中山大學
SUN YAT-SEN UNIVERSITY

计算机视觉

（大作业 · 任务一）

学 院 名 称	：	数据科学与计算机学院
专业（班级）	：	17 软件工程 1 班
学 生 姓 名	：	曾峥
学 号	：	17343006
时 间	：	2019 年 12 月 14 日

目录

实验要求 2

名片校正 3

1. 找到顶点 3

1.1 前后景分割 4

1.2 下采样 6

1.3 Canny 检测 7

1.4 Hough 变换 8

1.5 顶点排序 14

2. Warping 矫正 16

2.1 傅里叶变换原理 17

2.2 获取顶点信息 17

2.3 透视变换 18

3. 实验过程 19

主要部分切割 20

4. 图像预处理 21

5. 膨胀以及腐蚀操作 22

6. 求联通块（2-step 算法） 22

7. 联通块聚集为群 23

8. “群“的处理 28

9. 处理所有的图 33

实验总结 34

类的封装 34

提交文件 37

实验要求

●任务 1:名片校正，名片主要部分切割，汉子字符训练数据集的收集

1. 名片校正:主要是把名片从背景图像中切割出来,只保留名片本身的完整图像信息(不含背景图等),可以采用前面的 Edge+Hough 变换,或者 Edge+ Ransac,或者 Image Segmentation+ Hough 变换.

2. 名片主要部分切割:对于名片图像,可以分为如下几个主要的部分:姓名, Title, 单位信息,电话号码等.这里的方法类似作业 5 的思路.
3. 每位同学收集 30 张名片，并拍照发给 TA(要求光照均匀且充足明亮,拍照清晰).

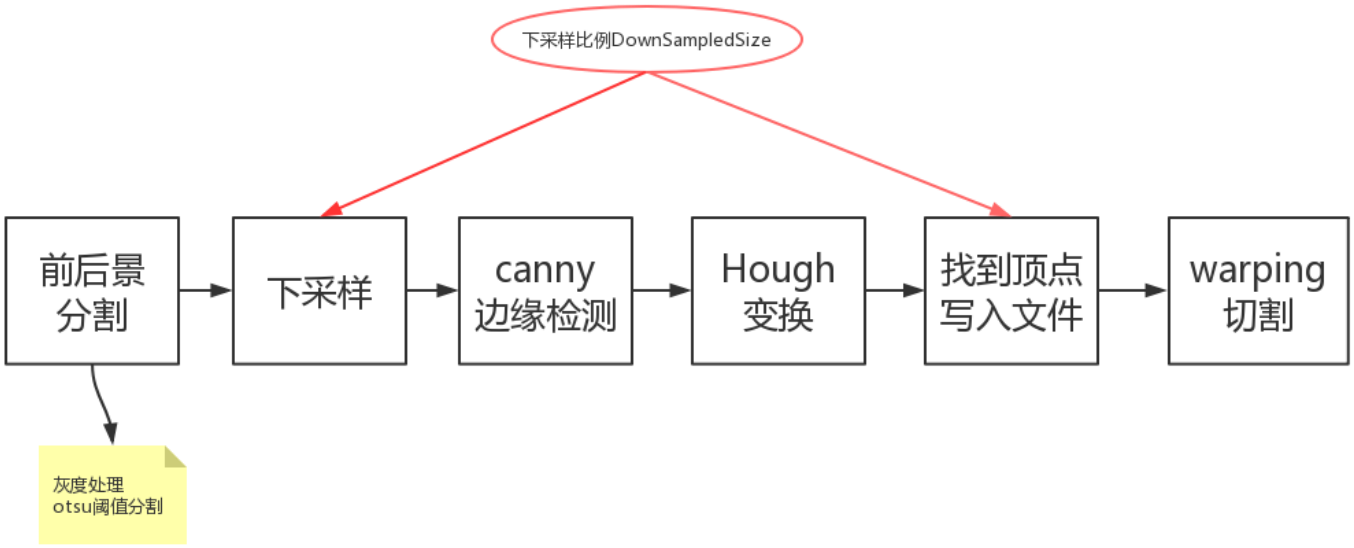
类的封装见文档最后。

名片校正

主要为**找到顶点**和 **warping 矫正**两个部分。

通过前后景分割、canny 边缘检测、Hough 变换找到 4 个顶点，用 c++语言，之前的代码上稍作改动；

warping 是用 python 的 opencv 的工具来做矫正。



1. 找到顶点

在 cmd 下执行命令即可。

- 在 window 下 g++ 编译基于 CImg 的程序，执行如下：

  g++ -o main.exe main.cpp -O2 -lgdi32

- 在 linux 下 g++编译基于 CImg 的程序，执行如下：

  g++ -o main.exe main.cpp -O2 -L/usr/X11R6/lib -lm -lpthread -lX11

（ps: 由于之前都是用 vs2019 直接调试执行的，所以在 main 文件的头文件有轻微的差别，vs 如下）

```
#include "EdgeDetect.cpp"
#include "Tool.cpp"
#include "concon.cpp"
#include <cstdlib>
#include <time.h>
```

如果用 g++编译，头文件应该全部换成 h.

下面具体介绍**步骤**以及在**每一个步骤之后达到的效果**：

1.1 前后景分割

这个之前写过，原理不再赘述，如下：

伪算法：

1. 对输入图像进行灰度处理；
2. otsus 算法求出阈值
   2.1 求出标准直方图
   2.2 求出使类间方差 ICV 最大的阈值 t
3. 根据阈值将图像二值化；
4. 图像反转。

代码如下：

```
CImg<float> EdgeDetect::Binarization(CImg<float> srcImg, float alpha)
{
    CImg<float> img = RGBtoGray(srcImg);
    int otsusValue = otsu(img, alpha);
    cimg_forXY(img, x, y)
    {
        img(x, y) = img(x, y) >= otsusValue ? 255 : 0;
    }
    return img;
}
```

其中，otsu 的代码为：

```
void normalizedHistogram(CImg<int> img, double* hist)
{
    int height = img.height();
    int width = img.width();
    int N = height * width;
    CImg<double> histImg = img.histogram(256, 0, 255);
    //histImg.display_graph("Histogram", 3);
    int i = 0;
    cimg_forXY(histImg, x, y)
    {
        hist[i] = histImg(x, y) = double(histImg(x, y) / N);
        //cout << histImg(x, y) << " " << hist[i] << endl;
        i++;
    }
    //histImg.display_graph("Histogram", 3);
}

int otsu(CImg<float>& image, float alpha)
{
    double hist[256];
    normalizedHistogram(image, hist);

    double omega[256];
    double mu[256];

    omega[0] = hist[0];
    mu[0] = 0;
    for (int i = 1; i < 256; i++)
    {
        omega[i] = omega[i - 1] + hist[i];
        mu[i] = mu[i - 1] + i * hist[i];
    }
    double mean = mu[255];
    double max = 0;
    int k_max = 0;
    for (int k = 1; k < 255; k++)
    {
        double PA = omega[k];
        double PB = 1 - omega[k];
        double value = 0;
        if (fabs(PA) > 0.001 && fabs(PB) > 0.001)
        {
            double MA = mu[k] / PA;
            double MB = (mean - mu[k]) / PB;
            value = pow(PA, alpha) * (MA - mean) * (MA - mean) + pow(PB, alpha) * (M
B - mean) * (MB - mean);
```

```
            if (value > max)
            {
                max = value;
                k_max = k;
            }
        }
        //qDebug() <<k << " " << hist[k] << " " << value;
    }
    return k_max;
}
```

1.2 下采样

即缩小图像，效果大概和 resize 相同，不过下采样是按比例缩小图像，resize 是重新设定宽高。

```
CImg<float> EdgeDetect::downSampling(CImg<float> img, int num) {
    CImg<float> downed = CImg<float>(img._width / num, img._height / num, 1, 1);
    cimg_forXY(downed, x, y)
    {
        int sum = 0;
        for (int xx = num * x; xx < num * (x + 1); xx++) {
            for (int yy = num * y; yy < num * (y + 1); yy++)
                sum += img(xx, yy);
        }
        downed(x, y) = sum / pow(num, 2);
    }
    return downed;
}
```

效果如下：

原图：

生成图：

名下变小了很多，有利于提升整个算法的速度。

1.3 Canny 检测

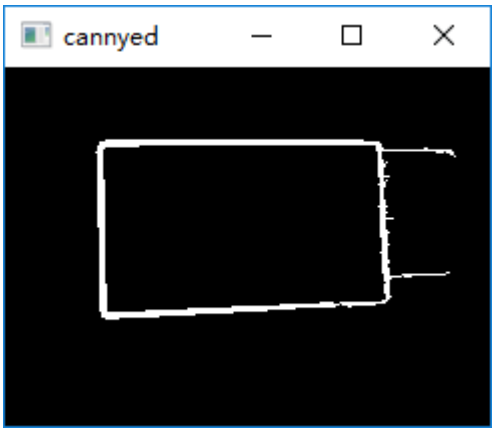
这个就是用了之前的 canny 代码，代码太多就不贴了。

参数以及在代码中的调用方法如下：

```
#define gFilterX 5
#define gFilterY 5
#define sigma 1
#define thresholdLow 120
#define thresholdHigh 140
Concon canny;
    CImg<float> result = canny.canny(img, gFilterX, gFilterY, sigma, thresholdLow, thresh
oldHigh);
```

效果如下，原图为：

生成图：



1.4 Hough 变换

伪算法：

1. Hough 空间变换
2. 在 Hough 空间中找到局部最大值点
3. 将所有的局部最大值点进行投票排序
4. 前 4 个最大值点即为我们要找的边缘
5. 根据边缘找到 4 个交点
6. 将这 4 个交点写入 txt 文件

主要代码如下：

```
houghImage = houghTransform(imgCannyed);    //hough 变换
houghReCompute(houghImage);      //求局部峰值
findEdge(imgDowned);             //在图中找到这个 4 条边
findVertex();                    //根据 4 条边找到 4 个顶点
drawVertexLines(outputImage, DownSampledSize);  //顶点信息，在原图中标出
```

其中，

函数 `CImg<float> EdgeDetect::houghTransform(CImg<float> img)` 为：

```
CImg<float> EdgeDetect::houghTransform(CImg<float> img) {
```

```
    int width = img._width, height = img._height, maxLength, row, col;
    maxLength = sqrt(pow(width / 2, 2) + pow(height / 2, 2));
    int w = maxLength;
    int h = thetaSize;
    CImg<float> houghImage = CImg<float>(w, h);
    houghImage.fill(0);

    cimg_forXY(img, x, y) {
        if (img(x, y) == 255) {
            //int x0 = x / 2, y0 = y / 2 ;
            int x0 = x - width / 2, y0 = height / 2 - y;
            for (int i = 0; i < thetaSize; i++) {
                int rho = x0 * setCos[i] + y0 * setSin[i];
                if (rho >= 0 && rho < maxLength) {
                    houghImage(rho, i)++;
                }
            }
        }
    }
    //houghImage.display("HOUGH");
    return houghImage;
}
```

函数 `void EdgeDetect::houghReCompute(CImg<float>& houghImage)` 为：

```
void EdgeDetect::houghReCompute(CImg<float>& houghImage) {
    int width = houghImage._width, height = houghImage._height, size = windowSize, max;
    for (int i = 0; i < height; i += size / 2) {
        for (int j = 0; j < width; j += size / 2) {
            max = getMaxHough(houghImage, size, i, j);
            for (int y = i; y < i + size; ++y) {
                for (int x = j; x < j + size; ++x) {
                    if (houghImage._atXY(x, y) < max) {
                        houghImage._atXY(x, y) = 0;
                    }
                }
```

```
            }
        }
    }
    int count = 0;
    cimg_forXY(houghImage, x, y) {
        if (houghImage(x, y) != 0) {
            cout << "x:" << x << "  y:" << y << "vote: " << houghImage(x, y) << endl;
            count++;
            if (myLines.empty())
                myLines.push_back(new Line(x, y, houghImage(x, y)));
            else {
                int i;
                for (i = 0; i < myLines.size(); i++) {
                    int dotrho = myLines[i]->rho;
                    int dottheta = myLines[i]->theta;
                    if (abs(x - dotrho) <= 5 && abs(y - dottheta) <= 5) {

                        myLines[i]->rho = x;
                        myLines[i]->theta = y;
                        break;
                    }
                }
                if (i == myLines.size()) {
                    myLines.push_back(new Line(x, y, houghImage(x, y)));
                    cout << "(" << x << "," << y << ")" << endl;
                }
            }
        }
    }
    cout << count << endl;
    //houghImage.display("NEW HOUGH");
}
```

用到辅助函数 `int EdgeDetect::getMaxHough(CImg<float>& img, int& size, int& y, int& x):`

```
int EdgeDetect::getMaxHough(CImg<float>& img, int& size, int& y, int& x) {
    int width = (x + size > img._width) ? img._width : x + size;
```

```
    int height = (y + size > img._height) ? img._height : y + size;
    int max = 0;
    for (int j = x; j < width; j++) {
        for (int i = y; i < height; i++) {
            max = (img(j, i) >= max) ? img(j, i) : max;
        }
    }
    return max;
}
```

函数 `void EdgeDetect::findEdge(CImg<float>& img)` 为：

```
void EdgeDetect::findEdge(CImg<float>& img) {
    sort(myLines.begin(), myLines.end(), cpm);
    int width = img._width, height = img._height, maxLength;
    edge = CImg<float>(width, height, 1, 1, 0);
    for (int i = 0; i < pointNumber; i++) {
        int theta = myLines[i]->theta, rho = myLines[i]->rho;
        cimg_forXY(edge, x, y) {
            int x0 = x - width / 2, y0 = height / 2 - y;
            if (rho == (int)(x0 * setCos[theta] + y0 * setSin[theta])) {
                edge(x, y) += 255.0 / 2;
                //imgtemp(x, y, 0, 2) = 255;
            }
        }
    }
}
```

函数 `void EdgeDetect::findVertex()` 为：

```
void EdgeDetect::findVertex() {
    unsigned char red[3] = { 255, 0, 0 };
    //img(0,0) - img(width-1,height-1)
    for (int y = 1; y < imgDowned._height - 1; y++) {
        for (int x = 1; x < imgDowned._width - 1; x++) {
            int arr[9];
            arr[0] = edge(x, y);
```

```
            arr[1] = edge(x + 1, y);
            arr[2] = edge(x, y + 1);
            arr[3] = edge(x + 1, y + 1);

            if (arr[0] + arr[1] + arr[2] + arr[3] >= 255.0 * 3 / 2) {


                if (myVertexs.empty())
                    myVertexs.push_back(new Vertex(x, y));
                else {
                    int i;
                    for (i = 0; i < myVertexs.size(); i++) {
                        int dotx = myVertexs[i]->x;
                        int doty = myVertexs[i]->y;
                        if (abs(x - dotx) <= 15 && abs(y - doty) <= 15) {
                            myVertexs[i]->x = x;
                            myVertexs[i]->y = y;
                            break;                          }
                    }
                    if (i == myVertexs.size())
                        myVertexs.push_back(new Vertex(x, y));
                }
            }
        }
    }
}
```

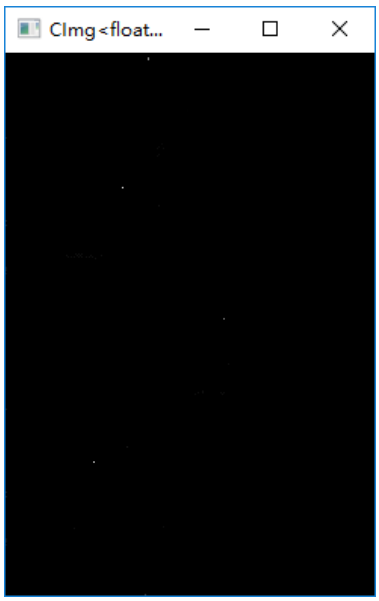
函数 `void EdgeDetect::drawVertexLines(CImg<float>& img, int downSampledSize)` 为：

```
void EdgeDetect::drawVertexLines(CImg<float>& img, int downSampledSize)
{
    sort(myVertexs.begin(), myVertexs.end(), compare1);//y 方向排序
    sort(myVertexs);     //x 方向排序

    string txtPath = "Vertex4_txt\\";
    CreateFolder(txtPath);
    txtPath = txtPath + name + ".txt";
```

```
    cout << "name:" << name << endl;
    cout << "txtPath:" << txtPath << endl;
    ofstream file(txtPath.c_str(), fstream::out);
    if (file) cout << " new file created" << endl;
    else cout << "failed!!\n";

    for (int i = 0; i < myVertexs.size(); i++) {
        cout << i << " " << (myVertexs[i]->x) * DownSampledSize << " " << (myVertexs[i]->
y) * DownSampledSize << endl;
        file << (myVertexs[i]->x) * DownSampledSize << " " << (myVertexs[i]->y) * DownSam
pledSize << endl;
    }
    file.close();

    const double yellow[] = { 255, 255, 0 };
    unsigned char red[3] = { 255, 0, 0 };
    for (int i = 0; i < myVertexs.size(); i++) {
        int x = myVertexs[i]->x;
        int y = myVertexs[i]->y;
        img.draw_circle(x * downSampledSize, y * downSampledSize, 50, red);
    }
}
```

其中，直接生成的 hough 图像为：

找到局部峰值之后：



最后，在原图中标出 4 个顶点如下：

1.5 顶点排序

写入顶点前排序：

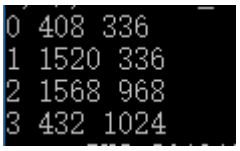
```
sort(myVertexs.begin(), myVertexs.end(), compare1); //y 方向排序
sort(myVertexs);      //x 方向排序
```

代码如下：

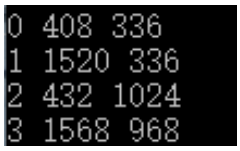
```
bool compare1(Vertex* a, Vertex* b) {
    return a->y < b->y;
}
void sort(vector<Vertex*>& Vertexs) {
    if (Vertexs.size() > 3) {
        if (Vertexs[0]->x > Vertexs[1]->x) {
            Vertex* temp = new Vertex(Vertexs[0]->x, Vertexs[0]->y);
            Vertexs[0] = Vertexs[1];
            Vertexs[1] = temp;
        }
        if (Vertexs[2]->x > Vertexs[3]->x) {
            Vertex* temp = new Vertex(Vertexs[2]->x, Vertexs[2]->y);
            Vertexs[2] = Vertexs[3];
            Vertexs[3] = temp;
        }//2,3
    }
}
```

排序很简单，没有用通用的算法，因为知道是 4 个，而且名片顶点的位置本身就很规整。

由于名片基本都是正放的，因此在存储顶点信息的时候，我先将名片按照位置进行排序，最后里面的格式为左上、右上、左下、右下：

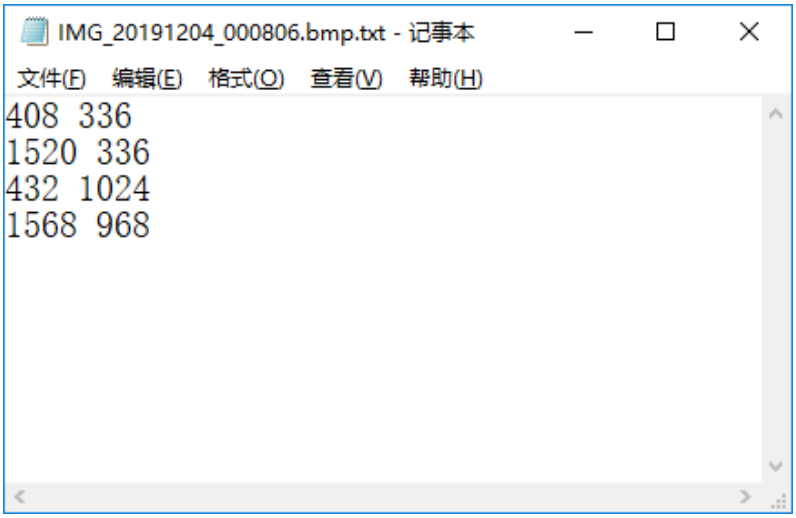


（排序之前）



（排序之后）

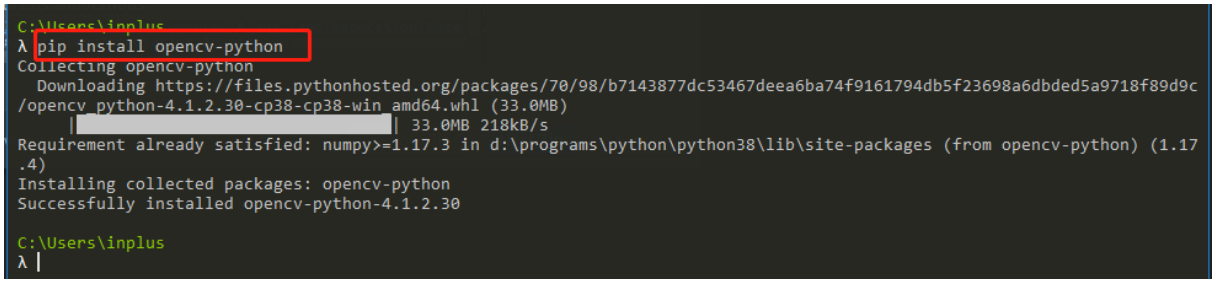
最后生成的 txt 文件如下：



2. Warping 矫正

这个是调用了 python 的 opencv 库。

环境配置如下，命令 **pip install opencv-python**：



基于透视的图像矫正如下：

伪算法：

1. 获取图像
2. 获取图像四个顶点
3. 形成变换矩阵
4. 透视变换

主函数代码如下：

```
def main():
    """
    # 原图像，获得名称
    # 找到对应的txt 文件
    # 切割
    # 保存图像
    :return:
    """
    __dirpath = 'C:\\Users\\inplus\\Desktop\\Project1\\Project1'
    pathSource = os.path.join(__dirpath, 'Dataset')
    pathTxt = os.path.join(__dirpath, 'Vertex4_txt')
    pathDest = os.path.join(__dirpath, 'cutImg')

    for file in os.listdir(pathSource):
        print('正在处理图像' + file + '...')
        img = getImg(pathSource,file)
        pts1, pts2 = getPositions(pathTxt, file)
        print('4 个顶点的坐标为: \n', pts1)
        result = Warping(img, pts1, pts2)
        cv2.imshow("result_img", result)
        cv2.imwrite(os.path.join(pathDest, file), result)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
    print('全部处理完毕！')
```

下面具体介绍每个步骤：

2.1 傅里叶变换原理

基于傅里叶变换的图像矫正原理如下：

2 维图像的傅立叶变换可以用以下数学公式表达:

$$F(k,l)=\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}f(i,j)e^{-i2\pi(\frac{ki}{N}+\frac{lj}{N})}$$

$$e^{ix}=\cos x+i\sin x$$

式中 f 是空间域(spatial domain)值， F 则是频域(frequency domain)值。

转换之后的频域值是复数， 因此，显示傅立叶变换之后的结果需要使用实数图像(real image) 加虚数图像(complex image), 或者幅度图像(magitude image)加相位图像(phase image)。 在实际的图像处理过程中，仅仅使用了幅度图像，因为幅度图像包含了原图像的几乎所有我们需要的几何信息。然而，如果你想通过修改幅度图像或者相位图像的方法来间接修改原空间图像，你需要使用逆傅立叶变换得到修改后的空间图像，这样你就必须同时保留幅度图像和相位图像了。

由于数字图像的离散性，像素值的取值范围也是有限的。比如在一张灰度图像中，像素灰度值一般在 0 到 255 之间。 因此，我们这里讨论的也仅仅是离散傅立叶变换(DFT)。 如果你需要得到图像中的几何结构信息，那你就要用到它了。

在频域里面，对于一幅图像，高频部分代表了图像的细节、纹理信息；低频部分代表了图像的轮廓信息。如果对一幅精细的图像使用低通滤波器，那么滤波后的结果就剩下了轮廓了。这与信号处理的基本思想是相通的。如果图像受到的噪声恰好位于某个特定的“频率”范围内，则可以通过滤波器来恢复原来的图像。傅里叶变换在图像处理中可以做到：图像增强与图像去噪，图像分割之边缘检测，图像特征提取，图像压缩等等。

（此处参考于博文）

2.2 获取顶点信息

首先读取 txt 文件，由于之前已经按照顶点的位置排过序了，这里就默认是按照左上、右上、左下、右下

的数据，处理如下：

```
def getPositions(pathTxt, file):
    path = os.path.join(pathTxt, file+'.txt')
    # 原图中(左上、右上、左下、右下)
    pos = np.loadtxt(path)
    pos = np.float32(pos)
    # 校正后
    # pts1 = np.float32([[488, 560], [2312, 592], [408, 1688], [2392, 1656]])
    # pts2 = np.float32([[0, 0], [2312 - 488, 0], [0, 1688 - 560], [2312 - 488, 1688 - 560]])
    width = int(pos[1][0] - pos[0][0])
    height = int(pos[3][1] - pos[0][1])
    pos2 = np.float32([[0, 0],
                       [width, 0],
                       [0, height],
                       [width, height]])
    return pos, pos2
```

用 pos 保存源文件中顶点的位置，pos2 表示期望变换后这 4 个点在哪个位置，那么应该在一张图的 4 个顶点。

注释中有个例子可以很好得展示这种关系：

原图中，顶点得位置分别是`[[488, 560], [2312, 592], [408, 1688], [2392, 1656]]`

那么，在变换后：

这张图的宽 `width` 应该是右上点.x - 左上点.x,

高 `height` 应该是右下角.y - 左上角.y

因此，最后 4 个点的位置，应该是`[[0, 0],[width, 0],[0, height],[width, height]]`

2.3 透视变换

获得了顶点信息之后，就可以生成变换矩阵来透视变换了，代码为：

```
def Warping(img, pts1, pts2):
    # 原图中书本的四个角点(左上、右上、左下、右下),与变换后矩阵位置
    width = int(pts2[3][0])
    height = int(pts2[3][1])
    # 生成透视变换矩阵；进行透视变换
    M = cv2.getPerspectiveTransform(pts1, pts2)
    result = cv2.warpPerspective(img, M, (width, height))
    # 显示
```

```
    # cv2.imshow("original_img",img)
    # cv2.imshow("result", result)
    cv2.destroyAllWindows()
    return result
```

其中，`M = cv2.getPerspectiveTransform(pts1, pts2)`

`result = cv2.warpPerspective(img, M, (width, height))`

两行是调用 opencv 的库函数。

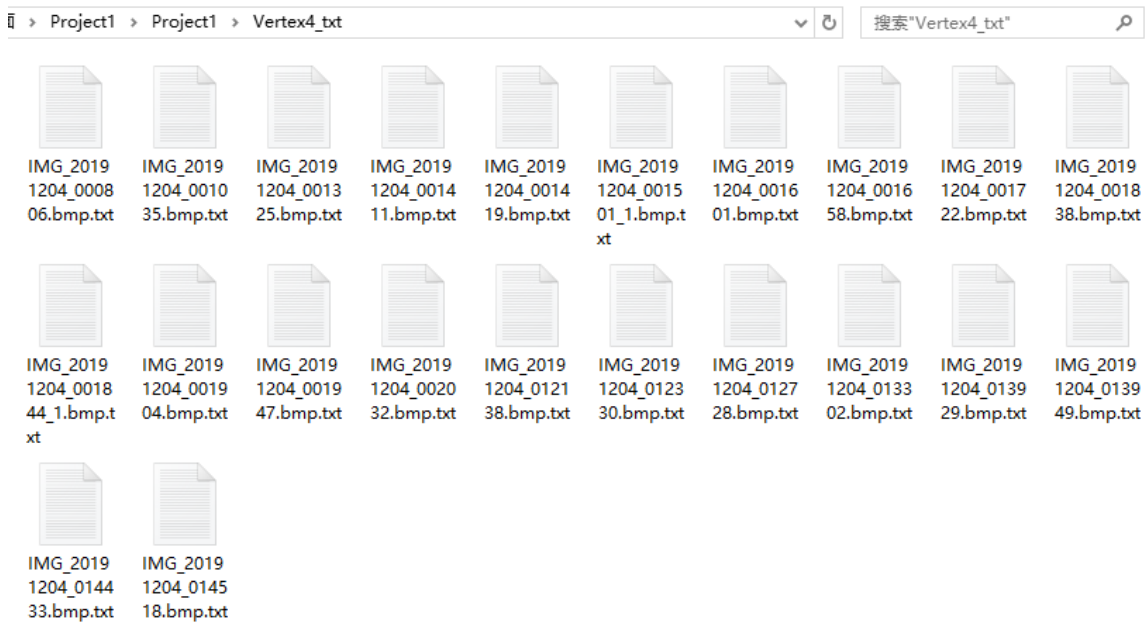
最后将结果保存起来，即可。

3. 实验过程

- 执行 g++命令，如下：

```
C:\Users\inplus\Desktop\Project1\Project1
λ g++ -o main.exe main.cpp -O2 -lgdi32

C:\Users\inplus\Desktop\Project1\Project1
λ main.exe
```

最后所有的名片的顶点信息：



获得所有顶点信息之后，就可以利用 python 脚本矫正图像了。

- 执行 pytho 脚本

```
C:\Users\inplus\Desktop\py>python test.py
正在处理图像IMG_20191204_000806.bmp...
正在处理图像IMG_20191204_001325.bmp...
正在处理图像IMG_20191204_001411.bmp...
正在处理图像IMG_20191204_001501_1.bmp...
正在处理图像IMG_20191204_001722.bmp...
正在处理图像IMG_20191204_001838.bmp...
正在处理图像IMG_20191204_001844_1.bmp...
正在处理图像IMG_20191204_001904.bmp...
正在处理图像IMG_20191204_012728.bmp...
正在处理图像IMG_20191204_013302.bmp...
正在处理图像IMG_20191204_013929.bmp...
正在处理图像IMG_20191204_013949.bmp...
正在处理图像IMG_20191204_014433.bmp...
正在处理图像IMG_20191204_014518.bmp...
全部处理完毕！
```

最后生成的图片如下：

主要部分切割

主要流程如下，根据已经得到的名片，得到名片上的不同群体。

代码如下：

```
void ImgSeg::findTarget(string input, string mode, string output, int index)
{
    Binarization(input, 1.0); //图片预处理
    img.dilate(8); //膨胀以及腐蚀
    img.erode(8);
    cptConBlocks(); //求连通块，筛选出需要的元素
    deleteBlocks();//删除不符合条件的联通块，以及将联通块加入群
    deleteGroups(); //删除不符合条件的群
}
```

下面具体介绍每一个步骤。

4. 图像预处理

这里和之前的作业思路相似，Otsu 法自动阈值法将前后景分割，如下：

```
void ImgSeg::Binarization(string input,  float alpha)
{
    CImg<float> tempImg;
    tempImg.load(input.c_str());
    tempImg.resize(840, 600);
    img = RGBtoGray(imgOriginal);       //灰度处理

    int otsusValue = otsu(img, alpha);//前后景分割阈值 otsus 算法
    cout << "otsusValue = " << otsusValue << endl;
    cimg_forXY(img, x, y)
    {
        img(x, y) = img(x, y) >= otsusValue ? 255 : 0;
    }
    img = converse(img);
}
```

注意把所有的图像都 resize 到一个大小，方便后续的批量处理。

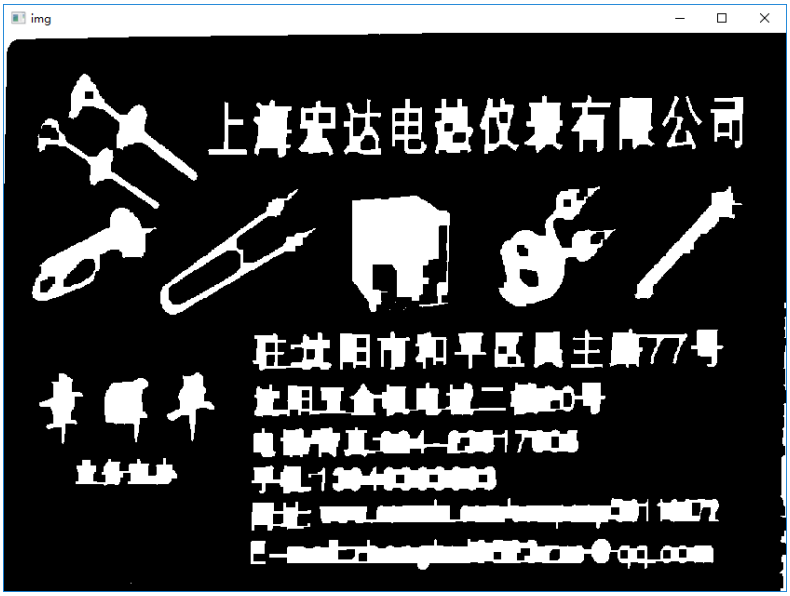
5. 膨胀以及腐蚀操作

Dilate 和 erode 函数即可，参数调的比较大，最后的像素点基本都黏在一起了。

最后的效果如下：

处理前：

处理后：



6. 求联通块（2-step 算法）

下面这符图中，如果考虑 4 邻接，则有 3 个连通区域；如果考虑 8 邻接，则有 2 个连通区域。

项目中实现的是 8 邻接。

伪算法：

1. 逐行扫描图像，我们把每一行中连续的白色像素组成一个序列称为一个团(run)，并记下它的起点 start、它的终点 end 以及它所在的行号。
2. 对于除了第一行外的所有行里的团，如果它与前一行中的所有团都没有重合区域，则给它一个新的标号 label；如果它仅与上一行中一个团有重合区域，则将上一行的那个团的标号 label 赋给它；如果它与上一行的 2 个以上的团有重叠区域，则给当前团赋一个相连团的最小标号，并将上一行的这几个团的标记写入等价对，说明它们属于一类。
3. 将等价对转换为等价序列，每一个序列需要给一相同的标号，因为它们都是等价的。从 1 开始，给每个等价序列一个标号
4. 遍历开始团的标记，查找等价序列，给予它们新的标记 label。
5. 将每个团的标号填入标记图像中。
6. 结束。

至此，我们求出了每个团 run 的标号 label，因此现在把同属一个 label 的团的像素点放入一个数据结构。

将每个连通块打包成块：

遍历所有团，找出属于一个连通块的团中的所有像素点。

将这个连通块标号 label 和属于这个 label 的所有像素点放入一个 Block。

7. 联通块聚集为群

群的数据结构如下：

```
struct Group {
    bool isValid;
    vector<int> boundry;
    vector<vector<int>> blocks;
    Group(bool v, vector<int> _boundry, vector<vector<int>> _blocks) {
        isValid = v;
        boundry = _boundry;
        blocks = _blocks;
    }
```

```
};
//在私有变量中申明一个 Group 类型的 vector
vector<Group> myGroups;            //集群
```

这个步骤很关键。步骤如下：

伪算法：

1. 遍历所有的块
2. 找出符合条件的块，加入群。
   a) 如果有群，求这个联通块与这个联通块的距离
      i. 距离小于阈值，则加入这个群。
      ii. 否则，新建一个群。
   b) 没有群，新建一个群。

代码如下：

```
/*筛选合格的联通块，返回 true*/
bool ImgSeg::isValidTest(map<int, vector<Plot>>::iterator iter)
{
    //
     Code,判断这个联通块是否符合条件。在下一点中列出条件。
    //
    // 遍历 myGroup，查看有无相近的群
    // 如果找到一个，那么就加入这个群，如果找不到，就新建一个然后加入
    int index = findGroup(x_low, x_high, y_low, y_high);
    if (-1 != index)
        pushToGroup(myGroups[index], x_low, x_high, y_low, y_high);
    else {
        addNewGroup(x_low, x_high, y_low, y_high);
    }
    cout << "found!   ";
    return true;
}
```

- 首先将不符合要求的联通块排除，不考虑加群。

规则：

3. 考虑到膨胀腐蚀之后的字都连成一排，因此对宽不做要求。
4. 在联通块的高 height 上，以名片上人名的高为最大值，如果比人名还高，排除；
5. 如果联通块的像素点特别少，排除；
6. 如果联通块的尺寸很小，排除；
7. 如果联通块的宽高比过于小或者过于大，排除；

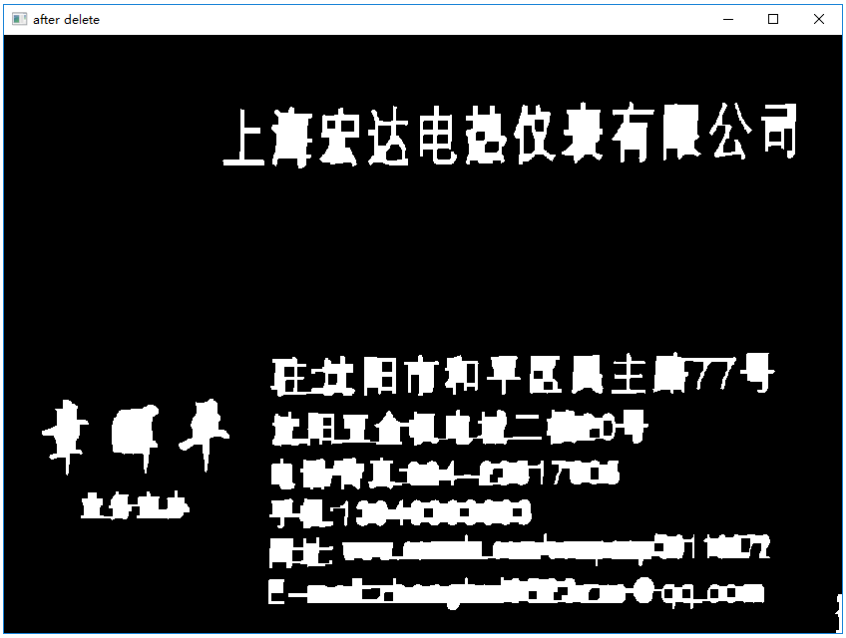
代码：

```
    cout << iter->second.size() << " ";
    int x_low, x_high, y_low, y_high;
    x_low = y_low = INT_MAX;    //正无穷
    x_high = y_high = INT_MIN;  //负无穷
    //找出边界
    for (Plot plot : iter->second) {
        int x = plot.x;
        int y = plot.y;
        x_low = x <= x_low ? x : x_low;
        x_high = x >= x_high ? x : x_high;
        y_low = y <= y_low ? y : y_low;
        y_high = y >= y_high ? y : y_high;
    }
    //连通块的宽、高、宽高比
    int width = abs(x_high - x_low);
    int height = abs(y_high - y_low);
    float size = (float)width / (float)height;

    if (iter->second.size() <= 70) return false;
    if ((width > 90 && height > 90) ) return false;
    if (width < 10 && height < 10) return false;
    if (size > 20 || size < 0.05) return false;
```

对于不符合要求而被排除的联通块，可以在图片上删去查看效果，代码如下：

```
void ImgSeg::deleteBlocks() {
    for (auto iter = blocks.begin(); iter != blocks.end(); iter++) {
        //求边界
        if (!isValidTest(iter)) {   //验证连通块有效性，求有效块的边界
            for (Plot plot : iter->second) {
                img(plot.x, plot.y) = 0;//像素点删除
            }
        }
    }
}
```

最后有效的联通块显示如下：



- 对于符号要求的联通块，我们可以将其加入群，如果附近有群就加入，如果没有则自己建立一个新的群，也就是上面代码中的 findGroup 函数，返回这个群的序号。

```
/*找到合群的序号*/
int ImgSeg::findGroup(int x_low,int x_high, int y_low, int y_high) {
    Point a(x_low, y_low);
    int width = x_high - x_low;
    int height = y_high - y_low;
    int MIN = 999999, pos = -1;

    for (int i = 0; i < myGroups.size(); i++) {
        //cout << "Group " << i << ":" << endl;
        vector<int> temp = myGroups[i].boundry;
        vector<vector<int>> blks = myGroups[i].blocks;
        int minDist = 999999;   //和每一群的最小距离
        for (auto ele : blks) {
            Point b(ele[0], ele[2]);
            int tempDistance = getDistance(a, b);
            if (tempDistance < minDist)
                minDist = tempDistance;
        }
        //cout << "minDist: " << minDist << endl;
        if (minDist < MIN) {
            pos = i;
```

```
            MIN = minDist;;
        }
    }
    if (MIN < 100) {
        cout << "MIN: " << MIN << " pos: " << pos << endl;
        return pos;
    }
    cout << "MIN: " << MIN << " pos: " << pos << endl;

    return -1;
}
```

其中，用到辅助函数 `int getDistance(Point a, Point b)`求两个点的距离，值得一提。

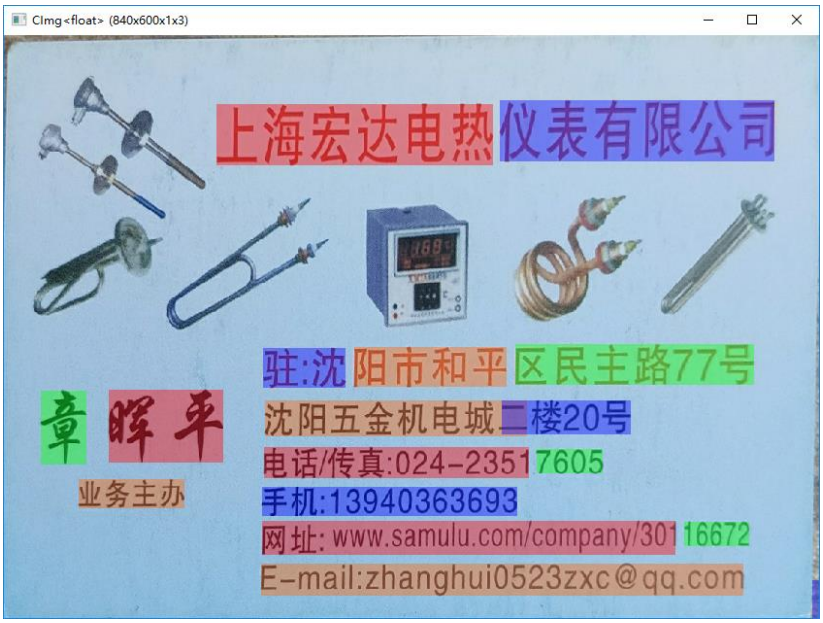
求一个**联通块**到一个**群**的距离，我定义为**这个联通块**到这个**群中所有联通块**的距离的最小值。

那么两个联通块的距离我定义为左上顶点的加权距离，这是因为在名片处理中，一个信息的文字基本在一行中，很少出现在多行的情况，所以如果两个联通块在 x 方向上的距离很小时，那么他们极有可能在一个群中。

在这里我的加权距离阈值是设置成了 100。

```
int getDistance(Point a, Point b) {
    int weight = 5;
    return sqrt(pow(a.x - b.x, 2) + pow(weight*(a.y - b.y), 2));
}
```

最后求得不同群用不同颜色标出如下：



可以看到由于联通块遍历的顺序为由上到下，因此，仍然出现了在一行的联通块在不同的群中的问题，下

面对于这个问题进行优化 。

8. “群“的处理

现在对于群要做一些优化，目标如下：

（1） 对于有重叠的群，合并。

（2） 对于两个群的距离相近而且在 y 上位置相近的群，合并。

合并操作的伪算法如下：

> **伪代码（合并）：**
> 1. 记录一个下标 pos, 以及它需要比较的下标 i
> 2. 对于 group[pos]，往后寻找和他重合的群，如果找到了，将那个群作废并更新这个群的信息：
>   a) i 那个群 isValid 设为 false
>   b) i 那个群的所有联通块加入到 pos 群
>   c) 同时 pos 的边界也需要更新。

代码如下：

```
void ImgSeg::combiGroups() {
    // 合并相交的群
    int len = myGroups.size();
    int pos = 0;
    for (pos = 0; pos < len; pos++) {
        for (int i = pos + 1; i < len; i++) {
            if (myGroups[i].isValid) {
                  //重合
                if (isOverLap(myGroups[pos], myGroups[i])) {
                    //第二个的 bloc 加入第一个 pos
                    //更新第一个 group 的信息
                    for (auto ele : myGroups[i].blocks) {
                        //ele. x_high, y_low, y_high
                        pushToGroup(myGroups[pos], ele[0], ele[1], ele[2], ele[3]);
                    }
                    myGroups[i].boundry = getBox(myGroups[i].blocks);
                    myGroups[i].isValid = false;
                }
            }
```

```
        }
        // 删除 false 的群
        vector<Group>::iterator iter = myGroups.begin();
        while (iter != myGroups.end()) {
            if (!(*iter).isValid) {
                cout << "erase a group!!";
                iter = myGroups.erase(iter);
            }
            else iter++;
        }
        len = myGroups.size();
    }
    //合并距离相近的群,合并 3 次
    int distance = 50;
    int count = 3;
    while (count--) {
        for (pos = 0; pos < len; pos++) {
            for (int i = pos + 1; i < len; i++) {
                if (myGroups[i].isValid) {
                    if (isClose(myGroups[pos], myGroups[i], distance)) {
                        //第二个的 bloc 加入第一个 pos
                        //更新
                        for (auto ele : myGroups[i].blocks) {
                            //ele. x_high, y_low, y_high
                            pushToGroup(myGroups[pos], ele[0], ele[1], ele[2], ele[3]);
                        }
                        myGroups[i].boundry = getBox(myGroups[i].blocks);
                        myGroups[i].isValid = false;
                    }
                }
            }
            // 删除 false 的群
            //代码同上，这里不展出
            len = myGroups.size();
        }
    }
}
```

在判断上，首先有一个判断是否重叠：

前提：

（1） 两个矩形的边均与 x 轴或 y 轴平行，即轴对齐的矩形

（2） 将第一个矩形记做 A，第二个矩形记做 B

（3） 判断矩阵 A 与矩阵 B 是否重叠（边沿重叠也认为是重叠）

先解决出“不重叠”的情况，如图，B 矩阵，可能在 A 的左侧、右侧、上侧、下侧。如果用公式表示，即

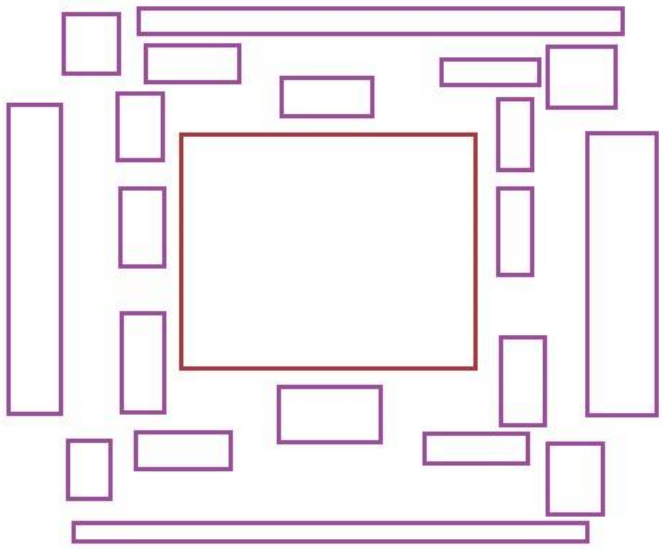
$$(p2.y \leq p3.y) \vee (p1.y \geq p4.y) \vee (p2.x \leq p3.x) \vee (p1.x \geq p4.x)$$

则，两个矩阵重叠时，公式为

$$\neg[(p2.y \leq p3.y) \vee (p1.y \geq p4.y) \vee (p2.x \leq p3.x) \vee (p1.x \geq p4.x)]$$

根据德·摩根定律可转换为

$$(p2.y > p3.y) \wedge (p1.y < p4.y) \wedge (p2.x > p3.x) \wedge (p1.x < p4.x)$$



最后，代码如下：

```
/**
 * @brief 判断两个轴对齐的矩形是否重叠
 * @param rc1 第一个矩阵
 * @param rc2 第二个矩阵
 * @return 两个矩阵是否重叠（边沿重叠，也认为是重叠）
 */
bool ImgSeg::isOverLap(Group a, Group b) {
    int x1 = a.boundry[0], y1 = a.boundry[2];
    int width1 = a.boundry[1] - a.boundry[0];
    int height1 = a.boundry[3] - a.boundry[2];
```

```
    int x2 = b.boundry[0], y2 = b.boundry[2];
    int width2 = b.boundry[1] - b.boundry[0];
    int height2 = b.boundry[3] - b.boundry[2];

    if (x1 + width1 > x2&&
        x2 + width2 > x1&&
        y1 + height1 > y2&&
        y2 + height2 > y1
        )
        return true;
    else
        return false;
}
```

判断两个矩形相近的代码：

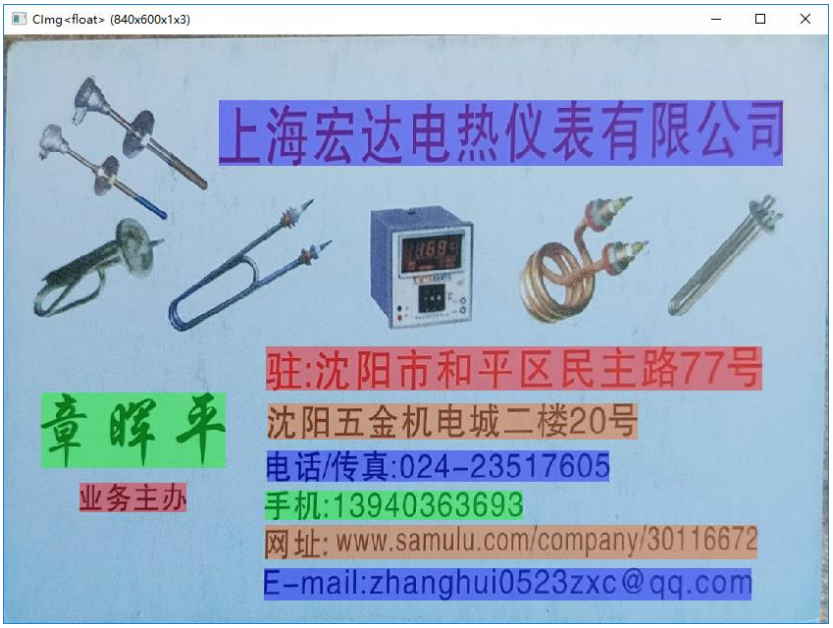
```
/**
 * @brief 判断两个轴对齐的矩形是否相近，相近阈值为 distance
 * @param a 第一个矩阵
 * @param b 第二个矩阵
 * @return 两个矩阵是否相近（距离<=distance）
 */
bool ImgSeg::isClose(Group a, Group b, int distance) {
    int x1 = a.boundry[0], y1 = a.boundry[2];
    int width1 = a.boundry[1] - a.boundry[0];
    int height1 = a.boundry[3] - a.boundry[2];

    int x2 = b.boundry[0], y2 = b.boundry[2];
    int width2 = b.boundry[1] - b.boundry[0];
    int height2 = b.boundry[3] - b.boundry[2];

    int derta_Y = abs(a.boundry[2] - b.boundry[2]);
    if (derta_Y < 10) {
        // a 在 b 左边，比较 a 的右边界与 b 的左边界
        if (x1 < x2)
        {
            if (x1 + width1 + distance > x2)
```

```
                return true;
            return false;
        }
        // b 在 a 左边，比较 b 的右边界与 a 的左边界
        else {
            if (x2 + width2 + distance > x1)
                return true;
            return false;
        }
    }
    return false;
}
```

最后，效果如下：

合并前：

合并后：

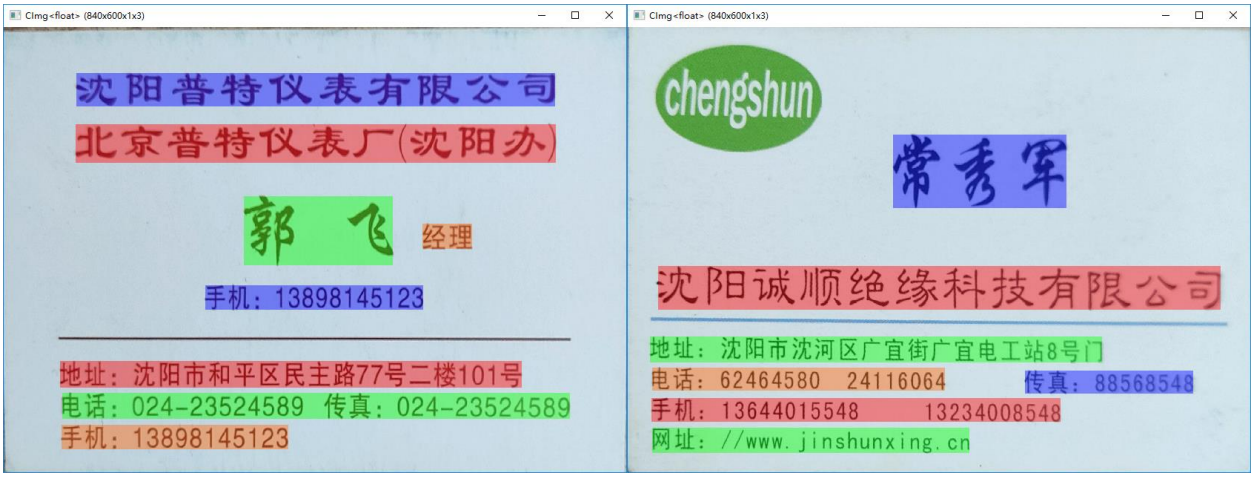


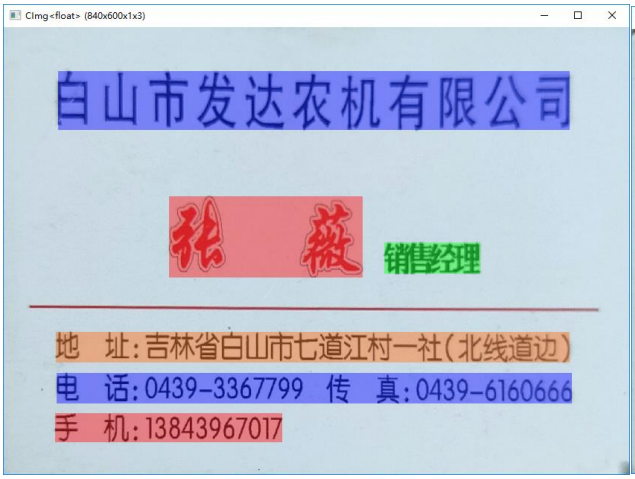
可以看到，效果还是很好的。

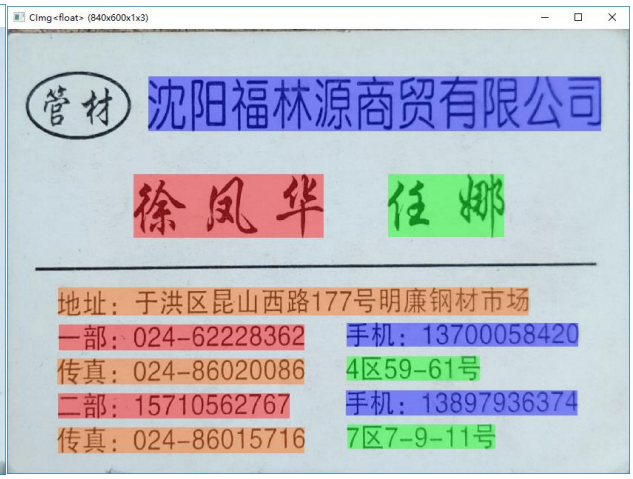
9. 处理所有的图

由于要调参数，因此有些特殊的名片参数不一样。

主函数为：

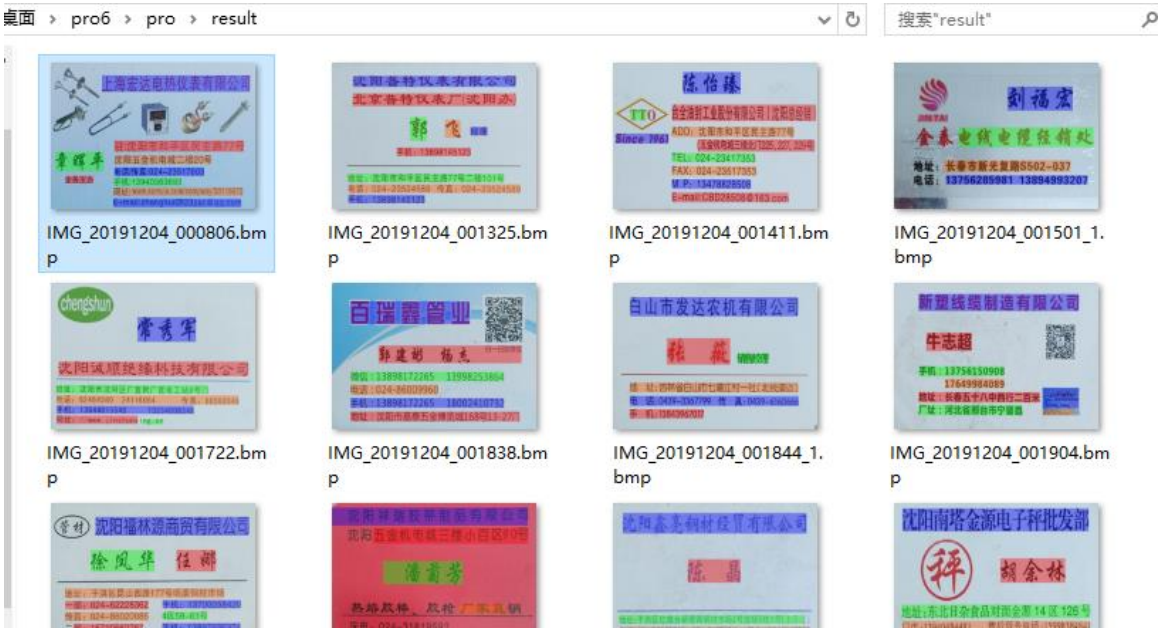






。。。。。。没有完全列出

放入文件夹 result\\中，如下：



实验总结

这个作业挺难的，主要需要调试 canny 参数，另外需要对于不同的名片调距离参数，使不同的元素放在一个群中。

类的封装

```
class ImgSeg {
```

```
public:
    ImgSeg() {}
    ~ImgSeg();
    void findTarget(string, string, string, string, int, int);//每个图像调用一次

private:
    void reset();
    void Binarization(string);                  //灰度处理，二值处理
    void Binarization(string, float);
    void cptConBlocks();
    void deleteBlocks();
    int findGroup(int, int, int, int);
    bool pushToGroup(Group&, int x_low, int x_high, int y_low, int y_high);
    bool addNewGroup(int x_low, int x_high, int y_low, int y_high);

    void deleteGroups();
    bool isOverLap(Group, Group);
    bool isClose(Group, Group, int);
    void combiGroups();
    void getImg(CImg<float> & );


    void getImgFiles(vector<vector<int> >& vc, int state);

    CImg<float> RGBtoGray(CImg<float>);
    CImg<float> BinSimple(CImg<float>, int);    //简单阈值处理，二值化
    int otsu(CImg<float>& image, float);
    CImg<float> converse(CImg<float>);
    void normalizedHistogram(CImg<int> img, double* hist);
    void fillRunVectors(CImg<int>& bwImage, int& NumberOfRuns,
        vector<int>& stRun, vector<int>& enRun, vector<int>& rowRun);
    void firstPass(vector<int>& stRun, vector<int>& enRun, vector<int>& rowRun, int NumberOfRun
s,
        vector<int>& runLabels, vector<pair<int, int>>& equivalences, int offset);
    void replaceSameLabel(vector<int>& runLabels, vector<pair<int, int>>& equivalence);

    void drawABox(vector<int>, int, string, float);
    vector<int> getBox(vector<vector<int>>&);
    vector<Plot> boundToPlots(vector<int> vc, int offset);
    vector<string> getDigit(int);

    bool isValidTest(map<int, vector<Plot>>::iterator iter);
    void checkDigit(vector<vector<vector<int>>> & imgs, vector<string>& strs);
    void sortVC(vector<vector<int> >& vc, int state);
    vector<CImg<float>> cutImgs(CImg<float>& img, vector<vector<int> >& vc, int state);
    vector<vector<int>> combiVC(vector<vector<int> >& vc);

    CImg<float> imgOriginal;//原图
```

```
    CImg<float> img;         //二值图
    CImg<float> imgForCut;   //用于剪切中间生成数字图像

    //key 是 label,values 是点集。
    map<int, vector<Plot>> blocks;          //连通块（label，点集）
    vector<Group> myGroups;                 //集群
    int myDistance;
    string MyInput;
    string myName;

    int myIndex;
    string myMode;
    string myOutput;
};

#endif
```

部分数据结构如下：

```
struct Plot {
    int x, y;
    Plot(int x_, int y_) :x(x_), y(y_) {}
};
struct Block {
    int label;
    vector<Plot> Plots;
    int x_low, x_high, y_low, y_high;//块边界

    Block(int _label) :label(_label) {
        Plots.clear();
    }
};
struct Group {
    bool isValid;
    vector<int> boundry;
    vector<vector<int>> blocks;
    Group(bool v, vector<int> _boundry, vector<vector<int>> _blocks) {
        isValid = v;
        boundry = _boundry;
        blocks = _blocks;
    }
};
```

提交文件

这个提交包括 c++项目文件夹、python 代码文件、report.pdf 实验报告

考虑到图像数据太大了，作业上传不了，所以只上传了部分的结果。

Pro

pro1 用于获得顶点数据

pro2 是 python 代码，用来矫正名片

pro3 用来分割一张名片上的不同部分