中山大學
SUN YAT-SEN UNIVERSITY

《计算机视觉与模式识别》
Experiment 5
Testing Report

学院名称	:	数据科学与计算机学院
学生姓名	:	吴若宇
学号	:	14301030
专业（班级）	:	14级计算机科学与技术
时间	:	2017 年 4 月 5 日

矫正标准普通 A4 纸

1、 测试环境

Ubuntu 15.10

2、 测试数据

Dateset2 里面的图片

3、 测试代码及过程

整个实验流程包括边缘检测、霍夫变换和透视变换（Perspective transform）。

1）边缘检测

由于 canny 检测比较慢，所以这次的作业直接采用了 prewitt 算子。

```
void getEdge() {
    // rgb2gray
    CImg<unsigned char> grayImg(img.width(), img.height(), 1, 1, 0);
    cimg_forXY(this->img, x, y) {
        grayImg(x, y, 0) = 0.299 * this->img(x, y, 0) + 0.587 * this->img(x, y, 1) + 0.144 * this->img(x, y, 2);
    }

    // naive prewitt approach
    grayImg.blur(3);
    edgeImg = CImg<unsigned char>(img.width(), img.height(), 1, 1, 0);
    CImg_3x3(I, double);
    cimg_for3x3(grayImg, x, y, 0, 0, I, double) {
        const double ix = Inc - Ipc;
        const double iy = Icp - Icn;
        edgeImg(x, y) = std::sqrt(ix*ix + iy*iy);
    }
    edgeImg.display();
}
```

首先将原图转化为灰度图，对得到的灰度图进行高斯滤波，然后通过 prewitt 算子对图像进行卷积，进行边缘提取。这样替代没有降低边缘提取的质量，还让速度大大提升。

Prewitt 算子是一个 3*3 的矩阵：

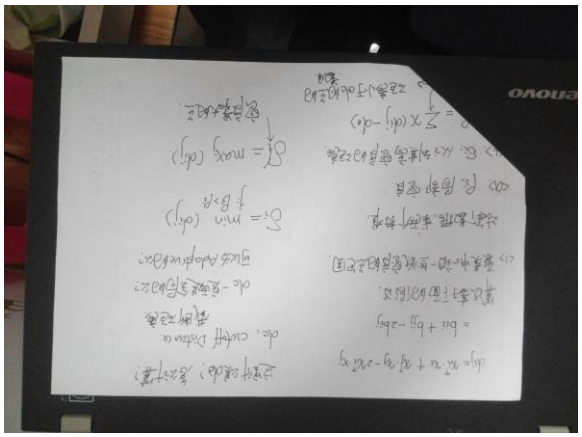
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

通过横向模板G_x和纵向模板G_y，我们可以算出每个点的横向梯度和纵向梯度，再通过下式算出总的梯度，

$$G = \sqrt{G_x^2 + G_y^2}$$

2）霍夫变换（voting 算法）

在 Ex4 的实验报告中进行了说明，这里不再赘述。需要提及的是，由于我的霍夫变换算法是把所有投票结果（并且是局部最大值）放在一个最小堆里面，所以在出现下图情况的时候



由于折起来的一角相对于其他四条边的长度要小，边缘点也相对少，所以在霍夫空间投票的时候的权值也相对小，从而四条主要的边还是能够被提取出来的。寻找最大值的代码如下：

```
void findMax() {
    while (max.size() < 4) {
        max.push(Point(-1, -1, 0));
    }

    cimg_forXY(houghSpace, theta, rho) {
        if (houghSpace(theta, rho) > max.top().weight) {
            if (isLocalMax(theta, rho)) {
                if (inRegion(theta, rho)) {
                    max.pop();
                    max.push(Point(theta, rho, houghSpace(theta, rho)));
                }
            }
        }
    };
}
```

```
bool isLocalMax(int theta, int rho) {
    for (int thetaShift = -thetaFilterWidth; thetaShift <= thetaFilterWidth; thetaShift ++) {
        for (int rhoShift = -rhoFilterWidth; rhoShift <= rhoFilterWidth; rhoShift ++) {
            int t = theta + thetaShift;
            int r = rho + rhoShift;
            if (thetaShift == 0 && rhoShift == 0)
                continue;
            if (r < 0 || r >= houghSpace.height())
                continue;
            // 1 degree is very close to 359 degree, so we have to take
            // it into our window's consideration
            if (t < 0)
                t += 360;
            if (t >= 360)
                t %= 360;
            if (houghSpace(t, r) > houghSpace(theta, rho))
                return false;
        }
    }
    return true;
}
```

寻找最大值的时候，因为极坐标原点的选取，我这里 359°和 1°实际上也是很接近的，所以我们需要进行一些简单的处理，保证能够取到局部最大值。

3）透视变换（perspective transform）

（a）角点对应

在找到四个角点之后，我们就把原图的四个角点和我们想要映射到的图的四个角点一一对应起来，这个模块是在 perspectiveTransform 这个类中的构造函数中实现的，代码如下：

```
perspectiveTransform(const std::vector<Point>& _originalCorner, const std::vector<Point>& _changedCorner, CImg<double> _source) {
    this->originalCorner = _originalCorner;
    this->changedCorner = _changedCorner;
    this->source = _source;

    if (originalCorner.size() != changedCorner.size()) {
        throw "points sets size are not equal";
    }
    if (originalCorner.size() != 4) {
        throw "points set size are not 4";
    }

    std::sort(originalCorner.begin(), originalCorner.end(), [](Point& p1, Point& p2)
                        {return pow(p1.theta, 2) + pow(p1.rho, 2) < pow(p2.theta, 2) + pow(p2.rho, 2);});
    std::sort(changedCorner.begin(), changedCorner.end(), [](Point& p1, Point& p2)
                        {return pow(p1.theta, 2) + pow(p1.rho, 2) < pow(p2.theta, 2) + pow(p2.rho, 2);});

    for (int i = 0; i < originalCorner.size(); i ++) {
        cornerMap.push_back(make_pair(originalCorner[i], changedCorner[i]));
    }
}
```

其中原始的角点就是原图经过霍夫变换找到的四个角点，而我们默认的变换之后的角点是原图的左上、左下、右下和右上四个点。然后对于两组角点我们分别按照点到原点（0，0）的距离进行排序，两组角点的点按照顺序两两配成一对，就得到了我们的对应关系，核心代码为下面几行：

```
std::sort(originalCorner.begin(), originalCorner.end(), [](Point& p1, Point& p2)
                        {return pow(p1.theta, 2) + pow(p1.rho, 2) < pow(p2.theta, 2) + pow(p2.rho, 2);});
std::sort(changedCorner.begin(), changedCorner.end(), [](Point& p1, Point& p2)
                        {return pow(p1.theta, 2) + pow(p1.rho, 2) < pow(p2.theta, 2) + pow(p2.rho, 2);});

for (int i = 0; i < originalCorner.size(); i ++) {
    cornerMap.push_back(make_pair(originalCorner[i], changedCorner[i]));
}
```

（b）获取透视变换的矩阵

透视变换是将图片投影到一个新的视平面，通用的变换公式为

$$[x', y', w'] = [u, v, w]\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

u, v 是原始图片坐标，对应得到变换后的图片坐标 x, y。其中 $x = x'/w', y = y'/w'$，重写这个公式可以得到：

$$x = \frac{x'}{w'} = \frac{a_{11}u + a_{21}v + a_{31}}{a_{13}u + a_{23}v + a_{33}}$$

$$y = \frac{y'}{w'} = \frac{a_{12}u + a_{22}v + a_{32}}{a_{13}u + a_{23}v + a_{33}}$$

从上式我们可以看出，变换矩阵的每一部分都有其意义：$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$表示线性变换，$\begin{bmatrix} a_{31} & a_{32} \end{bmatrix}$表示平移，$\begin{bmatrix} a_{13} & a_{23} \end{bmatrix}^T$产生透视变换。

w 和a_{33}我们一般设置为 1，所以一共有 8 个参数。由于我们有四对(u, x)和四对(v, y)，所以这个方程组我们是可以解出来的。我们把这个方程重新组织一下可以得到：

$$u_i a_{11} + -u_i x_i a_{13} + v_i a_{21} + -v_i x_i a_{23} + a_{31} = x_i$$

$$u_i a_{12} + -u_i y_i a_{13} + v_i a_{22} + -v_i y_i a_{23} + a_{22} = y_i$$

其中，$1 \le i \le 4$，也就是我们的四对角点。我们可以把它变为一个矩阵乘法：

$$\begin{bmatrix} u_1 & 0 & -u_1x_1 & v_1 & 0 & -v_1x_1 & 1 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & u_1 & -u_1y_1 & 0 & v_1 & -v_1y_1 & 0 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

这就是一个$Ax = b$，我们想求的是x，所以我们可以通过以下公式求需要的参数：

$$x = A^{-1}b$$

所以，我们对矩阵 A 赋值如下：

```
// give values to A
for (int row = 0; row < 4; row ++) {
    A(0, row) = cornerMap[row].second.theta;
    A(1, row) = 0;
    A(2, row) = -cornerMap[row].second.theta * cornerMap[row].first.theta;
    A(3, row) = cornerMap[row].second.rho;
    A(4, row) = 0;
    A(5, row) = -cornerMap[row].second.rho * cornerMap[row].first.theta;
    A(6, row) = 1;
    A(7, row) = 0;
}
for (int row = 4; row < 8; row ++) {
    A(0, row) = 0;
    A(1, row) = cornerMap[row - 4].second.theta;
    A(2, row) = -cornerMap[row - 4].second.theta * cornerMap[row - 4].first.rho;
    A(3, row) = 0;
    A(4, row) = cornerMap[row - 4].second.rho;
    A(5, row) = -cornerMap[row - 4].second.rho * cornerMap[row - 4].first.rho;
    A(6, row) = 0;
    A(7, row) = 1;
}
```

对矩阵 b 赋值如下：

```
// give values to b
for (int row = 0; row < 4; row ++) {
    b(0, row) = cornerMap[row].first.theta;
}
for (int row = 4; row < 8; row ++) {
    b(0, row) = cornerMap[row - 4].first.rho;
}
```

再通过一个代数运算，我们就能得到我们需要的变换矩阵：

```
CImg<double> transformMatrixColumn = A.get_invert() * b;

for (int i = 0; i < 8; i ++) {
    transformMatrix.push_back(transformMatrixColumn(0, i));
}
transformMatrix.push_back(1);
```

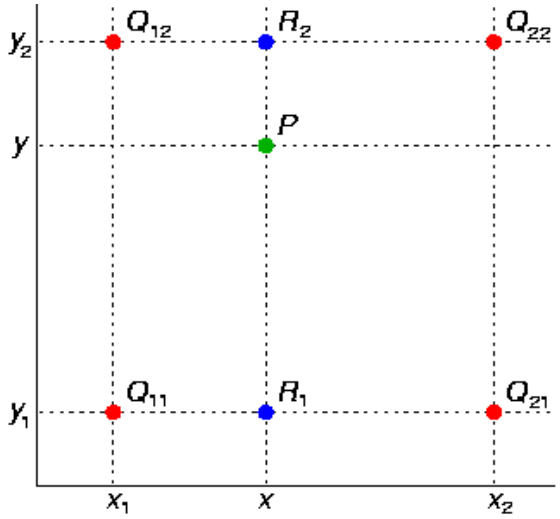
（c）双线性插值

双线性插值和我们以前接触到的插值差不多，其改进的地方在于它是在两个方向上分别进行一次线性插值。假设我们想得到未知函数f在点$P=(x,y)$的值，假设我们已知函数f在$Q_{11}=(x_1,y_1),Q_{12}=(x_1,y_2),Q_{21}=(x_2,y_1),Q_{22}=(x_2,y_2)$四个点的值，那么我们可以首先在 x 方向进行插值，得到

$$f(R_1)\approx\frac{x_2-x}{x_2-x_1}f(Q_{11})+\frac{x-x_1}{x_2-x_1}f(Q_{21})\quad\text{Where}\quad R_1=(x,y_1),$$
$$f(R_2)\approx\frac{x_2-x}{x_2-x_1}f(Q_{12})+\frac{x-x_1}{x_2-x_1}f(Q_{22})\quad\text{Where}\quad R_2=(x,y_2).$$

再在 y 方向进行插值

$$f(P)\approx\frac{y_2-y}{y_2-y_1}f(R_1)+\frac{y-y_1}{y_2-y_1}f(R_2).$$



我们把两个三个公式合在一起，可以得到下式：

$$f(x,y)\approx\frac{f(Q_{11})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y_2-y)+\frac{f(Q_{21})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y_2-y)$$
$$+\frac{f(Q_{12})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y-y_1)+\frac{f(Q_{22})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y-y_1).$$

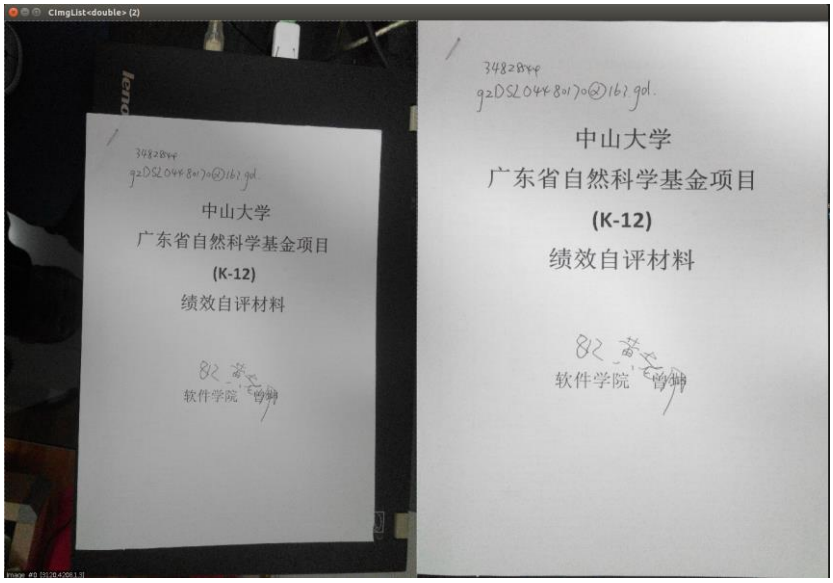
根据以上公式，我们能很快的写出以下程序：

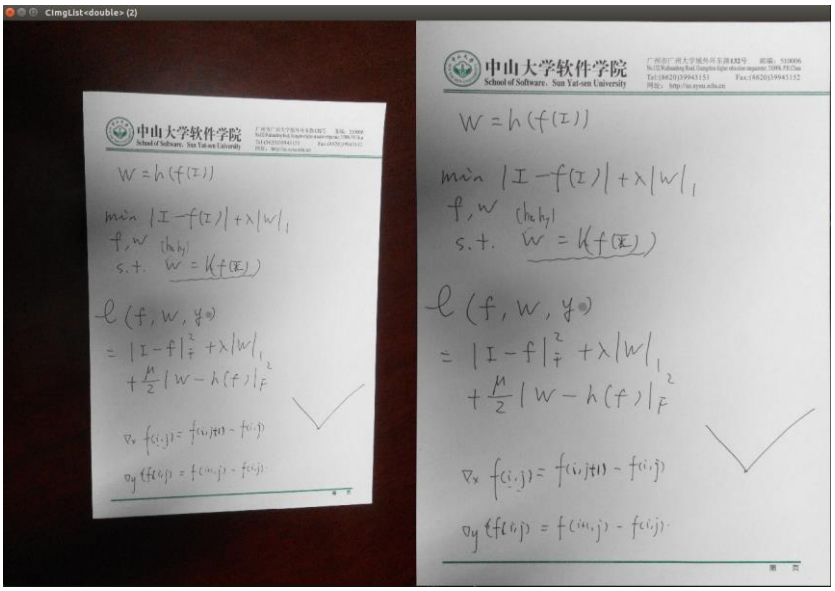
```
void bilinearInterpolation() {
    transformed = CImg<double>(source.width(), source.height(), 1, 3);
    cimg_forXY(transformed, u, v) {
        double x = (transformMatrix[0] * u + transformMatrix[3] * v + transformMatrix[6]) / (transformMatrix[2] * u + transformMatrix[5] * v + 1);
        double y = (transformMatrix[1] * u + transformMatrix[4] * v + transformMatrix[7]) / (transformMatrix[2] * u + transformMatrix[5] * v + 1);
        int x1 = floor(x);
        int x2 = x1 + 1;
        int y1 = floor(y);
        int y2 = y1 + 1;
        for (int channel = 0; channel < source.spectrum(); channel ++) {
            // if some transformed points fall outside of source image
            if (x1 < 0 || x2 >= source.width())
                continue;
            if (y1 < 0 || y2 >= source.height())
                continue;

            transformed(u, v, channel) = (source(x1, y1, channel) * (x2 - x) * (y2 - y)
                                        + source(x2, y1, channel) * (x - x1) * (y2 - y)
                                        + source(x1, y2, channel) * (x2 - x) * (y - y1)
                                        + source(x2, y2, channel) * (x - x1) * (y - y1)) / ((x2 - x1) * (y2 - y1));
        }
    };
}
```

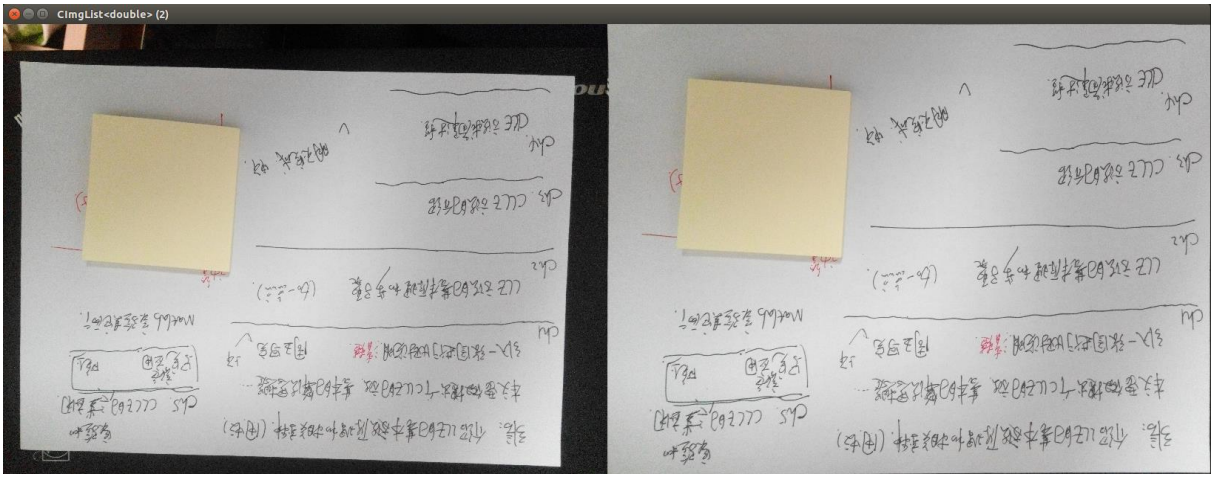
这里得到的结果也就是我们最终想要的结果了。

4、 实验结果

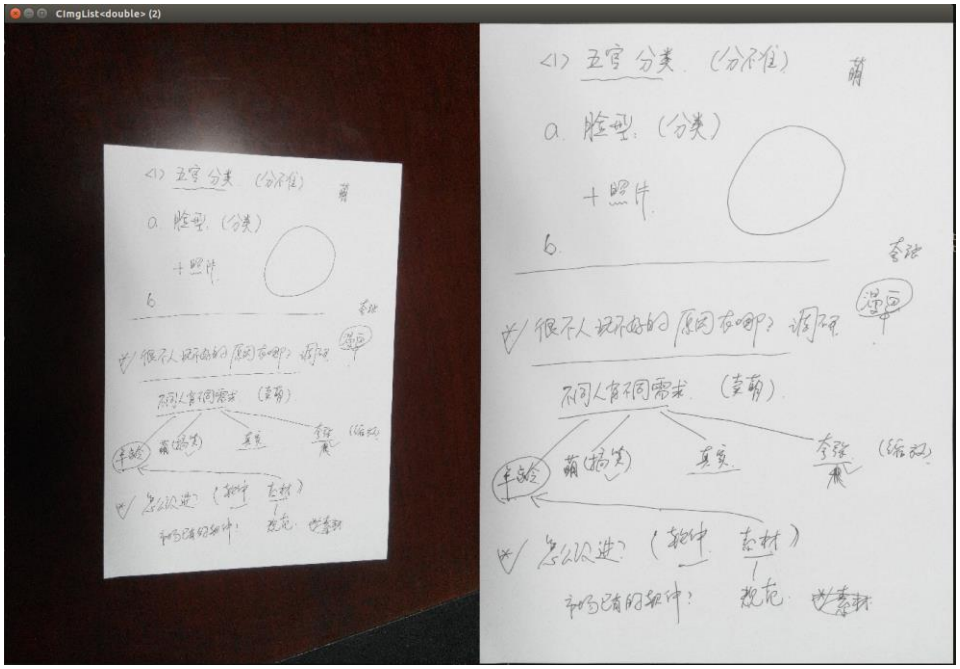
以下为这次实验的实验结果，左图为原图，右图为处理之后得到的结果

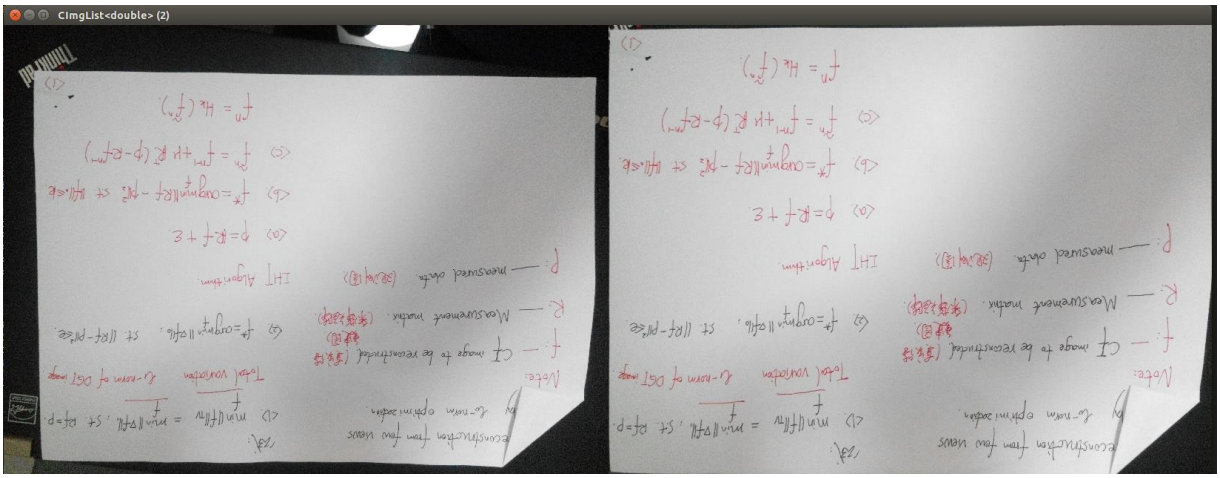


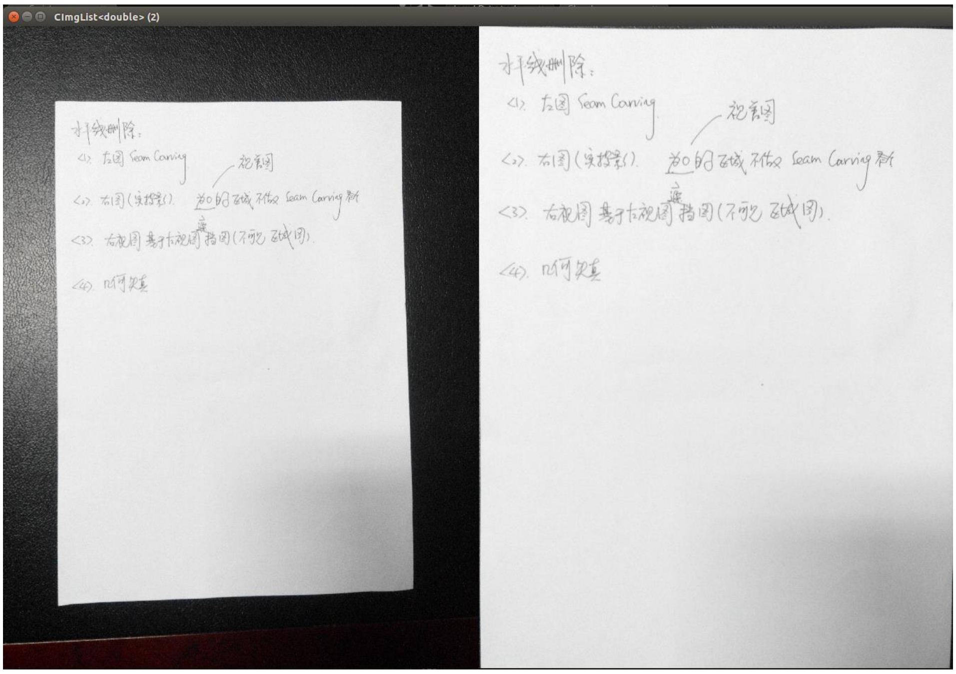


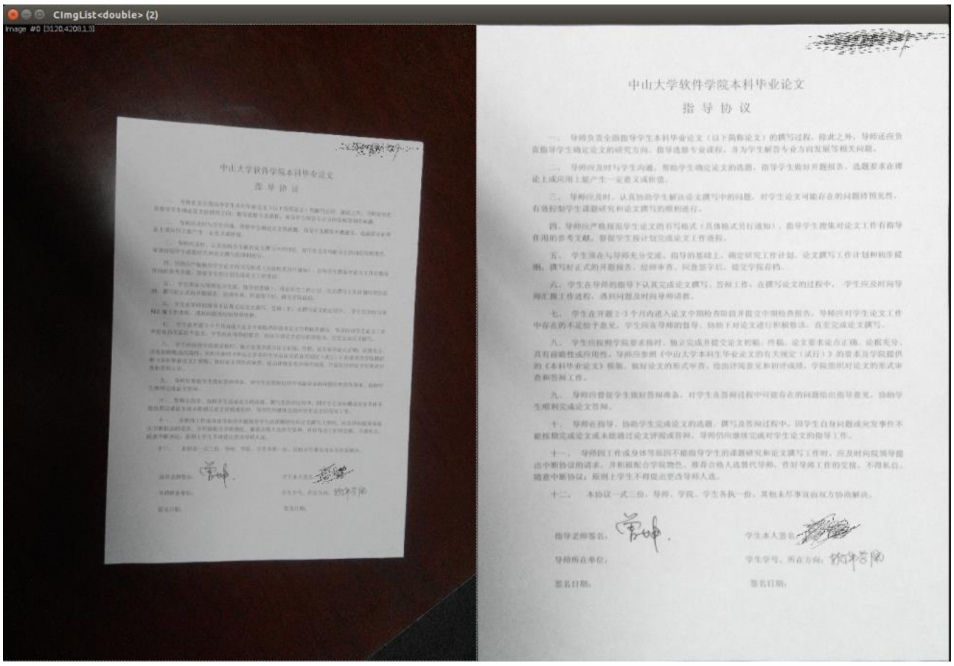


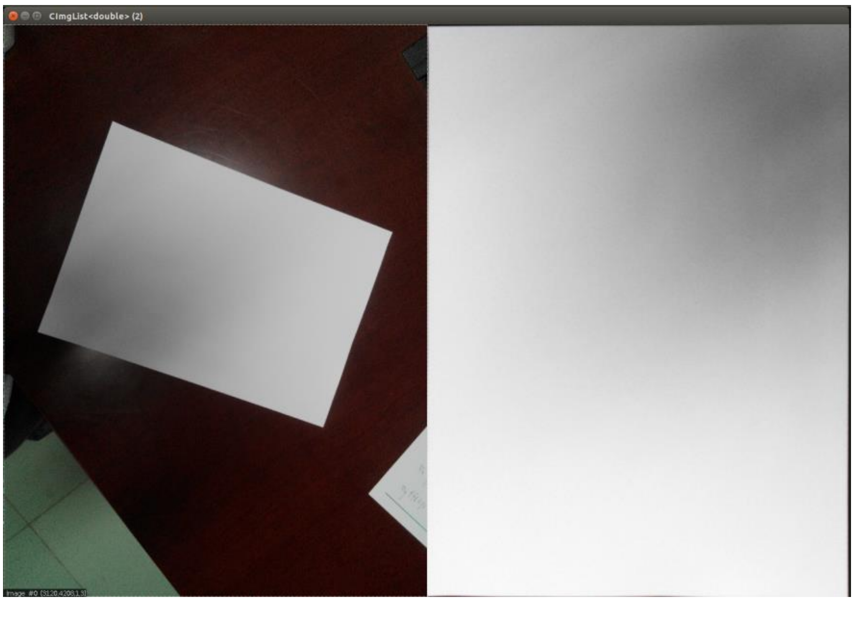

CImgList<double> (2)
中山大学软件学院
School of Software, Sun Yat-sen University

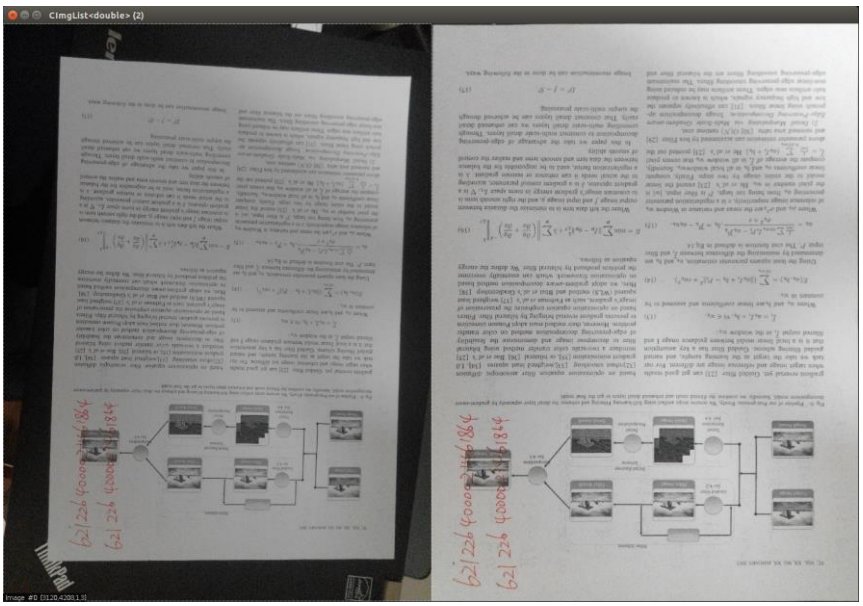

CImgList<double> (2)


CImgList<double> (2)

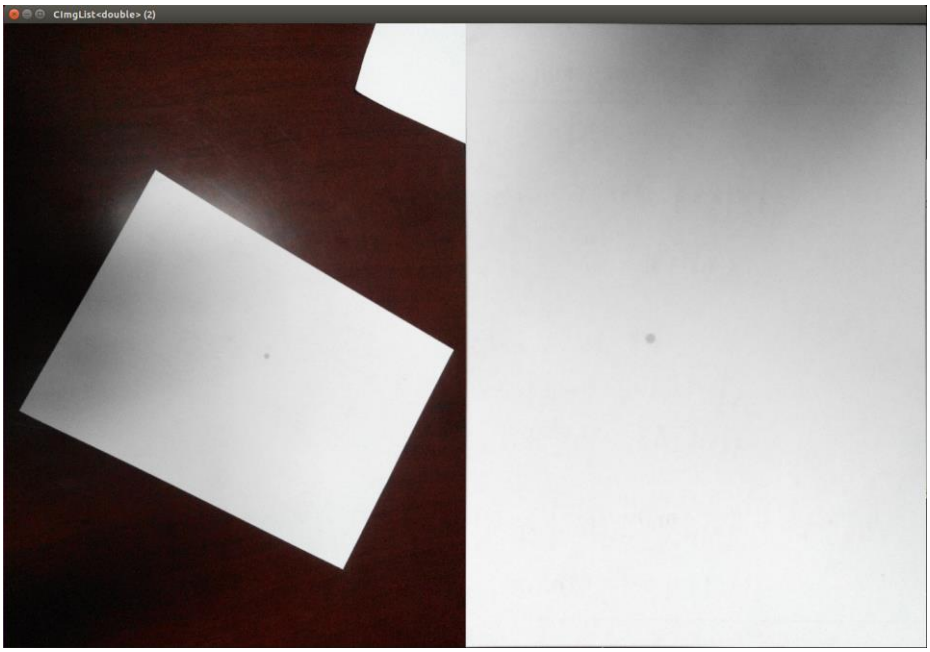

CImgList<double> (2)

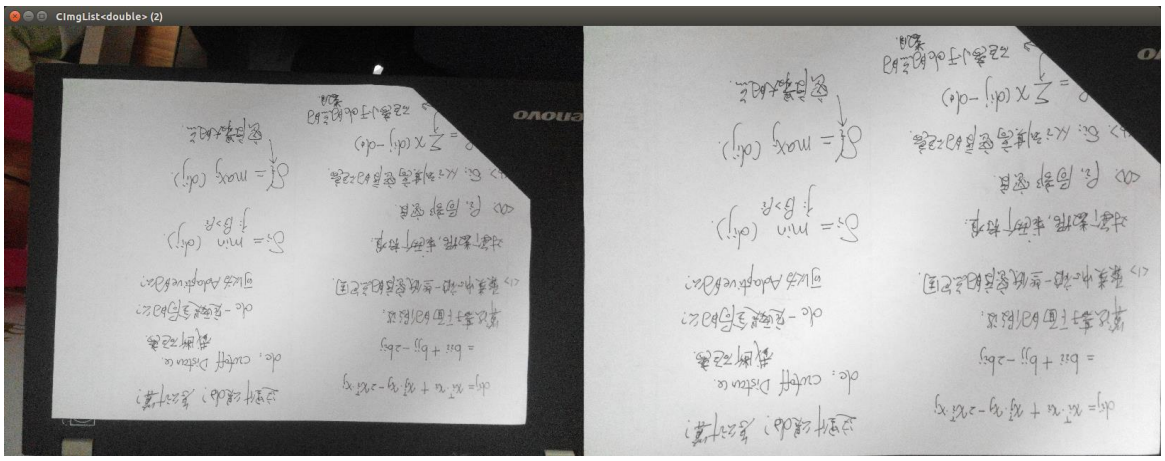

CImgList<double> (2)
中山大学软件学院本科毕业论文
指导协议

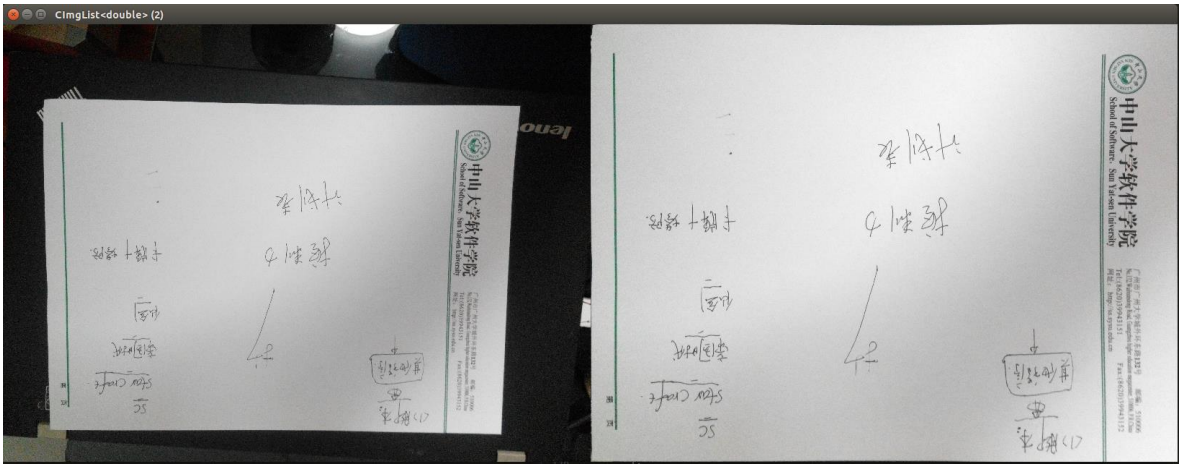

CImgList<double> (2)

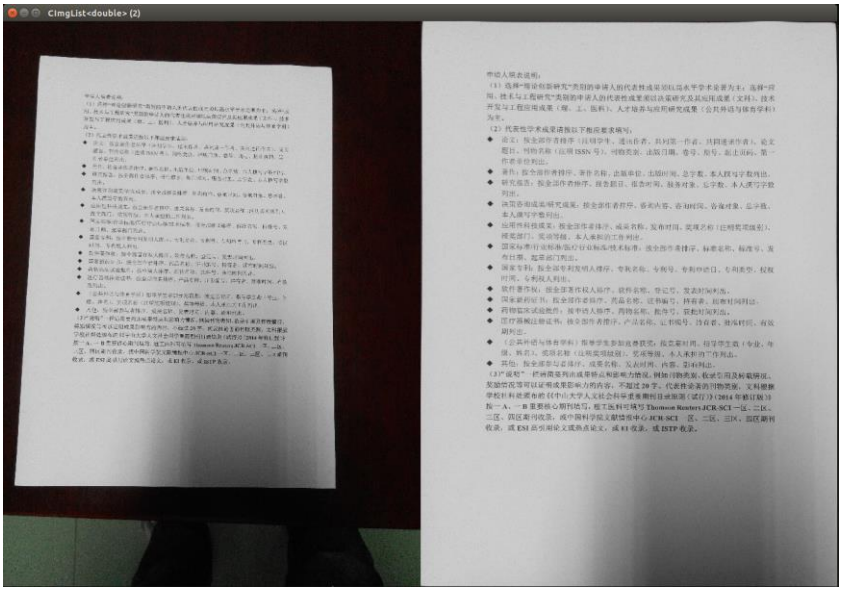

CImgList<double> (2)

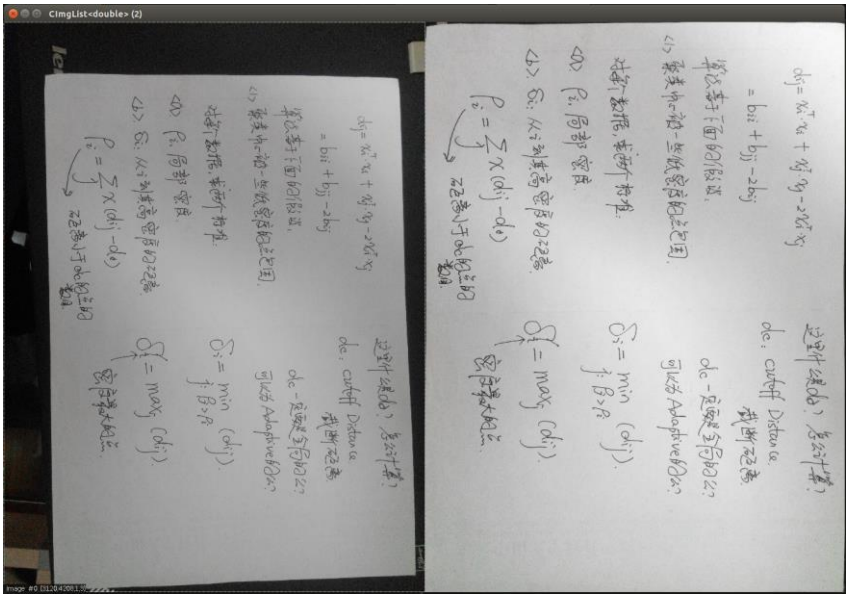

CImgList<double> (2)


CImgList<double> (2)


CImgList<double> (2)


CImgList<double> (2)
中山大学软件学院
School of Software Sun Yat-sen University


CImgList<double> (2)



5、 总结及提高

我们可以看到，这次实验得到的结果非常理想，而且因为用 prewitt 算子替代了 canny 方法，所以不仅速度快，也不用进行参数调整。（所有的图片都可以不用修改）在网上搜索的时候也看到了一些别的解决方案，比如把原图得到的不规则的四边形分割成两个三角形，再对两个三角形进行矫正。这样做的话从变换上来看的话比较简单，而且我们附加题做 Image Morphing 的时候，也要用到三角形的变换，所以这一个方案是非常有诱惑力的。但是看到如果用三角形来做 A4 纸矫正的话，中间对角线的位置的畸变可能会比较严重，很多博客都是推荐用透视变换来做。考虑到反正附加题会写一遍三角形的变换，所以这次还是使用了透视变换来做。