Night Vision Lane Detection

Yash Chandak

Abstract:

Lane detection plays a key role in the vision-based driver assistance system and is used for vehicle navigation, lateral control, collision prevention, or lane departure warning system. We present an adaptive method for detecting lane marking based on the intensity of road images in night scene which is the cause of numerous accidents. First, a region of interest (ROI) image is extracted from the original image and converted to its grayscale image. After that, we segment out regions likely to be lane using a novel idea based on lane width. Finally, based on the structural and orientation properties of a lane only those segments are kept which match with these required properties. Once lane segments are obtained, hough transform can be applied to estimate lane boundaries. Cognitive methods are then applied to combine results of previous frame with the current one to avoid any miscalculation. Experiment results indicate that the proposed approach was robust and accurate in all conditions and even in night scene.

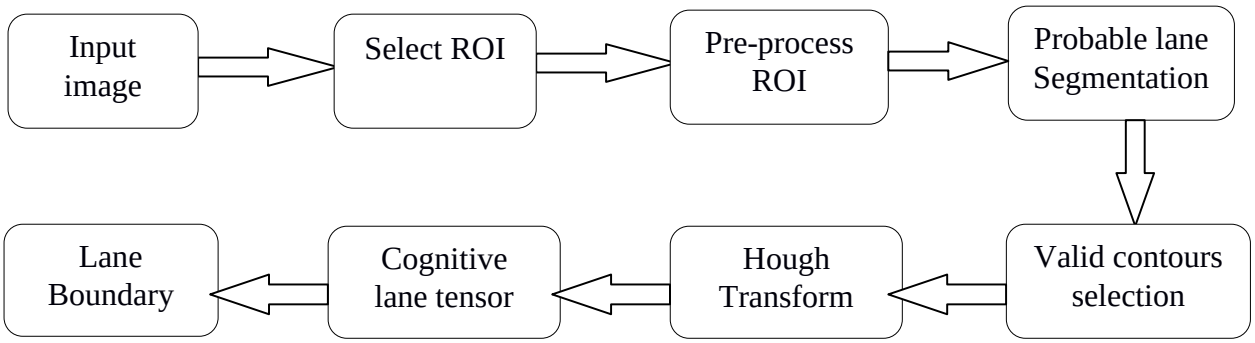
Introduction:

Driver safety has always been an area of interest to research since driver distractions account for numerous accidents on highways. In multiple studies performed by the National Highway Traffic Safety Administration (NHTSA), it was shown that over 20% of the accidents are due driver distractions. Further analysis showed that ≈ 60% of these accidents were caused by drivers who were talking on the phone, adjusting GPS devices or tampering with the CD player. The high fatality rate has prompted industry and academic institutes to focus on embedding smart systems in the automobile that can aid a driver during a commute.

With the increased speed and smaller sizes of complex electronics, intelligent devices are beginning to be integrated into vehicles giving rise to Driver Assistance Systems. Driver Assistance System (DAS) is a synonym for an intelligent system that is a means of providing assistance to the driver. DAS include GPS devices, cruise control, automatic transmissions, anti-lock brakes, traction control etc. This document focuses On lane detection, which is also a form of a Driver Assistance System?

Lane detection is the process of locating lane markers on the road and presenting these locations to an intelligent system. The applications of a lane detecting system could be as simple as pointing out lane locations to the driver on an external display, to more complicated tasks such as predicting a lane change in the immediate future in order to avoid potential collisions with other vehicles. Some of the interfaces used to detect lanes include cameras, laser range images, LIDAR and GPS devices. Our method relies on the use of night vision cameras to accomplish the task.

Methodology:



A) Selection of Region of Interest:

Many papers deal with finding the vanishing point of the image and considering only the region below as ROI. Computing vanishing point is not a trivial task and requires additional computational process. We exploited the mounted camera's property which is so placed that nearly bottom half of the image is the road.

By doing so we don't need to find the vanishing point and the bottom half of the image is taken into account for further processing. It gives a nice approximation considering the computational overhead required for calculating vanishing point.

B) Pre-processing of the ROI:

Unlike day time images, night vision barely provides with any color property of the image which would have otherwise provided us with unique colour set for lane. Along with only meagre hue variation and low saturation levels even the brightness variation of such images is very delicate and highly dependent on external lighting (like street lamps, vehicle's tail/head lamp, signals and the glare because of all these).

To get the intensity value of each pixel, we change the color space from RGB to Black and white using the following equation:

Gray = (Red * 0.299 + Green * 0.587 + Blue * 0.114)

This helps in faster computation without rejecting any valuable information. Bad lighting conditions can make the image harder to process and might result in false results. To avoid this, median filtering was applied. A square kernel of window size 5 was taken and passed over the entire image. The pixel lying in the centre was then replaced with the median value in the kernel. It helped in removing any susceptible noise while maintaining edges. The resultant ROI image's contrast and brightness values were modified to enhance the gradient changes and other boundary features.

C) Probable lane marking segmentation:

Primary step in the lane marking detection is to identify probable lane marking segments. Proper selection is inevitable as it will serve the basis of further processing. Edge detection techniques (eg. Canny and sobel) and thresholding (global and locally adaptive) were possible approaches but both had their own limitations over here. Canny results in either over-dividing the image region or under-dividing it. It doesn't take into account the property of expanded neighbourhood. Edges are detected only based on the immediate surroundings. Dynamic threshold for the entire image using Otsu's method failed because of the varying light conditions in the entire image. Intensity of lane segments themselves varied widely from regions near the headlights/tail lights of the vehicles to shadowed area, whereas adaptive thresholding detected many noise apart from lane markings.

$$d1 = I(x,y) - I(x-\delta,y)$$
$$d2 = I(x,y) - I(x+\delta,y)$$
$$D = d1 + d2 - \left|\left(I(x+\delta,y) - I(x-\delta,y)\right)\right|$$

$$L = 0.5 * I(x,y)$$

$$L(x,y) = \begin{cases} if(d1 > 0 \; and \; d2 > 0 \; and \; D > L) & 255 \\ else & 0 \end{cases}$$

The possible lane markings were selected based on the above parameters. δ Represents the lane width. The lane being brighter in intensity relative to its sides, only if both sides are darker and the sum of the intensity value difference on either side is between a given range then only the pixel is considered as a part of lane segment. The range was calculated using numerous sample point and plotting them. It was found to vary between the 0.25 and 0.75 based on output image's lighting conditions, which after pre-processing the image for better contrast and brightness can be ideally chosen to be around 0.5. Any difference less than that come under noise or non-lane parts. The lane width varies based on distance because of the perspective. Near the base it's maximum whereas near the vanishing point it's minimal. Lane width at any row(r) of the image is calculated using the following formula:

$$\delta_r = min + (max - min) * \frac{\left(r - r_{vanishingpoint}\right)}{\left(r_{total} - r_{vanishingpoint}\right)} + \varepsilon$$

max and min represent the maximum and minimal lane width possible in the given image. Keeping ε value 5 helps in avoiding noises. Max value is dependent on the image size and the mounted camera's position. If the camera is kept very low then because of high perspective and being closer to lane, lane width will be larger near the base as compared to the time when camera is mounted on the top. Minimum by default always remains 0 at vanishing point; otherwise it can be adjusted if required. Once max is set, the above formula can be used to get lane width at varying distances dynamically. Dynamically changing the lane width helps in accurate selection of lanes.

D) Selecting only possible lane segments from above processed image:

The above segmentation process often selects other unwanted noise or regions similar to lane (eg, milestones, edges of vehicles, railings, trees, lampposts, headlight glare etc.).

We exploited the geometric features of a lane segment and based on its property we selected only valid segments. First of all contours were selected from the above binary image using the [Suzuki85] algorithm[1].Then a minimum area rectangle was drawn around it to get its orientation, length and breadth properties.

Lane segment properties taken into consideration were:

- Segment area. Area of segments below minArea threshold denote unwanted object and hence were rejected.
- Ratio of sides. Being a line segment, the ratio of its length to breadth should be greater than 4:1 at least. Only segments with higher ratio were taken into account. Segments having area less than certain threshold but more than minArea can possibly represent small broken centre lane markings and hence ratio for them was brought down to 2:1.
- Orientation. Lane segments by the virtue of their nature are never close to horizontal (unless extremely steep turn is encountered). This property helped us in removing highlighted vehicle bumpers and other segments which were otherwise being treated as false positives.
- Vertical lane segments are possible only if the vehicle is on the lane, in such cases the lane segment will be near the bottom centre region of the image only. Every other vertical segment detected cannot be lanes hence are discarded.
- A min area rectangle was bounded to the detected segment. Lane being very close to rectangles, if the segment area was not close to area of bounding rectangle then the segment was rejected.

The lane segments were the resultant segments that passed the above test. Minor false positives still remained but those can be easily avoided by further process.

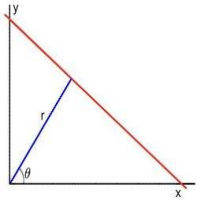
[1]: Suzuki, S. and Abe, K., Topological Structural Analysis of Digitized Binary Images by Border Following. CVGIP 30 1, pp 32-46 (1985)

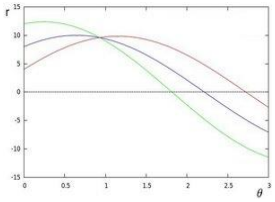
E) Hough Tranform:

<P.S. here the orientation of the segments are known, this knowledge can be exploited to link them using better approaches that will have lesser computational cost and greater efficiency also>

Line segments are either continuous or broken based upon where they are marked.
Along with this because of poor light conditions and glare from other vehicle's head/tail lamp often continuous lane segments are not obtained.

To obtain a proper lane a fixed continuous lane boundary is required. We used the hough transform that serves the purpose well and provides us with the connected lane segment.





Hough transform is computed In the Polar coordinate system: Parameters: (rho, theta).
Hence, a line equation can be written as:

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right)x + \left(\frac{r}{\sin\theta}\right)$$

In general for each point (x0, y0), we can define the family of lines that goes through that point as:

$$r_\theta = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta$$

If for a given (x0, y0) we plot the family of lines that goes through it, we get a sinusoid. Same operation as above is performed for all the points in an image. If the curves of two different points intersect in the plane {theta - r}, that means that both points belong to a same line. By selecting the lines which have number of intersections and are above the required threshold we can get the major lane markings in the image.

<TODO: >

F) Cognitive lane tensor:

Knowledge of lane which was computed in previous frames can be used to predict the lane in upcoming frames. It can be very useful in regions where either lanes are not marked or marked as false negatives i.e which are undetectable by the above procedure because of any unwanted reason. It can also serve as a tensor such that the currently detected lanes dont deviate drastically. This will help in avoiding consequences of false positives.

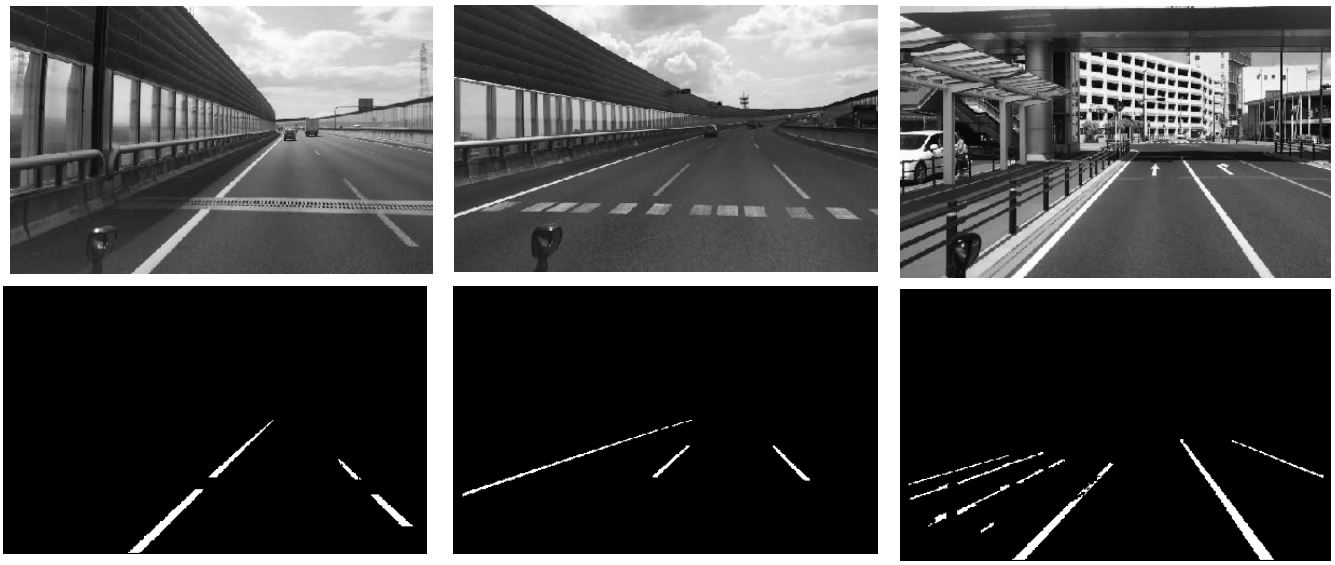
Results

Figure 1: Input Image

Figure 2: Pre-processed and possible lanes selected

Figure 3: Final lane markings selected

Other input images and their final results below:



<source_code>

```
/* ===== NightVision lane Detection ====== */

/*TODO
 * improved edge linking
 * remove blobs whose axis direction doesnt point towards vanishing pt
 * Parallelisation
 * lane prediction
*/


#include <opencv2/core/core.hpp>
#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include <vector>
#include <algorithm>
#include <math.h>
#include <time.h>

using namespace std;
using namespace cv;
clock_t start, stop;

class LaneDetect
{
public:
  Mat currFrame; //stores the upcoming frame
  Mat temp;      //stores intermediate results
  Mat temp2;     //stores the final lane segments

  int diff, diffL, diffR;
  int laneWidth;
  int diffThreshTop;
  int diffThreshLow;
  int ROIrows;
  int vertical_left;
  int vertical_right;
  int vertical_top;
  int smallLaneArea;
  int longLane;
  int  vanishingPt;
  float maxLaneWidth;

  //to store various blob properties
  Mat binary_image; //used for blob removal
  int minSize;
  int ratio;
  float  contour_area;
```

```
  float blob_angle_deg;
  float bounding_width;
  float bounding_length;
  Size2f sz;
  vector< vector<Point> > contours;
  vector<Vec4i> hierarchy;
  RotatedRect rotated_rect;


  LaneDetect(Mat startFrame)
  {
    //currFrame = startFrame;                              //if image has to be processed at original size

    currFrame = Mat(320,480,CV_8UC1,0.0);                  //initialised the image size to 320x480
    resize(startFrame, currFrame, currFrame.size());       // resize the input to required size

    temp      = Mat(currFrame.rows, currFrame.cols, CV_8UC1,0.0);//stores possible lane markings
    temp2     = Mat(currFrame.rows, currFrame.cols, CV_8UC1,0.0);//stores finally selected lane
marks

    vanishingPt    = currFrame.rows/2;                     //for simplicity right now
    ROIrows        = currFrame.rows - vanishingPt;         //rows in region of interest
    minSize        = 0.00015 * (currFrame.cols*currFrame.rows);  //min size of any region to be
selected as lane
    maxLaneWidth   = 0.025 * currFrame.cols;               //approximate max lane width based on
image size
    smallLaneArea  = 7 * minSize;
    longLane       = 0.3 * currFrame.rows;
    ratio          = 4;

    //these mark the possible ROI for vertical lane segments and to filter vehicle glare
    vertical_left  = 2*currFrame.cols/5;
    vertical_right = 3*currFrame.cols/5;
    vertical_top   = 2*currFrame.rows/3;

    namedWindow("lane",2);
    namedWindow("midstep", 2);
    namedWindow("currframe", 2);
    namedWindow("laneBlobs",2);

    getLane();
  }

  void updateSensitivity()
  {
    int total=0, average =0;
    for(int i= vanishingPt; i<currFrame.rows; i++)
      for(int j= 0 ; j<currFrame.cols; j++)
          total += currFrame.at<uchar>(i,j);
    average = total/(ROIrows*currFrame.cols);
    cout<<"average : "<<average<<endl;
  }
```

```
void getLane()
{
  //medianBlur(currFrame, currFrame,5 );
  // updateSensitivity();
  //ROI = bottom half
  for(int i=vanishingPt; i<currFrame.rows; i++)
    for(int j=0; j<currFrame.cols; j++)
         {
           temp.at<uchar>(i,j)    = 0;
           temp2.at<uchar>(i,j)   = 0;
         }

  imshow("currframe", currFrame);
  blobRemoval();
}

void markLane()
{
  for(int i=vanishingPt; i<currFrame.rows; i++)
    {
         //IF COLOUR IMAGE IS GIVEN then additional check can be done
         // lane markings RGB values will be nearly same to each other(i.e without any hue)

         //min lane width is taken to be 5
         laneWidth =5+ maxLaneWidth*(i-vanishingPt)/ROIrows;
         for(int j=laneWidth; j<currFrame.cols- laneWidth; j++)
           {

             diffL = currFrame.at<uchar>(i,j) - currFrame.at<uchar>(i,j-laneWidth);
             diffR = currFrame.at<uchar>(i,j) - currFrame.at<uchar>(i,j+laneWidth);
             diff  =  diffL + diffR - abs(diffL-diffR);

             //1 right bit shifts to make it 0.5 times
             diffThreshLow = currFrame.at<uchar>(i,j)>>1;
             //diffThreshTop = 1.2*currFrame.at<uchar>(i,j);

             //both left and right differences can be made to contribute
             //at least by certain threshold (which is >0 right now)
             //total minimum Diff should be atleast more than 5 to avoid noise
             if (diffL>0 && diffR >0 && diff>5)
               if(diff>=diffThreshLow /*&& diff<= diffThreshTop*/ )
                    temp.at<uchar>(i,j)=255;
           }
    }

}

void blobRemoval()
{
  markLane();
```

```
   // find all contours in the binary image
   temp.copyTo(binary_image);
   findContours(binary_image, contours,
                hierarchy, CV_RETR_CCOMP,
                CV_CHAIN_APPROX_SIMPLE);

   // for removing invalid blobs
   if (!contours.empty())
     {
         for (size_t i=0; i<contours.size(); ++i)
           {
            //====conditions for removing contours====//

            contour_area = contourArea(contours[i]) ;

            //blob size should not be less than lower threshold
            if(contour_area > minSize)
              {
                rotated_rect    = minAreaRect(contours[i]);
                sz              = rotated_rect.size;
                bounding_width  = sz.width;
                bounding_length = sz.height;


                //openCV selects length and width based on their orientation
                //so angle needs to be adjusted accordingly
                blob_angle_deg = rotated_rect.angle;
                if (bounding_width < bounding_length)
                  blob_angle_deg = 90 + blob_angle_deg;

                //if such big line has been detected then it has to be a (curved or a normal)lane
                if(bounding_length>longLane || bounding_width >longLane)
                  {
                    drawContours(currFrame, contours,i, Scalar(255), CV_FILLED, 8);
                    drawContours(temp2, contours,i, Scalar(255), CV_FILLED, 8);
                  }

                //angle of orientation of blob should not be near horizontal or vertical
                //vertical blobs are allowed only near center-bottom region, where centre lane mark
is present
                //length:width >= ratio for valid line segments
                //if area is very small then ratio limits are compensated
                else if ((blob_angle_deg <-10 || blob_angle_deg >-10 ) &&
                    ((blob_angle_deg > -70 && blob_angle_deg < 70 ) ||
                     (rotated_rect.center.y > vertical_top &&
                      rotated_rect.center.x > vertical_left && rotated_rect.center.x < vertical_right)))

                  {

                    if ((bounding_length/bounding_width)>=ratio ||
(bounding_width/bounding_length)>=ratio
```

```
                            ||(contour_area< smallLaneArea &&
((contour_area/(bounding_width*bounding_length)) > .75) &&
                              ((bounding_length/bounding_width)>=2 ||
(bounding_width/bounding_length)>=2)))
                        {
                            drawContours(currFrame, contours,i, Scalar(255), CV_FILLED, 8);
                            drawContours(temp2, contours,i, Scalar(255), CV_FILLED, 8);
                        }
                    }
                }
            }
        }
   imshow("midstep", temp);
   imshow("laneBlobs", temp2);
   imshow("lane",currFrame);
  }


  void nextFrame(Mat &nxt)
  {
   //currFrame = nxt;                    //if processing is to be done at original size

   resize(nxt ,currFrame, currFrame.size()); //resizing the input image for faster processing
   getLane();
  }

  Mat getResult()
  {
   return temp2;
  }

};//end of class LaneDetect


void makeFromVid(string path)
{
  Mat frame;
  VideoCapture cap(path); // open the video file for reading

  if ( !cap.isOpened() )  // if not success, exit program
    cout << "Cannot open the video file" << endl;

  //cap.set(CV_CAP_PROP_POS_MSEC, 300); //start the video at 300ms

  double fps = cap.get(CV_CAP_PROP_FPS); //get the frames per seconds of the video
  cout << "Input video's Frame per seconds : " << fps << endl;

  cap.read(frame);
  LaneDetect detect(frame);

  while(1)
    {
```

```
    bool bSuccess = cap.read(frame); // read a new frame from video
    if (!bSuccess)                   //if not success, break loop
         {
          cout << "Cannot read the frame from video file" << endl;
          break;
         }

    cvtColor(frame, frame, CV_BGR2GRAY);

    //start = clock();
    detect.nextFrame(frame);
    //stop =clock();
    // cout<<"fps : "<<1.0/(((double)(stop-start))/ CLOCKS_PER_SEC)<<endl;

    if(waitKey(10) == 27) //wait for 'esc' key press for 10 ms. If 'esc' key is pressed, break loop
         {
          cout<<"video paused!, press q to quit, any other key to continue"<<endl;
          if(waitKey(0) == 'q')
            {
              cout << "terminated by user" << endl;
              break;
            }
         }
   }
}

int main()
{
   makeFromVid("/home/yash/opencv-2.4.10/programs/output.avi");
   // makeFromVid("/home/yash/opencv-2.4.10/programs/road.m4v");
  waitKey(0);
  destroyAllWindows();
}
```