浙江大学
ZHEJIANG UNIVERSITY

数字图像处理与机器视觉 实验报告

作业名称 HW5 形态学操作

姓　　名 杨逍宇

学　　号 3220105453

电子邮箱 3220105453@zju.edu.cn

联系电话 13518290755

导　　师 蔡声泽/曹雨齐/姜伟

2025 年 5 月 4 日

1 已实现的功能简述及运行简要说明

1.1 已实现的功能简述：

(1). `blur_restoration.cpp` 实现对一幅二值图像，比较不同结构元素下腐蚀、膨胀、开运算和闭运算的结果

(2). 尝试细化/骨架提取或孔洞填充等操作

(3). 生成结果图像并保存在 `assets` 文件夹中

项目目录树如下：

```
/
├── assets
│   ├── morphology_result.png  基础形态学操作结果
│   ├── extended_result.png  高级形态学操作结果
│   └── src.png  操作对象
├── src
│   └── morphology.cpp  形态学操作代码
├── build
│   └── morphology.exe  对应的可执行文件
├── docs
│   └── 实验报告 HW5.pdf  HW5 实验报告
├── CMakeLists.txt
└── README.md
```

1.2 运行说明：

- 输入图像路径：例如`../assets/src.png`
- 运行可执行文件，屏幕会输出不同形态学操作下得到的结果
- 可通过修改文件中的参数，例如 kernel_size 来得到不同的效果

2 开发与运行环境

本实验使用的软件和工具如下：

- 开发环境：Visual Studio Code on Ubuntu22.04
- 编程语言：C++
- 库：OpenCV 4.7.0
- 构建工具：CMake

3 算法基本思路

3.1 基础形态学操作

- **腐蚀**：用结构元素扫描图像，取邻域内最小值

$$dst(x,y) = \min_{(i,j)\in kernel} src(x+i, y+j)$$

- **膨胀**：用结构元素扫描图像，取邻域内最大值

$$dst(x,y) = \max_{(i,j)\in kernel} src(x+i, y+j)$$

- **开运算**：先腐蚀后膨胀，消除小物体

$$open(src) = dilate(erode(src))$$

- **闭运算**：先膨胀后腐蚀，填充小孔洞

$$close(src) = erode(dilate(src))$$

3.2 高级形态学操作

- **细化算法**：迭代删除边界像素直至单像素宽度,Zhang-Suen 算法

$$\begin{cases} \text{邻域前景数} \in [2,6] \\ 0\to1 \text{ 跳变次数} = 1 \\ \text{方向约束条件} \end{cases}$$

- **孔洞填充**：基于形态学重建的改进算法

$$result = src \cup R_{src^c}(marker)$$

- **骨架提取**：基于迭代边界点删除策略的经典骨架化方法

$$\begin{cases} 2 \leq N(p_1) \leq 6 & \text{邻域前景数约束 } A(p_1) = 1 \\ \text{邻域跳变次数约束 } p_2p_4p_6 = 0 \text{ 且} p_4p_6p_8 = 0 & \text{方向约束条件} \end{cases}$$

4 算法实现要点

4.1 形态学操作核心代码

```
// 结构元素生成
vector<Mat> kernels = {
    getStructuringElement(MORPH_RECT, Size(5,5)),
    getStructuringElement(MORPH_ELLIPSE, Size(5,5)),
    getStructuringElement(MORPH_CROSS, Size(5,5))
};

// 基础操作实现
Mat applyOperation(InputArray src, int op_type,
                   InputArray kernel) {
    Mat result;
    if(op_type == MORPH_ERODE) {
        erode(src, result, kernel);
    } else if(op_type == MORPH_DILATE) {
        dilate(src, result, kernel);
    } else {
        morphologyEx(src, result, op_type, kernel);
    }
    return result;
}
```

4.2 孔洞填充改进算法

```
cv::Mat morphReconstructFill(const cv::Mat& src) {
    CV_Assert(src.type() == CV_8UC1);
```

```
    cv::Mat inverted;
    cv::bitwise_not(src, inverted);
    cv::Mat marker = cv::Mat::zeros(src.size(), CV_8UC1);
    cv::rectangle(marker, cv::Rect(1,1,src.cols-2,src.rows-2), 255, cv::
        FILLED);
    cv::Mat prev;
    do {
        prev = marker.clone();
        cv::dilate(marker, marker, cv::Mat()); // 膨胀操作
        cv::bitwise_and(marker, inverted, marker); // 约束于原图的反转
    } while (cv::countNonZero(prev != marker) > 0);
    cv::Mat result;
    cv::bitwise_or(src, marker, result);
    return result;
}

cv::Mat contourFill(cv::Mat src) {
    std::vector<std::vector<cv::Point>> contours;
    cv::findContours(src, contours, cv::RETR_CCOMP, cv::
        CHAIN_APPROX_SIMPLE);
    for (int i=0; i<contours.size(); ++i) {
        if (cv::contourArea(contours[i]) < 1000) // 过滤小轮廓
            cv::drawContours(src, contours, i, 255, cv::FILLED);
    }
    return src;
}

cv::Mat connectivityFill(cv::Mat src) {
    cv::Mat labels, stats, centroids;
    int n = cv::connectedComponentsWithStats(src, labels, stats, centroids
        );
    for(int i=1; i<n; ++i) {
        if(stats.at<int>(i, cv::CC_STAT_AREA) < 100) // 小区域视为孔洞
            cv::rectangle(src, cv::Rect(stats.at<int>(i, cv::CC_STAT_LEFT),
                                        stats.at<int>(i, cv::CC_STAT_TOP),
                                        stats.at<int>(i, cv::CC_STAT_WIDTH),
                                        stats.at<int>(i, cv::CC_STAT_HEIGHT)),
                                           255, -1);
```

```
    }
    return src;
}
```

4.3 细化算法实现

```
cv::Mat ThinImage(const cv::Mat& src) {
    cv::Mat dst;
    cv::ximgproc::thinning(src, dst, cv::ximgproc::THINNING_ZHANGSUEN);
    return dst;
}
```

4.4 骨架提取算法实现

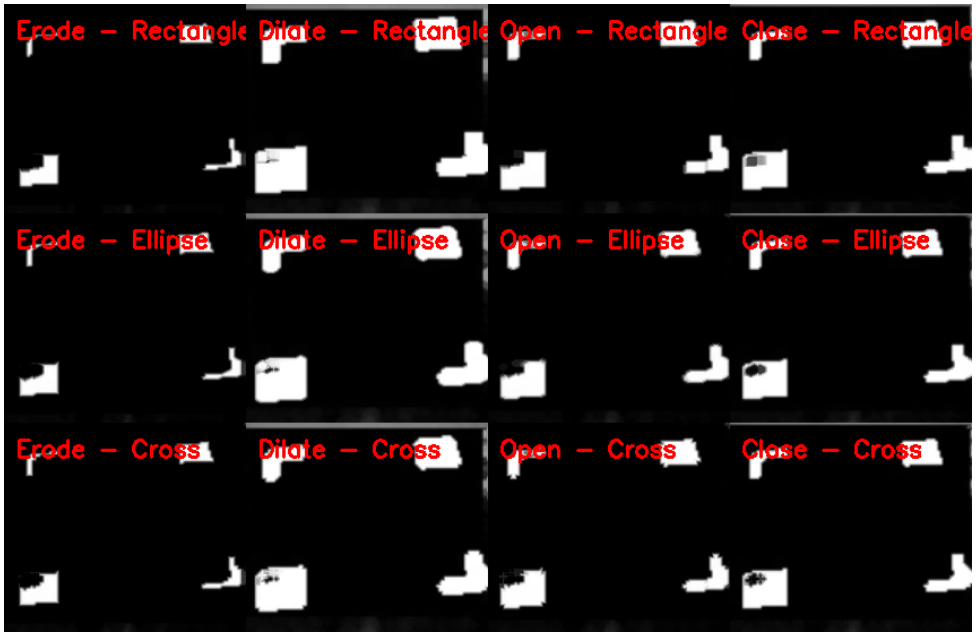
```
cv::Mat Skeletonize(const cv::Mat& src) {
    cv::Mat skel(src.size(), CV_8UC1, cv::Scalar(0));
    cv::Mat temp, eroded;
    cv::Mat element = cv::getStructuringElement(cv::MORPH_CROSS, cv::Size
        (3, 3));

    bool done;
    do {
        cv::erode(src, eroded, element);
        cv::dilate(eroded, temp, element);
        cv::subtract(src, temp, temp);
        cv::bitwise_or(skel, temp, skel);
        eroded.copyTo(src);
        done = (cv::countNonZero(src) == 0);
    } while (!done);

    return skel;
}
```

5 实验结果及分析

原始测试图像：

5.1 不同结构元素影响

对测试图像使用 3 种结构元素（矩形、椭圆、十字）进行处理，得到结果如下：



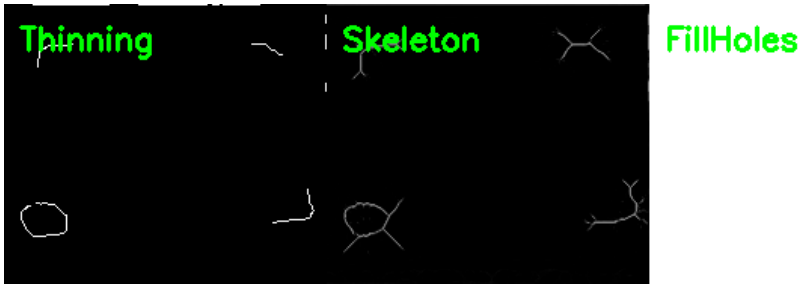
上述结果图像排列成一个矩阵，从左到右依次为 `Erode`,`Dilate`,`Open`,`Close`, 然后分别使用不同形态的结构元素，从上到下依次排列为 `Rectangle`,`Ellipse`,`Cross`。

从上述实验发现：

- 矩形核处理结果保留直角特征
- 椭圆核处理边缘更光滑
- 十字核对斜向特征敏感

5.2 高级操作效果

对测试图像使用细化，骨架提取，孔洞填充处理，得到结果如下：



参数影响分析：

- 细化迭代次数：过多导致断裂，不足留有冗余
- 结构元素大小：过大破坏细节，过小处理不彻底
- 孔洞填充，我尝试了三种方法，但得到的结果均为全黑或全白，认为可能是因为左下角小正方形的边缘并不完全连接，因此不能将其中心填满。

6 结论与心得体会

通过本实验，我们得出以下结论：

- 结构元素形状对处理效果有决定性影响
- 开闭运算组合可有效处理复杂形态特征
- 改进的孔洞填充算法边界处理更稳定

实验改进方向：

- 实现自适应结构元素选择算法
- 结合连通域分析优化孔洞判断
- 开发交互式参数调节界面
- 对实验进行预处理后再进行孔洞填充等操作