**实验报告**

1. **实验与测试环境**

Windows

Dev C++

1. **代码目录说明**

代码目录即src目录，结构说明：

CannyEdgeDetection为改写并封装后的代码，里面含有可执行程序Canny.exe；

Code0、Code1与Code2为原代码，我抽中改写的为Code1；

output为实验输出的结果图像；

test\_Data为测试图像。

1. **实验步骤（代码说明）**

**算法原理：**

图像的Canny边缘检测算法，是一种理论和实际效果比较靠谱的算法，它大致有下面几个步骤：

1. 将输入的RGB图像转换成单通道的灰度图像

     2. 计算灰度图像的一阶梯度，该算法选择Sobel算子计算dx和dy两个方向的梯度。

     3. 对图像的梯度幅值进行非极大值抑制，这一步是Canny算法的独到之处经过这一步后，真正的边缘点会被暴露出来，并将其位置保存下来。

     4. 双阈值检测：将保存下来的已确认为边缘的点，以每个点为中心，将相邻的8个像素由1变成2，即将曲线的轮廓进行不断的延伸。

1. **将输入的RGB图像转换成单通道的灰度图像**

代码：

int Canny::toGrayScale(CImg<unsigned char> img)

{

    gray.assign(img.width(), img.height(), 1, 1, 0);

    cimg\_forXY(img, x, y) {

        int R = img(x, y, 0);

        int G = img(x, y, 1);

        int B = img(x, y, 2);

        double A = (R \* 0.212671 + G \* 0.715160 + B \* 0.072169);

        gray(x, y, 0) = gray(x, y, 0) = gray(x, y, 0) = A;

    }

    //gray.display("gray");

    return 1;

}

分析：

该步骤主要是把图片变成灰度图，应用了公式R \* 0.212671 + G \* 0.715160 + B \* 0.072169。

1. **高斯过滤（原代码好像没有）**

代码：

int Canny::filter(CImg<unsigned char> img, double sigma) {

    gauss.assign(img.width(), img.height(), 1, 1, 0);

    double sum, pSum;

    cimg\_forXY(img, x, y) {

        if(x < 1 || x >= img.width() - 1 || y < 1 || y >= img.height() - 1){

            continue;

        } //3x3的filter，图像的4条边需要跳过

        sum = 0;

        pSum = 0;

        for (int i = -1; i < 2; i++) {

            for (int j = -1; j < 2; j++) {

                sum += gaussian(i, j, sigma);

                pSum += gaussian(i, j, sigma) \* img(x + i, y + j);

            }

        }

        gauss(x, y) = pSum / sum;

    }

    //gauss.display("gauss");

    return 1;

}

分析：

高斯过滤采用了3\*3的模型，该步骤使图像变得模糊，减少了噪点。

1. **利用Sobel算子计算图像的梯度与方向**

代码:

int Canny::sobel(CImg<unsigned char> img) {

    magnitude.assign(img.width(), img.height(), 1, 1, 0);

    orientation.assign(img.width(), img.height(), 1, 1, 0);

    double sumx, sumy;

    int width = img.width();

    int height = img.height();

    cimg\_forXY(img, x, y) {

        if(x < 1 || x >= width - 1 || y < 1 || y >= height - 1){

            continue;

        }  //3x3的filter，图像的4条边需要跳过

        sumx = (-1) \* img(x-1, y-1) + 1 \* img(x+1, y-1) + (-2) \* img(x-1, y) + 2 \* img(x+1, y) + (-1) \* img(x-1, y+1) + 1 \* img(x+1, y+1);

        sumy = (-1) \* img(x-1, y-1) + 1 \* img(x-1, y+1) + (-2) \* img(x, y-1) + 2 \* img(x, y+1) + (-1) \* img(x+1, y-1) + 1 \* img(x+1, y+1);

        magnitude(x, y) = sqrt(sumx\*sumx + sumy\*sumy);

        if (sumx == 0)

            orientation(x, y) = 90;

        else

            orientation(x, y) = atan(sumy / sumx) \* (180.0 / PI);

    }

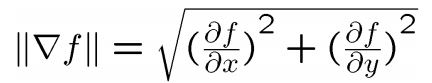
    //magnitude.display("magnitude");

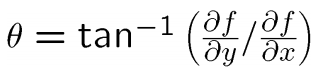
    return 1;

}

分析：

这一步采用3\*3的Sobel算子，把算子代入到像素点上即可。需要计算出幅值与方向，利用公式：





1. **非极大值抑制**

代码：

int Canny::nonMaxSupp() {

    non.assign(magnitude.width(), magnitude.height(), 1, 1);

    cimg\_forXY(magnitude, x, y) {

        if(x < 1 || x >= magnitude.width() -1 || y < 1 || y >= magnitude.height() - 1){

            continue;

        } //3x3的算子，图像的4条边需要跳过

        float Tangent = orientation(x, y);

        //cout << Tangent << " ";

        non(x, y) = magnitude(x, y);

        //有四个方向

        //0: -0.4142 < tan θ <= 0.4142  Horizontal Edge

        //1: |tan θ| >= 2.4142  Vertical Edge

        //2: -2.4142 < tan θ <= -0.4142  -45 Degree Edge

        //3: 0.4142 < tan θ <= 2.4142  45 Degree Edge

        //画图更易理解

        //Horizontal Edge

        if (((-22.5 < Tangent) && (Tangent <= 22.5)) || ((157.5 < Tangent) && (Tangent <= -157.5)))

        {

            //如果不是比左右两边大就为0

            if ((magnitude(x, y) < magnitude(x + 1, y)) || (magnitude(x, y) < magnitude(x - 1, y)))

                non(x, y) = 0;

        }

        //Vertical Edge

        if (((-112.5 < Tangent) && (Tangent <= -67.5)) || ((67.5 < Tangent) && (Tangent <= 112.5)))

        {

            //如果不是比上下两边大就为0

            if ((magnitude(x, y) < magnitude(x , y + 1)) || (magnitude(x, y) < magnitude(x , y - 1)))

                non(x, y) = 0;

        }

        //-45 Degree Edge

        if (((-67.5 < Tangent) && (Tangent <= -22.5)) || ((112.5 < Tangent) && (Tangent <= 157.5)))

        {

            //如果不是比-45度对角两边上大就为0

            if ((magnitude(x, y) < magnitude(x + 1, y + 1)) || (magnitude(x, y) < magnitude(x - 1, y - 1)))

                non(x - 1, y- 1) = 0;

        }

        //45 Degree Edge

        if (((-157.5 < Tangent) && (Tangent <= -112.5)) || ((22.5 < Tangent) && (Tangent <= 67.5)))

        {

            //如果不是比45度对角两边上大就为0

            if ((magnitude(x, y) < magnitude(x - 1, y + 1)) || (magnitude(x, y) < magnitude(x + 1, y - 1)))

                non(x - 1, y - 1) = 0;

        }

    }

    //non.display("non");

    return 1;

}

分析：

这一步要利用到上个步骤所计算出来的幅值与方向，根据每个像素点的方向，计算该点是否为该方向上最大值的点，是则保持，不是则被抛弃。

1. **双阈值检测（包括滞后阈值法）**

代码：

int Canny::hysterThres(double lowThresh, double highThresh) {

    //If a pixel gradient is higher than the upper threshold,

    // the pixel is accepted as an edge

    //If a pixel gradient value is below the lower threshold,

    // then it is rejected.

    //If the pixel gradient is between the two thresholds,

    // then it will be accepted only if it is connected to a pixel that is above the upper threshold.

    if (lowThresh > 255)

        lowThresh = 255;

    if (highThresh > 255)

        highThresh = 255;

    edges.assign(non.width(), non.height(), 1, 1, 0);

    cimg\_forXY(non, x, y) {

        if (non(x, y) >= highThresh)

        {

            followEdges(x, y, lowThresh, highThresh);

        }

    }

    //edges.display("edges");

    return 1;

}

void Canny::followEdges(int x, int y, double lowThresh, double highThresh) {

    //递归的方法实现边缘跟踪

    edges(x, y) = 255;

    for (int i = x - 1; i < x + 2; i++) {

        for (int j = y - 1; j < y + 2; j++) {

            if(i < 0 || i >= non.width()  || j < 0 || j >= non.height()) continue;

            if((j != y || i != x) && non(x, y) >= lowThresh && edges(x, y) != 255)

            {

                    followEdges(x, y, lowThresh, highThresh);

            }

        }

    }

}

分析：

这一步当中，先初始化每个像素点为0，然后对原图进行遍历，查找到大于最大阈值的像素点（此为强边缘点），进行边缘追踪（利用函数followEdge递归实现），即把在高低阈值之间的点（即弱边缘点）符合条件的转化为强边缘点，而小于阈值的被抛弃。

1. **把相邻的边缘连成长的线条，并删除长度小于 20 的 Edge**
2. **测试结果与分析**

代码：

int Canny::optimizeLine(CImg<unsigned char> img, int distance) {//distance 是连接两个edge的最大距离

    line = img;

    //用于计算某一个像素点是否为边缘点

    //判断方法为查看以这个点为中心的八邻域，如果附近只有1个像素点为255, 其他7个为0则是边缘点

    bool isEdge[1000][1000];

    bool \_isEdge[1000][1000];

    cimg\_forXY(line, x, y) {

        isEdge[x][y] = false;

        if(x < 1 || x >= img.width() - 1 || y < 1 || y >= img.height() - 1 || line(x, y) == 0){

            continue;

        } //3x3的filter，图像的4条边需要跳过

        int edge = 0;

        for (int i = x - 1; i < x + 2; i++) {

            for (int j = y - 1; j < y + 2; j++) {

                if (!(i == x && j == y)) {

                    if (line(i, j) == 255)

                        edge++;

                }

            }

        }

        if (edge == 1) {}

            \_isEdge[x][y] = isEdge[x][y] = true;

        //删除单个孤立的点

        if (edge == 0)

            line(x, y) = 0;

    }

    cimg\_forXY(line, x, y) {

        if(x < distance || x >= img.width() - distance || y < distance || y >= img.height() - distance || !isEdge[x][y]){

            continue;

        } //3x3的filter，图像的4条边需要跳过

        for (int i = x - distance; i <= x + distance; i++) {

            for (int j = y - distance; j <= y + distance; j++) {

                if (isEdge[i][j]) {

                int white[] = {255, 255, 255};

                line.draw\_line(x, y, i, j, white);

                isEdge[i][j] = false;

                isEdge[x][y] = false;

                }

            }

        }

    }

    //删除长度少于20的连线

    //判断如果两个边界点的距离小于20，就删除这两个边界点组成的矩阵内所有白点，这样的话即使两个边界点分别是两条直线的话也无所谓

    //反正是这样的话这两边界点之间都是黑色区域，删除也无所谓

    cimg\_forXY(line, x, y) {

        distance = 20;

        if (isEdge[x][y] == true) {

            int begin\_x = x - distance > 0 ? x - distance : 0;

            int begin\_y = y - distance > 0 ? y - distance : 0;

            int end\_x = x + distance < img.width() - 1 ? x + distance : img.width() - 1;

            int end\_y = y + distance < img.height() - 1 ? y + distance : img.height() - 1;

            for (int i = begin\_x; i <= end\_x; i++) {

                for (int j = begin\_y; j <= end\_y; j++) {

                    if (isEdge[i][j] == true) {

                        int max\_x = x >= i ? x : i;

                        int max\_y = y >= j ? y : j;

                        int min\_x = max\_x == x ? i : x;

                        int min\_y = max\_y == y ? j : y;

                        for (int ii = min\_x; ii <= max\_x; ii++) {

                            for (int jj = min\_y; jj <= max\_y; jj++) {

                                line(ii, jj) = 0;

                            }

                        }

                        \_isEdge[i][j] = false;

                        \_isEdge[x][y] = false;

                    }

                }

            }

        }

    }

    //删除经过上一步处理完可能存在的单个孤立的噪声点

    cimg\_forXY(line, x, y) {

        if(x < 1 || x >= img.width() - 1 || y < 1 || y >= img.height() - 1 || line(x, y) == 0){

            continue;

        } //3x3的filter，图像的4条边需要跳过

        int edge = 0;

        for (int i = x - 1; i < x + 2; i++) {

            for (int j = y - 1; j < y + 2; j++) {

                if (!(i == x && j == y)) {

                    if (line(i, j) == 255)

                        edge++;

                }

            }

        }

        //删除单个孤立的点

        if (edge == 0)

            line(x, y) = 0;

    }

    //line.display("line");

}

分析：

这一步是额外增加的函数，对图像进一步优化，步骤有点繁琐。首先先找出在边缘上的顶点，然后找出其周围（这个距离我设为5）的顶点连接起来，连接过后会有一些孤立的边缘（长度小于20），我们删除它，根据这个算法还需要再删除周围无其余边缘点的点。

1. 测试结果与分析
2. 我首先对lena.bmp进行了不同参数（不同阈值范围）的测试。

结果如下：

15~35



30~70



40~70



80~100



分析：

可以看到使用的高阈值越高，一些噪点被清除得越好，然而也丢失了一些本该是边缘的点。本来是想尝试固定高阈值，改变低阈值（也是在改变阈值范围），看有什么变化，实验结果是不太明显。

1. 实验的每一个步骤

原图：



灰度图：



高斯过滤：



Sobel：



非最大化抑制：



滞后双阈值：



边缘连接与删除：



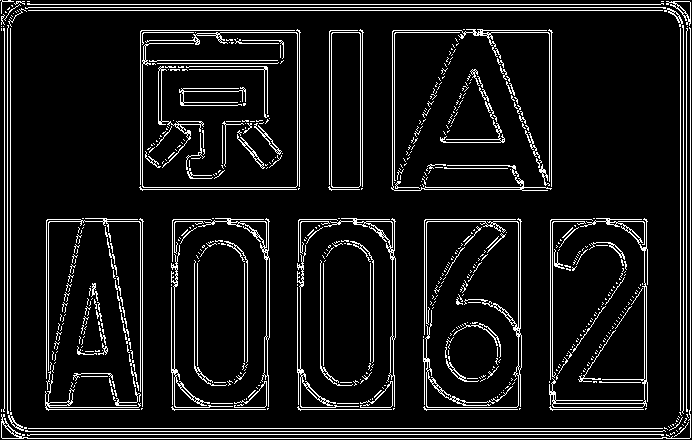
1. 测试对边缘的优化

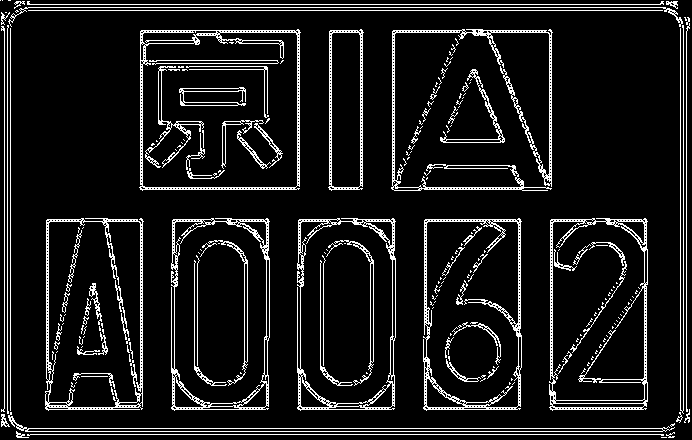
可以看到在lena这张图中减少了一些噪点与短的边缘，一些地方边缘也更清晰。（下图为优化后）





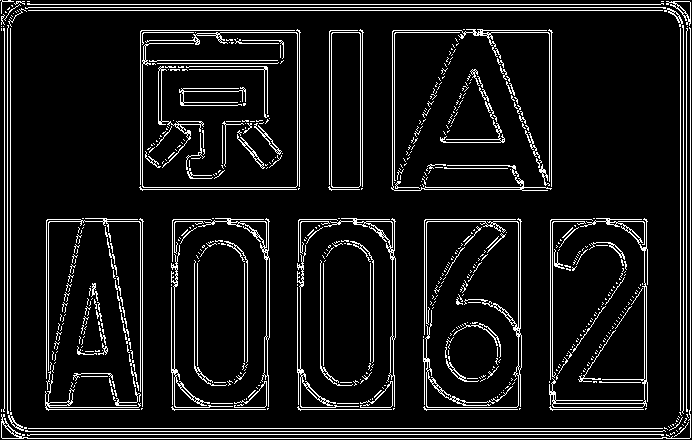
为此我还选用了边缘检测效果最好的图作对比，可以明显看出优化后边缘更清晰，不过也有缺陷，一些不该连上的边缘也连上了（在测试lena时，会发现阈值选择不当，甚至效果更差，出来的结果图像很丑）；同时删除一些短的边也会错删除一些本是重要的边缘的边，而且我认为先删除还是先延长边缘会有不同的效果（测试中尝试过）。多次实验结果表明，想要这一步效果好，在阈值应选择上大一点。

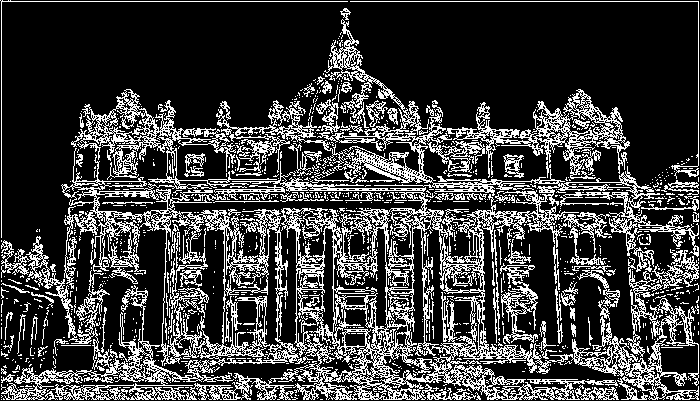
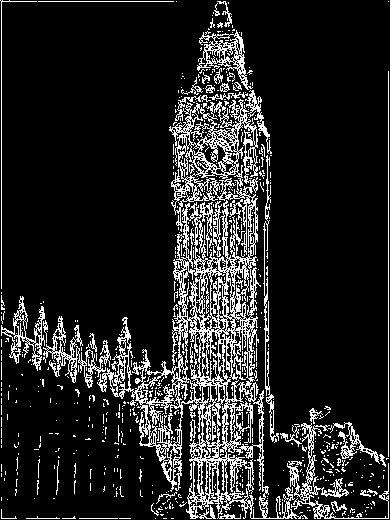
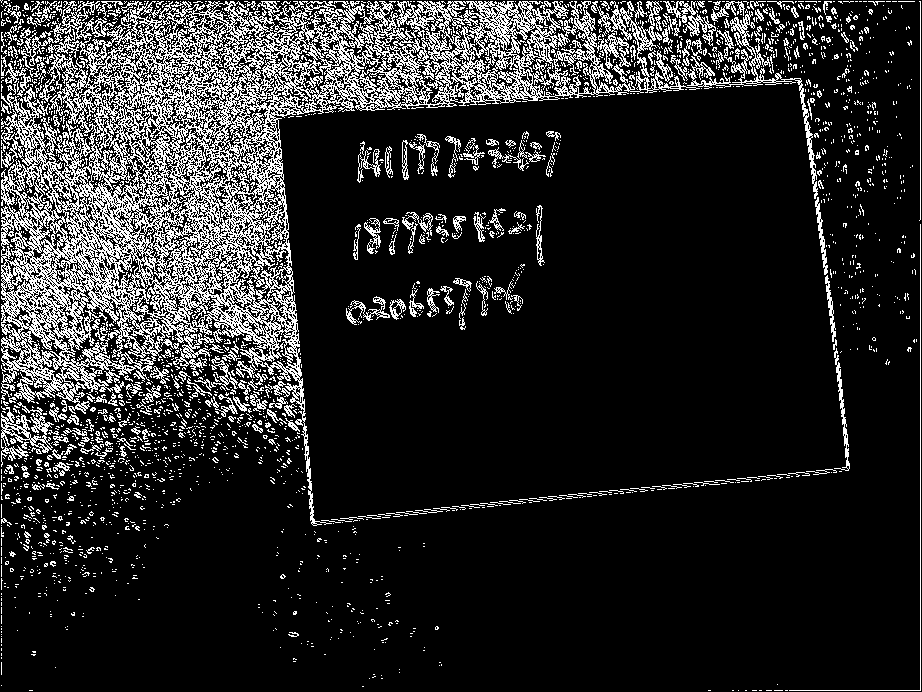




1. 不同的测试对象对比



****

****

1. **实验总结**

本此实验对Canny检测算法的一些源代码进行的解读与改写，改写的过程中我不仅参考给到的Code1，还参考了一些其他代码的实现方式去理解这个算法的步骤，直接看代码理解算法果然快多了。还有，最后增加的函数，效果其实差强人意，这是需要改进的地方，需要搜索一些在这方面的资料。