Lane Detection in Automotive

Contents
Introduction .................................................................................................................................................. 2
Image Processing ......................................................................................................................................... 2
Reading an image ....................................................................................................................................... 3
RGB to Gray ................................................................................................................................................ 3
Mean and Gaussian filtering .................................................................................................................... 6
Defining our Region of Interest ................................................................................................................ 10
BirdsEyeView Transformation .................................................................................................................. 11
Horizontal Sobel ........................................................................................................................................ 12
Binarization (OTSU or other) .................................................................................................................... 15
Selecting relevant points ........................................................................................................................... 16
Polynomial Regression .............................................................................................................................. 17
Kalman Filtering (optional) ....................................................................................................................... 18
Drawing Lanes (optional) .......................................................................................................................... 20
Perspective transformation (Next Year Maybe) ...................................................................................... 20
Introduction

Before we begin discussing about Driving Functions and mathematical models of the vehicle, we must first discuss about sensing the environment around the vehicle. Lane Detection is one of the many components that try to offer realistic information about the surrounding world.

![Figure 1 Lane Detection Example](image-url)

The full chain of effects regarding Lane Detection falls inside the area of Digital Image Processing.

Image Processing

“In computer science, digital image processing is the use of computer algorithms to perform image processing on digital images.” – Wikipedia

When we talk about image processing we refer to all the algorithms, mathematical functions and techniques used to obtain or classify information from images in the form of two dimensional matrices. It can be considered a type of digital signal processing.

“Artificial intelligence (AI), sometimes called machine intelligence, is intelligence demonstrated by machines, in contrast to the natural intelligence displayed by humans and other animals. In computer science AI research is defined as the study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals. Colloquially, the term "artificial intelligence" is applied when a machine mimics "cognitive" functions that humans associate with other human minds, such as "learning" and "problem solving".” – Wikipedia

Practically, AI and Machine Learning encompasses algorithms that can make some predictions based on a set of known data. Object Detection is mainly based on Machine Learning and AI concepts. The Lane Detector we’ll be working with doesn’t use any AI techniques, but Neural Network techniques are being used for more modern Lane Detectors.

As you may imagine, developing a library with all the fundamental mathematical methods for Image Processing is relatively complicated. To avoid this issue altogether, we’ll be using a library called OpenCV.

Arnautu Stefan, Continental, C&S ADAS
OpenCV (Open Source Computer Vision Library) is an opensource image processing library for C/C++, Python and Java.

Reading an image

There are multiple ways of working with images. You can simply read one image at a time (with a certain format – JPEG, PNG, BMP etc.), you can read a video (with various formats – avi, mpeg etc.), or you can have access to a video camera and get each image in real time. The individual images received from video cameras are referred to as image frames.

A black and white image is, in its all simplicity, just a two dimensional matrix with values. Those values usually vary from 0 to 255, meaning the image is an 8bit image (there are other images that have data with a higher resolution, like 10bits or 16bits). Getting access to that matrix however, is not as straightforward as it may seem. We would need what is called a decoder. Obviously, the decoder is needed because the pixel matrix is encoded in a certain way. This is where all the formats come from, JPEG, BMP, PNG and many others. To capture frames you would need a driver for the specific camera, in order to interpret the data sent by the video sensor.

Fortunately, OpenCV already has those decoders (and encoders) implemented and has access to specific drivers in case you ever use a camera.

Opening a sequence of pictures:

```cpp
cv::VideoCapture sequence(data_frames); // open image sequence
if(!sequence.isOpened()) // check if succeeded
{
    std::cout << "file " << data_frames << " not found or could not be opened" << std::endl;
    return 0;
}

cv::Mat img_ld;
sequence >> img_ld; //read first frame to know the width and height
```

Opening one single image:

```cpp
cv::Mat image;
const char* image_file = "/home/uidg5179/Work/Kitti/2011_09_26 drive 0015 extract/image = cv::imread(image_file, CV_LOAD_IMAGE_COLOR);

cv::imshow("Display Image", image);
```

RGB to Gray

Like we said before, a black and white image is simply a two dimensional matrix with values from 0 to 255, signifying the “gray level”. But what is a color image? Well, it’s three black and white pictures put together. Each “gray” image represents the amount of Green, Red or Blue of the full color image. We refer to these images as the “3 channels” of the color image.

These three channels are combined in a certain way for our eyes to perceive the original color picture. If we would want to transform a color image into a grayscale image, we would need to know how the color image itself is formed.

Arnautu Stefan, Continental, C&S ADAS
A straightforward way of doing this is applying the average of the three channels.

\[
Y = \frac{R}{3} + \frac{G}{3} + \frac{B}{3}
\]

If we would implement this equation we would notice our “grayscale” image doesn’t look quite right.

Experimentally, we’ve noticed that our eyes perceive colors in with different levels. We can describe this mathematically as a weighted average. The weights have been found empirically.

\[
Y = 0.2126 \times R + 0.7152 \times G + 0.0722 \times B
\]

The OpenCV function looks like this:

```c++
//RGB to Gray conversion
cv::cvtColor(OriginalImg, OriginalImg, CV_BGR2GRAY);
```
We will implement our own function that transforms a color image to a grayscale image. In the DSP project, inside the LaneDetection header file we can find the prototype of the function:

```cpp
/** 
* rgb2gray transforms an RGB color image to grayscale.
* This function takes a color image and computes for each color pixel the grayscale equivalent.
* \param [in] img1 is the input color image.
* \param [out] img2 is the output grayscale image.
*/
void rgb2gray(cv::Mat img1, cv::Mat img2); //Exercise 1 RGB2GRAY
```

The input of the function (color image) will be `img1` and the output (grayscale image) will be `img2`. They both are of type `cv::Mat`, an OpenCV matrix object.

The implementation will be done inside the function in `LaneDetection.c` source file:

```cpp
void LD::rgb2gray(cv::Mat img1, cv::Mat img2){
    //write code here
}
```

To access the pixels of the color image (red, green and blue channels) we will use the OpenCV method below. Note the `cv::Vec3b` type – it is a

```
img1.at<cv::Vec3b>(i,j)[0]
```

The array of length 3 represents the 3 channels at the point `(i,j)` – *(rows, columns)*.

- `[0]` – Blue Channel
- `[1]` – Green Channel
- `[2]` – Red Channel

The grayscale image pixel is accessed in a similar fashion. Note the `uchar` type. We don’t need an array to access the pixel since there’s only one channel – the gray one.

```
img2.at<uchar>(i,j)
```

Putting all of this together, if we want to change one pixel value of `img2` with the value of the green channel of the color image, we can do something like this:

```
img2.at<uchar>(i,j) = img1.at<cv::Vec3b>(i,j)[1];
```

Obviously, if we want to change the entire matrix, we would need to use two for loops to go through the entire two dimensional array.
To test your freshly implemented function, you’ll have to go to LD::LaneDetection(cv::Mat OriginalImg) function inside the LaneDetection.c source file (line 498) and make sure the lines look like this:

```cpp
//RGB to Gray conversion
cv::Mat grayImage(OriginalImg.rows, OriginalImg.cols, CV_8UC1);
//cv::cvtColor(OriginalImg, grayImage, CV_BGR2GRAY);
//Exercise 1 RGB2GRAY
LD::rgb2gray(OriginalImg, grayImage);

#if LD_SHOW_GRAYSCALE
    cv::imshow("Grayscale", grayImage);
#endif
```

Make sure the LD_SHOW_GRAYSCALE define is set to 1 in the LaneDetection.h header file! Otherwise the application won’t show you the result.

Exercise 1: Your task will be to change the pixel values of the entire img2 with the weighted average (the weights described higher up) of the 3 color channels. Implement RGB to Gray function. (Normal AND weighted average)

Mean and Gaussian filtering

In Digital Signal Processing theory, ideal signals don’t have noise. They look like perfect sinuses. In the real world, signals are generally noisy. We don’t like noise. Noise bad. There are multiple types of noise, but the most common one can be removed using a “mean filter” or a more complex “gaussian filter”. An image can be seen as a 2D signal.

The mean filter basically takes each pixel of an image and replaces that pixel with the arithmetic mean of all the pixel values inside the window you chose. For example, if the window size is 3x3, the middle pixel value is replaced with the average of all the 9 pixel values inside that window. The window is also called a “kernel”.

Arnautu Stefan, Continental, C&S ADAS
A gaussian filter is very similar to the mean filter, just that the weights inside the window follow a gaussian function.

Normalized Gaussian curves with expected value $\mu$ and variance $\sigma^2$. The corresponding parameters are $\alpha = \frac{1}{\sigma\sqrt{2\pi}}$, $b = \mu$ and $c = \sigma$.
OpenCV filtering:

```cpp
// Eliminate noise
cv::GaussianBlur(img, img, cv::Size(5, 5), 0, 0);
```

The mean filter is similar to the `GaussianBlur` function but it is a lot simpler. Our mean filter will have a window of 3x3 (if you can make the window size dependent on a function parameter you get extra internet points).

In the LaneDetection header you will find the `meanFilter` function prototype:

```cpp
/** \brief meanFilter applies a 3x3 mean filter on the given image.
 * This function filters a given grayscale image with a 3x3 mean filter.
 * \param [in] img1 is the input grayscale image.
 * \param [out] img2 is the output grayscale filtered image.
 */
void meanFilter(cv::Mat img1, cv::Mat img2); // Exercise 2 MEANFILTER
```

In this case, `img1` is the input grayscale image (transformed earlier by the `rgb2gray` function) and `img2` is the output image, also grayscale, but blurred by the filter.

The `meanFilter` (or any kind of blur filter) should diminish the noise of the image sensor. The side effect is smoothing out the edges of the image (blurring it).
Example:

You can see that the 3x3 filter doesn't seem to have much of an effect on the image. If the image itself would have more noise however, the filter would remove a part of it and also not remove the edges that much. The 3x3, 5x5 filters are usually a good compromise. Higher window sizes are generally used in photo editing software for the blurring effect.

The implementation of the function is fairly straightforward. Using what we learned in the rgb2gray function (how to access gray pixels and how to modify an entire image matrix), we will create our own mean filter.

Like in the previous exercise, in the LD::LaneDetection function, make sure you call the meanFilter function:
Write your code inside the function in the LaneDetection.c source file:

```c
void LD::meanFilter(cv::Mat img1, cv::Mat img2){
    //write code here
}
```

You will basically need to have two for loops (as in the rgb2gray function) to go through the entire image, but here, img2 will receive the value of the average of the nine pixels in the img1 matrix. **You need to make sure the for loops don’t go outside the boundaries of the image itself.**

```
img2(i,j) = \( \text{img1}(i-1,j-1) + \text{img1}(i-1,j) + \text{img1}(i-1,j+1) + \text{img1}(i,j-1) + \text{img1}(i,j) + \text{img1}(i,j+1) + \ldots \)
```

To see the output of the function make sure the `LD_SHOW_BIRDSEYEVIEW` and `LD_SHOW_GAUSSFILTER` defines in the LaneDetection.h header file are set to 1.

Exercise 1: Implement an average filter (3x3 window size).

Completely optional:

Homework 1: Implement an average (or gaussian) filter function with window size as a parameter.

Homework 2: Implement a gaussian filter (3x3 window size).

**Defining our Region of Interest**

In order to make our job easier, we would like to lower our search window. In technical terms, this means choosing our ROI (Region of Interest). For this specific application, the first thing we would do, is limit our search only to the lower half of the image, since lane markers don’t (usually) appear on the blue sky. We can go even further and select something like a trapezoid, since we know that the lane markers can be found 90% of the time inside that area (see Figure 8).
The way the frames present themselves at this point still isn’t ideal for us. We could apply an edge detector (explained in the next part) and see how things go from there, but it would be really nice if the lane markers were more... vertical.

If we could look at the street from above the lane markers would appear parallel (and on the image sensor they would appear vertical). This is exactly what transforming to Birds Eye View is. Mathematically, it’s a perspective transformation and it is the subject of Linear Algebra.

Ideally, the transformation should be done automatically, knowing the position and orientation of the camera relative to the road. We don’t have that at the moment, but we have a trick. We know the lane markers should be parallel lines. So if we can select two pairs of two points (4 in total) and somehow figure out the math to transform the image such that those 4 points will define two parallel lines, we’re set!
OpenCV comes to our rescue again with the following function:

```c
generic_DSP.M = getPerspectiveTransform(src_vertices, dst_vertices); //Get birds eye view transform
```

- `src_vertices` represents the four points in the original image
- `dst_vertices` represents the four points in the BirdsEyeView image
- `M` is the matrix transformation obtained

**Figure 12 Original Grayscale Image with ROI Mask**

**Figure 13 Birds Eye View of ROI**

**Horizontal Sobel**

In this part, we’ll go into what “edge detectors” are. The simplest one would be the “Sobel Edge Detector”. This edge detector is based on a “kernel”, similar to the mean/gaussian filter.

The kernel window for Sobel is this:
The implementation is almost exactly the same as the implementation of the mean filter, just that the weights are different. The kernel above is useful only for detecting \textbf{vertical edges}. If we want to detect edges oriented at a different angle, we would need to rotate the kernel too. To test this, you can use an image with vertical edges only and apply the horizontal kernel and the vertical kernel and compare the images.

In LaneDetection, we only need to use the horizontal kernel, for obvious reasons.

\[\begin{matrix}
+1 & 0 & -1 \\
+2 & 0 & -2 \\
+1 & 0 & -1 \\
\end{matrix}\]

\textit{Figure 14 Sobel Matrix - From Wikipedia}

The function prototype of our horizontal Sobel is in LaneDetection.h header file:

```c
void Horizontal_Sobel(cv::Mat img, cv::Mat &img1, cv::Mat &img2); //Exercise 3 SOBEL EDGE DETECTOR
```

Arnautu Stefan, Continental, C&S ADAS
The function takes as input a grayscale image and outputs two images containing LOW-HIGH edges and HIGH-LOW edges. Let’s see what the difference between a LOW-HIGH edge and a HIGH-LOW edge is.

If you have this kind of image as an input:

![Image 1](image1.png)

The output of the sobel edge detector should be something like this:

![Image 2](image2.png)

The first white line represents the **LOW-HIGH edge** (black to white gradient) and the second white line represents the **HIGH-LOW edge** (white to black transition). The two white lines are not exactly identical. The difference between them is their sign! You won’t see that kind of image in your implementation (and you shouldn’t because you don’t need it). The reason is that you can’t really show negative values. In the code, you’ll see that after applying the OpenCV Sobel detector we’re changing all the values in the image with their absolute values. **This is done only to be able to visualize the image.**

```c
//Sobel test
int dist_x = 1; //try switching x and y to detect horizontal edges instead of vertical edges
int dist_y = 0;
int window_size = 3; //should be 3,5 or 7
cv::Mat img_sobel_test;
cv::Sobel(img, img_sobel_test, CV_16S, dist_x, dist_y, window_size, 1, 0, cv::BORDER_DEFAULT);
cv::imshow("sobel",img_sobel_test);
```

Our own implementation will be done in the LaneDetection.c source file:
The code itself should be almost identical to the meanFilter function, except that the weights of the pixel will differ, based on the Sobel kernel shown above.

```c
int sum1 = (pixel1 * 1+ pixel2 * 2+ pixel3 * 1+
    pixel4 * 0+ pixel5 * 0+ pixel6 * 0+
    pixel7 *-1+ pixel8 *-2+ pixel9 *-1 )/8; //normalize
```

The reason we are dividing by 8 at the end is to keep the value of the pixel inside “normal” boundaries (so it won’t overflow or underflow – **amplify** or **attenuate** the signal). This is called “normalization” (duh). Remember, we did the same thing in the meanFilter function, but we did it instinctively! We divided by 9 at the end. The simple answer as to why is that it’s an average filter (9 pixel values, divide by 9), but if you look at the **weights** you will see that they are all 1. We have 9 pixels so the sum should be divided by 9.

In the sobel operator, the sum of the absolute weights is 8, so the sum of the pixels should be divided by 8. (1+2+1 + 0+0+0 +1+2+1 = 8)

If you increase the size of the window (hint for the meanFilter with variable window size), you must take this into account as well. You will have to divide to a different number, to normalize the image. (5x5 window size -> 25 pixels -> weights are all equal to 1 -> divide by 25)

**Very important:** At the end of the operation, we have both positive and negative values. We’ll have to copy the absolute value of the negative values into img1, and the positive values into img2. This way, we’ll split LOW-HIGH edges (img1) and HIGH-LOW edges (img2).

**Exercise 1:** Horizontal Sobel implementation.

**Homework 1:** Full sobel implementation (comparison with horizontal only).

**Binarization (OTSU or other)**

After obtaining the Sobel Image, we would like to filter out the edges that are not very sharp and only leave the edges of the lane markers. It would also be nice if those edges would be white (value 255) and the background to be black (value 0). This process is called binarization and we’ll obtain a binary image (only two values exist, 0 and 255).
There are multiple ways of creating a binary image. The idea revolves around selecting a threshold in the image and transforming all the pixels that have a value lower than the selected threshold to 0 and the pixel above that threshold to 255. Not all images have the same optimal threshold however and selecting it automatically falls into the category of clustering methods.

The most known method for binarization is called “Otsu’s method”. OpenCV has this too. You go OpenCV!

**Selecting relevant points**

Selecting the relevant points from the Binary Image, we’re using a method called “sliding windows”. To know where the lanes begin in the image, we’re using a thing called a histogram. Without going into too much detail, the traditional histogram tells us how many pixels of a certain gray-level there are in the image and plots the number for all values in a graph.

A histogram looks something like this:
Our “Lane_Histogram” calculates something slightly different. It shows us at what column of the image there are the most white pixels (equal to 255). This way, we should find two peaks, and get the beginning of our two lane markers. After this, we move the window upwards (decreasing the row number) and shifting it a bit to the sides (plus and minus a certain percentage of the total column number) in order to find where the lane marker continues. We do this for the entire image.

**Polynomial Regression**

Polynomial regression is the process through which we find a cure that approximates a set of data points, like in the picture below. The curve can be a line (linear regression) or a higher degree polynomial.
In our case, the points are the pixels detected by our edge detector. After selecting the edges of the lane markers, we will use polynomial regression to retrieve the coefficients of the polynomial that approximates those points best. The degree of the polynomial used in our Lane Detector is 3 (Why we chose a 3\textsuperscript{rd} degree polynomial has something to do with the linear approximation of a clothoid model using taylor series and some physical constraints).

If the polynomial that we want to find looks like this:

\[
y = a_0 + a_1 * x + a_2 * x^2 + \cdots + a_{n-1} * x^{n-1} - \text{degree } n
\]

\[
y = a_0 + a_1 * x + a_2 * x^2 + a_3 * x^3 - \text{degree } 3
\]

Finding the coefficients would come down to solving the following linear equation (Linear Algebra again):

\[
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_m
\end{bmatrix} =
\begin{bmatrix}
1 & x_1 & \ldots & x_1^n \\
1 & x_2 & \ldots & x_2^n \\
\vdots & \vdots & \ddots & \vdots \\
1 & x_m & \ldots & x_m^n
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_n
\end{bmatrix}
\]

In OpenCV, this is done with the following function:

```cpp
//! we apply polyfit, which will result in 4 coefficients saved in dst
cv::polyfit(src_x, src_y, dst, 3);
```

Kalman Filtering (optional)

Alright, we managed to get our coefficients. We now have a functional lane detector. Now what? Well, we make it better, obviously.

If you look at the drawn lanes with the found coefficients, you’ll notice that from time to time the lane markers get pretty “wobbly” (It’s a technical term. Trust me, I’m an engineer.). The first idea that should come to mind is that the values are noisy and that we should somehow filter them. The problem with classical filters (mean filter, for example) is that they introduce a big delay in the signal. The stronger the filter, the bigger the delay. In real time systems, delays are a very big problem and they should be avoided as much as possible.
This is where the Kalman Filter comes in handy. It is great at filtering noise AND the delay introduced is only one cycle machine.

You can look at the Kalman Filter as a weighted average of two independent measurements. The idea is to select the weights in such a manner that you take into account the more precise measurement. Here’s where things get cool: how do you quantify precision? What is precision? How do you know which measurement is more precise?

\[ z = K \cdot x + (1 - K) \cdot y, \text{where } K \in [0,1] \text{ and is a real number} \]

The precision of a measurement can be seen as the inverse of the error of that measurement. The error is actually expressed mathematically by the variance of a signal.

\[
\mu = \frac{\sum_{i=0}^{N} x_i}{N}
\]

\[
\sigma^2 = \frac{\sum_{i=0}^{N} (x_i - \mu)^2}{N}
\]

Practically, this can be seen very nicely on a gaussian curve. The higher the variance, the less precise that signal is.

If we calculate the Kalman gain based on the variances of the two measurements, we could get a better approximation of the real value.

Arnautu Stefan, Continental, C&S ADAS
\[ K = \frac{\sigma_x^2}{\sigma_y^2 + \sigma_x^2}, \text{where } \sigma_x^2 \text{ and } \sigma_y^2 \text{ represent the variances of the two signals} \]

**Drawing Lanes (optional)**

At the end, we draw on the original frame the lanes simply for our own pleasure and because we like colorful things (you have to admit it looks cooler than some white numbers in a black console).

To draw the lanes correctly we need access to the polynomial coefficients to recalculate the “path” of the lane marker AND after that we need to do the same transformation we did in the BirdsEyeView chapter but in reverse. Mathematically, this translates to multiplying the array of points with the inverse of the transformation matrix.

We’re using the FillLanes function.

```cpp
namespace DSP{

/** \brief Draw Lanes is an LD Drawing Function.
 * \param[in] frame is a pointer to the image matrix of the current frame.
 * \param[in] corridor is the array with coefficients of the corridor.
 * \param[in] thickness is the thickness in pixels of the drawn lines.
 */
void DrawLanes(cv::Mat* ,cv::Mat, cv::Scalar, unsigned int );

/** \brief Fill Lanes is an LD Drawing Function.
 * \param frame is a pointer to the image matrix of the current frame.
 * \param lanes is the structure with the coefficients of the detected lane markers.
 * \param thickness is the thickness in pixels of the drawn lines.
 */
void FillLanes(cv::Mat* ,t_output LD, unsigned int );
}
for(int x=0;x<tempImage.rows-1;x++)
{
    cv::line(tempImage,points inner left[x],points outer left[x],purple.thickness); //Draw left lane
    cv::line(tempImage,points inner right[x],points outer right[x],blue.thickness); //Draw right lane
    cv::line(tempImage,points inner left[x],points inner right[x],green.thickness); //Draw street in the middle
}
//warp the image back to real world view
cv::warpPerspective(tempImage, tempImage, generic_DSP.M.inv(), (*frame).size(), cv::INTER_LINEAR, cv::BORDER_CONSTANT);
(*frame) = (*frame) + tempImage;
```

**Perspective transformation (Next Year Maybe)**

The main problem with our LaneDetector at this point is that what we detected doesn’t really translate to real world coordinates. The polynomial coefficients do not tell us if the lane markers are 2 meters away or 2 centimeters away. We need to know the position of the camera relative to the highway and some distortion parameters introduced by the lens of the camera. The position of the camera is described by the “extrinsic parameters” and the distortion of the camera is described by the “intrinsic parameters”. Mathematically, they are all cumulated inside the “CAMERA MATRIX” (dun dun dun).
This is generally a very mathematically heavy subject and is part of Linear Algebra (again). We will not tackle it today, but I like to mention it, in case some of you are wondering what the next steps would be.