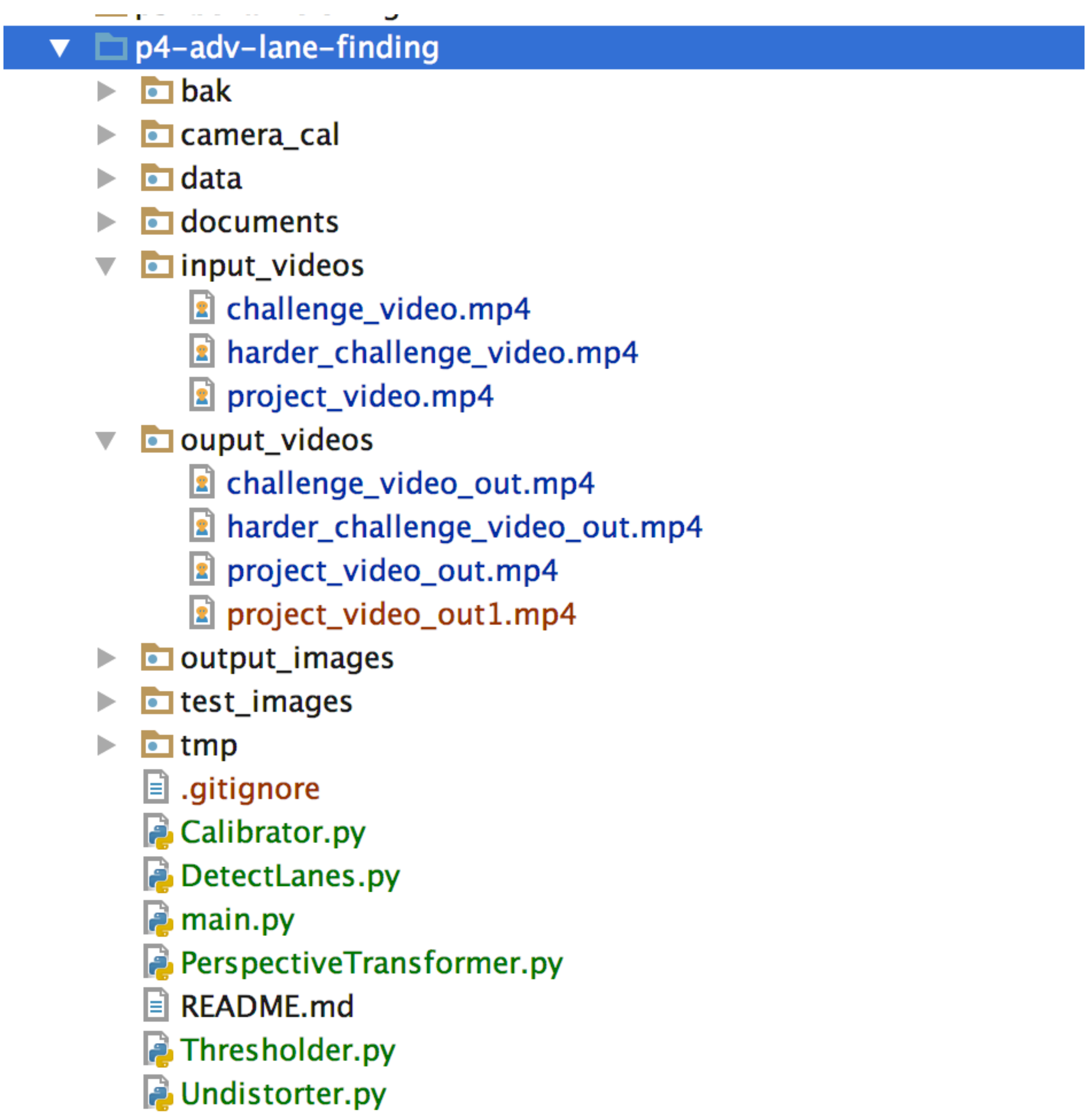
Advanced Lane Finding Project

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.

Source Code Directory Structure



```
▼ p4-adv-lane-finding
    ▶ bak
    ▶ camera_cal
    ▶ data
    ▶ documents
    ▼ input_videos
          challenge_video.mp4
          harder_challenge_video.mp4
          project_video.mp4
    ▼ ouput_videos
          challenge_video_out.mp4
          harder_challenge_video_out.mp4
          project_video_out.mp4
          project_video_out1.mp4
    ▶ output_images
    ▶ test_images
    ▶ tmp
      .gitignore
      Calibrator.py
      DetectLanes.py
      main.py
      PerspectiveTransformer.py
      README.md
      Thresholder.py
      Undistorter.py
```

main.py is the main program. If you run the main program, it will read the videos from ./input_videos directory and create the outout video in output_videos. You can edit this program to run which input video you would like to process using pipeline

Camera Calibration

For extracting lane lines that bend it is crucial to work with images that are distortion corrected. In that process, the first step is to calibrate the key attributes in Calibrate class.

- Caliberator.py class implements the calibration using the images from ./camera_cal directory

- Since we do not need to calibrate every time, I have implemented option to save the results.

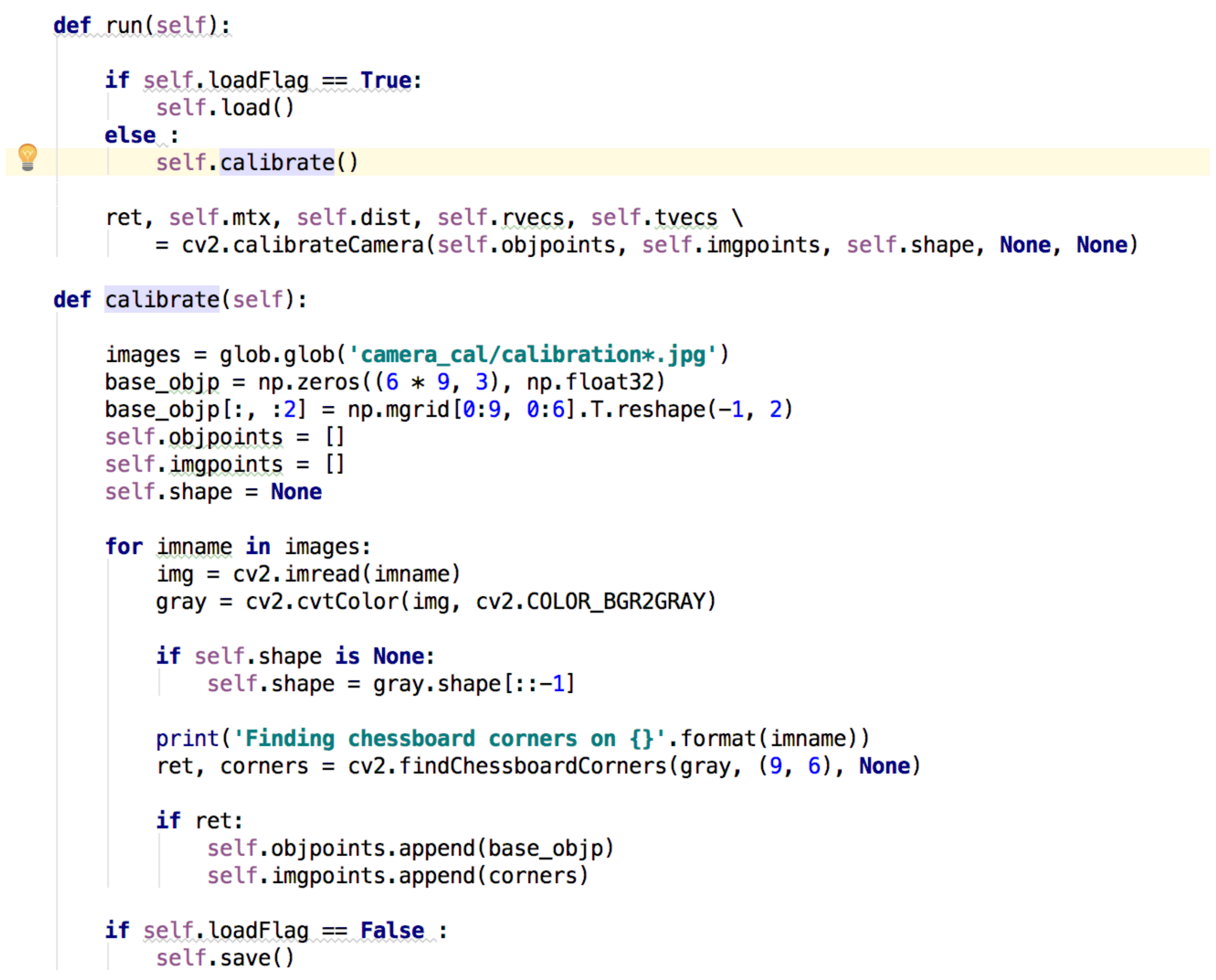
- Non image-degrading distortions can easily be corrected using test targets. Samples of chessboard patterns recorded with the same camera that was also used for recording the video are provided in the `camera_cal` folder.

- We start by preparing "object points", which are (x, y, z) coordinates of the chessboard corners in the world (assuming coordinates such that z=0).

- Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.`objpoints` and `imgpoints` are then used to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I used the provided 20 camera calibration images. These images are in ./camera_cal subdirectory

  If the Calibrator class is invoked with loadFlag= True, then the results are loaded from previously saved calibration information.

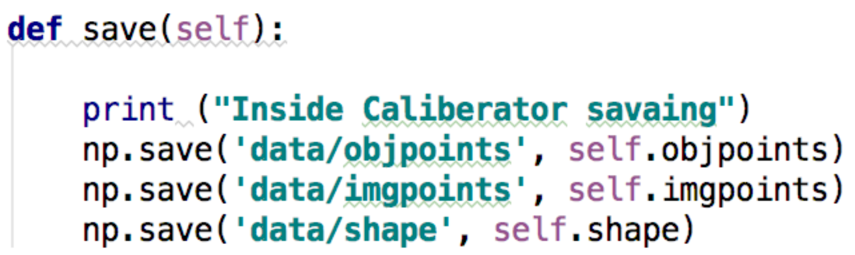
```
loadFlag = True
calibrator = Calibrator(loadFlag)
```



```python
def run(self):

    if self.loadFlag == True:
        self.load()
    else :
        self.calibrate()

    ret, self.mtx, self.dist, self.rvecs, self.tvecs \
        = cv2.calibrateCamera(self.objpoints, self.imgpoints, self.shape, None, None)

def calibrate(self):

    images = glob.glob('camera_cal/calibration*.jpg')
    base_objp = np.zeros((6 * 9, 3), np.float32)
    base_objp[:, :2] = np.mgrid[0:9, 0:6].T.reshape(-1, 2)
    self.objpoints = []
    self.imgpoints = []
    self.shape = None

    for imname in images:
        img = cv2.imread(imname)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        if self.shape is None:
            self.shape = gray.shape[::-1]

        print('Finding chessboard corners on {}'.format(imname))
        ret, corners = cv2.findChessboardCorners(gray, (9, 6), None)

        if ret:
            self.objpoints.append(base_objp)
            self.imgpoints.append(corners)

    if self.loadFlag == False :
        self.save()
```

- The calibrate method above iterate on each image, finding the chessboard corners and corresponding coordinates on the board using `cv2.findChessboardCorners()`

- The results are stored under ./data directory as follows.



```python
def save(self):

    print ("Inside Caliberator savaing")
    np.save('data/objpoints', self.objpoints)
    np.save('data/imgpoints', self.imgpoints)
    np.save('data/shape', self.shape)
```

Image Undistortion

I applied this distortion correction to the test image using the `cv2.undistort()` function. This is implemented in Undistorter.py class

Image Filtering using Threshold

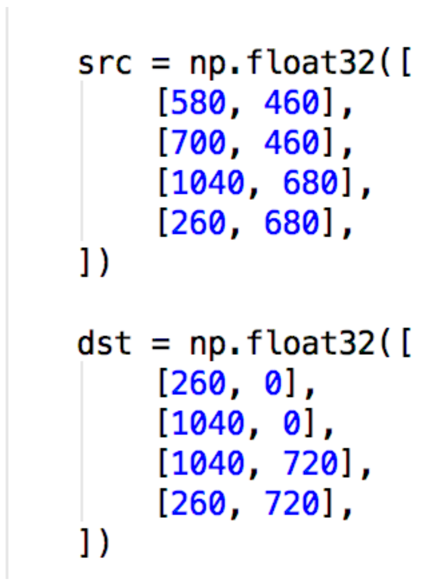
I have used a combination of direction, color and gradient thresholds to generate a binary image. Thresholder.py class implements following methods

```
- applyDirectionThreshold()

- After trying different values, minimum direction 0.7 and max direction 1.2
  seems identifying the lanes properly.

  self.thresh_dir_min = 0.7
  self.thresh_dir_max = 1.2


- applyMangnitudeThreshold()

  self.thresh_mag_min = 50
  self.thresh_mag_max = 255


- applyColorThreshold()
```

From tutorial, we can see that color space like HLS is robust. S channel is probably best bet, given that it's cleaner than the H channel is performing a bit better than the R channel or simple gray scaling. The S channel is still doing a fairly robust job of picking up the lines under very different color and contrast conditions, while the other selections look messy.

Image Transformation & Warping

- I have implemented the transformation to bird's eye view in PerspectiveTransformer.py class. The method returns warped perspective image.
- Following are the source and destination coordinates



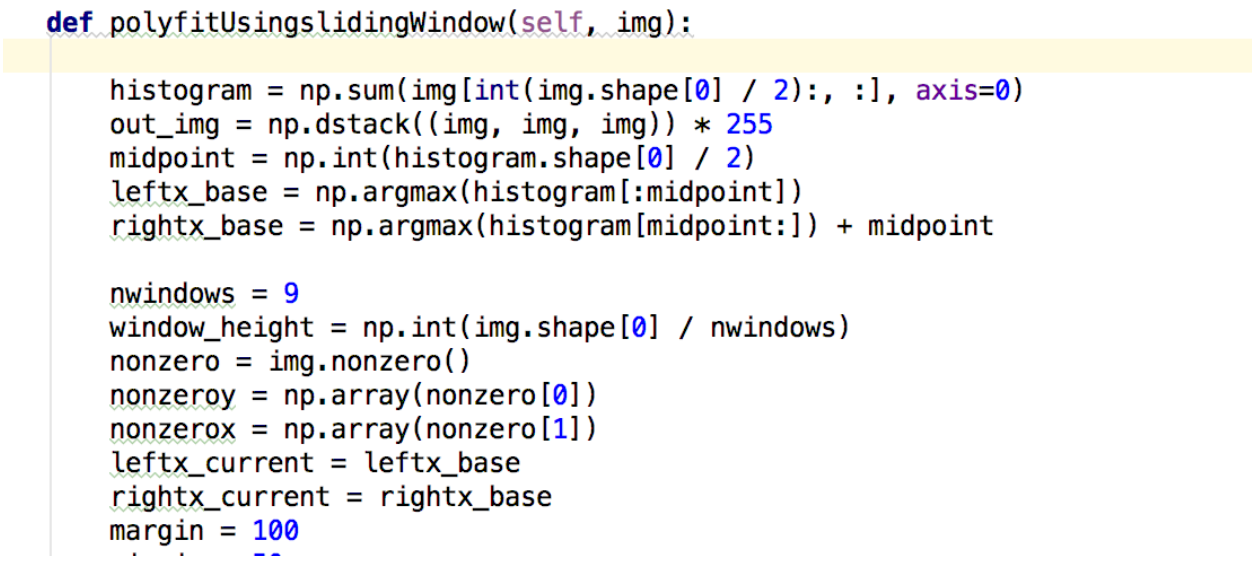
```
src = np.float32([
    [580, 460],
    [700, 460],
    [1040, 680],
    [260, 680],
])

dst = np.float32([
    [260, 0],
    [1040, 0],
    [1040, 720],
    [260, 720],
])
```

Lane detection

I have implemented lane detection using sliding window in DetectLanes.py class. The high level logic as follows

- The function polyfitUsingslidingWindow() takes the bottom half of a binarized and warped lane image to compute a histogram of detected pixel values.
- The result is smoothened using a gaussia filter and peaks are subsequently detected using.
- The function returns the x values of the peaks larger than thresh as well as the smoothened curve
- The sliding window approach which takes an binary (3 channel) image img and computes the average x value center of all detected pixels in a window centered at center_point of width width. It slices a binary image horizontally in 6 zones and applies next window to each of the zones. The center_point of each zone is chosen to be the center value of the previous zone.Thereby subsequent windows follow the lane line pixels if the road bends. The function returns a masked image of a single lane line seeded at center_point.
- Given a binary image left_binary of a lane line candidate all properties of the line are determined within an instance of a Line class.

- Sanity checks are performed and successful detections are pushed into a FIFO que of max length n. Each time a new line is detected all metrics are updated. If no line is detected the oldest result is dropped until the queue is empty and peaks need to be searched for from scratch.

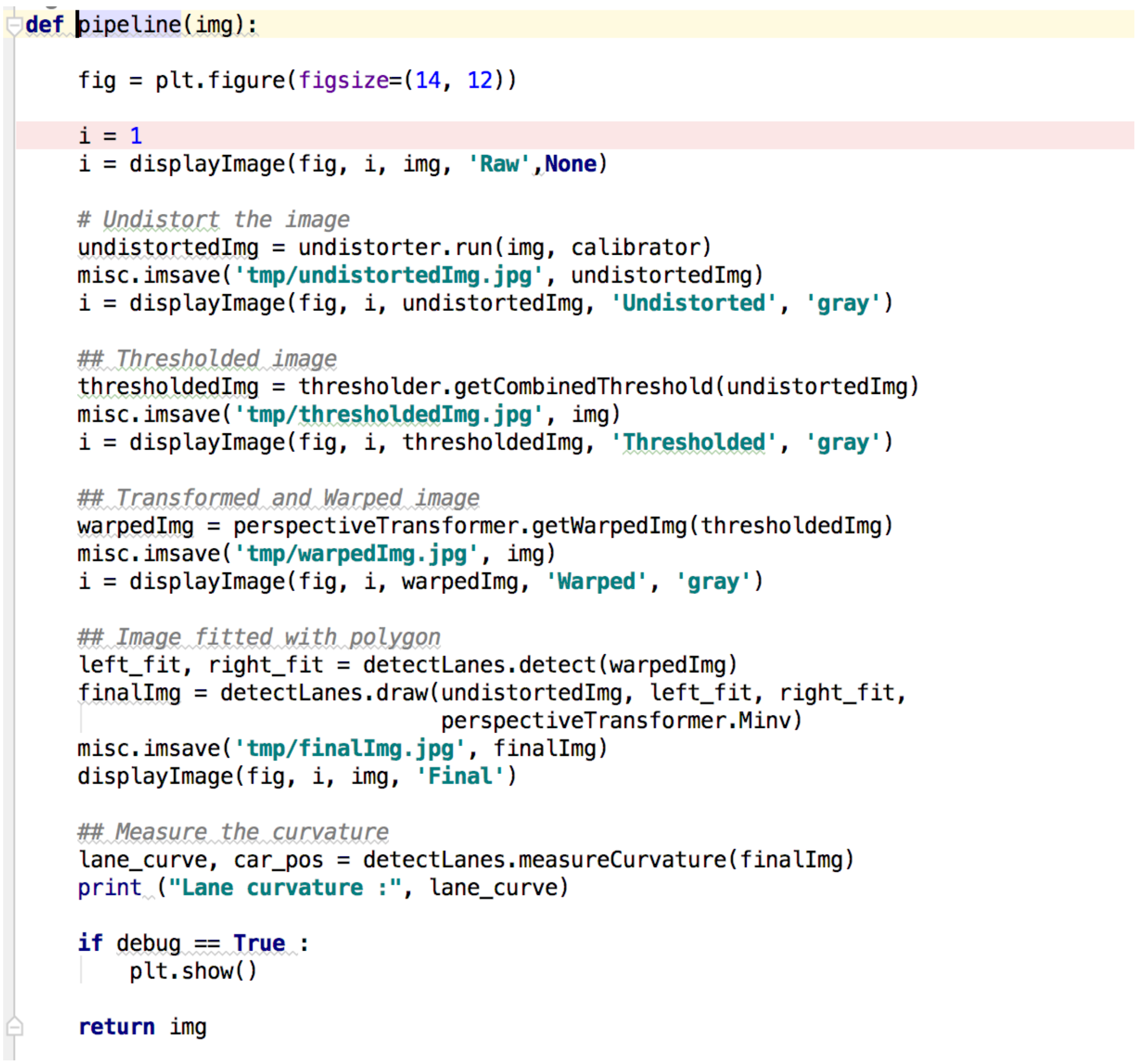


```
def polyfitUsingslidingWindow(self, img):

    histogram = np.sum(img[int(img.shape[0] / 2):, :], axis=0)
    out_img = np.dstack((img, img, img)) * 255
    midpoint = np.int(histogram.shape[0] / 2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    nwindows = 9
    window_height = np.int(img.shape[0] / nwindows)
    nonzero = img.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    leftx_current = leftx_base
    rightx_current = rightx_base
    margin = 100
```

Measuring lane curvature

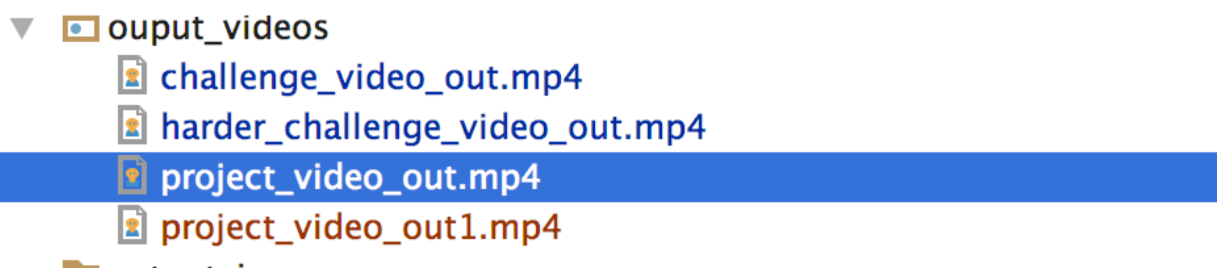
I then created a polygon using the curves of each computed polyline and warped back the result using the reversed source and destination. I finally drew this polygon on the undistorted image.

Pipeline (single images)



```
def pipeline(img):

    fig = plt.figure(figsize=(14, 12))

    i = 1
    i = displayImage(fig, i, img, 'Raw',None)

    # Undistort the image
    undistortedImg = undistorter.run(img, calibrator)
    misc.imsave('tmp/undistortedImg.jpg', undistortedImg)
    i = displayImage(fig, i, undistortedImg, 'Undistorted', 'gray')

    ## Thresholded image
    thresholdedImg = thresholder.getCombinedThreshold(undistortedImg)
    misc.imsave('tmp/thresholdedImg.jpg', img)
    i = displayImage(fig, i, thresholdedImg, 'Thresholded', 'gray')

    ## Transformed and Warped image
    warpedImg = perspectiveTransformer.getWarpedImg(thresholdedImg)
    misc.imsave('tmp/warpedImg.jpg', img)
    i = displayImage(fig, i, warpedImg, 'Warped', 'gray')

    ## Image fitted with polygon
    left_fit, right_fit = detectLanes.detect(warpedImg)
    finalImg = detectLanes.draw(undistortedImg, left_fit, right_fit,
                                perspectiveTransformer.Minv)
    misc.imsave('tmp/finalImg.jpg', finalImg)
    displayImage(fig, i, img, 'Final')

    ## Measure the curvature
    lane_curve, car_pos = detectLanes.measureCurvature(finalImg)
    print ("Lane curvature :", lane_curve)

    if debug == True :
        plt.show()

    return img
```

Output Results



```
▼ ouput_videos
    challenge_video_out.mp4
    harder_challenge_video_out.mp4
    project_video_out.mp4
    project_video_out1.mp4
```

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?

- The implementation overall went pretty smoothly.
- It might fail in extreme lighting conditions or in countries with lanes line colors different than white and yellow, or if they are not well visually defined (worn out or missing).
- I could make it more robust by handling more lightning conditions and lane line colors, and also by adding recovery options in case I don’t detect any lane line, or if they differ too much from the previously detected lines.