浙江大学

课程名称：计算机视觉

姓　　名：刘佳润

学　　院：计算机科学与技术学院

专　　业：数字媒体技术

学　　号：3180105640

指导教师：潘纲

2020 年 12 月 9 日

浙江大学实验报告

课程名称：计算机视觉 实验类型：综合

实验项目名称：直线检测、圆检测

学生姓名：刘佳润 专业：数字媒体技术 学号：3180105640

同组学生姓名： 指导老师：潘纲

实验地点： 实验日期：2020 年 11 月 28 日

一、 实验内容和要求

输入一张彩色图像，要求能够检测出其中的直线、圆。

二、 实验器材

C++ OpenCV 4.5.0

开发平台：Visual Studio 2019 Debug x64

三、 具体实现

1. 直线检测——利用 Hough 变换

- 构造 Hough 空间

直线检测的 Hough 空间是对直角坐标系下的(x, y)映射到极坐标系下的(ρ, θ)后，以ρ和θ为轴建立的坐标系空间。映射关系如下图所示：

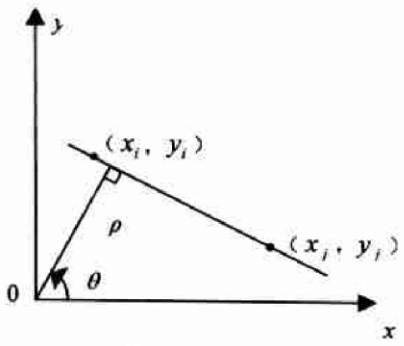


图　直线的参数式表示

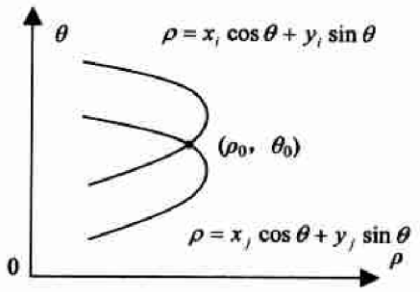


图　笛卡儿坐标映射到参数空间

对于一副大小为 D×D 的图像，通常 ρ 的取值范围为$[-2\sqrt{D}/2, 2\sqrt{D}/2]$,$\theta$ 的取值范围为[−90◦,90◦]。计算方法与直角坐标系中累加器的计算方法相同，最后得到最大的 A 所对应的(ρ,θ)。累加器的构造代码如下：

```
// build the accumulator
int RMax = cvRound(sqrt(2.0) * (img_binary.rows > img_binary.cols ? img_binary.rows : img_binary.cols) / 2.0);
float** accu;
accu = new float* [2 * RMax];
for (int i = 0; i < 2 * RMax; i++)
{
    accu[i] = new float[180];
}
for (int i = 0; i < 2 * RMax; i++)
{
    for (int j = 0; j < 180; j++)
    {
        accu[i][j] = 0;
    }
}
```

- 边缘检测

我自己实现了 Sobel 算子的检测，但是在面对比较复杂的图形时，效果不是特别好。在此处将 Sobel 边缘检测和 Canny 边缘检测都写出来，并比较结果。

Sobel 算子采用了边缘中心权值为 2 的版本，分别对 x 和 y 方向做了卷积，并且计算了方向。

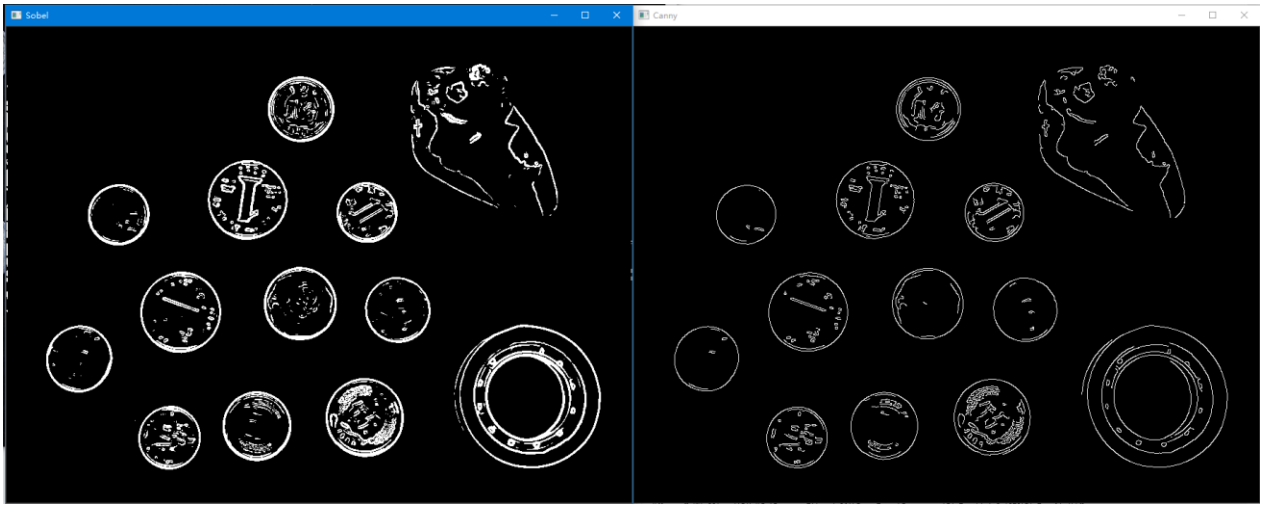
```
// Edge extraction using Sobel kernal
void edgeExtract(Mat img, Mat& mag, Mat& dist)
{
    float acc_dx = 0, acc_dy = 0;           // accumulators
    float k1[] = { -1,-2,-1, 0,0,0, 1,2,1 };  // {-2,-4,-2,0,0,0,2,4,2};    // sobel kernal dx
    float k2[] = { -1,0,1,-2,0,2,-1,0,1 };  // {-2,0,2,-4,0,4,-2,0,2};    // sobel kernal dy

    for (int i = 0; i < img.rows; i++)
    {
        for (int j = 0; j < img.cols; j++)
        {
            acc_dx = acc_dy = 0;
            for (int n = -1; n < 2; n++)
            {
                for (int m = -1; m < 2; m++) {
                    if (i + n > 0 && i + n < img.rows && j + m > 0 && j + m < img.cols) {
                        acc_dx += (float)img.at<uchar>(i + n, j + m) * k1[((m + 1) * 3) + n + 1];
                        acc_dy += (float)img.at<uchar>(i + n, j + m) * k2[((m + 1) * 3) + n + 1];
                    }
                }
            }
            // generate a binary image with edge extracted.
            // thres = 100
            mag.at<float>(i, j) = (sqrtf(acc_dy * acc_dy + acc_dx * acc_dx)) > 100 ? 255 : 0;
            dist.at<float>(i, j) = atan2f(acc_dy, acc_dx);
        }
    }
}
```

Canny 边缘检测使用了 OpenCV 自带的函数：先进行一次高斯平滑，掩膜的大小由图像本身大小决定，然后用 Canny 函数进行检测。

```
GaussianBlur(img_grey, img_grey, Size(5, 5), 0, 0);
Canny(img_grey, mag, 100, 200);
```

对同一张图片进行两种方法滤波，结果如下：



左图为自己实现的 Sobel 算子检测结果，运算速度一般；右图为平滑滤波+Canny 检测的结果，运算速度很快。

- 对边缘点统计，投票，更新累加器

对检测完毕的边缘图像中的“边缘点”，在每一个 θ 进行遍历，计算出对应的 ρ，投票（对应在 Hough 空间的累加器里加 1）。

```
for (int y = 0; y < img_binary.rows; y++)
{
    for (int x = 0; x < img_binary.cols; x++)
    {
        if ((float)img_binary.at<uchar>(y, x) > 250.0)
        {
            Point2f edge_point(x - img_binary.cols / 2, y - img_binary.rows / 2);
            for (int theta_index = 0; theta_index < 180; theta_index++)
            {
                int rho = cvRound((edge_point.x * cos_thetas[theta_index]) + (edge_point.y * sin_thetas[theta_index]));
                int theta = thetas[theta_index];
                accu[rho + RMax][theta_index]++;
            }
        }
    }
}
```

- 按照一定的阈值进行输出和绘制

遍历 Hough 空间（累加器），对于最多的交点（也就是比较大的值）处，提取 ρ 和 θ 还原原直线信息，并且绘制在图上。

结果分析与改进请见第四部分。

2. 圆检测——利用 Hough 变换

一个圆的确定需要三个参数：两个参数用来确定圆心，一个参数用来确定半径。

- 构建 Hough 空间

圆检测的 Hough 累加器可以理解成一个三维数组（空间盒），长宽是原图像的长宽，深度是对半径的预测和累计。对于圆检测首先规定了一个半径的范围，只检测半径长度在该范围内的圆。

```
// accumulator accu[HEIGHT][WIDTH][DEPTH]
double*** accu;

accu = new double** [img_binary.rows];
for (int i = 0; i < img_binary.rows; ++i)
{
    accu[i] = new double* [img_binary.cols];

    for (int j = 0; j < img_binary.cols; ++j)
        accu[i][j] = new double[radiusRange];
}
for (int i = 0; i < img_binary.rows; ++i)
{
    for (int j = 0; j < img_binary.cols; ++j)
    {
        for (int k = 0; k < radiusRange; k++)
            accu[i][j][k] = 0;
    }
}
```

- 遍历边缘检测图像，统计与投票

对于边缘点，遍历所有可能的半径 r，根据圆的参数方程进行解析，得出两个可以确定圆心的点，对相应的累加器加 1。

```
// Go through the binary image
for (int y = 0; y < img_binary.rows; y++)
{
    for (int x = 0; x < img_binary.cols; x++)
    {
        if ((float)img_binary.at<float>(y, x) > 250.0)  //threshold image
        {
            for (int r = minRadius; r < radiusRange; r++)
            {

                int x0 = cvRound(x + r * cos(dist.at<float>(y, x)));
                int x1 = cvRound(x - r * cos(dist.at<float>(y, x)));
                int y0 = cvRound(y + r * sin(dist.at<float>(y, x)));
                int y1 = cvRound(y - r * sin(dist.at<float>(y, x)));
                // voting
                inc_if_inside(accu, x0, y0, img_binary.rows, img_binary.cols, r);
                inc_if_inside(accu, x1, y1, img_binary.rows, img_binary.cols, r);
            }
        }
    }
}
```

- 根据阈值还原检测圆

```
vector<Point3f> bestCircles;
// Compute best circles
for (int y0 = 0; y0 < img_binary.rows; y0++)
{
    for (int x0 = 0; x0 < img_binary.cols; x0++)
    {
        for (int r = minRadius; r < radiusRange; r++)
        {
            // Decide the center by thresholding the h_space
            if (accu[y0][x0][r] > threshold)
            {
                Point3f circle(x0, y0, r);
                int i;
                for (i = 0; i < bestCircles.size(); i++)
                {
                    int xCoord = bestCircles[i].x;
                    int yCoord = bestCircles[i].y;
                    int radius = bestCircles[i].z;
                    if (abs(xCoord - x0) < distance && abs(yCoord - y0) < distance)
                    {
                        if (accu[y0][x0][r] > accu[yCoord][xCoord][radius])
                        {
                            bestCircles.erase(bestCircles.begin() + i);
                            bestCircles.insert(bestCircles.begin(), circle);
                        }
                        break;
                    }
                }
                if (i == bestCircles.size()) {
                    bestCircles.insert(bestCircles.begin(), circle);
                }
            }
        }
    }
}
```

在筛选与重现的过程中做了一次根据距离的筛选，即两个圆的圆心之间的距离，以防止很多重复。

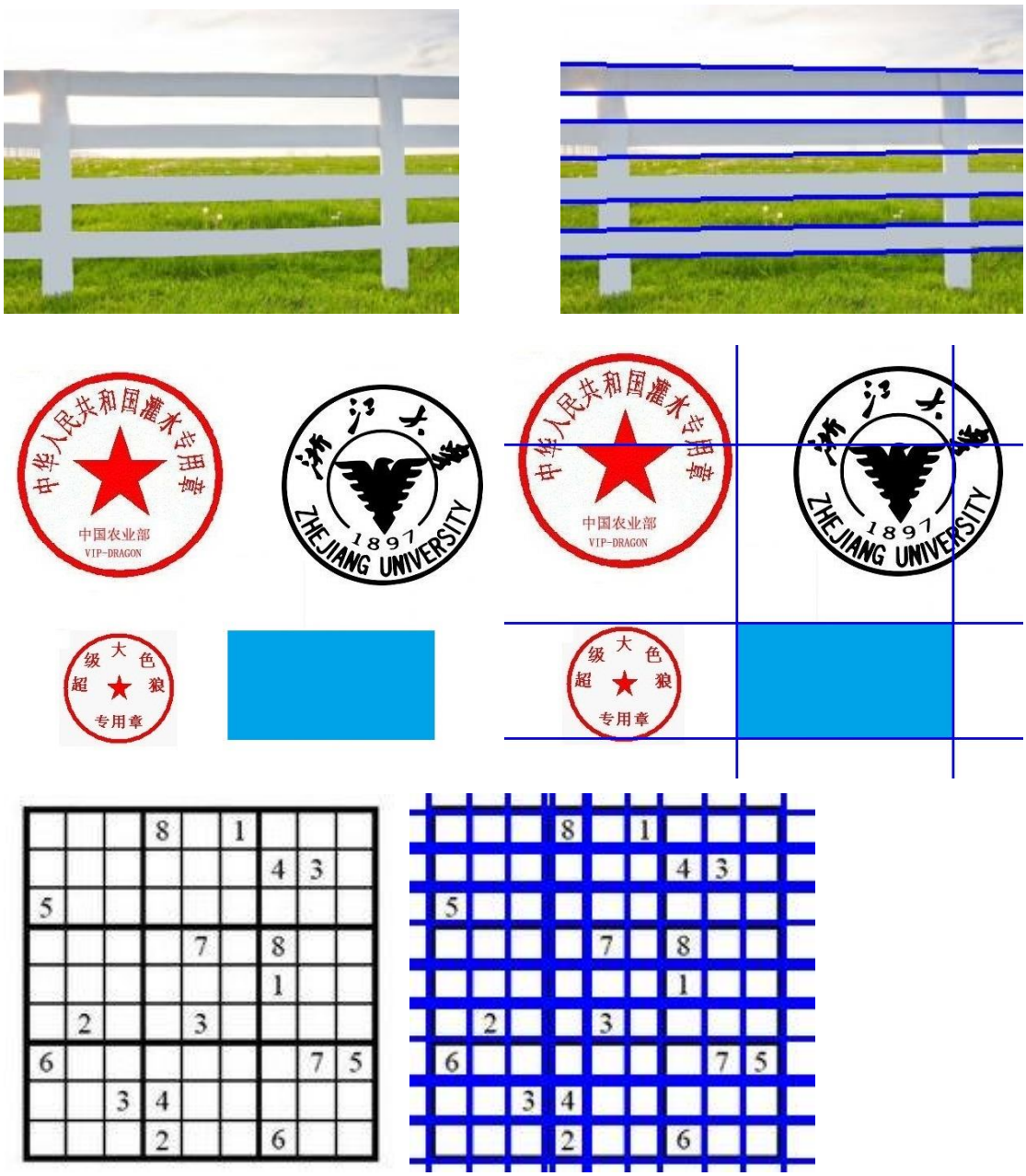
四、 实验结果与分析

1. 直线检测分析

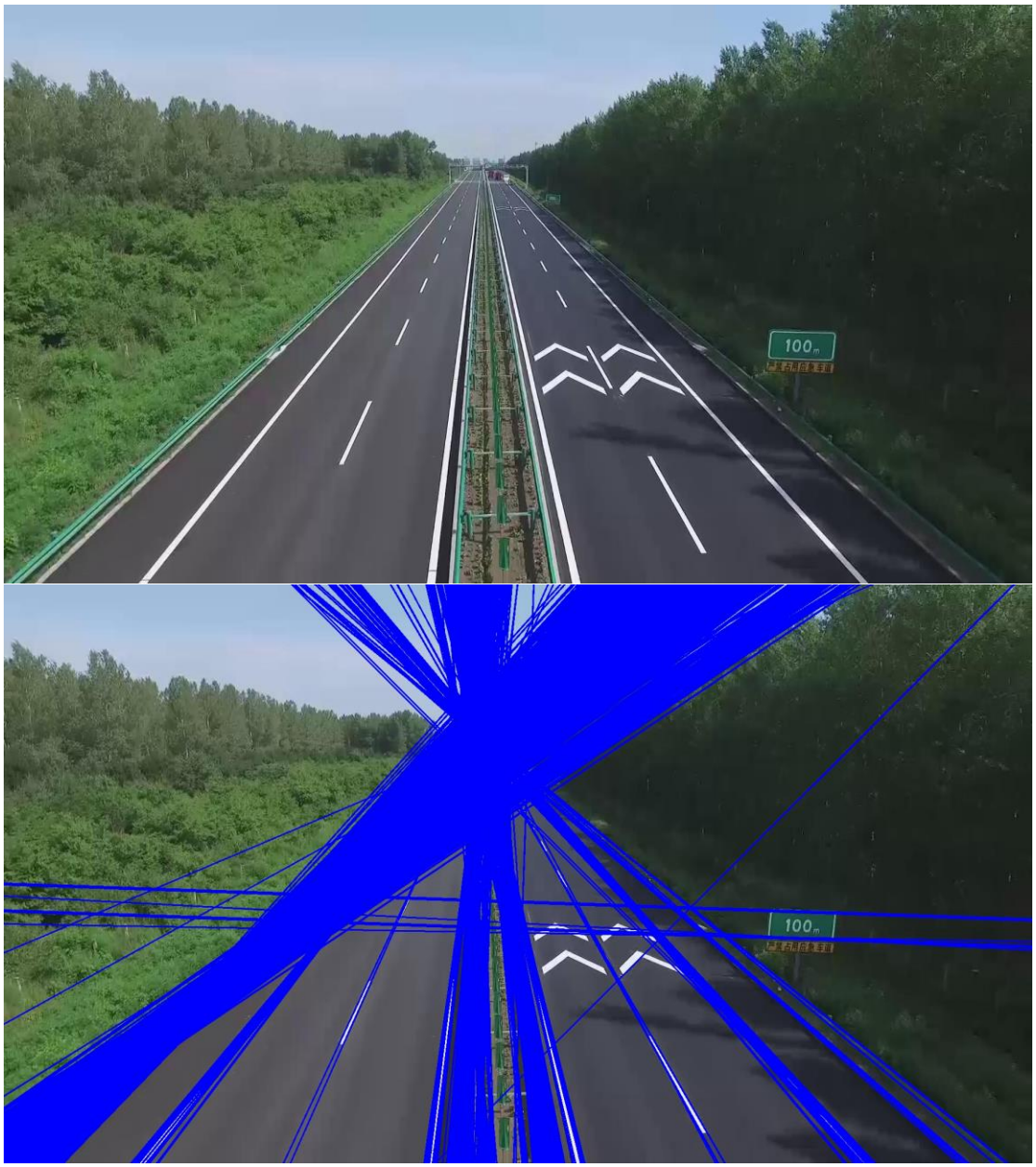
遍历全图的计算量很大，主要是在每一个需要检测的点的余弦值和正弦值的计算，因此在计算之前先对 0 到 180 的所有角度的余弦值和正弦值做了哈希表，用于后面的直接索引。这样一来加速了算法。

```
int thetas[180];
for (int i = 0; i < 180; i++)
    thetas[i] = i;
float cos_thetas[180], sin_thetas[180];
for (int i = 0; i < 180; i++)
{
    cos_thetas[i] = cos(thetas[i] / 180.0 * CV_PI);
    sin_thetas[i] = sin(thetas[i] / 180.0 * CV_PI);
}
```

在对一般图片的检测中，呈现出的结果比较良好：



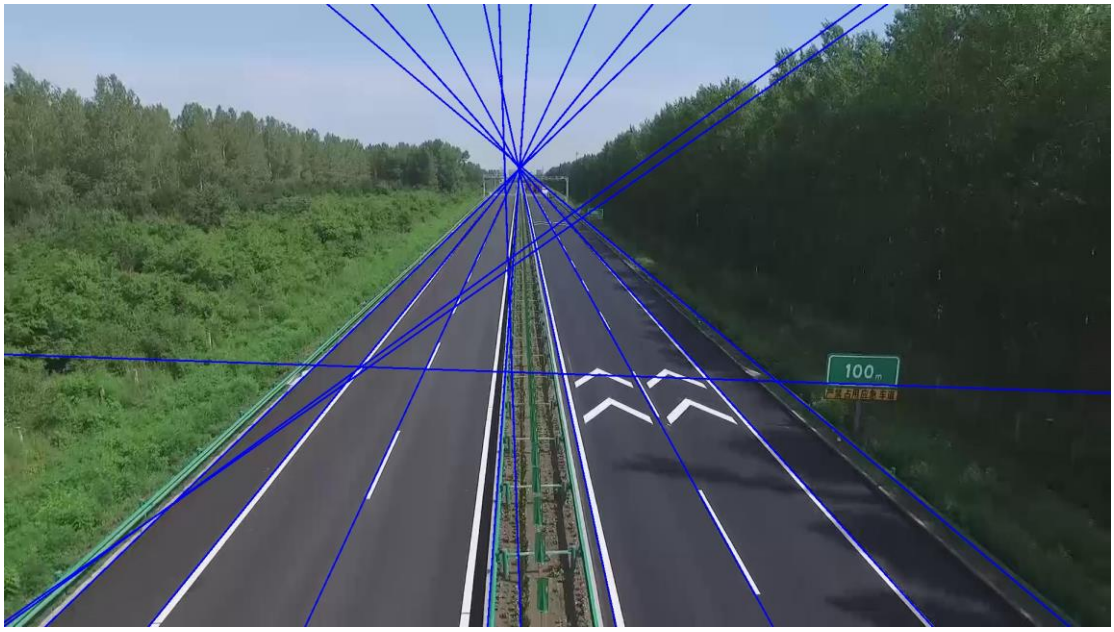
但是对于 highway 这张信息比较复杂的图片，用同样的滤波方式和检测出来的效果却不如人意。除了关键的几条直线被检测出来之外，其他的一些不希望出现的线条也被检测了出来，而且重复性非常大：



所以加入了一步非极大值抑制，对累加器的一定邻域内选择最大值输出，减少了冗余的直线。

```
for (int y = 0; y < 2 * RMax; y++)
{
    for (int x = 0; x < 180; x++)
    {
        if (accu[y][x] > threshold)
        {
            int max = accu[y][x];
            // 非极大值抑制
            for (int ly = -14; ly <= 14; ly++)
            {
                for (int lx = -14; lx <= 14; lx++)
                {
                    if ((ly + y >= 0 && ly + y < 2 * RMax) && (lx + x >= 0 && lx + x < 180))
                    {
                        if (accu[ly + y][lx + x] > max)
                        {
                            max = accu[ly + y][lx + x];
                            ly = lx = 15;
                        }
                    }
                }
            }
            if (max > accu[y][x])
                continue;
```

效果有了很大改观。但是可以看出，对于噪音的鲁棒性不是很好：



2. 圆检测分析

效果比较好，但是由于设定了圆心之间的距离阈值，所以同心圆无法检测。

五、 心得与体会

这次实验感觉难度比较大，在仔细学习了算法以后，就着手开始自己实现 Hough 变换了。遇到了以下几个大问题，被用相应方法自己解决：

- 运算速度慢——将正弦值和余弦值列哈希表索引以加速；
- 圆检测的过程中有不希望出现的点被累加器计算——检测是否在原图像范围内，并且对 Hough 空间进行了一次二值化；
- 直线检测比较杂乱——在不改变阈值的情况下增加非极大值抑制，有一定效果。

但是还有问题，主要是在直线检测方面，对于噪音的影响鲁棒性很差。通过改变高斯滤波算子大小和 Canny 算子大小，甚至是用了形态学变化，效果都不是很好。

这次实验收获很大，自己基本实现了 Hough 变换检测直线和圆的算法，编程能力有了提升，阅读源码的能力也有了提升，同时自己实现了一下 Sobel 算子检测边缘的算法，效果也符合预期。