实验名称	大作业——基于霍夫变换的车道线检测		
学号	1120180484	姓名	付宇

一、实验目的

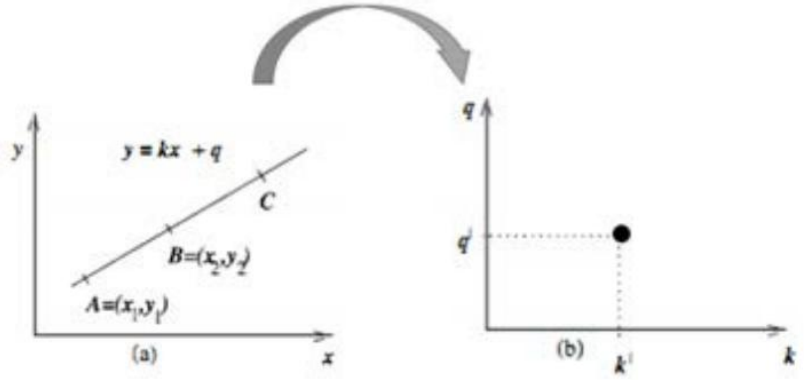
通过了解霍夫变换的基本原理，掌握通过极坐标将原空间转化为霍夫空间记录检测到的直线，最终将检测到的车道线标记出来。

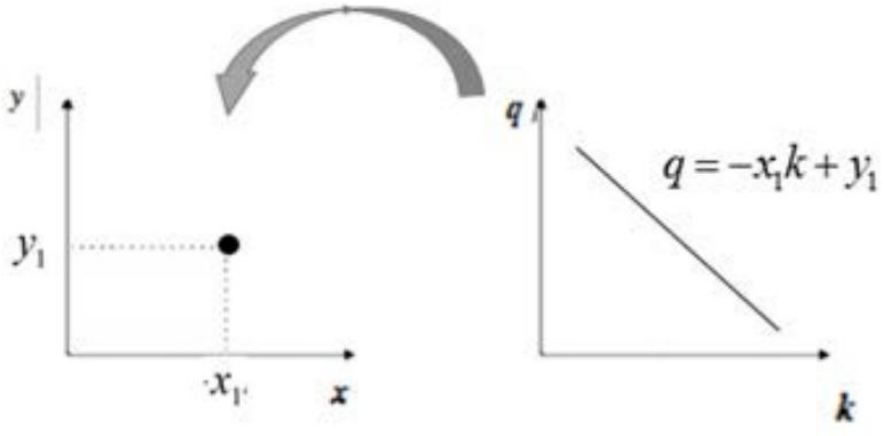
二、实验原理

首先我们先介绍一下基本的霍夫变换的原理。

在我们熟悉的 X-Y 坐标系中，我们将一条直线表示为 y=ax+b，其中 a 代表斜率，b 代表截距。我们知道通过点(x0，y0)的直线可以有无数条，每一条对应不同的斜率和截距。那么，我们可以将上述直线的表达式做出转化，将斜率 a 和截距 b 作为变量进行建系，得到 A-B 坐标系。此时，式子就变化为：b=-xa+y。

通过这一变换，我们可以得到在 X-Y 坐标系中的两点（x1，y1)、(x2，y2）确定了一条直线，这条直线转化到 A-B 坐标空间中对应一个点，即斜率为 a=（y2-y1）/（x2-x1）的截距（同样可由两点求得）为 b 的一个点。同理，在 A-B 坐标系中两条直线相交得到的交点对应着 X-Y 坐标空间中的一条直线。





同样的，在初中我们就学到，在 X-Y 坐标系中，是无法表示平行于 Y 坐标轴的直线的。那么，我们同样引入极坐标系将该问题化解。

有了霍夫变换的基本知识，我们就可以将他运用到实践当中去。

本次实验，经过查阅资料，我们先通过加权平均法将彩色图转化为灰度图，该方法是根据人的亮度感知系统进行调节，得到 R、G、B 三个维度占有的比例（分别是 0.3、0.59、0.11）进行调和，得到的图形近似看作灰度图，即 I(x,y) = 0.3 * I_R(x,y) +0.59 * I_G(x,y)+ 0.11 * I_B(x,y)

得到灰度图后，根据检测车道线的实际情况，我们对图像下本部分进行检测，以避免上半部分的直线对实验结果进行干扰。

随后，我们通过高斯滤波，降低噪声对实验结果的影响程度。基本的高斯滤波的原理就是通过高斯滤波算子对图片进行卷积，就可以获得较为平滑的图片。

之后，我们通过将图片二值化，并通过某个阈值获得车道线并保留下来，在此基础上进行边界提取，获得检测到的车道线的边界，最后在进行霍夫变换，保留检测到的直线，在原图中进行绘制。

最后，我们就可以得到检测到车道线的实验图。

三、实验步骤

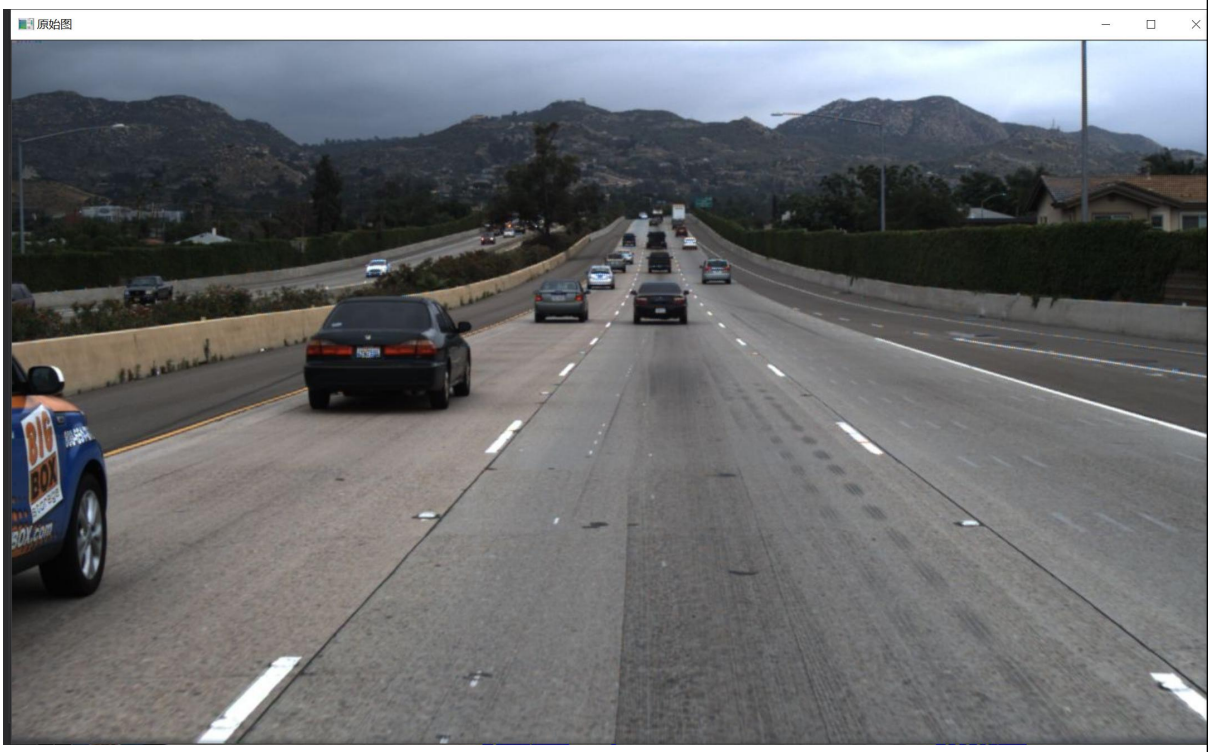
1. 编写转化为灰度图的函数 IamgeGraying(Mat src, Mat img)
2. 编写获取高斯算子的函数 GaussMark(Size size, double sigma)
3. 编写进行卷积的函数 gaussian(Mat* _src, Mat mask, int _size)
4. 编写提取边界的函数 BoundaryExtraction(Mat src)
5. 编写进行霍夫变换的函数 HoughTransform(Mat img, int threshold)

6. 通过上述实验原理的基本步骤对图片进行处理

在文末粘贴上完整代码。

四、实验结果

通过上述步骤处理后，我们可以分别得到灰度图、平滑灰度图、二值图、边界图、最终处理图，我们以某一张实验图片为例：

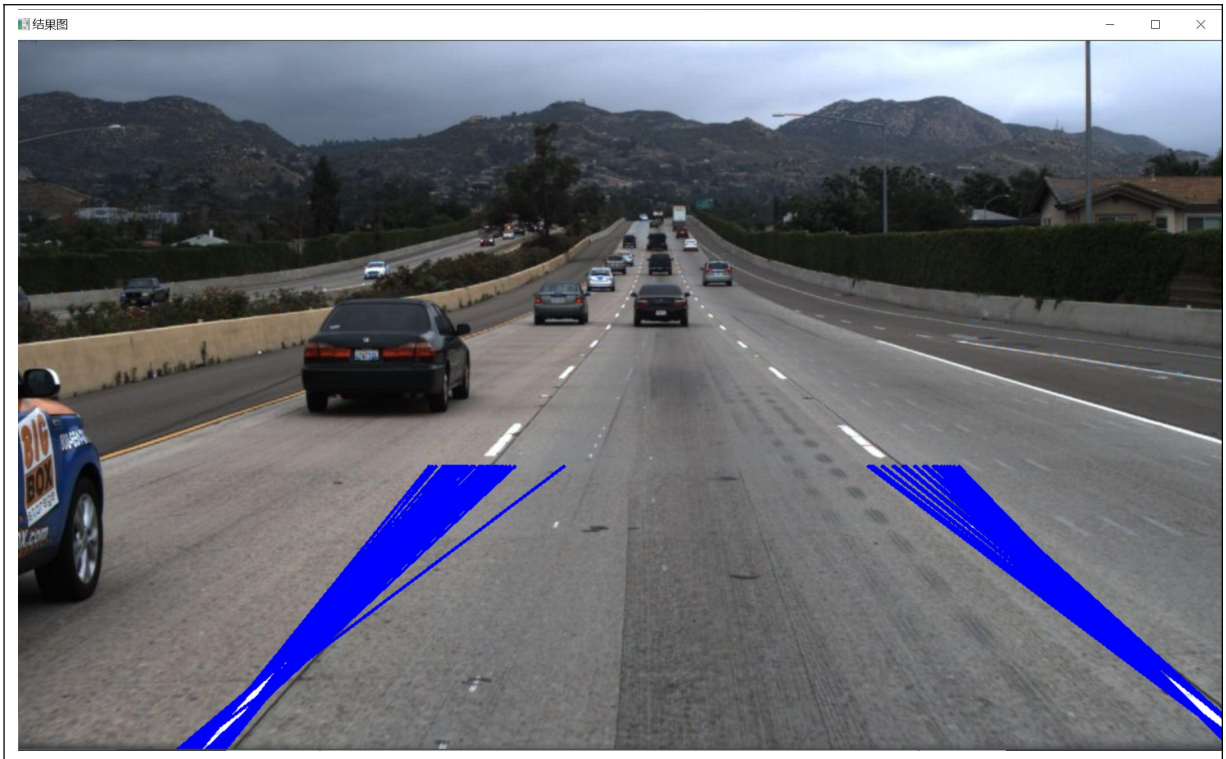



灰度图


高斯滤波后平滑图


二值灰度图


边界图



五、实验分析

通过以上实验结果，我们可以得到通过霍夫变换检测直线可以获得最直线检测较好的结果，途中出现的多条直线表示了对边界的提取存在一些误差，例如地上的车道线边缘有一些凹凸不平的部分（例如上图的示例图片），导致直线的绘制上出现了无法完全重合的情况。

同时，采用霍夫变换的方法可以对直线做出较好效果的检测，但是对弯曲的车道线只能将进出的部分近似视为直线进行检测，不能对较远处但仍处于视线内的弯道车道线进行良好的检测。通过这种方式，我们对其他数据集中的图片进行检测并进行评分。

下面为代码：

```
#include<iostream>
#include<math.h>
#include<string.h>
#include<stdlib.h>
#include <memory.h>
#include <vector>
```

```
#include<opencv2/opencv.hpp>

#define PI 3.1415926
using namespace std;
using namespace cv;

void IamgeGraying(Mat src, Mat img)//利用加权平均法获得灰度图
{
    float R;
    float G;
    float B;

    for (int y = 0; y < src.rows; y++)
    {
        uchar* data = img.ptr<uchar>(y);
        for (int x = 0; x < src.cols; x++)
        {
            R = src.at<Vec3b>(y, x)[2];
            B = src.at<Vec3b>(y, x)[0];
            G = src.at<Vec3b>(y, x)[1];
            data[x] = (int)(R * 0.3 + G * 0.59 + B * 0.11);
        }
    }
}

void ThreShold(Mat src, Mat img) //灰度图二值化
{
    float start_i = 0.6 * float(src.rows); //获得0.6倍高度一下的处理图片

    int average = 0;
    for (int y = start_i; y < src.rows; y++)
    {
        for (int x = 0; x < src.cols; x++)
        {
            if (src.at<uchar>(y, x) > average) average = src.at<uchar>(y, x);
        }
    }
    average = average - 50;//通过调整阈值获得更好的检测效果

    for (int y = 0; y < src.rows; y++)
    {
        for (int x = 0; x < src.cols; x++)
        {
            uchar* data = img.ptr<uchar>(y);
```

```
            if (src.at<uchar>(y, x) > average && y >= start_i && y < src.rows)//在检测范围并且通过阈值判断如何进行二值化
            {
                data[x] = 255;
            }
            else {
                data[x] = 0;
            }
        }
    }
}

Mat GaussMark(Size size, double sigma) //根据指定滤波器大小和sigma生成滤波器
{
    Mat mask;
    mask.create(size, CV_64F);
    int h = size.height;
    int w = size.width;

    int center_h = h / 2;
    int center_w = w / 2;

    double sum = 0;
    double x;
    double y;

    int i, j;

    for (i = 0; i < h; ++i)
    {
        y = pow(i - center_h, 2);
        for (j = 0; j < w; ++j)
        {
            x = pow(j - center_w, 2);
            double s = (exp(-(x + y) / (2 * sigma * sigma))) / (2 * PI * sigma * sigma); //二维高斯分布实现
            mask.at<double>(i, j) = s;
            sum += s;
        }
    }
    mask = mask / sum; //归一化处理

    return mask;
}
```

```
Mat GaussianFilter(Mat img, Mat mark) //高斯平滑滤波处理
{
    Size size = img.size(); //图片大小
    Mat gaussian_img;
    gaussian_img = Mat::zeros(img.size(), img.type());

    int extend_h = mark.rows / 2;
    int extend_w = mark.cols / 2;

    Mat extend_img;
    copyMakeBorder(img, extend_img, extend_h, extend_h, extend_w, extend_w, BORDER_REPLICATE);
//对图片边缘进行拓展以生成相同大小的图片

    int i, j;

    for (i = extend_h; i < img.rows + extend_h; i++) //图片的每一排像素进行遍历
    {
        for (j = extend_w; j < img.cols + extend_w; j++) //图像的每一列像素进行遍历
        {
            double sum[3] = { 0.0 };

            for (int r = -extend_h; r <= extend_h; r++) //滤波处理
            {
                for (int c = -extend_w; c <= extend_w; c++)
                {
                    Vec3b rgb = extend_img.at<Vec3b>(i + r, j + c); //对读取的三通道图片进行
处理
                    sum[0] += rgb[0] * mark.at<double>(r + extend_h, c + extend_w);
                    sum[1] += rgb[1] * mark.at<double>(r + extend_h, c + extend_w);
                    sum[2] += rgb[2] * mark.at<double>(r + extend_h, c + extend_w);
                    //sum = sum + gaussian_img.at<uchar>(i + r, j + c) * mark.at<double>(r +
extend_h, c + extend_w);
                }
            }

            for (int k = 0; k < img.channels(); k++) //防止处理后数据超出[0, 255]的范围
            {
                if (sum[k] < 0)
                {
                    sum[k] = 0;
                }
                else if (sum[k] > 255)
                {
```

```
                    sum[k] = 255;
                }
            }

            Vec3b rgb =
{ static_cast<uchar>(sum[0]),static_cast<uchar>(sum[1]),static_cast<uchar>(sum[2]) };
            gaussian_img.at<Vec3b>(i - extend_h, j - extend_w) = rgb;
        }
    }

    return gaussian_img;

}



void gaussian(Mat* _src, Mat mask, int _size)
{
    Mat temp = (*_src).clone();

    double sum = 0.0;
    // 遍历图片像素
    for (int i = 0; i < (*_src).rows; i++)
    {
        for (int j = 0; j < (*_src).cols; j++)
        {
            //不对边缘进行处理，避免越界的情况
            if (i > (_size / 2) - 1 && j > (_size / 2) - 1 && i < (*_src).rows - (_size / 2) &&
j < (*_src).cols - (_size / 2))
            {
                sum = 0.0;
                for (int k = 0; k < _size; k++)
                {
                    for (int l = 0; l < _size; l++)
                    {
                        sum += (*_src).ptr<uchar>(i - k + (_size / 2))[j - l + (_size / 2)] *
mask.at<double>(k, l);
                    }
                }
                // 放入中间结果,计算所得的值与没有计算的值不能混用
                temp.ptr<uchar>(i)[j] = sum;
            }
        }
    }
    // 放入原图
```

```
    (*_src) = temp;
}



void Gaussian(Mat* src, Size size, double sigma)
{
    Mat array = GaussMark(size, sigma); //获取高斯算子数组

    int n = size.height;
    gaussian(src, array, n);
    return;
}

Mat BoundaryExtraction(Mat src) //进行边界提取
{
    Mat src_clone(src.rows, src.cols, CV_8UC1);

    for (int y = 1; y < src.rows - 1; y++)
    {
        uchar* data = src_clone.ptr<uchar>(y);
        for (int x = 1; x < src.cols - 1; x++)
        {
            int flag = 0;
            for (int i = -1; i < 2; i++)
            {
                for (int j = -1; j < 2; j++)
                {
                    if (src.at<uchar>(y + i, x + j) == 0) //通过梯度对边界进行获取
                    {
                        flag = 1;
                        break;
                    }
                }
            }
            if (flag == 0)
            {
                data[x] = 255;
            }
            else data[x] = 0;
        }
    }
    //将获取的边界坐标进行还原
    for (int y = 0; y < src.rows; y++)
    {
```

```
            uchar* data = src_clone.ptr<uchar>(y);
            for (int x = 0; x < src.cols; x++)
            {
                data[x] = src.at<uchar>(y, x) - src_clone.at<uchar>(y, x);
            }
        }
        return src_clone;
}

vector<float> HoughTransform(Mat img, int threshold)
{
        int row, col;
        int i, k;

        int angle, p;//极坐标空间的参数极角angle，极径p

        //设置累加器对直线进行筛选
        int** socboard;
        int* buf;
        int w, h;
        int Size;
        int offset;

        w = img.cols;
        h = img.rows;

        vector <float> lines;
        //申请累加器空间并初始化
        Size = w * w + h * h;
        Size = 2 * sqrt(Size) + 100;
        offset = Size / 2;

        socboard = (int**)malloc(Size * sizeof(int*));
        if (!socboard)
        {
            printf("Memery apply error.\n");
            return lines;
        }

        for (i = 0; i < Size; i++)
        {
            socboard[i] = (int*)malloc(181 * sizeof(int));
            if (socboard[i] == NULL)
            {
```

```
            printf("Buffer apply error.\n");
            return lines;
        }
        memset(socboard[i], 0, 181 * sizeof(int));
    }

    //遍历图像并进行计数，得数搞得将被选中并绘制直线
    uchar src_data;
    p = 0;
    for (row = 0; row < img.rows; row++)
    {
        for (col = 0; col < img.cols; col++)
        {
            //遍历像素点
            src_data = img.at <uchar>(row, col);

            if (src_data == 255)
            {

                for (angle = 0; angle < 181; angle++)
                {
                    p = col * cos(angle * PI / 180.0) + row * sin(angle * PI / 180.0) + offset;

                    if (p < 0)
                    {
                        printf("at (%d,%d),angle:%d,p:%d\n", col, row, angle, p);
                        printf("warrning!");
                        printf("size:%d\n", Size / 2);
                        continue;
                    }
                    socboard[p][angle]++;
                }
            }
        }
    }

    //遍历得数，选出的书超过阈值的直线
    int count = 0;
    int Max = 0;
    int kp, kt, r;
    kp = 0;
    kt = 0;
    for (i = 0; i < Size; i++)//对极径进行遍历
    {
```

```
        for (k = 0; k < 181; k++)//对角度进行遍历
        {
            //通过遍历所有极坐标方程对应得数进行选择
            if (socboard[i][k] > Max)
            {
                Max = socboard[i][k];
                kp = i - offset;
                kt = k;
            }

            if (socboard[i][k] >= threshold)
            {
                r = i - offset;
                lines.push_back(-1.0 * float(std::cos(k * PI / 180) / std::sin(k * PI / 180)));
                lines.push_back(float(r) / std::sin(k * PI / 180));
                count++;
            }
        }
    }
    //释放资源
    for (int e = 0; e < Size; e++)
    {
        free(socboard[e]);
    }
    free(socboard);
    return lines;
}


int main()
{
    Mat Mask = GaussMark(Size(5, 5), 0.8);

    vector<float>lines;

    Mat src = imread("11.jpg", 1);
    imshow("原始图", src);

    Mat src_clone = src;

    float detect_range = 0.6 * float(src.rows);

    Mat gray_scale(src.rows, src.cols, CV_8UC1);//大小与原图相同的八位单通道图
    Mat hdimg = src;
```

```
    IamgeGraying(src, gray_scale);
    imshow("灰度图", gray_scale);

    hdimg = gray_scale;
    Gaussian(&hdimg, Size(5,5), 1.5f);
    imshow("高斯滤波后平滑图", hdimg);


    ThreShold(hdimg, gray_scale);// 灰度图二值化
    imshow("二值灰度图", gray_scale);

    gray_scale = BoundaryExtraction(gray_scale);
    imshow("边界图", gray_scale);

    lines = HoughTransform(gray_scale, 25);

    for (int i = 0; i < lines.size(); i = i + 2)
    {
        float k = lines[i];
        float b = lines[i + 1];
        if (k == 0)
        {
            continue;
        }

        int y1 = int((detect_range - b) / k);
        int y2 = int((src.rows - b) / k);
        Point p1(y1, int(detect_range));
        Point p2(y2, int(src.rows));

        if (y1 >= 0 && y1 < src.cols && y2 >= 0 && y2 < src.cols)
        {
            line(src_clone, p1, p2, Scalar(255, 0, 0), 2);
        }
    }

    imshow("结果图", src_clone);

    waitKey(0);
    return 0;
}
```