

第8章

形态学图像处理

目录

- 8.1 形态学图像处理基础
- 8.2 腐蚀与膨胀
- 8.3 开运算与闭运算
- 8.4 击中/击不中变换
- 8.5 二值图像形态学处理应用
- 8.6 灰度图像的形态学处理

背景知识

- 形态学研究物体形状结构，以及物体各部分之间排列及其相互关系，最早应用于语言学及生物学研究领域。
- 数学形态学（Mathematical morphology），又称数字形态学（Digital morphology），是描述和分析数字化对象形状的一种方法，如数字图像中的物体形态。数学形态学与数字计算机相伴而生，是建立在集合论和拓扑学基础之上的日益重要的图像分析工具。
- 数字图像由像素组成，属性相同的像素汇聚成具有特定形状的像素集合，数学形态学常用于增强这些像素集合的某些形状特征，以便对其进行计数或识别。数学形态学的基本运算包括腐蚀、膨胀、开运算、闭运算等。

背景知识

- 数学形态学(Mathematical Morphology)诞生于1964年，是由法国巴黎矿业学院博士生赛拉(J. Serra)和导师马瑟荣，第一次引入了形态学的表达式，建立了颗粒分析方法(Granulometry)，奠定了这门学科的理论基础。
- 数学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。





8.1 形态学图像处理基础

区域与集合

- 二值图像像素只能取两个离散值之一，一个代表“黑”、另一个代表“白”，这两个离散值分别取0和1，也可分别取0和255，视程序语言环境而定。
- 图像分析时，通常称值1像素为前景、值0像素为背景。汇聚在一起的前景像素，形成具有特定形状的区域，它们的形状结构及其相互位置关系，对图像目标的识别和理解至关重要。

0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0
0	1	1	1	0	1	1	0	0
0	1	1	1	0	0	1	1	1
0	1	1	1	0	0	1	1	0
0	1	1	1	0	0	1	0	0
0	1	1	1	0	0	1	0	0
0	1	1	1	0	0	0	0	0



Region-based segmentation

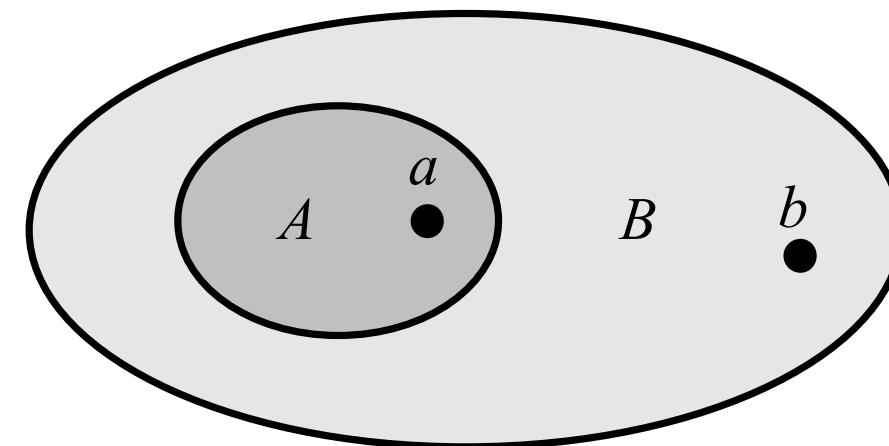
Let us first determine markers of the coin background. These markers are pixels that unambiguously act as either object or background. The markers are found at the two extreme histogram of grey values:

```
>>> markers = np.zeros_like(coins)
```

二值图像示例。硬件显示时将取值为1的像素显示为“白色”、0显示为“黑色”，而印刷时常相反。取值为1的像素用白色或黑色显示，取决于事先的约定。

区域与集合

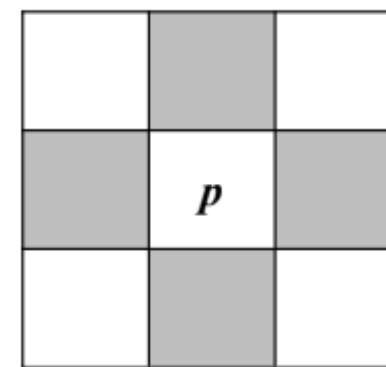
- 集合是由具有某种特定性质的、具体的或抽象的对象汇总而成的集体，集合中的每一个对象称为该集合的元素。
- 因前景像素取值为1，像素之间也满足某种位置关系约束，常用“集合”来描述图像中前景像素汇聚而成的区域，用“元素”来描述汇聚区域中的像素。



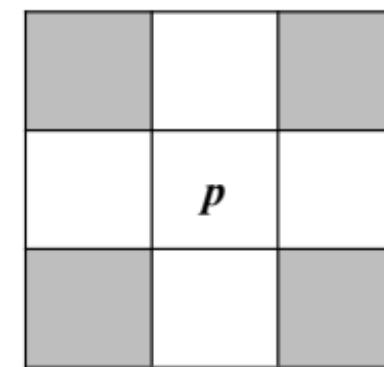
元素、集合与像素、区域之间的对应关系

连通性与区域

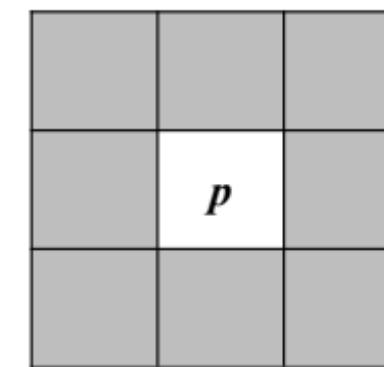
- 对二值图像而言，连通性（connectivities）用于描述取值为1的邻接像素之间的位置关系。
- 邻域：设像素 p 的坐标为 (x,y) ，其邻域是指以坐标 (x,y) 为中心的一组相邻像素构成的集合。
 - 4-邻域
 - 4-对角邻域
 - 8-邻域（8-neighbors）



4-邻域



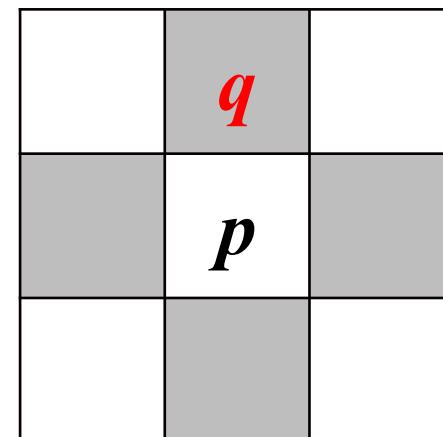
4-对角邻域



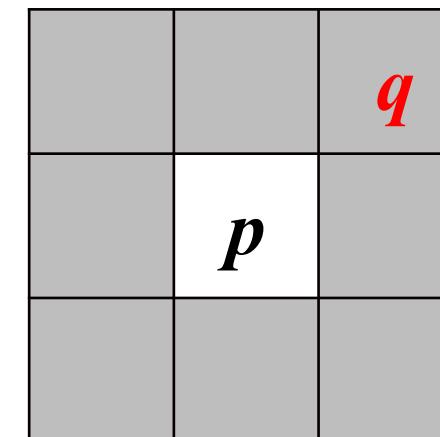
8-邻域

邻接、连通

- **4-邻接和4-连通：**假定像素 p 和 q 的值均为1，如果 q 是 p 的4-邻域像素之一，那么称像素 p 和 q 彼此为4-邻接（4-adjacency），且二者是相互连通的，其连通性称为4-连通（4-connected）。
- **8-邻接和8-连通：**假定像素 p 和 q 的值均为1，如果 q 是 p 的8-邻域像素之一，那么称像素 p 和 q 彼此为8-邻接（8-adjacency），且二者是相互连通的，其连通性称为8-连通（8-connected）。



4-邻接和4-连通



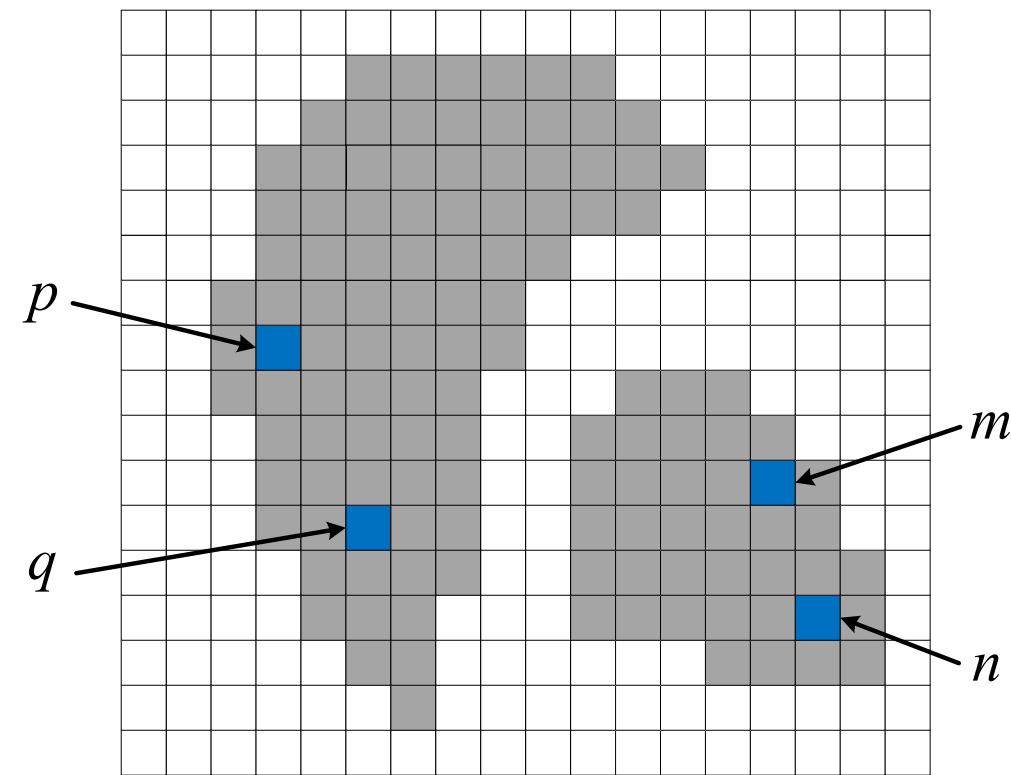
8-邻接和8-连通

通路

◆ **通路：**假定像素 p 的坐标为 (x, y) , 像素 q 的坐标为 (s, t) , 若二者之间存在一个像素序列:

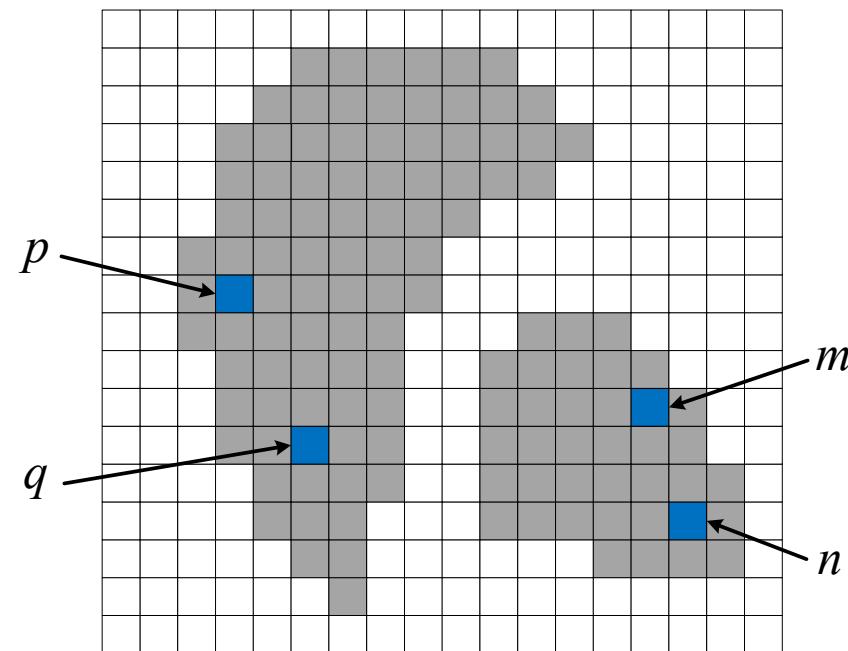
$$(x, y), (x_1, y_1), \dots, (x_n, y_n), (s, t)$$

- 序列中相邻两个像素彼此之间是连通的（4-连通或8-连通），则称像素 p 和 q 之间存在一个**通路**，如果 p 和 q 是同一个像素，则称通路是**闭合通路**。



连通分量、连通集

◆ **连通分量**：令 S 是图像中取值为1的像素子集，如果 S 中任意两个像素 p 和 q 之间存在一个通路，则称像素 p 和 q 是连通的。与像素 p 相连通的所有像素构成的集合，称为 S 的一个连通分量 (connected component) 。



- 通路、连通分量、连通域示意。用灰色表示取值为1的前景像素。
- 像素 p 和 q 之间、 m 和 n 之间存在通路，但 p (或 q)，与 m (或 n) 之间不存在通路。
- 图中前景像素形成了两个连通域或连通分量。

区域 region

- **区域**: 令 R 是图像中取值为1的前景像素的一个子集, 如果 R 是连通集, 则称 R 为一个连通区域 (connected region), 简称连通域、区域 (region)。
- 如果像素之间的连通性为4-连通, 由此形成的连通域称**4-连通域** (4-connected region)。如果像素之间的连通性为8-连通, 由此形成的连通域称**8-连通域** (8-connected region)。

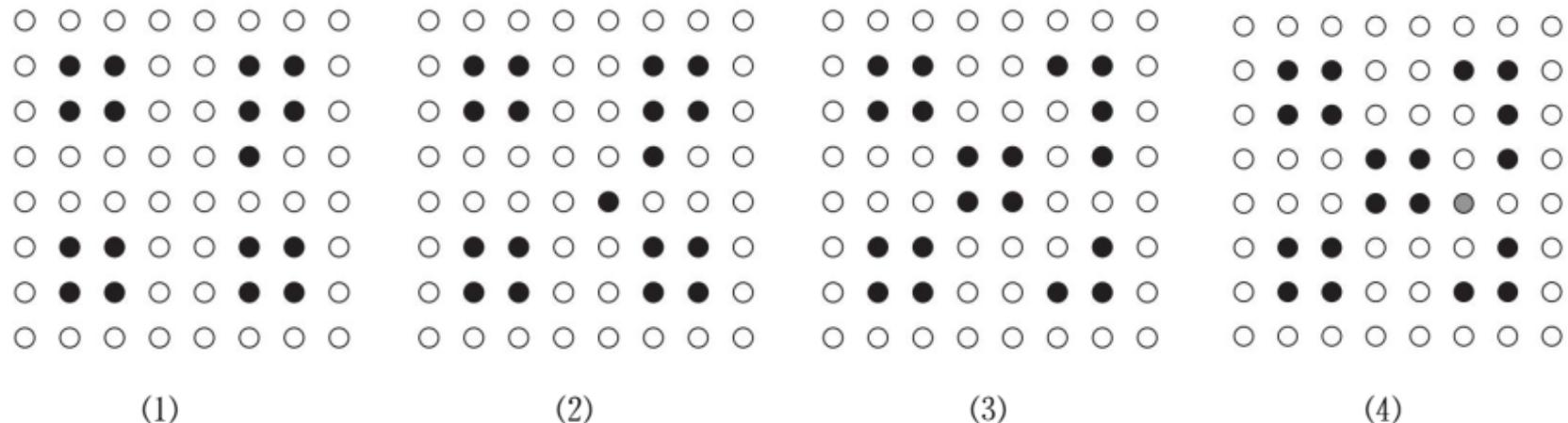


图 8.4 采用不同连通性定义的连通域, 用黑点表示取值为 1 的前景像素。图(1)中的前景像素形成了 4 个 4-连通域; 图(2)中的前景像素形成了 5 个 4-连通域、或 3 个 8-连通域; 图(3)中的前景像素形成了 5 个 4-连通域、或 3 个 8-连通域; 图(4)中如果将图中灰色点改为前景像素, 则融合为 1 个 8-连通域。

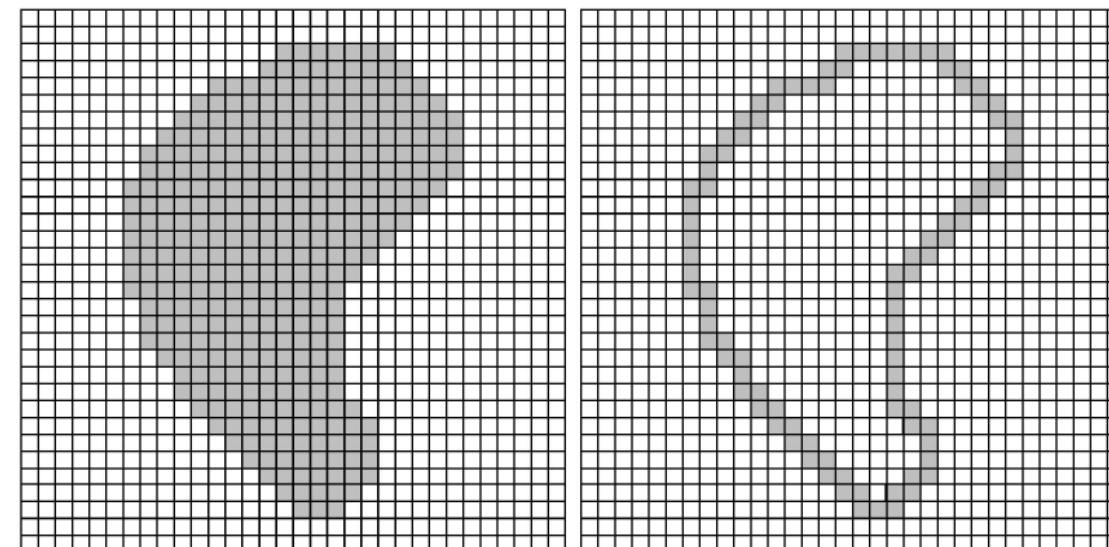
区域 region

- 例：列出所给图中的4-连通域和8-连通域。

	A				
	B		D	E	
		C			
	G	H		J	K
	F		I		

区域的边界

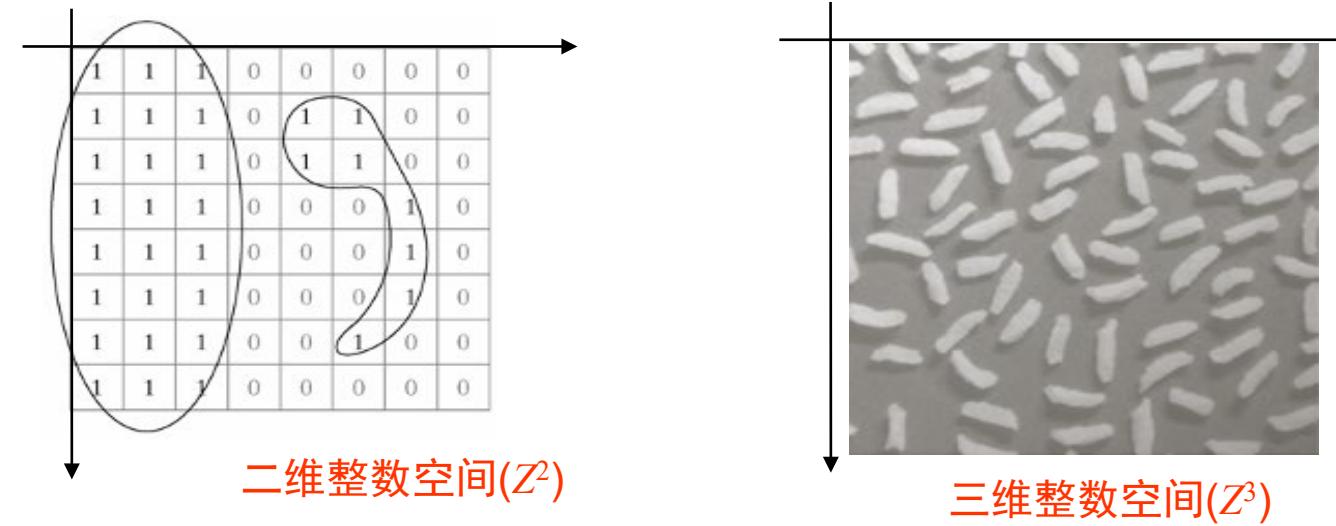
- ◆ **区域R的内边界**，也称边缘或轮廓，是区域R中某些像素的集合，构成边界的像素至少有一个邻域点**不在**区域R中。注意，采用4-邻域或8-邻域得到的边界一般不相同。上述定义的边界点因位于区域内部，故称内边界。
- ◆ **区域R的外边界**，是区域R周边取值为0的某些背景像素的集合，构成外边界的背景像素至少有一个邻域点**在**区域R中。同样，采用4-邻域或8-邻域得到的外边界一般也不相同。



区域的内边界（8-邻域），用灰色表示取值为1的前景像素

数学形态学中的集合和元素

- 对于二值图像，数学形态学中的集合是二维整数空间(Z^2)的一个子集，集合的每个元素都是该平面空间上的一个点（像素），并用一个二维坐标向量 (x, y) 表示，简称为 a 、 b 等。
- 对于灰度图像，数学形态学中的集合可以表示为三维整数空间(Z^3)上分量的集合。集合中每个元素的两个分量是像素的坐标 (x, y) ，第3个分量对应于像素的离散灰度级值 $f(x, y)$ 。

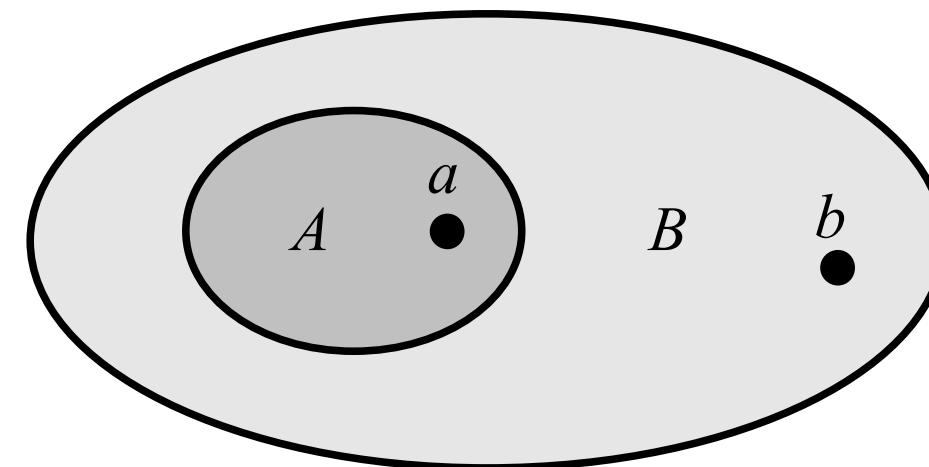


数学形态学中的集合和元素

- 令 A 为二值图像中的一个集合

- 如果像素 $a=(x_1, y_1)$ 是集合 A 的一个元素，则称 a 属于 A ，记为 $a \in A$ ；
- 若像素 $b=(x_2, y_2)$ 不是集合 A 的元素，则称 b 不属于 A ，记为 $b \notin A$ 。
- 如果一个集合没有元素，则称其为空集，用符号 \emptyset 表示。

从位置关系上来看，若 $a \in A$ 则意味着像素 a 在区域 A 内，若 $b \notin A$ 意味着像素 b 不在区域 A 内。



元素、集合与像素、区域之间的对应关系

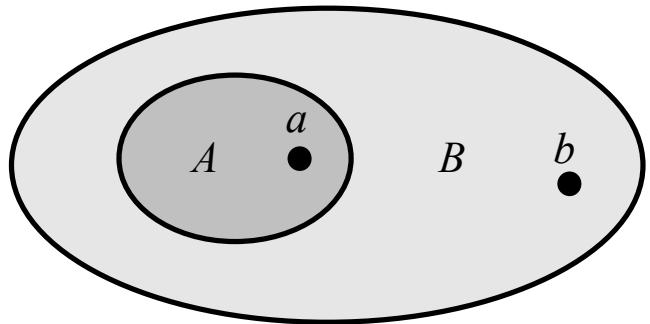
集合运算

- **包含:** 对于两个集合 A 和 B , 如果集合 A 的每一个元素 a ($a \in A$) 又是集合 B 的元素 ($a \in B$), 则称 A 包含于 B , 记作 $A \subseteq B$ 。
- **并集:** $A \cup B = \{a \mid a \in A \text{ 或 } a \in B\}$
- **交集:** $A \cap B = \{a \mid a \in A \text{ 且 } a \in B\}$
- **补集:** $A^c = \{a \mid a \notin A\}$
- **差集:** $A - B = \{a \mid a \in A \text{ 且 } a \notin B\} = A \cap B^c$
- **集合的平移:** 集合 B 平移到点 $p = (x_0, y_0)$, 定义为

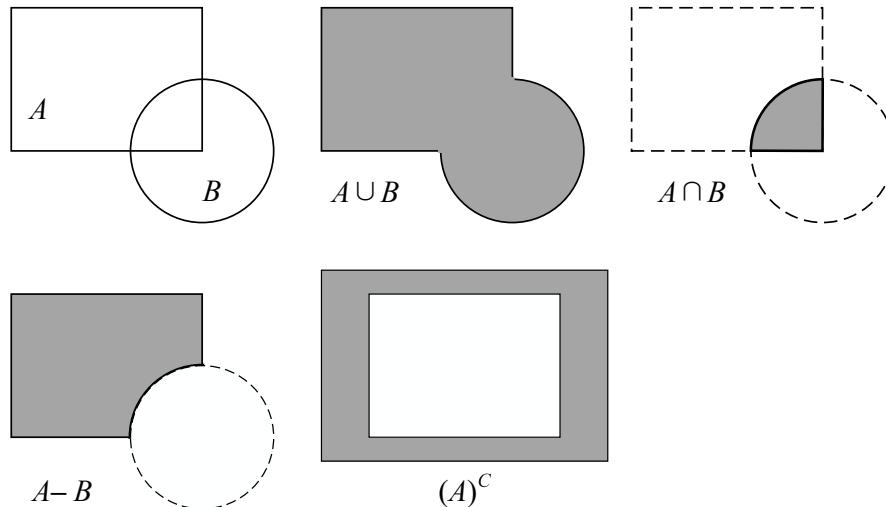
$$(B)_z = \{a \mid a = p + b, b \in B\}$$

若 b 坐标表示为 (x, y) , 则 c 为 $(x+x_0, y+y_0)$

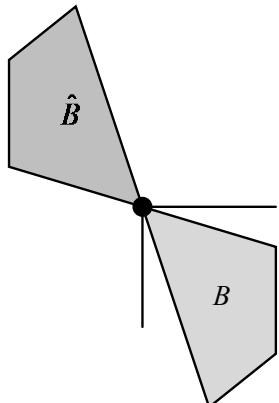
集合运算



元素、集合与像素、区域之间的对应关系

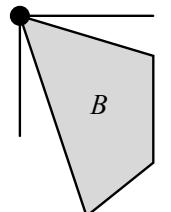


集合的并集、交集、补集、差集运算示意，
灰色区域为运算结果



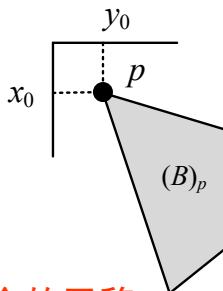
(1)集合的反射

$$\hat{B} = \{a \mid a = -b, b \in B\}$$



(2)集合的平移

$$(B)_z = \{a \mid a = p + b, b \in B\}$$

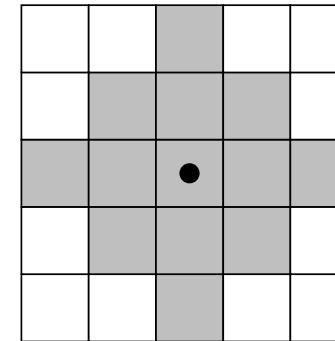
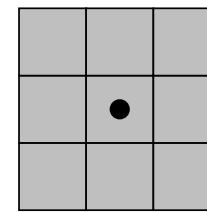
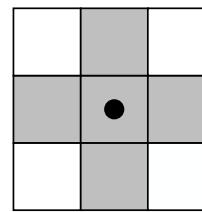
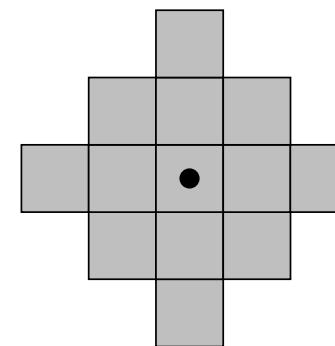
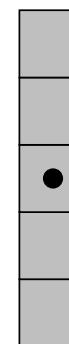
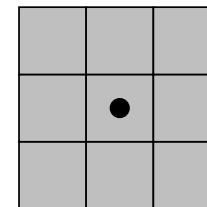
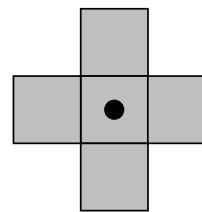


结构元素 structure element

- 形态学图像处理也是一种邻域处理，它用一个称为**结构元素** (structuring element) 的集合或子图块，以某种方式作用于每个像素，从而改变图像中的区域形状及其之间关系，达到分析图像区域特征的目的。
- 结构元素由取值为0和1的点构成，所有取值为1的点形成一个特定的形状，这类结构元素又称为**平坦结构元素** (flat structuring element) 。
- 在构造一个结构元素时，要规定其尺寸大小，并通过指定0和1的位置分布得到特定形状，同时还要指定一个原点，作为结构元素参与形态学运算的参考点，结构元素中各成员的坐标就是依据该原点所建立的坐标系来确定。

结构元素 structure element

- 典型结构元素的形状及其二维数组表示法
- 结构元素的尺寸和形状选择，取决于具体的问题，但一般来说，矩形结构元素倾向于保留锐利的目标区域棱角，而圆盘形结构元素倾向于使围绕目标区域棱角变圆滑。



十字形 (cross) 、 方形 (square) 、 线形 (line) 、 菱形 (diamond)

结构元素 structure element

- OpenCV提供的结构元素构造函数cv.getStructuringElement(), 可以创建矩形、十字形、椭圆形等三种结构元素。

```
#创建一个3×3的“+”字形结构元素  
kernel_cross = cv.getStructuringElement(cv.MORPH_CROSS,(3,3))  
  
#创建一个3×3的正方形结构元素  
kernel_square = cv.getStructuringElement(cv.MORPH_RECT,(3,3))  
  
#创建一个5×1的矩形结构元素,相当于垂直线结构元素  
kernel_rect = cv.getStructuringElement(cv.MORPH_RECT,(1,5))  
  
#创建一个5×5的椭圆形结构元素  
kernel_ellipse = cv.getStructuringElement(cv.MORPH_ELLIPSE,(7,7))
```

结构元素 structure element

- Scikit-image在morphology模块中提供了square、rectangle、diamond、disk、octagon、star等二维结构元素，以及cube、octahedron、ball等三维结构元素创建函数。也可以直接通过构造NumPy数组、给指定元素赋值的方式创建结构元素。

```
selem_sq = morphology.square(5)      # 创建一个5×5的正方形结构元素  
selem_rect = morphology.rectangle(3,5) # 创建一个3×5的矩形结构元素  
selem_dia = morphology.diamond(3)    # 创建一个半径为3的菱形结构元素  
selem_disk = morphology.disk(3)      # 创建一个半径为3的圆盘形结构元素  
selem_oct = morphology.octagon(7,4)   # 创建一个7×4的八边形结构元素  
selem_sq3 = np.ones((3,3),np.uint8)   # Numpy数组创建3×3的正方形结构元素
```



8.2 腐蚀与膨胀

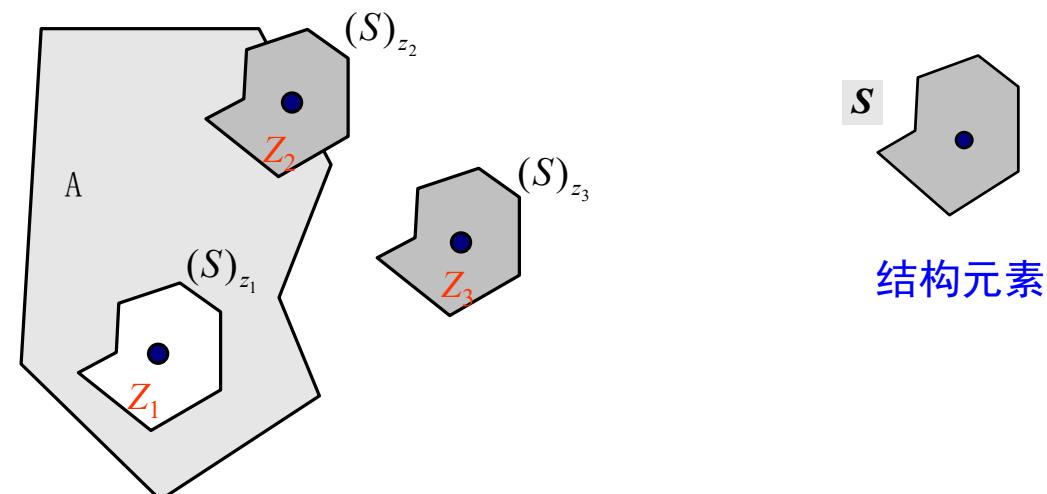
腐蚀 Erosion

- 设 A 是二值图像中的集合， S 为结构元素。用 S 对 A 进行腐蚀（Erosion），或 A 被 S 腐蚀，记作 $A \ominus S$ ，定义为：

$$A \ominus S = \{ z \mid (S)_z \subseteq A \}$$

上式表明， A 被 S 腐蚀的结果，是将 S 平移到 z 后、 S 仍包含在 A 中的所有 z 构成的集合。

S 被 z 平移的三种可能的状态



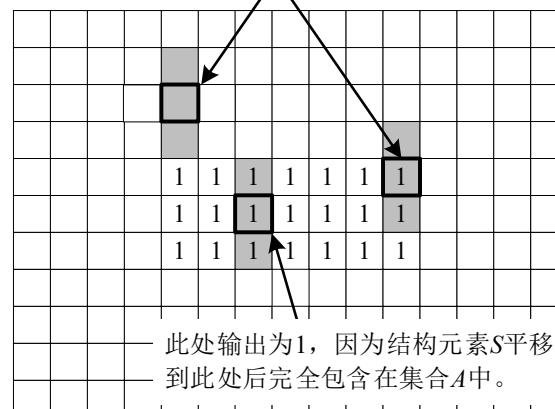
腐蚀 Erosion

(1)集合 A

1
1
1

(2)线形结构元素S

这些位置的腐蚀输出为0，因为结构元素S平移到此处后已不包含在集合A中。

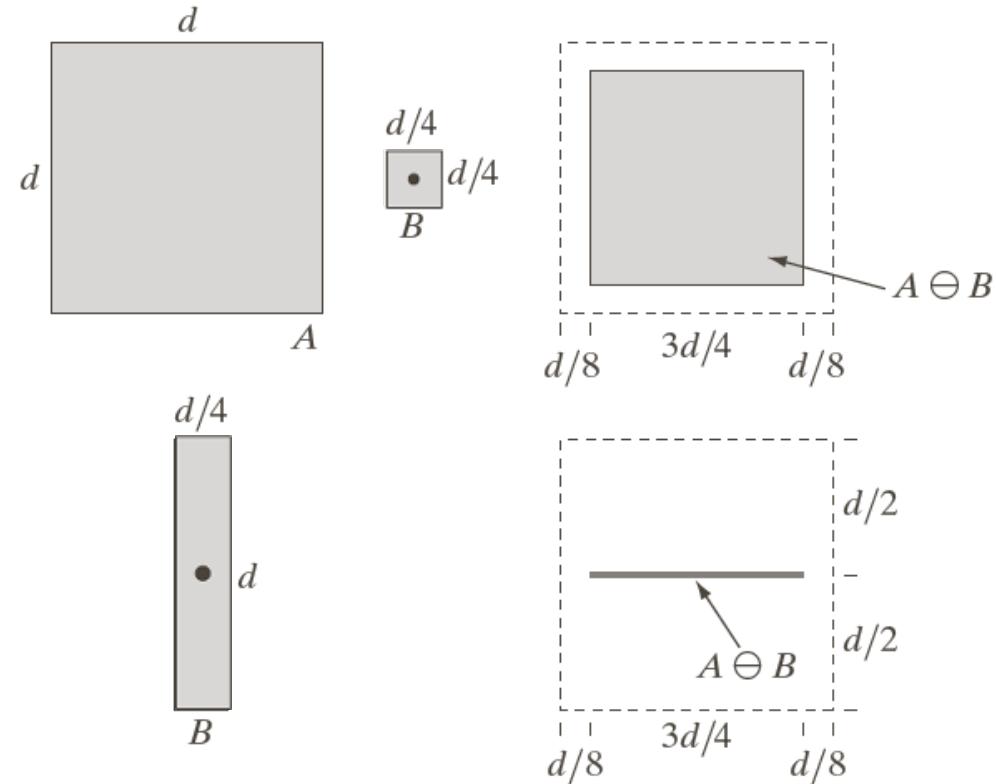


(3) 结构元素S平移到的几个位置及其输出

此处输出为1，因为结构元素S平移到此处后完全包含在集合A中。

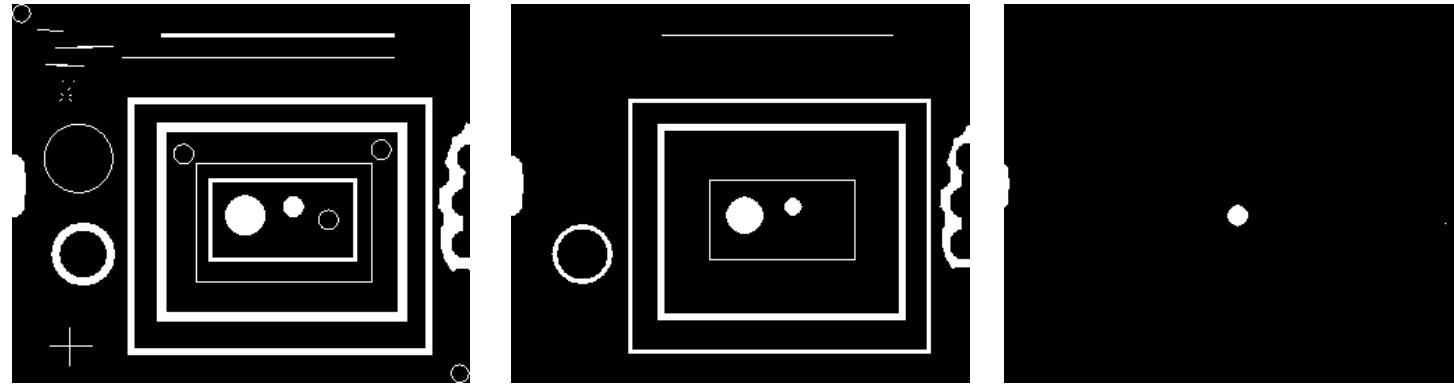
(4)集合 A 被 S 腐蚀的结果

腐蚀 Erosion



动画演示

示例：形态学腐蚀运算



(1)原二值图像

(2)用 3×3 方形结构元素腐蚀

(3)用 15×15 椭圆形结构元素腐蚀

```
#导入本章示例用到的包
```

```
import numpy as np  
import cv2 as cv  
from skimage import io, color, util, morphology, filters, measure  
from scipy import ndimage  
import matplotlib.pyplot as plt  
%matplotlib inline
```

示例：形态学腐蚀运算

```
#OpenCV: 二值图像形态学腐蚀
img = cv.imread('./imagedata/blobs.png', 0) #读入二值图像
#创建3×3方形结构元素
kernel_square = cv.getStructuringElement(cv.MORPH_RECT, (3,3))
#kernel_square = np.ones((3,3),np.uint8)
#对二值图像进行腐蚀
img_erode1 = cv.erode(img, kernel_square, iterations = 1)
#img_erode1 = cv.morphologyEx(img, cv.MORPH_ERODE, kernel_square)
#创建大小为15×15椭圆形结构元素
kernel_ellipse = cv.getStructuringElement(cv.MORPH_ELLIPSE, (15,15))
#对二值图像进行腐蚀
img_erode2 = cv.erode(img, kernel_ellipse)
#显示结果（略，详见本章Jupyter Notebook可执行笔记本文件）
```

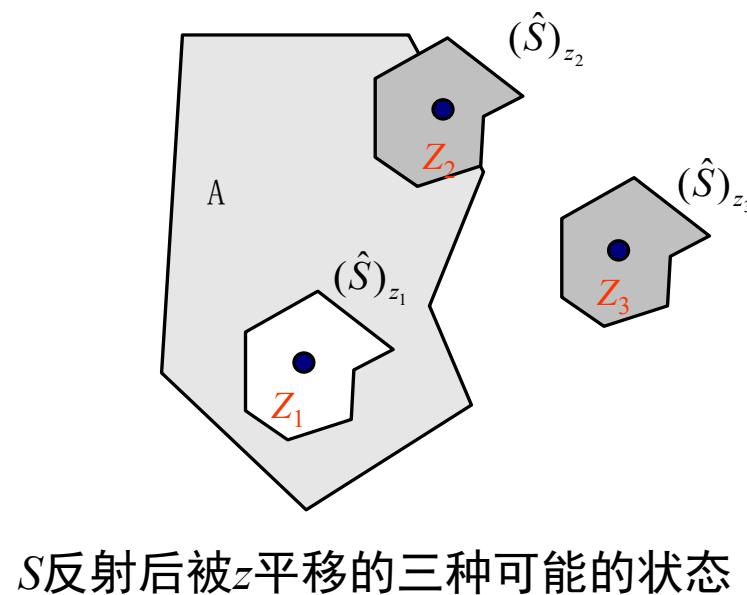
示例：形态学腐蚀运算

```
#Scikit-image: 二值图像形态学腐蚀示例
img = io.imread('./imagedata/blobs.png') #读入二值图像
#创建3×3正方形结构元素
selem_sq = morphology.square(3)
#对二值图像进行腐蚀
imgout1 = morphology.binary_erosion(img, selem_sq)
#结果数据类型为bool型,将其转换为uint8型
imgout1 = util.img_as_ubyte(imgout1)
#创建半径为7个像素的圆盘形结构元素
selem_disk = morphology.disk(7)
#对二值图像进行腐蚀
imgout2 = morphology.binary_erosion(img, selem_disk)
#结果数据类型为bool型,将其转换为uint8型
imgout2 = util.img_as_ubyte(imgout2)
```

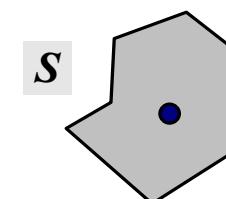
膨胀 Dilation

- 设 A 是二值图像中的集合， S 为结构元素。用 S 对 A 进行膨胀（dilation），或 A 被 S 膨胀，定义为：

$$A \oplus S = \left\{ z \mid (\hat{S})_z \cap A \neq \emptyset \right\}$$



上式表明， A 被 S 膨胀的结果是将 S 反射后平移到 z 、 \hat{S} 与 A 至少有一个元素重叠的所有 z 的集合。



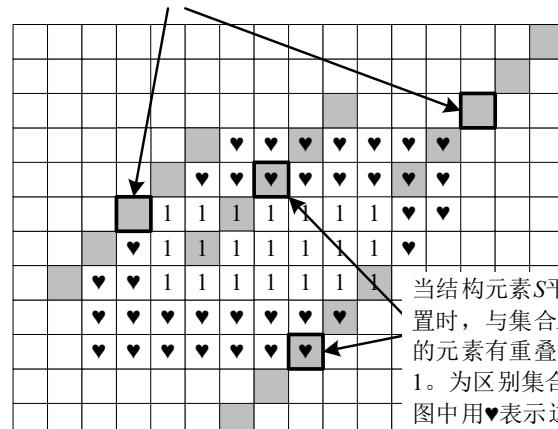
结构元素

膨胀 Dilation

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(1)集合A

结构元素S平移到这些位置，与集合A中取值为1的元素无重叠，故输出为0。



(3)结构元素S平移到的几个位置及其输出

1
1
1
1
1

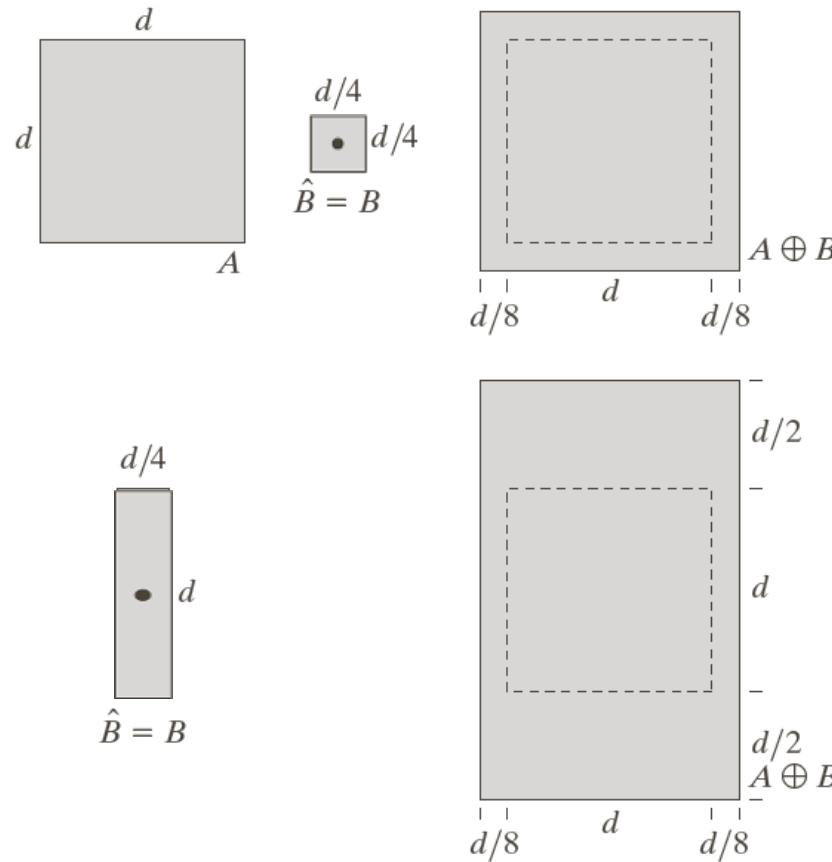
(2)线形结构元素S, 原点对称, 故有 $\hat{S} = S$

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0
0
0
0
0 0

(4)集合A被S膨胀的结果

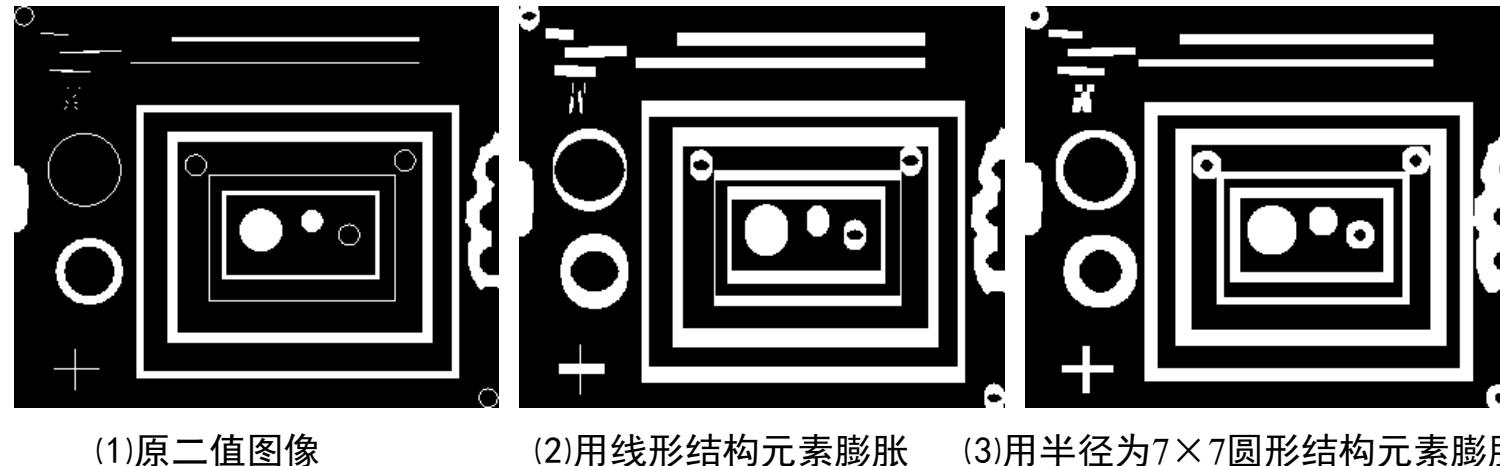
当结构元素S平移到这些位
置时，与集合A中取值为1
的元素有重叠，故输出为
1。为区别集合A中元素，
图中用♥表示这些位置。

膨胀 Dilation



动画演示

示例：二值图像形态学膨胀



```
#OpenCV: 二值图像形态学膨胀
img = cv.imread('./imagedata/blobs.png', 0) #读入二值图像
kernel_line = np.ones((7,1),np.uint8)      #创建高7像素垂直线形结构元素
img_dilate1 = cv.dilate(img, kernel_line)  #膨胀
#img_dilate1 = cv.morphologyEx(img, cv.MORPH_DILATE, kernel_line, iterations = 1)
#创建7×7圆形结构元素
kernel_ellipse = cv.getStructuringElement(cv.MORPH_ELLIPSE, (7,7))
img_dilate2 = cv.dilate(img, kernel_ellipse) #膨胀
```

示例：二值图像形态学膨胀

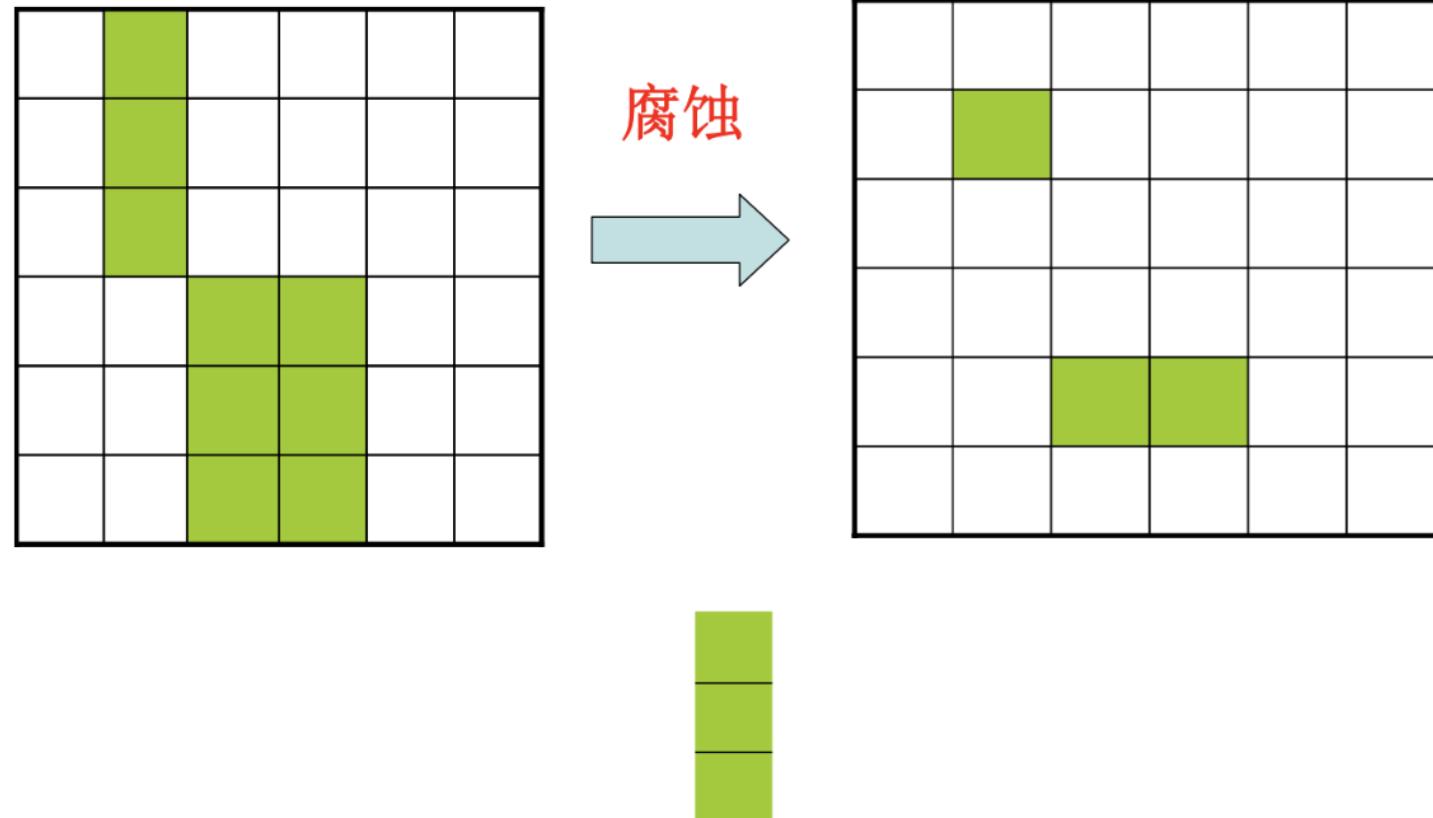
```
#Scikit-image: 二值图像形态学膨胀示例
#读入二值图像
img = io.imread('./imagedata/blobs.png')
#创建高7像素垂直线形结构元素
selem_line = np.ones((7,1),np.uint8)
#膨胀
imgout1 = morphology.binary_dilation(img, selem_line)
#返回结果数据类型为bool型,将其转换为uint8型
imgout1 = util.img_as_ubyte(imgout1)
#创建半径为3个像素的圆盘形结构元素
selem_disk = morphology.disk(3)
#膨胀
imgout2 = morphology.binary_dilation(img, selem_disk)
#返回结果数据类型为bool型,将其转换为uint8型
imgout2 = util.img_as_ubyte(imgout2)
```

腐蚀与膨胀的作用

- 腐蚀使图像中物体区域缩小，可以把小于结构元素的物体(毛刺、小凸起)去除，这样选取不同大小的结构元素，就可以在原图像中去掉不同大小的物体。
 - 如果两个物体之间有细小的连通，那么当结构元素足够大时，通过腐蚀运算可以将两个物体区域分开。
- 膨胀使图像中物体区域扩大，将与物体接触的有关背景点合并到该物体中，使边界向外部扩张。膨胀可以用来填补物体中的空洞，或桥接小的裂缝。

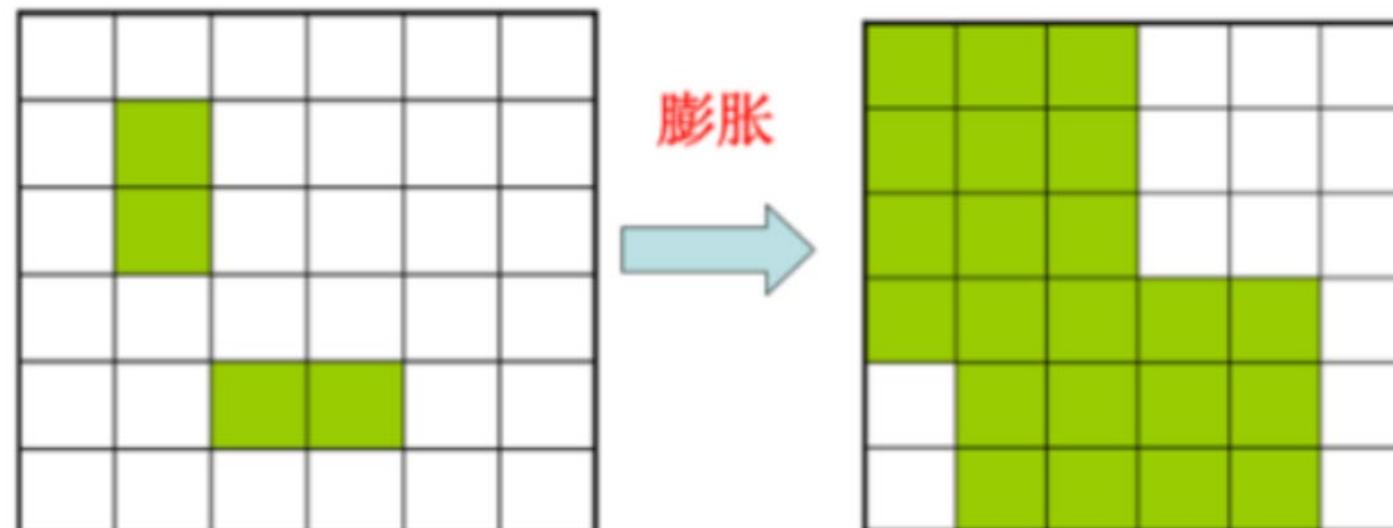
腐蚀与膨胀

- 例：绘制腐蚀后的结果：



腐蚀与膨胀

- 例：绘制膨胀后的结果：



原始图像

膨胀结果





8.3 开运算与闭运算

开运算与闭运算

- **开运算**: 用结构元素 S 对集合 A 先腐蚀、再膨胀的过程称为开运算 (Opening) , 表示为 $A \circ S$, 定义为:

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

- **闭运算**: 用结构元素 S 对集合 A 先膨胀、再腐蚀的过程称为闭运算 (Closing) , 表示为 $A \bullet S$, 定义为:

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

示例：开运算和闭运算的形态学滤波去噪



```
#OpenCV: 开运算/闭运算的形态学滤波去噪
img = cv.imread('./imagedata/noisy_rectangle.png', 0) #读入二值图像
#创建25×25正方形结构元素
kernel_square = cv.getStructuringElement(cv.MORPH_RECT, (25,25))
img_open = cv.morphologyEx(img, cv.MORPH_OPEN, kernel_square) #开运算
img_close = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel_square) #闭运算
#对图像先开运算再闭运算
img_oc = cv.morphologyEx(img, cv.MORPH_OPEN, kernel_square)
img_oc = cv.morphologyEx(img_oc, cv.MORPH_CLOSE, kernel_square)
```

示例：开运算和闭运算的形态学滤波去噪

```
#Scikit-image: 形态学开运算/闭运算的形态学滤波去噪示例
img = io.imread('./imagedata/noisy_rectangle.png') #读入二值图像
#创建25*25正方形结构元素
selem_sq = morphology.square(25)
imgout1 = morphology.binary_opening(img,selem_sq) #开运算
#返回结果数据类型为bool型,将其转换为uint8型
imgout1 = util.img_as_ubyte(imgout1)
imgout2 = morphology.binary_closing(img,selem_sq) #闭运算
#返回结果数据类型为bool型,将其转换为uint8型
imgout2 = util.img_as_ubyte(imgout2)
#对图像先开运算再闭运算
imgout3 = morphology.binary_opening(img,selem_sq)
imgout3 = morphology.binary_closing(imgout3,selem_sq)
#返回结果数据类型为bool型,将其转换为uint8型
imgout3 = util.img_as_ubyte(imgout3)
```

开运算和闭运算的功能

- **开运算先用腐蚀运算去除图像中小于结构元素的前景区域，剩下的前景区域被随后的膨胀运算恢复到近似原始尺寸。**
- **开运算一般能断开前景区域之间狭窄的连结，消除指定小尺寸的前景区域、或前景区域中细的突出物，使区域的轮廓变得光滑。**
- **闭运算可以填补前景区域中小于结构元素的孔洞和缝隙，或令前景区域中的孔洞和缝隙变小。**
- **闭运算能弥合前景区域间狭窄的间断，去除小的孔洞，并填补轮廓线中的断裂。**



8.4 击中/击不中变换

击中/击不中变换

- 击中/击不中(Hit-or-Miss)变换是形状检测的一个基本工具。
- 设 A 是二值图像中的集合，令 S 代表两个无重叠成员的结构元素(S_1, S_2)，其中，结构元素 S_1 描述了与取值为1的前景像素有关的特定结构形状；结构元素 S_2 描述了与 S_1 邻域取值为0背景像素有关的特定结构形状。用 S 对 A 做击中/击不中变换 (hit-or-miss)，定义为：

$$A \odot S = (A \ominus S_1) \cap (A^C \ominus S_2)$$

示例：用击中/击不中变换检测二值图像中的“十”字型结构

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	1	1
1	0	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	0	0
1	1	0	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0
1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0	0
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1
1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(1) 集合 A

(2) 结构元素 S_1 和 S_2

(3) $A \ominus S_1$

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	1	1
1	0	0	0	1	1	1	1	1	1	0	0	1	1	1	1	0	0	0	0
1	1	0	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0
1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	0	0	0
1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0

(4) 集合 A 的补集 A^C

(5) $A^C \ominus S_2$

(6) $(A \ominus S_1) \cap (A^C \ominus S_2)$

示例：用击中/击不中变换检测二值图像中的“十”字型结构

```
#OpenCV: 形态学击中/击不中变换
#构造一幅二值图像
imgbw = np.array(
    [[0,0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,1,0,0,0,0,0,0,0,0,0,0],
     [0,0,1,0,0,0,1,1,1,1,0,0,0,0],
     [0,1,1,1,0,0,0,0,0,0,1,1,0,0],
     [0,0,1,0,0,0,0,0,0,0,1,1,1,0],
     [0,0,0,0,1,0,0,0,0,0,0,1,0,0],
     [0,0,0,0,1,1,1,0,0,0,0,0,0,0],
     [0,0,0,0,1,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0,0,0,0]]).astype(np.uint8)
#创建3×3十字形结构元素，-1为背景，1为前景
kernel = np.array([[-1,1,-1],
                   [ 1,1, 1],
                   [-1,1,-1]])
#击中/击不中变换
imgout = cv.morphologyEx(imgbw, cv.MORPH_HITMISS, kernel)
#显示计算结果数组
print('hit_or_miss result:\n', imgout.astype(np.uint8))
```



8.5 二值图像形态学处理应用

边界提取

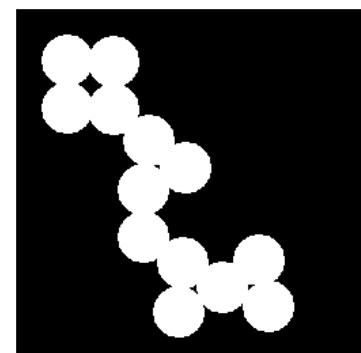
- ◆ 区域 R 内边界的提取方法为：先用结构元素 S 对 R 腐蚀，然后用 R 减去上述腐蚀结果，即：

$$\beta(R) = R - (R \ominus S)$$

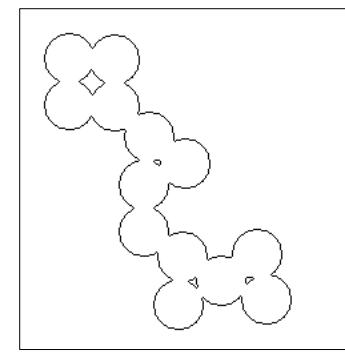
- ◆ 区域 R 外边界的提取方法为，先用结构元素 S 对 R 膨胀，然后用膨胀结果减去 R ，即：

$$\beta(R) = (R \oplus S) - R$$

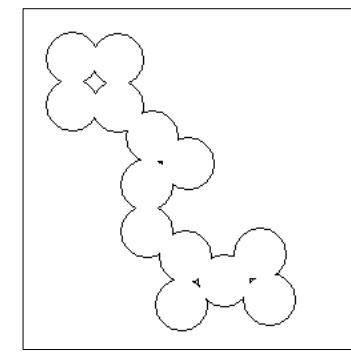
其中，结构元素 S 多采用 3×3 “+”字形或 3×3 方形。



(1)二值图像



(2)提取的内边界



(3)提取的外边界

边界提取

```
#二值图像中区域边界的提取
img = io.imread('./imagedata/circles.png') #读入二值图像
#创建3×3“十”字形结构元素
se = np.array([[0,1,0], [1,1,1], [0,1,0]]).astype(np.uint8)
#区域内边界
img_erosion = morphology.binary_erosion(img, se) #对图像腐蚀
#结果数据类型为bool型，将其转换为uint8型
img_erosion = util.img_as_ubyte(img_erosion)
#从原图像中减去腐蚀结果得到区域内边界
img_Boundary1 = img - img_erosion
#区域外边界
img_dilation = morphology.binary_dilation(img, se) #对图像膨胀
#结果数据类型为bool型，将其转换为uint8型
img_dilation = util.img_as_ubyte(img_dilation)
#膨胀结果减去原图像得到区域外边界
img_Boundary2 = img_dilation - img
```

孔洞填充

- 一个孔洞定义为由前景像素相连的边界所包围的一个背景区域。孔洞填充就是把孔洞对应的背景像素赋值为前景。
- A 表示一个包含子集的集合，其子集的元素均是区域的 8 连通边界点。目的是从边界内的一个孔洞点 p 开始，用“1”填充整个区域。
- 方法（迭代过程）：

条件膨胀， S 为适当的对称性结构元素

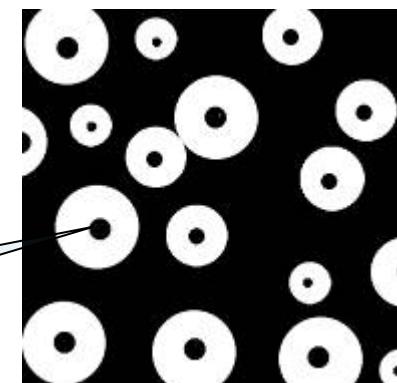
$$X_0 = X$$

$$X_k = (X_k \oplus S) \cap A^C, k = 1, 2, 3, \dots$$

直到 $X_k = X_{k+1}$

最终结果 = $X_k \cup A$

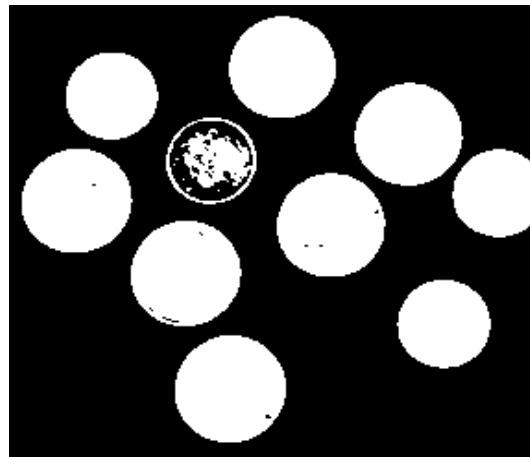
hole



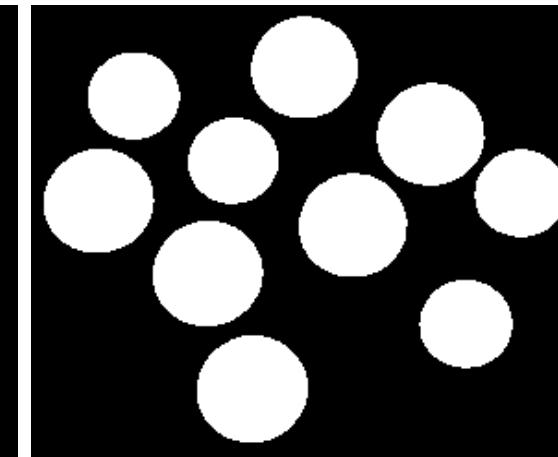
示例：孔洞填充



(1)硬币图像



(2)阈值分割得到的二值图像



(3)孔洞填充后的图像

#SciPy: 孔洞填充示例

```
img = io.imread('./imagedata/coins.jpg') #读取一幅灰度图像
#对灰度图像进行阈值分割得到二值图像
thresh = filters.threshold_otsu(img) #采用Otsu方法得到最优阈值
img_bw = img > thresh #阈值分割,结果数据类型为bool型
img_fill = ndimage.binary_fill_holes(img_bw) #填充孔洞
#结果数据类型为bool型,将其转换为uint8型
img_fill = util.img_as_ubyte(img_fill)
```

示例：孔洞填充

#OpenCV: 区域孔洞填充

```
img = cv.imread('./imagedata/coins.jpg', 0) #读取一幅灰度图像
#对灰度图像进行阈值分割得到二值图像,
th, imgbw = cv.threshold(img, 0, 255, cv.THRESH_OTSU|cv.THRESH_BINARY)
img_floodfill = imgbw.copy()
rows, cols = img.shape[0:2] #获取图像高/宽
mask = np.zeros([rows+2, cols+2], np.uint8)
#找到背景点, floodFill函数中的seedPoint应是背景点
exit_flag = False
for i in range(imgbw.shape[0]):
    for j in range(imgbw.shape[1]):
        if(imgbw[i][j]==0):
            seedPoint=(j,i)
            exit_flag = True
            break
        if exit_flag: break
cv.floodFill(img_floodfill, mask, seedPoint, 255) #采用漫水填充
img_floodfill_inv = cv.bitwise_not(img_floodfill) #对填充结果取非
#将imgbw与img_floodfill_inv进行“或”运算, 得到最终结果
img_filled = cv.bitwise_or(imgbw, img_floodfill_inv)
```

连通域的标记提取

#OpenCV: 连通域的提取之标记

#构造一幅二值图像

```
imgbw = np.array(  
    [[0,0,0,0,0,0,0,0,0,0,0,0,0,0],  
     [0,0,1,0,0,0,0,0,0,0,0,0,0,0],  
     [0,0,1,0,0,0,1,1,1,0,0,0,0,0],  
     [0,1,1,1,0,0,0,0,0,0,1,1,0,0],  
     [0,0,1,0,0,0,0,0,0,0,1,1,1,0],  
     [0,0,0,0,0,1,0,0,0,0,0,1,0,0],  
     [0,0,0,0,1,1,1,0,0,0,0,0,0,0],  
     [0,0,0,0,0,1,0,0,0,0,0,0,0,0],  
     [0,0,0,0,0,0,0,0,0,0,0,0,0,0]]).astype(np.uint8)
```

#对二值图像连通域进行标记

```
retval, img_labels=cv.connectedComponents(imgbw, connectivity=8)
```

#显示图像中前景区域数量，标记后的图像

```
print('Number of regions:', retval-1)
```

```
print('Labeled image:\n', img_labels)
```

输出结果为：

Number of regions: 4

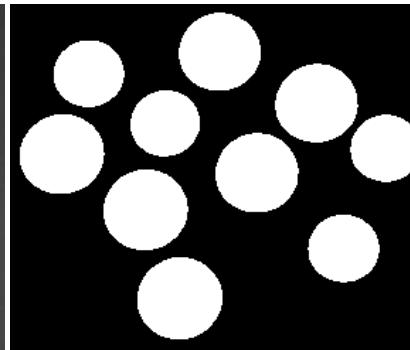
Labeled image:

```
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0]  
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0]  
 [0 0 1 0 0 0 2 2 2 2 0 0 0 0]  
 [0 1 1 1 0 0 0 0 0 0 3 3 0 0]  
 [0 0 1 0 0 0 0 0 0 0 3 3 0 0]  
 [0 0 0 0 4 0 0 0 0 0 3 0 0 0]  
 [0 0 0 0 4 4 4 0 0 0 0 0 0 0]  
 [0 0 0 0 4 0 0 0 0 0 0 0 0 0]  
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

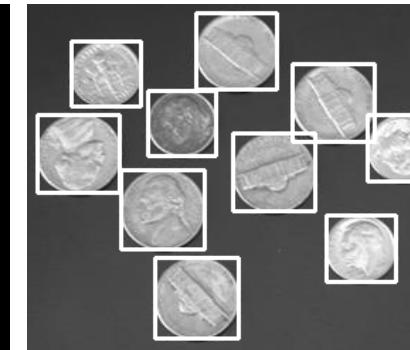
示例：连通域的标记提取与属性测量



硬币图像



阈值分割及孔洞填充结果



叠加包围盒的连通域

```
#OpenCV: 二值图像中连通域的提取及区域属性计算
```

```
img = cv.imread('./imagedata/coins.jpg',0) #读取一幅灰度图像
rows, cols = img.shape[0:2] #获取图像高/宽
#采用Otsu方法对灰度图像进行阈值分割得到二值图像
th, imgbw = cv.threshold(img, 0, 255, cv.THRESH_OTSU|cv.THRESH_BINARY)
#填充孔洞, 采用SciPy函数
img_filled = ndimage.binary_fill_holes(imgbw)
#结果数据类型为bool型, 将其转换为uint8型
img_filled = util.img_as_ubyte(img_filled)
```

示例：连通域的标记提取与属性测量

```
#对二值图像进行标记，并计算图像中连通域的属性  
retval, labels, stats, centroids=cv.connectedComponentsWithStats(img_filled)  
#将每个区域的包围盒叠加绘制到原图像上  
#返回区域属性stats为retval×5的二维数组，第1行为标号0的背景区域属性  
#[x,y,width,height,area]  
imgresult = img.copy()  
for rlabel in range(1,stats.shape[0]): #画包围盒  
    rect = stats[rlabel,0:4]  
    imgresult = cv.rectangle(imgresult, rect, 255,2)  
#显示结果（略）
```



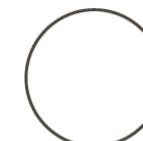
8.6 灰度图像的形态学处理

灰度图像形态学处理

- 二值形态学的 4 个基本运算，即腐蚀、膨胀、开、闭运算，可推广到灰度图像空间。与二值形态学不同的是，这里运算对象不再看作集合而看做离散图像函数。设 $f(x, y)$ 是输入图像， $b(x, y)$ 是结构元素。
- 结构元素：非平坦、平坦



(1) 非平坦结构元素



(2) 平坦结构元素



(3) 非平坦灰度值剖面示意 (4) 平坦灰度值剖面示意



$$\hat{b}(x, y) = b(-x, -y)$$

灰度图像的腐蚀

- 令 f 表示一幅灰度图像、 b 表示平坦结构元素，用 b 对图像 f 腐蚀，记作 $[f \ominus b]$ ，定义为：

$$[f \ominus b](x, y) = \min_{(s, t) \in b} \{f(x + s, y + t)\}$$

- 把结构元素 b 的原点平移到图像 f 的像素 (x, y) 处，该像素被腐蚀后的灰度值，为图像 f 与结构元素 b 重合区域内像素值的最小值。

- 若采用非平坦结构元素 b_N 对图像 f 腐蚀，定义为：

$$[f \ominus b_N](x, y) = \min_{(s, t) \in b} \{f(x + s, y + t) - b_N(s, t)\}$$

- 若结构元素 b 为非平坦， (x, y) 及其邻域像素值与结构元素 b 的值对应相减，再统计排序取其最小值作为腐蚀运算输出。

灰度图像的腐蚀

- **灰度腐蚀结果：**

- 对灰度图像的腐蚀，将导致图像整体变暗，
- 较亮的纹理结构将收缩变小、甚至消失，暗的纹理结构将会扩大，程度取决于结构元素 b 的尺寸大小。

- **灰度图像的腐蚀运算效果，与“第3章 空域滤波”介绍的最小值滤波器有相似之处。**



(1) 原图像



(2) 用 5×5 椭圆形结构元素腐蚀



(3) 用 5×5 椭圆形结构元素膨胀

灰度图像的膨胀

- 令 f 表示一幅灰度图像、 b 表示平坦型结构元素，用 b 对图像 f 膨胀，记做 $f \oplus b$ ，定义为：

$$[f \oplus b](x, y) = \max_{(s,t) \in b} \{ f(x-s, y-t) \} , \quad \hat{b} = b(-x, -y)$$

- 运算机理：先对结构元素 b 进行反射得到 \hat{b} ，然后把 \hat{b} 的原点平移到像素 (x, y) 处，该像素被膨胀后的灰度值，为图像 f 与 \hat{b} 重合区域内像素值的最大值。
- 用非平坦的结构元素 b_N 对输入图像 f 进行膨胀的定义：

$$[f \oplus b_N](x, y) = \max_{(s,t) \in b_N} \{ f(x-s, y-t) + b_N(s, t) \}$$

灰度图像的膨胀

• 灰度膨胀结果：

- 对灰度图像的膨胀，将导致图像整体变亮，较暗的纹理结构将收缩变小、甚至消失，亮的纹理结构将会扩大，程度取决于结构元素 b 的尺寸大小。
- 对灰度图像的膨胀效果，与“第3章 空域滤波”介绍的最大值滤波器有相似之处。



(1)原图像



(2)用 5×5 椭圆形结构元素腐蚀



(3)5 \times 5椭圆形结构元素膨胀

示例：灰度图像的腐蚀与膨胀



(1)原图像



(2)用 5×5 椭圆形结构元素腐蚀



(3) 5×5 椭圆形结构元素膨胀

```
#OpenCV: 灰度图像的腐蚀/膨胀
img = cv.imread('./imagedata/cameraman.tif', 0) #读入一幅灰度图像
#创建大小为 $5 \times 5$ 圆形结构元素
kernel_ellipse = cv.getStructuringElement(cv.MORPH_ELLIPSE,(5,5))
imgout1 = cv.erode(img, kernel_ellipse) #灰度腐蚀
imgout2 = cv.dilate(img, kernel_ellipse) #灰度膨胀
#显示结果（略）
```

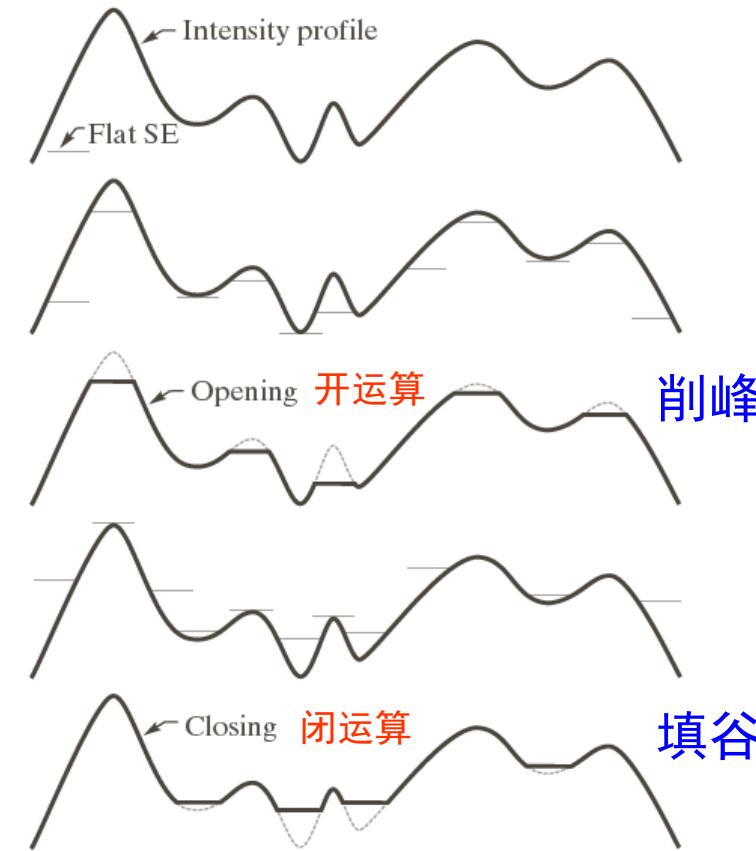
灰度图像的开运算和闭运算

- **开运算**: 用结构元素 b 对灰度图像 f 先腐蚀再膨胀的过程。

$$f \circ b = (f \ominus b) \oplus b$$

- **闭运算**: 用结构元素 b 对灰度图像 f 先膨胀再腐蚀的过程。

$$f \bullet b = (f \oplus b) \ominus b$$



灰度图像的开运算和闭运算

- **开运算可以消除与结构元素相比尺寸较小的亮细节，而保持图像整体灰度值和大的亮区域基本不受影响。**
 - ◆ 第一步的腐蚀去除了小的亮细节并同时减弱了图像亮度；
 - ◆ 第二步的膨胀增加了图像亮度，但又不重新引入前面去除的细节。
- **闭运算消除与结构元素相比尺寸较小的暗细节，而保持图像整体灰度值和大的暗区域基本不受影响。**
 - ◆ 第一步的膨胀去除了小的暗细节并同时增强了图像亮度；
 - ◆ 第二步的腐蚀减弱了图像亮度但又不重新引入前面去除的细节。

示例：灰度图像的开运算和闭运算



```
#OpenCV: 灰度图像的开运算/闭运算示例
img = cv.imread('./imagedata/cameraman.tif', 0) #读入一幅灰度图像
kernel_square = cv.getStructuringElement(cv.MORPH_RECT, (3,3)) #创建3×3方形结构元素
img_open = cv.morphologyEx(img, cv.MORPH_OPEN, kernel_square) #灰度开运算
img_close = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel_square) #灰度闭运算
#对图像先开运算再闭运算
img_oc = cv.morphologyEx(img, cv.MORPH_OPEN, kernel_square)
img_oc = cv.morphologyEx(img_oc, cv.MORPH_CLOSE, kernel_square)
```

灰度图像形态学处理应用

- 顶帽变换 (Top-hat 变换) :

$$T_{hat} = f - (f \circ b)$$

开运算：削峰

- 底帽变换 (Bottom-hat 变换) :

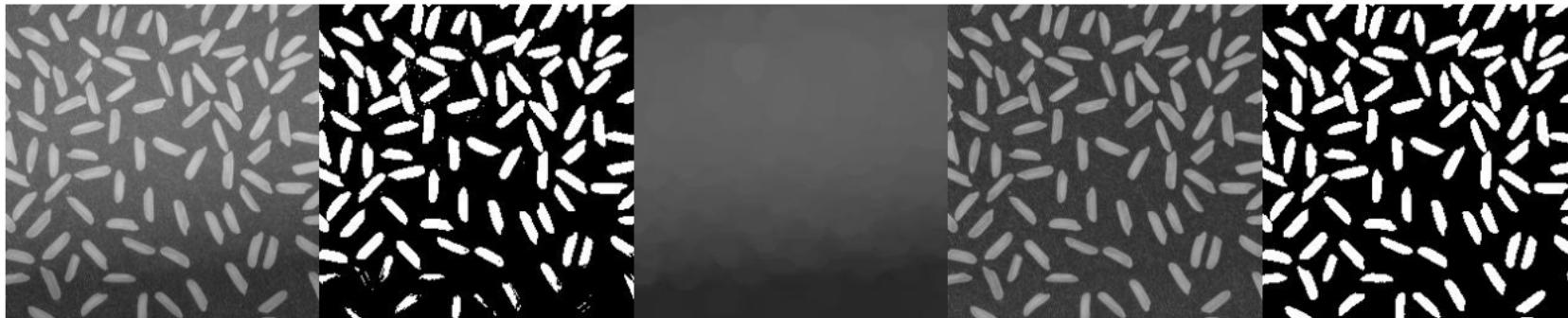
$$B_{hat} = (f \bullet b) - f$$

闭运算：填谷

- 主要应用：

- 用一个结构元素通过开运算或闭运算，从一幅图像中删除物体，得到所谓的背景图像，然后从原图像中减去背景图像，从而得到一幅仅保留期望物体的图像。
- 顶帽变换用于提取暗背景上的亮物体；
- 底帽变换用于提取亮背景上的暗物体。
- 如：校正不均匀光照的影响

示例：顶帽变换校正图像不均匀光影响



(1)米粒图像 (2)原图 Otsu 阈值分割 (3)开运算提取的背景 (4)顶帽变换结果 (5)对变换后图像阈值分割

```
#OpenCV: 采用顶帽变换校正图像不均匀光照对阈值分割的影响
img = cv.imread('./imagedata/rice.png', 0) #读取一幅灰度图像
#采用Otsu方法对灰度图像进行阈值分割得到二值图像
th, img_bw = cv.threshold(img, 0, 255, cv.THRESH_OTSU|cv.THRESH_BINARY)
#创建大小为19×19椭圆形结构元素
kernel_ellipse = cv.getStructuringElement(cv.MORPH_ELLIPSE,(19,19))
img_open = cv.morphologyEx(img, cv.MORPH_OPEN, kernel_ellipse) #灰度开运算
imgres1 = cv.subtract(img, img_open) #得到顶帽变换结果
#直接进行顶帽变换
imgres2 = cv.morphologyEx(img, cv.MORPH_TOPHAT, kernel_ellipse)
#对背景光照校正后的图像进行阈值分割
th, img_bw2 = cv.threshold(imgres2, 0, 255, cv.THRESH_OTSU|cv.THRESH_BINARY)
```

示例：利用顶帽变换去除灰度图像中的小目标区域

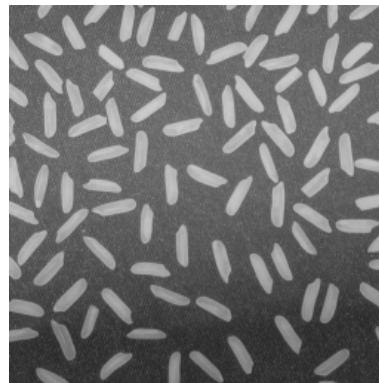


```
#OpenCV: 利用顶帽变换去除灰度图像中的小目标区域
img = cv.imread('./imagedata/hubble_deep_field.jpg', 0) #以灰度方式读取一幅彩色图像
#创建大小为9×9的椭圆形结构元素
kernel_ellipse = cv.getStructuringElement(cv.MORPH_ELLIPSE,(9,9))
img_tophat = cv.morphologyEx(img, cv.MORPH_TOPHAT, kernel_ellipse) #顶帽变换
imgout = cv.subtract(img, img_tophat) #原图像减去顶帽变换结果
```

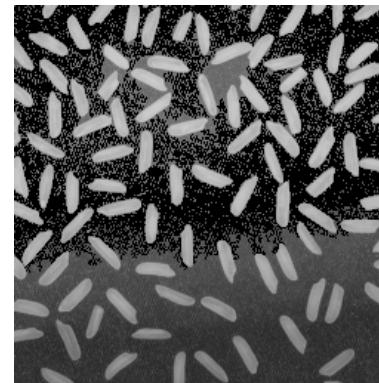
漫水填充

- 漫水填充 (Flood fill) 是一种用指定的灰度值或颜色填充连通区域，通过设置可连通像素的上、下限以及连通方式来达到不同的填充效果的方法。
- 漫水填充经常被用来标记或分离图像的一部分以便对其进行进一步处理或分析，也可以用来从输入图像获取掩膜区域。掩膜会加速处理过程，或只处理掩膜指定的像素点，操作的结果总是某个连续的区域。
- 漫水填充方法类似于图像处理软件中的“油漆桶”工具，用指定颜色填充一个密闭区域。
 - OpenCV函数cv.floodFill
 - Scikit-image函数morphology.flood_fill

示例：图像的漫水填充

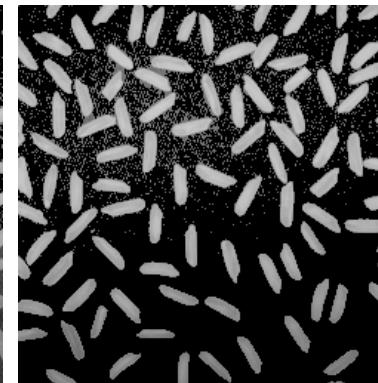


(1) 原灰度图像



(2) 漫水填充

seed_point=(1,1)
灰度值上下限容差tolerance=10



(3) 更换种子点及容差漫水填充

seed_point=(250,250)
灰度值上下限容差tolerance=60



(4) 原彩色图像



(5) 漫水填充

seedPoint=(0,0),
各颜色分量下限值容差loDiff=(20,20,50)
上限值容差upDiff=(20,20,50)



(6) 更换颜色分量容差漫水填充

颜色分量下限值容差loDiff=(50,50,150)
上限值容差upDiff=(50,50,150)

示例：图像的漫水填充

```
#漫水填充示例，类似于图像处理软件中的“油漆桶”工具
img_g = io.imread('./imagedata/rice.png')      #读取一幅灰度图像
img_c = io.imread('.\imagedata\Bridewedding.jpeg') #读入一幅彩色图像
#Scikit-image：对灰度图像的背景区域进行漫水填充
img_gfilled1 = morphology.flood_fill(img_g,seed_point=(1,1), new_value=0,tolerance=10)
#采用不同的种子点及灰度值容差
img_gfilled2 = morphology.flood_fill(img_g,seed_point=(250,250), new_value=0,tolerance=60)
#OpenCV：将彩色图像的蓝色背景区域漫水填充为白色
rows, cols = img_c.shape[:2] #获取图像的尺寸
mask = np.zeros([rows+2, cols+2], np.uint8) #创建mask
img_cfilled1 = img_c.copy() #复制图像数据
cv.floodFill(img_cfilled1, mask, (0,0), (255,255,255), (20,20,50), (20,20,50),cv.FLOODFILL_FIXED_RANGE)
#采用另一组颜色分量值容差
mask = np.zeros([rows+2, cols+2], np.uint8) #创建mask
img_cfilled2 = img_c.copy() #复制图像数据
cv.floodFill(img_cfilled2, mask, (0,0), (255,255,255),(50,50,150),(50,50,150), cv.FLOODFILL_FIXED_RANGE)
```

Q&A