Presentation

1. Background

描述
ROBOWATCH 是全球最早将移动机器人和安全市场成功衔接起来的技术公司，开创了无人平台技术和遥控车辆技术在安全领域的应用先河，开创了德国服务机器人的技术路线和应用体系。
我们欢迎您加入 ROBOWATCH 上海大家庭，作为一名软件（实习）工程师。您将通过 2-3 个月时间对于无人驾驶有详细地了解，并负责选择一个开源的识别车道线的算法并实现功能。

Based on simple lane line recognition (picture reading, grayscale, marginal extraction, ROI area selection, Hough effect recognition line) completed last month, several problems were found:
1. Image preprocessing is too simple, resulting in a specific situation such as the road is too white, tree shadow interference, can not accurately filter the lane line;
2. The feature extracted by the Hough effect is that the straight line in the picture causes the recognition function of the curve to be completely lost;
3. The lane line identification is to assist the lane line offset alarm system in the ADAS system, so it is necessary to measure the offset distance of the car to the center of the lane line in real time, which is not solved in the simple lane line identification report.

2. Purpose

Using C++, open source algorithms are used to implement easy lane recognition and to exercise unmanned image processing capabilities. Solve the above problems in the simple lane recognition project.

3. Preparation

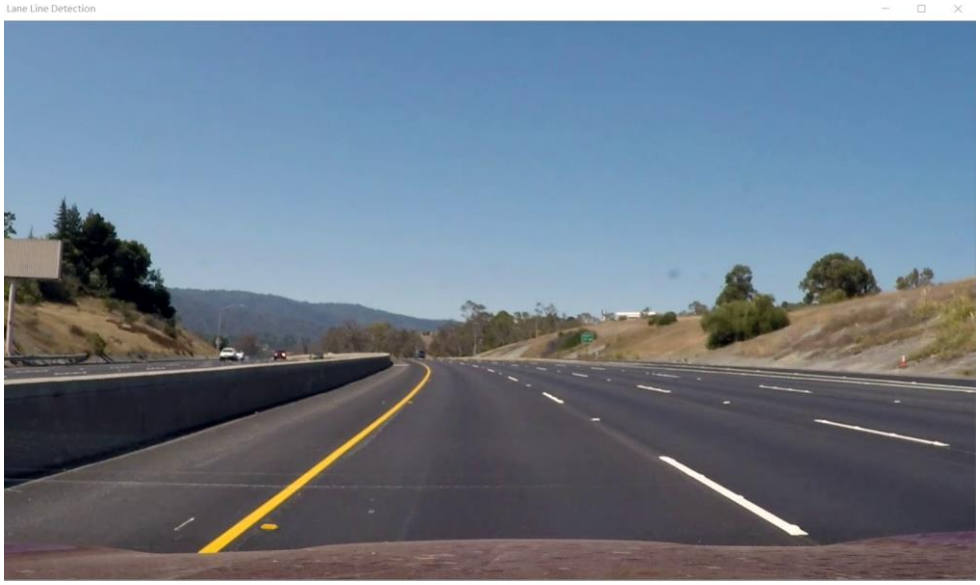
Platform : I chose Ubuntu Linux14.04 again. Since the previous project was based on the ubuntu platform, I made this choice because the OpenCV installation has been completed.

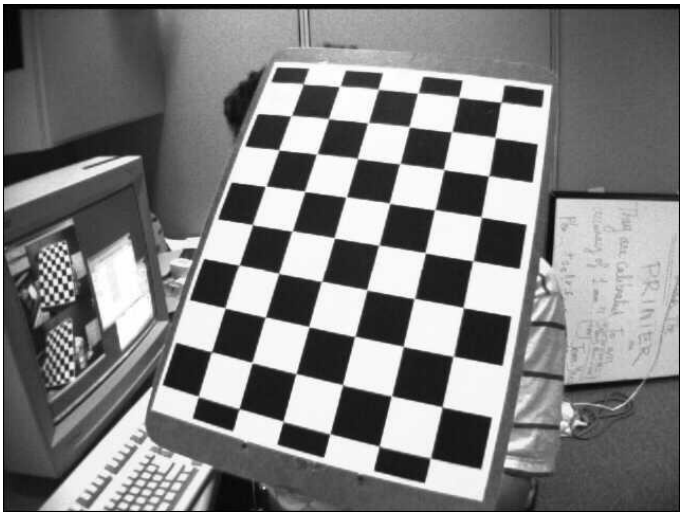
Software : As mentioned in the previous article, we need to use an already formed image processing library OpenCV 3.4.6 for image processing.

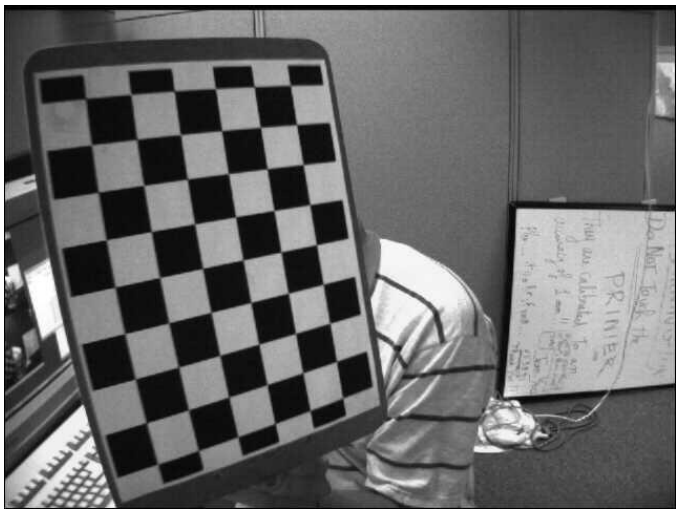
Picture : A number of curved pictures (video capture), and the a number of checkerboard photos for camera calibration (comes with OpenCV).

Video : This project uses a section of highway bend video with a size of 1280*720, which

can be downloaded from https://github.com/udacity/CarND-Advanced-Lane-Lines.







Processing

1. Step 1 : Image preprocessing --- Camera Calibration

Based on the explanation of the camera correction principle given by OpenCV, we need to get the camera parameters here:

$$\text{Camera matrix (3*3 matrix)} \begin{bmatrix} fx & 0 & Cx \\ 0 & fy & Cy \\ 0 & 0 & 1 \end{bmatrix},$$

$$\text{Distortion coefficient (1*5 matrix)} \quad [k1 \quad k2 \quad p1 \quad p2 \quad k3]。$$

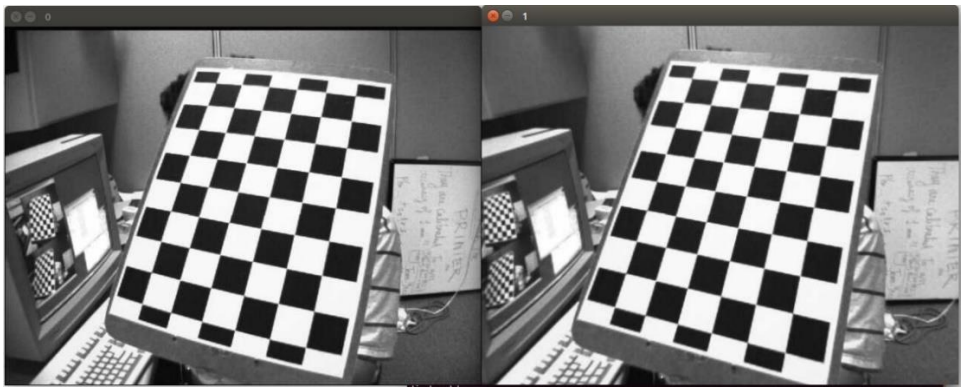
Since there is no camera nearby, I decided to use OpenCV's own image: left01(02, 03, ...,14).png and right01(02, 03, ...,14).png located in the opencv/samples/data directory.

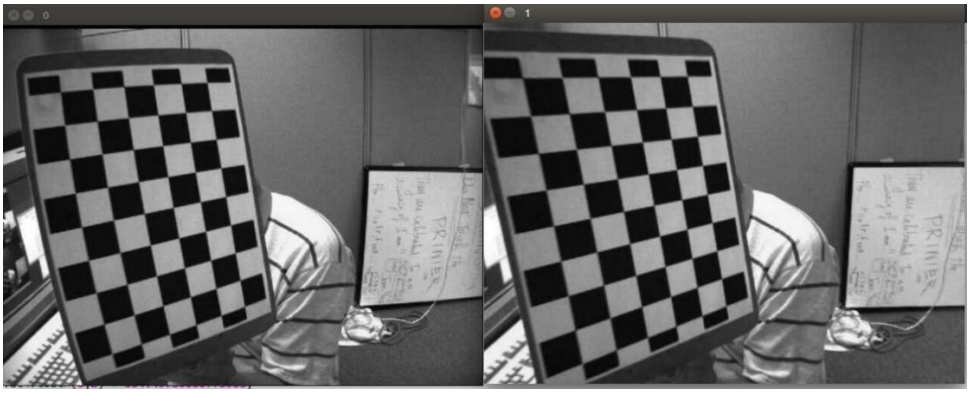
First call OpenCV's own program: imagelist_creator.cpp located in the opencv/samples/cpp directory to generate the file imglist.yml containing the image list. After successful, call opencv and then take the calibration.cpp in the same directory of the program to generate the parameters containing the required parameters. Out_camera_data.yml, where we can find the data we need: camera matrix$\begin{bmatrix} fx & 0 & Cx \\ 0 & fy & Cy \\ 0 & 0 & 1 \end{bmatrix}$, Distortion coefficient$[k1 \quad k2 \quad p1 \quad p2 \quad k3]$。

Use the data, the specific code is as follows:

```
int main()
{
    Mat img, gray;
    cv::Mat distCoeff;
    distCoeff = cv::Mat::zeros(5, 1, CV_64FC1);
    double k1 = -0.28947423596733812;
    double k2 = 0.11691606633399480;
    double p1 = 0.00056987179790175943;
    double p2 = -0.00071772002488826433;
    double k3 = -0.036552074090605596;

    distCoeff.at<double>(0, 0) = k1;
    distCoeff.at<double>(1, 0) = k2;
    distCoeff.at<double>(2, 0) = p1;
    distCoeff.at<double>(3, 0) = p2;
```

```
    distCoeff.at<double>(4, 0) = k3;

    cv::Mat cameraMatrix;
    cameraMatrix = cv::Mat::eye(3, 3, CV_32FC1);

    cameraMatrix.at<float>(0, 0) = 539.33428512225385;
    cameraMatrix.at<float>(0, 1) = 0.0;
    cameraMatrix.at<float>(0, 2) = 334.49736056773935;
    cameraMatrix.at<float>(1, 0) = 0;
    cameraMatrix.at<float>(1, 1) = 539.33428512225385;
    cameraMatrix.at<float>(1, 2) = 241.01542127818917;
    cameraMatrix.at<float>(2, 0) = 0;
    cameraMatrix.at<float>(2, 1) = 0;
    cameraMatrix.at<float>(2, 2) = 1;

    img = imread("D:\\opencv-
3.4.6\\opencv\\sources\\samples\\data\\right08.jpg");
    Mat dst;
    undistort(img, dst, cameraMatrix, distCoeff);
    imshow("0", img);
    imshow("1", dst);
    waitKey(0);
    return 0;
}
```





2. Step 2 : Image preprocessing --- Perspective Transform

In real life, the two lane lines are parallel to each other, but in the perspective of the camera, the lane line will gradually narrow from near to far until it merges into a point, and the lane line in this state will affect the test. As a result, here we need to use perspective transformation to make the lane lines parallel in image processing.

OpenCV provides us with two open source algorithms to implement this transformation. First we need to use getPerspectiveTransform(const cv::Point2f* src, const cv::Point2f* dst), the former is the selected initial rectangular four-point coordinate array. The latter is the target four-point coordinate array that you want to transform to.

Based on the image of 1280*720: the starting coordinates here are (577, 460), (700, 460), (1112, 720), (232, 720), and the target point is (300, 0). (950, 0), (950, 720), (300, 720).

Secondly, we use warpPerspective (InputArray src, OutputArray dst, InputArray M, Size dsize, int flags=INTER_LINEAR, int borderMode=BORDER_CONSTANT, const Scalar& borderValue=Scalar()) to achieve the purpose of perspective transformation.

Detailed parameters:

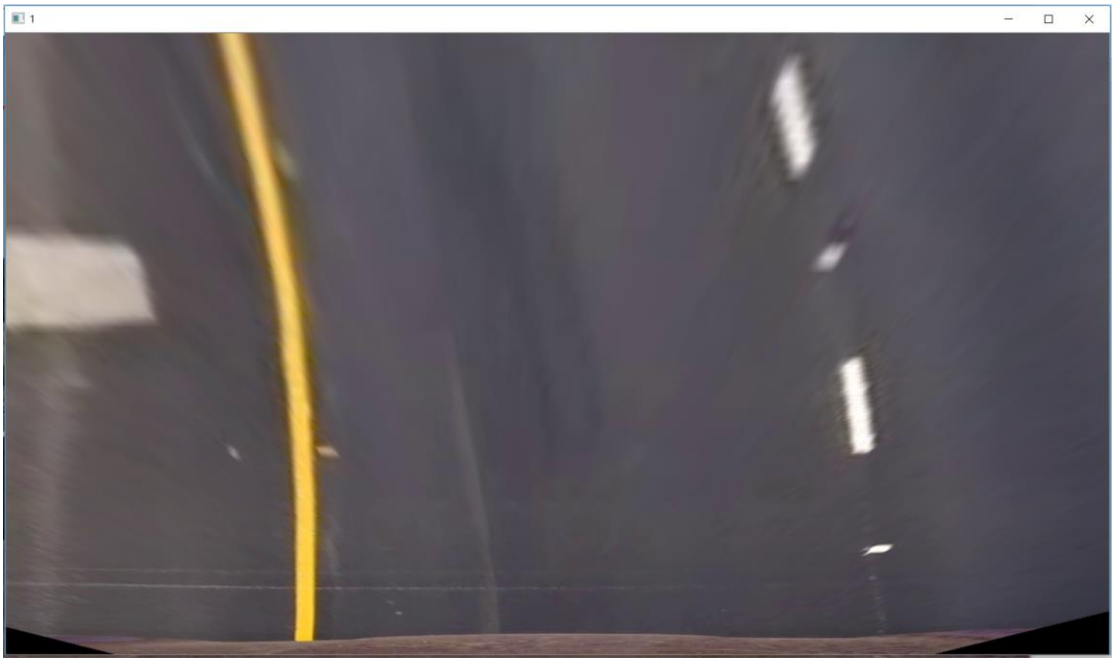
InputArray src: the input image
OutputArray dst: the output image
InputArray M: matrix of perspective transformation
Size dsize: the size of the output image
Int flags=INTER_LINEAR: Interpolation method for output images
Int borderMode=BORDER_CONSTANT: How image boundaries are handled
Const Scalar& borderValue=Scalar(): The color setting of the border, the default is 0

Specific code:

```
void Perspective(const cv::Mat& src, cv::Mat& dst, const int& width, const
int& height) {
    vector<Point2f> points(4), points_after(4);
    points[0] = Point2f(577, 460);
    points[1] = Point2f(700, 460);
    points[2] = Point2f(1112, 720);
    points[3] = Point2f(232, 720);

    points_after[0] = Point2f(300, 0);
    points_after[1] = Point2f(950, 0);
    points_after[2] = Point2f(950, 720);
    points_after[3] = Point2f(300, 720);
```

```
    Mat transform = getPerspectiveTransform(points, points_after);

    warpPerspective(src, dst, transform, Size(width, height), INTER_LINEAR);
}
```

在主程序中通过 Perspective(TempImage, ThreshImage, Frame.cols, Frame.rows);来实现透视转换以及逆透视转换的功能。



3. Step 3 : Image preprocessing --- Thresholding

In order to make the computer more efficient and reduce the error when processing the captured road image, we need to filter the road image. In this project, I chose Sobel filtering (X direction and XY direction), LUV filtering (L channel). , HLS (S channel) filtering to achieve only the lane line in the image, this can reduce the time memory used by the computer to analyze the image.

After the threshold filtering process, we will get a binary map of the lane line, the white area RGB value is (255, 255, 255), and the relative black area RGB value is (0, 0, 0), this is the next step. Find the basis for the lane line. The binary map is as follows:

After the threshold filtering process, the image of only the lane line is obtained, and the effect is achieved by a combination of various filtering :

```
TempImage = (absm & mag & luv) | (hls & luv);
```

Specific code:

Sobel X:

```
void abs_sobel_thresh(const cv::Mat& src, cv::Mat& dst, const char& orient,
const int& thresh_min, const int& thresh_max) {
    cv::Mat src_gray, grad;
    cv::Mat abs_gray;
    //transform into grayscale
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    if (orient == 'x') {
        cv::Sobel(src_gray, grad, CV_64F, 1, 0);
```

```
        cv::convertScaleAbs(grad, abs_gray);
    }
    if (orient == 'y') {
        cv::Sobel(src_gray, grad, CV_64F, 0, 1);
        cv::convertScaleAbs(grad, abs_gray);
    }
    //binary
    cv::inRange(abs_gray, thresh_min, thresh_max, dst);
}
```

Sobel XY:

```
void mag_thresh(const cv::Mat& src, cv::Mat& dst, const int& sobel_kernel,
const int& thresh_min, const int& thresh_max) {
    cv::Mat src_gray, gray_x, gray_y, grad;
    cv::Mat abs_gray_x, abs_gray_y;
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    cv::Sobel(src_gray, gray_x, CV_64F, 1, 0, sobel_kernel);
    cv::Sobel(src_gray, gray_y, CV_64F, 0, 1, sobel_kernel);
    cv::convertScaleAbs(gray_x, abs_gray_x);
    cv::convertScaleAbs(gray_y, abs_gray_y);
    cv::addWeighted(abs_gray_x, 0.5, abs_gray_y, 0.5, 0, grad);
    cv::inRange(grad, thresh_min, thresh_max, dst);
}
```

HLS:

```
void hls_select(const cv::Mat& src, cv::Mat& dst, const char& channel, const
int& thresh_min, const int& thresh_max) {
    cv::Mat hls, grad;
    vector<cv::Mat> channels;
    cv::cvtColor(src, hls, cv::COLOR_RGB2HLS);
    cv::split(hls, channels);
    switch (channel)
    {
        case 'h':
            grad = channels.at(0);
            break;
        case 'l':
            grad = channels.at(1);
            break;
        case 's':
            grad = channels.at(2);
```

```
            break;
        default:
            break;
    }
    cv::inRange(grad, thresh_min, thresh_max, dst);
}
```

LUV：

```
void luv_select(const cv::Mat& src, cv::Mat& dst, const char& channel, const
int& thresh_min, const int& thresh_max) {
    cv::Mat luv, grad;
    vector<cv::Mat> channels;
    cv::cvtColor(src, luv, cv::COLOR_RGB2Luv);
    cv::split(luv, channels);
    switch (channel)
    {
        case 'l':
            grad = channels.at(0);
            break;
    case 'u':
            grad = channels.at(1);
            break;
    case 'v':
            grad = channels.at(2);
            break;
    default:
            break;
    }
    cv::inRange(grad, thresh_min, thresh_max, dst);
}
```

Main：

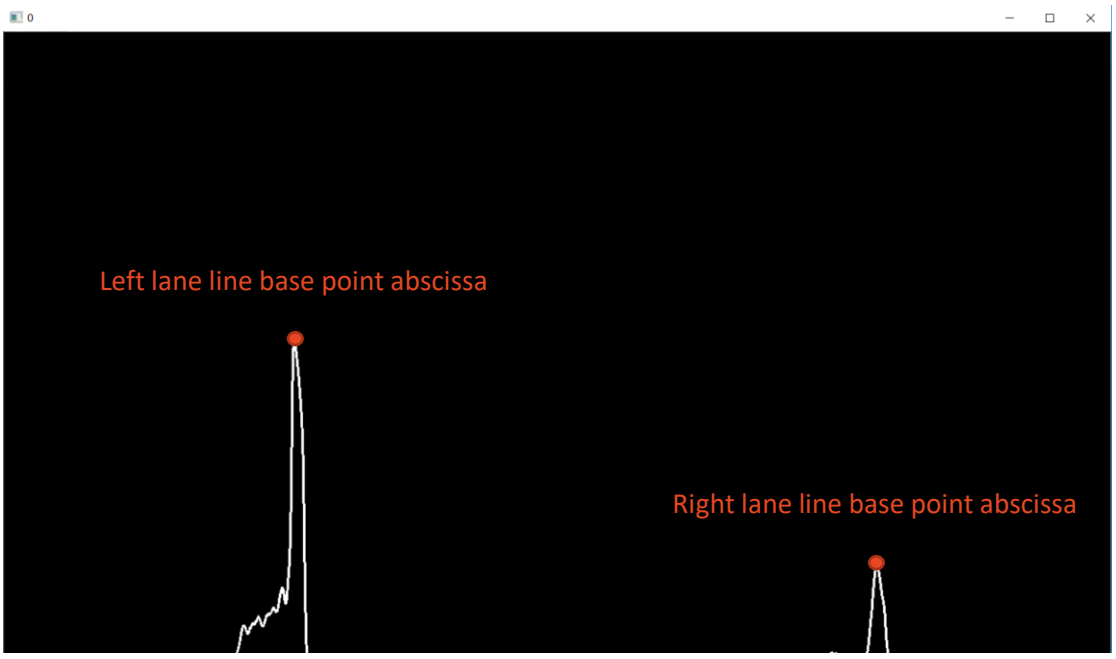
```
abs_sobel_thresh(Frame, absm, 'x', 55, 200);
mag_thresh(Frame, mag, 3, 45, 150);
hls_select(Frame, hls, 's', 120, 255);
luv_select(Frame, luv, 'l', 180, 255);
TempImage = (absm & mag & luv) | (hls & luv);
```

Combined with the perspective transformation of step two, we will obtain parallel binary lane line images through the code Perspective (TempImage, ThreshImage, Frame.cols,

Frame.rows):

4. Step 4 : Lane line detection --- find and locate lines

The number of white pixel points of the corresponding coordinates of the superimposed display image is calculated by the histogram Histogram obtained by the perspective-converted lane line binary map obtained in the third step. The base point coordinate of the left lane line is the peak of the left half of the histogram, and the base point of the corresponding right lane line is the peak of the right half of the histogram. The histogram is shown in the figure below:



Histogram statistics white pixel code：

```
for (int i = 0; i < 1280; i++)
            {
                Array[i] = 0;
            }
            for (int row = 0; row < Frame.rows; row++)
            {
                for (int col = 0; col < Frame.cols; col++)
                {
                    if (ThreshImage.at<uchar>(row, col) == 255)
                    {
                        Array[col]++;
                    }
                }
            }

            for (int col = 0; col < Frame.cols; col++)
            {
```

```
                line(DispImage, Point(col, 720 - Array[col]), Point(col + 1, 720
    - Array[col + 1]), Scalar(255), 2, LINE_8);
                }
```

After grasping the coordinates of the base line of the left and right lane lines, use the sliding window to fit the sliding window polynomial fitting to locate the lane line. Here, 12 sliding windows with a width of 150px are selected.

Specific code:

```
for (int i = 0; i < 12; i++)
    {
        for (int WinRow = 720 - 60 * (i + 1); WinRow < 720 - 60 * (i); WinRow++)
        {
            CurrRowP = ThreshImage.ptr<uchar>(WinRow);
            CurrRowP += ((LeftBase - 75) * 3);
            for (int lWinCol = LeftBase - 75; lWinCol < LeftBase + 75; lWinCol++)
            {
                if (((*CurrRowP) != 0) || ((*(CurrRowP + 1)) != 0) || ((*(CurrRowP +
2)) != 0))
                {
                    lxPositon += lWinCol;
                    lNum++;
                }
                CurrRowP += 3;
            }

            CurrRowP = ThreshImage.ptr<uchar>(WinRow) + ((RightBase - 75) * 3);
            for (int rWinCol = RightBase - 75; rWinCol < RightBase + 75; rWinCol++)
            {
                if (((*CurrRowP) != 0) || ((*(CurrRowP + 1)) != 0) || ((*(CurrRowP +
2)) != 0))
                {
                    rxPositon += rWinCol;
                    rNum++;
                }
                CurrRowP += 3;
            }
        }
        //绘制滑动窗
        if (lNum > 0)
        {
            LeftBase = (lxPositon / lNum);
            lNum = 0;
```

```
            lxPositon = 0;
        }

        if (rNum > 0)
        {
            RightBase = (rxPositon / rNum);
            rNum = 0;
            rxPositon = 0;
        }
```

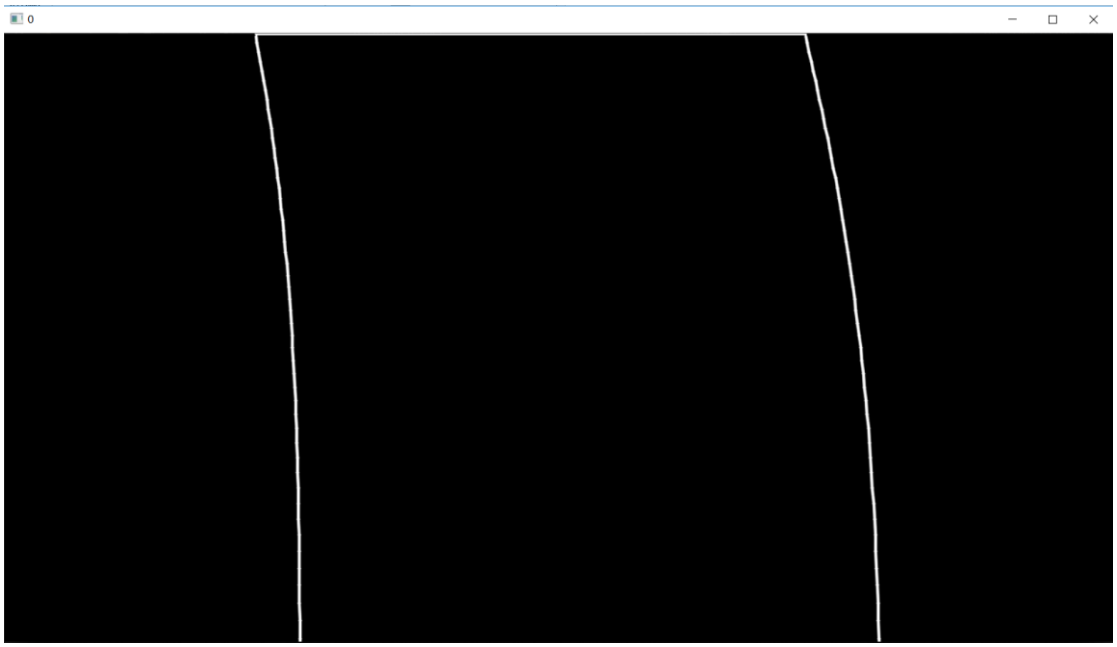
After finding the feature points on all the curves, we need to filter out three points to do the feature curve fitting. Here I select the top and bottom of the image and the three central feature points, and use Bezier curve fitting to realize the lane line drawing.

Specific code:

```
    if (0 == i)
            {
                LeftBaseTemp = LeftBase;
                RightBaseTemp = RightBase;
            }
            if (i == 4)
            {
                ControlPts[1].x = LeftBase;
                ControlPts[1].y = (720 - 60 * (i + 2));
                ControlPts[1].z = 0;

                ControlPtsR[1].x = RightBase;
                ControlPtsR[1].y = (720 - 60 * (i + 2));
                ControlPtsR[1].z = 0;
            }
            if (i == 7)
            {
                ControlPts[2].x = LeftBase;
                ControlPts[2].y = (720 - 60 * (i + 2));
                ControlPts[2].z = 0;

                ControlPtsR[2].x = RightBase;
                ControlPtsR[2].y = (720 - 60 * (i + 2));
                ControlPtsR[2].z = 0;
            }
            if (i == 10)
            {
                ControlPts[3].x = LeftBase;
```

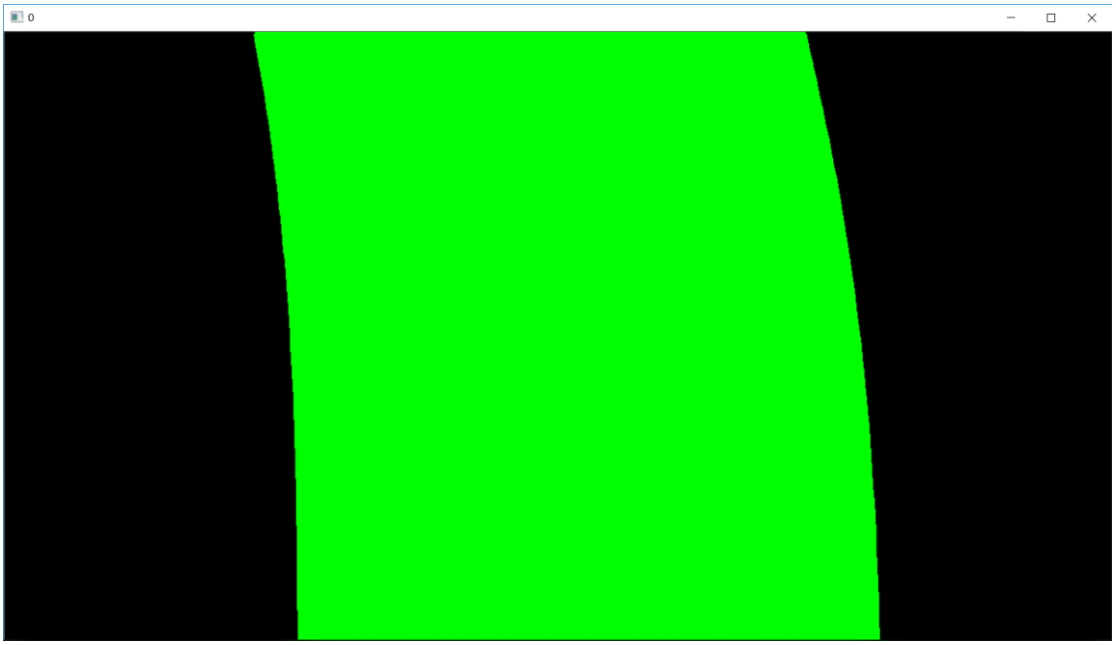
```
                    ControlPts[3].y = (720 - 60 * (i + 2));
                    ControlPts[3].z = 0;

                    ControlPtsR[3].x = RightBase;
                    ControlPtsR[3].y = (720 - 60 * (i + 2));
                    ControlPtsR[3].z = 0;
                    line(WinSlip, Point(ControlPts[3].x, ControlPts[3].y),
Point(ControlPtsR[3].x, ControlPtsR[3].y), Scalar(255), 2, LINE_AA);
               }
          }
          DrawBezier(WinSlip, ControlPts);
          DrawBezier(WinSlip, ControlPtsR);
```

Through the above steps, we will get a binary map of the curved lane line:



The color of the lane line contour formed by the four vertices of the quadrilateral is filled:

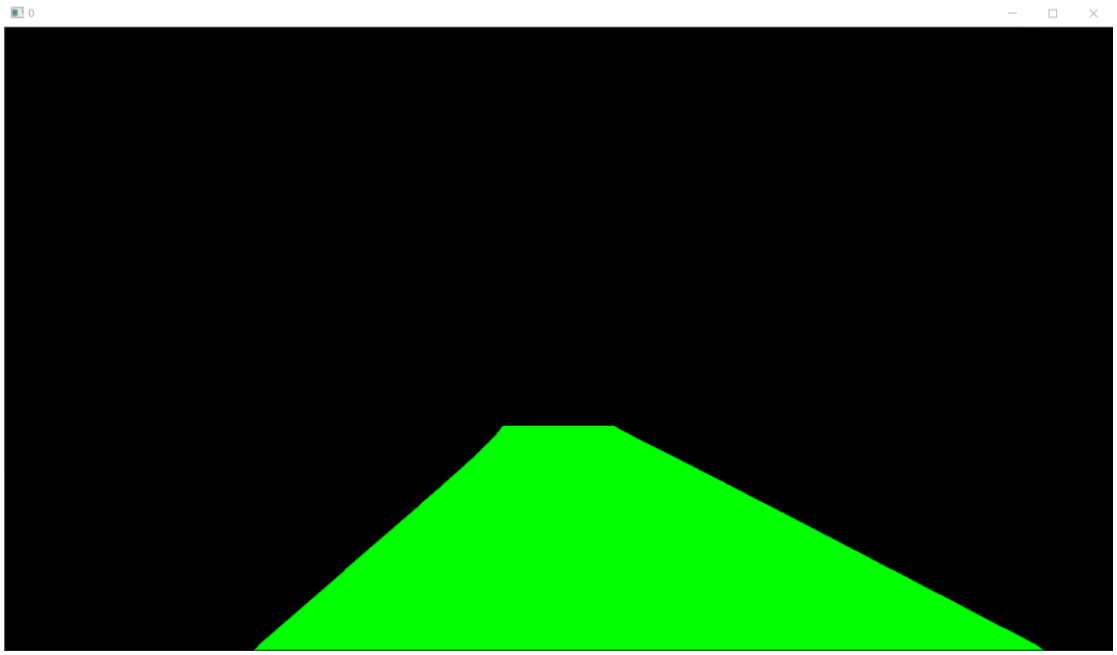
```
cv::findContours(WinSlip, Contours, Hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE,
Point(0, 0));
          cv::cvtColor(WinSlip, WinSlip, COLOR_GRAY2BGR);
          for (int i = 0; i < Contours.size(); i++)
          {
               drawContours(WinSlip, Contours, i, Scalar(0, 255, 0), -1,
LINE_8);
          }
```



Then, through the inverse perspective change (the reverse step of step 2), the obtained curve contour map is projected onto the screen captured by the camera in real time to realize the lane region recognition.

Specific code:

```
void inverse_Perspective(const cv::Mat& src, cv::Mat& dst, const int& width, const int& height) {
        vector<Point2f> points(4), points_after(4);
        points[0] = Point2f(577, 460);
        points[1] = Point2f(700, 460);
        points[2] = Point2f(1112, 720);
        points[3] = Point2f(232, 720);

        points_after[0] = Point2f(300, 0);
        points_after[1] = Point2f(950, 0);
        points_after[2] = Point2f(950, 720);
        points_after[3] = Point2f(300, 720);

        Mat transform = getPerspectiveTransform(points_after,points );

        warpPerspective(src, dst, transform, Size(width, height), INTER_LINEAR);
    }

    inverse_Perspective(WinSlip, TempImage, Frame.cols, Frame.rows);
```



```
FinalDisp = Frame * 0.8 + TempImage * 0.2;
```

5. Step 5 : Lane line detection --- Caculation of curvature radius and center offset distance

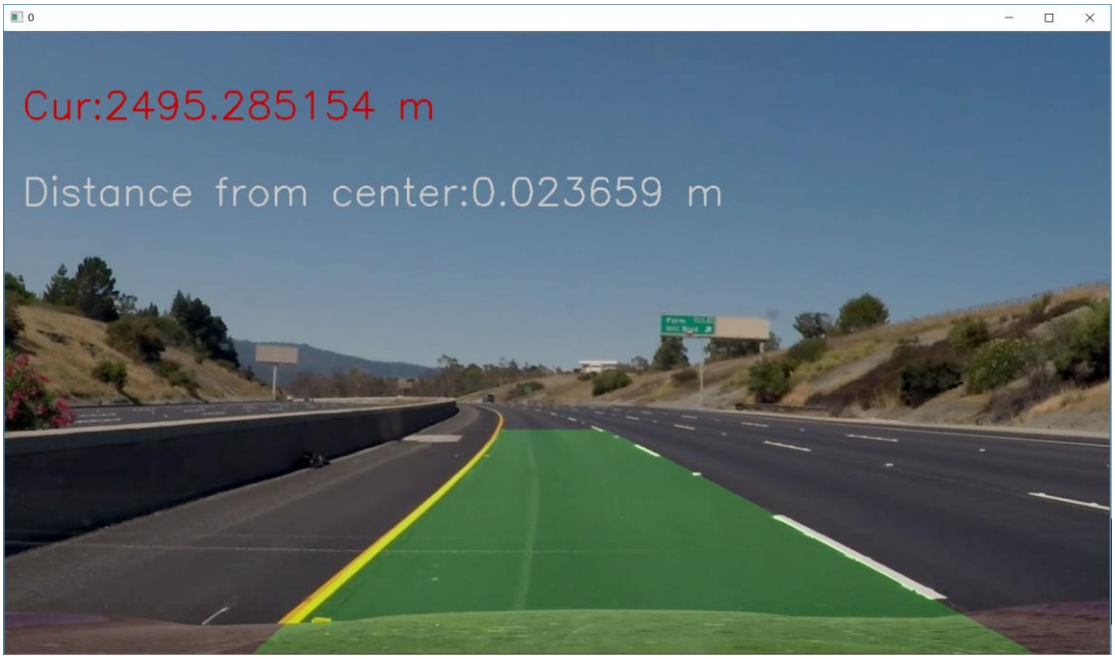
For the convenience and humanized application of the project, the calculation of the radius of curvature (convenient for the vehicle to know the extent of the turn) and the center deviation distance calculation (to facilitate the vehicle in the center of the lane line).

Specific code :

```
double Curvature(Point p1, Point p2, Point p3)
{
    double Cur;
    double Radius = 0.0;
    cv::Point p0;
    if (1 == Collinear(p1, p2, p3))
    {
        Cur = 0.0;
    }
    else
    {
        double a = p1.x - p2.x;
        double b = p1.y - p2.y;
        double c = p1.x - p3.x;
        double d = p1.y - p3.y;
        double e = ((p1.x * p1.x - p2.x * p2.x) + (p1.y * p1.y - p2.y * p2.y)) /
2.0;
        double f = ((p1.x * p1.x - p3.x * p3.x) + (p1.y * p1.y - p3.y * p3.y)) /
2.0;
        double det = b * c - a * d;

        p0.x = -(d * e - b * f) / det;
        p0.y = -(a * f - c * e) / det;
        Radius = Distance(p0, p1);
    }
    return Radius;
}

    Cura.x = (ControlPts[0].x + ControlPtsR[0].x) / 2;
            Cura.y = (ControlPts[0].y + ControlPtsR[0].y) / 2;
            Curb.x = (ControlPts[2].x + ControlPtsR[2].x) / 2;
            Curb.y = (ControlPts[2].y + ControlPtsR[2].y) / 2;
            Curc.x = (ControlPts[3].x + ControlPtsR[3].x) / 2;
            Curc.y = (ControlPts[3].y + ControlPtsR[3].y) / 2;
            Cur = Curvature(Cura, Curb, Curc);
```

```
            cv::putText(Frame, format("Cur:%f m", Cur), Point(20, 100),
FONT_HERSHEY_SIMPLEX, 1.5, Scalar(0, 0, 255), 2, LINE_8);

            distance = (Curb.x - Frame.cols / 2) * 3.75 / (ControlPtsR[3].x -
ControlPts[3].x);
            cv::putText(Frame, format("Distance from center:%f m", distance),
Point(20, 200), FONT_HERSHEY_SIMPLEX, 1.5, Scalar(255, 255, 255), 2, LINE_8);
```

Displayed on the captured image:



Conclusion

1. Total code

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;
using namespace std;

struct UserData
{
    Mat Image;
    vector<Point2f>Points;
};

int Divs = 50;

/*****************************************************************************
************************/

Point3d PointAdd(Point3d p, Point3d q);
Point3d PointTimes(float c, Point3d p);
Point3d Bernstein(float u, Point3d* p);
void DrawBezier(Mat& Image, Point3d* pControls);
double Distance(Point p1, Point p2);
int Collinear(Point p1, Point p2, Point p3);
double Curvature(Point p1, Point p2, Point p3);
void Perspective(const cv::Mat& src, cv::Mat& dst, const int& width, const int&
height);
void inverse_Perspective(const cv::Mat& src, cv::Mat& dst, const int& width, const
int& height);

Point3d PointAdd(Point3d p,Point3d q)
{
    p.x += q.x;
    p.y += q.y;
    p.z += q.z;
    return p;
}

Point3d PointTimes(float c,Point3d p)
{
```

```
    p.x *= c;
    p.y *= c;
    p.z *= c;
    return p;
}

//P1*t^3 + P2*3*t^2*(1-t) + P3*3*t*(1-t)^2 + P4*(1-t)^3 = Pnew
Point3d Bernstein(float u,Point3d *p)
{
    Point3d a, b, c, d, r;
    a = PointTimes(pow(u,3),p[0]);
    b = PointTimes(3*pow(u,2)*(1-u),p[1]);
    c = PointTimes(3*u*pow((1-u),2),p[2]);
    d = PointTimes(pow((1-u),3),p[3]);

    r = PointAdd(PointAdd(a,b),PointAdd(c,d));
    return r;
}

void DrawBezier(Mat &Image,Point3d *pControls)
{
    Point NowPt, PrePt;
    for (int i = 0; i <=Divs; i++)
    {
        float u = (float)i / Divs;
        Point3d NewPt = Bernstein(u,pControls);

        NowPt.x = (int)NewPt.x;
        NowPt.y = (int)NewPt.y;
        if (i>0)
        {
            line(Image,NowPt,PrePt,Scalar(255),2,LINE_AA,0);
        }
        PrePt = NowPt;
    }
}

/*********************************************************************************
**********************/
double Distance(Point p1, Point p2)
{
    double Dis;
    double x2, y2;
    x2 = (p1.x - p2.x)*(p1.x - p2.x);
```

```
    y2 = (p1.y - p2.y)*(p1.y - p2.y);
    Dis = sqrt(x2 + y2);
    return Dis;
}

int Collinear(Point p1, Point p2, Point p3)
{
    double k1, k2;
    double kx1, ky1, kx2, ky2;
    if (p1.x == p2.x && p2.x == p3.x)
    {
        return 1;
    }
    else
    {
        kx1 = p2.x - p1.x;
        kx2 = p2.x - p3.x;
        ky1 = p2.y - p1.y;
        ky2 = p2.y - p3.y;
        k1 = ky1 / kx1;
        k2 = ky2 / kx2;
        if (k1 == k2)
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
}

double Curvature(Point p1, Point p2, Point p3)
{
    double Cur;
    double Radius = 0.0;
    cv::Point p0;
    if (1 == Collinear(p1, p2, p3))
    {
        Cur = 0.0;
    }
    else
    {
        double a = p1.x - p2.x;
```

```
            double b = p1.y - p2.y;
            double c = p1.x - p3.x;
            double d = p1.y - p3.y;
            double e = ((p1.x * p1.x - p2.x * p2.x) + (p1.y * p1.y - p2.y * p2.y)) /
2.0;
            double f = ((p1.x * p1.x - p3.x * p3.x) + (p1.y * p1.y - p3.y * p3.y)) /
2.0;
            double det = b * c - a * d;

            p0.x = -(d * e - b * f) / det;
            p0.y = -(a * f - c * e) / det;
            Radius = Distance(p0, p1);
        }
        return Radius;
    }

    void Perspective(const cv::Mat& src, cv::Mat& dst, const int& width, const int&
    height) {
        vector<Point2f> points(4), points_after(4);
        points[0] = Point2f(577, 460);
        points[1] = Point2f(700, 460);
        points[2] = Point2f(1112, 720);
        points[3] = Point2f(232, 720);

        points_after[0] = Point2f(300, 0);
        points_after[1] = Point2f(950, 0);
        points_after[2] = Point2f(950, 720);
        points_after[3] = Point2f(300, 720);

        Mat transform = getPerspectiveTransform(points, points_after);

        warpPerspective(src, dst, transform, Size(width, height), INTER_LINEAR);
    }

    void inverse_Perspective(const cv::Mat& src, cv::Mat& dst, const int& width, const
    int& height) {
        vector<Point2f> points(4), points_after(4);
        points[0] = Point2f(577, 460);
        points[1] = Point2f(700, 460);
        points[2] = Point2f(1112, 720);
        points[3] = Point2f(232, 720);

        points_after[0] = Point2f(300, 0);
        points_after[1] = Point2f(950, 0);
```

```
    points_after[2] = Point2f(950, 720);
    points_after[3] = Point2f(300, 720);

    Mat transform = getPerspectiveTransform(points_after,points );

    warpPerspective(src, dst, transform, Size(width, height), INTER_LINEAR);
}

void abs_sobel_thresh(const cv::Mat& src, cv::Mat& dst, const char& orient, const
int& thresh_min, const int& thresh_max) {
    cv::Mat src_gray, grad;
    cv::Mat abs_gray;
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    if (orient == 'x') {
        cv::Sobel(src_gray, grad, CV_64F, 1, 0);
        cv::convertScaleAbs(grad, abs_gray);
    }
    if (orient == 'y') {
        cv::Sobel(src_gray, grad, CV_64F, 0, 1);
        cv::convertScaleAbs(grad, abs_gray);
    }
    cv::inRange(abs_gray, thresh_min, thresh_max, dst);
}

void hls_select(const cv::Mat& src, cv::Mat& dst, const char& channel, const int&
thresh_min, const int& thresh_max) {
    cv::Mat hls, grad;
    vector<cv::Mat> channels;
    cv::cvtColor(src, hls, cv::COLOR_RGB2HLS);
    cv::split(hls, channels);
    switch (channel)
    {
    case 'h':
        grad = channels.at(0);
        break;
    case 'l':
        grad = channels.at(1);
        break;
    case 's':
        grad = channels.at(2);
        break;
    default:
        break;
    }
```

```
    cv::inRange(grad, thresh_min, thresh_max, dst);
}

void luv_select(const cv::Mat& src, cv::Mat& dst, const char& channel, const int&
thresh_min, const int& thresh_max) {
    cv::Mat luv, grad;
    vector<cv::Mat> channels;
    cv::cvtColor(src, luv, cv::COLOR_RGB2Luv);
    cv::split(luv, channels);
    switch (channel)
    {
    case 'l':
        grad = channels.at(0);
        break;
    case 'u':
        grad = channels.at(1);
        break;
    case 'v':
        grad = channels.at(2);
        break;
    default:
        break;
    }
    cv::inRange(grad, thresh_min, thresh_max, dst);
}

void mag_thresh(const cv::Mat& src, cv::Mat& dst, const int& sobel_kernel, const
int& thresh_min, const int& thresh_max) {
    cv::Mat src_gray, gray_x, gray_y, grad;
    cv::Mat abs_gray_x, abs_gray_y;
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    cv::Sobel(src_gray, gray_x, CV_64F, 1, 0, sobel_kernel);
    cv::Sobel(src_gray, gray_y, CV_64F, 0, 1, sobel_kernel);
    cv::convertScaleAbs(gray_x, abs_gray_x);
    cv::convertScaleAbs(gray_y, abs_gray_y);
    cv::addWeighted(abs_gray_x, 0.5, abs_gray_y, 0.5, 0, grad);
    cv::inRange(grad, thresh_min, thresh_max, dst);
}

/**********************************************************************************
***/
```

```
int main(void)
{
    VideoCapture Capture("D:\\opicture\\test1.mp4");
    Mat Frame, TempImage, ImageDest, LabImage, HlsImage, ThreshImage,
ThreshImagez, ThreshImagey, FinalDisp, H, Hinv;
    Mat DispImage = Mat::zeros(720, 1280, CV_8UC1);
    Mat WinSlip = Mat::zeros(720, 1280, CV_8UC1);
    Mat absm, mag, hls, luv, lab, dst, persp, blur;

    vector<vector<Point>> Contours;
    vector<Vec4i> Hierarchy;
    vector<Point2f> DestSrc;
    vector<Mat> Channels;

    UserData Data;

    int Array[1280] = { 0 };
    int LeftBaseTemp = 0;
    int RightBaseTemp = 0;
    int LeftBase = 10000, Thresh = 720, RightBase = 10000;
    int Lock = 0;

    double lxPositon = -1, rxPositon = -1, lNum = 0, rNum = 0, Cur = 0.0, distance
= 0.0, r = 0.0;
    Point3d ControlPts[40];
    Point3d ControlPtsR[40];
    Point Cura, Curb, Curc;
    uchar* CurrRowP;


    while (Capture.read(Frame))
    {
        TempImage = Frame.clone();
        Data.Image = TempImage;

        abs_sobel_thresh(Frame, absm, 'x', 55, 200);
        mag_thresh(Frame, mag, 3, 45, 150);
        hls_select(Frame, hls, 's', 120, 255);
        luv_select(Frame, luv, 'l', 180, 255);
        TempImage = (absm & mag & luv) | (hls & luv);
        Perspective(TempImage, ThreshImage, Frame.cols, Frame.rows);
        ImageDest = Mat::zeros(Frame.size(), CV_8UC3);
```

```
        WinSlip = Mat::zeros(720, 1280, CV_8UC1);
        Mat Test = Mat::zeros(720, 1280, CV_8UC1);


        if (0 == Lock)
        {
            Lock = 1;
            for (int i = 0; i < 1280; i++)
            {
                Array[i] = 0;
            }
            for (int row = 0; row < Frame.rows; row++)
            {
                for (int col = 0; col < Frame.cols; col++)
                {
                    if (ThreshImage.at<uchar>(row, col) == 255)
                    {
                        Array[col]++;
                    }
                }
            }

            for (int col = 0; col < Frame.cols; col++)
            {
                line(DispImage, Point(col, 720 - Array[col]), Point(col + 1, 720
- Array[col + 1]), Scalar(255), 2, LINE_8);
            }

            for (int row = 0; row < Frame.rows; row++)
            {
                for (int col = 0; col < Frame.cols / 2; col++)
                {
                    if (DispImage.at<uchar>(row, col) == 255)
                    {
                        if (row < Thresh)
                        {
                            Thresh = row;
                            LeftBase = col;
                        }
                    }
                }
            }

            Thresh = 720;
```

```
            for (int row = 0; row < Frame.rows; row++)
            {
                for (int col = Frame.cols / 2; col < Frame.cols; col++)
                {
                    if (DispImage.at<uchar>(row, col) == 255)
                    {
                        if (row < Thresh)
                        {
                            Thresh = row;
                            RightBase = col;
                        }
                    }
                }
            }

            ControlPts[0].x = LeftBase;
            ControlPts[0].y = 720;
            ControlPts[0].z = 0;

            ControlPtsR[0].x = RightBase;
            ControlPtsR[0].y = 720;
            ControlPtsR[0].z = 0;
        }
        else
        {
            ControlPts[0].x = LeftBaseTemp;
            ControlPts[0].y = 720;
            ControlPts[0].z = 0;

            ControlPtsR[0].x = RightBaseTemp;
            ControlPtsR[0].y = 720;
            ControlPtsR[0].z = 0;

            LeftBase = LeftBaseTemp;
            RightBase = RightBaseTemp;
            line(WinSlip, Point(ControlPts[0].x, ControlPts[0].y),
Point(ControlPtsR[0].x, ControlPtsR[0].y), Scalar(255), 2, LINE_AA);
        }
            cv::cvtColor(ThreshImage, ThreshImage, COLOR_GRAY2BGR);

            for (int i = 0; i < 12; i++)
            {
                for (int WinRow = 720 - 60 * (i + 1); WinRow < 720 - 60 * (i);
WinRow++)
```

```
                {
                    CurrRowP = ThreshImage.ptr<uchar>(WinRow);
                    CurrRowP += ((LeftBase - 75) * 3);
                    for (int lWinCol = LeftBase - 75; lWinCol < LeftBase + 75;
lWinCol++)
                    {
                        if (((*CurrRowP) != 0) || ((*(CurrRowP + 1)) != 0) ||
((*(CurrRowP + 2)) != 0))
                        {
                            lxPositon += lWinCol;
                            lNum++;
                        }
                        CurrRowP += 3;
                    }


                    CurrRowP = ThreshImage.ptr<uchar>(WinRow) + ((RightBase -
75) * 3);
                    for (int rWinCol = RightBase - 75; rWinCol < RightBase + 75;
rWinCol++)
                    {
                        if (((*CurrRowP) != 0) || ((*(CurrRowP + 1)) != 0) ||
((*(CurrRowP + 2)) != 0))
                        {
                            rxPositon += rWinCol;
                            rNum++;
                        }
                        CurrRowP += 3;
                    }
                }
                if (lNum > 0)
                {
                    LeftBase = (lxPositon / lNum);
                    lNum = 0;
                    lxPositon = 0;
                }

                if (rNum > 0)
                {
                    RightBase = (rxPositon / rNum);
                    rNum = 0;
                    rxPositon = 0;
                }
```

```
                if (0 == i)
                {
                    LeftBaseTemp = LeftBase;
                    RightBaseTemp = RightBase;
                }
                if (i == 4)
                {
                    ControlPts[1].x = LeftBase;
                    ControlPts[1].y = (720 - 60 * (i + 2));
                    ControlPts[1].z = 0;

                    ControlPtsR[1].x = RightBase;
                    ControlPtsR[1].y = (720 - 60 * (i + 2));
                    ControlPtsR[1].z = 0;
                }
                if (i == 7)
                {
                    ControlPts[2].x = LeftBase;
                    ControlPts[2].y = (720 - 60 * (i + 2));
                    ControlPts[2].z = 0;

                    ControlPtsR[2].x = RightBase;
                    ControlPtsR[2].y = (720 - 60 * (i + 2));
                    ControlPtsR[2].z = 0;
                }
                if (i == 10)
                {
                    ControlPts[3].x = LeftBase;
                    ControlPts[3].y = (720 - 60 * (i + 2));
                    ControlPts[3].z = 0;

                    ControlPtsR[3].x = RightBase;
                    ControlPtsR[3].y = (720 - 60 * (i + 2));
                    ControlPtsR[3].z = 0;
                    line(WinSlip, Point(ControlPts[3].x, ControlPts[3].y),
Point(ControlPtsR[3].x, ControlPtsR[3].y), Scalar(255), 2, LINE_AA);
                }
            }

            DrawBezier(WinSlip, ControlPts);
            DrawBezier(WinSlip, ControlPtsR);
            cv::findContours(WinSlip, Contours, Hierarchy, RETR_EXTERNAL,
CHAIN_APPROX_SIMPLE, Point(0, 0));
            cv::cvtColor(WinSlip, WinSlip, COLOR_GRAY2BGR);
```

```
            for (int i = 0; i < Contours.size(); i++)
            {
                drawContours(WinSlip, Contours, i, Scalar(0, 255, 0), -1,
LINE_8);
            }
            inverse_Perspective(WinSlip, TempImage, Frame.cols, Frame.rows);
            Cura.x = (ControlPts[0].x + ControlPtsR[0].x) / 2;
            Cura.y = (ControlPts[0].y + ControlPtsR[0].y) / 2;
            Curb.x = (ControlPts[2].x + ControlPtsR[2].x) / 2;
            Curb.y = (ControlPts[2].y + ControlPtsR[2].y) / 2;
            Curc.x = (ControlPts[3].x + ControlPtsR[3].x) / 2;
            Curc.y = (ControlPts[3].y + ControlPtsR[3].y) / 2;
            Cur = Curvature(Cura, Curb, Curc);
            cv::putText(Frame, format("Cur:%f m", Cur), Point(20, 100),
FONT_HERSHEY_SIMPLEX, 1.5, Scalar(0, 0, 255), 2, LINE_8);

            distance = (Curb.x - Frame.cols / 2) * 3.75 / (ControlPtsR[3].x -
ControlPts[3].x);
            cv::putText(Frame, format("Distance from center:%f m", distance),
Point(20, 200), FONT_HERSHEY_SIMPLEX, 1.5, Scalar(255, 255, 255), 2, LINE_8);
            //*/
            FinalDisp = Frame * 0.8 + TempImage * 0.2;

            cv::imshow("0", FinalDisp);
            //cv::imshow("Lane Line Detection", FinalDisp);
            cv::waitKey(1);
        }
        return 1;
    }
```

2. Remaining problem

- Low real-time processing efficence
- Curve detection in a small part of time is not accurate
- The actual test of the code cannot be performed due to the lack of the camera and the experimental car.

3. Project application

Lane lane offset warning system LDWS and lane keeping system LCA in the advanced driver assistance system ADAS system.