Self-Driving Car

Project 4: Advanced Lane Finding

Kimon Roufas

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

In [1]:

```
#importing some useful packages
# import matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import cv2
import glob
import pprint
from random import randint
from collections import deque
%matplotlib inline

# Import everything needed to edit/save/watch video clips
from moviepy.editor import VideoFileClip, ImageSequenceClip
from IPython.display import HTML

print('Numpy version: {}'.format(np.__version__))
print('OpenCV version: {}'.format(cv2.__version__))
# print('MatPlotLib version: {}'.format(matplotlib.__version__)) # v2.0.2
```

```
Numpy version: 1.13.1
OpenCV version: 3.1.0
```

Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

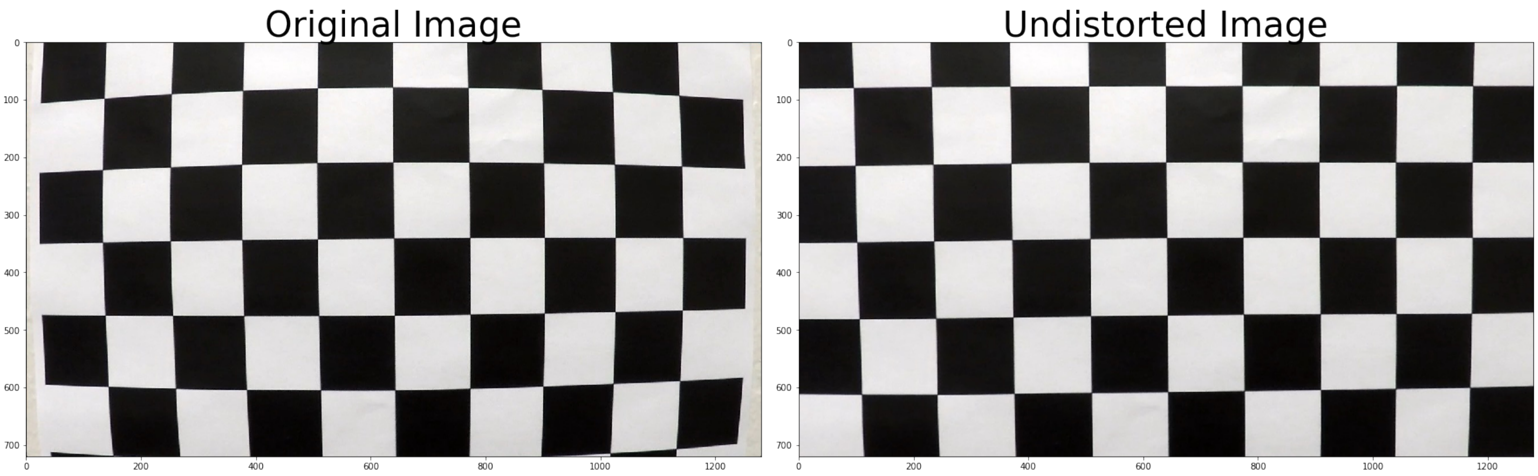
The code for this step is shown below. I wrote a helper function (`get_object_points_and_image_points(images)`) that takes a list of image file names, reads them in and returns the object points and image points. A second function, `calc_camera_coefficients(objpoints, imgpoints)`, calculates the camera calibration coefficients and then `undistort(img, calibration)` will undistort any image.

One of the calibration images did not contain all 6 rows of corners so I had to eliminate it from the calibration set.

In [2]:

```
def get_object_points_and_image_points(images):
    # arrays to store object points and image points from all teh images
    objpoints = [] # 3D points in real world space
    imgpoints = [] # 2D points in image plane

    # Prepare object points, like (0,0,0), (1, 0, 0), (2, 0, 0), etc...
    nx, ny = (9, 6)
    objp = np.zeros((nx*ny, 3), np.float32)
    objp[:,:2] = np.mgrid[0:nx, 0:ny].T.reshape(-1,2) # x, y coordinates

    for fname in images:
        # read in each image
        img = mpimg.imread(fname)
        # convert image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        # find the chessboard corners
        ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)
        # if corners are found, add the object points, image points
        if ret == True:
            imgpoints.append(corners)
            objpoints.append(objp)
        return (objpoints, imgpoints)


def calc_camera_coefficients(img, objpoints, imgpoints):
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img
.shape[1::-1], None, None)
    return (mtx, dist) if ret else None


def undistort(img, calibration):
    mtx, dist = calibration
    undist = cv2.undistort(img, mtx, dist, None, mtx)
```

```
    return undist



def display_two_images(img1, title1, img2, title2):
    f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
    f.tight_layout()
    ax1.set_title(title1, fontsize=40)
    ax1.imshow(img1)
    ax2.set_title(title2, fontsize=40)
    ax2.imshow(img2)
    plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)


# images = glob.glob('./camera_cal/calibration*.jpg')
images = [
    './camera_cal/calibration2.jpg',
    './camera_cal/calibration3.jpg',
    './camera_cal/calibration4.jpg',
    './camera_cal/calibration5.jpg',
    './camera_cal/calibration6.jpg',
    './camera_cal/calibration7.jpg',
    './camera_cal/calibration8.jpg',
    './camera_cal/calibration9.jpg',
    './camera_cal/calibration10.jpg',
    './camera_cal/calibration11.jpg',
    './camera_cal/calibration12.jpg',
    './camera_cal/calibration13.jpg',
    './camera_cal/calibration14.jpg',
    './camera_cal/calibration15.jpg',
    './camera_cal/calibration16.jpg',
    './camera_cal/calibration17.jpg',
    './camera_cal/calibration18.jpg',
    './camera_cal/calibration19.jpg',
    './camera_cal/calibration20.jpg'
]
objpoints, imgpoints = get_object_points_and_image_points(images)

# show a sample image that has been undistorted using the calibration coefficien
ts
img = mpimg.imread('./camera_cal/calibration1.jpg')
calibration = calc_camera_coefficients(img, objpoints, imgpoints)
undistorted = undistort(img, calibration)
display_two_images(img, 'Original Image', undistorted, 'Undistorted Image')
```



Helper Functions

In [18]:

```
images = [
    './test_images/straight_lines1.jpg',
    './test_images/straight_lines2.jpg',
    './test_images/test1.jpg',
    './test_images/test2.jpg',
    './test_images/test3.jpg',
    './test_images/test4.jpg',
    './test_images/test5.jpg'
]
test_images = []
for fname in images:
    test_images.append(mpimg.imread(fname))
straight_lines1, straight_lines2, test1, test2, test3, test4, test5 = test_image
s



font                    = cv2.FONT_HERSHEY_SIMPLEX
fontScale               = 1
fontColor               = (255,255,255)
lineType                = 2

def image_print(image, line_num, text, color=fontColor):
    cv2.putText(image,text, (10, 40+((line_num-1)*40)), font, fontScale, color,
lineType)


def threshold_binary(img, s_thresh=(170, 255), sx_thresh=(20, 100)):
    img = np.copy(img)
    # Convert to HLS color space and separate the V channel
    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS).astype(np.float)
    l_channel = hls[:,:,1]
    s_channel = hls[:,:,2]
    # Sobel x
    sobelx = cv2.Sobel(l_channel, cv2.CV_64F, 1, 0) # Take the derivative in x
    abs_sobelx = np.absolute(sobelx) # Absolute x derivative to accentuate lines
away from horizontal
    scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))
```

```
    scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))

    # Threshold x gradient
    sxbinary = np.zeros_like(scaled_sobel)
    sxbinary[(scaled_sobel >= sx_thresh[0]) & (scaled_sobel <= sx_thresh[1])] =
1

    # Threshold color channel
    s_binary = np.zeros_like(s_channel)
    s_binary[(s_channel >= s_thresh[0]) & (s_channel <= s_thresh[1])] = 1
    # Stack each channel
    # Note color_binary[:, :, 0] is all 0s, effectively an all black image. It m
ight
    # be beneficial to replace this channel with something else.
    color_binary = np.dstack(( np.zeros_like(sxbinary), sxbinary, s_binary))
    # Combine the two binary thresholds
    combined_binary = np.zeros_like(sxbinary)
    combined_binary[(s_binary == 1) | (sxbinary == 1)] = 1
    return (color_binary, combined_binary)


def get_warper_coordinates(img_shape):
    max_x, max_y = (img_shape[1], img_shape[0])
    src = np.float32(
        [[max_x / 2 - 60, max_y / 2 + 100],
        [((max_x / 6) - 0), max_y],
        [(max_x * 5 / 6) + 50, max_y],
        [(max_x / 2 + 60), max_y / 2 + 100]])
    dst = np.float32(
        [[(max_x / 4), 0],
        [(max_x / 4), max_y],
        [(max_x*3/4), max_y],
        [(max_x*3/4), 0]])
    return (src, dst)


def warp_transform(img_shape):
    src, dst = get_warper_coordinates(img_shape)
    M = cv2.getPerspectiveTransform(src, dst)
    return M


def warp_inverse_transform(img_shape):
    src, dst = get_warper_coordinates(img_shape)
    Minv = cv2.getPerspectiveTransform(dst, src)
    return Minv


def warper(img, M):
    img_size = (img.shape[1], img.shape[0])
    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)
    return warped
```

```
def draw_lines(img:np.array, pts:np.array):

    new_img = np.copy(img)
    cv2.polylines(new_img, np.int32([pts]), False, (255,0,0), 3)
    return new_img


frame_averaging_length = 5
# max_bad_lines = 5

# Define a class of to hold useful information relevant to the frames being anal
ysed
class Lanes():
    def __init__(self):
        self.left = Line()
        self.right = Line()
        self.frame_index = 0
        self.detected = False
        self.detected_count = [0]
        self.good_frames = []
        self.bad_frames = []
        self.good_binary_warped_frames = []
        self.histogram = []
        self.curvature_diff = []
        self.inv_curvature_diff = []
        self.center_distance = []
        self.width = []


# Define a class to receive the characteristics of each line detection
class Line():
    def __init__(self):
        # was the line detected in the last iteration?
        self.detected = False
        # x values of the last n fits of the line
        self.recent_xfitted = deque(maxlen=frame_averaging_length)
        #average x values of the fitted line over the last n iterations
        self.bestx = None
        #polynomial coefficients averaged over the last n iterations
        self.best_fit = None


        #polynomial coefficients for the most recent fit
        self.current_fit = [np.array([False])]  # *** DONE ***
        #radius of curvature of the line in some units
        self.radius = None # *** DONE ***
        #distance in meters of vehicle center from the line
        self.line_base_pos = None # *** DONE ***


        #difference in fit coefficients between last and new fits
        self.diffs = np.array([0,0,0], dtype='float') # TODO: POPULATE THIS TO D
O A SANITY CHECK BETWEEN FRAMES
        #x values for detected line pixels
```

```
        self.allx = None # *** DONE ***

        #y values for detected line pixels
        self.ally = None # *** DONE ***


def clear_state():
    global lanes
    lanes = Lanes()

clear_state()
```

Pipeline (single images)

1. Provide an example of a distortion-corrected image.

Here it is!

In [4]:

```
undistorted1 = undistort(test1, calibration)
display_two_images(test1, 'Original Image', undistorted1, 'Undistorted Image')
```

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

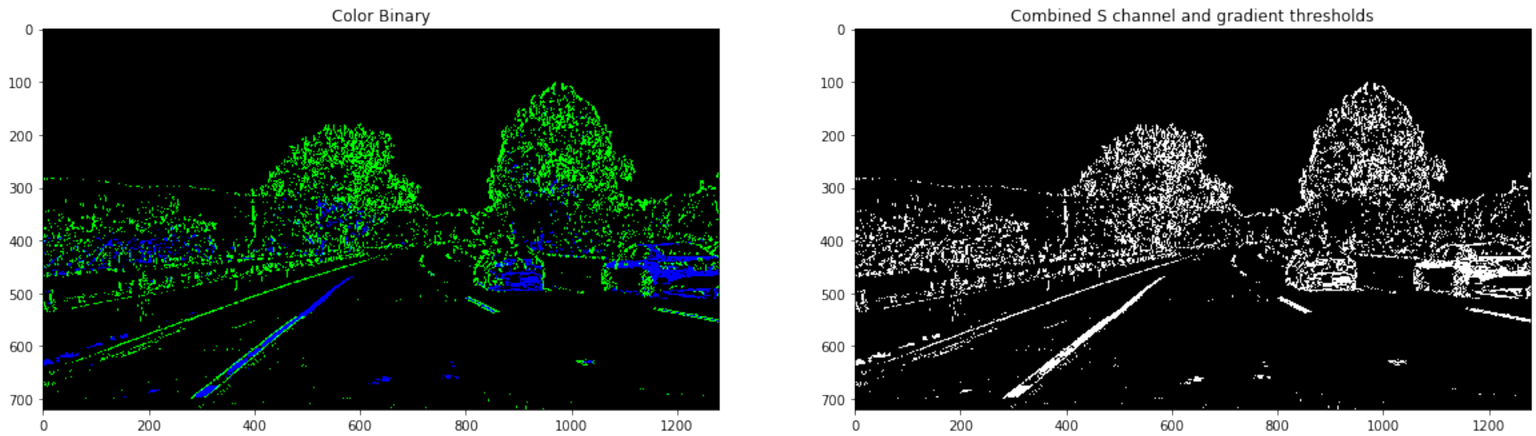
I used a combination of color and gradient thresholds to generate a binary image (see `threshold_binary()` above). Here's an example of my output for this step. The left image uses two colors to show the contribution of the gradient and color thresholding separately. The right image combines the two for the final result.

In [5]:

```
undistorted = undistort(test4, calibration)
color_binary, threshold_binary_image = threshold_binary(undistorted)

# Plotting thresholded images
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.set_title('Color Binary')
ax1.imshow(color_binary)
ax2.set_title('Combined S channel and gradient thresholds')
ax2.imshow(threshold_binary_image, cmap='gray')
```

Out[5]:

```
<matplotlib.image.AxesImage at 0x1159be358>
```



3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform includes a couple functions breaking apart the problem into getting the perspective change source and destination coordinates (`get_warper_coordinates()`), and warping an image (`warper()`). The definitions of these functions are above in the **Helper Functions** section. An additional helper to draw the lines (`draw_lines()`) from the coordinates was written as well.

These functions were used to generate the example images below. The coordinates used for the transformation are also shown below. I verified that my perspective transform was working correctly by adding the lines to copies of the images

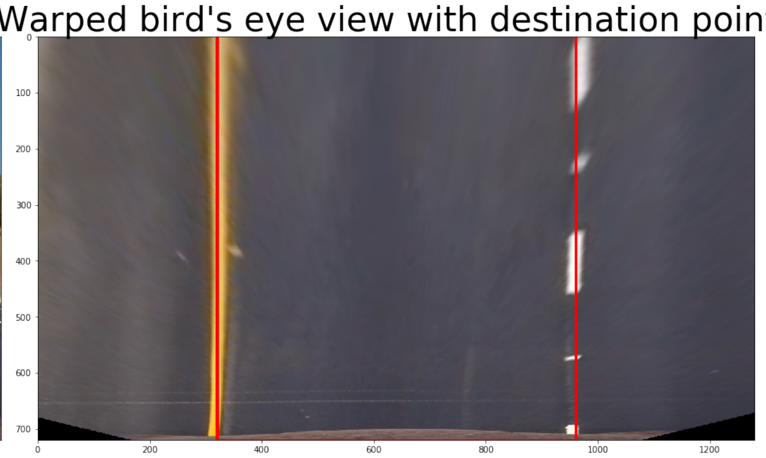
In [6]:

```
test_image = np.copy(straight_lines1)
src, dst = get_warper_coordinates(test_image.shape)
M = warp_transform(test_image.shape)
Minv = warp_inverse_transform(test_image.shape)

print('Source coordinates (src):')
pprint.pprint(np.int32(src))
print('\nDestination coordinates (dst):')
pprint.pprint(np.int32(dst))

undistorted = undistort(test_image, calibration)
undistorted_with_lines = draw_lines(undistorted, src)
warped = warper(undistorted, M)
warped_with_lines = draw_lines(warped, dst)

# display the test images
display_two_images(undistorted_with_lines, 'Undistorted Image', warped_with_line
s, 'Warped bird\'s eye view with destination points')
```

```
Source coordinates (src):
array([[ 580,  460],
       [ 213,  720],
       [1116,  720],
       [ 700,  460]], dtype=int32)

Destination coordinates (dst):
array([[320,   0],
       [320, 720],
       [960, 720],
       [960,   0]], dtype=int32)
```

Undistorted Image

Warped bird's eye view with destination points



4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

At this point I am unsure which of the two methods discussed in the class will obtain the best lane line polynomials. So I implemented both of them. I removed the second method to save space, but it can be seen in the other jupyter notebook file in the repo.

An example image showing the result follows the code.

In [7]:

```
# some parameters

# 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
```

In [34]:

```
def find_and_fit_polynomial_lanes(binary_warped, draw_boxes=False):
    lanes.frame_index += 1
    # Assuming you have created a warped binary image called "binary_warped"
    # Histogram of the bottom half of the image
    histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)
    lanes.histogram.append(histogram)

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

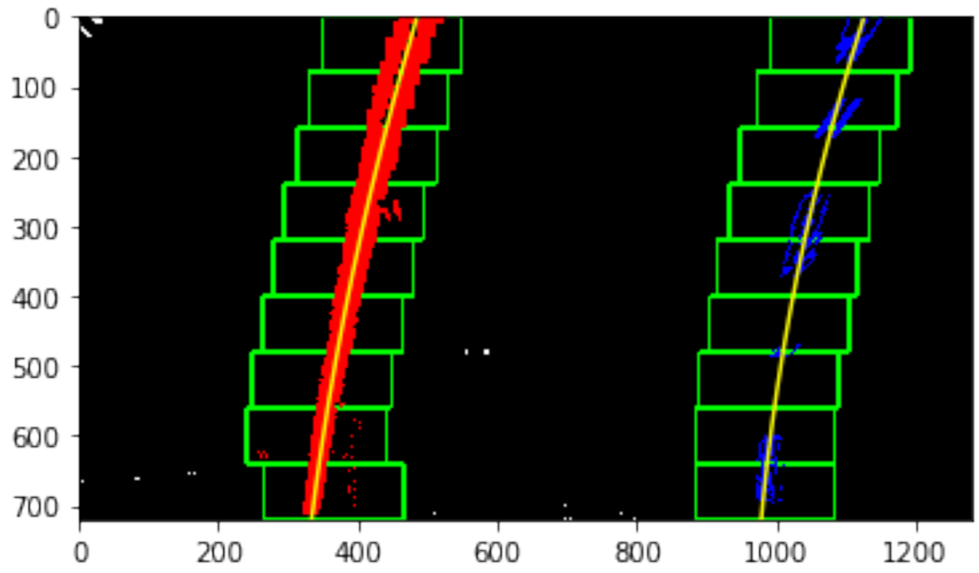
```
    left_lane_inds = []
    right_lane_inds = []

    # 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
        if draw_boxes:
            cv2.rectangle(out_img,(win_xleft_low,win_y_low),(win_xleft_high,win_
y_high), (0,255,0), 3)
            cv2.rectangle(out_img,(win_xright_low,win_y_low),(win_xright_high,wi
n_y_high), (0,255,0), 3)
        # 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_hi
gh)).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 posit
ion
        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
    lanes.left.current_fit = np.polyfit(lefty, leftx, 2) # left_fit
    lanes.right.current_fit = np.polyfit(righty, rightx, 2) # right_fit

    left_fit_in_meters = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
    right_fit_in_meters = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix, 2)

    generate_x_and_y_from_polylines(lanes.left.current_fit, lanes.right.current_
fit)
```

```
    return (out_img, left_fit_in_meters, right_fit_in_meters, nonzeroy, nonzerox
, left_lane_inds, right_lane_inds)


def fit_polynomial_lanes(binary_warped):
    lanes.frame_index += 1
    # 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])
    margin = 100

    (A, B, C) = lanes.left.current_fit
    left_lane_inds = ((nonzerox > (A*(nonzeroy**2) + B*nonzeroy + C - margin)) &
                      (nonzerox < (A*(nonzeroy**2) + B*nonzeroy + C + margin)))

    (A, B, C) = lanes.right.current_fit
    right_lane_inds = ((nonzerox > (A*(nonzeroy**2) + B*nonzeroy + C - margin))
&
                       (nonzerox < (A*(nonzeroy**2) + B*nonzeroy + C + 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
    lanes.left.current_fit = np.polyfit(lefty, leftx, 2) # left_fit
    lanes.right.current_fit = np.polyfit(righty, rightx, 2) # right_fit

    # Generate x and y values for plotting
    # lanes.left.allx = left_fitx
    # lanes.right.allx = right_fitx
    # lanes.left.ally = lanes.right.ally = plot_y
    generate_x_and_y_from_polylines(lanes.left.current_fit, lanes.right.current_
fit)

    return None


# Generate x and y values for plotting
def generate_x_and_y_from_polylines(left_fit, right_fit):
    plot_y = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )

    (A, B, C) = left_fit
    left_fitx = A * plot_y**2 + B * plot_y + C

    (A, B, C) = right_fit
    right_fitx = A * plot_y**2 + B * plot_y + C
```

```
    lanes.left.allx = left_fitx

    lanes.right.allx = right_fitx
    lanes.left.ally = lanes.right.ally = plot_y
#     return (left_fit_x, right_fit_x, plot_y)


# EXAMPLE IMAGE
test_image = np.copy(test3)
# test_image = debug_image
M = warp_transform(test_image.shape)
undistorted = undistort(test_image, calibration)
color_binary, threshold_binary_image = threshold_binary(undistorted)
binary_warped = warper(threshold_binary_image, M)

out_img, left_fit_in_meters, right_fit_in_meters, \
nonzeroy, nonzerox, left_lane_inds, right_lane_inds = find_and_fit_polynomial_la
nes(binary_warped, draw_boxes=True)


# left_fitx, right_fitx, ploty = generate_x_and_y_from_polylines(left_fit, right
_fit)

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(lanes.left.allx, lanes.left.ally, color='yellow')
plt.plot(lanes.right.allx, lanes.right.ally, color='yellow')
plt.xlim(0, 1280)
plt.ylim(720, 0)
```

Out[34]:

```
(720, 0)
```



5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

Below are helper functions to calculate the radii of curviture in pixel space and in real world space (in meters). Following those, are the calculated values for the last test images above. The values seem reasonable when calculated for a test image.

In [35]:

```
def calculate_curvature_in_pixels(polynomial_fit, y_eval):
    curve_radius = ((1 + (2*polynomial_fit[0]*y_eval + polynomial_fit[1])**2)**1
.5) / np.absolute(2*polynomial_fit[0])
    return curve_radius



def calculate_curvature_in_meters(polynomial_fit_in_meters, y_eval):
    curve_radius = ((1 + (2*polynomial_fit_in_meters[0]*y_eval*ym_per_pix + poly
nomial_fit_in_meters[1])**2)**1.5) / np.absolute(2*polynomial_fit_in_meters[0])
    return curve_radius



def calculate_curvatures(left_polynomial, right_polynomial, img_shape):
    max_x, max_y = (img_shape[1], img_shape[0])
    lanes.left.radius = calculate_curvature_in_meters(left_polynomial, max_y)
    lanes.right.radius = calculate_curvature_in_meters(right_polynomial, max_y)
    lanes.curvature_diff.append(1/lanes.left.radius - 1/lanes.right.radius)



def calculate_lane_metrics(left_polynomial, right_polynomial, img_shape):
    calculate_curvatures(left_polynomial, right_polynomial, img_shape)
    max_x, max_y = (img_shape[1], img_shape[0])
    left_pos = left_polynomial[0]*max_y**2 + left_polynomial[1]*max_y + left_pol
ynomial[2]
    right_pos = right_polynomial[0]*max_y**2 + right_polynomial[1]*max_y + right
_polynomial[2]

    lanes.left.line_base_pos = max_x*xm_per_pix - left_pos
    lanes.right.line_base_pos = right_pos - max_x*xm_per_pix

    lanes.center_distance.append((left_pos + right_pos - max_x) * xm_per_pix / 2
)

    lanes.width.append((right_pos - left_pos) * xm_per_pix)


# PIXEL SPACE EXAMPLE CURVATURE CALCULATION
# Define y-value where we want radius of curvature
# I'll choose the maximum y-value, corresponding to the bottom of the image
img_shape = binary_warped.shape
max_x, max_y = (img_shape[1], img_shape[0])
y_eval = max_y
```

```
left_radius_in_pixels = calculate_curvature_in_pixels(lanes.left.current_fit, y_eval)
right_radius_in_pixels = calculate_curvature_in_pixels(lanes.right.current_fit, y_eval)
print('left: {0:.0f}, right: {1:.0f} (in pixel space)'.format(left_radius_in_pixels, right_radius_in_pixels))
# Example values: 1926.74 1908.48


# REAL WORLD SPACE EXAMPLE CURVATURE CALCULATION
calculate_lane_metrics(lanes.left.current_fit, lanes.right.current_fit, img_shape)
print('left: {0:.1f}m, right: {1:.1f}m (in real world space)'.format(lanes.left.radius, lanes.right.radius))
print('distance from center: {0:.2f}m'.format(lanes.center_distance[-1]))
print('lane width: {0:.2f}m'.format(lanes.width[-1]))
```

```
left: 4376, right: 3261 (in pixel space)
left: 4814.1m, right: 3685.4m (in real world space)
distance from center: 0.09m
lane width: 3.41m
```

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

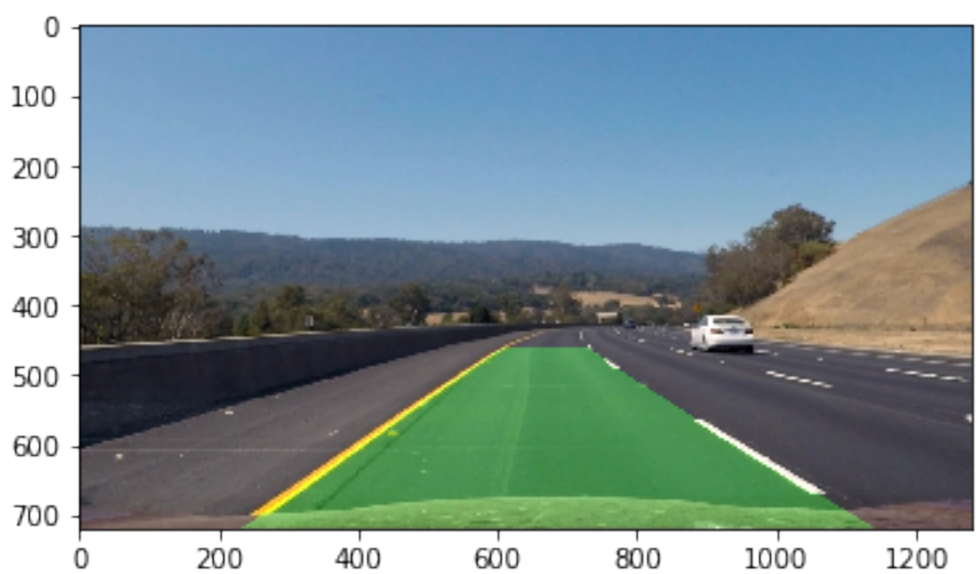
I implemented this step in the function `add_lane_lines_to_image()` shown below. An example follows.

In [37]:

```
def add_lane_lines_to_image(base_image, Minv, left_plot_x, right_plot_x, plot_y)
:
    # Create an image to draw the lines on
    color_warp = np.zeros_like(base_image).astype(np.uint8)

    # Recast the x and y points into usable format for cv2.fillPoly()
    pts_left = np.array([np.transpose(np.vstack([left_plot_x, plot_y]))])
    pts_right = np.array([np.flipud(np.transpose(np.vstack([right_plot_x, plot_y
])))])
    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 matr
ix (Minv)
    newwarp = cv2.warpPerspective(color_warp, Minv, (base_image.shape[1], base_i
mage.shape[0]))
    # Combine the result with the original image
    result = cv2.addWeighted(base_image, 1, newwarp, 0.3, 0)
    return result


result = add_lane_lines_to_image(undistorted, Minv, lanes.left.allx, lanes.right
.allx, lanes.left.ally)
plt.imshow(result)

# print(type(lanes.left.allx))
# print(lanes.left.allx.shape)
```

Out[37]:

<matplotlib.image.AxesImage at 0x117a97d68>



Pipeline

In [29]:

```
clear_state()

max_undetected_frames = 5
min_radius = 300
max_curvature_diff = 0.0005
min_lane_width = 3.0
max_lane_width = 4.0

def sanity_check():
#     result = True
    # check for reasonable curvature
    result = lanes.left.radius > min_radius and lanes.right.radius > min_radius
    # check curvature is similar
    result = abs(lanes.curvature_diff[-1]) < max_curvature_diff and result
    # check for reasonable lane width
    result = (lanes.width[-1] >= min_lane_width) and (lanes.width[-1] <= max_lan
e_width) and result
    # check frame to frame jumping around?

    return result


def annotate_image(image:np.array, color=(255, 255, 255)) -> np.array:
    image_print(image, 1, 'Frame: {0}'.format(lanes.frame_index), color)
    image_print(image, 2, 'Curvature - left: {0:.0f}m, right: {1:.0f}m'.format(l
anes.left.radius, lanes.right.radius), color)
    image_print(image, 3, 'Diff of Inverse Curvatures: {0:.5f}(1/m)'.format(lane
s.curvature_diff[-1]), color)
    image_print(image, 4, 'Center Distance: {0:.2f}m'.format(lanes.center_distan
ce[-1]), color)
    image_print(image, 5, 'Lane Width: {0:.2f}m'.format(lanes.width[-1]), color)

# debug_image = None

def update_best_lane_lines():
    lanes.left.recent_xfitted.append(lanes.left.allx)
    lanes.left.bestx = np.average(lanes.left.recent_xfitted, axis=0)

    lanes.right.recent_xfitted.append(lanes.right.allx)
    lanes.right.bestx = np.average(lanes.right.recent_xfitted, axis=0)


def lane_line_pipeline(image:np.array) -> np.array:
    undistorted = undistort(image, calibration)
    color_binary, threshold_binary_image = threshold_binary(undistorted)
    binary_warped = warper(threshold_binary_image, M)

    if (lanes.detected):
        fit_polynomial_lanes(binary_warped)
    else:
        find_and_fit_polynomial_lanes(binary_warped, draw_boxes=True)
```

```
    img_shape = binary_warped.shape

    calculate_lane_metrics(lanes.left.current_fit, lanes.right.current_fit, img_
shape)

    sanity_check_passed = sanity_check()

    binary_warped_frame = np.dstack((binary_warped, binary_warped, binary_warped
))*255
    annotate_image(out_img, color=(255, 0, 0))

    if sanity_check_passed:
        lanes.detected = True
        lanes.detected_count.append(min(lanes.detected_count[-1]+1, max_undetect
ed_frames))
        update_best_lane_lines()

        result = add_lane_lines_to_image(undistorted, Minv, lanes.left.bestx, la
nes.right.bestx, lanes.left.ally)
        annotate_image(result)

        lanes.good_frames.append(result)
#         lanes.good_binary_warped_frames.append(out_img)
    else:
        lanes.detected = False
        lanes.detected_count.append(max(lanes.detected_count[-1]-1, 0))
#         reuse_best_lane_line()
        result = add_lane_lines_to_image(undistorted, Minv, lanes.left.bestx, la
nes.right.bestx, lanes.left.ally)
        annotate_image(result)

        lanes.bad_frames.append(result)

#     global debug_image
#     if (lanes.frame_index == 42):
#         debug_image = image

    return result
```

Tuning

This is just some test code that I used to inspect and tune based on all of the test images. One of the test images is a bit whacky, but I'm going to try handling it in the video stream before working on tuning parameters.

In [ ]:

```
# f, ax_array = plt.subplots(7, figsize=(30,30))
# for i in range(len(test_images)):
#     image = test_images[i]
#     M = warp_transform(image.shape)
#     Minv = warp_inverse_transform(image.shape)
#     result = lane_line_pipeline(image)
#     ax_array[i].imshow(result)
```

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's a [link to my video result](./output_videos/project_video.mp4)

Project Video

In [31]:

```
clear_state()

filename = 'project_video.mp4'
output_filename = 'output_videos/'+filename
good_output_filename = 'output_videos/good_'+filename
bad_output_filename = 'output_videos/bad_'+filename
good_binary_warped_output_filename = 'output_videos/good_binary_warped_'+filenam
e



#
clip1 = VideoFileClip(filename)
# clip1 = VideoFileClip(filename).subclip(38, 42)
# clip1 = VideoFileClip(filename).subclip(0, 1)

output_clip = clip1.fl_image(lane_line_pipeline) #NOTE: this function expects co
lor images!!
%time output_clip.write_videofile(output_filename, audio=False)

histogram_array_1 = np.array(lanes.histogram)

# if (lanes.good_frames):
#     good_clip = ImageSequenceClip(lanes.good_frames, fps=25)
#     good_clip.write_videofile(good_output_filename)
# else:
#     print('No good frames to save!')

# if (lanes.bad_frames):
#     bad_clip = ImageSequenceClip(lanes.bad_frames, fps=1)
#     bad_clip.write_videofile(bad_output_filename)
# else:
#     print('No bad frames to save!')

# if (lanes.good_binary_warped_frames):
#     good_clip = ImageSequenceClip(lanes.good_binary_warped_frames, fps=25)
#     good_clip.write_videofile(good_binary_warped_output_filename)
# else:
#     print('No good binary warped frames to save!')

# if (debug_image):
#     plt.imshow(debug_image, interpolation='nearest')
```

```
[MoviePy] >>>> Building video output_videos/project_video.mp4
[MoviePy] Writing video output_videos/project_video.mp4

100%|██████████| 1260/1261 [03:04<00:00,  6.64it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: output_videos/project_video.mp4

CPU times: user 3min 29s, sys: 46.6 s, total: 4min 16s
Wall time: 3min 4s
[MoviePy] >>>> Building video output_videos/good_project_video.mp4
[MoviePy] Writing video output_videos/good_project_video.mp4

100%|██████████| 1239/1239 [00:25<00:00, 47.71it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: output_videos/good_project_video.mp4

[MoviePy] >>>> Building video output_videos/bad_project_video.mp4
[MoviePy] Writing video output_videos/bad_project_video.mp4

 96%|█████████▌| 22/23 [00:00<00:00, 130.42it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: output_videos/bad_project_video.mp4

No good frames to save!
```

In [ ]:

```

```

Challenge Video

In [ ]:

```python
# filename = 'challenge_video.mp4'
# output_filename = 'output_videos/'+filename
# clip1 = VideoFileClip(filename)
# clear_state()
# output_clip = clip1.fl_image(lane_line_pipeline) #NOTE: this function expects color images!!
# %time output_clip.write_videofile(output_filename, audio=False)
# histogram_array_2 = np.array(debug.histogram)
```

Harder Challenge Video

In [ ]:

```
# filename = 'harder_challenge_video.mp4'
# output_filename = 'output_videos/'+filename
# clip1 = VideoFileClip(filename)
# clear_state()
# output_clip = clip1.fl_image(lane_line_pipeline) #NOTE: this function expects color images!!
# %time output_clip.write_videofile(output_filename, audio=False)
# histogram_array_3 = np.array(debug.histogram)
```

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

Initially I implemented a technique that searched each frame for the lane lines. I realized that this would be a bit slower than using the lane lines from the previous frame as a starting point. In order to tune the algorithm, I collected collections of frames that didn't pass the sanity check and created separate videos to visually inspect the case. I also plotted some metrics seen below to help me set the thresholds in the sanity check function.

Once I ammended my algorithm to use the last frame's lane lines to filter the image, this helped eliminate a lot of noise and improved the graphs below significantly. It was nice to see the before and after.

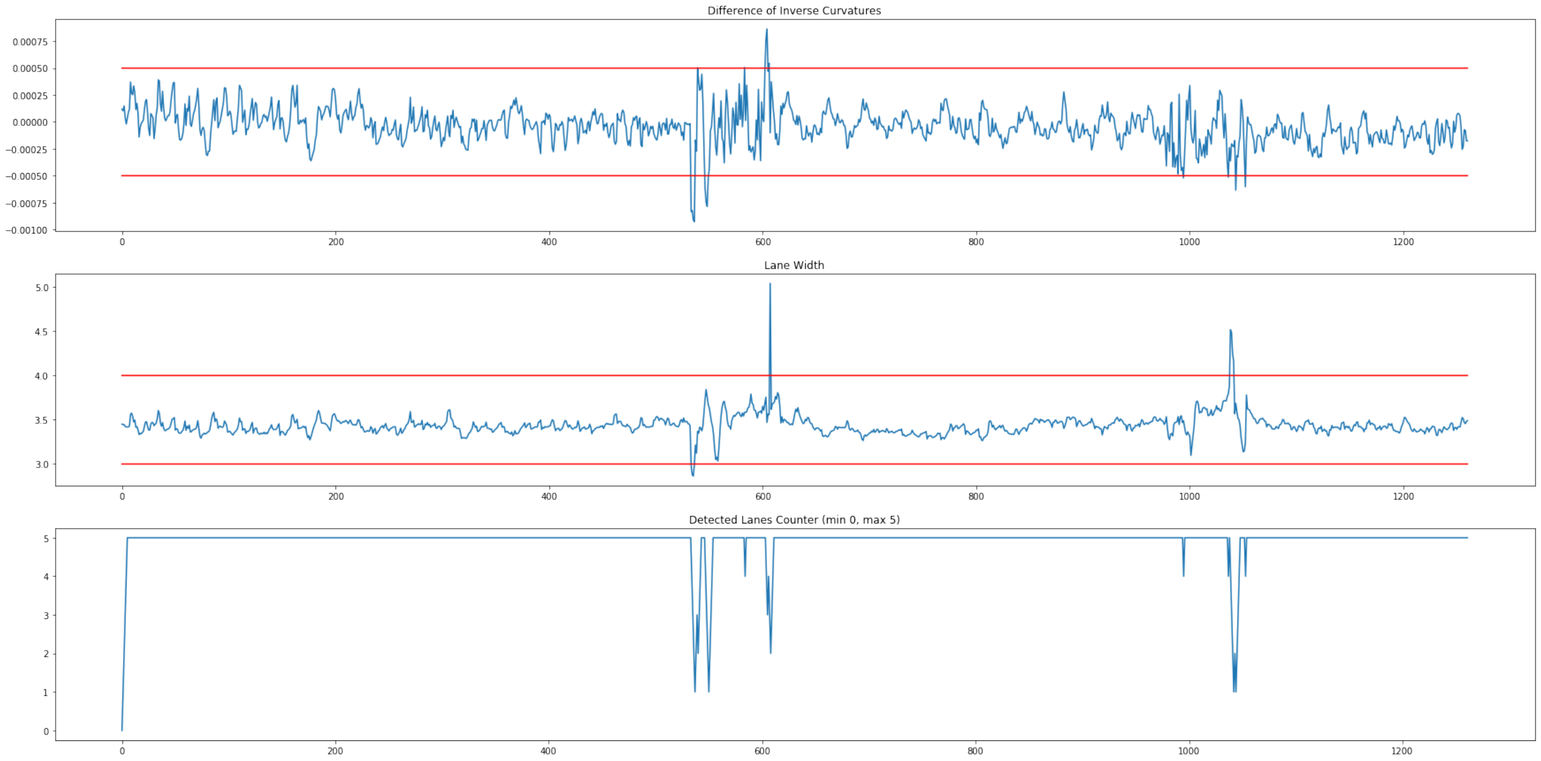
What I have inlcuded below are the final graphs. The red lines indicate the thresholds and you can see where they were exceeded.

In [32]:

```
# f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,20))
# ax1.set_title('Project Video')
# ax1.imshow(histogram_array_1, interpolation='none', cmap='gnuplot')
# ax2.set_title('Challenge Video')
# ax2.imshow(histogram_array_2, interpolation='none', cmap='gnuplot')
# ax3.set_title('Harder Challenge Video')
# ax3.imshow(histogram_array_3, interpolation='none', cmap='gnuplot')


f, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(30,15))

# SHOW DIFFEENCE OF INVERSE CURVATURES
curve_diff = np.array(lanes.curvature_diff)
high_limit = np.ones_like(curve_diff)*max_curvature_diff
low_limit = np.copy(high_limit) * -1

ax1.set_title('Difference of Inverse Curvatures')
ax1.plot(curve_diff)
ax1.plot(high_limit, color='red')
ax1.plot(low_limit, color='red')

# limits = np.copy(curve_diff)
# limits = limits[abs(limits) > max_curvature_diff]
# print(len(limits))
# print(limits)

# SHOW LANE WIDTH
width = np.array(lanes.width)
high_limit = np.ones_like(width) * max_lane_width
low_limit = np.ones_like(width) * min_lane_width

ax2.set_title('Lane Width')
ax2.plot(width)
ax2.plot(high_limit, color='red')
ax2.plot(low_limit, color='red')

# SHOW THE DETECTED LANES COUNTER
detected_count = np.array(lanes.detected_count)

ax3.set_title('Detected Lanes Counter (min 0, max {0})'.format(max_undetected_fr
ames))
ax3.plot(detected_count)
```

Out[32]:

```
[<matplotlib.lines.Line2D at 0x1034bfb70>]
```



In [ ]: