Self-driving cars program - project 2: Advanced Lane Detection

Francesco Boi

Contents

1 Content of the project

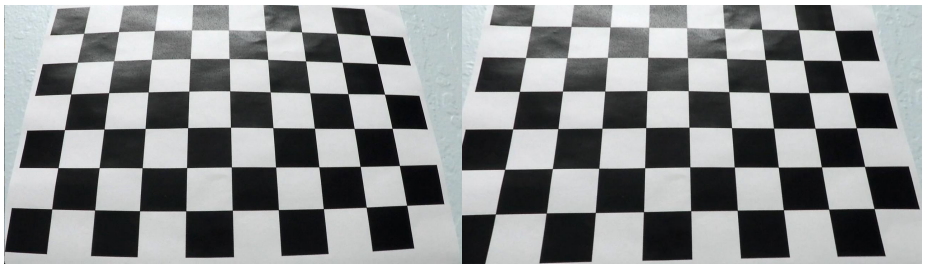
Here is the content of the project:

- *writeup.pdf* (this file): report of the project;
- *camera_cal*: folder containing the images to be used for camera calibration;
- *P2.ipynb*: ipython notebook with the code for advanced lane detection;
- *test_images*: folder containing the images to be used as tests;
- *output_images*: folder containing a subfolder for each image in *test_images*; each subfolder contains the resulting images for every step of the pipeline;
- *project_video.mp4*: input video of the project
- *challenge_video.mp4*: additional video;
- *harder_challenge_video.mp4*: additional video;
- *output*: folder containing the annotated videos resulting from the pipeline and matrices to undistort, warp and unwarp the images;

2 Project goals

The 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.



(a) Original image. (b) Undistorted image.

Figure 1: Result of the distortion correction on calibration image `calibration3.jpg`.

3 Camera calibration

Camera calibration is performed from cell 7 to 10 of the jupyter notebook, in the section *Camera calibration*. The assumption is that the chessboard lies on the plane with $z = 0$, hence the object points are the same for each calibration image.

Cell 8 (executed cell 6) prepares `objp`, a list containing (x, y, z) coordinates of the chessboard corners in the world for a single image. Since `objp` is the same for every calibration image, a copy of it will be appended every time chessboard corners are successfully detected 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.

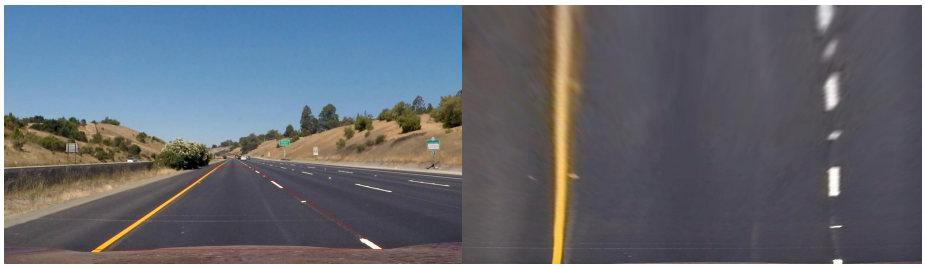
In cell 10 (executed cell 7) `objpoints` and `imgpoints` are used to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. Relevant matrices are saved in the pickle file `output/wide_dist_pickle.p` as dictionary.

Once the matrix has been calculated, the image can be corrected with the function `cv2.undistort(image, mtx, dist, None, mtx)`. Results of the distortion correction are shown in the folder `output_images/ calibration_result`; an example is shown in Figure 1.

4 Perspective transform

Perspective transform is performed from cell 14 to cell 16 (section *Perspective transform*).

In cell 15 (executed cell 10) the matrices transformation are calculate using the image `test_images/straight_lines1.jpg`. On this image, a trapezoid has been defined containing the straight lines of the image, whose points are defined in `src` array (for visualisation purposes, it is drawn over the original image). From an eye-bird point of view, the trapezoid corresponds to a rectangle whose width is equal to the bottom lane width in the image and height equal to the image



(a) Original image. (b) Eye-bird image.

Figure 2: Result of the distortion correction and perspective transformation on test image `straight_lines1.jpg`.

height itself: these new coordinates are defined in `dst` array. The following source and destination points have been defined:

Points used for perspective transform	
Source points	Destination points
(250, 680)	(250, img1.shape[0])
(592, 450)	(250, 0)
(690, 450(	(1060, 0)
(1060, 680)	(1060, img1.shape[0])

The transformation and inverse transformation matrices are calculated with:

```
M = cv2.getPerspectiveTransform(src, dst)
Minv = cv2.getPerspectiveTransform(dst,src)
```

and are saved in `output/persp_pickle.p` pickle file as a dictionary to be retrieved later.

The image is transformed with the function:

```
res_img = cv2.warpPerspective(img, M, (img.shape[1], img.shape
    [0]), flags=cv2.INTER_LINEAR)
```

Results of applying distortion correction and perspective transform on test images are contained in the folder `output_images/perspective_transform` (see the example Figure 2).

5 Pipeline description

Now that all required matrices have been calculated, it is possible to define a pipeline consisting of the following steps:

1. undistort the image;
2. perform magnitude thresholding to obtain a binary image;
3. perform colour thresholding to obtain a binary image;

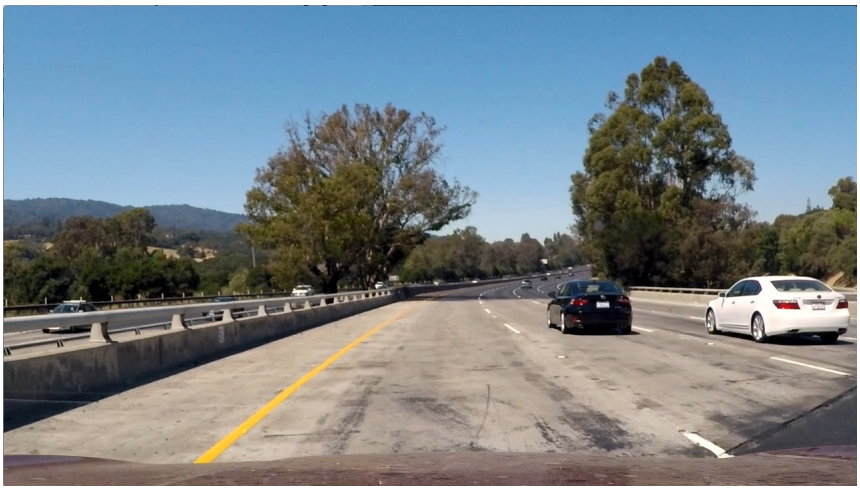


Figure 3: Original image `test1.jpg` used to demonstrate the result of the pipeline.

4. combine the magnitude thresholding and colour thresholding binary images;
5. apply region of interest mask;
6. apply perspective transform;
7. find pixel lanes;
8. fit two second order degree lines on pixel lanes;
9. calculate vehicle offset;
10. calculate line curvature;
11. draw the fitted lines and apply the inverse perspective transform.

All the intermediate steps of the pipeline applied to the test images have been saved in `output_images`, with each subfolder for each test image. This has been done in the loop contained in cell 28 (executed cell 17), section *Analyse test images*. For demonstration purposes `test_images/test1.jpg` is considered (Figure 3).

5.1 Undistort the image

As already told, this step is applied by calling `cv2.undistort(image, mtx, dist, None, mtx);`. An example of this processing applied to the chosen image is shown in Figure 4.

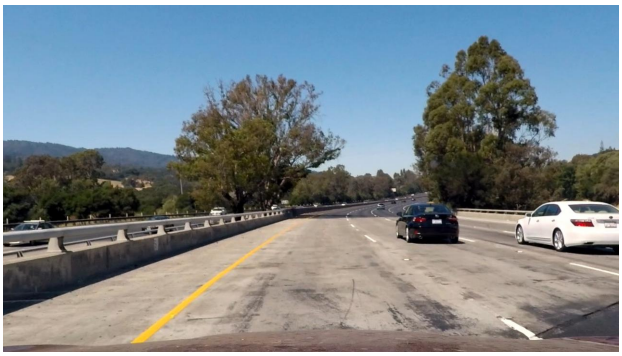


Figure 4: Undistorted image result.

5.2 Gradient thresholding

The functions to perform gradient thresholds are defined in cell 18 (executed cell 12), section *Magnitude functions*. The magnitude thresholding step has been defined with the function:

```
def combine_magnitudes(img, ksize=3, k_size_dir=15,
    sobel_thresh_x=(30, 100), sobel_thresh_y=(30, 100),
    mag_thresh=(30, 100), dir_thresh=(0.85,1.15)):
  gradx = abs_sobel_thresh(img, orient='x', kernel_size=ksize,
    thresh_min=sobel_thresh_x[0], thresh_max=sobel_thresh_x[1])
  grady = abs_sobel_thresh(img, orient='y', kernel_size=ksize,
    thresh_min=sobel_thresh_y[0], thresh_max=sobel_thresh_y[1])
  mag_binary = mag_threshold(img, sobel_kernel=ksize,
    mag_thresh=mag_thresh)
  dir_binary = dir_threshold(img, sobel_kernel=k_size_dir,
    thresh=dir_thresh)
  combined_mag_thresholds = np.zeros_like(dir_binary)
  combined_mag_thresholds[((gradx == 1) & (grady == 1)) |
    ((mag_binary == 1) & (dir_binary == 1))] = 1
  combined_mag_threshold_coloured = np.dstack(
    ( gradx, grady, dir_binary)) * 255
  return combined_mag_thresholds,
      combined_mag_threshold_coloured
```

`abs_sobel_thresh`, `mag_threshold` and `dir_threshold` are the functions defined in the lectures so for seek of brevity they will not be reported here. The function simply calculates the derivatives along x and y, the magnitude and the direction of the gradient, taking for each component the absolute values and applying a minimum and maximum threshold to each independently. The threshold values are not too different from the ones used in the lectures. As suggested in the lecture, the individual resulting binary images are combined as:

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

The final result is shown in Figure 5. To see the contribution of its components and for debugging purposes, the function also returns an RGB image where the

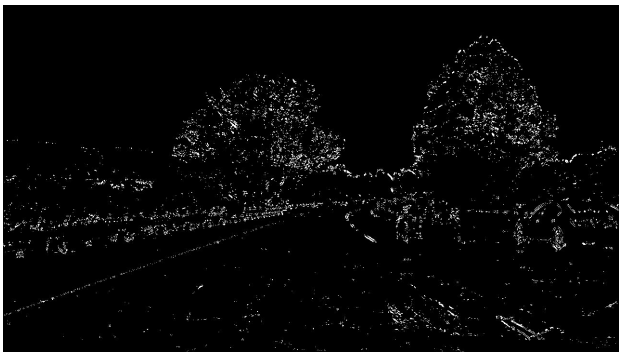


Figure 5: Example of binary image resulting from the magnitude threshold analysis.

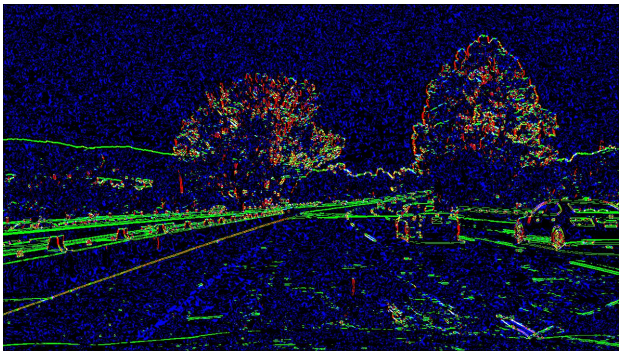


Figure 6: Example of RGB image returned by the function `combine_magnitudes` where the red channel is the derivative along x, the green channel the one along y and the blue channel is the direction threshold analysis.

red channel is the derivative along x, the green channel the one along y and the blue channel is the direction threshold analysis (Figure 6).

5.3 Colour threshold

Instead of applying colour thresholding on the RGB space, it is easier to apply them in the HLS or HSV space. In cell 23 (executed cell 14), section *Find colour threshold values*, the HLS and HSV channels are shown in two images in the RGB space. From Figure 7, it seems that white lane detection seems more straightforward in the HLS space, so we will perform the analysis in this space.

To perform the colour threshold the function `colour_threshold` has been defined in cell 18 (executed cell 12), section *Magnitude functions*:

```
def colour_threshold(img, s_thresh=(150, 255),
    l_thresh=(150, 255), h_thresh=(120,250), v_thresh=(180,250)):
  hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
```

(a) HLS channels displayed in the RGB space (b) HSV channels displayed in the RGB space

Figure 7: Comparison between HLS and HSV colour space.

```
4   h_channel = hls[:,:,0]
5   l_channel = hls[:,:,1]
6   s_channel = hls[:,:,2]
7   # Convert to HSV
8   hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
9   v_channel = hsv[:,:,2]
10
11  s_binary = np.zeros_like(s_channel)
12  s_binary[(s_channel >= s_thresh[0]) & (s_channel <= s_thresh
        [1])] = 1
13
14  l_binary = np.zeros_like(l_channel)
15  l_binary[(l_channel >= l_thresh[0]) & (l_channel <= l_thresh
        [1])] = 1
16
17  h_binary = np.zeros_like(h_channel)
18  h_binary[(h_channel >= h_thresh[0]) & (h_channel <= h_thresh
        [1])] = 1
19
20  v_binary = np.zeros_like(v_channel)
21  v_binary[(v_channel >= v_thresh[0]) & (v_channel <= v_thresh
        [1])] = 1
22  # Stack each channel to view their individual contributions in
        green and blue respectively
23  # This returns a stack of the two binary images, whose
        components you can see as different colors
24  colour_binary_stacked_hls = np.dstack(( h_binary, l_binary,
        s_binary)) * 255
25  colour_binary_stacked_hsv = np.dstack(( h_binary, s_binary,
        v_binary)) * 255
26
27  # Combine the two binary thresholds
28  combined_colour_binary = np.zeros_like(s_binary)
29  combined_colour_binary[((h_binary==1)&(s_binary==1)&(l_binary
        ==1))] = 1
30  return combined_colour_binary, colour_binary_stacked_hls,
        colour_binary_stacked_hsv
```

The function converts the RGB image into the HLS and HSV representations. Each channel is thresholded to obtain a binary image from each of them. Then

Figure 8: Binary image resulting from colour thrsholding

a single binary image is returned by performing pixelwise AND on the H,L and S channels. For visualisation and debugging purposes, the function returns also the previously seen HLS and HSV representations displayed in the RGB space, understand the contribution of each channel separately.

The function is called three times with different thresholds from the pipeline: one to get the yellow lane, one to get the white lane and a third time to get some other parts of the white lane that are difficult to find, the parts that are further ahead. Then the three resulting images are pixelwise ORed:

```
h_thresh_yellow=(0, 30)
l_thresh_yellow=(80, 190)
s_thresh_yellow=(150, 255)

h_thresh_white=(0, 170)
l_thresh_white=(100, 255)
s_thresh_white=(100, 255)

h_thresh_white2=(130, 170)
l_thresh_white2=(160, 240)
s_thresh_white2=(0, 150)
yellow_lane, _, _ = colour_threshold(undistort_img,
  h_thresh=h_thresh_yellow, l_thresh=l_thresh_yellow,
  s_thresh=s_thresh_yellow)
white_lane, _, _ = colour_threshold(undistort_img,
  h_thresh=h_thresh_white, l_thresh=l_thresh_white,
  s_thresh=s_thresh_white)
white_lane2, _, _ = colour_threshold(undistort_img,
  h_thresh=h_thresh_white2, l_thresh=l_thresh_white2,
  s_thresh=s_thresh_white2)
combined_colour_thr_binary = white_lane | white_lane2 |
  yellow_lane
```

The result is shown in Figure 8.

Figure 9: Binary image resulting from the combination of gradient thresholding step and colour thresholding step.

5.3.1 Finding the colour thresholds

To find the colour thresholds, cell 26 (executed cell 16) plots the HLS and HSV representations in the colour space using `\%matplotlib tk` backend, so that it is possible to hover with the mouse over the pixels to check their value and get an idea of the thresholds.

5.4 Combine the two thresholds

The binary image resulting from gradient thresholding and the one resulting from colour thresholding are pixelwise ORed to get a single image.

```
combined_binary = np.zeros_like(combined_mag_thresholds)
combined_binary[(combined_colour_thr_binary == 1) |
  (combined_mag_thresholds == 1)] = 1
```

Result is shown in Figure 9.

5.5 Region of interest selection

The region of interest has been explained in the previous project. The vertices used in this case are the following:

```
vertices = np.array([[[int(xsz*0.1), ysz], [int(xsz*0.95), ysz],
    [int(0.56*xsz), int(0.61*ysz)],
    [int(0.48*xsz), int(0.61*ysz)]]])
```

The result is shown in Figure 10.

5.6 Apply perspective transform

The perspective transform has already been explained in section 4. The result is shown in Figure 11

Figure 10: Binary image the region of interest mask.

Figure 11: Image resulting from the perspective transformation.

5.7 Find the lanes pixels

The lanes pixels are found by the function `find_lane_pixels` defined in the cell 20 (execution cell 13), section *Functions for lane detection*. The function takes as input the warped binary image, but it must be rescaled so the non-zero values are 255: it is sufficient to multiply the binary image by 255 for this. The function has been explained in the lecture. It takes a histogram of the bottom half of the image with the call `histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)`. The histogram is expected to have two peaks, one on left side and one on the right that are found with the piece of code:

```
midpoint = np.int16(histogram.shape[0]//2)
leftx_base = np.argmax(histogram[:midpoint])
rightx_base = np.argmax(histogram[midpoint:]) + midpoint
```

To find the lane pixels two series of windows one above the other have to be defined, one for each lane: the active pixels that will lie within these windows will constitute the lane pixels. We start by defining one window for each lane at the bottom of the image having the same x coordinate of the peak (its width and height are hyperparameters). The non-zero pixels that lie within this window are the first lane pixels. The next window is positioned above the other but it might be recentred: if the number of pixels within the previous window is above the threshold `min_pix`, their mean is used to recentre the next window; if not the old position is used. The lines of code in the function that do this are the following:

```
nwindows = 9
# Set the width of the windows +/- margin
margin = 100
# Set minimum number of pixels found to recenter window
minpix = 50

leftx_current = leftx_base # max of the left part histogram
rightx_current = rightx_base # max of the right histogram

# 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
  ### TO-DO: Find the four below boundaries of the window ###
  win_xleft_low  = leftx_current - margin
  win_xleft_high = leftx_current + margin
  win_xright_low = rightx_current - margin
  win_xright_high = rightx_current + margin

  # Identify the nonzero pixels in x and y within the window #

  good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy <
      win_y_high) &
```

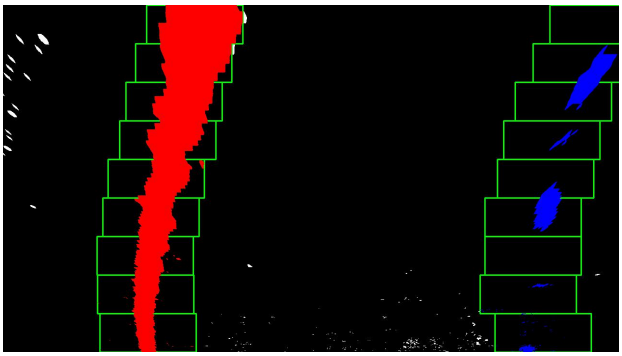


Figure 12: Image showing the windows to classify the left lane pixels in red, the right lane pixels in blue and the non-lane pixels in white.

```
28        (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_high)
              ).nonzero()[0]
29    good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy <
          win_y_high) &
30        (nonzerox >= win_xright_low) &  (nonzerox <
              win_xright_high)).nonzero()[0]
31    if len(good_left_inds) > minpix:
32      leftx_current = np.int16(np.mean(nonzerox[good_left_inds]))
33    if len(good_right_inds) > minpix:
34       rightx_current = np.int16(np.mean(nonzerox[good_right_inds
             ]))
```

The total number of windows is given by the height of the image divided by the height of the window. The visualisation of this step is shown in Figure 12

5.8 Fit the second degrees line

The function that fits the left and right lines is `fit_polynomial` defined in the cell 20 (execution cell 13), section *Functions for lane detection*. This function calls `find_lane_pixels`: whereas `find_lane_pixels` is explicitly called when analysing the test images to show the result of the process, in the video pipeline it will be called only from `fit_polynomial`. After finding the lane pixels by calling `find_lane_pixels(..)`, the function fits the second order polynomials by calling `np.polyfit(y, x, deg=2)` (y is the axis along the height of the image, x the one along its width) which returns the polynomial coefficients. Then the line points are calculated and draw on the image in cyan with the code:

```
1 ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.
      shape[0] )
2 try:
3   left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty +
        left_fit[2]
4   right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty +
        right_fit[2]
5 except:
```

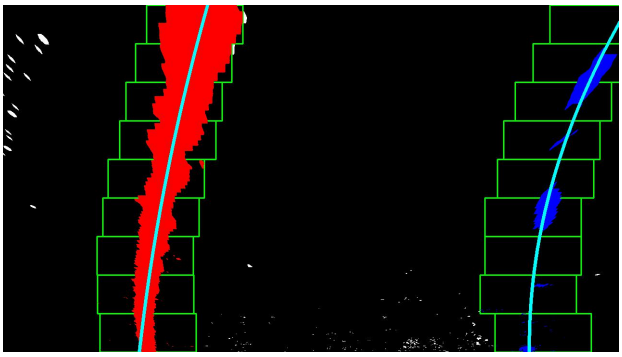


Figure 13: Image showing result of fitting two second degrees polynomial on each lane pixels.

```
...
cv2.polylines(out_img, [pts_left], False, color=(0,255,255),
    thickness=5)
cv2.polylines(out_img, [pts_right], False, color=(0,255,255),
    thickness=5)
```

The result is shown in Figure 13.

5.9 Vehicle offset

The vehicle offset is calculated with the function `vehicle_offset` defined in the cell 20 (execution cell 13), section *Functions for lane detection*

```
def vehicle_offset(left_fit, right_fit, shape):
  global xm_per_pix
  lane_center = ((left_fit[0] + right_fit[0])*shape[0]**2 +
    (left_fit[1] + right_fit[1])*shape[0] +
    left_fit[2] + right_fit[2])//2
  car_center = shape[1]/2  # we assume the camera is centered in
      the car
  center_offset = (lane_center - car_center) * xm_per_pix
  return center_offset
```

The function calculates the centre of the lane and the car centre, subtracts them and converts the result from pixels to metres. The car centre is half the width of the image (since the camera is supposed to be mounted exactly on the midpoint of the car). To calculate the lane centre, the function sums the values of the lines at the bottom of the image and divides it by two. In my case the lines have the abscissa starting at the top of the image, so their value is calculated with the input `shape[0]`, where `shape` is the image shape.

The result is printed on the resulting image itself together with radius of curvature of the two lines and other information (see Figure 14).

Figure 14: Final result of the pipeline.

5.10 Line curvature

The curvature of each line is calculated using the efficient method described in the lecture and implemented in the function `measure_curvature_efficient`, defined in the cell 20 (execution cell 13), section *Functions for lane detection*, that converts the line coefficients before applying the curvature equation. The coefficients are converted to get a value in metres by the following code:

```
left_fit_cp = left_fit.copy()
right_fit_cp = right_fit.copy()
left_fit_cp[0] = left_fit[0]* xm_per_pix/(ym_per_pix**2)
left_fit_cp[1] = left_fit[1]* xm_per_pix/(ym_per_pix)
right_fit_cp[0] = right_fit[0]* xm_per_pix/(ym_per_pix**2)
right_fit_cp[1] = right_fit[1]* xm_per_pix/(ym_per_pix)
```

Then the equation described in the lecture is applied. The result is printed on the resulting image itself together with vehicle offset and other information (see Figure 14).

5.11 Draw the fitted lines and apply the inverse perspective transform

For this step, first an empty image is created where the two lines are dawn, then the inverse perspective transform is applied to this image and the result is merged with the undistorted image. The code which does this starts from line 127 of the cell 28 responsible for analysing the test images (section *Analyse test images*):

```
rev_img = np.zeros_like(undistort_img)
cv2.polylines(rev_img, [points_left], False, color=(255,0,0),
    thickness=40)
cv2.polylines(rev_img, [points_right], False, color=(255,0,0),
    thickness=40)
unwarped_lanes = cv2.warpPerspective(rev_img, Minv, (rev_img.
    shape[1], rev_img.shape[0]),
  flags=cv2.INTER_LINEAR)
```

```
result = weighted_img(initial_img=undistort_img, img=
    unwarped_lanes)
```

`weighted_img` has also been used in the first project. The final result is shown in Figure 14.

6 Video pipeline

The pipeline steps have been grouped into a function to be called when analysing the video. Two versions have been implemented: a simple one consisting of the previously described steps and an advanced version which implements sanity check and that searches line pixels around the line fitted on the previous video frame so as to save computation by avoiding performing the hystogram analysis each time.

6.1 Simple version of the pipeline video

The pipeline video steps are implemented in the function `pipeline(image)` defined in cell 29 (executed cell 18). The steps are identical to the previous ones, except that `find_lane_pixels` is not explicitly called, since the function `fit_polynomial` is responsible for this call.

For debugging and visualisation purposes, some intermediate steps of the pipeline are shown on the top right of the video, so that one can get a larger picture of what is going on behind the curtains. The lines of code responsible for this are the following:

```
scale_percent = 20 # percent of original size
width = int(combined_binary.shape[1] * scale_percent / 100)
height = int(combined_binary.shape[0] * scale_percent / 100)
dim = (width, height)
# plot resized images from the analysis on top right for
    debugging purposes
resized = cv2.resize(res_img, dim, interpolation = cv2.
    INTER_AREA)
resized2 = cv2.resize(warped_binary_white, dim, interpolation =
    cv2.INTER_AREA)
resized3 = cv2.resize(combined_binary_white, dim, interpolation
    = cv2.INTER_AREA)
resized4 = cv2.resize(region_of_intrst_white, dim, interpolation
     = cv2.INTER_AREA)

result[:height, result.shape[1]-2*width:result.shape[1]-width] =
     weighted_img(initial_img=resized2,
    img=result[:height, result.shape[1]-2*width:result.shape[1]-
        width], alpha=1, beta=0.5)
result[:height, result.shape[1]-width:result.shape[1]] =
    weighted_img(initial_img=resized,
    img=result[:height, result.shape[1]-width:result.shape[1]],
        alpha=1, beta=0.5)
result[height:2*height, result.shape[1]-2*width:result.shape[1]-
    width] = weighted_img(initial_img=resized3,
```

```
16      img=result[height:2*height, result.shape[1]-2*width:result.
            shape[1]-width], alpha=1, beta=0.5)
17  result[height:2*height, -width:] = weighted_img(initial_img=
        resized4,
18      img=result[height:2*height, -width:], alpha=1, beta=0.5)
```

Basically, the images resulting from the intermediate steps of the pipeline to be shown are downsampled and blended with the pixels of the top right region of the final image of the pipeline.

6.1.1 Video output

The output video of the project is in `output/project_video.mp4` and it can be downloaded from this link.

6.2 Advanced Video pipeline

The new pipeline steps are implemented in the function `pipeline_advanced(image)` defined in cell 30 (executed cell 19). Steps 1 through 6 are the same of the previous pipeline.

6.2.1 Line class

As suggested in the lecture, a class `Line` has been defined in cell 30 (executed cell 19). The goal of this class is to average the last fitted lines so as to get a better result by smoothing sudden changes such as those caused by bumps. The number of fitted lines the class keeps track of is defined by the class variable `Line.MAX_COUNT`. The instance variable `recent_fits` is an array containing the most recent fits, `recent_xfitted` an array containing x values corresponding to the most recent fits, `best_fit` contains the average of `recent_fits` and it is the one used to draw the lines, `idx` is a variable to keep track of the oldest element to be substituted at the next iteration. The class offers the method `update` to be called when the fit is valid so as to update the arrays and reperform the means and `invalidate` when the found lines are not consistent.

Note that now the curvature calculation is performed in the `update` method since the class stores all the required information. The class also calculates how far the found line is from the middle of the image, so as to simplify the calculation of the vehicle offset later (subsubsection 6.2.3).

```
1 self.radius_of_curvature = measure_curvature(ploty=ploty, fit=
      self.best_fit)
2 line_pos = self.best_fit[0]*img_shape[0]**2+self.best_fit[1]*
      img_shape[0]+ self.best_fit[2]
3 car_center = img_shape[1]/2  # we assume the camera is centered
      in the car
4 self.line_base_pos  = abs(line_pos - car_center) * xm_per_pix
```

Two instances are needed: one for the left line and one for the right. Since the instances must keep their states between different function calls, they have been defined at the bottom of the new pipeline function as function instances

together with other two variables required later (since functions are objects in Python):

```
def pipeline_advanced(image):
    ...
    return result
pipeline.left_line = Line()
pipeline.right_line = Line()
pipeline.subsequent_invalid_frames = 0
pipeline.first_initialisation = True
```

6.2.2 Changes in steps 7 and 8: finding the lanes and fit the polynomial

First a new method for searching new lane pixels has been implemented in cell 30, section *Functions for lane detections* with the function `search_around_poly`. Instead of performing the windowing analysis from scratch, the function exploits the `best_fit` of each `Line` instance, if any, and labels as lane pixels all those pixels that lie between a left shifted and a right shifted versions of the lines. The amount of shifting is given by the variable `margin` set to 100. The core of the function that selects lane pixels is the following::

```
def search_around_poly(binary_warped, left_fit, right_fit):
  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)))
```

The function then calls `fit_poly_search_around` to fit the two polynomials. The rest of the function works similarly to `search_lane_pixels`.

If the `Line` instances have stored a `best_fit` variable then we search lane pixels around the two lines. The result of `search_around_poly` is checked for consistency by the function `sanity_check`, which checks that the two lines have a comparable curvature and that the two lines are parallel by checking their distance at the bottom, middle and top of the eye-bird image in the function `check_parallel_line` (cell 30):

```
def check_parallelline(fit_left, fit_right, ploty):
  global xm_per_pix
  THRESHOLD = 0.35
  bottom_lines_dist = (calc_line_dist(fit_left, ploty[-1]) -
      calc_line_dist(fit_right, ploty[-1]))*xm_per_pix
  middle_lines_dist = (calc_line_dist(fit_left, ploty[int(len(
      ploty)/2)]) -
    calc_line_dist(fit_right, ploty[int(len(ploty)/2)]))*
        xm_per_pix
  top_lines_dist = (calc_line_dist(fit_left, 0) - calc_line_dist
      (fit_right, 0))*xm_per_pix
  bottom_middle_diff = abs(bottom_lines_dist-middle_lines_dist)<
      THRESHOLD
```

```
    bottom_top_diff = abs(bottom_lines_dist-top_lines_dist)<
        THRESHOLD
    middle_top_diff = abs(middle_lines_dist-top_lines_dist)<
        THRESHOLD
    return bottom_middle_diff and bottom_top_diff and
        middle_top_diff


def sanity_check(ploty, curr_fit_left, curr_fit_right):
    if not (curr_fit_left.any() and curr_fit_right.any()):
      return False
    sanity = True
    left_curv = measure_curvature(ploty=ploty, fit=curr_fit_left)
    right_curv = measure_curvature(ploty=ploty, fit=curr_fit_right
        )
    curv_diff = abs(left_curv - right_curv)
    if curv_diff>10000:
      sanity=False
    bottom_line_dist = calc_line_dist(curr_fit_left, ploty[-1]) -
        calc_line_dist(curr_fit_right, ploty[-1])
    if not check_parallelline(curr_fit_left, curr_fit_right, ploty
        ):
      sanity = False
    return sanity
```

If the check is passed then the `Line`s instances are updated, otherwise invalidated and the counter `pipeline.subsequent_invalid_frames` is incremented to keep track of the subsequent invalid frames. If we get 3 subsequent invalid frames, then the calculation of the lines is performed using the histogram and windows and no check is performed in this case. The code is the following:

```
if pipeline.left_line.best_fit.any() and pipeline.right_line.
    best_fit.any() and pipeline.subsequent_invalid_frames<3:
  which_method = "poly search "
  res_img, left_fit, right_fit, ploty, points_left, points_right
       = search_around_poly(
      warped*255,pipeline.left_line.best_fit, pipeline.
          right_line.best_fit)
  pipeline.res_img = res_img
  is_sane, reason = sanity_check(ploty, left_fit, right_fit)
  if is_sane:
      pipeline.right_line.update(right_fit, points_right, ploty,
           region_of_intrst.shape)
      pipeline.left_line.update(left_fit, points_left, ploty,
          region_of_intrst.shape)
      sanity_check_txt = "sanity check OK"
      pipeline.subsequent_invalid_frames = 0
  else:
      pipeline.subsequent_invalid_frames += 1
      pipeline.right_line.invalidate()
      pipeline.left_line.invalidate()
      sanity_check_txt = "sanity check NOT OK: " + reason
else:
  which_method = "Search anew"
  pipeline.subsequent_invalid_frames = 0
  res_img, left_fit, right_fit, ploty, points_left, points_right
       = fit_polynomial(warped*255)
```

```
21    pipeline.res_img = res_img
22    pipeline.points_left = points_left
23    pipeline.points_right = points_right
24    pipeline.right_line.update(right_fit, points_right, ploty,
          region_of_intrst.shape)
25    pipeline.left_line.update(left_fit, points_left, ploty,
          region_of_intrst.shape)
```

6.2.3 Vehicle offset

The vehicle offset is calculated using the lines position with respect to the centre of the image, which is stored in the two lines instances:

```
1 center_offset = (pipeline.right_line.line_base_pos - pipeline.
      left_line.line_base_pos)/2
```

6.2.4 Video output

The output video of the project is in `output/project_video.mp4` and it can be downloaded from this link. Note the delay in the algorithm to adapt to lane changes. This is good when the car goes over a bump but it takes some frames to adapt from a curvature to straight line.

7 Challenge videos

Both `pipeline()` and `pipeline_advanced()` do not work well with the challenge videos.

7.1 Challenge video

The output videos of this challenge are `output/challenge_video.mp4` for the simple pipeline and `output/challenge_video_advanced.mp4` for the advanced pipeline. The `pipelien_davanced` algorithm seems to work a little better in this case.

I can see two difficulty in this video: first of all the lane lines are only visible near the car but they are too blurred to be detected further away. Secondly, the algorithm seems to get tricked by the discontinuity in the road that are sometimes parallel to the lane lines and by the shades on the left caused the wall. These lines are probably detected by the magnitude threshold step and classified as lane pixels.

7.1.1 Steps to be tried

A possible solution might be to select a better and stricter region of interest, also by removing the pixels by the two lane lines. However, this might cause problems in case of curvature, since less pixels will be available to fit the lines. Another approach might be to rely only on colour threshold, so as to find only differences given by the white and yellow lane lines.

7.2 Harder challenge video

The output videos of this challenge are `output/harder_challenge_video.mp4` for the simple pipeline and `output/harder_challenge_video_advanced.mp4` for the advanced pipeline.

This video has other difficulties from the previous one. First of all, video frames have a lot of textures that tricks the algorithm. The texture is given by the fact that the road is narrow, so the region of interest can do nothing but to select also pixels that are on the side of the road and that belongs to other objects in the image, such as trees, cars coming from the opposite directions and so on. Secondly, the road curvatures have a very small radius, compared to the project video, so we have less lane pixels on which to fit the polynomials. Furthermore, the curvatures change rapidly and drastically: for example at the second 2 of the video we a right curvature after a left one: in this case a higher order polynomial fit would be needed. This is further exacerbated by the fact that there are sudden changes of brightness that make difficult choosing threshold values that would all the cases.

7.2.1 Steps to be tried

Again, also here one might try to choose a better region of interest to rule out non-useful objects from the image. Also given given the radius of curvature, one can try to get only the region near the car (closer to the bottom of the image) since the upper part is less useful than in the project video. Another solution is to choose adaptive thresholds according to the brightness of the frame or perform an analysis only in the HLS space, so as to look only at the colour channel discarding the light channel.