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.

Rubric Points

Writeup / Readme

1- Provide a Writeup / README that includes all the rubric points and how you addressed each one. This is the writeup.

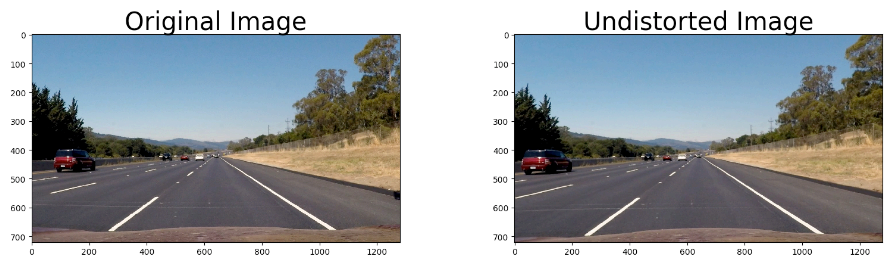
Camera Calibration and distortion correction

1.- Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for the camera calibration is in the **calibrate.py** file. It simply loads the images from "./images/camera_cal" and uses the checkerboard corners to compute the camera matrix and distortion coeficients. Result data is stored in the "wide_dist_pickle.p" file to be used by the rest of the programs.

Calibrate.py may work just computing the camera matrix and distortion once and then use it just to generate an undistorted image and test perspectives.

Results of the execution of calibrate.py are:



where there is the original and the undistorted image.

Pipeline

The pipeline is in the **Advanced_Lanes.py** program.

Auxiliary programs are

- **Calibrate.py** to get the calibration matrix and distortion coeficients

- **filter_design.py** for testing filters
- **filter_parameters.py** for testing and searching filter parameters

In the **Advanced_Lanes.py** in

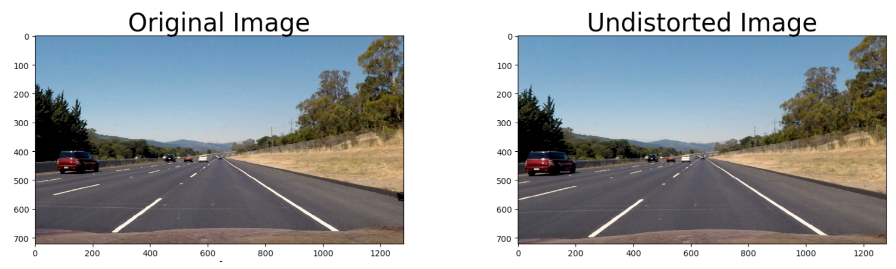
- line 2214 must be changed to the video folder
- line 2219 must have an 0, 1 or 2 for the different videos, project, challenge or more_challenge
- lines 2246 to 2248 have examples of use. Only one must be uncomented.

Options as log folders, video output name, etc are defined I the

Process_video for videos

Process_folder for images

1.- Provide an example of a distortion corrected image:



It has already been presented in the image calibration criteria.

2.- Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

All image processing is in the **process_image** function. It receives an image and returns an array of warped, filtered images.

The process used is:

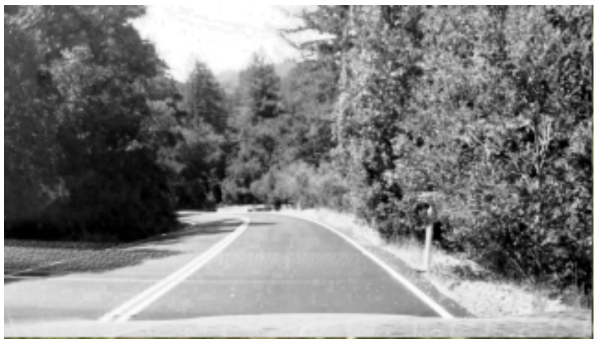
- Undistort image
- Convert to **HSV** and separate the channels
- Apply **EqualizeHist** to value and saturation channels
- Apply a **GaussianBlur** with a 7x7 kernel to value and saturation channels
- Call a special **remove_dark** function to be explained
- Apply a set of filters and warp the resulting images

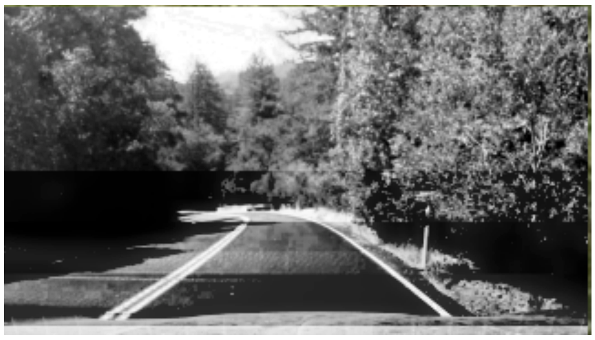
The **remove_dark** function was originally used to remove dark lines and soft some edges. Finally is a more general light equalization. It's process is:

- Divide the lower half of the image (with the exception of the bottom 20 pixels) in **n** horizontal slides.
- For each slide compute the average **V** value in the center of the image (between 20% and 80% of the width)
- Select all points below this average value and apply to them a BoxFilter with 50 by 50 pixels) in **S** and **V** channels

- `Just for the V channel move all below average intensities between 0 and 50 and expand the ones over the average to 50–255`

The idea is that low intensity features are blurred and will generate less gradients as high intensities have a greater range. Here is an example of the value channel, first the original image:

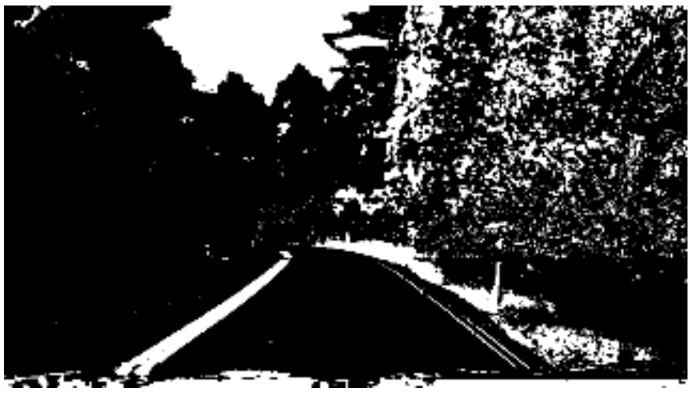




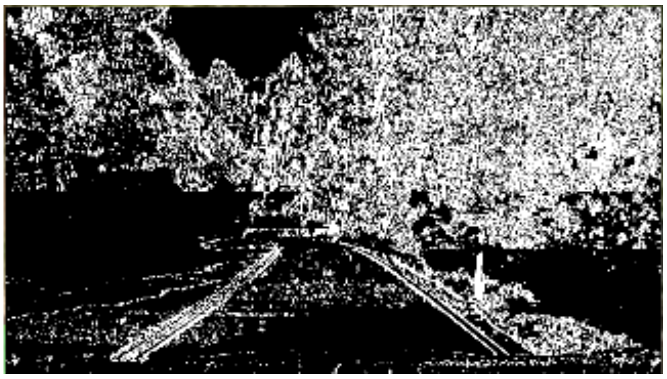
Left image is just after the equalization and GaussianBlur, right one is the same but after **remove_dark** with 3 slides.

To get the bitmap images I have used 2 filters and an additional color filter.

First one is the one talked in the course, just selecting pixels with gradient in x, and y direction and those with special values and saturations. It is in the **gradient_filter** function that uses the **color_threshold** and the **abs_sobel_thresh**. Here there is an example of the same image:

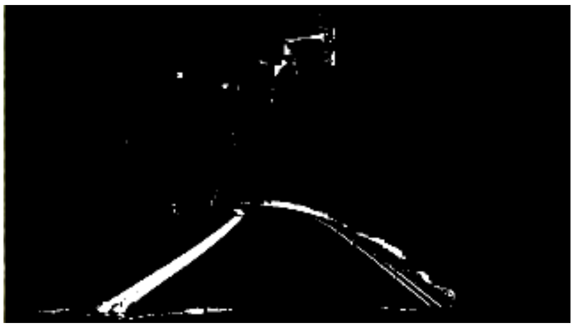


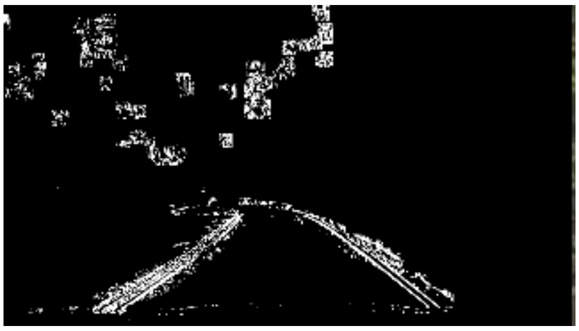
The other algorithm uses similar sobel computations linked with values in gradient x and saturation but "or's" them together so usually is messier. It is implemented in the **comples_sobel** function.



Mainly in the third video color is specially important. As you may see in the last image, there is a lot of noise near the right lines.

Selecting hue is difficult but a way to do it is select just the **road**, expand its borders and intersect with the results of the other algorithms so we have a mask that may be intersected with the other results giving





Which are a lot better.

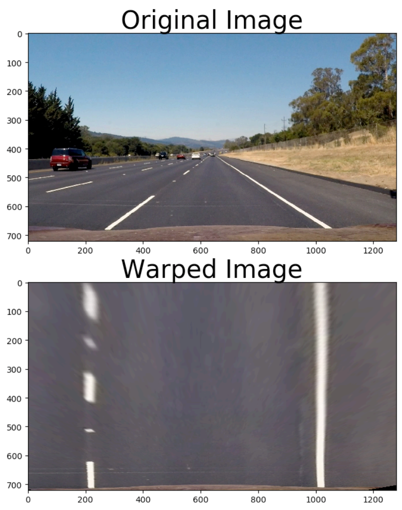
So we have 4 possibilities in total. Depending the situation we may like to use one or another result so the main filtering function, **super_filter** may return an array of all identified as “gr”, “so”, “grt” and “sot” corresponding to the images presented.

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

At the beginning the perspective transformation was computed manually and stored in the same file as camera information, but afterwards a small piece of code has been used to generate the transformation just from 1 parameter that may be linked to the camera focal length and information about how to compress the image so bends are visible when warped. It is in the **perspective** function at the beginning of the file.

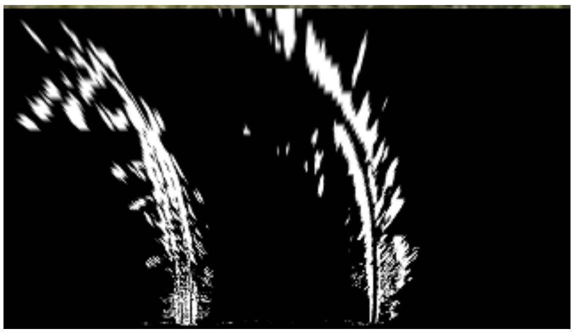
```
## perspective generates a Perspective transformation given f
#
#   f is a variable that tries to get the focal of the camera
#   h is the top point. Its points will be mapped to y = 0.0
#       Points over h will not be considered for fitting.
#   size is the size of the std image
#   shrink is a proportion of the image width to be used to
#   reduce width between lanes in pixels so curves fit in the window
#   If shrink != 0 then the meters for x pixels should be modified accordingly
#

def perspective(f=1.3245, h=460, size=(1280, 720), shrink=0.0, xmpp=0.004):
    l = size[0] * 0.2
    r = size[0] * 0.8
    l1 = l + size[0] * shrink
    r1 = r - size[0] * shrink
    b = size[1]
    src = np.float32([[l, b], [l + (b - h) * f, h], [r - (b - h) * f, h], [r, b]])
    dst = np.float32([[l1, b], [l1, 0.], [r1, 0.], [r1, b]])

    # Compute new xm_per_pix

    new_xmpp = xmpp / (r1 - l1) * (r - l)
    print(f)
    print(src)
    print(dst)
    M = cv2.getPerspectiveTransform(src, dst)
    ret, Minv = cv2.invert(M)

    return M, Minv, new_xmpp
```

Also as the conversion in the x direction from pixels to meters is modified accordingly and returned.



As seen in the warped image lane lines are parallel in the warped image but in the videos things are not easy as sometimes, probably due to differences in height in the road or inclination of the car, lane lines doesn't always transform to parallel lines.

For the previous filtered images we get:



Which are similar and quite parallel at the bottom but not so clear at the top.

4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

Before going to line recognition we must decide which of the filters to use. That is an interesting question and surely there must exist a way to select them before applying. I just compute all lines in every case and get the ones that seem better.

I use two methods, the sliding window method and an existing fit method for selecting points and fitting second order polynomials to the points.

The two methods build a **Measure** object with information from the sides. In the explanation of the algorithm it is explained what is a **Measure** object.

Sliding Window is as explained in the course. We get the two centers at the bottom by getting the maximum of an histogram of a convolution of an all ones window over the bottom ¼ of the image.

From here we look at next level centers around last level centers with the optimization that if no points are found in the window, the "movement" of the centroid from last one are carried on to next level.

It is implemented in the **sliding_window** and **find_window_centroids** functions. Points are packed as a **Fit** object that also computes the fit and the residuals and some data as radius. All in the class **Fit**.

World coefficients are computed from the warped image units as it is not necessary to do another fit (just checked the equations ☺).

Also did some work to check that the average of the residuals are really the sigma^2 of the points against the computed points.

In fact the **residuals** interpreted as the squared standard deviation of the "fit" are very important in the algorithm. It is explained in the algorithm section.

The **existing_fit** method is implemented in **known_lines_fit** function. It just looks for points a "margin" around the current fit with the exception that we use a **advanced** fit as explained in the algorithm section.

Fit lines are computed I **Fit.compute_fit(self)**:

```
def compute_fit(self):

    aux = np.polyfit(self.y_values, self.x_values, 2, full=True)
    self.coeficients = aux[0]
    if len(aux[1]) > 0:
        self.residuals = aux[1][0] / len(self.x_values)

    else:
        self.residuals = 500  # Must work, usually too few points
```

A difference with what has been taught is that my fit units are reversed in y. y = 0 is the bottom of the screen. That has some interesting properties for the fit ($Ay^2 + By + C$):

- C is the position at x = 0
- B is the "direction" of the lane at x = 0
- 2A is the "curvature"

The first property makes very easy to get the position of the car and the lanes and move one fit to check parallelism with another one. Just made C equal (or 0).

This is clear in the lane creation in the **sliding_windows_fit** and the **known_lines_fit** in:

```
left_lane = Lane_Measure.new_lane_measure_from_data(leftx, maxy-lefty,
     l_points, side='left',
     filter_x=filter_name,
     method='windows',
     xm=world.calibration.xm,
     ym=world.calibration.ym)
```

where lefty has been changed to maxy-lefty.

5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

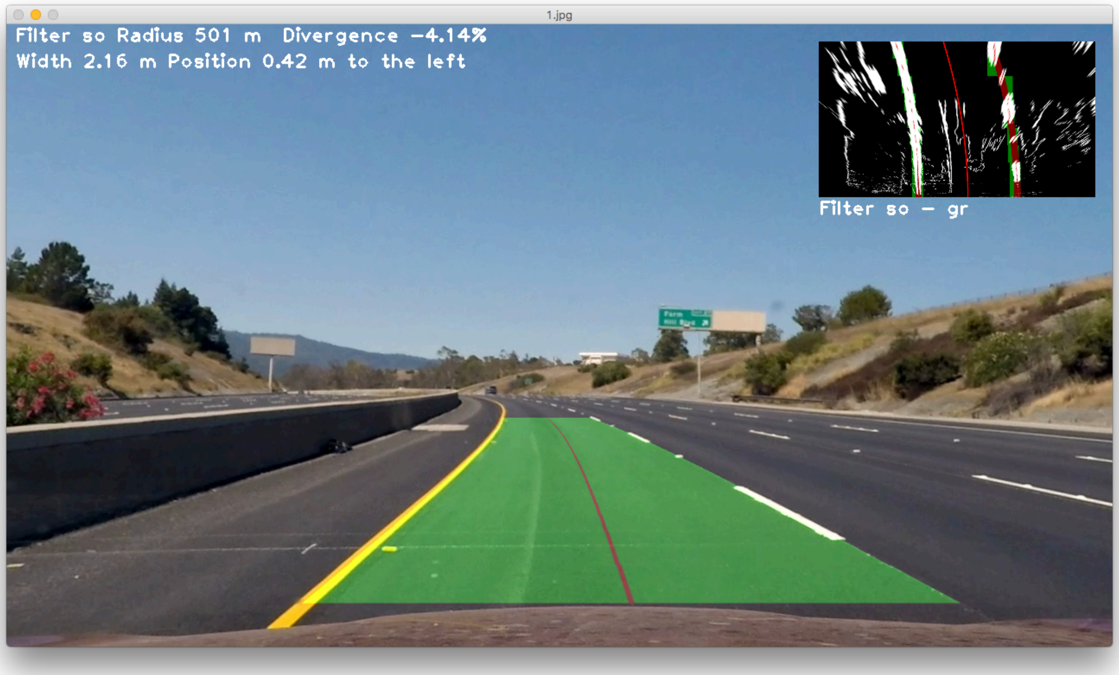
Radius computed in the **radius** and **world_radius** methods of the **Fit** class:

```
def world_radius(self, y):
    curverad = ((1 + (2 * self.world_coeficients[0] * y +
self.world_coeficients[1]) ** 2) ** 1.5) / np.absolute(
        2 * self.world_coeficients[0])
```

Position is computed in the **position** and the **positon_w** methods of the **Belief** class:

```
## Position returns the offset from the center of the lane
#  < 0 means I am at right, > 0 at left
#
def position(self):

    image_center = self.left_data.get_shape()[1] / 2.0
    lane_center = self.center_lane.get_x(0)

    lane_offset = lane_center - image_center  # < 0 means I am at right, >
0 at left

    return lane_offset

## position_w gives the same as position in world coordinates
#
def position_w(self):
    return self.position() * self.left_data.fit.xm
```

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.


1.jpg
Filter so Radius 501 m Divergence -4.14%
Width 2.16 m Position 0.42 m to the left
Filter so - gr

here we have an image where it is clearly marked the lanes, written some stats, specified which filters have been selected and an image of the warped image and the sliding windows and adjusted lines on the warped image. The width of the red lines in the warped (with exception of the center one) is the square root of the residuals (*2).

This code is in the **process_una_imatge** and calls some functions to get the small warped image and write the data (**plot_lines**, **build_edited_image).**

7 Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!)

Processed video may be found at https://youtu.be/HGnzDAI0g3o

Challenge processed video may be found at https://youtu.be/u_L7ynTDt8o

In the videos, in the small window with the warped image, the red line markers width at left and right show the error an there is also a blue line (not easy to see) that shows the last frame adjustment.

In the challenge video a second window shous the processed mask image.

In videos 2 frames have been processed and then the points added to provide some smoothing and solving the broken lines problem a little.

Filter names may change as the system chooses one or the other.

Radius is central line radius averaged over 5 frames.

Divergence is (Width of lane at top – width of lane at bottom)/width of lane at bottom

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?

Well, although it seemed easier, I have spent more time in this project than in any other. Specialy I devised a probabilistic method that I believe is quite interesting as it is easier to modify and update than the average ones.

So most problem have been in the area of image processing. How to eliminate shadows, what is really important in the image, how to made the algorithms more adaptive.

First video has only the "clear lane" problems and they can be solved by the "forget frame" as they are in quite stable sections.

Second video is difficult because the shadows in the first bridge. I am not very sure of my solution. It oscillates a little but doesn't crash against walls. Also I found the perspective a little different and works better if the area where we look ahead is a bit shorter. It has inspired the **remove_dark** function.

The last more challenging video presents different problems as vegetation so near the lines mix everything. Finally use of the hue was a solution. So the **color_filter** function was borne.

It is more bendy with shorter radius some parameters must be adjusted or the program rejects all fits. Also in many images lines are not parallel and I need to use a shorter lane view to be able to do something.

Really I am not satisfied by my results in the more challenge videos but have given me better insight and I have lost in them many many hours.

2. Scoring line and lane fits

As you know from the filtering chapter, I generate different filtered images and then select and combine the lanes. The big question is:

How I select the better lines?

The idea is give to each pair of lines (left, right) a score and get the ones with best score but.. Which score?

Essentially there is a **sanity check** (method **sanity_check** in the **world** class that marks a line as OK or bad line. Now it just looks at lane width and position but may check other things.

The there is the score process:

- Score each line (left and right)
- Score the pair (match? Are parallel?)
- Multiply all together

Then the **select_best** method of **world** class uses this score to select the best pair from all combinations from all filters.

Single line score is computed as the product of 2 values :

$$p_{sigma} = e^{-\frac{\sqrt{residuals}}{400}}$$

which represent a number between 0 and 1 representing the quality of the fit with respect to the lane width (400)

$$p_d = 0.2 * number\ of\ bins\ with\ more\ that\ 100\ points \text{ whic}$$

where we compute an histogram of y values and give 0.2 points to each bin with more than 100 values and are computed in the Lane_Measure class.

```
 ## p_dist returns a number between 0 and 1 where 0 means all bins of a 5
bin histogram are full
#   and 1 means are empty

def p_dist(self):

    ymax = self.get_shape()[0]
    histo = np.histogram(self.fit.y_values, bins=5, range=(0, ymax))

    p = 0.0
    for v in histo[0]:
        if v < 100:
            p += 0.2

    return p

## p_sigma is a value that if sigma = 0 its value is 1 and if sigma is
very big its value is 0
#   it is scaled by a 400 factor that is the "average width" of a lane

def p_sigma(self):

    return np.exp(-np.sqrt(self.sigma2)/400)

## p_score is a value with 0 means a perfect score and 1 means a horrible
one
#
#   It depends that my distribution of points is right and I have a low
sigma

def p_score(self):

    p_d = 1 - self.p_dist()
    p_s = self.p_sigma()

    #print("    Scoring Lane {} p_dist {:0.2f} p_sigma
{:0.2f}".format(self.filter, p_d, p_s))
    return p_d * p_s
```

Then for each pair of lines, left and right, I compute a p_measure that tries to evaluate if they are parallel and reasonable.

To do it I consider that at each y value there is a Normal Distribution with sigma^2 equal to the fit distribution. Have done some tests varying the value of the standard deviation with y but it doesn't seem so important. It is a possible way to better measure evaluation. It is done in the **Measure** class

```python
## p_ok is the result of some calculation thats gives it a POK
#
#   Now is the p entre els dos fits moved offset
#

def p_measure(self):

    if self.left_data.sigma2 is None:
        print("Left")

    if self.right_data.sigma2 is None:
        print("Left")


    #print(" sigma left {:0.2f} right
{:0.2f}".format(self.left_data.sigma2, self.right_data.sigma2))
    sig2 = np.power(math.sqrt(self.left_data.sigma2) +
math.sqrt(self.right_data.sigma2), 2)

    offset = self.right_data.get_x(0)-self.left_data.get_x(0)

    maxy = self.left_data.get_shape()[0]
    yvals = np.linspace(0, maxy - 1, maxy)
    xv1 = self.left_data.get_x(yvals) + offset
    xv2 = self.right_data.get_x(yvals)

    p = gaussian(xv2, xv1, sig2)

    pm = np.average(p)
    return pm
```

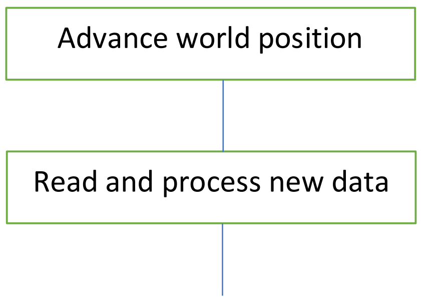
So now we have just to multiply the left line score, the right line score and the measure score to get a final one.

3.- Combining information from video frames and getting averages and a world estimate.

In the video we must use the temporal coherence to get better fits. I have used a simple, Kalman like algorithm but see many points to get it better.

All my information is in the **World** class in the form of the **working_belief**.

My algorithm follows the classic loop:



Update working belief

Advance world position

I have manually computed an estimates speed of about 24 pixels / frame. It is used to get a new fit supposing that the car has moved this distance. Also there is an error in speed that gets propagated through the fit increasing the **working_belief** sigma^2.

This error parameter is really used to tune the speed of response of the filter. If it is very low it doesn't hears to measures and the opposite.

It is implemented in the **Fit** class **move** method:

```
def move(self, d, sigma, maxy):

    new_fit = Fit()
    new_fit.xm = self.xm
    new_fit.ym = self.ym

    new_fit.coeficients = copy.copy(self.coeficients)
    new_fit.coeficients[1] = (2 * self.coeficients[0] * d) +
self.coeficients[1]

    # Sigma is added sigmas byt x sigma must be computed fromsigma in v
    derror = (2 * self.coeficients[0] * maxy + self.coeficients[1])*sigma

    new_fit.residuals = np.power(np.sqrt(self.residuals) + abs(derror), 2)

    new_fit.compute_world_coeficients()

    return new_fit
```

Read and process new data

That is what has been commented I most of the writeup. Essentially get a frame, process it with the filters, call the line recognition and select best fits.

It is implemented in the **process** function and the two fit functions discussed.

Update working belief

Ok, now we got new measures and must update our belief. First we select the best fits as explained and then combine it in class **world** method **update** using the product of Normal Distributions:

```
def update(self, measures):

    # we select best measure

    best, evaluation = self.select_best(measures)

    if best is None : # No good measure
        self.skipped += 1

        if self.skipped >= self.max_skipped:    # Things are really bad.
```

```
Perhaps my world is all messed about.
            self.reset()

        return

    # First case, we know nothing about the world
    # Try the best measure we have and let it a go.
    #
    # Probably we need some aditional sanity check just in case

    if self.working_belief is None:

        self.working_belief = Belief()
        self.working_belief.frame = self.frame
        self.working_belief.left_data = best.left_data
        self.working_belief.right_data = best.right_data
        self.working_belief.warped_image = best.warped_image
        self.working_belief.compute_data()
        self.working_belief.compute_center_lane()
        self.working_belief.measures = [best]
        self.compute_lane_width()
        return self.working_belief


    # All seems correct, update our position in the world

    self.working_belief.compose(best, evaluation)
    self.skipped = 0
    self.history[:0] = [self.working_belief]

    # Now we must compute new width lane

    self.compute_lane_width()

    p = self.working_belief.p_measure()

    return self.working_belief
```

The big problem is when to skip a frame and when to detect our working_belief is so messed up it is useless.

For the moment we have 2 parameters in **world**, **max_skipped** that triggers a **world.reset** and **use_windows** that forces an sliding windows fit, and the advance function is only used in the first skipped frames. From then on it doesn't updates the data as errors tend to increase too much.

Sanity Check clearly needs to be tuned but at least I pass videos 1 and 2.

In the beginning of **challenge_video** things are difficult by the light changes in the tunnel. The method explained heps to solve it by having two filters with different parameters. The system automatically selects one for left lines and the other for right ones.

Skipped frames and recovery

That is an area that also merits some more study. Not only the limits in the sanity check are important but also when to trigger sliding windows search or when to reset all the world data. Not much work done over this.

Program Structure and Classes

Most of the Image Processing functions are just this, functions.

process_image is the main one. Applies corrections, calls the filters and applies perspective.

super_filter applies the different filters.

Sliding_windows_fit is the sliding windows algorithm

Known_lines_fit is the known line algorithm

Plot_lines and **build_edited_image** do that, build the edited image

Process_an_image is used to process just an image

Process_folder processes all the images in a folder

Process_video processes a video

All data is stored in some classes from bottom to top

Fit represents a line fit with all its data. Essentially mathematics.

Lane_Measure is an envelope over the **Fit** with more specific data and functions and more meaning.

Measure are a pair of lines. It is used as the superclass of the **Belief**. It has 2 **Lane_Measures**, one for each side and some logic for analysis.

Belief represents the knowledge in an instant. It computes the center line and knows how to update itself or advance in the future.

World is the all encompassing class. Is what I know and the logic for updating and checking lanes.

Calibration is just an auxiliary class for storing camera calibration data, perspective transformation matrix and scale factors.

4.- Things to better

There are many things to get a better system. Some of them may be :

- Getting better estimations of errors and probabilities
- Better sanity checks. Perhaps more complex
- Better filters
- Made the system flexible with bend curves, etc.
- Obtain speed from the image

Well, and probably many many things