作业 4

输入图像:

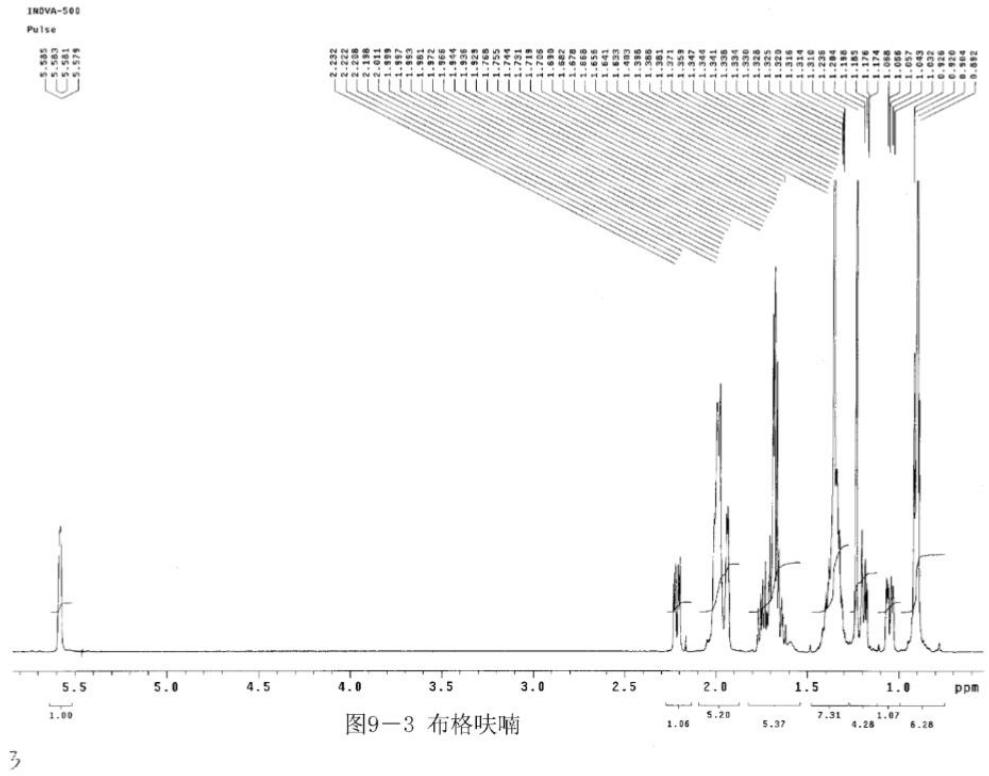


图9－3 布格呋喃

3

任务一：把图像的前后景分割.

前后景分割采用 OTSU 算法获得最佳阈值，OTSU 算法采用类间方差最大的阈值作为筛选阈值，具体步骤如下:

1. 获得图像的灰度直方图
2. 计算当前阈值下图像前景和后景的比例，分别为 w0,w1
3. 计算当前阈值下图像前后景平均灰度，分别为 u0,u1
4. 计算 g = w0*w1*(u0-u1)^2
5. 对所有灰度值计算 g，找到最大的 g 对应的 T 作为阈值

重要代码如下:

获得灰度图和灰度直方图:

这里用大小为 256 的 int 数组 gray_level 作为直方图:

```
void otsu::getGray()
{
    cimg_forXY(Src, x, y) {
        Gray(x, y) = 0.299 * Src(x, y, 0) + 0.587 * Src(x, y, 1) + 0.114 * Src(x, y, 2);
        int col = Gray(x, y);
        gray_level[col] ++;//灰度直方图
    }
}
```

找最佳阈值:

```
void otsu::findThreshold()
{
    double max_g = 0;
    int best_t = 0;
    for (int i = 0; i < 256; i++) {
        double w0 = 0;
        double w1 = 0;
        double u0 = 0;
        double u1 = 0;
        for (int j = 0; j < 256; j++) {
            if (j < i) {
                w1 += gray_level[j];
                u1 += gray_level[j] * j;
            }//背景的灰度平均值和背景点数量
            else {
                w0 += gray_level[j];
                u0 += gray_level[j] * j;
            }//前景的灰度平均值和前景点数量
        }
        int total = Gray._width * Gray._height;
        if (w1 != 0)
            u1 = u1 / w1;
        if (w0 != 0)
            u0 = u0 / w0;
        w1 = w1 / total;//计算前后景点比例
        w0 = w0 / total;
        double g = w0 * w1 * pow((u0 - u1), 2);
```

```
        if (g > max_g) {
            max_g = g;
            best_t = i;
        }
    }
    threshold = best_t;
}
```

根据最佳阈值进行前后景分割：

```
void otsu::seg()
{
    getGray();
    findThreshold();
    cimg_forXY(Gray, x, y) {
        if (Gray(x, y) > threshold) {
            Gray(x, y) = 255;
        }
        else {
            Gray(x, y) = 0;
        }
    }
    Gray.display();
    Gray.save(save_path.c_str());
}
```

分割前后的结果对比：

分割前：

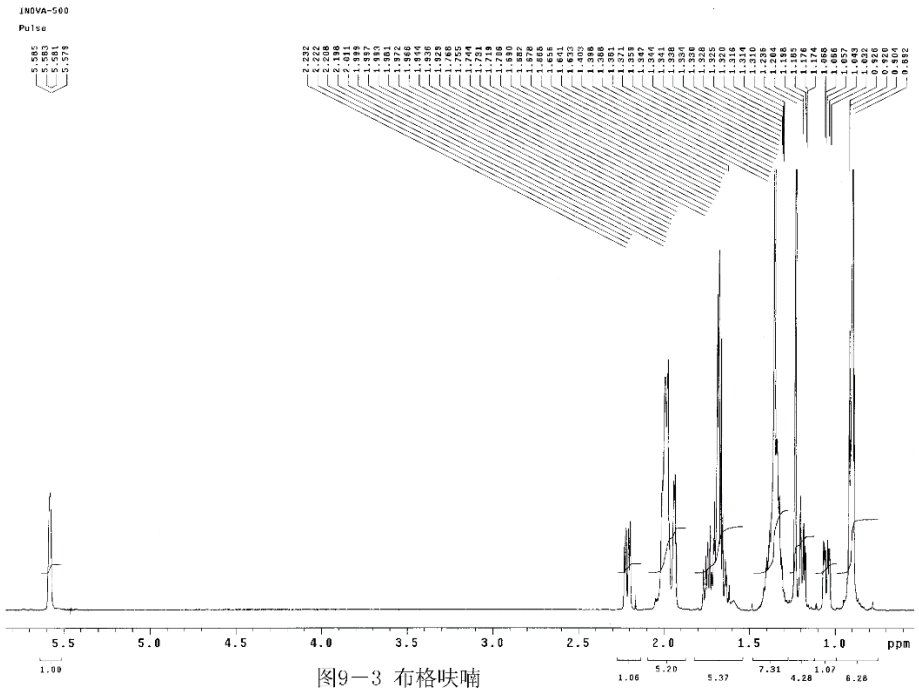


图9－3 布格呋喃

3

分割后:

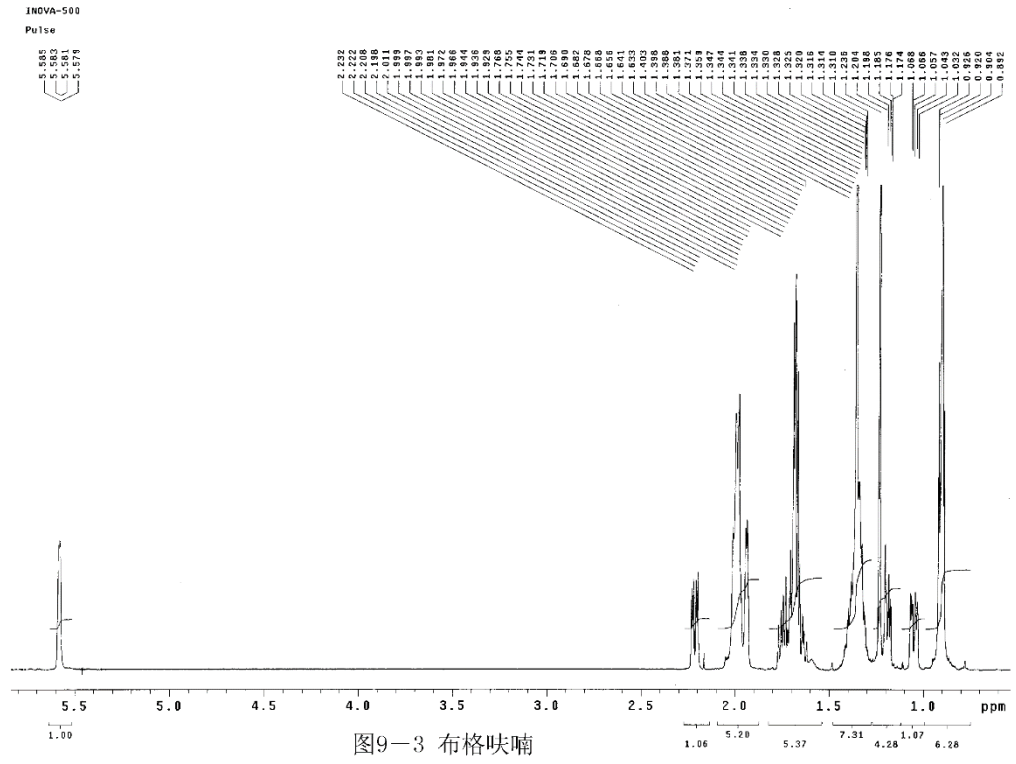


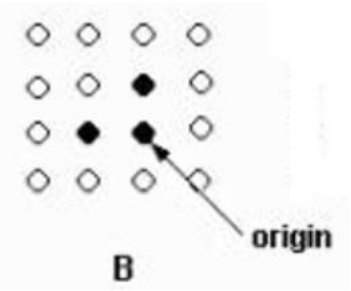
图9－3 布格呋喃

3

任务二： 把图像中的数字切割出来，基本思路如下:

- 先做图像的 Delate(膨胀操作).
- 求出图像中的连通块.
- 去除黑色像素大于 T(T=100)的联通块.
- 在原图上把联通块(黑色像素<=T)用红色框标记(即用红色框围住联通块)

1. 图像膨胀操作采用如下的结构元素:



具体做法是用 origin 在 Src 上遍历，如果 B 中为黑色的点对应在 Src 上也为黑色，那么结果图 result 上相应位置的点就为黑色。主要代码如下:

```
1.  void Delate::Do_delate()
2.  {
3.      cimg_forXY(Src, x, y) {
4.          if (x == 0 && y >= 1 && (Src(x, y) == 0 || Src(x, y - 1) == 0))//第一
    行
5.          {
6.              result(x, y) = 0;
7.          }
8.          if (x >= 1 && y >= 1 && (Src(x, y) == 0 || Src(x - 1, y) == 0 || Src
    (x, y - 1) == 0))
9.          {
10.             result(x, y) = 0;
11.         }
12.         if (y == 0 && x >= 1 && (Src(x, y) == 0 || Src(x - 1, y) == 0))//第一
    列
13.         {
14.             result(x, y) = 0;
```

```
        }
    }
    result.display();
    result.save(save_path.c_str());
}
```

膨胀后的结果与膨胀前进行对比：

膨胀前：

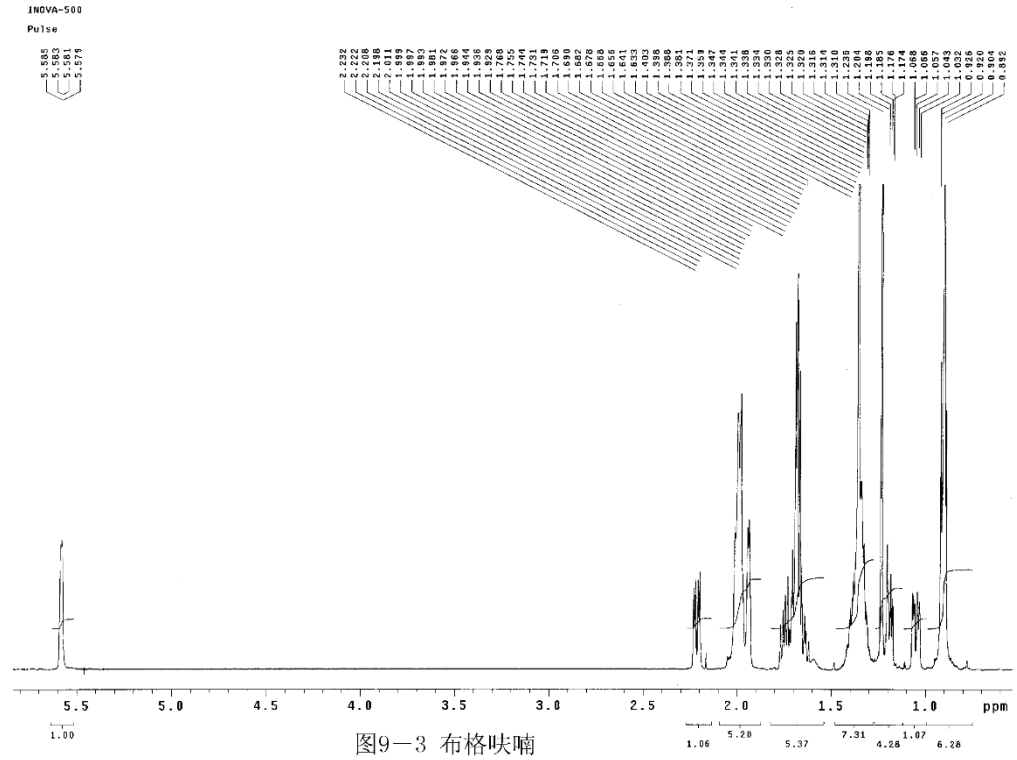


图9—3 布格呋喃

膨胀后：

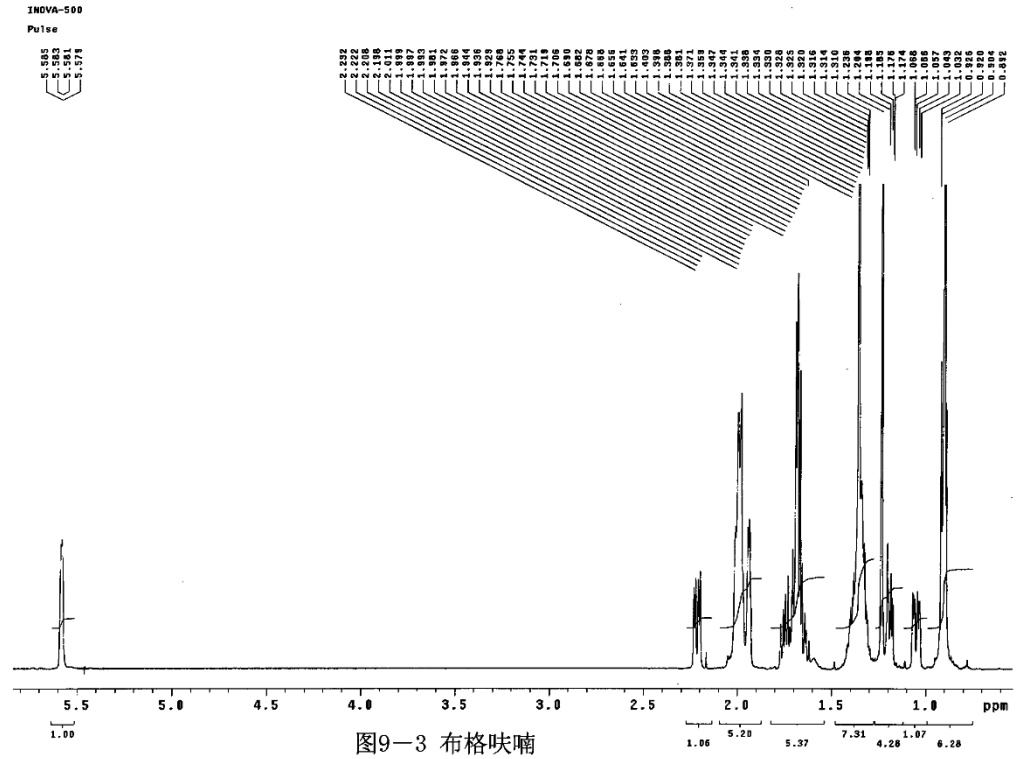


图9—3 布格呋喃

可以看出膨胀后的图片黑色部分明显比膨胀前粗。

2. 然后是求连通块并筛选

获得连通块采用 Seed-Filling 算法，具体步骤如下：

1) 扫描图像，直到当前像素点 B(x,y) == 0；
2) 将 B(x,y)作为种子（像素位置），并赋予其一个 label，然后将该种子相邻的所有前景像素都压入栈中；
3) 弹出栈顶像素，赋予其相同的 label，然后再将与该栈顶像素相邻的所有前景像素都压入栈中；
4) 重复 b 步骤，直到栈为空；

此时，便找到了图像 B 中的一个连通区域，该区域内的像素值被标记为 label；

5）重复第 1）- 4）步，直到扫描结束；

这里标记的方法比较麻烦，因为要判断一个点是否被标记要遍历已标记的所有点，我的方法是每次选出一个点就把原图中该点位置灰度改为 255，这样可以达到和标记一样的效果。另外由于连通块拐角位置可能只有一个像素相邻而且如果不是直角的话两个像素可能是对角的关系，所以我采用的是八邻域相邻。这里采用了数据结构 Area_point，声明如下：

```
struct  Area_point
{
    int x;
    int y;
    int label;
    Area_point(int x_, int y_, int label_)
    {
        x = x_;
        y = y_;
        label = label_;
    }
};
```

另外连通块用 Area_point 类型的 vector 存储。具体的代码如下：

获得连通块：

```
void Link_area::get_Link_Area()
{
    cimg_forXY(Src, x, y) {
        if (Src(x, y) == 0)//找到下一个种子点
        {
            temp.clear();
            label++;
            Area_point base(x, y, label);//种子点
            Src(x, y) = 255;//把种子点像素改为背景
            temp.push_back(base);
            queue<Area_point> p;
            p.push(base);
```

```
            while (!p.empty())
            {
                if (p.front().x >= 1 && p.front().y >= 1 && p.front().x < Src.width() - 1 && p.front().y < Src.height() - 1)
                {
                    for (int i = -1; i < 2; i++)//遍历八个邻域
                    {
                        for (int j = -1; j < 2; j++)
                        {
                            if (Src(p.front().x + i, p.front().y + j) == 0)//如果连通就加入队列
                            {
                                p.push(Area_point(p.front().x + i, p.front().y + j, label));
                                Src(p.front().x + i, p.front().y + j) = 255;

                                temp.push_back(Area_point(p.front().x + i, p.front().y + j, label));
                            }
                        }
                    }//八个点遍历完
                }
                p.pop();
            }
            link_area.push_back(temp);//获得一个连通块
        }
    }
    //cout << link_area.size();
}
```

然后是筛选出黑色像素小于 T 的连通块，为了防止噪声点的影响，这里选的其实是大于 T_min 小于 T_max 的连通块：

```
void Link_area::filtrate()
{
    vector<vector<Area_point>>temp;
    for (int i = 0; i < link_area.size(); i++)
    {
        if (link_area[i].size() <= T_max && link_area[i].size()>T_min)
            temp.push_back(link_area[i]);
    }
    cout << temp.size() << endl;
    link_area.clear();
```

```
    link_area = temp;
}
```

3. 在原图上把联通块(黑色像素<=T)用红色框标记

   在上面的步骤筛选出的其实已经是字母和数字了，这里只需要对筛选出来的连通块进行绘图即可，做法是找到能包围连通块的最小矩形，然后调用 CImg 的函数 draw_rectangle 即可。这里有辅助函数 get_diagonal，作用是找到能包围连通块的矩阵左上角和右下角点的坐标供后面的调用。做法是遍历连通块，找到 x 和 y 坐标的最大最小值，代码如下：

```
vector<Area_point> get_diagonal(vector<Area_point>area)
{
    int x_min = area[0].x;
    int x_max = 0;
    int y_min = area[0].y;
    int y_max = 0;
    for (int i = 0; i < area.size(); i++)
    {
        if (area[i].x > x_max)
            x_max = area[i].x;
        if (area[i].x < x_min)
            x_min = area[i].x;
        if (area[i].y > y_max)
            y_max = area[i].y;
        if (area[i].y < y_min)
            y_min = area[i].y;
    }
    vector<Area_point>temp;
    temp.push_back(Area_point(x_min, y_min, 0));
    temp.push_back(Area_point(x_max, y_max, 0));
    return temp;
}
```

找到坐标后就能调用函数进行绘制，代码如下：

```
void Link_area::Draw_square()
```

```
{
    unsigned char red[3] = { 255, 0, 0 };
    for (int i = 0; i < link_area.size(); i++)
    {
        vector<Area_point>temp = get_diagonal(link_area[i]);
        result.draw_rectangle(temp[0].x - 3, temp[0].y - 3, temp[1].x + 3, temp[1].y + 3, red,1, ~0U);
    }
    result.display();
}
```

绘制的结果如下：

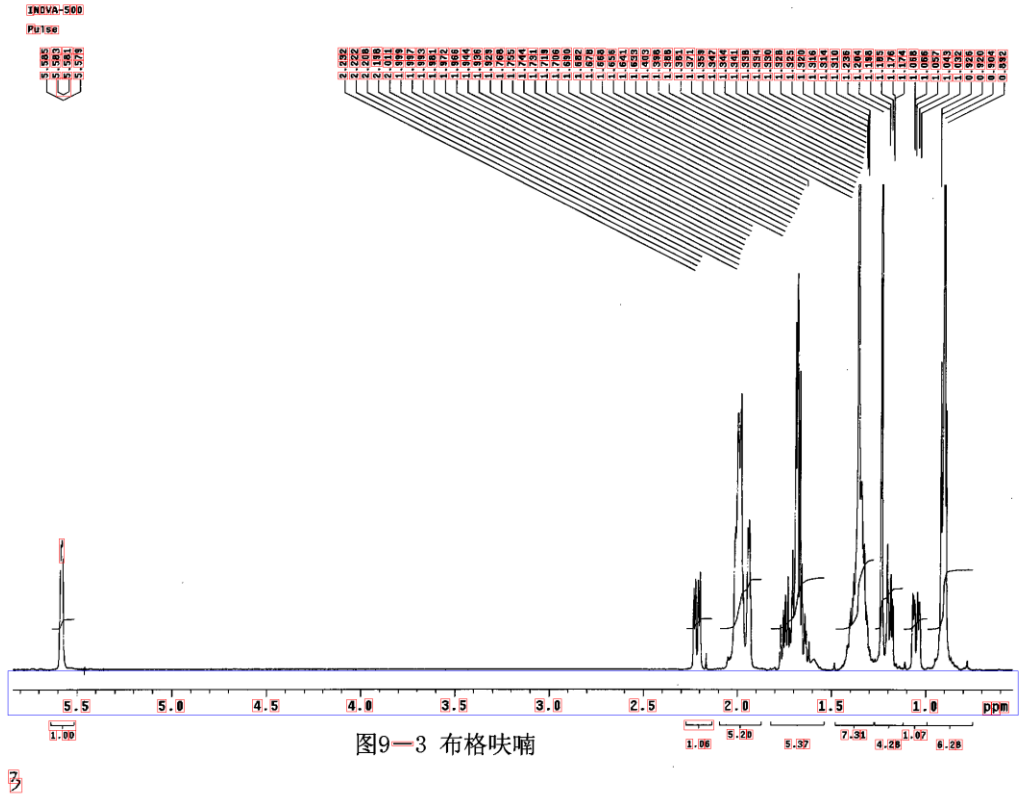


图9—3 布格呋喃

任务三： 把图像中标尺 OCR，基本思路:

- 计算标尺对应的位置和区域(即上图的蓝色框区域).
- 识别标尺图像中的刻度数字，可以调用 OpenCV.

找标尺的位置思路是用上次的霍夫变换找到霍夫空间中投票数最多的直线（即最长的直线）。但是我采用的是比较取巧的方法，对上面得到的连通块进行适当的筛选，如果阈值选的合理可以直接找到标尺的位置，不过这个方法

没有普适性。用这样的方法得到的标尺如下：

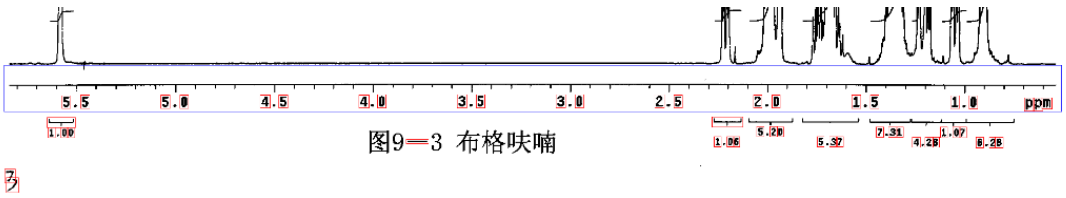


图9—3 布格呋喃

为了识别数字的方便，把标尺单独提出来并进行修剪，得到只有数字的部分：

代码如下：

```
1.  void Link_area::get_Scaleplate(string save_path)
2.  {
3.      unsigned char blue[3] = { 0, 0, 255 };
4.      vector<Area_point>scaleplate = get_diagonal(Scaleplate[1]);
5.      result.draw_rectangle(scaleplate[0].x - 10, scaleplate[0].y - 35, scalep
    late[1].x + 10, scaleplate[1].y + 35, blue, 1, ~0U);
6.      result.display();
7.      result.save(this->save_path.c_str());
8.      scal = CImg<unsigned char>(scaleplate[1].x - scaleplate[0].x - 100 , sca
    leplate[1].y - scaleplate[0].y + 10, 1, 1);
9.      cimg_forXY(scal, x, y)
10.     {
11.         scal(x, y) = result(scaleplate[0].x - 5 + x, scaleplate[0].y + y + 2
    0);
12.     }
13.     scal.display();
14.     int wid = scal._width;
15.     int hei = scal._height;
16.     scal.save(save_path.c_str());
17. }
```

然后就是对这个图片进行数字识别，我是调用 openCV 库做的，因为图片处理比较方便，这里配置 openCV 库真的花了很久时间。具体识别思路如下：

1. 首先把图片二值化得到二值图
2. 然后对二值图进行分割，得到一个一个数字，方法如下：
    1) 先左右切，对每列进行遍历，直到列的像素和不为 0，记下此时的位

置 left，然后遍历，当列像素和重新为 0 时记下位置 right，这样即可把数字左右分割开。

2) 然后上下切，方法与左右切一样。

3. 这样得到能包含数字的最小矩阵形成的图片，然后将这些图片与模板的 0-9 进行比对，这里先去掉了像素过少的图片（小数点）。比对的方法是先把模板图用上面的方法进行切割，然后把模板和数字 resize 成同样大小，然后用对应像素点值相减，然后求返回图片的整个图片的像素点值得平方和，和哪个模板匹配时候返回图片的平方和最小则就识别为哪个数字。主要代码如下：

左右切割：

```
int cutLeft(Mat& src, Mat& leftImg, Mat& rightImg)//左右切割
{
    int left, right;
    left = 0;
    right = src.cols;

    int i;
    for (i = 0; i < src.cols; i++)
    {
        int colValue = getColSum(src, i);//统计所有列像素的总和
        if (colValue > 0)//扫描直到列像素的总和大于 0 时，记下当前位置 left
        {
            left = i;
            break;
        }
    }
    if (left == 0)
    {
        return 1;
    }

    //继续扫描
    for (; i < src.cols; i++)
    {
```

```
        int colValue = getColSum(src, i);
        if (colValue == 0)//继续扫描直到列像素的总和重新等于 0 时，记下当前位置 right
        {
            right = i;
            break;
        }
    }
    int width = right - left;//分割图片的宽度则为 right - left
    Rect rect(left, 0, width, src.rows);//构造一个矩形，参数分别为矩形左边顶部的 X 坐标、Y 坐标，右边底部的 X 坐标、Y 坐标（左上角坐标为 0，0）
    leftImg = src(rect).clone();
    Rect rectRight(right, 0, src.cols - right, src.rows);//分割后剩下的原图
    rightImg = src(rectRight).clone();
    cutTop(leftImg, leftImg);//上下切割
    return 0;
}
```

处理模板函数：

```
void get_Template()
{
    Mat src;
    char name[50];
    for (int i = 0; i < 10; i++)
    {
        Mat temp;
        sprintf_s(name, "C:\\Users\\HP\\Desktop\\template\\%d.png", i);
        src = imread(name);
        cvtColor(src, temp, COLOR_BGR2GRAY);
        threshold(temp, temp, 100, 255, 1);
        //namedWindow("Image");
        //imshow("Image", temp);//显示分割后的图片
        //waitKey(0);
        //destroyWindow("Image");

        Mat leftImg, rightImg;
        int res = cutLeft(temp, leftImg, rightImg);
        imwrite(name, leftImg);
    }
}
```

（注意：这个函数只在没有处理过模板时调用，之后得到处理过的模板

不需要再次调用）

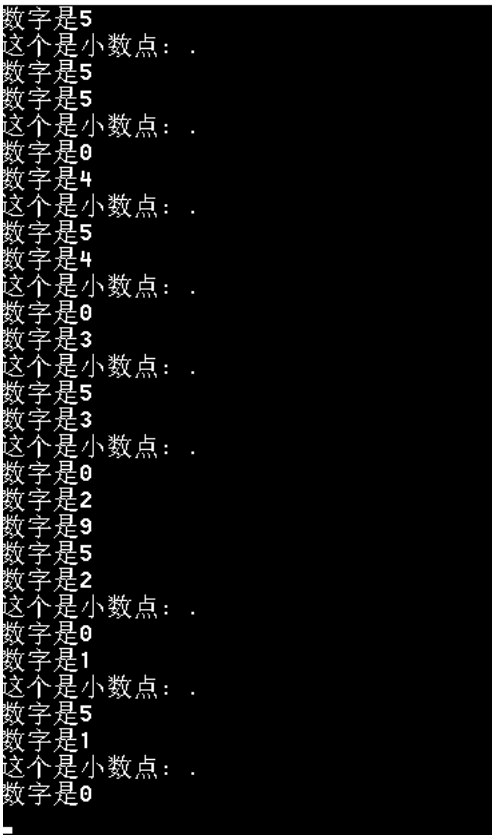
数字识别函数：

```
int  getSubtract(Mat &src, int TemplateNum) //数字识别
{
    Mat img_result;
    int min = 1000000;
    int serieNum = 0;
    if (src.cols < 7)
    {
        printf("这个是小数点: .\n");
        return 0;
    }

    for (int i = 0; i < TemplateNum; i++) {
        char name[50];
        sprintf_s(name, "C:\\Users\\HP\\Desktop\\template\\%d.png", i);
        Mat Template = imread(name, CV_LOAD_IMAGE_GRAYSCALE);//读取模板
        threshold(Template, Template, 100, 255, CV_THRESH_BINARY);
        threshold(src, src, 100, 255, CV_THRESH_BINARY);
        resize(src, src, Size(32, 48), 0, 0, CV_INTER_LINEAR);
        resize(Template, Template, Size(32, 48), 0, 0, CV_INTER_LINEAR);//调整尺寸
        //imshow(name, Template);

        /*让需要匹配的图分别和 10 个模板对应像素点值相减，然后求返回图片的整个图片的像素点值得平方和，和哪个模板匹配时候返回图片的平方和最小则就可以得到结果*/
        absdiff(Template, src, img_result);//AbsDiff，OpenCV 中计算两个数组差的绝对值的函数。
        int diff = 0;
        getPXSum(img_result, diff);//获取所有像素点和
        if (diff < min)//像素点对比
        {
            min = diff;
            serieNum = i;
        }
    }
    printf("数字是%d\n", serieNum);
    return serieNum;
}
```

识别的效果如下：



标尺如下：

可见都识别正确。

测试的代码如下：

```
1.   #include"otsu.h"
2.  #include"CImg.h"
3.  #include"Delate.h"
4.  #include"Link_area.h"
5.  #include"number_recognition.h"
6.  #include<iostream>
7.  using namespace std;
8.  int main() {
9.      CImg<unsigned char>img;
10.     img.load_bmp("C:/Users/HP/Desktop/H-Image.bmp");
11.     otsu test(img, "C:/Users/HP/Desktop/otsu.bmp");
12.     test.seg();
13.     img.load_bmp("C:/Users/HP/Desktop/otsu.bmp");
14.     Delate test1(img,"C:/Users/HP/Desktop/delate.bmp");
```

```
    test1.Do_delate();
    img.load_bmp("C:/Users/HP/Desktop/delate.bmp");
    Link_area test2(img, "C:/Users/HP/Desktop/link_area.bmp");
    test2.get_Link_Area();
    test2.filtrate();
    test2.Draw_square();
    test2.get_Scaleplate("C:/Users/HP/Desktop/Scaleplate.bmp");
    Number_Detect test3 = Number_Detect("C:/Users/HP/Desktop/Scaleplate.bmp"
, "C:/Users/HP/Desktop/result.bmp");
    test3.get_Number();
    return 0;
}
```

测试时应注意修改文件位置