

浙江大学实验报告

专业：计算机科学与技术

姓名：沈一芑

学号：322010827

日期：2023/11/2

课程名称：____图像信息处理____指导老师：____宋明黎____成绩：____

实验名称：____图像二值化及形态学操作____

1. 实验目的和要求

1.1. 实验目的

- 依托实验一，将灰度图进行二值化操作
- 通过不同方法选取最优二值化阈值
- 了解图像的形态学操作
- 对图像进行腐蚀、膨胀、开闭运算的形态学操作

1.2. 实验要求

- 代码读取灰度图
- 使用 OSTU 法、滑动窗口法寻找阈值，进行图像二值化
- 对二值化图像进行腐蚀、膨胀、开闭运算等操作

2. 实验内容和原理

2.1. 实验原理

2.1.1. Construct a Binary Image 灰度图二值化

- Thresholding the grayscale image by reset the pixel value, 即通过确定阈值，将灰度图中小于阈值的点均设为 0（或 255），大于阈值的点均设为 255（0）。

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

- 有多种方法确定阈值。以下几种为例：

■ 均值法：

将整张灰度图的灰度值累加后除以像素点数，得到灰度均值作为阈值。

■ OSTU 法（全局）

将图片依照灰度分为前景与背景，并遍历 0-255 以寻找一个最佳阈值使得前景与背景的内部方差最小，两者间的方差最大。

$$\begin{aligned}w_f &= \frac{N_{Fgrd}}{N}, w_b = \frac{N_{Bgrd}}{N}, w_f + w_b = 1 \\ \mu &= w_f * \mu_{Fgrd} + w_b * \mu_{Bgrd} \\ \sigma_{between}^2 &= w_f(\mu_{Fgrd} - \mu)^2 + w_b(\mu_{Bgrd} - \mu)^2 \\ &= w_f(\mu_{Fgrd} - w_f * \mu_{Fgrd} - w_b * \mu_{Bgrd})^2 + w_b(\mu_{Bgrd} - w_f * \mu_{Fgrd} - w_b * \mu_{Bgrd})^2 \\ &\rightarrow w_b w_f (\mu_f - \mu_b)^2\end{aligned}$$

■ 滑动窗口法（局部）

由于 OSTU 法针对全局进行，当整张图片局部间亮度差异过大时计算出的阈值不能适用于所有亮度区域。故而设置一滑动窗口（sliding_window）在整张图片上进行步长为 stride 的滑动，对每一块区域单独使用 OSTU 计算阈值。

此处步长不一定与窗口大小相等。对于窗口在滑动过程中重复计算阈值的部分，有多种处理方法。其一是用靠后的滑动窗口阈值直接取缔先前的阈值；其二是将重合部分取均值处理。

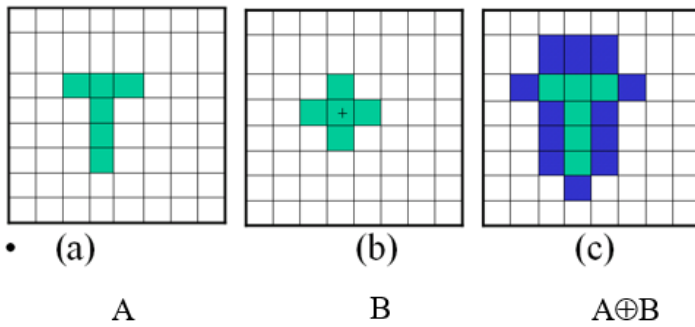
2.1.2. Morphology 形态学

- 用一定的形态结构元素去度量和提取图像中的对应形状，从而对图像进行分析和识别。其原理为集合论。通过形态学操作，我们能对图像进行化简、滤波、效果增强。
-

2.1.3. dilation 膨胀操作

- 选取一个结构元素并使其中心点遍历灰度图。灰度图与结构元素重合的部分只要存在灰度为 255 的像素点，就将结构元素中心点在灰度图中对应位置的灰度置为 255。即：扩大了判定范围，在整个结构元素范围内寻找符合要求的点来更新原点，从而实现膨胀的效果。
- 从数学角度，是将结构元素的每一像素点置为 1，分别与灰度图中对应像素点进行 & 运算，并将结果进行 | 运算：

$$A \oplus B = \{z | (B)_z \cap A \neq \emptyset\}$$

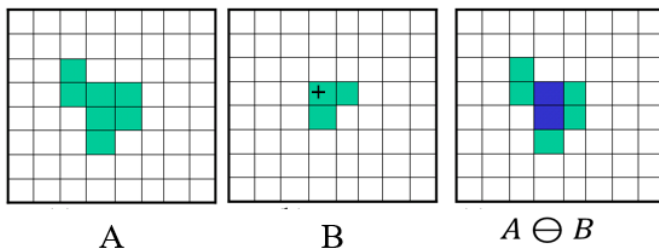


- 实际效果：扩张边界、填补空洞。

2.1.4. erosion 腐蚀操作

- 选取一个结构元素并使其中心点遍历灰度图。灰度图与结构元素重合的部分必须全部为 255 的像素点，否则将结构元素中心点在灰度图中对应位置的灰度置为 0。即：扩大了判定范围，在整个结构元素范围内遍历所有点来约束原点，从而实现腐蚀的效果。
- 从数学角度，是将结构元素的每一像素点置为 1，分别与灰度图中对应像素点进行 $\&$ 运算，并将结果进行 $\&$ 运算：

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

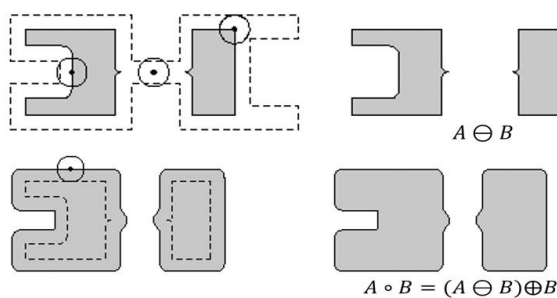


- 实际效果：消除边界、过滤噪点。

2.1.5. opening 开运算操作

- 先进行腐蚀操作，再进行膨胀操作。

$$A \circ B = (A \ominus B) \oplus B$$

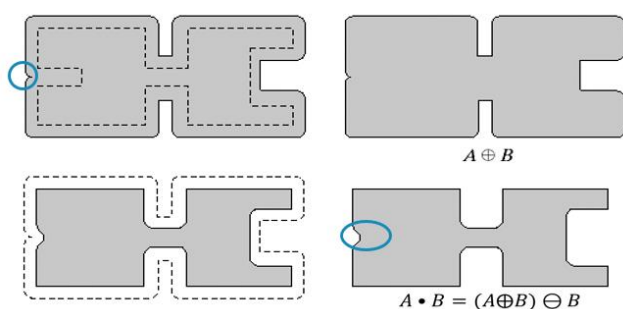


- 实际效果：在稀疏处消除毛刺，平滑边缘，但保留原有形状。

2.1.6. closing 闭运算操作

- 先进行膨胀操作，再进行腐蚀操作。

$$A \bullet B = (A \oplus B) \ominus B$$



- 实际效果：填补空洞、连接细小断点、在最大程度保留原有形状时平滑边缘。

2.2. 实验内容

- 使用 lab1 中方法，读入 BMP 图片，并将 RGB 色彩空间转为 YUV，进而生成灰度图。
- 使用 OSTU 算法/滑动窗口将灰度图二值化，生成 binary image。
- 对 binary image 进行膨胀（dilation）操作。
- 对 binary image 进行腐蚀（erosion）操作。
- 对 binary image 进行开运算（opening）操作。
- 对 binary image 进行闭运算（closing）操作。

3. 实验步骤与分析

3.1. 准备工作

- 定义结构体并声明变量如下；每个变量对应一张 BMP 图片：

```
typedef struct{
    BMFH bmfh;
    BMIH bmiH;
    RGBQUAD rgbquad[256];
    BYTE *data;
    int LineBytes, WidthBMP, HeightBMP, SizeImageBMP;
    //LineBytes is to compliment the width given that bytes mod 4 == 0
    //size of image part, instead of whole bmp
}IMAGE;
.
```

- IMAGE input, gray, binary, erosion, dilation, opening, closing;

- 由于 binary 需在 gray 的基础上生成, erosion、dilation、opening、closing 需要在 binary 的基础上生成, 故定义函数 copyBMP 将 base 指针的内容赋给 copy 指针, 实现图像的复制:

```

• void copyBMP(IMAGE *base, IMAGE *copy){
•
•     memcpy(&(copy->bmfh), &(base->bmfh), sizeof(BMFH));           //copy the
    bmp file header
•     memcpy(&(copy->bmiH), &(base->bmiH), sizeof(BMIH));           //copy the
    bmp information header
•     copy->bmiH.biBitCount=8;
•     copy->WidthBMP=base->WidthBMP;
•     copy->HeightBMP=base->HeightBMP;
•     copy->LineBytes=(copy->WidthBMP+3)/4*4;                         //bytes mod 4
    =0. compliment
•     copy->bmiH.biSizeImage=copy->LineBytes*copy->HeightBMP;       //calculate
    the size of the image data part
•     copy->SizeImageBMP=copy->bmiH.biSizeImage;
•     copy->bmfh.bfSize=copy->bmfh.bfOffBits+copy->bmiH.biSizeImage; //calculate
    the size of the whole bmp file
•     copy->data=new BYTE[copy->bmiH.biSizeImage];
•
•     memset(copy->data, 0, sizeof(BYTE)*copy->SizeImageBMP);
•     for(int i=0;i<256;i++){                                       //initialize
    the rgbquad
•         copy->rgbquad[i].rgbBlue=i;
•         copy->rgbquad[i].rgbGreen=i;
•         copy->rgbquad[i].rgbRed=i;
•     }
•     for(int i=0;i<copy->HeightBMP;i++){
•         for(int j=0;j<copy->WidthBMP;j++){
•             copy->data[copy->LineBytes*i+j]=base->data[base->LineBytes*i+j];
•         }
•     }
•
• }

```

- 形态学操作中定义两种基础结构元 structure_element_square 和 structure_element_cross, 分别对应 3*3 矩形与五格十字形结构元。
- 结构元以数组形式存储, 第零位存储该结构元包含的像素点数 n; 随后 n 位记录从左到右从上到下各个像素点到中心点的 x、y 坐标偏移。

- struct Coord{
- int x, y;
- };
- Coord struct_element_square[10]={9,9}, {-1,-1},{0,-1},{1,-1},{-1,0},{0,0},{1,0},{-1,1},{0,1},{1,1}};
- Coord struct_element_cross[6]={5,5}, {0,-1},{-1,0},{0,0},{1,0},{0,1}};

-

3.2. 灰度图二值化

3.2.1. OSTU 算法:

- 将 gray 拷贝至 binary。
- 遍历整张灰度图，统计 i=0-255 各灰度值占比，记录在 prob[] 中。
- 进行概率累加，记录于 prob_sum[]；进行概率与值累加，记录在 prob_i_sum[]；计算类间方差 variance，记录下取得最大类间方差的灰度值作为阈值。
- 根据阈值遍历整张灰度图，进行阈值比较生成二值图。

```

1. void gray_to_binary_ostu(IMAGE *gray, IMAGE *binary){
2.
3.     copyBMP(gray, binary);
4.
5.     float prob[255]={}, sum_prob[255]={}, sum_i_prob[255]={}, variance[255]={};
6.     int PixelNum=gray->HeightBMP*gray->WidthBMP;
7.     double mean_front, mean_back, variance_max, threshold;
8.
9.     for(int i=0;i<gray->HeightBMP;i++){
10.        for(int j=0;j<gray->WidthBMP;j++){
11.            int graynum=gray->data[gray->LineBytes*i+j];
12.            if(graynum>255)graynum=255;
13.            if(graynum<0)graynum=0;
14.            prob[graynum]++;
15.        }
16.    }

```

```

17.     for(int i=0;i<=255;i++)
18.         prob[i]=prob[i]/PixelNum;
19.
20.     for(int i=0;i<=255;i++){
21.         if(i==0){
22.             sum_prob[i]=prob[i];
23.             sum_i_prob[i]=prob[i]*i;
24.         }else{
25.             sum_prob[i]=sum_prob[i-1]+prob[i];
26.             sum_i_prob[i]=sum_i_prob[i-1]+prob[i]*i;
27.         }
28.     }
29.
30.     for(int i=0;i<=255;i++){
31.         if(sum_prob[i]==0 || sum_prob[i]==1)
32.             continue;
33.         mean_front=(1.0)/sum_prob[i]*sum_i_prob[i];
34.         mean_back=(1.0)/(1-sum_prob[i])*(sum_i_prob[255]-sum_i_prob[i]);
35.         variance[i]=sum_prob[i]*(1-sum_prob[i])*pow((mean_front-mean_back), 2);
36.         if(variance[i]>=variance_max){
37.             variance_max=variance[i];
38.             threshold=i;
39.         }
40.     }
41.     cout << threshold << endl;
42.
43.     for(int i=0;i<binary->HeightBMP;i++){                                //go through
        each pixel
44.         for(int j=0;j<binary->WidthBMP;j++){                            //grayscale
            only needs to calculate Y
45.             double Y=gray->data[gray->LineBytes*i+j];
46.             if(Y>=threshold) Y=255;
47.             else if(Y<threshold) Y=0;
48.             binary->data[binary->LineBytes*i+j]=Y;                        //double ->
            int to improve precision
49.         }
50.     }
51. }
52.

```

3.2.2. 滑动窗口法

- OSTU 算法只能针对全局进行。故而当图片有明显亮度区别时，会有部分区域全白或全黑。为解决上述问题，使用滑动窗口，对图片不同区域分别进行 OSTU 算法。而为模糊边缘，避免分块导致明显边界，引入步长 `stride` (`stride <= window_size`)。通过将窗口重合，重合部分取均值，对窗口边缘进行平滑处理。
- 将 `gray` 拷贝至 `binary`。
- 通过 `i`、`j` 作为窗口的横向、纵向起点，遍历整个灰度图。每次起点增加步长=`stride` 大小。对于不能整除的部分，通过特殊判断扩大边缘窗口的长或宽，获得适应性的 `slide_width` 与 `slide_height` 大小。
- 对每个窗口进行如上的 OSTU 算法，将窗口阈值累加至 `threshold_sum[]` 中对应像素点位置，并将 `threshold_num[]` 记录的窗口数量自增。
- 滑动窗口对整张图片分区使用 OSTU 算法后，遍历所有像素点，在对应处计算均值 `threshold_sum/threshold_num`，将其作为该像素点的阈值。

```
1. void gray_to_binary_sliding(IMAGE *gray, IMAGE *binary){
2.
3.     copyBMP(gray, binary);
4.
5.     int window_size=150, stride=100, slide_height, slide_width;
6.     float prob[255]={}, sum_prob[255]={}, sum_i_prob[255]={};
7.     int PixelNum=gray->HeightBMP*gray->WidthBMP;
8.     double mean_front, mean_back, variance_max, threshold, variance;
9.     WORD *threshold_sum=new WORD[binary->HeightBMP*binary->WidthBMP];
10.    WORD *threshold_num=new WORD[binary->HeightBMP*binary->WidthBMP];
11.    for(int i=0;i<binary->HeightBMP*binary->WidthBMP;i++){
12.        threshold_sum[i]=threshold_num[i]=0;
13.
14.        for(int i=0;i<binary->HeightBMP;i+=stride){
15.
16.            if(binary->HeightBMP<i+window_size) slide_height=binary->HeightBMP-i;
17.            else slide_height=window_size;
18.
19.            for(int j=0;j<binary->WidthBMP-stride;j+=stride){
20.                if(binary->WidthBMP<j+window_size) slide_width=binary->WidthBMP-
                j;
21.                else slide_width=window_size;
22.
23.                memset(prob, 0, sizeof(prob));
```



```

24.     memset(sum_prob, 0, sizeof(sum_prob));
25.     memset(sum_i_prob, 0, sizeof(sum_i_prob));
26.     PixelNum=slide_height*slide_width;
27.     variance=variance_max=0;
28.
29.     for(int ii=i;ii<i+slide_height;ii++){
30.         for(int jj=j;jj<j+slide_width;jj++){
31.             int graynum=gray->data[gray->LineBytes*ii+jj];
32.             if(graynum>255)graynum=255;
33.             if(graynum<0)graynum=0;
34.             prob[graynum]++;
35.         }
36.     }
37.     for(int k=0;k<=255;k++){
38.         prob[k]=prob[k]/PixelNum;
39.
40.     for(int k=0;k<=255;k++){
41.         if(k==0){
42.             sum_prob[k]=prob[k];
43.             sum_i_prob[k]=prob[k]*k;
44.         }else{
45.             sum_prob[k]=sum_prob[k-1]+prob[k];
46.             sum_i_prob[k]=sum_i_prob[k-1]+prob[k]*k;
47.         }
48.     }
49.
50.     for(int k=0;k<=255;k++){
51.         if(sum_prob[k]==0 || sum_prob[k]==1)
52.             continue;
53.         mean_front=(1.0)/sum_prob[k]*sum_i_prob[k];
54.         mean_back=(1.0)/(1-sum_prob[k])
55.             *(sum_i_prob[255]-sum_i_prob[k]);
56.         variance=sum_prob[k]*(1-sum_prob[k])
57.             *pow((mean_front-mean_back), 2);
58.         if(variance>=variance_max){
59.             variance_max=variance;
60.             threshold=k;
61.         }
62.     }
63.
64.     for(int ii=i;ii<i+slide_height;ii++){
65.         for(int jj=j;jj<j+slide_width;jj++){
66.             threshold_sum[binary->WidthBMP*ii+jj]+=threshold;
67.             threshold_num[binary->WidthBMP*ii+jj]+=1;

```

```

68.     }
69.     }
70. }
71. }
72.
73. cout << "before print" << endl;
74.
75. for(int i=0;i<binary->HeightBMP;i++){
76.     for(int j=0;j<binary->WidthBMP;j++){
77.         double Y=gray->data[gray->LineBytes*i+j];
78.         if(threshold_num[binary->WidthBMP*i+j]==0)
79.             threshold_num[binary->WidthBMP*i+j]=1;
80.         double Z=threshold_sum[binary->WidthBMP*i+j]
81.             /threshold_num[binary->WidthBMP*i+j];
82.         if(Y>=Z) Y=255;
83.         else Y=0;
84.         binary->data[binary->LineBytes*i+j]=Y;
85.     }
86. }
87. }
88.

```

3.3. Dilation 膨胀操作

- 将 binary 拷贝至 dilation。
- 遍历二值图全部像素点，将每一个像素点视作结构元素的中心点，调用结构元素矩阵来获取结构元素像素在二值图中对应像素点的偏置。
- 根据边界判断与 dilation 特性，决定中心像素的 0/255 值。

```

1. void do_dilation(IMAGE *input, IMAGE *output){
2.     copyBMP(input, output);
3.     for(int i=0;i<output->HeightBMP;i++){
4.         for(int j=0;j<output->WidthBMP;j++){
5.             int windowX, windowY;
6.             output->data[i*output->LineBytes+j]=0;
7.             for(int k=1;k<=struct_element_square[0].x;k++){
8.                 windowX=i+struct_element_square[k].x;
9.                 windowY=j+struct_element_square[k].y;
10.                 if(windowX<0||windowY<0||windowX>=output->HeightBMP||windowY>=output->WidthBMP)
11.                     continue;
12.                 if(input->data[windowX*input->LineBytes+windowY]==255){
13.                     output->data[i*output->LineBytes+j]=255;

```

```

14.             break;
15.         }
16.     }
17. }
18. }
19. }

```

3.4. Erosion 腐蚀操作

- 将 binary 拷贝至 erosion。
- 遍历二值图全部像素点，将每一个像素点视作结构元素的中心点，调用结构元素矩阵来获取结构元素像素在二值图中对应像素点的偏置。
- 根据边界判断与 erosion 特性，决定中心像素的 0/255 值。

```

1. void do_erosion(IMAGE *input, IMAGE *output){
2.     copyBMP(input, output);
3.     int all=0, count0=0, count1=0;
4.     for(int i=0;i<output->HeightBMP;i++){
5.         for(int j=0;j<output->WidthBMP;j++){
6.             int windowX, windowY;
7.             output->data[i*output->LineBytes+j]=255;
8.             for(int k=1;k<=struct_element_cross[0].x;k++){
9.                 windowX=i+struct_element_cross[k].x;
10.                windowY=j+struct_element_cross[k].y;
11.                if(windowX<0|windowY<0|windowX>=output->HeightBMP|windowY>=output->WidthBMP){
12.                    // output->data[i*output->LineBytes+j]=0;
13.                    // break;
14.                    continue;
15.                }
16.                if(input->data[windowX*input->LineBytes+windowY]==0){
17.                    output->data[i*output->LineBytes+j]=0;
18.                    break;
19.                }
20.            }
21.        }
22.    }
23. }

```

3.5. Opening 开运算

- 依照定义和函数接口，先 erosion 再 dilation。

```
1. void do_opening(IMAGE *input, IMAGE *output){  
2.     do_erosion(input, output);  
3.     do_dilation(output, output);  
4. }
```

3.6. Closing 闭运算

- 依照定义和函数接口，先 dilation 再 erosion。

```
1. void do_closing(IMAGE *input, IMAGE *output){  
2.     do_dilation(input, output);  
3.     do_erosion(output, output);  
4. }
```

4. 实验环境及运行方法

实验环境：Windows10 系统

TDM-GCC 4.9.2 64-bit 编译器

使用 vscode；代码也可直接在 dev-c++ 打开并编译运行。

5. 实验结果展示


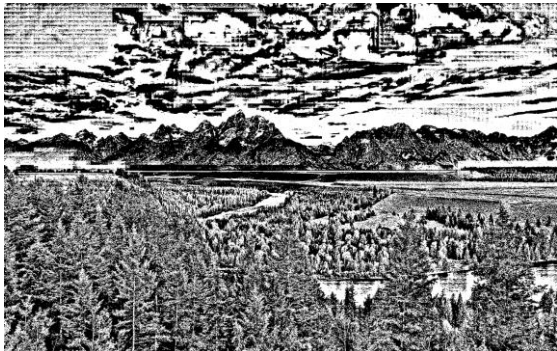
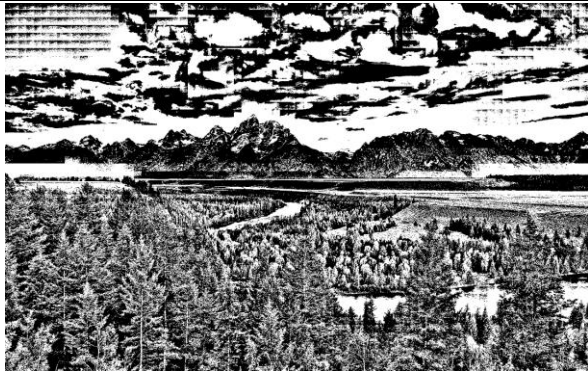

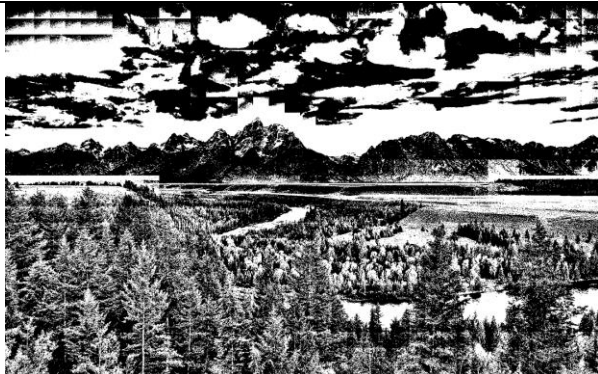
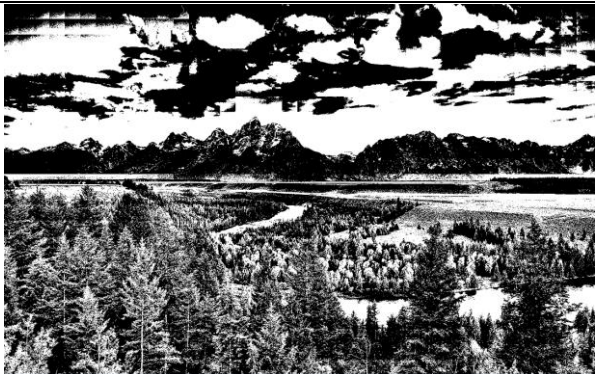
本次实验首先对三张图片进行了二值化操作以比对不同图片、不同算法的区别。三张图片分别为：scenery（局部有细腻亮度区别的风景）、flower（亮度均匀的风景）、classic（经典的区域亮度差别大书页）。

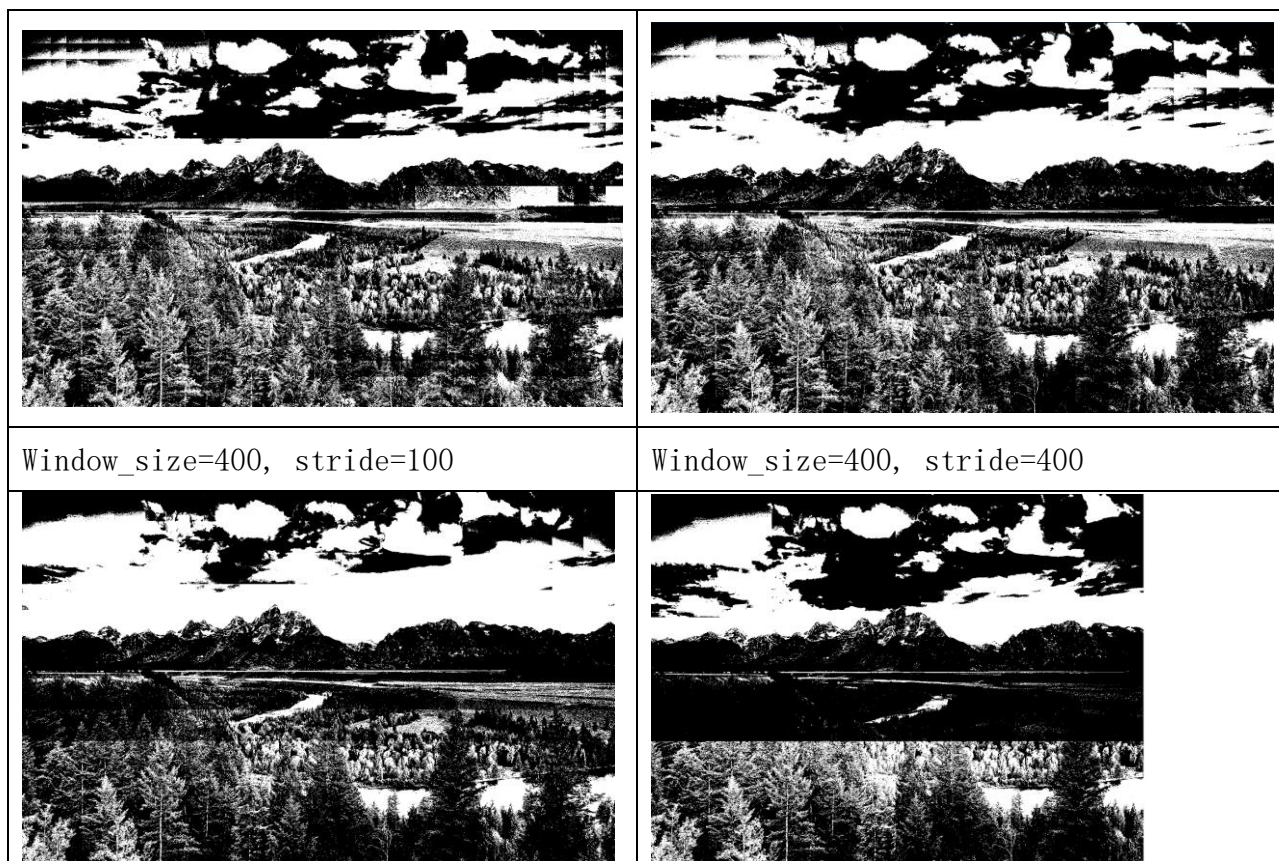


	OSTU	滑动窗口
Scener y		
Flower		
Classi c	<p><small>© 2005 Springer Science+Business Media B.V. All rights reserved.</small></p>	<p><small>© 2005 Springer Science+Business Media B.V. All rights reserved.</small></p>

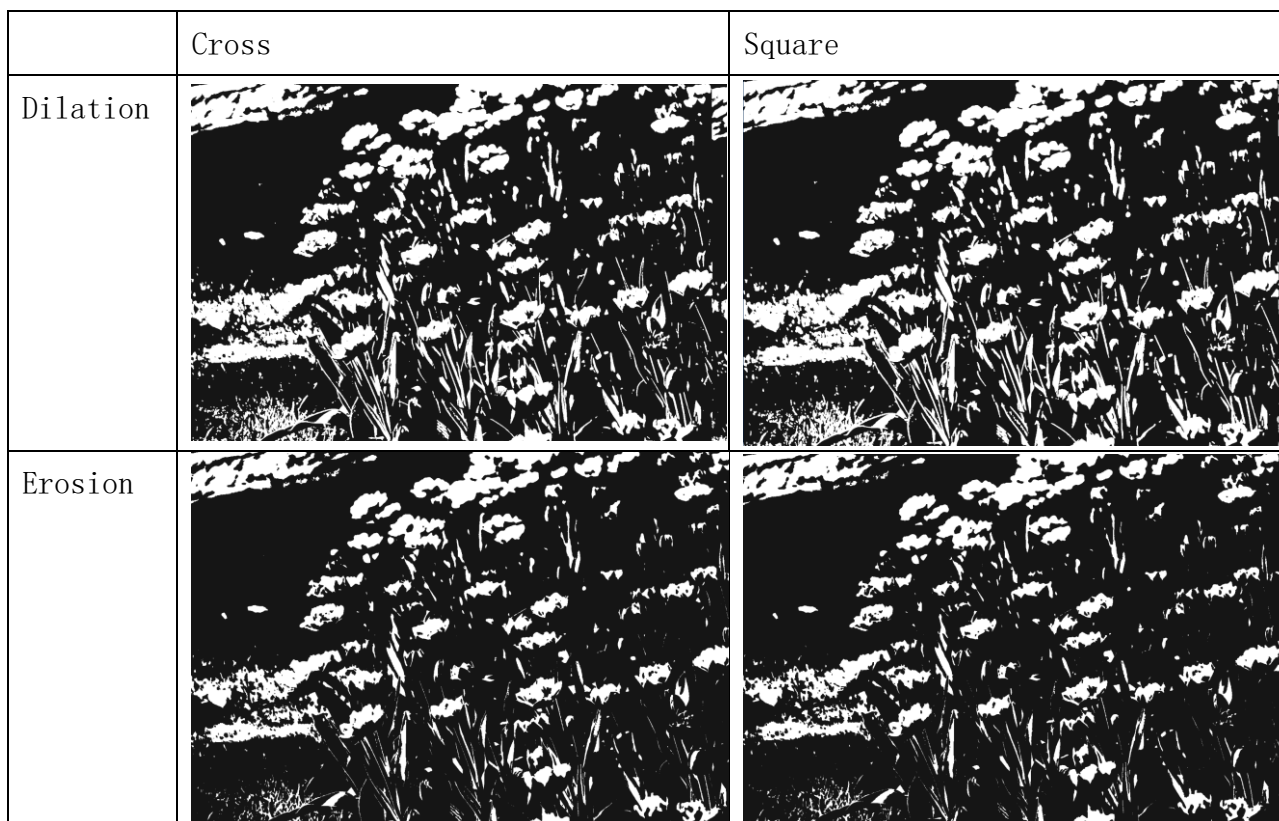
其次，通过更改滑动窗口的窗口尺寸 window_size 与步长 stride，可以比较不同参数下的二值化效果。





Window_size=150, stride=50	Window_size=150, stride=100	Window_size=150, stride=150
Window_size=100, stride=25	Window_size=100, stride=50	Window_size=100, stride=100
Window_size=25, stride=25	Window_size=25, stride=10	Window_size=10, stride=10

Window_size=10, stride=10	Window_size=30, stride=15
	
Window_size=50, stride=25	Window_size=50, stride=50
	
Window_size=100, stride=50	Window_size=150, stride=50
	
Window_size=150, stride=100	Window_size=200, stride=100



最终，对 flower 图片进行 ostu 下的形式化操作：



Opening		
Closing		

六、心得体会

通过这次实验我理解了灰度图二值化的方法，并通过 c++ 实现了 ostu 全局算法和滑动窗口算法。在此之上，我了解了通过形态学和集合运算来处理、增强图像的方法，如腐蚀、膨胀、开闭运算等。

通过比对实验结果，我发现在同样的实验纲领下，不同的具体实现方法有不同的效果。比如，ostu 算法的全局和局部实现在不同的图片有极为不同的效果，有个自适应的领域。比如对于 flower，全局算法更多的保留了花丛原有的形态，而对于 scenery，滑动窗口保留了天空云朵的细节。但这两者都还没能很好解决图片不同区域亮度区别大的问题，有待继续优化。

而对于滑动窗口参数的选择，window_size 越小，处理内容越细腻，保留细节越多，但噪点和干扰也越多。在 window_size 固定的基础上，步长等于窗口大小时，部分复杂图片会产生明显的边界。而通过缩减步长，增大重合部分，可以有效平滑边缘。

对于重合部分的阈值选择，则是均值的效果 > threshold 矩阵平滑处理的效果 > 新阈值取代旧阈值的效果。

最后，在形式学操作中，选择不同的结构元也会让 erosion、dilation 产生细微区别。Cross 保留了更多细节，但边缘也更锐利。Square 则处理较为平滑。本次没有比对结构元形状相同、大小不同，略有遗憾。

而留给本次实验最大的问题是，在调参过程中，手工处理十分复杂。如何利用大模型对图片进行打分，自动选择最好的参数，是一个感兴趣的方向。

综上，本次实验收获满满，希望继续探索对图像的处理。