Udacity CarND Project 4
Advanced Lane Detection

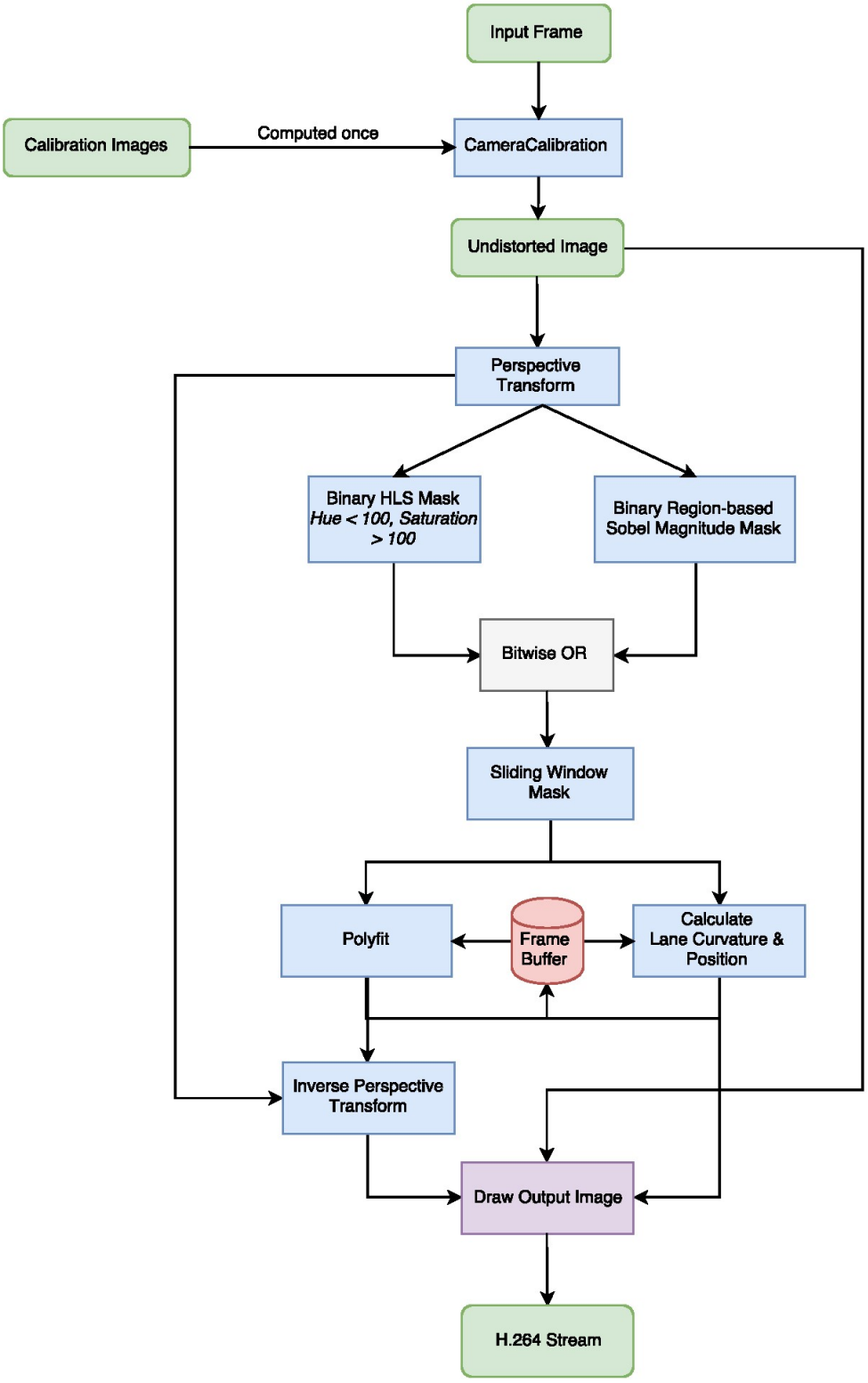
PROJECT GOALS

The goals / steps of this project are the following:

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

Pipeline

Here is an overview of my final pipeline that I implemented for lane detection on video data. The next sections cover each individual component in more detail:



CAMERA CALIBRATION

In order to perform camera calibration, I decided to create a **`CameraCalibration`** class, in order to make it easier to undistort images on the fly throughout the project. The benefit of using a class in this scenario is that it means that I only need to compute the matrix and coefficients once, and it can then be applied easily at any point in my pipeline. It works like this:

```
# Calibrate the camera and create the camera matrix & distortion coefficients
calibration = CameraCalibration(imgs=list_of_images, height=6, width=9)

# Then, when running on a new image:
undistorted = calibration.undistort(img)
```

When initialised, the class takes in a list of images, and the size of the chessboard it is supposed to detect in the image. In our scenario, we have a chessboard which is 9x6, so this is passed to the class.

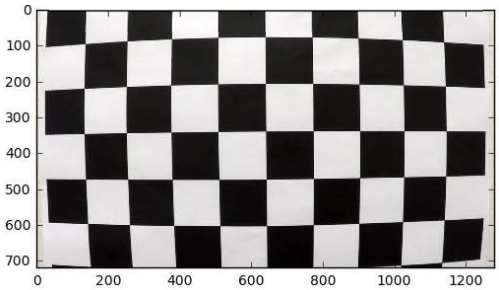
In order to compute the camera matrix and distortion coefficients, **`CameraCalibration`** uses OpenCV's **`findChessboardCorners`** function on grayscale version of each input image in turn, to generate a set of image points. I then use **`numpy.mgrid`** in order to create a matrix of (x, y, z) object points.

Once I have the image points and the matching object points for the entire calibration dataset, I then use **`cv2.calibrateCamera`** to create the camera matrix and distortion coefficients. These variables are stored inside the class, until **`calibration.undistort(img)`** is called, at which point they are passed into cv2.undistort, returning an undistorted version of the user's input image.

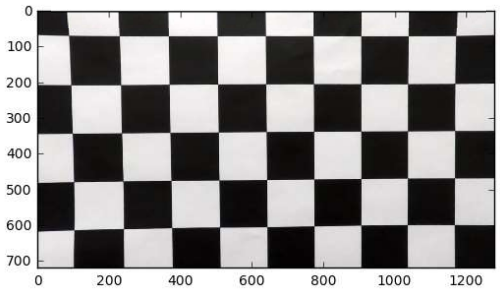
The class will gracefully skip any images that have parts of the chessboard cut-off, printing a warning. I found that this was the case with three of the images in the Udacity dataset.

The full code for calibration can be seen in the `calibration.py` in this directory.

Before Calibration

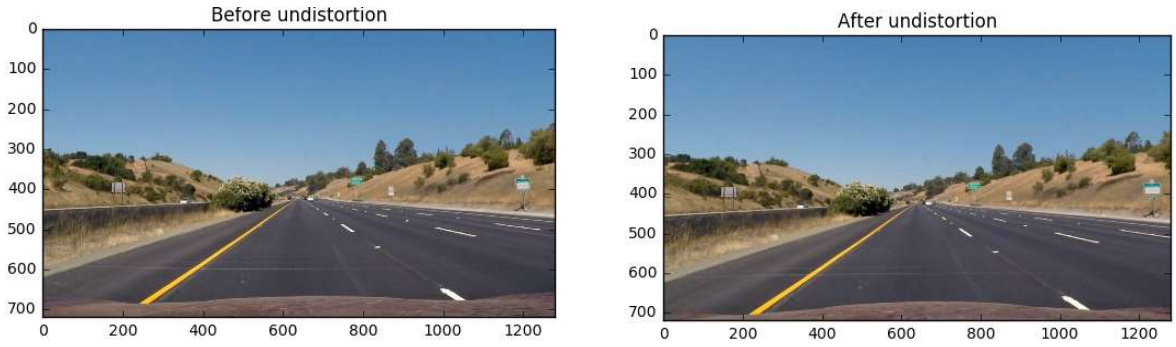


After Calibration



PIPELINE: DISTORTION-CORRECTION

Here is the output of the camera-calibrated distortion correction on my image:



PIPELINE: BINARY THRESHOLDING

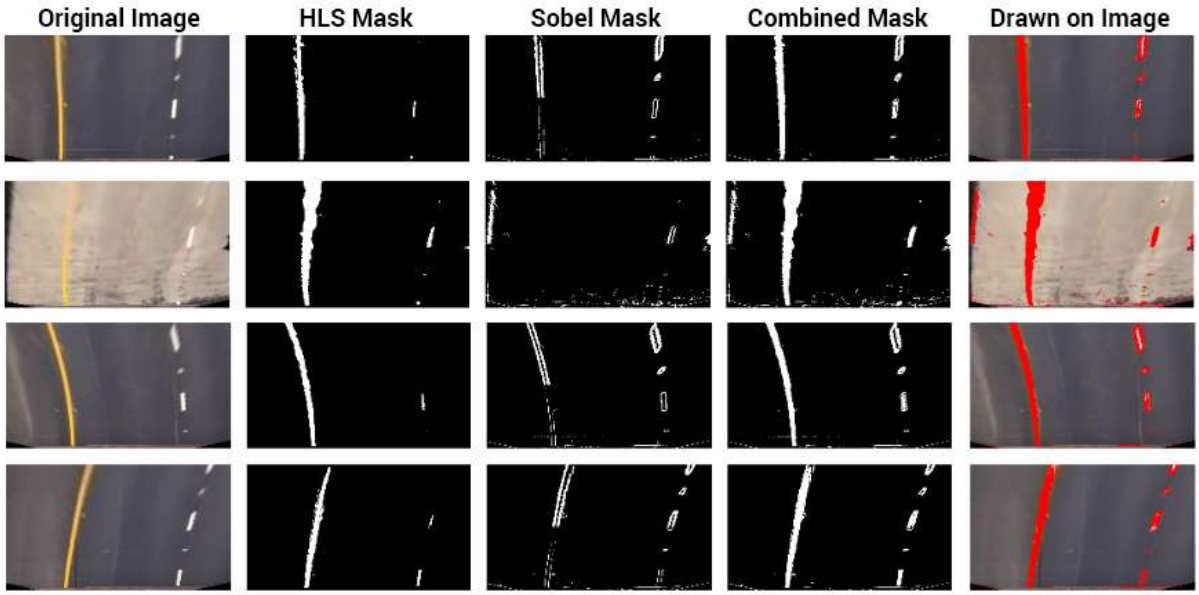
To create my binary image mask, I used a combination of different transforms.

First, I used **`cv2.cvtColor()`** to convert to HLS space, where I created a binary mask – detecting pixels where the hue < 100 and saturation > 100 as lane line pixels.

After this, I also created a mask which used a threshold on the absolute Sobel magnitude. I decided to go with a mask which applied different Sobel thresholds to the top and bottom halves of the image, as the top half of the image had smoother gradients due to the perspective transform. In the end, I selected any pixel with > 10 sobel magnitude at the top of the image, and > 35 at the bottom of the image.

I found that the HLS threshold worked better closer to the car, and the Sobel threshold worked better further away. Thus, a bitwise OR of these two thresholds gave me one that I was very happy with.

My binary thresholding code can be found in the `mask_image` function in cell 5 of `pipeline.ipynb`



PIPELINE: PERSPECTIVE TRANSFORM

For my perspective transform, I wrote a **`PerspectiveTransformer`** class, which in a similar fashion to the Camera Calibration allows me to warp/unwarp perspective in a single line while only having to compute the transformation matrix once.

I used the **`cv2.getPerspectiveTransform`** function to compute the transformation and inverse transformation matrices, and then applied these to images using **`cv2.warpPerspective`**. The code can be seen below:

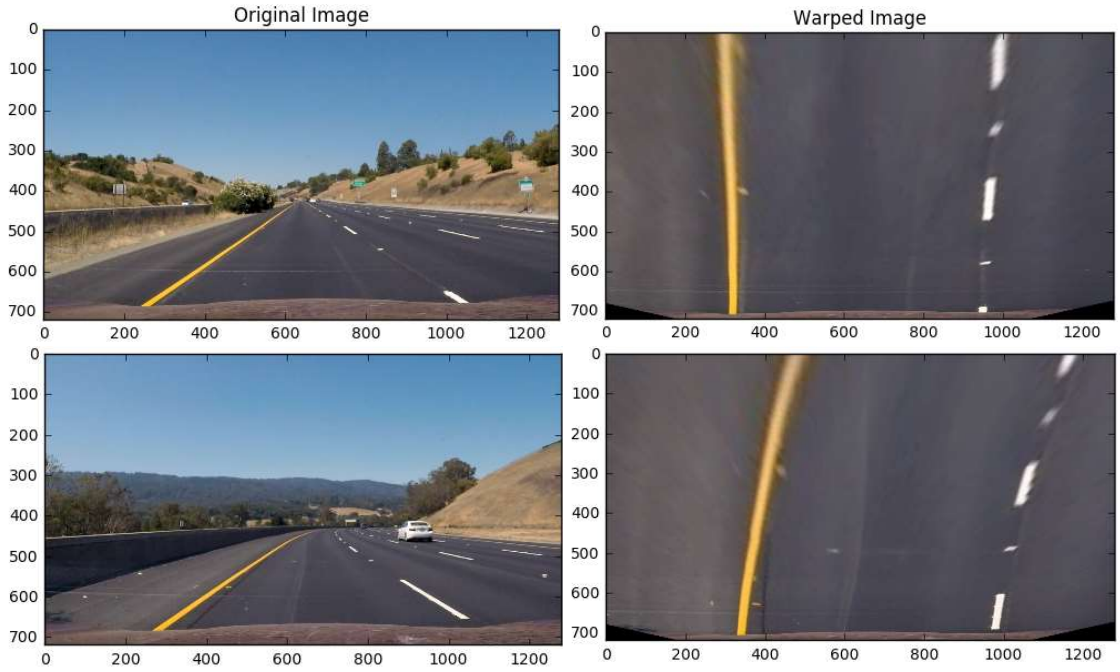
```
# Class for perspective transforms
class PerspectiveTransformer():
    def __init__(self, src, dist):
        self.Mpersp = cv2.getPerspectiveTransform(src, dst)
        self.Minv = cv2.getPerspectiveTransform(dst, src)

    # Apply perspective transform
    def warp(self, img):
        s = image.shape
        return cv2.warpPerspective(img, self.Mpersp, (s[1], s[0]))

    # Reverse perspective transform
    def unwarp(self, img):
        s = image.shape
        return cv2.warpPerspective(img, self.Minv, (s[1], s[0]))
```

as well as **in cells 4 and 12 in `pipeline.ipynb`.**

For my transformation, I decided to use the following **`src`** and **`dst`** points:

```
src = [[585, 460], [203, 720], [1127, 720], [695, 460]]
dst = [[320, 0], [320, 720], [960, 720], [960, 0]]
```

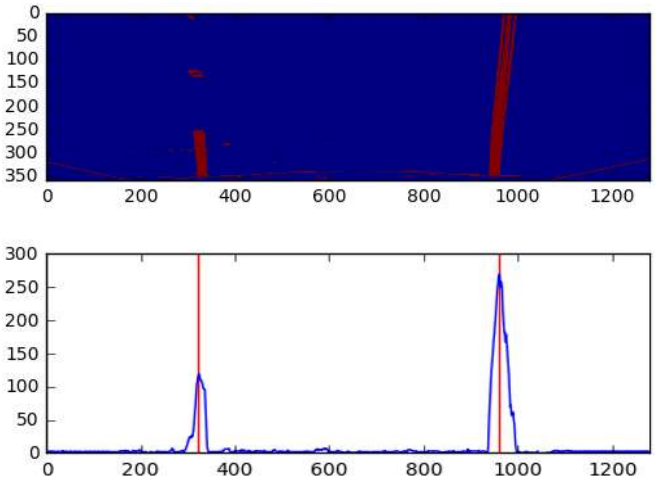
This led to the following transformation:



Pipeline: Lane-fitting

After creating the binary mask, the next task was to find and fit the lane lines. This was done with a multi-stage approach.

Firstly, in order to find where the lane line begins, I created a histogram of the masked pixels in the bottom half of the image, and selected the biggest peak in both the left and right sides as the starting point:



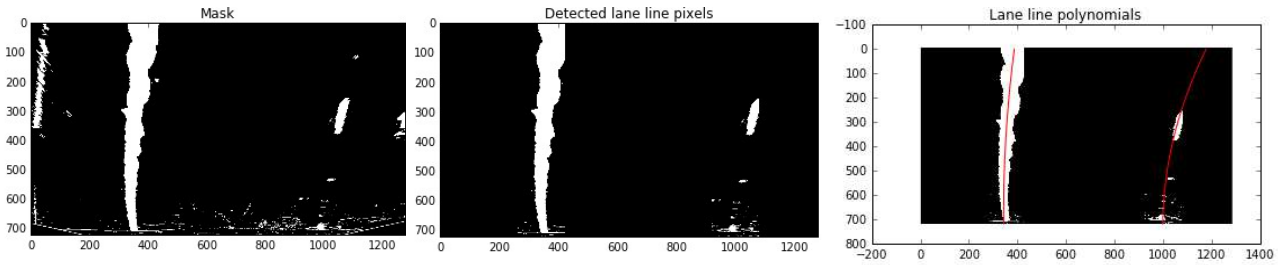
I then used **`numpy`** to split the image into a specified number of chunks (10), and in each chunk, repeated the following process:

1) Take the mean pixel x-position of the last chunk (or histogram)
2) Select all masked pixels within 80 pixels of this value
3) Add the coordinates of these pixels to an array

This meant that I ended up with an array of (x, y) values for both the left and right lane lines.

I used **`numpy.polyfit`** to then fit a quadratic curve to each lane line, which can then be plotted on the input frame.

In order to smooth out the final curve, I took a weighted mean with the last frame's polynomial coefficients. (w=0.2)



The code for histogram peak detection, sliding window and lane fitting are in cells 6 and 7 in the notebook.

PIPELINE: LANE CURVATURE

In order to calculate the lane curvature radius, I scaled the x and y coordinates of my lane pixels and then fit a new polynomial to the real-world-sized data. Using this new polynomial, I could then use the radius of curvature formula below to calculate the curve radius in metres at the base of the image:

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

I performed this on both the left and right lane lines, and then took the mean of these values for my final curvature value. I found that the value produced by the formula was quite susceptible to noise, so in my final pipeline I took a weighted average with the calculated curvature of the last frame (w=0.05), allowing the displayed curvature value to be a more accurate and smooth representation.

```
# Calculate the lane line curvature
def get_curvature(poly, mask):
    yscale = 30 / 720 # Real world metres per y pixel
    xscale = 3.7 / 700 # Real world metres per x pixel

    # Convert polynomial to set of points for refitting
    ploty = np.linspace(0, mask.shape[0]-1, mask.shape[0])
    fitx = poly[0] * ploty ** 2 + poly[1] * ploty + poly[2]

    # Fit new polynomial
    fit_cr = np.polyfit(ploty * yscale, fitx * xscale, 2)

    # Calculate curve radius
    curverad = ((1 + (2 * fit_cr[0] * np.max(ploty) * yscale + fit_cr[1]) ** 2)
                ** 1.5) / np.absolute(2 * fit_cr[0])
    return curverad
```

This code can also be seen in cell 8 of the notebook. The weighted average is computed in the `process_frame` function in cell 11.

PIPELINE: LANE POSITION

In order to compute the position of the car in the lane, I used the lane polyfit to find the bottom position of the left and right lanes respectively. Assuming the width of the lane was 3.7 metres, I calculated the scale of the transformed image, and then used the distance between the centre of the image and the centre of the lane to calculate the offset of the car.

I did not use a weighted mean with the last frame here, as I found the offset value to be much more stable than the lane curvature.

```python
# Find the offset of the car and the base of the lane lines
def find_offset(l_poly, r_poly):
    lane_width = 3.7  # metres
    h = 720  # height of image (index of image bottom)
    w = 1280 # width of image

    # Find the bottom pixel of the lane lines
    l_px = l_poly[0] * h ** 2 + l_poly[1] * h + l_poly[2]
    r_px = r_poly[0] * h ** 2 + r_poly[1] * h + r_poly[2]

    # Find the number of pixels per real metre
    scale = lane_width / np.abs(l_px - r_px)

    # Find the midpoint
    midpoint = np.mean([l_px, r_px])

    # Find the offset from the centre of the frame, and then multiply by scale
    offset = (w/2 - midpoint) * scale
    return offset
```

This code can also be seen in cell 10 in the notebook.

Pipeline: Final Output

My **`process_frame`** function outputs the original image with the lane area drawn on top, and the curvature radius and lane offset in the top left corner.

Here are some example output frames:

VIDEO OUTPUT

The result of my algorithm on the project video can be seen in **`processed_video.mp4`** in this directory or at:

https://www.youtube.com/watch?v=MiTto2aWWpc

DISCUSSION

The part of my pipeline that needed the most tuning, and the one I feel makes the biggest difference is the masking of lane line pixels. I found it difficult to get a good mask on the white lane lines, especially in the lighter coloured road, which meant that I had to use a combination of two masks in order to get a good result on the project video. However, I still feel that given radically different lighting or road conditions, then I would be left with a bad mask, and as such, the polyfit would fail to find the lane lines.

In order to make the lane line pixel detection more robust, I could have added some normalisation of lighting, or used some form of shape detection in order to find the distinct shape of lane lines (strong gradients on the edge, very weak gradients in the centre), and remove any noisy pixels on the road. In addition, I could have implemented some sort of outlier detection which discarded pixels which appeared in very small groups.

Finally, given more time, I would have liked to implement an iterative algorithm, which bases the search location of the next frame based on the detected lane in the last frame, instead of recreating the sliding window. This could likely have made my algorithm more robust to jitter and made it run at a faster speed.