Advanced Lane Finding Project

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.

Computing the camera calibration using chessboard images

The first step in this project is getting of distortion caused by camera by applying camera calibration function provided by OpenCV.

Camera distortion affects some visual characteristics of the objects in the image, it can make a straight lines look curved which will give wrong assumptions about lanes and objects.

To perform Camera Calibration using a chessboard pattern. First we have to prepare the points coordinates that present the coordinates of the chessboard's squares corners. Then we loop through each image, convert it to a grayscale, and apply findChessboardCorners function to find the chessboard corners. After that we use all the points that we found to calibrate the camera using calibrateCamera() function, the output of the calibration that we need is two matrices; the calibration matrix and distortion.

Check the python notebook to see how we Implement the steps.

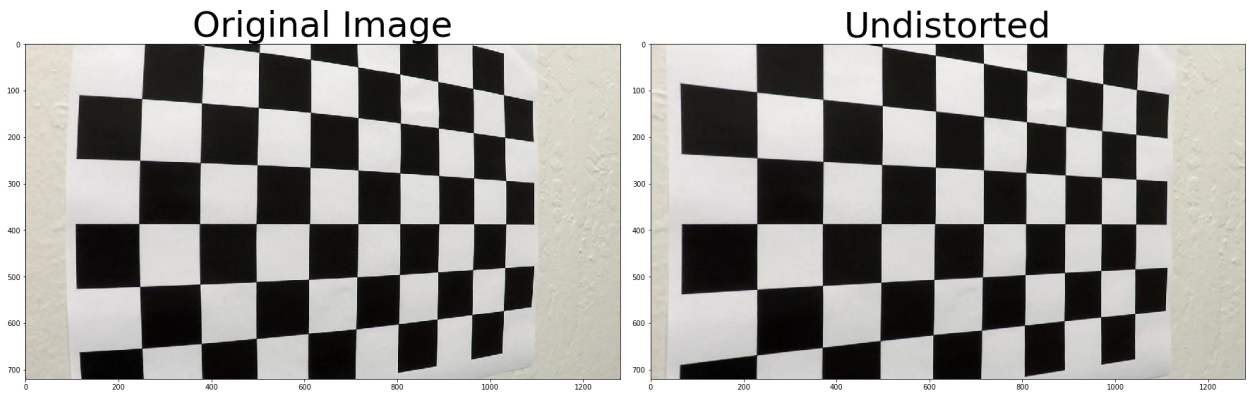
I found it useful to use Pickle library to save data in python. I used it to save the calibration matrix and distortion matrix for the camera, so I can just load it from the disk an proceed.

In order to test the calibration matrices I re-defined the un-distortion function in a shorter version that is easier to use. Also I made a function that plots a 2 by 1 images. The plotting function will be very useful in testing the pipeline steps. The following block shows the definition for each.

```
unDisImg (img,mtx,dist)
```

subPlot1by2(img1,img2, title = "Processed Image")

Using the previously defined function to test and show the result of camera un-distortion, I use one of the chessboard images and applied the un-distortion matrix on it, here is the result:



Lane Preprocessing Functions

The following show the functions used to extract and mask the left and right lanes pixels. The final function combinedThresholds() combines all the extracted masks in a single binary mask trying to get the best result.

The used filtering masks are:

- gradient absolute value
  - in x and y directions
- gradient magnitude
- gradient direction
- Color masks one HLS space:
  - for yellow lane
  - for white lane

For testing, all these binary masks are combined in different ways to strengthen the final binary mask. You can see use one of the following combination:

- **'gradients':**

```
combines all the gradients as:

    gradients[((gradx == 1) & (grady == 1) ) | ((mag_binary
== 1) & (dir_binary == 1))] = 1
```

- **'colours':**

```
combines the two color masks:
```

```
colour[((lMask == 1) | (sMask == 1))] = 1
```

- **'all-OR':**

  Logical 'OR' combination between the gradients mask and the colour mask

  ```
  combined[(gradients == 1) | (colour == 1)] = 1
  ```

- **'all-AND':**

  Logical 'OR' combination between the gradients mask and the colour mask

  ```
  combined[(gradients == 1) & (colour == 1)] = 1
  ```

- **'all-A':**

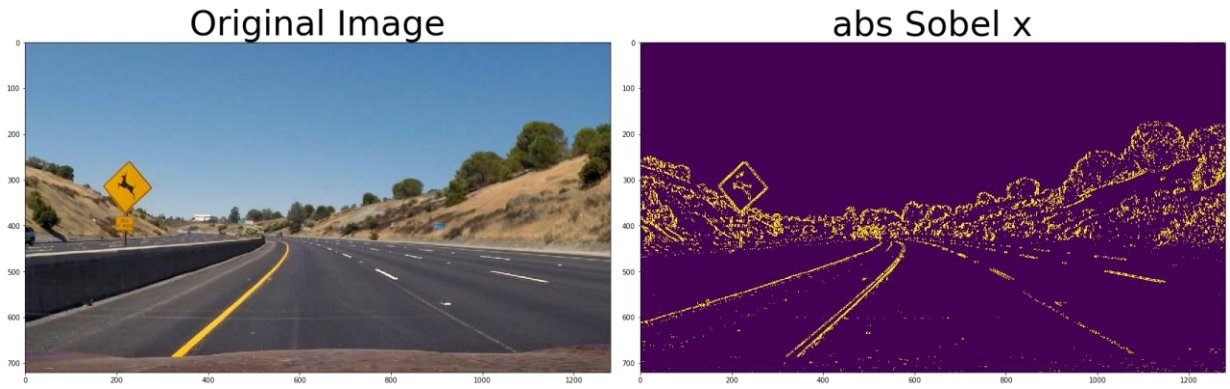
  Just another combination trial…

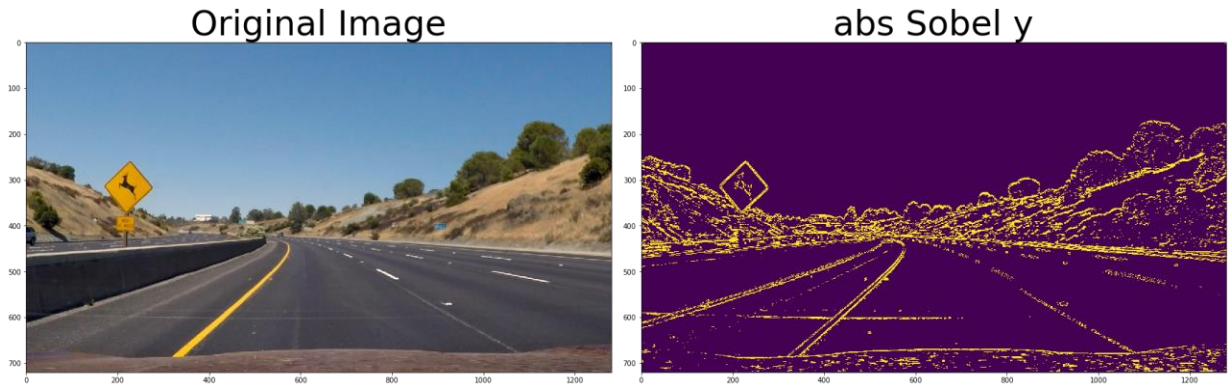
  ```
  combined[ ((lMask == 1) | (sMask == 1)) | ((gradx == 1) & (
  mag_binary == 1) & (dir_binary == 1) )] = 1
  ```

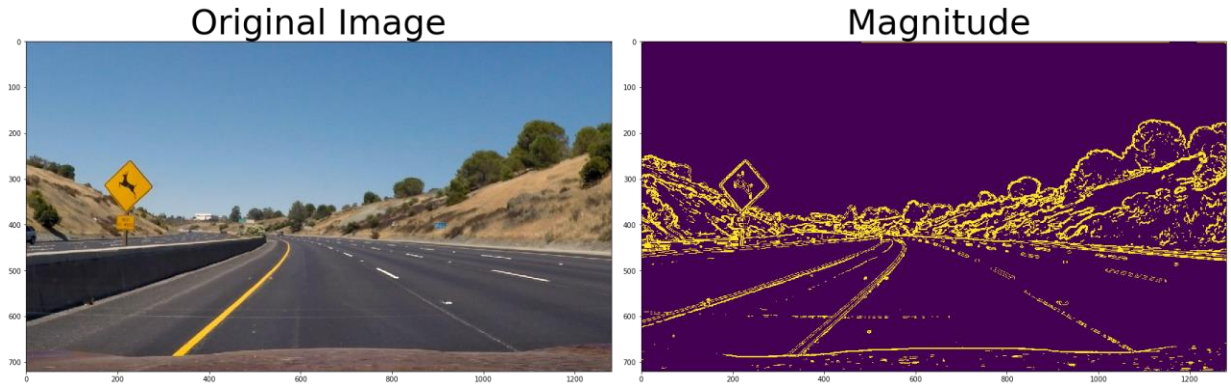
Check the Ipython notebook for the Implementation details.

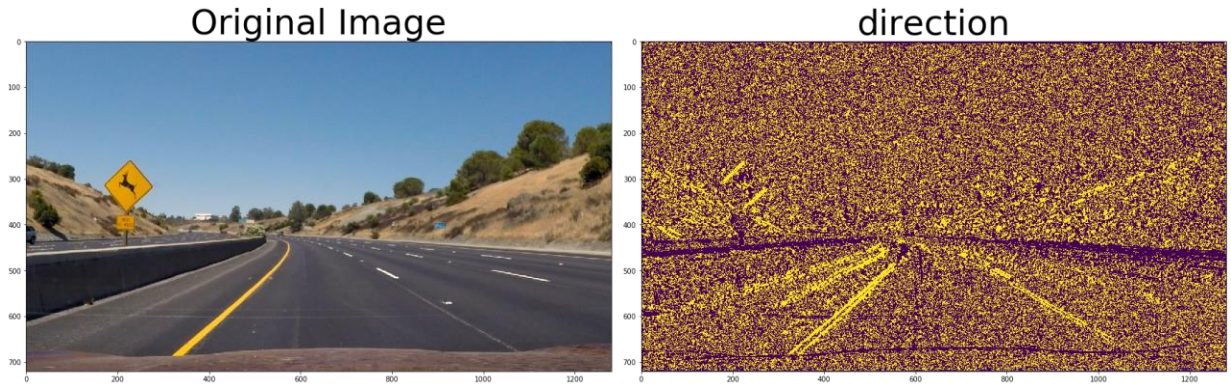
The Following images show the result of applying each binary mask on a test image, then the result of combining them all to get the finally binary mask.

Gradients:

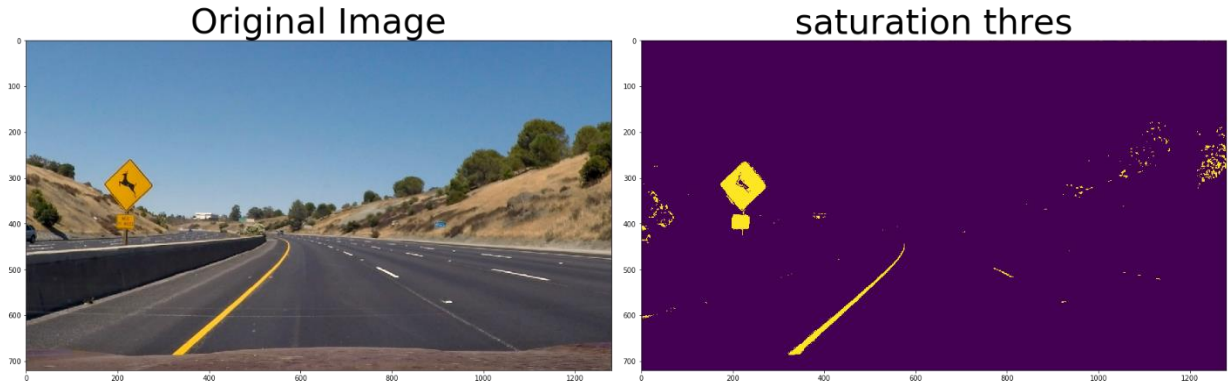




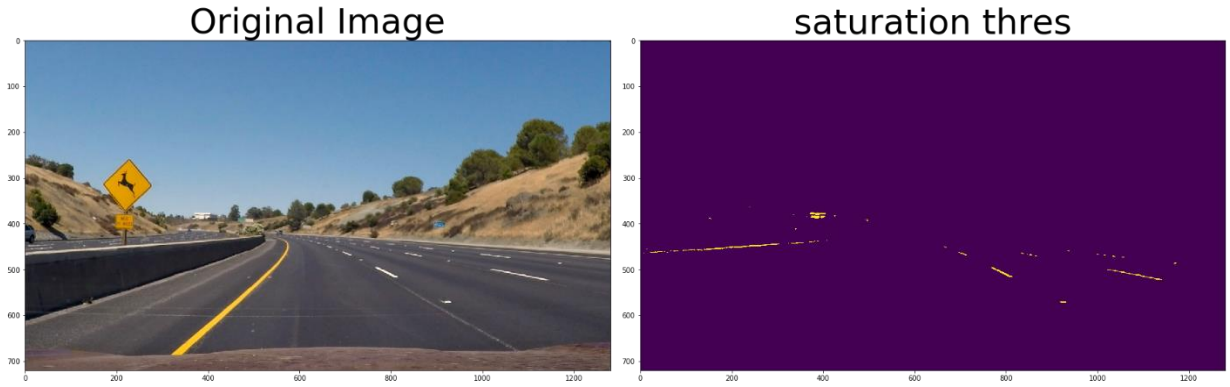




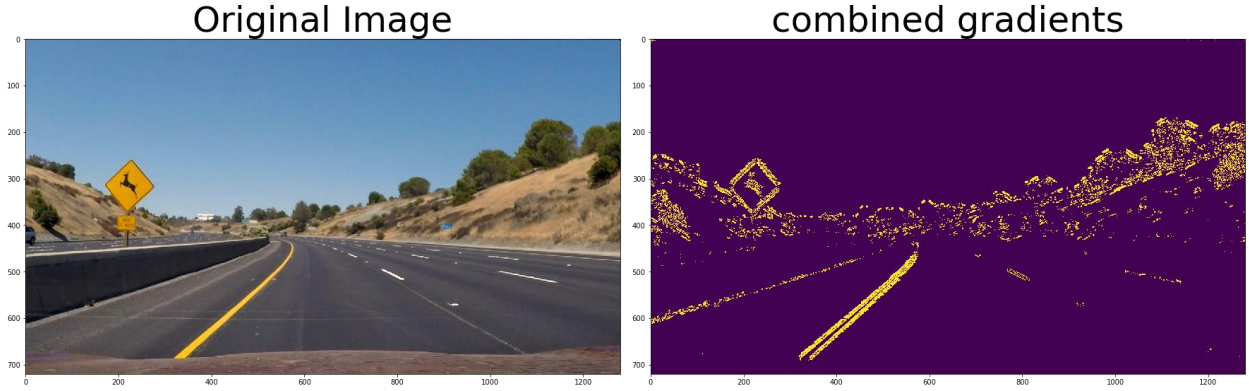
Color binary mask for the Yellow line:

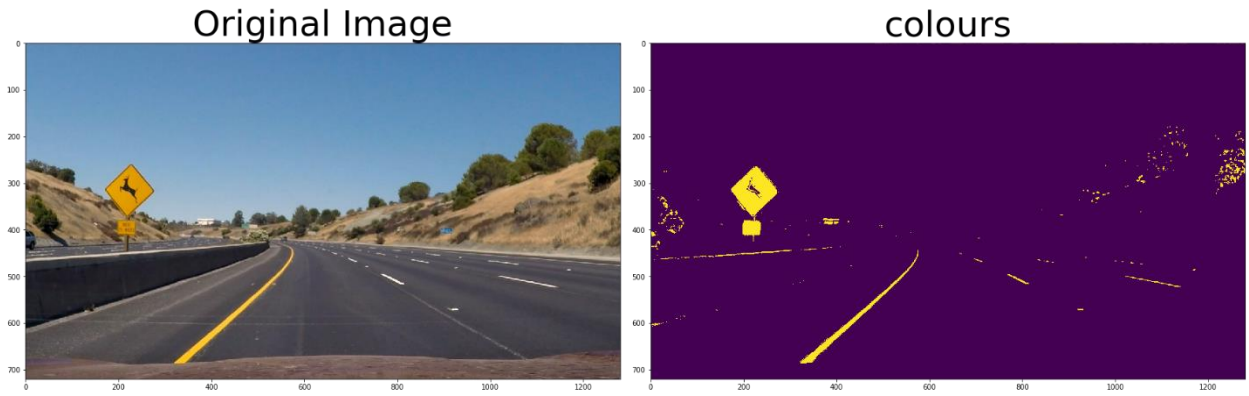


Color binary mask for the white line:

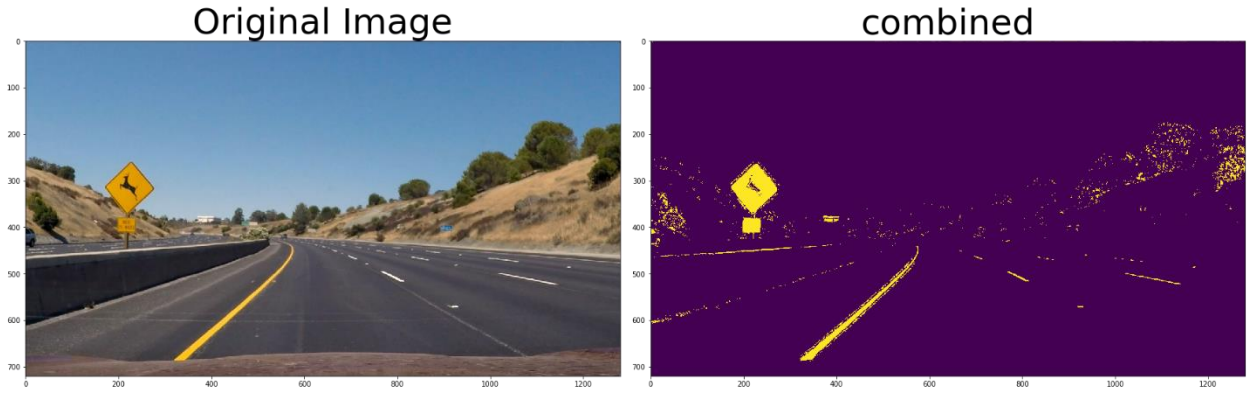


Testing the final combination with the different options introduced before:





Logical ORed gradient and colors



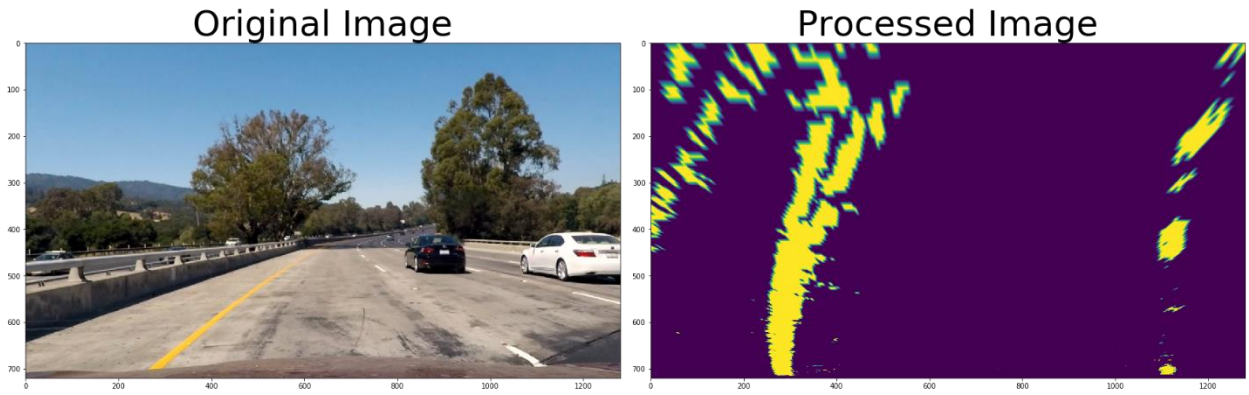
Perspective transform

Here I'll find the Perspective transform Matrix between the road region and the top view of the road using perspective_transform(), I also defined the warpedImage() function that perform the transform

The output of the perspective_transform() function is two matrices to perform the transform in two ways original view to top view and top view back to the original view, which we will be using when overlaying the road area back on the organa image later.

An example of applying the transform:

Applying the top view on binary masked image will show the lane pixels as follows:



Lane line fitting:

The adopted approach to define road lanes here is by finding the pecks in a histogram. The histogram is analyzed over sliding sections of the image called windows. The peak of the histogram in each window represent the lane. The histogram peak is located where more pixels lanes in the binary mask are located.

The lane is first processed by creating sliding windows as in find_lane_pixels() function. Starting from the bottom of the image, the lanes starting points are defined, moving to the higher windows each window center is redefined based on the histogram peak from the previous one. The centers of the windows are then used to fit a second order polynomial describing the lane line. this process is done for the left and right lane lines.

To make the process faster in videos, the previous process is done only for the first frame, after that we can build a search window based on the lane line found in the previous frame. search_around_poly() function implements that and keeps updating the lane line in the process while processing a video

Lane Info & Lane Overlay

The following functions provides a visualization for the extracted lane info. measure_curvature_real() measures the lane curvature, to be honest I am not sure about the result as we don't have a reference for the true calculations. Note that I adjusted the xm_per_pix value to 3.7/840, when I measured the lane width in pixels manually it was almost 840 pixels.

My laneinfo() function computes the lane width by evaluating the lines polynomial at the same 'y' position, it also finds the vehicle position in reference to the lane center.

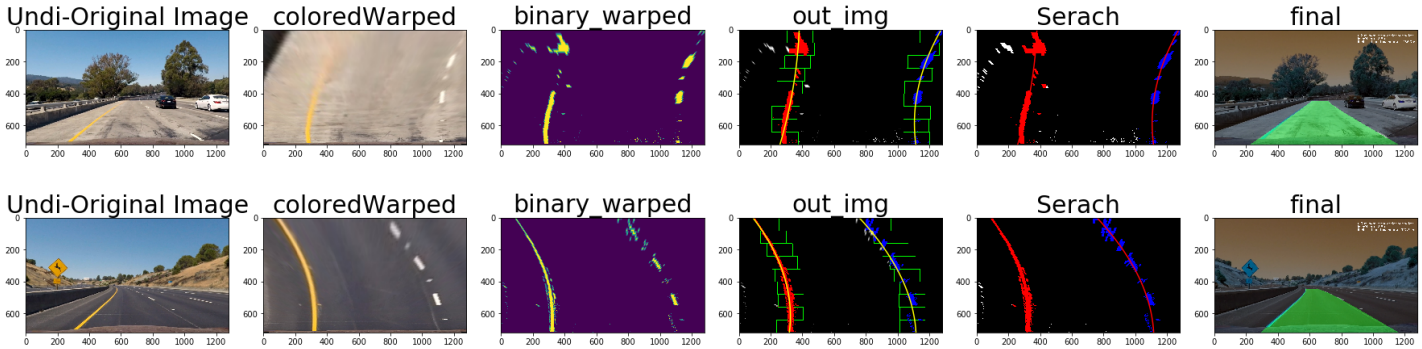
The overlay() function, visualized the lane are in green color, while placing the curvature info in the top right corner.

Pipeline Single Image

The pipeline foe a single image will include the following steps:

1. Reading the image
2. Undistort the image
3. Extract the binary mask for the lane pixels
4. Build the search area either by using the sliding windows or by searching around the previously known lane line.
5. Fit the detected pixels (or the centers of the windows) to a second order polynomial
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Overlay the road area back on the original image, and show the lane curvature and vehicle position info on the frame.

The following examples shows all the phases:



A better view is available when you double click the example plot in Ipython notebook.

Writing the pipeline() function in a way that works for the first frame and other frames, where it can take the previous lane lines polynomials as input. this will help me writing the videoProcess() function.

```
def pipeline(image,savedMtx,savedDist, M, unwrapMtx, left_fit = None,
right_fit = None , firstFrame = True):
```

```
        undismage = unDisImg(image,savedMtx,savedDist)
        masked = combinedThresholds(undismage, HLS = 1, Mode = 'all-OR')
        #coloredWarped, _ = perspective_transform(image)
        binary_warped = warpedImage(masked,M)
        if firstFrame:
            out_img, left_fitx,right_fitx, ploty, left_fit ,right_fit=
    fit_polynomial(binary_warped)
        else:
            _ ,left_fitx, right_fitx, ploty,left_fit,right_fit =
    search_around_poly(binary_warped,left_fit,right_fit)

        radius = measure_curvature_real(ploty,left_fitx,right_fitx)
        final = overlay(image, left_fitx,right_fitx, radius, unwrap_m=unwrapMtx )
    return final , right_fit, left_fit
```

Testing the pipeline function in single frame mode (firstFrame = True) where it does not depend on a previous info from other frames.

Copulating every thing in a single function to process the video, I am using OpenCV to read and write the video frames, Note that how I am using the pipeline() function for the first frame and the following frames .

First frame:

```
    outImg , right_fit, left_fit = pipeline(frame,savedMtx,savedDist, M,
    unwrapMtx, left_fit = None, right_fit = None , firstFrame = True)
```

Other Frames:

```
    outImg , right_fit, left_fit = pipeline(frame,savedMtx,savedDist, M,
    unwrapMtx, left_fit = left_fit, right_fit = right_fit , firstFrame = False)
```

I am using the pipeline() in firstFrame mode every 15 frames to update the searching windows in case we are shifting away.

Discussion

The suggested pipeline from the lessons worked very well, but for the challenge videos the pipeline fails for shadows, and the road edges. A more robust pipeline is required here. Better binary masks might solve the issue I tried some combination listed above but they failed also. Maybe a Deep learning technique is required here.
One more note here, the pipeline is also slow, I think the only way to make it faster is resizing the input frame, since all the other functions used can't be removed or reduced for faster performance.

Overall I learned a lot doing this ProJet, I learned about the Pickle library also I learned how to pass a default parameter to a function which was so powerful.