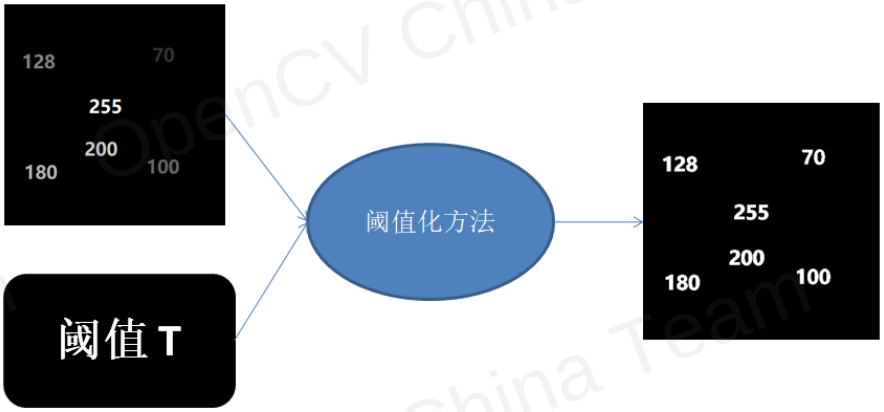
# 第 4 章 二值图像的形态学操作

## 4.1 阈值

**什么是阈值**

最简单的阈值定义：假设阈值 T，对于小于 T 的像素值用 0 替代，对于大于 T 的像素值用最大值(max)替代，得到输出结果称为阈值化图像，对灰度图像完成的阈值化操作常常被称为灰度图像二值化，简称图像二值化。




4. 1-1

从上图可以看出阈值操作涉及到两个很重要的环节，一个是阈值化方法，另外一个是阈值 T 选择(阈值查找算法)。上图我们假设阈值 T = 0 所以得到如下：

$$dst(x,y)=\begin{cases}0\\ \max\ if\ src(x,y)>T\end{cases}\quad src(x,y)\text{表示原像素值，T为阈值}$$

**阈值化方法**

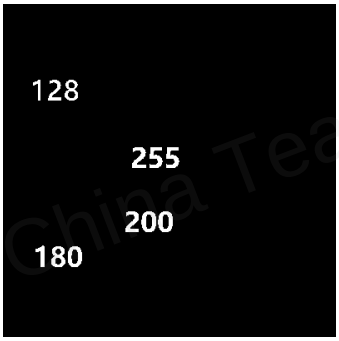
常见的阈值化方法有如下几种

1.二值化

$$dst(x,y)=\begin{cases}\max\ if\ src(x,y)>T\\ 0\end{cases}$$

如果像素值大于阈值 T，则用最大值(max)替换，否则用 0 替换。表示如下(设 T=127，下同)：


原图(4.1-2)



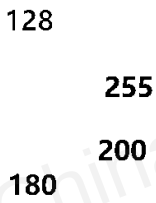

二值化 T=127(4.1-3)

2.二值化反

$$dst(x,y)=\begin{cases}0 \ if \ src(x,y)>T \\ \max\end{cases}$$

如果像素值大于阈值 T，则用 0 替换，否则用最大值替换。表示如下：


原图(4.1-4)




二值化反(4.1-5)

3.阈值截断

$$dst(x,y)=\begin{cases}T, \ if \ src(x,y)>T \\ src(x,y)\end{cases}$$

当像素值大于阈值 T 时候，用阈值 T 替代，否则像素值保留原值，这种方式称为阈值截断，表示如下：


原图(4.1-6)


截断(4.1-7)

4.阈值取零

$$dst(x,y)=\begin{cases}src(x,y), \ if \ src(x,y)>T \\ 0\end{cases}$$

当然原图像素值大于阈值 T 时候，原值保留输出，否则用 0 替换，图示如下：


原图(4.1-8)

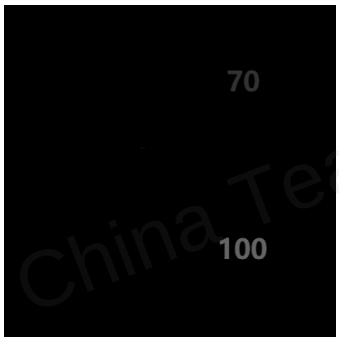

阈值取零(4.1-9)

5.阈值取零反

$$dst(x,y)=\begin{cases}0, & if\ src(x,y)>T\\ src(x,y)\end{cases}$$

如果原像素值大于阈值 T，则取 0，否则保留原像素值不变，图示如下：

原图(4.1-10)



阈值取零反(4.1-11)

**简单阈值**

OpenCV 中阈值化方法函数为

```
double cv::threshold(
    InputArray src,
    OutputArray dst,
    double        thresh,
    double        maxval,
    int  type
)
```

- src 表示输入图像，类型支持 8U 与 32F
- dst 表示输出图像，跟输入类型一致
- thresh 表示阈值，当 type 只为上述五种阈值化类型时候，阈值才起作用
- maxval 表示最大值
- type 表示阈值化类型,主要有如下五种(分别对应上面提到五种阈值化方法):
    1) THRESH_BINARY
    2) THRESH_BINARY_INV
    3) THRESH_TRUNC
    4) THRESH_TOZERO
    5) THRESH_TOZERO_INV

返回值 double 类型表示返回的阈值，只有当阈值为自动计算时候返回阈值才有意义，其它情况下，返回值等于输入参数 thresh。

当设置手动阈值为 127 时，二值化输出的代码如下：

```
double ret = threshold(src, binary, 127, 255, THRESH_BINARY);
imshow("binary", binary);
```

**自适应阈值**

OpenCV 中手动阈值通过 threshold 函数实现二值化方式以外，还支持自适应阈值二值化，其相关函数分别如下：

```
void cv::adaptiveThreshold(
    InputArray    src,
    OutputArray dst,
    double        maxValue,
    int  adaptiveMethod,
    int  thresholdType,
    int  blockSize,
    double        C
)
```

src 输入图像，单通道 8 位的图像
dst 输出图像，与输入图像类型相同
maxvalue 最大值
adaptiveMethod 自适应方法，支持两种自适应阈值方法
    cv::ADAPTIVE_THRESH_MEAN_C = 0,
    cv::ADAPTIVE_THRESH_GAUSSIAN_C = 1
thresholdType 阈值化方法，为前面的五中阈值化方法之一
blockSize 自适应阈值时候的像素块大小
C 表示常量

自适应阈值不需要手动设置阈值，其基本原理是先对图像做模糊，跟模糊方式不同可以分为盒子模糊与高斯模糊，然后用原图跟模糊之后图像做减法之后加上常量 C 最终输出，表示如下：

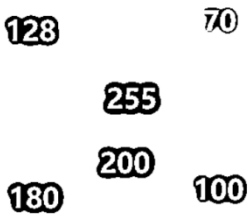
$$D = S - B + C = \begin{cases} 255 \text{ 如果大于 } 0 \\ 0 \end{cases}$$

调用代码如下：

```
adaptiveThreshold(gray, binary, 255,
                  ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY,
                  25, 10);
imshow("ada-binary", binary);
```

gray 表示输入,binary 是输出， ADAPTIVE_THRESH_GAUSSIAN_C 表示自适应方法，THRESH_BINARY 表示二值化，25 表示块大小，10 表示常量 C。执行结果如下：


原图(4.1-12)

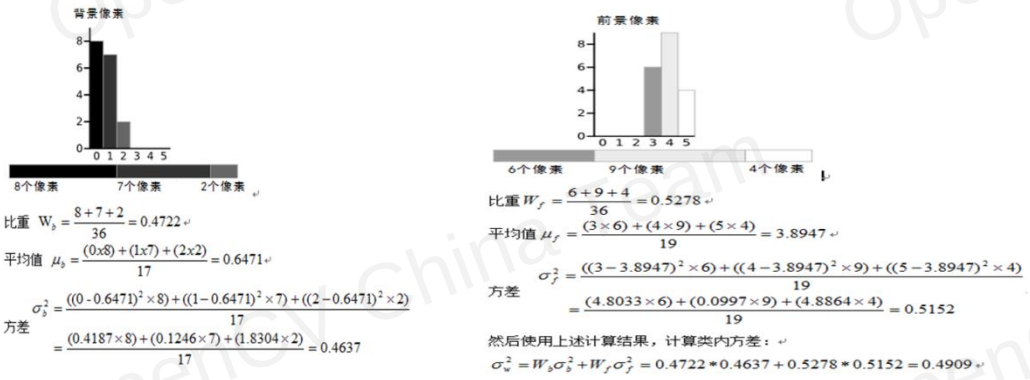



自适应阈值输出(4.1-13)

**OTSU 与 Triangle 阈值法**

OTSU 跟 Triangle 都是基于直方图分布实现的全局阈值计算的方法，简单点说，它们会自动查找计算得到阈值 T，不再需要手动指定阈值。

其中 OTSU 的是通过计算类间最大方差来确定分割阈值的阈值选择算法，OTSU 算法对直方图有两个峰，中间有明显波谷的直方图对应图像二值化效果比较好，而对于只有一个单峰的直方图对应的图像分割效果没有双峰的好。Triangle 三角法基于直方图的单峰与斜边的最大距离确定阈值，这种方法对直方图单峰分布的图像效果比较好。下面是 OTSU 的算法说明(4.1-14)：




(图片来自网络)

上图假设 T={0,1,2,3,4,5}时候，分别计算均值与方差，最终求得两个部分权重方差之和，最终比较这些方差之和，最小方差对应的阈值即为最终找到的阈值 T。

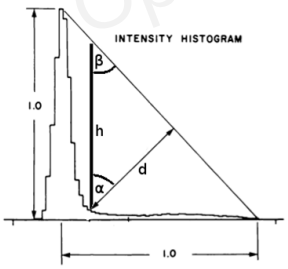
OpenCV 中 OTSU 算法使用只需要在
threshold 函数的 type 类型声明 THRESH_OTSU 即可，代码演示如下

```
01 double ret = threshold(gray, binary, 127, 255, THRESH_BINARY
02                | THRESH_OTSU);
03 imshow("binary", binary);
```

运行结果如下：


原图(4.1-15)　　OTSU 二值化(4.1-16)

三角法阈值寻找的方法如下所示(4.1-17)：




$$h^2 = d^2 + d^2 <=> d = sqrt(\frac{h^2}{2})$$

$$d = \sin(0.7854) * h$$

(G. W. Zack et al, Automatic Measurement of Sister Chromatic Exchange Frequency)

寻找 d 最大所对应的 h 对应的直方图的级别即为阈值 T。如果直方图的最高峰不在左侧的时候则需要首先对数据取反得到最高峰在左侧，这样得到阈值为 255-T。

threshold 函数的 type 类型声明 THRESH_TRIANGLE 即可

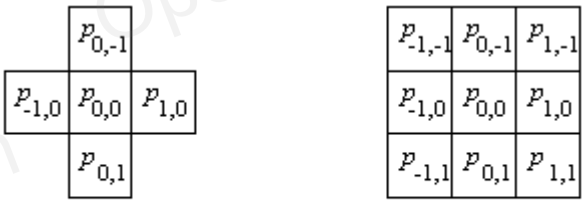
```
01 double ret = threshold(gray, binary, 127, 255, THRESH_BINARY
02             | THRESH_TRIANGLE);
03 imshow("binary", binary);
```

运行结果如下：


原图(4.1-18)　　三角法结果(4.1-19)

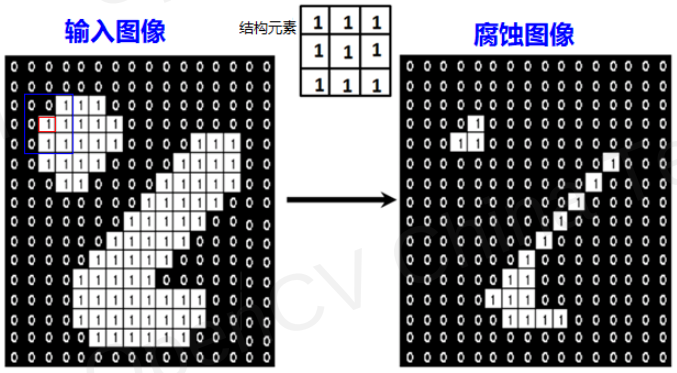
## 4.2 腐蚀与膨胀

腐蚀与膨胀是二值图像形态学处理中最基本的两个操作，形态学的操作都必须有一个结构元素。结构元素不是一个像素，而且一个几何形状的像素块，常见的结构元素为矩形、十字交叉、圆形等，图示如下(4.2-1)：




(a)　　(b)

上图 P(0,0)表示结构元素的中心像素。腐蚀的定义为，用结构元素重叠部分的像素最小值替换中心像素作为输出，对二值图像就是用 0(黑色)来替代 255(白色)，以 3x3 结构元素 b 为示例如下：

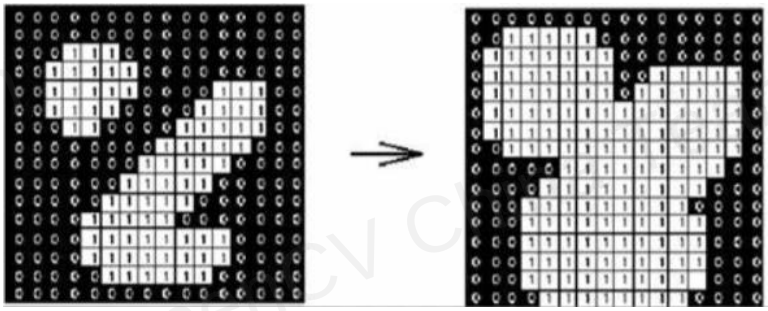



(4.2-2)

(图片来自网络)

上图中左侧输入图像的蓝色矩形表示结构元素所在在区域，红色是结构元素中心位置，结构元素所在区域最小值为 0(黑色)所以中心像素值 1(白色)会被替换为 0(黑色)。就这样结构元素从左到右，从上到下在输入图像上完成每个像素的操作，得到最终的腐蚀图像。

膨胀操作与腐蚀操作几乎相似，唯一不同的是用最大值替换结构元素的中心像素，对二值图像来说即为 255(白色)替换 0(黑色)。


图(4.2-3)

(图片来自网络)

从上面可以看出，腐蚀会让白色的区域变小，膨胀则会让白色区域变大！

OpenCV 中腐蚀函数为：

```
void cv::erode(
    InputArray          src,
    OutputArray dst,
    InputArray          kernel,
    Point       anchor = Point(-1,-1),
    int         iterations = 1,
    int         borderType = BORDER_CONSTANT,
    const Scalar & borderValue = morphologyDefaultBorderValue()
)
```

膨胀函数为：

```
void cv::dilate(
    InputArray  src,
    OutputArray dst,
    InputArray          kernel,
    Point       anchor = Point(-1,-1),
    int         iterations = 1,
    int         borderType = BORDER_CONSTANT,
    const Scalar & borderValue = morphologyDefaultBorderValue()
)
```

参数解释如下：

src 表示输入图像，支持 CV_8U, CV_16U, CV_16S, CV_32F or CV_64F

dst 表示输出，类型必须与输入图像一致

kernel 表示结构元素

anchor 表示中结构元素的输出位置，默认 Point(-1,-1)表示中心位置

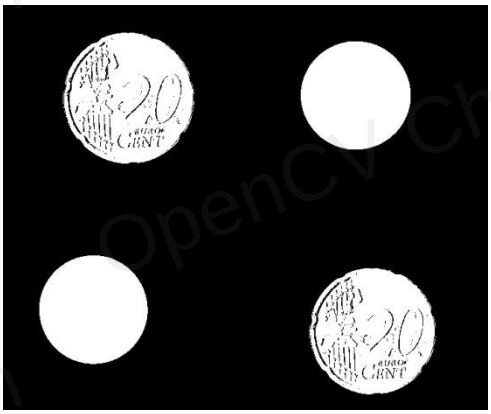
iterations 表示应用多少次数

borderType 表示边缘像素填充方式

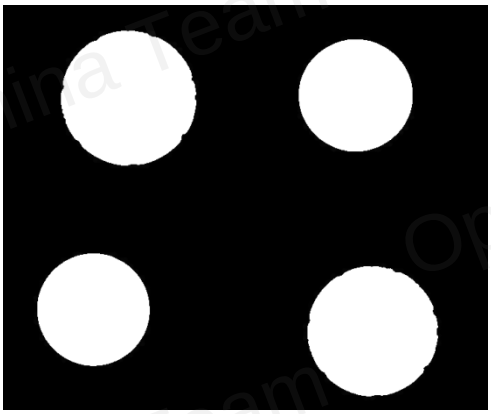
borderValue 常量填充时候，填充值

代码演示-膨胀

```
Mat image = imread("D:/dilation_example.jpg");
int kSize = 7;
Mat kernel1 = getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(kSize, kSize));
imshow("image", image);
Mat imageDilated;
dilate(image, imageDilated, kernel1);
imshow("dilate", imageDilated);
```



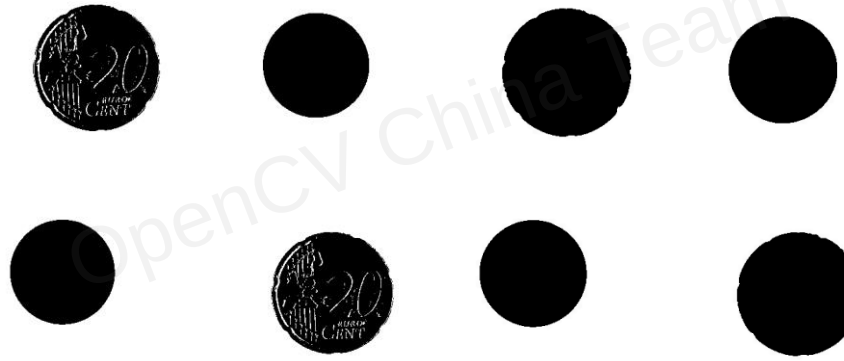
输入图像(4.2-4)



膨胀图像(4.2-5)

代码演示-腐蚀

```
Mat image = imread("D:/erosion_example.jpg");
int kSize = 7;
Mat kernel1 = getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(kSize, kSize));
imshow("image", image);
Mat imageEroded;
erode(image, imageEroded, kernel1);
imshow("erode", imageEroded);
```



原图(4.2-6)

腐蚀图像(4.2-7)

**注意：**OpenCV 中所有的二值图像分析都是黑色(0)作为背景、白色(255)作为前景对象为提前下运行各种二值分析算法。

## 4.3 开运算与闭运算

开/闭运算(操作)是通过腐蚀与膨胀组合得到的形态学运算，开运算对输入图像执行先腐蚀后膨胀操作，闭运算是对图像执行先膨胀后腐蚀操作。

开运算 = 腐蚀 + 膨胀

闭运算 = 膨胀 + 腐蚀

开运算会清除二值图像的小的白色噪声像素块，闭运算会填充二值图像小的黑色空洞区域。

OpenCV 中的开闭运算

OpenCV 中开闭运算一起共享一个 API 函数，通过其中的一个参数设置来决定是开运算还是闭运算。

```
void cv::morphologyEx(
    InputArray  src,
    OutputArray dst,
    int         op,
    InputArray          kernel,
    Point       anchor = Point(-1,-1),
    int         iterations = 1,
    int         borderType = BORDER_CONSTANT,
    const Scalar & borderValue = morphologyDefaultBorderValue()
)
```

src 表示输入图像，支持 CV_8U, CV_16U, CV_16S, CV_32F or CV_64F

dst 表示输出，类型必须与输入图像一致

op 表示是什么运算，当为 MORPH_OPEN 表示为开运算，当为 MORPH_CLOSE 表示闭运算

kernel 表示结构元素

anchor 表示中结构元素的输出位置，默认 Point(-1,-1)表示中心位置

iterations 表示应用多少次数

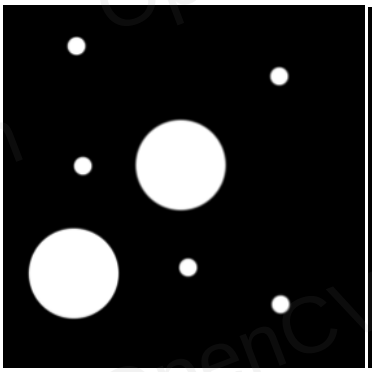
borderType 表示边缘像素填充方式

borderValue 常量填充时候，填充值

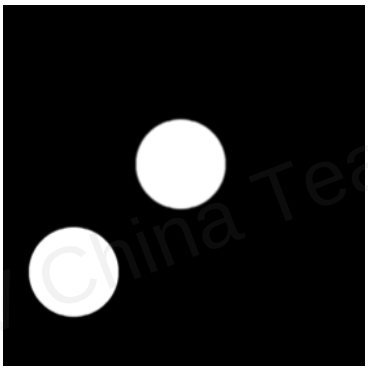
**代码演示-开运算**

方法一：先腐蚀后膨胀

```
Mat image = imread("D:/opening.png");
// Specify Kernel Size
int kernelSize = 10;
// Create the kernel
Mat element = getStructuringElement(MORPH_ELLIPSE, Size(2 * kernelSize + 1, 2 * kernelSize + 1),
                                    Point(kernelSize, kernelSize));
Mat imEroded;
// Perform erosion
erode(image, imEroded, element, Point(-1, -1), 1);
Mat imOpen;
// Perform dilation
dilate(imEroded, imOpen, element, Point(-1, -1), 1);
imshow("m1", imOpen);
imwrite("D:/m1.png", imOpen);
```



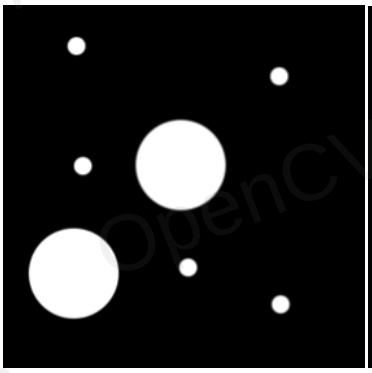
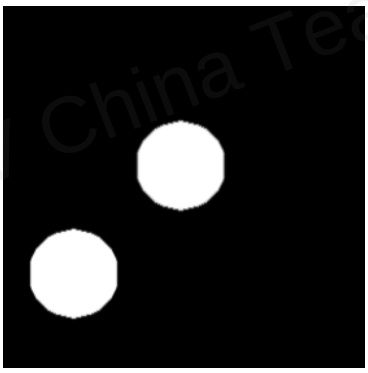
原图(4.3-1)



开运算(4.3-2)

方法二：直接使用 morphologyEx 函数

```
Mat imageMorphOpened;
morphologyEx(image, imageMorphOpened, MORPH_OPEN, element, Point(-1, -1), 3);
```

原图(4.3-3)　　开运算(4.3-4)

**代码演示-闭运算**

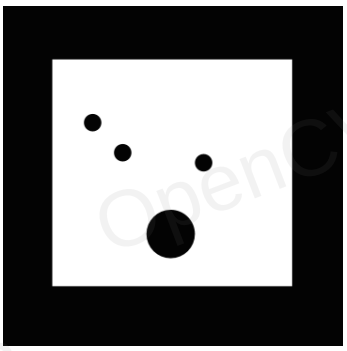
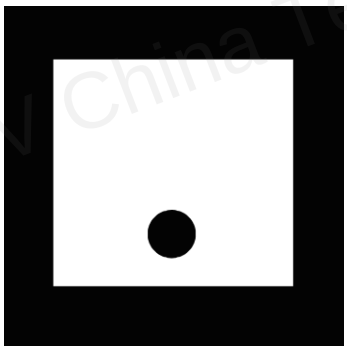
方法一：先膨胀再腐蚀

```
Mat image = imread("D:/closing.png");
imshow("input", image);
// Specify kernel size
int kernelSize = 10;
// Create kernel
Mat element = getStructuringElement(MORPH_ELLIPSE, Size(2 * kernelSize + 1, 2 * kernelSize + 1),
                                    Point(kernelSize, kernelSize));

Mat imDilated;
// Perform Dilation
dilate(image, imDilated, element);
Mat imClose;
// Perform erosion
erode(imDilated, imClose, element);
```

方法二：

```
Mat image = imread("D:/closing.png");
imshow("input", image);
// Specify kernel size
int kernelSize = 10;
// Create kernel
Mat element = getStructuringElement(MORPH_ELLIPSE, Size(2 * kernelSize + 1, 2 * kernelSize + 1),
                                    Point(kernelSize, kernelSize));
Mat imageMorphclosed;
// Create a structuring element
morphologyEx(image, imageMorphclosed, MORPH_CLOSE, element);
```

运行结果如下：

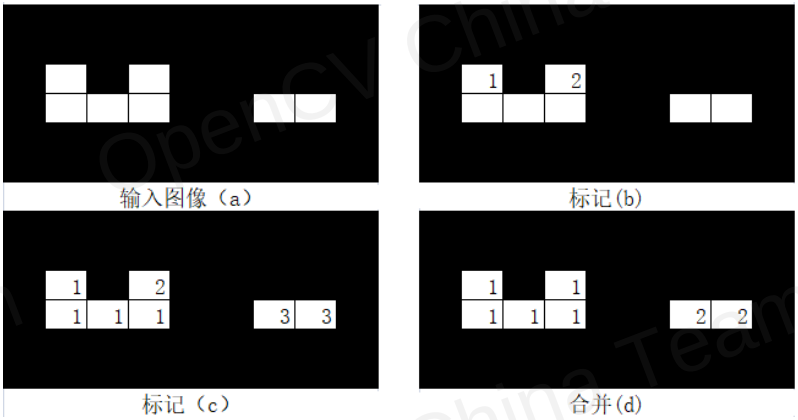



原图(4.3-5)　　闭运行(4.3-6)

## 4.4 连通区域分析

连通区域分析算法通过扫描二值图像的每个像素点，对像素值相同的而且相互连通像素点标记为相同的标签，最终标签相同/等价的像素点属于同一个连通区域。扫描的方式可以是从

上到下，从左到右，基于每个像素单位，对于一幅有 N 个像素的图像来说，最大连通区域个数为 N/2。最常见的连通区域扫描算法是两步法，图示如下(4.4-1)：

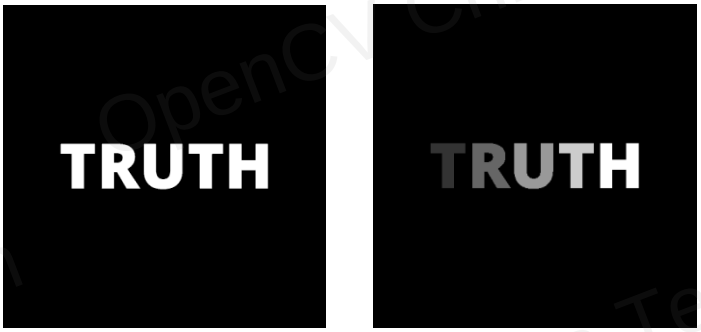



(图片来自网络)

图中(a)表示输入图像、(b),(c)表示两步法第一步标记过程，(d)表示两步法中的第二步，等价合并过程。标记过程是从左向右从上到下扫描每个像素点，遇到前景像素点(像素值为 255)，首先检查是否有邻域被标记像素点，如果有一个或者多个邻域像素点被标记，则选择最小的标记作为当前前景像素点的标记。合并过程通过检查连通等价，对连通区域替换最小等价标记，最终得到输出。

OpenCV 中的连通区域分析(Connected Component Analysis-CCA)

连通区域分析完成二值图像中的对象块标记，因此可以实现对二值图像中的对象计数，下面通过一个例子来说明。




输入图像(4.4-2)　　标签输出(4.4-3)

上图左侧(4.4-2)是一张二值图像，有五个 BLOB 对象，分别是 T、R、U、T、H。连通区域分析会对每个 BLOB 对象生成标签作为标识，背景标记为 0，T 标签为 1、R 标签为 2，以此类推。代码实现如下：

```
Mat image = imread("D:/truth.png", IMREAD_GRAYSCALE);
imshow("input", image);

// Threshold Image
Mat imThresh;
threshold(image, imThresh, 127, 255, THRESH_BINARY);

// Find connected components
Mat imLabels;
int nComponents = connectedComponents(imThresh, imLabels);
Mat imLabelsCopy = imLabels.clone();

// First let's find the min and max values in imLabels
Point minLoc, maxLoc;
double minVal, maxVal;

// The following line finds the min and max pixel values
// and their locations in an image.
minMaxLoc(imLabels, &minVal, &maxVal, &minLoc, &maxLoc);

// Normalize the image so the min value is 0 and max value is 255.
imLabels = 255 * (imLabels - minVal) / (maxVal - minVal);

// Convert image to 8-bits
imLabels.convertTo(imLabels, CV_8U);
```

运行代码，生成的标记 imLabels 显示在 4.4-3。

显示彩色输出

只是灰度输出标签图像，灰度图像的区别不是很明显，可以通过 OpenCV 支持的颜色匹配把灰度标签图像转为彩色图像输出，让差异显而易见。首先我们需要把像素归一化到 0~255 之间，为了实现像素归一化，我们需要得到图像最大与最小值，然后对图像的每个像素值减去最小值，再除以最大与最小值之差，图像像素值范围被缩减到 0~1 区间，然后再乘以 255，就得到了 0~255 之间的输出图像，再对输出图像进行颜色匹配就会获得最终的彩色图像输出。

什么是颜色匹配

假设我们想在地图上显示不同地区的温度，可能会在地图上显示不同灰度级别，灰度值低表示寒冷区域，灰度值越高看上去越亮的区域表示越热的地区，这种表示方法有两个弊端：

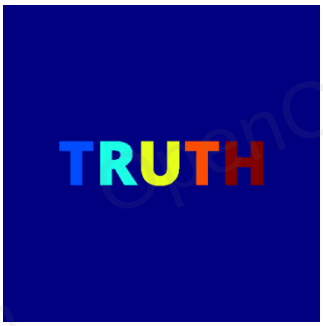
1. 人脸的视觉系统对微小灰度值差异并不敏感，无法区分两个地区微小温度差异，但是我们的视觉系统对彩色图像有足够的敏感度
2. 彩色表示更有意义，热的地区可以用红色表示，冷的地区可以用蓝色表示，区分明显。

温度表示只是一个例子，我们经常用这种伪彩色的方法来表示高度、密度、压力等单个数据变换。

OpenCV 中包含 12 可以把灰度图像转换为伪彩色的颜色匹配方法，其调用的函数为 applyColorMap，这里以 COLORMAP_JET 颜色匹配方法为例，代码实现如下：

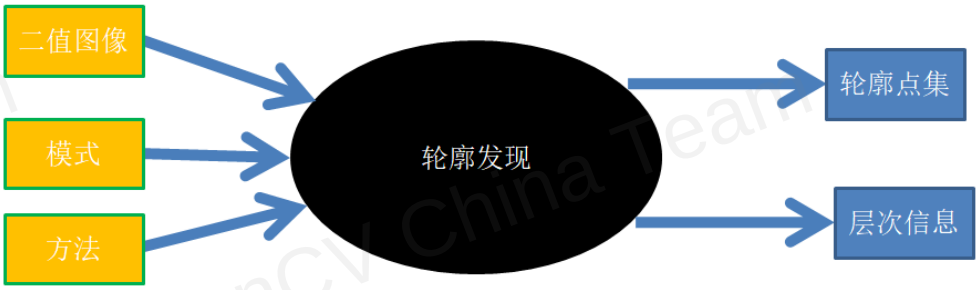
```
// Make a copy of the image
imLabels = imLabelsCopy.clone();

double minValue, maxValue;
minMaxLoc(imLabels, &minValue, &maxValue, &minLoc, &maxLoc);

// Normalize the image so the min value is 0 and max value is 255.
imLabels = 255 * (imLabels - minValue) / (maxValue - minValue);

// Convert image to 8-bits
imLabels.convertTo(imLabels, CV_8U);

// Apply a color map
Mat imColorMap;
applyColorMap(imLabels, imColorMap, COLORMAP_JET);
```

最终生成的彩色图像 imColorMap 显示如下(4.4-4)：




## 4.5 轮廓

二值图像的沿着边缘连续的像素点被成为轮廓，轮廓是图像基本特征之一，在图像几何分析、对象检测与识别中都非常有用。寻找轮廓就是提取每个轮廓的点集、对应的层次信息，实现结构化输出。寻找轮廓处理过程表示如下图(4.5-1)：




上图左侧是输入数据与方法参数，中间是黑色表示寻找轮廓处理，右侧是输出信息。

输入信息包括：

a. 二值图像，单通道字节类型的图像

b. 模式，表示对二值图像进行轮廓发现时候所输出的层次信息，最常用的模式有两种 RETR_EXTERNAL 与 RETR_TREE，前者表示只发现最外层轮廓，后者表示发现所有轮廓并树形结构表示。图示如下(4.5-2)：

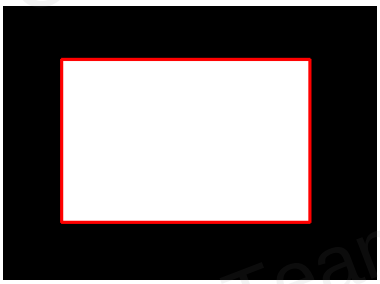

RETR_EXTERNAL


RETR_TREE

上图现实，RETR_EXTERNAL 模式下左侧是只会寻找最外层的轮廓，右侧会发现所有轮廓

c. 方法，方法会影响输出的轮廓点集多少，最常用的两个轮廓点集提取的方法分别是 CHAIN_APPROX_SIMPLE 与 CHAIN_APPROX_NONE，这两个方法提取轮廓点的区别如下图(4.5-3)：

CHAIN_APPROX_SIMPLE



CHAIN_APPROX_NONE

从上图可以看出，选择左侧 CHAIN_APPROX_SIMPLE 方法时生成四个点表示轮廓，选择右侧 CHAIN_APPROX_NONE 方法时生成矩形四个边全部点来表示轮廓。

在轮廓发现输出的层次信息中，有如下四个部分组成

| hierarchy[i][0] | hierarchy[i][1] | hierarchy[i][2] | hierarchy[i][3] |
|---|---|---|---|

hierarchy[i][0] 表示同一层下一个轮廓索引

hierarchy[i][1] 表示同一层上一个轮廓索引

hierarchy[i][2] 表示当前轮廓第一个孩子轮廓索引

hierarchy[i][3] 表示当前轮廓的父轮廓

上述关系中如果没有则返回-1 表示。

**使用 OpenCV 轮廓分析**

OpenCV 中的寻找轮廓的函数为：

```
01 void cv::findContours(
02     InputArray  image,
03     OutputArrayOfArrays contours,
04     OutputArray hierarchy,
05     int      mode,
06     int      method,
07     Point    offset = Point()
08 )
```

参数解释如下：

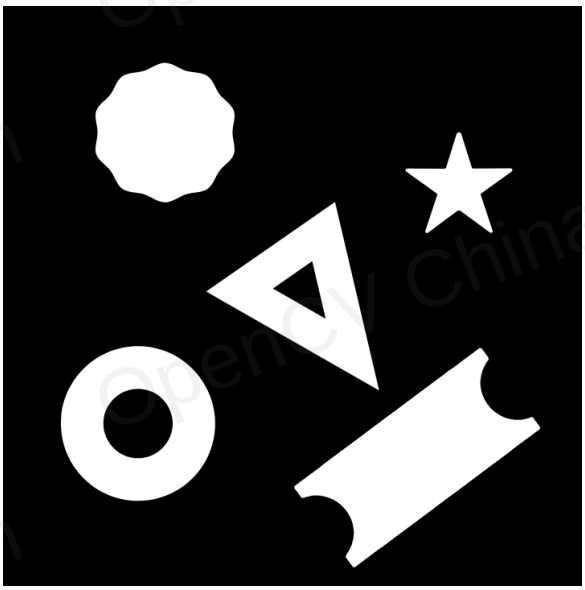
image 表示输入图像，单通道八位二值图像

contours 表示输出的所有轮廓，每个轮廓都是一个点集用 vector<Point>表示

hierarchy 表示层次信息，用来描述轮廓之间的拓扑结构关系

mode 表示轮廓返回的方式，有 RETR_EXTERNAL, RETR_LIST, RETR_CCOMP, RETR_TREE
method 表示轮廓编码方式，支持有：
CHAIN_APPROX_NONE, CHAIN_APPROX_SIMPLE, CHAIN_APPROX_TC89_L1
Offset 表示可选的每个轮廓点的偏移量，默认为 0

测试的输入图像(4.5-4)：

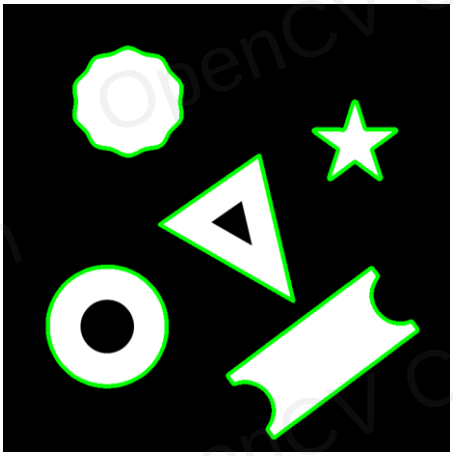


OpenCV 中寻找轮廓的代码如下：

```
Mat image = imread("D:/Contour.png");
imshow("input", image);

Mat imageGray;
cvtColor(image, imageGray, COLOR_BGR2GRAY);

// 轮廓发现
vector<vector<Point> > contours;
vector<Vec4i> hierarchy;

findContours(imageGray, contours, hierarchy, RETR_LIST, CHAIN_APPROX_SIMPLE);
cout << "Number of contours found = " << contours.size();

// 绘制轮廓
Mat drawing = image.clone();
for (size_t i = 0; i< contours.size(); i++) {
    drawContours(drawing, contours, -1, Scalar(0, 255, 0), 6);
}
imshow("findContours-demo", drawing);
```

上述代码，首先加载一张图像，然后转换为灰度图像，调用寻找轮廓函数 findContours 找到全部的轮廓，然后通过 drawContour 绘制每一个轮廓，运行结果如下图(4.5-5)：

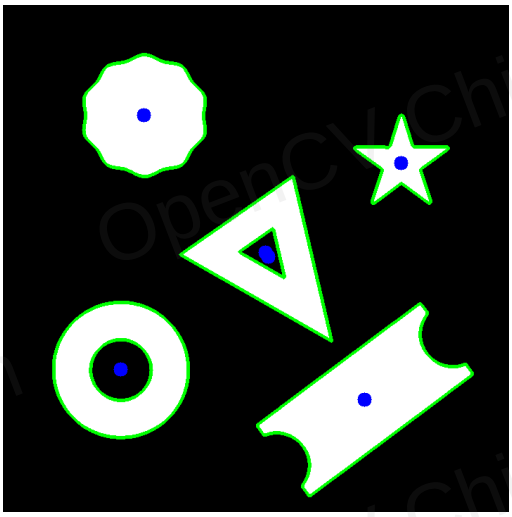
修改寻找轮廓函数的 mode 参数，把 RETR_LIST 改为 RETR_EXTERNAL，运行结果如下图(4.5-6)：



轮廓属性-中心位置

```
// Find all contours in the image
findContours(imageGray, contours, hierarchy, RETR_LIST, CHAIN_APPROX_SIMPLE);
// Draw all the contours
drawContours(image, contours, -1, Scalar(0,255,0), 3);

Moments M;
int x,y;
for (size_t i=0; i < contours.size(); i++) {
    // We will use the contour moments
    // to find the centroid
    M = moments(contours[i]);
    x = int(M.m10/double(M.m00));
    y = int(M.m01/double(M.m00));

    // Mark the center
    circle(image, Point(x,y), 10, Scalar(255,0,0), -1);
}
```

上述代码首先寻找轮廓，然后基于几何距计算得到每个轮廓的中心位置。运行显示如下图(4.5-7)：



轮廓属性-面积与周长

轮廓属性分析支持计算每个轮廓的面积与周长，代码如下：

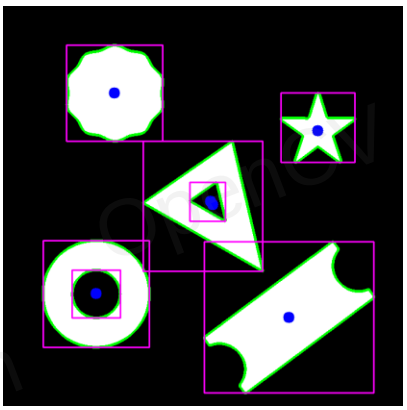
```
double area;
double perimeter;
for (size_t i=0; i < contours.size(); i++) {
    area = contourArea(contours[i]);
    perimeter = arcLength(contours[i],true);
    cout << "Contour #" << i+1 << " has area = " << area << " and perimeter = " << perimeter << endl;
}
```

轮廓属性-外接矩形

轮廓属性分析支持求得每个轮廓最大外接矩形，代码如下：

```
image = imageCopy.clone();
Rect rect;
for (size_t i=0; i < contours.size(); i++) {
    // Vertical rectangle
    rect = boundingRect(contours[i]);
    rectangle(image, rect, Scalar(255,0,255), 2);
}
```

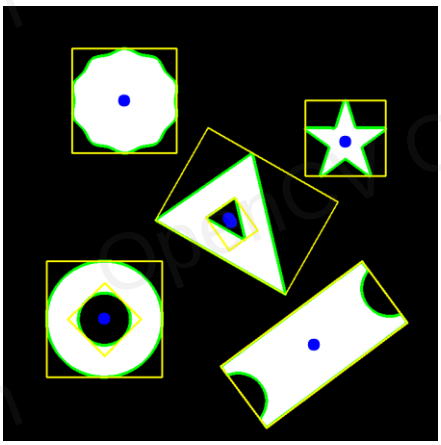
运行显示如下图(4.5-8)



计算每个轮廓的最小外接矩形，代码如下：

```
RotatedRect rotrect;
Point2f rect_points[4];
Mat boxPoints2f,boxPointsCov;

for (size_t i=0; i < contours.size(); i++) {
    // Rotated rectangle
    rotrect = minAreaRect(contours[i]);
    boxPoints(rotrect, boxPoints2f);
    boxPoints2f.assignTo(boxPointsCov,CV_32S);
    polylines(image, boxPointsCov, true, Scalar(0,255,255), 2);
}
```

代码运行结果如下图(4.5-9)

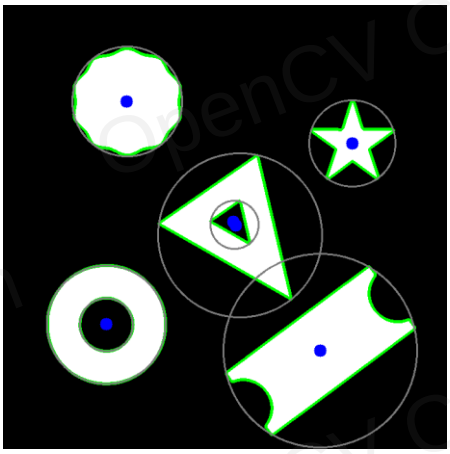


轮廓属性-拟合圆与椭圆

拟合一个外接矩形有时候不是想要的结果，更希望拟合一个圆或者椭圆。OpenCV 关于轮廓处理中也提供这些有用的函数调用，拟合最小外接圆的代码演示如下：

```
image = imageCopy.clone();
Point2f center;
float radius;
for (size_t i=0; i < contours.size(); i++) {
    // Fit a circle
    minEnclosingCircle(contours[i],center,radius);
    circle(image,center,radius, Scalar(125,125,125), 2);
}
```

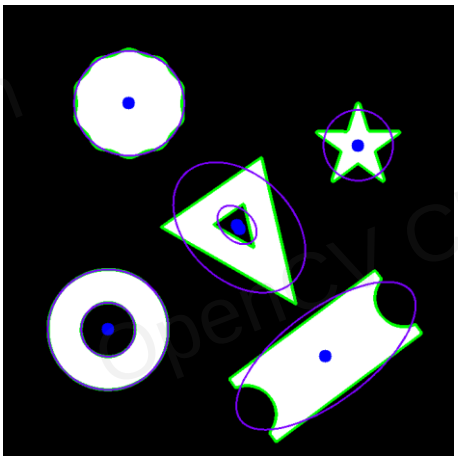
运行结果如下图(4.5-10)



拟合椭圆的代码实现如下：

```
RotatedRect rellipse;
for (size_t i=0; i < contours.size(); i++) {
    if (contours[i].size() < 5)
        continue;
    rellipse = fitEllipse(contours[i]);
    ellipse(image, rellipse, Scalar(255,0,125), 2);
}
```

特别需要注意的是拟合椭圆的时候轮廓必须要超过 5 个点才可以正确工作，所以需要先判断一下轮廓点数多少，然后在调用拟合函数。程序运行结果如下图(4.5-11)



## 4.6Blob 检测

什么是 Blob

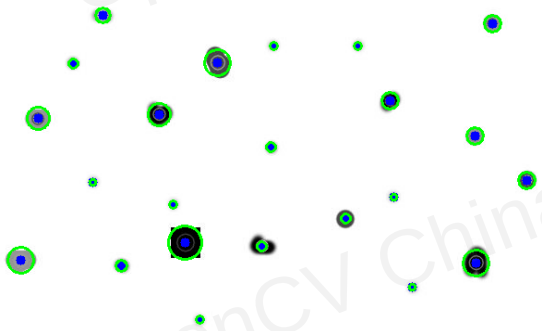
Blob 是图像中一组相互连通的像素点，它们具有一些共通的属性(比如：像素值)。上图(4.6-1)中所有深色连通区域都是 Blob。Blob 检测就是要把这些区域都找出来并标记。

Blob 检测

OpenCV 中提供了一套简单易用的功能实现 Blob 检测与根据各种不同属性进行过滤，下面就是一个简单的 Blob 检测例子，代码实现如下：

```
// Set up detector with default parameters
Ptr<SimpleBlobDetector> detector = SimpleBlobDetector::create();

std::vector<KeyPoint> keypoints;
detector->detect(img,keypoints);

int x,y;
int radius;
double diameter;
cvtColor(img, img, COLOR_GRAY2BGR);
for (int i=0; i < keypoints.size(); i++) {
    KeyPoint k = keypoints[i];
    Point keyPt;
    keyPt = k.pt;
    x=(int)keyPt.x;
    y=(int)keyPt.y;
    // Mark center in BLACK
    circle(img,Point(x,y),5,Scalar(255,0,0),-1);
    // Get radius of coin
    diameter = k.size;
    radius = (int)diameter/2.0;
    // Mark blob in GREEN
    circle(img, Point(x,y),radius,Scalar(0,255,0),2);
}
```

运行结果如下图(4.6-2):

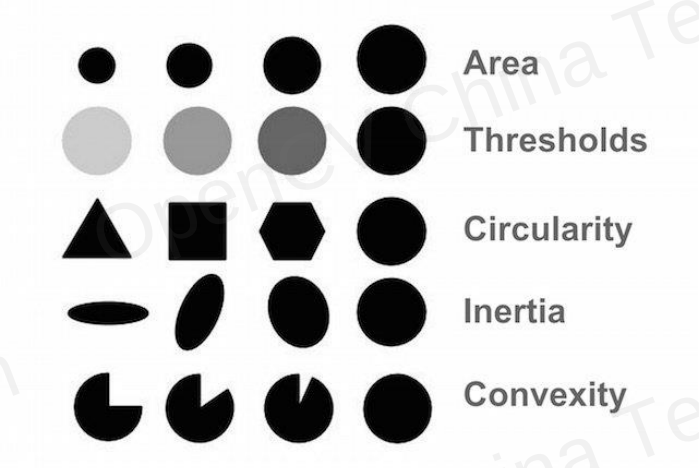


简单 Blob 检测原理

OpenCV 中的 SimpleBlobDetector 函数实现 Blob 检测基本原理就是通过一系列的参数设置完成的，主要参数意义解释如下：

1. Threshold(阈值)：通过阈值把输入图像转换为几个二值图像，在最小阈值(minThreshold)最大阈值(maxThreshold)之间设置指定的阈值间隔(thresholdstep)，第一个阈值就是 minThreshold，然后此增加 thresholdstep + minThreshold 为第二个阈值，以此类推直到最大阈值(maxthreshold)。
2. Grouping(分组)：阈值转的每个二值图像，连通的白色像素区域分组在一起，称为 Blob 区域。
3. Merging(合并)：每个 Blob 区域的中心位置可以计算得到，根据中心位置之间距离，小于 minDistBetweenBlobs 阈值的 Blob 将被合并。
4. Center & Radius Calculation 中心与半径计算：合并之后中心与半径将会重新计算

基于颜色、形状、大小的 Blob 过滤




SimpleBlobDetector 函数的参数可以实现上述各种方式过滤，实现对 Blob 对象的分类，支持的过滤分类方式有如下：

- **基于颜色：**首先设置 filterByColor = 1，设置 blobColor = 0 选择黑色区域 Blob，设置 blobColor = 255 选择白色区域 Blob
- **基于大小：**首先设置 filterByArea = 1，意思是启动根据面积大小来过滤；对两个面积参数 minArea 与 maxArea 分别设置合适大小的值实现过滤，例如：minArea=100 过滤所有像素小于 100 的 Blob。
- **基于形状：**基于形状实现 Blob 过滤，有三个不同的参数决定，下面三个为详细解释。
  **圆度：**Blob 形状多少程度上接近一个圆，例如一个正六边形比四边形有更高的圆度，启用圆度作为过滤条件，设置 filterByCircularity = 1；设置最小圆度(minCircularity) 与最大圆度(maxCircularity)为合适的值，计算公式如下：

$$Circularity = \frac{4\pi \times Area}{(perimeter)^2}$$

  圆的圆度为 1，正方形为 0.785，以此类推。
  **凸度：**评价一个 Blob 形状是不是一个凸多边形的程度，启用凸度过滤，设置 filterByConvexity = 1，然后设置 0 ≤ 最小凸度(minConvexity) ≤ 1 而且最大凸度(maxConvexity)小于等于 1。
  **惯性：**这个概念常被混淆，这里特别是指几何的短轴跟长轴比率，例如：圆等于 1、椭圆在 0~1 之间，直线为 0。启用惯性过滤设置 filterByInertia = 1，然后设置 0 ≤ 最小惯性(minInertiaRatio) ≤ 1 而且最大惯性(maxInertiaRatio)小于等于 1。

使用这些过滤条件设置的代码如下：

```cpp
// Setup SimpleBlobDetector parameters.
SimpleBlobDetector::Params params;

// Change thresholds
params.minThreshold = 10;
params.maxThreshold = 200;

// Filter by Area.
params.filterByArea = true;
params.minArea = 1500;

// Filter by Circularity
params.filterByCircularity = true;
params.minCircularity = 0.1;

// Filter by Convexity
params.filterByConvexity = true;
params.minConvexity = 0.87;

// Filter by Inertia
params.filterByInertia = true;
params.minInertiaRatio = 0.01;
detector = SimpleBlobDetector::create(params);
```

作业：硬币检测

总分值：30 分，两个部分！

主要知识点涉及：

- 图像阈值化
- 形态学操作
- Blob 检测
- 轮廓检测
- 连通区域分析

第一部分：

对上图完成硬币计数，找出每个硬币轮廓与中心位置。

第二部分：
对下图完成硬币计数跟中心位置确认，面积与周长测量。

作业提示：
首先完成图像灰度化，然后使用阈值方法实现二值化，对二值化结果进行形态学操作(开闭操作)、然后使用 SimpleBlobDetector 实现 Blob 分析或者轮廓发现分析完成硬币计数。

第一部分代码实现

```
// 加载图像
Mat img = imread("D:/CoinsA.png");
imshow("Original Image", img);

// 阈值化操作
Mat gray, binary;
cvtColor(img, gray, COLOR_BGR2GRAY);
vector<Mat> mv;
split(img, mv);
float t = threshold(mv[1], binary, 0, 255, THRESH_BINARY | THRESH_TRIANGLE);

// 形态学操作
Mat se = getStructuringElement(MORPH_RECT, Size(3, 3));
morphologyEx(binary, binary, MORPH_CLOSE, se);
morphologyEx(binary, binary, MORPH_OPEN, se, Point(-1, -1), 4);

// 轮廓发现
vector<Vec4i> hireachy;
vector<vector<Point>> contours;
findContours(binary, contours, hireachy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, Point());
Mat result = img.clone();
Point2f center;
float radius;

// 轮廓分析
for (size_t t = 0; t < contours.size(); t++) {
    minEnclosingCircle(contours[t], center, radius);
    circle(result, center, radius, Scalar(0, 255, 255), 2, 8, 0);
    Moments mm = moments(contours[t]);
    double cx = mm.m10 / mm.m00;
    double cy = mm.m01 / mm.m00;
    circle(result, Point(cx, cy), 2, Scalar(255, 0, 0), 2, 8, 0);
}

// 显示结果
imshow("binary", binary);
imwrite("D:/drawing.png", result);
```

运行显示：

第二部分代码实现：

```
// 加载图像
Mat img = imread("D:/CoinsB.png");
imshow("Original Image", img);

// 阈值化操作
Mat gray, binary;
cvtColor(img, gray, COLOR_BGR2GRAY);
float t = threshold(gray, binary, 0, 255, THRESH_BINARY|THRESH_OTSU);
imshow("binary", binary);
imwrite("D:/binary1.png", binary);

// 形态学操作
Mat se = getStructuringElement(MORPH_RECT, Size(3, 3));
morphologyEx(binary, binary, MORPH_OPEN, se, Point(-1, -1));

// 轮廓发现
vector<Vec4i> hireachy;
vector<vector<Point>> contours;
bitwise_not(binary, binary);
findContours(binary, contours, hireachy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, Point());
Mat result = img.clone();
Point2f center;
float radius;

// 轮廓分析
for (size_t t = 0; t < contours.size(); t++) {
    double area = contourArea(contours[t]);
    if (area < 1000) {
        continue;
    }
    RotatedRect rrt = fitEllipse(contours[t]);
    radius = min(rrt.size.width, rrt.size.height)/2.0;
    circle(result, rrt.center, radius, Scalar(0, 0, 255), 4, 8, 0);
    Moments mm = moments(contours[t]);
    double cx = mm.m10 / mm.m00;
    double cy = mm.m01 / mm.m00;
    circle(result, Point(cx, cy), 2, Scalar(255, 0, 0), 2, 8, 0);
}

// 显示结果
imshow("result", result);
imwrite("D:/drawing.png", result);
```

运行结果：