浙江大学实验报告

课程名称：图像信息处理 指导老师：宋明黎 成绩：

实验名称：BMP 图像的二值化操作与形态学基本变换

一、实验目的和要求

1. 了解二值图像 (binary image)，及二值图像的优缺点
2. 掌握由灰度图向二值图像转换 (binarization) 的大津算法，并在图片上实现
3. 了解形态学 (morphology) 的基本知识，并掌握膨胀 (dilation)、腐蚀 (erosion)、开操作 (open) 和闭操作 (close)

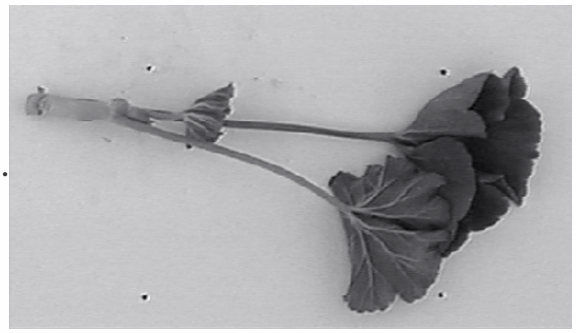
二、实验内容和原理

二值图像

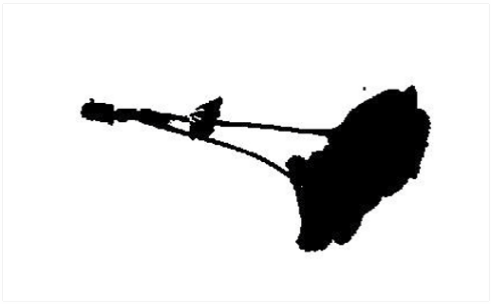
二值图像 (binary image) 中的像素值只有 0 和 1 (255)，在编程当中我们经常用$[0, 255]$来构建二值图像。

在实际应用中，二值图像的前景通常被置为1，后景被置为0.

优点	缺点
更小的内存需求	应用范围比较有限
更快的运行速度	表现力欠缺，不能表现物体的内部细节
为二值图像开发的算法往往可以用于灰度图像	无法控制对比度



灰度图像



二值图像

图像二值化

- 二值图像的构建，通常通过对灰度图像进行阈值化操作，将图像的像素进行重置。我们设置一个阈值 (Threshold)，将比阈值小的设为0，比阈值大的设为255。

$$\begin{cases} I(x,y) = 0 \quad if\ I(x,y) < Threshold \\ I(x,y) = 255 \quad if\ I(x,y) \geq Threshold \end{cases} \tag{1}$$

- 如何查找一个合适的阈值?
- **基本思想**：将二值化得到的二值图像视为两部分，一部分对应前景（Foreground），另一部分对应背景（Background）。尝试找到一个合适的 threshold 使得到的前景和背景的内部方差最小，而它们之间的方差则最大。

OTSU大津算法

基本推导过程：

$$\begin{aligned} \sigma^2_{within}(T) &= \frac{N_{Fgrd}(T)}{N}\sigma^2_{Fgrd}(T) + \frac{N_{Bgrd}(T)}{N}\sigma^2_{Bgrd}(T) \\ \sigma^2_{between}(T) &= \sigma^2 - \sigma^2_{within}(T) \\ &= (\frac{1}{N}\sum_{x,y}(f^2[x,y] - \mu^2)) - (\frac{N_{Fgrd}}{N}\sum_{x,y\in Fgrd}(f^2[x,y] - \mu^2_{Fgrd}) - \\ &\frac{N_{Bgrd}}{N}(\frac{1}{N_{Bgrd}}\sum_{x,y\in Bgrd}(f^2[x,y] - \mu^2_{Bgrd}) \\ &= \frac{N_{Fgrd}}{N}(\mu_{Fgrd} - \mu)^2 + \frac{N_{Bgrd}}{N}(\mu_{Bgrd} - \mu)^2 \\ &= \frac{N_{Fgrd}(T) \times N_{Bgrd}(T)}{N^2}(\mu_{Fgrd}(T) - \mu_{Bgrd}(T))^2 \end{aligned} \tag{2}$$

- 其中，N是像素总个数，N_{Fgrd}是给定T的情况下属于前景的像素个数。

大津算法的具体流程

1. 确定原始图像中像素的最大值和最小值；
2. 最小值加1作为 threshold 对原始图像进行二值化操作；
3. 根据对应关系确定前景和背景，分别计算当前 threshold 下的内部协方差和外部协方差；

4. 回到 **步骤2** 直到达到像素最大值；
5. 找到最大外部和最小内部协方差对应的 threshold.

形态学操作

数学形态学 (Mathematical morphology)，基础理论是集合论，一种简单的非线性代数算子，主要用于二值图像，可扩展到灰度图像，常用操作有噪声过滤、形状简化、细化、分割、物体描述等。

形态学的基本思想是用具有一定形态的结构元素去度量和提取图像中的对应形状以达到对图像分析和识别的目的。

形态学在图像处理中的基本运算有：**膨胀、腐蚀、开操作、闭操作**

膨胀

膨胀是将与物体**“接触”**的所有背景点合并到该物体中，使边界向外部扩张的过程。可以用来填补物体中的空洞。（其中**“接触”**的含义由结构元描述）

由B对A膨胀所产生的二值图象D是满足以下条件的点(x,y)的集合：如果B的原点平移到点(x,y)，那么它与A的交集非空。

$$A \bigoplus B = \{z|(B)_z \cap A \neq \emptyset\} \tag{3}$$

腐蚀

腐蚀是一种消除边界点，使边界向内部收缩的过程。可以用来消除小且无意义的物体。

由B对A膨胀所产生的二值图象D是满足以下条件的点(x,y)的集合：如果B的原点平移到点(x,y)，那么它与A的交集非空。

$$A \ominus B = \{(x,y)|(B)_{xy} \subseteq A\} \tag{4}$$

开运算

先腐蚀，后膨胀。

用来消除小物体、在纤细点处分离物体、平滑较大物体的边界的同时并不明显改变其面积。

$$A \circ B = (A \ominus B) \bigoplus B \tag{5}$$

闭运算

先膨胀，后腐蚀。

用来填充物体内细小空洞、连接邻近物体、平滑其边界的同时并不明显改变其面积。

$$A \bullet B = (A \bigoplus B) \ominus B \tag{6}$$

三、实验步骤与分析

图像二值化

一、朴素实现：为全图进行一次大津算法，找到阈值并应用于全图。

```
//  Otsu's method for thresholding (normal version)
void naiveThreshold(const BITMAP &a, BITMAP &b) {
    int width = a.getWidth();
    int height = a.getHeight();

    int maxPixel = 0, minPixel = 255;
    double sigma = 0;
    int threshold;

    //  find the maximum and minimum pixel value of the image
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int pixel = a.getPixel(i, j);
            maxPixel = max(maxPixel, pixel);
            minPixel = min(minPixel, pixel);
        }
    }

    for (int pixel = minPixel + 1; pixel <= maxPixel; pixel++) {
        double NBgrd = 0.0, NFgrd = 0.0;
        double uFgrd = 0.0, uBgrd = 0.0;

        //  calculate sigma for an arbitrary threshold value
        for (int k = 0; k < height; k++) {
            for (int l = 0; l < width; l++) {
                int p = a.getPixel(k, l);

                if (p < pixel) {
                    NBgrd += 1.0;
                    uBgrd += p;
                } else {
                    NFgrd += 1.0;
```

```
                    uFgrd += p;
                }
            }
        }

        uBgrd /= NBgrd;
        uFgrd /= NFgrd;

        int N = height * width;

        //  update the threshold value
        double tmp = (NBgrd / N) * (NFgrd / N) * (uBgrd - uFgrd) * (uBgrd - uFgrd);
        if (tmp > sigma) {
            sigma = tmp;
            threshold = pixel;
        }
    }

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            b.setPixel(i, j, (a.getPixel(i, j) < threshold) ? 0 : 255);
        }
    }
}
```

算法流程：

1. 遍历所有像素$[height, width]$，获取每一像素值并更新maxPixel minPixel
2. 从minPixel到maxPixel，寻找最佳阈值（计算内部协方差和外部协方差，并对σ进行更新）
3. 将阈值应用于所有像素，即小于阈值的像素值置为0，大于阈值的像素值置为255。

分析：

我们注意到，全局的大津算法有较高的性价比，能在最短的时间内查找得到全局阈值，并且对图像进行二值化。但是，这样的方式也存在问题，例如它无法处理阴影部分的信息，通常将亮度偏低（高）的有效信息全部置为0 (255)，从而导致大片的空白，整体的图片信息不够细腻。

因此，我们可以尝试移动窗格算法，来解决高频信息缺失和阴影的问题。

二、移动窗格实现：以每个像素为中心建立一个窗格，在窗格内进行大津算法，并将计算得到的阈值应用于该像素。

```
//  Otsu's method for thresholding (window version)
void otsuThreshold(const BITMAP &a, BITMAP &b, const int windowSize) {
    //  get the width and height of the image
    int width = a.getWidth();
    int height = a.getHeight();
```

```
    //  iterate through every pixel of the image
    for (int i = 0; i < height; i += 1) {
        for (int j = 0; j < width; j += 1) {
            int maxPixel = 0, minPixel = 255;
            double sigma = 0;
            int threshold;

            //  set the valid boundary of the window
            int r1 = (i - windowSize / 2 < 0) ? 0 : i - windowSize / 2;
            int r2 = (i + windowSize / 2 > height) ? height : i + windowSize / 2;
            int c1 = (j - windowSize / 2 < 0) ? 0 : j - windowSize / 2;
            int c2 = (j + windowSize / 2 > width) ? width : j + windowSize / 2;

            //  find the maximum and minimum pixel value in the window
            for (int k = r1; k < r2; k++) {
                for (int l = c1; l < c2; l++) {
                    int pixel = a.getPixel(k, l);
                    maxPixel = max(maxPixel, pixel);
                    minPixel = min(minPixel, pixel);
                }
            }

            //  iterate through every possible threshold value
            for (int pixel = minPixel + 1; pixel <= maxPixel; pixel++) {
                double NBgrd = 0.0, NFgrd = 0.0;
                double uFgrd = 0.0, uBgrd = 0.0;

                //  calculate sigma for an arbitrary threshold value
                for (int k = r1; k < r2; k++) {
                    for (int l = c1; l < c2; l++) {
                        int p = a.getPixel(k, l);

                        if (p < pixel) {
                            NBgrd += 1.0;
                            uBgrd += p;
                        } else {
                            NFgrd += 1.0;
                            uFgrd += p;
                        }
                    }
                }

                uBgrd /= NBgrd;
                uFgrd /= NFgrd;

                int N = (r2 - r1) * (c2 - c1);

                //  update the threshold value
                double tmp = (NBgrd / N) * (NFgrd / N) * (uBgrd - uFgrd) * (uBgrd -
uFgrd);
                if (tmp > sigma) {
                    sigma = tmp;
                    threshold = pixel;
```

```
                }
            }

            //  set the pixel value of the new image
            b.setPixel(i, j, (a.getPixel(i, j) < threshold) ? 0 : 255);

        }
    }
}
```

算法流程：

1. 遍历每个像素$[height, width]$，以每个像素为中心，构建窗格边界（注意处理越界的问题）。
2. 在局部窗格内，找到其中的最大像素和最小像素值
3. 对可能像素值进行遍历，计算得到该阈值下的σ，并更新得到最佳阈值
4. 对当前像素应用该阈值并进行更新

分析：

我们发现，移动窗格版本的大津算法保留了画面中更多的细节，尤其是高频信息和阴影部分的有效信息。同时，该算法比简单的分块法拥有更好的平滑度，画面整体的过渡更加自然。

当然，该算法也存在一个十分致命的问题——时间复杂度过高。由于我们没有对这个算法进行优化，在计算阈值的过程中事实上存在大量的重复计算。

膨胀

```
void BITMAP::dilation() {
    // set the kernel
    BYTE kernel[9] = {255, 0, 255, 0, 0, 0, 255, 0, 255};

    BITMAP tmp;
    tmp = *this;

    //  iterate through every pixel
    for (int i = 0; i < getHeight(); i++) {
        for (int j = 0; j < getWidth(); j++) {

            BYTE pixelValue = 255;
            //  iterate through every pixel in the kernel
            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++) {
                    int x = i + k;
                    int y = j + l;

                    //  check if the pixel is out of boundary
                    if (x < 0 || x >= getHeight() || y < 0 || y >= getWidth()) {
                        continue;
                    }
```

```
                    if ((kernel[(k + 1) * 3 + l + 1] == 0) && (getPixel(x, y) == 0)) {
                        pixelValue = 0;
                    }
                }
            }

            tmp.setPixel(i, j, pixelValue);
        }
    }

    //  update the pixel value
    *this = tmp;
}
```

算法流程：

1. 构建核，在这里我们采用3×3的十字核
2. 遍历所有像素$[height, width]$，若该核有覆盖到像素，就将像素中心置1.
3. 更新像素。

腐蚀

```
void BITMAP::erosion() {
    // set the kernel
    BYTE kernel[9] = {255, 0, 255, 0, 0, 0, 255, 0, 255};

    BITMAP tmp;
    tmp = *this;

    //  iterate through every pixel
    for (int i = 0; i < getHeight(); i++) {
        for (int j = 0; j < getWidth(); j++) {

            BYTE pixelValue = 0;
            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++) {
                    int x = i + k;
                    int y = j + l;

                    //  check if the pixel is out of boundary
                    if (x < 0 || x >= getHeight() || y < 0 || y >= getWidth()) {
                        continue;
                    }

                    if (kernel[(k + 1) * 3 + l + 1] == 0 && getPixel(x, y) != 0) {
                        pixelValue = 255;
                    }
                }
```

```
            }

            tmp.setPixel(i, j, pixelValue);
        }
    }

    //  update the pixel value
    *this = tmp;
}
```

算法流程：

1. 构建核，在这里我们采用3×3的十字核
2. 遍历所有像素$[height, width]$，若该核的所有像素都被覆盖，就将核中心像素置1.
3. 更新像素。

开操作

```
void BITMAP::open() {
    erosion();
    dilation();
}
```

闭操作

```
void BITMAP::close() {
    dilation();
    erosion();
}
```

四、实验环境及运行方法

- 实验环境：
  - 系统：MacOS Ventura 13.2.1
  - 编译器：Apple Clang version 14.0.0
- 运行方法：
  - 由于后续实验中需要频繁用到 lab1 中对图像最基本的灰度化和二值化操作，因此在本次实验中，我首先用Cpp对BITMAP类进行了封装，源文件为bmp.cpp和bmp.hpp

- 在本次实验中，我们主要实现了`naiveThreshold()` `otsuThreshold()` `dilation()` `erosion()` `open()` `close()` 这六个函数，为以示区分，我将这些函数全部放置在`main.cpp`中。
- 本次实验我们提供了一份`Makefile`，可以直接在根目录下`make`后`./bmp`运行可执行文件。
- 运行后，我们将会生成以下图片
  - `binarization.bmp`，大津算法生成的图片（由于移动窗格法运行时间较长，在源代码中进行了注释）
  - `dilation.bmp`，膨胀操作
  - `erosion.bmp`，腐蚀操作
  - `open.bmp`，开操作
  - `close.bmp`，闭操作
- 为方便测试，我提供了几张测试图片，放置在根目录下的`pic`文件夹内，在接下来的实验结果展示中同样会给予一定说明。

五、实验结果展示

图像二值化操作——大津算法实现

1. 初始版本大津算法（`normalThreshold()`）

原始图片

二值化操作后的图像

2. 移动窗格版本大津算法（`otsuThreshold()`）

- 注意到，移动窗格的窗格大小可以进行更改，我们对不同的窗格大小进行了测试。

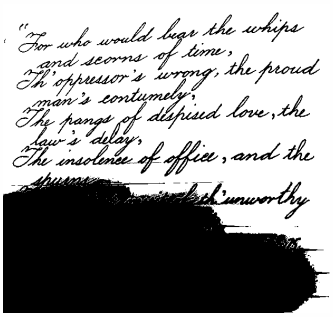
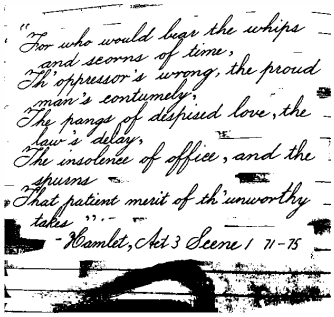
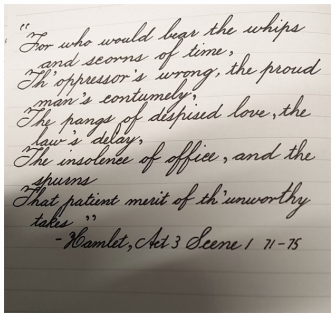
windowSize = 10

windowSize = 30

windowSize = 50

- 不难发现，windowSize的大小和图像的细腻程度成正相关。在保证窗格的局部性的条件下，如果窗格设置的太小，会导致图片中出现大量的噪声，反而不如最朴素的大津算法。当窗格设置得较大时，图片中的高频信息被很好地保留下来。

- 接下来，我们比较了两种算法对阴影区域的处理结果。

初始算法　　移动窗格算法　　原始图片

- 可以看到，移动窗格算法相比初始算法对阴影部分的信息做了很好的保留，但是，随之带来的也有一定的碎片噪声，对这一部分可以借助例如 `erosion()` `open()`等操作来消除。

图像形态学操作——膨胀、腐蚀、开、闭

1. 膨胀操作对字母缺失部分的补全


原始图片　　膨胀后图片

- 可以看到，图片中的缺口有较好的改善，但是，我们还需要选取更加恰当的核函数来获得更加好的效果。

2. 腐蚀操作对相连部分的消去


原始图片　　腐蚀后图片

- 可以明显看到腐蚀操作的效果。同样地，我们也应当选取更加好的核函数，或多进行几次操作来优化结果。
- 例如，当我们重复进行三次腐蚀操作后，消除的效果会更加优秀。

3. 开操作和闭操作对指纹图像的作用效果

原始图片

开操作

闭操作

- 注意到，由于“参照系”选择的不同（即我们关注的操作是针对高亮度区域还是低亮度区域），图像作用的结果可能会有不同。例如在这里，“闭操作”可以消除掉白色的噪声，而“开操作”可以放大噪声部分。同时，开闭操作可以在实现图像变换效果的同时尽可能保证图像的大小不变。

六、心得体会

在本次实验中，我熟悉了图像的二值化操作和形态学的基本操作，通过实际的分析和结果对比对不同操作的具体实现和优缺点有了更加全面的认识。

但是，这次实验中我也遇到了一定的问题。例如，本次实验我有大量时间花费在对 lab1 中图像基本操作的封装上，并且封装过程中未发现的 bug 暴露在了 lab2 中，而且花了大量的调试时间才发现（由于 C++ 中深浅拷贝的问题，只复制了数组指针而未对整个像素数组进行拷贝，导致图像的处理结果出现了异常）。另外，当我跨平台(Windows下) 测试时，发现图片处理出现了异常，在排查了很久之后才发现是没有指定文件的读写方式为`std::ios::binary`，而在MacOS平台下并未暴露这个问题。

因此，在未来我还需要更好地掌握对此类编程的调试方法，并学会验证算法本身的正确性。

期待未来的实验！