# clip1 = VideoFileClip("project_video.mp4")
white_clip = clip1.fl_image(pipeline) #NOTE: this function expects color images!!
%time white_clip.write_videofile(white_output, audio=False)
```

```
[MoviePy] >>>> Building video output_images/project_video_solution.mp4
[MoviePy] Writing video output_images/project_video_solution.mp4

 98%|| 50/51 [00:59<00:01,  1.30s/it]

[MoviePy] Done.
[MoviePy] >>>> Video ready: output_images/project_video_solution.mp4

CPU times: user 57.4 s, sys: 163 ms, total: 57.5 s
Wall time: 1min 3s
```



```
In [57]: HTML("""
         <video width="960" height="540" controls>
           <source src="{0}">
         </video>
         """.format(white_output))

Out[57]: <IPython.core.display.HTML object>

In [ ]:
```