项目介绍

1. 项目背景

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。

基于上个月完成的简单车道线识别（图片读取，灰度化，边缘化提取，ROI 区域选择，霍夫效应识别直线），发现了数个问题：

1. 图像预处理过于简单导致在特定情况下譬如路面过白，树阴影干扰，无法准确的过滤得到车道线；
2. 由于霍夫效应提取的特征是图片中的直线导致弯道的识别功能完全丧失；
3. 车道线识别是为了辅佐 ADAS 系统中的车道线偏移报警系统，故需要实时的测量汽车对于车道线中心的偏移距离，在简易车道线识别报告中并未解决该问题。

2. 目的

运用 C++，使用开源算法来实现简易车道线识别，锻炼无人驾驶图像处理能力。
解决上述在简易车道线识别项目中出现的问题。

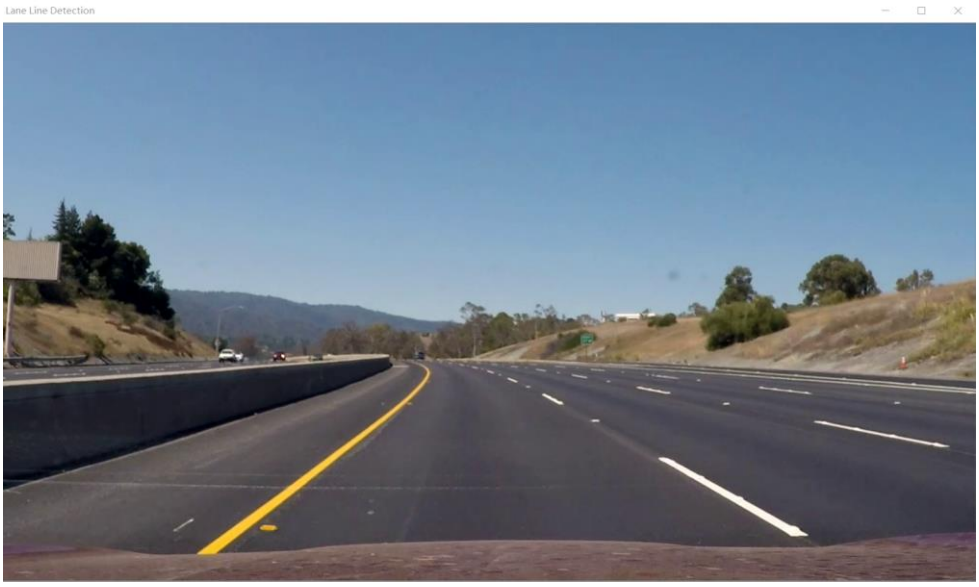
3. 前期准备相关平台，软件，图片，视频

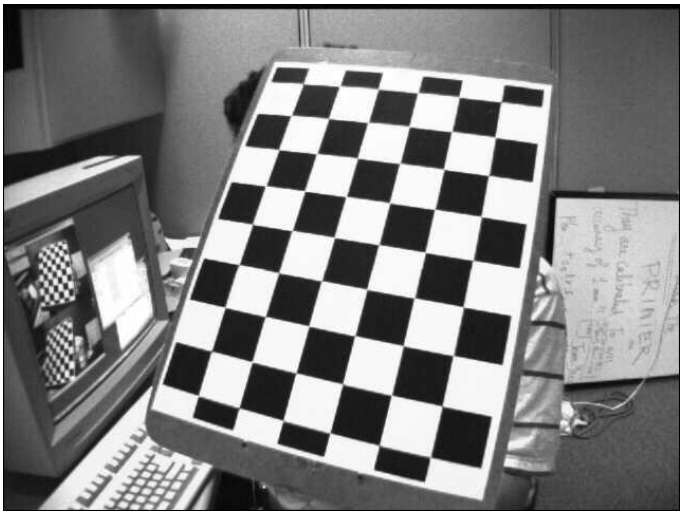
平台：我本人再次选用 Ubuntu Linux14.04，由于之前做的项目是基于 ubuntu 平台，加以由于 OpenCV 安装已经完成故而作此选择。

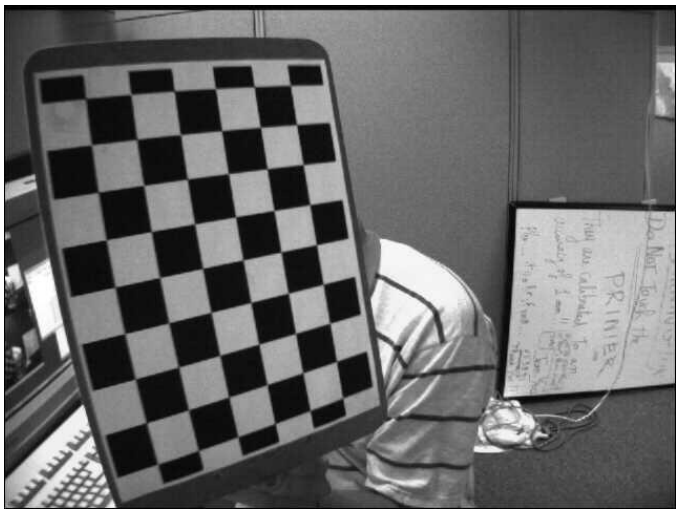
软件：前文已经提起过，为实现图像处理功能我们需用到一个已经成型的图像处理库 OpenCV 3.4.6。

图片：弯道图片若干（视频截取），相机矫正棋盘照片若干（OpenCV 自带）。

视频：本项目采用大小为 1280*720 的一段高速公路弯道视频，于 https://github.com/udacity/CarND-Advanced-Lane-Lines 该网址下载。


Lane Line Detection





项目进程

1. 实现步骤一 ： 图像预处理 --- 相机矫正 Camera Calibration

基于 OpenCV 官方给出的相机矫正原理的解释，我们需要在此处获取相机的参数：

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

$$\text{畸变系数（1*5 矩阵）}[k1 \quad k2 \quad p1 \quad p2 \quad k3]。$$

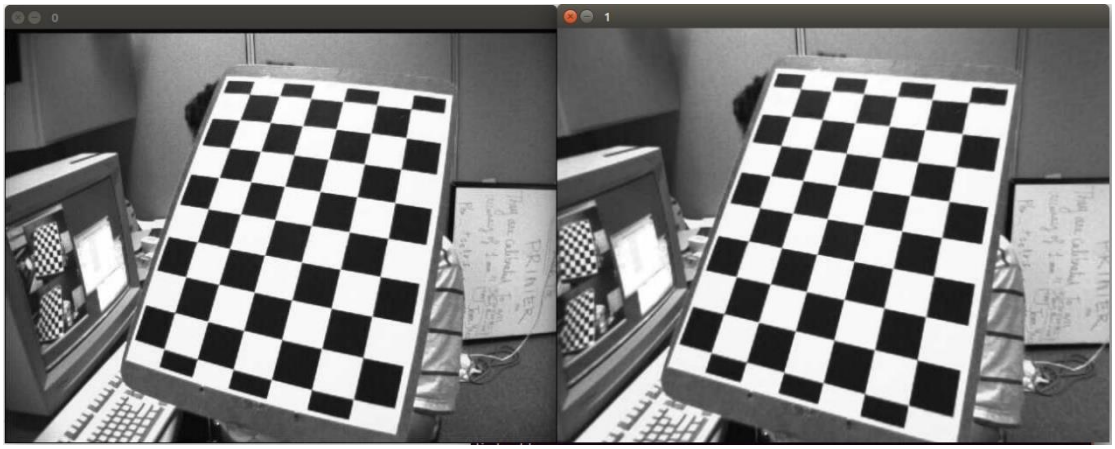
由于身边暂无摄像头，本人决定使用 OpenCV 自带图片：位于 opencv/samples/data 目录下的 left01(02, 03, …,14).png 以及 right01(02, 03, …,14).png。

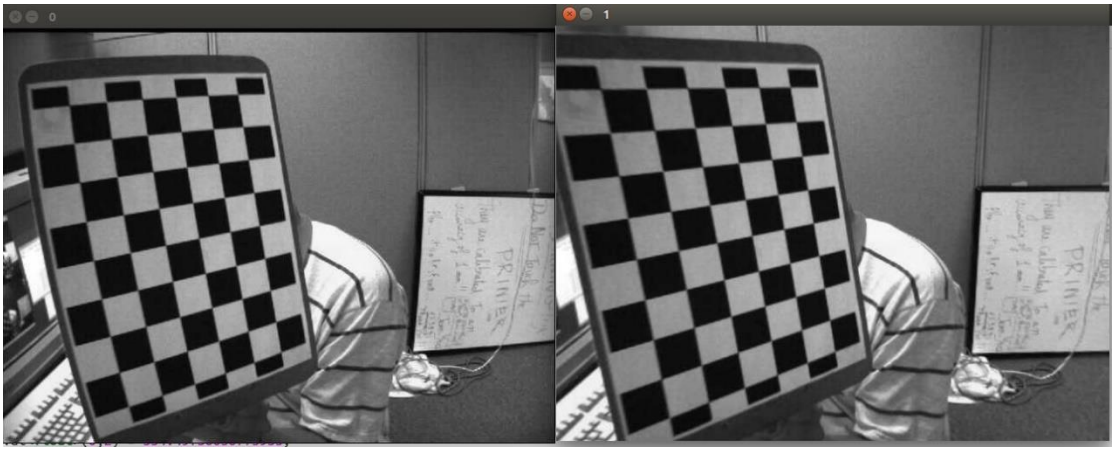
首先调用 OpenCV 自带程序：位于 opencv/samples/cpp 目录下的 imagelist_creator.cpp 来生成包含图片列表的文件 imglist.yml，成功后调用 opencv 再带程序相同目录下的 calibration.cpp 来生成包含需要参数的 out_camera_data.yml，我们在其中可以发现我们需要的数据：相机矩阵$\begin{bmatrix} fx & 0 & Cx \\ 0 & fy & Cy \\ 0 & 0 & 1 \end{bmatrix}$，畸变系数$[k1 \quad k2 \quad p1 \quad p2 \quad k3]$。

运用数据，具体代码如下：

```
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. 实现步骤二 ： 图像预处理 --- 透视变换 Perspective Transform

在现实生活中，两条车道线互相呈平行姿态，但是在照相机看来的视角中，车道线会由近及远的逐渐变窄直至汇聚成一个点，这样状态下的车道线会影响到测试结果，所以在此处我们需要动用透视变换来使车道线在图像处理时呈平行姿态。

OpenCV 提供给我们了两个开源算法来实现这一变换，首先我们需要动用 getPerspectiveTransform(const cv::Point2f* src, const cv::Point2f* dst)，前者为选定的初始矩形四点坐标数组，后者为想要变换到的目标四点坐标数组。

基于 1280*720 的图片下：此处起始坐标我定为(577, 460), (700, 460), (1112, 720), (232, 720)，目标点定为(300, 0), (950, 0), (950, 720), (300, 720)。

其次，我们动用 warpPerspective(InputArray src, OutputArray dst, InputArray M, Size dsize, int flags=INTER_LINEAR, int borderMode=BORDER_CONSTANT, const Scalar& borderValue=Scalar())来实现透视变换的目的。

参数详解：

InputArray src：输入的图像
OutputArray dst：输出的图像
InputArray M：透视变换的矩阵
Size dsize：输出图像的大小
int flags=INTER_LINEAR：输出图像的插值方法
int borderMode=BORDER_CONSTANT：图像边界的处理方式
const Scalar& borderValue=Scalar()：边界的颜色设置，一般默认是 0

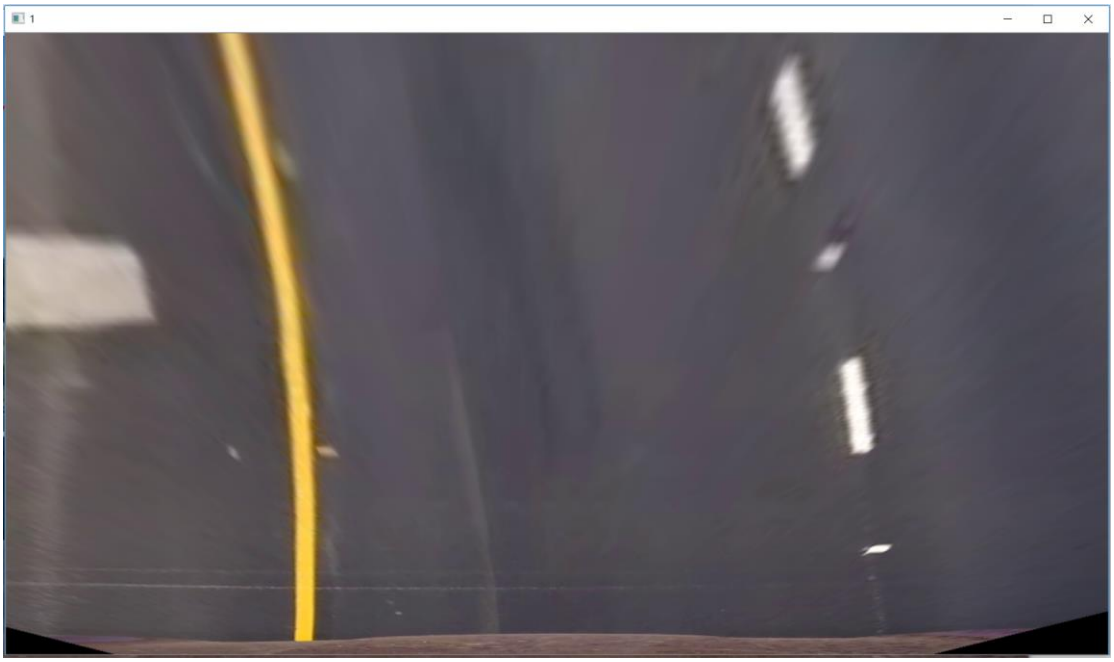
具体代码如下：

透视变换：

```
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. 实现步骤三：图像预处理 --- 阈值过滤 Thresholding

为了计算机在处理拍摄到的路面图像时更加有效率并且减小误差，我们需要将路面图像进行阈值过滤，在本项目中我选用了 Sobel 过滤（X 方向以及 XY 方向），LUV 过滤（L 通道），HLS（S 通道）过滤来达到只保留车道线在图像中，此举可以减少计算机分析图像所耗费的时间内存。

在阈值过滤处理之后，我们将得到一副车道线的二值图，白色区域 RGB 值为（255，255，255），相对的黑色区域 RGB 值为（0，0，0），这是下一步找到车道线的依据。二值图见如下：

经过阈值过滤的处理，已经得到了需要的只留下车道线的图像，由多种过滤结合实现该效果：`TempImage = (absm & mag & luv) | (hls & luv);`

具体代码：

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;
    //转换成为灰度图片
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    //使用cv::Sobel()计算x方向或y方向的导
    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);
    }
    //二值化
    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;
    //转换成为灰度图片
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    //使用cv::Sobel()计算x方向或y方向的导
    cv::Sobel(src_gray, gray_x, CV_64F, 1, 0, sobel_kernel);
    cv::Sobel(src_gray, gray_y, CV_64F, 0, 1, sobel_kernel);
    //转换成CV_8U
    cv::convertScaleAbs(gray_x, abs_gray_x);
    cv::convertScaleAbs(gray_y, abs_gray_y);
    //合并x和y方向的梯度
    cv::addWeighted(abs_gray_x, 0.5, abs_gray_y, 0.5, 0, grad);
    //二值化
    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);
    //分离通道
    cv::split(hls, channels);
    //选择通道
    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;
    }
    //二值化
    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);
    //分离通道
    cv::split(luv, channels);
    //选择通道
    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;
    }
    //二值化
    cv::inRange(grad, thresh_min, thresh_max, dst);
}
```

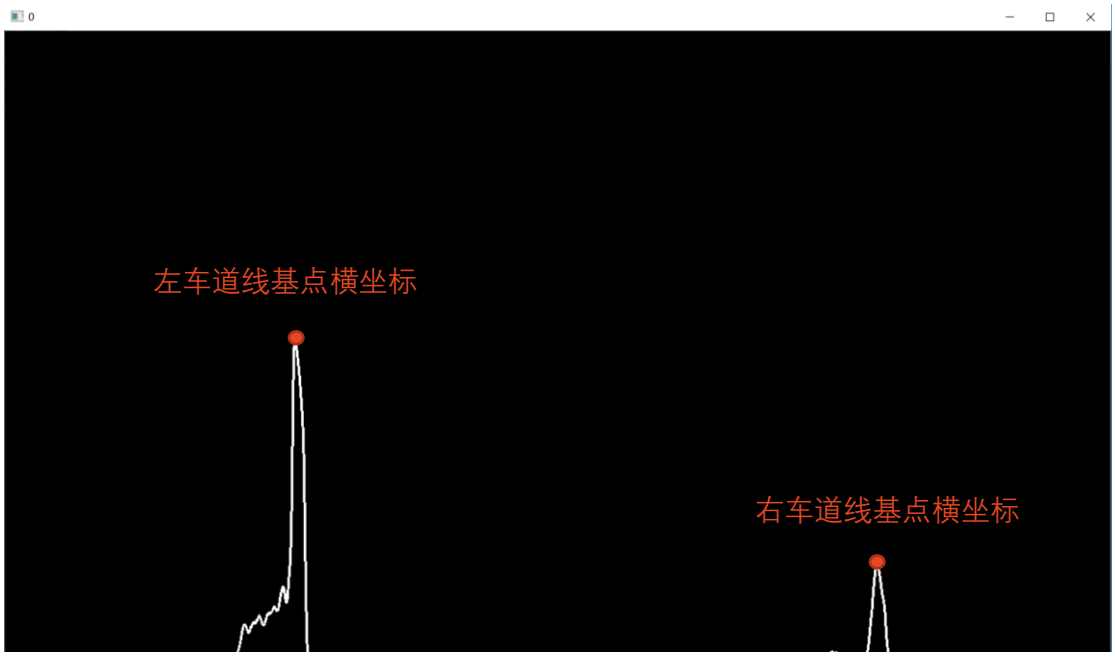
Main：

```
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);
```

再结合步骤二的透视变换，我们将通过代码 Perspective(TempImage, ThreshImage, Frame.cols, Frame.rows);得到互相平行的二值化车道线图像:

4. 实现步骤四 ：车道线识别 --- 找寻定位车道线

藉由步骤三中所获得的透视变换后的车道线二值图，再通过直方图 Histogram 来计算叠加显示图像对应坐标的白色像素点的数量值。左车道线的基点坐标即直方图中左半部分的峰值，对应的右车道线的基点即直方图中右半部分的峰值。直方图见下图：



直方图统计白色像素点代码：

```
for (int i = 0; i < 1280; i++) //统计数组清零
            {
                Array[i] = 0;
            }
            for (int row = 0; row < Frame.rows; row++)      //扫描行
            {
                for (int col = 0; col < Frame.cols; col++) //扫描列
                {
                    if (ThreshImage.at<uchar>(row, col) == 255) //统计每列白点的
个数
                    {
                        Array[col]++;
                    }
                }
            }

            for (int col = 0; col < Frame.cols; col++) //扫描列
            {
                line(DispImage, Point(col, 720 - Array[col]), Point(col + 1, 720
 - Array[col + 1]), Scalar(255), 2, LINE_8); //绘制直方图
```

```
                }
```

在掌握左右车道线基点坐标后，再使用滑窗式拟合 sliding window polynomial fitting 来定位车道线，在这里选择动用 12 个宽度为 150px 的滑窗。

具体代码:

```
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);  //指向窗口区域
            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);  //指
向窗口区域
            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;
}
```

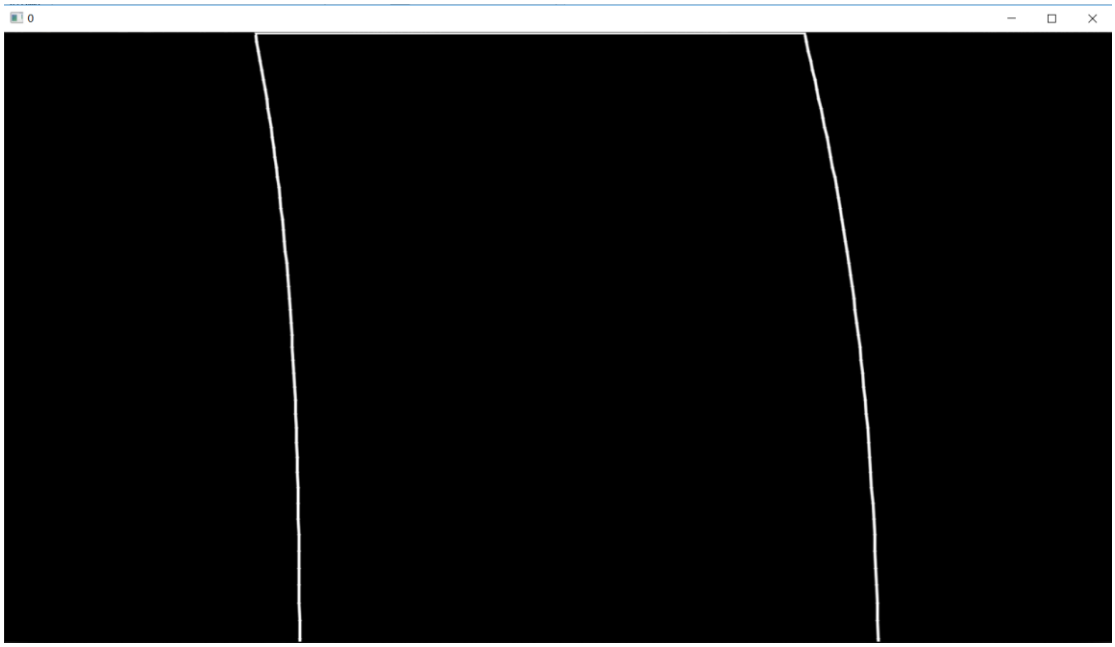
在找到所有曲线上的特征点后，需要进行筛选出三个点来做特征曲线拟合，此处我选用图像最上方最下方以及中心三个特征点，配合贝塞尔曲线拟合来实现车道线的绘制。

具体代码:

```
if (0 == i)  //保存此帧图片的倒数第二个滑动窗的中心坐标点的X值
        {
            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;
```

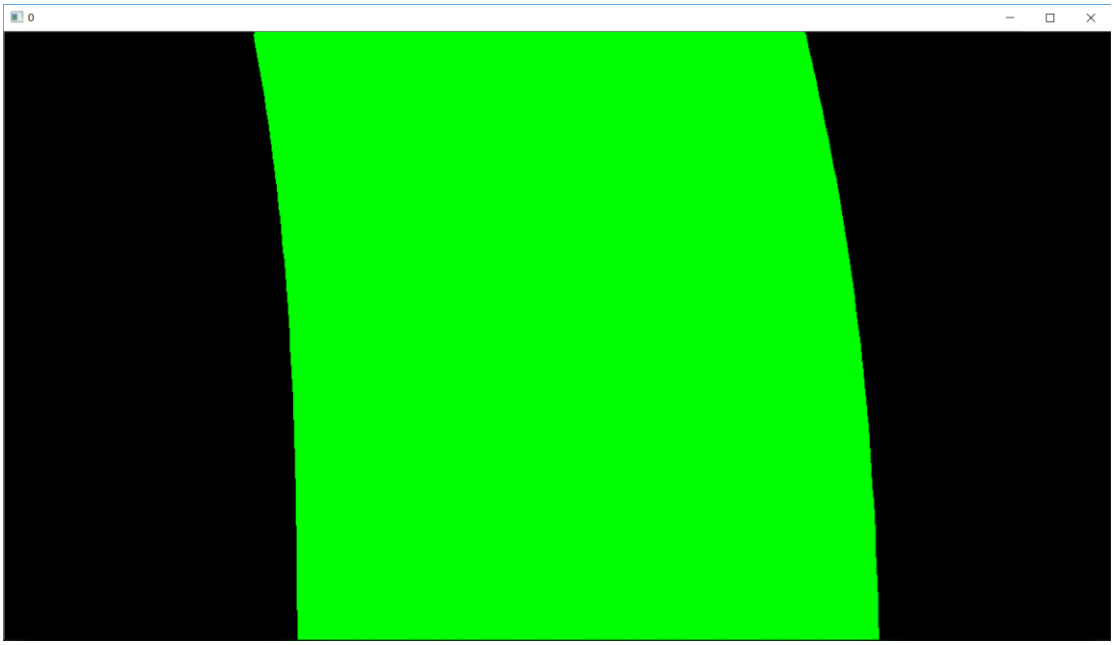
```
                    ControlPtsR[3].x = RightBase;
                    ControlPtsR[3].y = (720 - 60 * (i + 2));
                    ControlPtsR[3].z = 0;
                    line(WinSlip, Point(ControlPts[3].x, ControlPts[3].y),
Point(ControlPtsR[3].x, ControlPtsR[3].y), Scalar(255), 2, LINE_AA);
                }
            }
            DrawBezier(WinSlip, ControlPts); //相应子程序见报告末尾总代码
            DrawBezier(WinSlip, ControlPtsR); //相应子程序见报告末尾总代码
```

通过以上几步，我们将得到绘制好的弯道车道线二值图：



再通过该四边形的四个顶点形成的车道线轮廓填充颜色：

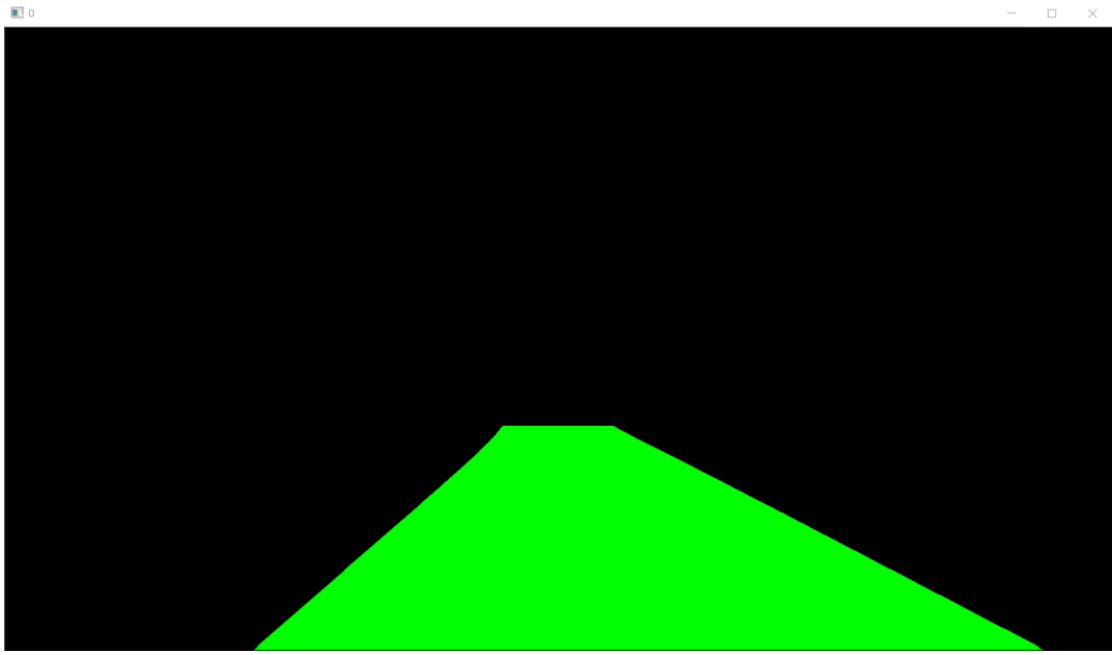
```
cv::findContours(WinSlip, Contours, Hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE,
Point(0, 0));
            cv::cvtColor(WinSlip, WinSlip, COLOR_GRAY2BGR);
            for (int i = 0; i < Contours.size(); i++)
            {
                drawContours(WinSlip, Contours, i, Scalar(0, 255, 0), -1,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



再通过逆透视变化（步骤二的相逆步骤）将得到的弯道轮廓图投影至摄像机实时摄取到的画面上去实现车道区域识别。

具体代码:

```
 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);  //逆透视变换
```



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

5. 实现步骤五 ：车道线识别 --- 曲率半径以及中心偏移距离运算

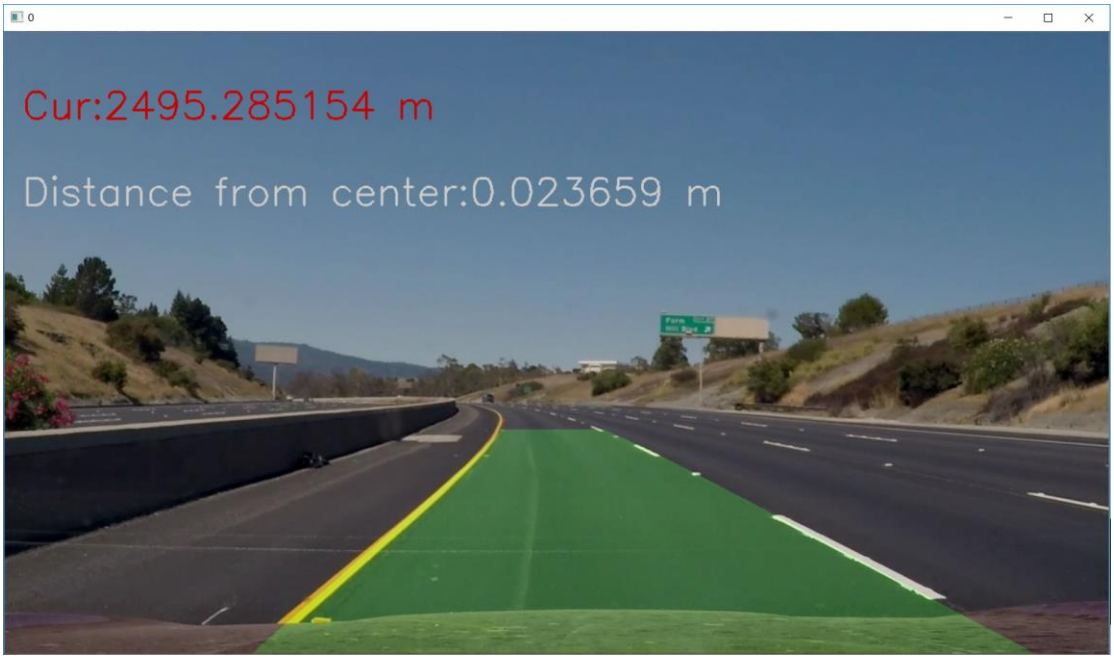
为了该项目后续更方便以及人性化的应用，此处也做了曲率半径的运算（方便车辆得知转弯的幅度）以及中心偏离距离运算（方便车辆维持在车道线中心）。

具体代码:

```
double Curvature(Point p1, Point p2, Point p3)
{
    double Cur; //求得的曲率
    double Radius = 0.0; //曲率半径
    cv::Point p0;
    if (1 == Collinear(p1, p2, p3)) //判断三点是否共线
    {
        Cur = 0.0;//三点共线时将曲率设为某个值 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);
```

显示至摄取到的图像上:



项目总结

1. 总代码

```
#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);

//两个向量相加，p=p+q
Point3d PointAdd(Point3d p,Point3d q)
{
    p.x += q.x;
    p.y += q.y;
    p.z += q.z;
    return p;
}

//向量和标量相乘 p=c*p
```

```
Point3d PointTimes(float c,Point3d p)
{
    p.x *= c;
    p.y *= c;
    p.z *= c;
    return p;
}

//计算贝塞尔方程的值
//变量u的范围在0-1之间
//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;
}

//绘制Bezier曲线
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; //两点间的距离
    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); //求平方根
    return Dis;
}

int Collinear(Point p1, Point p2, Point p3)  //判断3点是否共线，共线返回1
{
    double k1, k2;
    double kx1, ky1, kx2, ky2;
    if (p1.x == p2.x && p2.x == p3.x) //三点横坐标都相等，共线
    {
        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)   //AB与BC斜率相等，共线
        {
            return 1;
        }
        else
        {
            return 0;   //不共线
        }
    }
}

double Curvature(Point p1, Point p2, Point p3)
{
    double Cur; //求得的曲率
    double Radius = 0.0; //曲率半径
    cv::Point p0;
    if (1 == Collinear(p1, p2, p3)) //判断三点是否共线
    {
```

```
        Cur = 0.0;//三点共线时将曲率设为某个值 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;
    //转换成为灰度图片
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    //使用cv::Sobel()计算x方向或y方向的导
    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);
    }
    //二值化
    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);
    //分离通道
    cv::split(hls, channels);
    //选择通道
    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;
    }
    //二值化
    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);
    //分离通道
    cv::split(luv, channels);
    //选择通道
    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;
    }
    //二值化
    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;
```

```
    //转换成为灰度图片
    cv::cvtColor(src, src_gray, cv::COLOR_RGB2GRAY);
    //使用cv::Sobel()计算x方向或y方向的导
    cv::Sobel(src_gray, gray_x, CV_64F, 1, 0, sobel_kernel);
    cv::Sobel(src_gray, gray_y, CV_64F, 0, 1, sobel_kernel);
    //转换成CV_8U
    cv::convertScaleAbs(gray_x, abs_gray_x);
    cv::convertScaleAbs(gray_y, abs_gray_y);
    //合并x和y方向的梯度
    cv::addWeighted(abs_gray_x, 0.5, abs_gray_y, 0.5, 0, grad);
    //二值化
    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);  //显示车道线
    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;  //三个点用于计算曲率半径
    uchar* CurrRowP;


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

        //阈值过滤
        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);  //二值化后的左右车道线合并
        Perspective(TempImage, ThreshImage, Frame.cols, Frame.rows);  //透视变换
        //清空图像显示
        ImageDest = Mat::zeros(Frame.size(), CV_8UC3);
        WinSlip = Mat::zeros(720, 1280, CV_8UC1);
        Mat Test = Mat::zeros(720, 1280, CV_8UC1);


        //统计每列白点的个数
        if (0 == Lock)  //第一次进来
        {
            Lock = 1; //只在计算第一帧图像时使用
            for (int i = 0; i < 1280; i++) //统计数组清零
            {
                Array[i] = 0;
            }
            for (int row = 0; row < Frame.rows; row++)     //扫描行
            {
                for (int col = 0; col < Frame.cols; col++) //扫描列
                {
                    if (ThreshImage.at<uchar>(row, col) == 255) //统计每列白点的
个数
                    {
                        Array[col]++;
                    }
                }
            }
```

```
            for (int col = 0; col < Frame.cols; col++) //扫描列
            {
                line(DispImage, Point(col, 720 - Array[col]), Point(col + 1, 720
- Array[col + 1]), Scalar(255), 2, LINE_8); //绘制直方图
            }

            //查找直方图峰值对应的列，并记录该列
            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;  //重新赋予一个默认值,去除上一段程序赋值带来的影响
            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;  //第一帧图像，利用直方图找到左车道线的粗
略位置
            ControlPts[0].y = 720;
            ControlPts[0].z = 0;
```

```
            ControlPtsR[0].x = RightBase;//第一帧图像，利用直方图找到右车道线的粗略位置
            ControlPtsR[0].y = 720;
            ControlPtsR[0].z = 0;
        }
        else
        {
            ControlPts[0].x = LeftBaseTemp; //除第一帧外,以后每帧图像的最下方的滑动窗的基准点均以上一帧图像最下方的第二个滑动窗的基准点为准（因为每两帧之间车道线不会突变，这样处理后，检测车道线更加稳定）
            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); //二值化图像转化为3通道

            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);  //指向窗口区域
                    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);  //指向窗口区域
                    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;
                }

                //贝塞尔曲线拟合特征点选择
                if (0 == i)  //保存此帧图片的倒数第二个滑动窗的中心坐标点的X值
                {
                    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);
            //轮廓发现
            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);  //
透视变换
            //计算曲率
            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. 遗留问题

- 图像实时性不高
- 小部份时间的曲线探测不精确
- 由于摄像头以及实验车的缺失无法进行代码的实战测试

3. 项目应用

高级驾驶辅助系统 ADAS 系统中的车道线偏移警报系统 LDWS 以及车道保持系统 LCA。