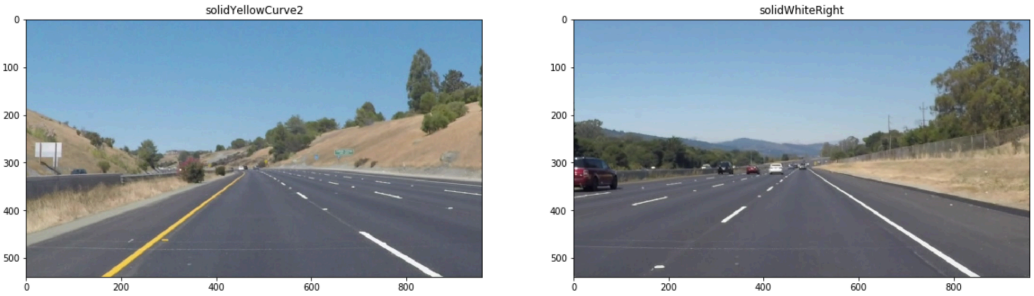
Lane Detection

Goal:

Process a video captured by a windshield mounted camera to materialize the driving lane ahead of a vehicle in motion. This lane has to be anchored on the existing roadway line markings.

These markings are combinations yellow or white and solid, broken, double lines as showed below:

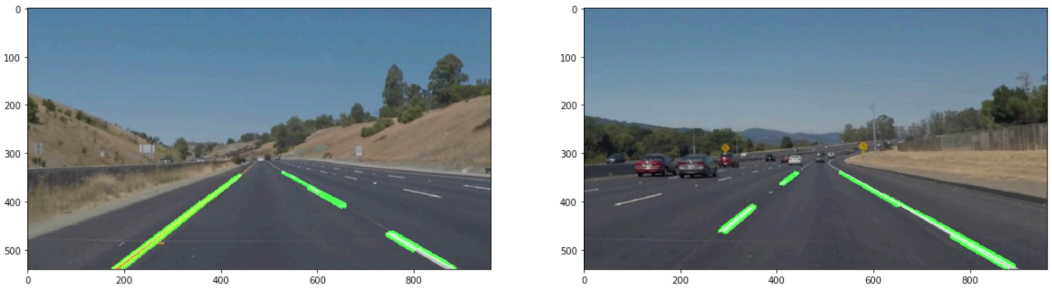


yellow solid, broken white line, solid white line

Lane detection general pipeline :

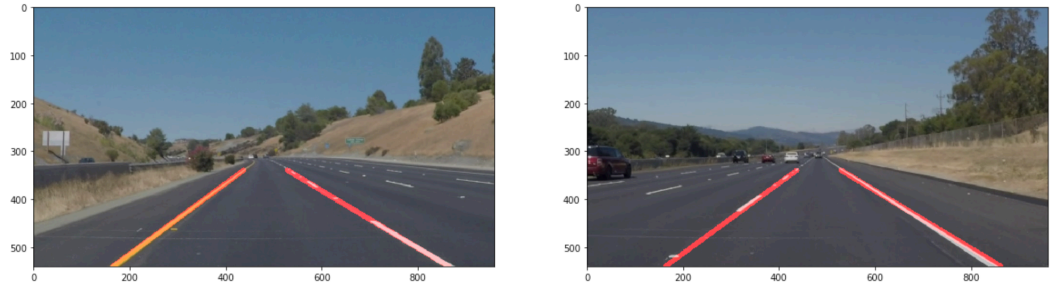
Lane detection can be broken in 3 main steps :

- line markings detection on a still frame:



line markings are detected and highlighted

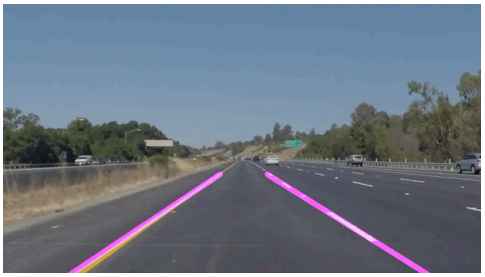
- static **lane** detection :



extension of detected markings on a still picture to form a lane

Philippe Rougier

• dynamic lane detection:



markings are detected and displayed and animated to the vehicle drive
(click on picture for video)

I. Static line marking detection :

Goal:

Detect line markings on the roadway.

Line markings detection Pipeline :

(See appendix Road markings detection pipeline)

Each step of the pipeline only retains meaningful features and passed to the next step. The final process outputs the detected lines, all other visual elements have been filtered out. These detected lines are then either :

- combined to the original image for visualization and validation of the process.
- sent to the next major process for lane detection.

1. contrast addition:

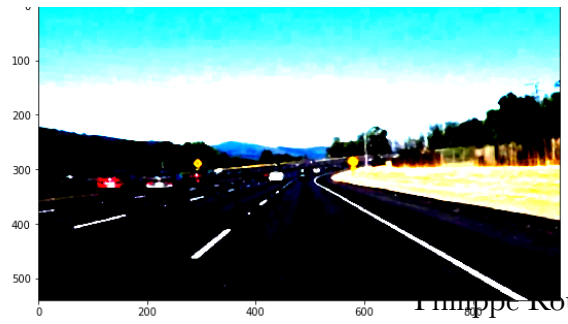
Problem:

Some lines are undetectable in faded regions of the picture. Some patterns are just noise and shouldn't be interpreted as legitimate line markings.

Solution:

A **blurring filter** adequately chosen and calibrated will reduce the noise to an acceptable level . Then a filter will selectively enhance the luminosity and overall contrast of the image.This combination reveals potential features faded in obscure area while removing specks and scratches.

Results:



The above original left picture shows the horizontal scratched lines on the camera lens (right above the red line) as well as the thin line next to the line markings (see blue line) These thin lines will have competed with real one later in the pipeline and were remove by filtering as seen on the right side picture. The added contrast does not improve the detection per se but visually helps for fine tuning the parameters of the pipeline.

implementation :

- a flattening *cv2.median* blur (5 pixel) to muffle all noises
- an image brightening and contrast enhancer function built on *cv2.addWeighted.* to visually enhance the picture while tuning of the noise reduction parameters *(credits: https://stackoverflow.com/users/9705687/bfris)*

2. White and Yellow filtering:

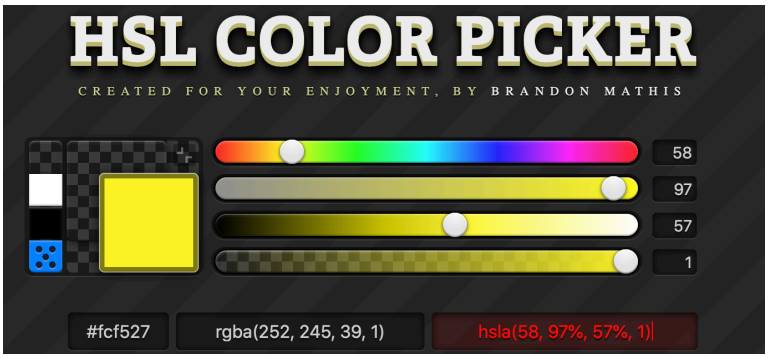
problem:

Only white or yellow lines should be detected, all other color lines ignored.

Solution:

Yellow and white filters are applied simultaneously so only yellow or white pixels from the original picture remain.
Although using RGB based filters is possible it appears difficult to narrow down on RBG ranges of desired colors. On the contrary identifying such ranges is a lot more intuitive using **HSL** (Hue,Saturation,Luminosity) color mapping.
Continuous HSL spans can be defined between high and low HSL values.

Picking a first ball park set of HSL ranges values for the filters can be done with an HSL color picker :(try HSL color picker (courtesy of Brandon Mathis))



the HSL values for Yellow as seen here are 58°, 97%,57%

implementation :

HSL Filtering is at three step process:

- converting from RGB to HSL of copy of the original image
- creating of 2 filters (bitmap) (yellow, white)
- applying these combined binary filters on the original image RGB image

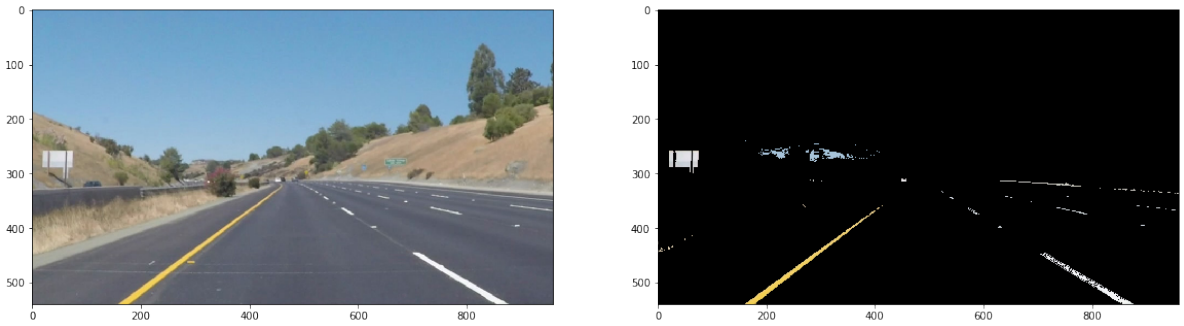
Trial and error led us converge on the following HSL spans:

lower_yellow= HSL_color_picker(40°,50%,30%)

higher_yellow=HSL_color_picker(70°,100%,83%)
lower_white=HSL_color_picker(0,0,75%)
higher_white=HSL_color_picker(360°,45%,100%)

-

Results :



white and yellow *filtering* example :

Narrowing down on the right settings eliminated the yellow grass borders but still retained the yellow markings.
On the contrary athough white lines were successfully picked up the irrelevant

3. Detect edges :

problem :

Edges are dramatic color variations from one area to its immediate surroundings (ie high gradient). At this step of the pipeline only edges (of any shapes) should be detected.

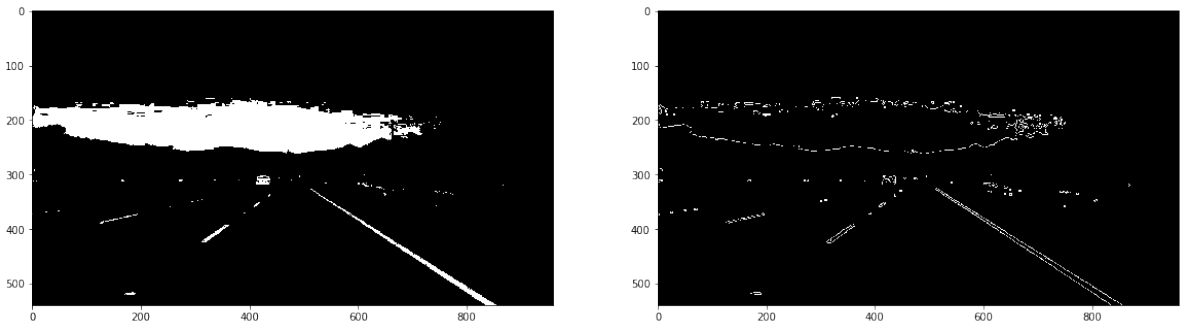
Solution:

The Canny Edge Detection algorithm detects high gradient points within a picture . It outputs a corresponding binary map :
- 1 for high* gradient pixels (* i.e above a chosen threshold)
- 0 for others

implementation :

Prerequisite : Canny runs on a 8 bit bitmap, so images need to be converted to a 256 shades of gray (8 bit)
cv2.Canny description and parameters list can be found here, parameters were tuned at:
cv2.Canny(img,continuous threshold= 50, primary threshold=100)

***Results** :*



canny edge detection:

All remaining edges are detected and outlined . So is the irrelevant patch of sky. The next step will fix this by eliminating edges out of a specific field of interest.

4. center region of the picture (field of interest) :

Problem :

Detection should only occur with the center lower half image corresponding approximately the usual position of a driving lane. All edges outside of this trapezius shaped region should be ignored**.**

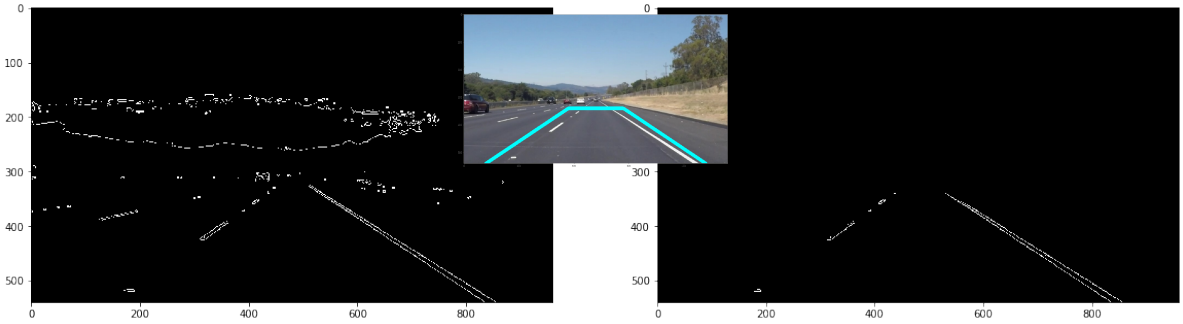
Solution:

All pixels representing edges located outside of a bottom centered trapezoidal region of the last binary picture are deemed irrelevant and therefore zeroed out.

implementation :

The sides (vertices) of this polygon and the edges binary picture are passed to a cv2 function (cv2.fillPoly). Like a cooky cutter, this function only outputs the edges within its polygonal shape

Results *:*



field of interest:

Irrelevant edges have been eliminated: vegetation, sky patch, other lanes markings .

5. lines detection:

Problem :

Only edges forming straight lines should be detected, highlighted and match the existing roadway lines

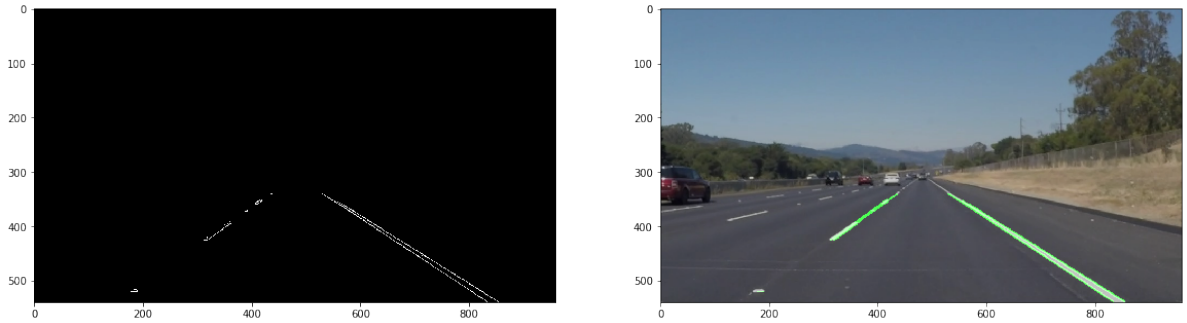
Solution:

The previous binary bitmap image representing the edges is passed to a hough transform function handling the line detection. In a nutshell this function detects all all points corresponding to a segment of line. Then it outputs the extremities (coordinates) of these detected segments.

implementation :

cv2.HoughLines *description and parameters list can be found here, parameters were tuned at: hough_lines(img, rho=1, theta=np.pi/180, threshold=12, min_line_len=10, max_line_gap=50)*

Results:



lines detection:

lines are detected and superimposed on the original for parameters fine tuning

II. Static lane detection :

Goal :

*Draw a continuous roadway **lane** based on detected line segments.*

Problems and solutions

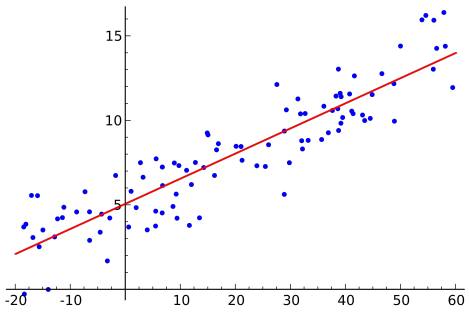
1. Best line approximation :

Problem:

Find the best approximation (straight line) passing through en ensemble of given points

Solution :

Linear regression.



Linear regression

Linear regression calculates **the best fit (line)** running through an ensemble of points. It outputs the line that minimizes the sum of euclidian distances to all points.

Implementation:

We 'll use the built in numpy linear regression (*polyfit(X,Y,1)* which returns both the **slope** and the **intercept** of the best fit. We'll use these values to draw each side of the lane running from the bottom of the screen to the top of the field of interest.

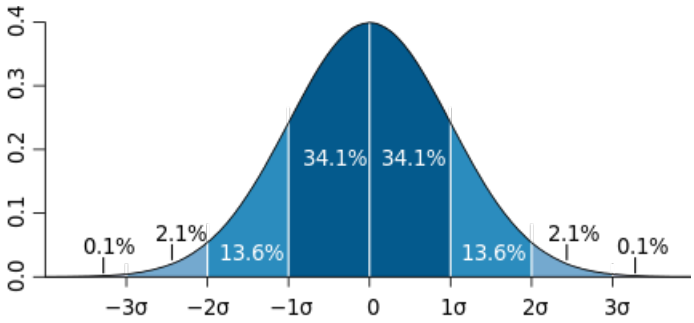
2. Filtering out outliers :

Problem:

Back to back Canny Edge and Hough Transform return multiple segments (even for lines appearing straight and continuous). Before finding the best fit through linear regression we'll vet the list of segments to filter out the outliers .

Solution:

Segments slopes on each side lines appear normally distributed.



normal distribution and standard deviation

For each side we applied two consecutive standard variation filter:

- first on the slope
- then the intercept.

We filter out segments outside of a mean centered **2** standard deviations **span** to**:**

- eliminate the obvious outliers (failing detection)
- eliminate the segments still in the bell curve but not close enough to the center. This to improve the quality of the sample we'll use to calculate the linear regression later.

Implementation:

np.mean, np.std are numpy built-in mean and standard deviation functions

results :

Once the list of segments vetted using 2 consecutive standard variation filterings , we 'll pass it to linear regression calculate the slopes and intercepts each side line. We then use these values to draw the lane running from the bottom of the screen to the top of the field of interest.

Although detection happens flawlessly on still pictures (aka frames), scaling up to a feed of frames (video) will present other challenges.

III. Dynamic lane detection:

Goal :

Draw a continuous roadway animated lane based on a succession of detected line segments .

Problems:

Extra developments are needed to transition from static lane detection on a picture to a succession of 25 frame/second dynamic detection. The specific challenges are:

- slopes and intercept fluctuates slightly from frame to frame as the vehicle progresses. These slight **oscillations** will have to be stabilized to improve the quality of the animation
- occasionally on a single frame on one or both sides no line is detected. This occasional **flickering** should also be doctored.

Solution:
(on each side i.e right, left)
We use a queue (“deque”) to create a shock absorber mechanism to deal with the erratic fluctuations in the detection flow:

- we store last top10 consecutive images final slopes and intercept (on each side right and left) in a queue (FIFO).
- every new frame F related values (zeros for no line detected) are added at one end the queues, as the values for the oldest frame (F-10) is popped out at the other side of the queue
- We average the values through the length of the queue (but consider that the zeros are not part of the length) and use these current averages to draw the lane.

Evaluation:

- stabilizes the **oscillations** (a 10 image length buffer is chosen knowing that the video speed is 25 image/second : **no perceived lag**)
- stabilization doesn’t affect the quality of the detection : sideways movements of the vehicle are still accounted for. As the vehicle shots sideways the shifted lane morph into the previous frame lane.
- addresses the flickering due to ”empty” frames: slopes and intercepts mimic the pattern of the last half second
- lane rightfully disappears after 10 consecutive empty frames (no lane are on on the the roadway)
- lane reappears instantaneously as soon as detected (no lag , no need to wait for the buffer to fill up)

Results:

1. Yellow lane, no obvious specific challenge:

video: lane detection without buffering

Erratic lines flashing lines across the screen appear when the majority of the line segments detected on a single frame are erratic (showing the limitations of the STD filtering) .

video: lane detection with buffering

Erratic side lines do not get displayed anymore but still contribute to the average buffer. Therefore the succession or erratic frames induces slight oscillations altering the quality of the lane display.

Solution:

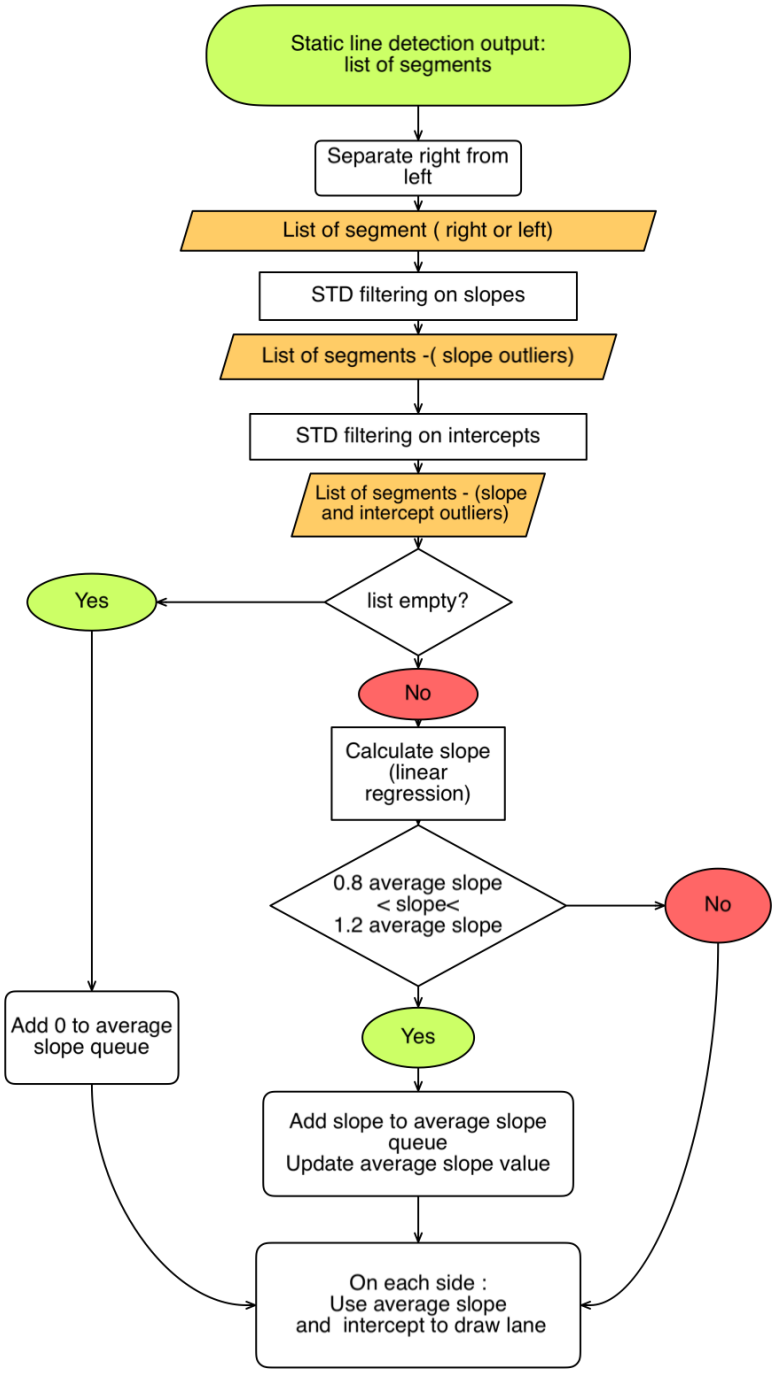
For each frame we compare the slope (right and left) to the corresponding average calculated with the FIFO values :

- we discard frames slopes off by more than 20% than the average .
- **we do not append this erratic slope into the queue**
- for these discarded frames we instead display the lane based on the average slope

In other word we deal with erratic frames like with the ones without any line but we don't add them to the queue.

Static and dynamic lane detection pipeline is summary

Static and dynamic lane detection pipeline:



Philippe Rougier

Results:

video: Detection with back to back STD slope and intercept filtering (click on picture for video)

The oscillations are no longer happening since erratic frames are not encounter for into the queue.

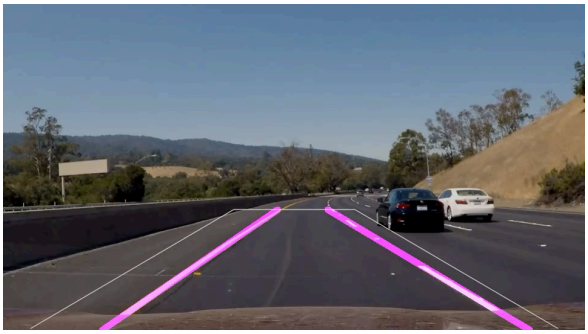
Finally we enjoy a smooth but still accurate ride 🚘!

2. Challenging roadway and lighting:

Shades, cracks on the roadway, irregular road markings are all detected. As they get mixed with real road markings, stable lane detection is no longer possible as showed bellow .See below all the lines segments detected before any frame by frame STD filtering, erratic frame filtering and buffering:

video: line segments detected without erratic frame filtering and buffering (click on picture for video)

Notice the clusters of erratic lines detected due to the quality of the roadway and the



Lane detection with back to back STD slope and intercept filtering and erratic frame management (click on picture for

As seen above back to back standard deviation filtering and average smoothing and erratic frames filtering improve the lane detection dramatically.

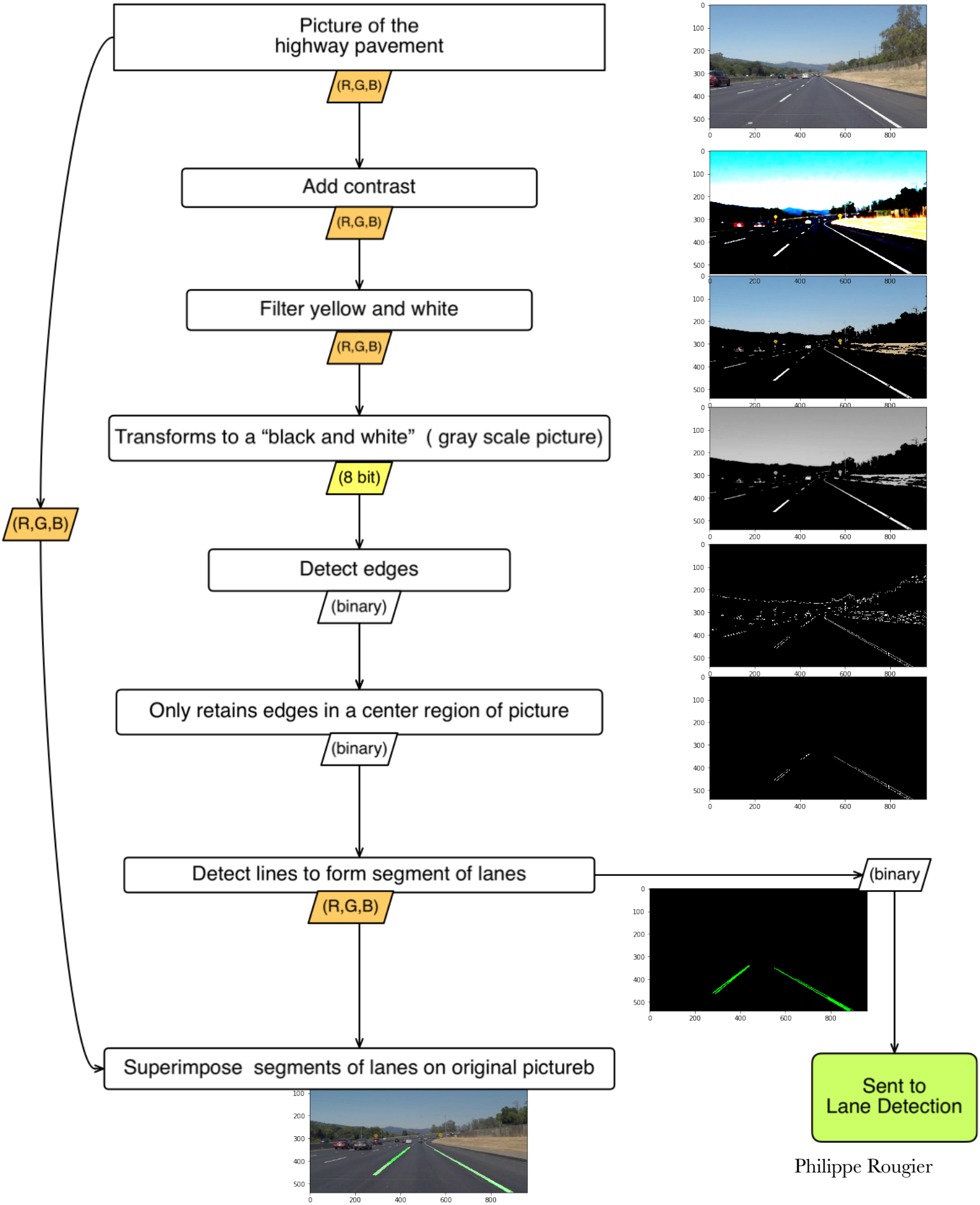
As an alternative, erratic frames could also be processed like the empty side line frames and added to the averaging queues as zero values.

Although the lane is now stable, the lines are loosely following the curve. We will address this out of this project scope challenge in a future development .

Please see appendix 1 next page

Appendix 1

Static Line Markings detection pipeline :



Philippe Rougier