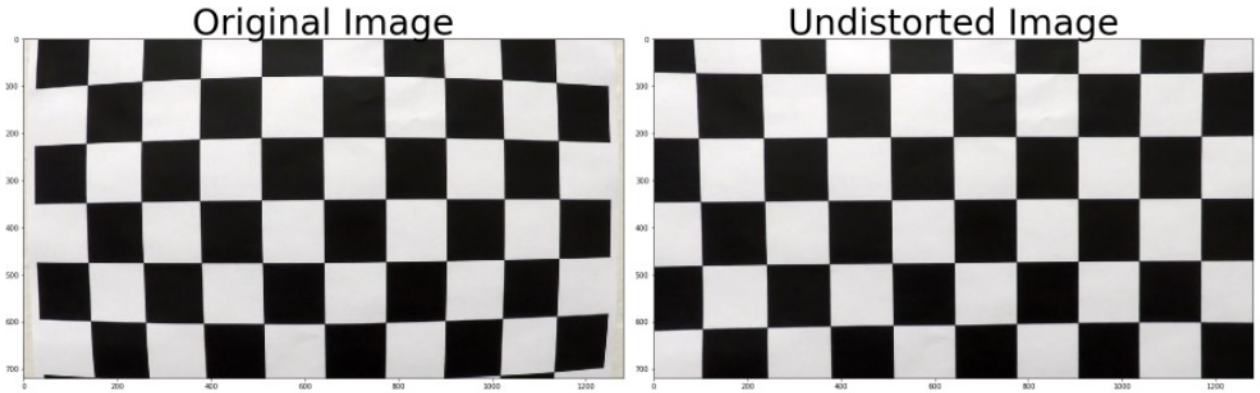
1. Camera Calibration

As first step I use the calibration camera method in order to rule out the unwanted embedded distortions produced by the device.

Threrefore, I implemented the procedure availing me of two OpenCV functions, respectively `findChessboardCorners` and `calibrateCamera`. Different images of chessboards taken from different angles were provided in order to calculate the parameters to adjust several type of potential distortions inducted by the camera.

Applying `findChessboardCorners` I was able to grasp the location of internal corners of the chessboards, which are used to fed to `calibrateCamera` which in turn it returns distortion coefficients and camera calibration. After that I used `undistort`, whose goal is to undistort a given image on the base of the coefficients passed.

The result of this few steps can be observed below:

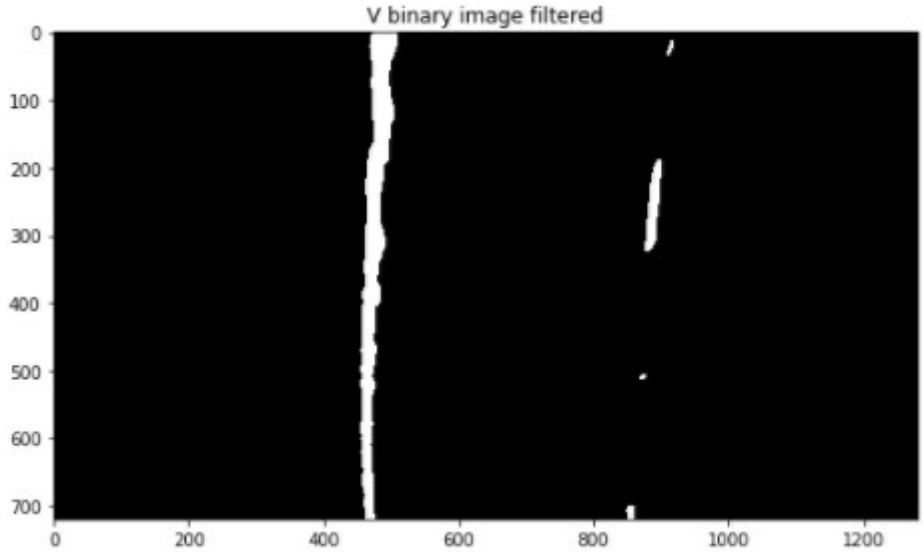


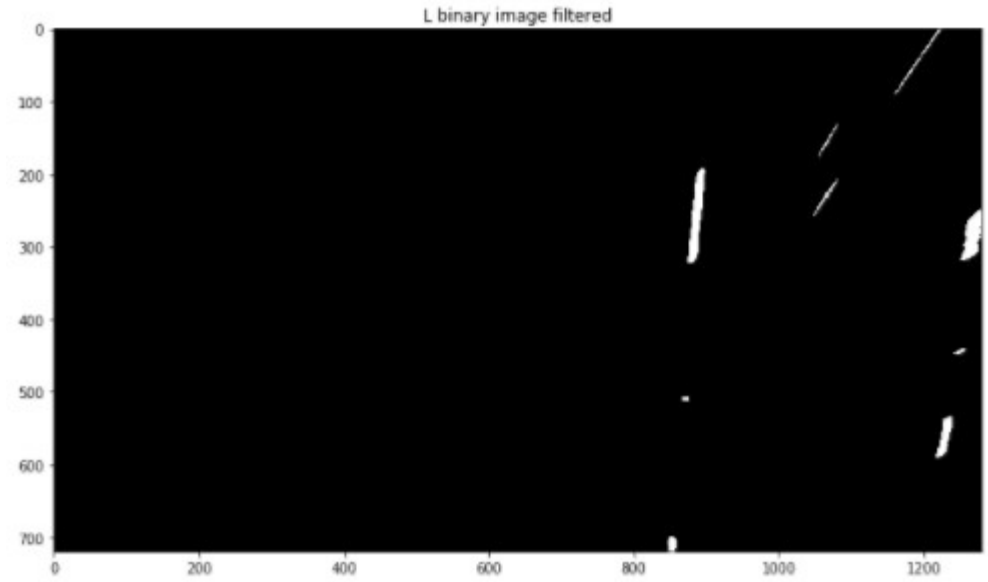
2. Pipeline

Below an image of the result of applying `undistort` over a test image of the dataset.

The effect of the process can be stand out at the corner of the image even if it isn't so marked.

The most interesting part of the project was all about trying to apply several colour spaces and different threshold to the sample image I chose. Exploring a myriad of combinations I came to the conclusion that it could have been right for me L channel of the HLS colour space to isolate white lines and both the B channel of the LAB colour space and V channel of the HSV colour space to isolate yellow lines. After multiple attempts observing the result on video output I opted for V channel as yellow lines seeker. In addition to that I firmly decided of not using any sort of gradient thresholds that in my opinion made worse the quality of the final outcome. On the other hand to be minimally tolerant to changes in lighting I tuned the threshold for each channel. Moreover, I normalized the maximum values of the HLS L channel and the V channel to 255, since the values the lane lines in these channels are affected on lighting conditions. Below are examples of thresholds in the HLS L channel and the V channel:





Below are the results of applying the binary thresholding pipeline to various sample images:













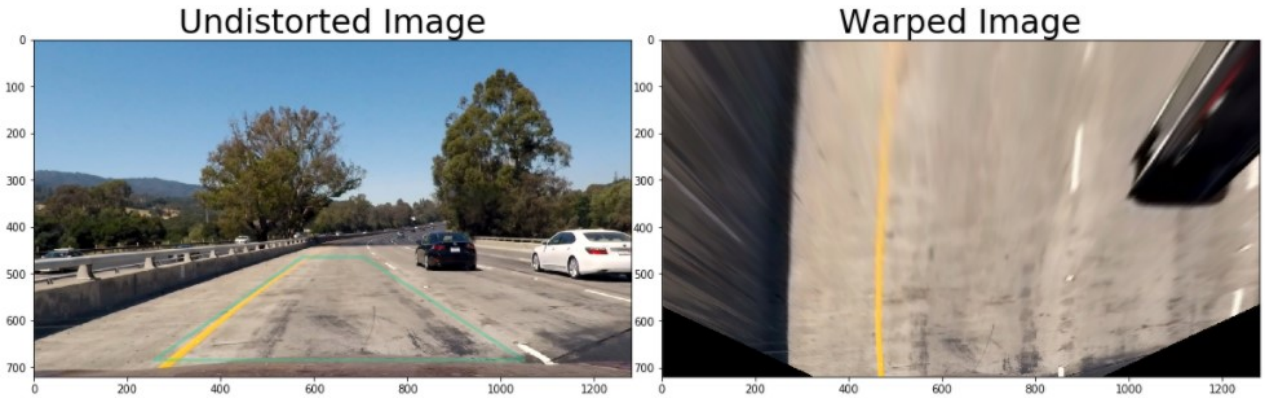


The code for my perspective transform is titled "Warped Image" in the Jupyter notebook, in the seventh and eighth code cells from the top. The `warp()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I hardwired the source and destination points in the following manner:

```
src = np.float32([(575,465),
                  (705,465),
                  (258,685),
                  (1050,685)])
dst = np.float32([(450,0),
                  (w-450,0),
                  (450,h),
                  (w-450,h)])
```

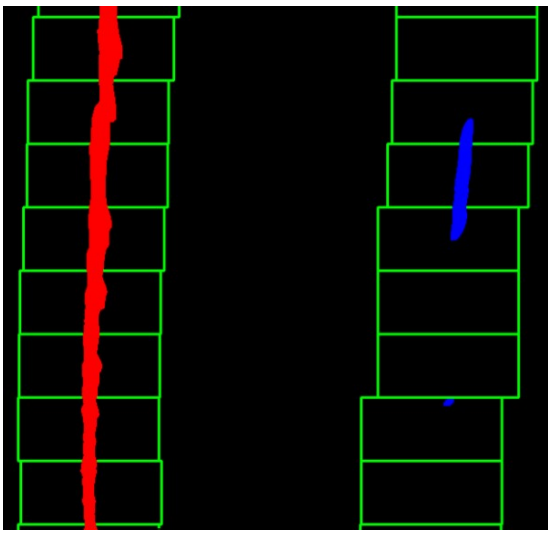
Where w stands for the width and h for the height of the original image.

Below the outcome a sample of bird's eye prospective starting from the original picture.

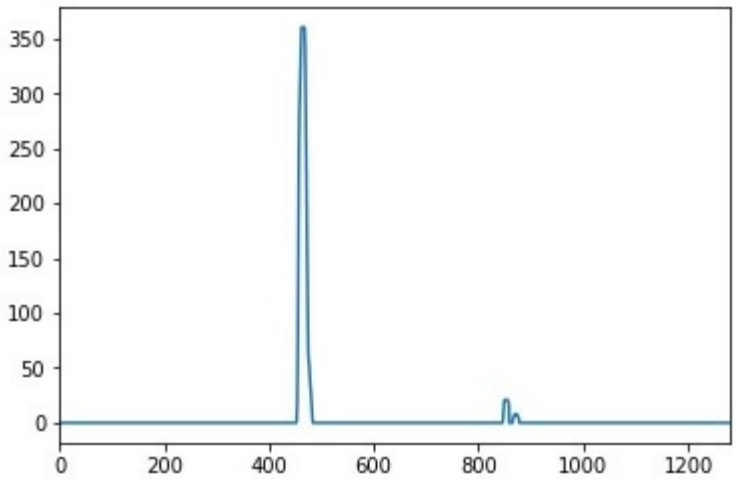


The functions `sliding_window_and_polyfit`(14th cell) and `polyfit_using_prev_fit`, which identify lane lines and fit a second order polynomial to both right and left lane lines, are clearly labeled in the Jupyter notebook as "Finding lines from bird's eye prospective" and "Find lines from previous frame".

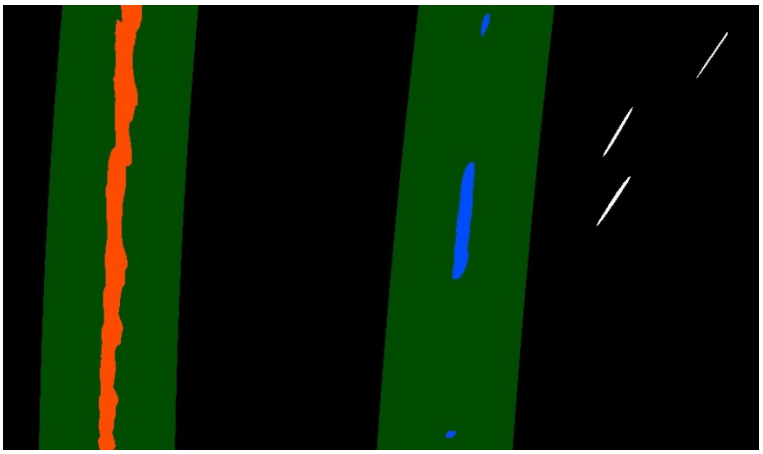
`sliding_window_and_polyfit`：
To make more robust the algorithm I chose to use to compute a histogram of the quarters just left and right of the midpoint, since this approach turned to be useful to reject lines from adjacent lanes. Besides, the function identifies ten windows from which to detect lane pixels, each one centered on the midpoint of the pixels from the window below. Doing this as you can see from the picture above the lane lines are tracked up to the top of the binary image. Numpy `polyfit()` method is used to fit a second order polynomial to each set of pixels.



The image below depicts the histogram generated by `sliding_window_and_polyfit`; the left and right lanes - the two peaks nearest the center - are well highlighted:



The `polyfit_using_prev_fit` function performs basically the same task, but with less computational approach relying on a previous fit (from a previous video frame) and only searching for lane pixels within a certain range of that fit. The green shaded area is the range from the previous fit, and the yellow lines and red and blue pixels are from the current image:



The radius of curvature calculated in the code derive from

$$\text{Radius of curvature} = \frac{\left[1 + \left(\frac{dy}{dx}\right)^2\right]^{3/2}}{\left|\frac{d^2y}{dx^2}\right|}$$

This is translated in python with the following

```
curve_radius = ((1 + (2*fit[0]*y_0*y_meters_per_pixel + fit[1])**2)**1.5) /
np.absolute(2*fit[0])
```

`fit[0]` is the first coefficient (the y-squared coefficient) of the second order polynomial fit, and `fit[1]` is the second (y) coefficient. `y_0` is the y position within the image upon which the curvature calculation is based . Instead `y_meters_per_pixel` is the factor used for converting from pixels to meters. In particulary this conversion was also used to generate a new fit with coefficients in terms of meters.

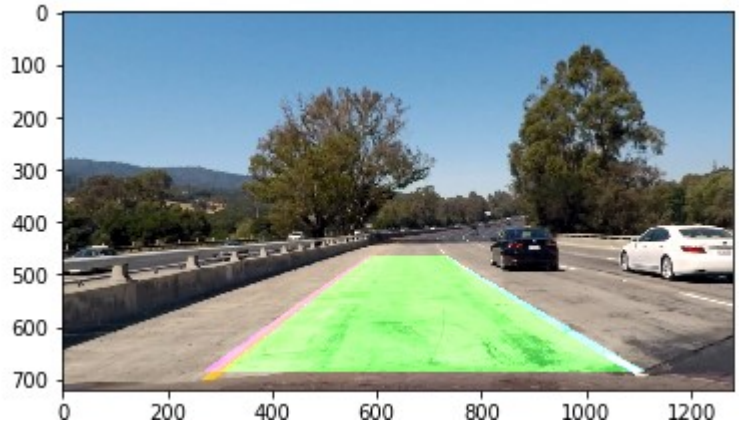
Therefore the position of the car with respect to the center of the lane is calculated like this:

```
lane_center_position = (r_fit_x_int + l_fit_x_int) /2
center_dist = (car_position - lane_center_position) * x_meters_per_pix
```

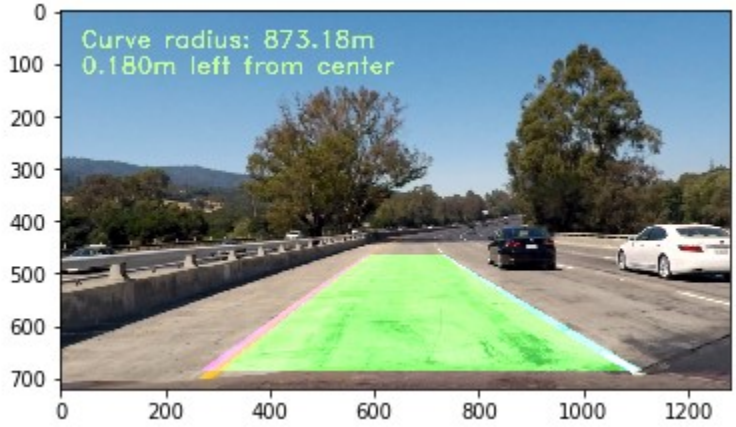
where `r_fit_x_int` and `l_fit_x_int` are the x-intercepts of the right and left fits, respectively.

So center_dist represents the difference between these intercept points and the image midpoint.

In " Drawing lanes on the original picture" (22th cell) you can find the results of the previous points. The draw lane function is appointed to generate the detection by means of a polygon based on plots of the left and right fits, warped back to the perspective of the original image using the inverse perspective matrix `Minv` and overlaid onto the original image.



Below is another example of the results of the `draw_additional_data` function, which writes text identifying the curvature radius and vehicle position data onto the original image:



3. Conclusion

Especially in the challenge video I'd like to improve my pipeline in order to get it more robust against different light conditions as it shown in the clip just under the bridge where my algorithm drew lane lines wobbly and not at all precise. Not to mention the harder challenge video, where anything worked.