第5章 图像分割

本章学习目标

- 理解图像分割的基本概念。
- 解析阈值分割的几种算法。
- 解析边缘检测的常用算子。
- 掌握霍夫变换的原理和函数。
- 掌握轮廓检测和绘制函数。
- 掌握区域分割的几种方法。
- 了解基于特定理论的图像分割方法。

本章首先对图像分割进行概述；然后重点介绍了阈值分割、边缘分割和区域分割3种传统图像分割技术及利用Python语言实现分割效果；最后介绍了几种基于特定理论的图像分割方法。

5.1 图像分割概述

图像分割是把图像划分成若干具有独特性质的区域，并提出感兴趣目标的技术和过程。这些区域是互不相交的，分割后所得区域的总和应覆盖整个图像。具有独特性质是指满足像素的灰度、颜色、纹理等某种特征的相似性准则。具有同一性质的区域可以是单个区域，也可以是多个区域。

图像分割是图像分析过程中最重要的步骤之一，是图像识别和图像理解的前提。分割质量的好坏直接影响目标物特征提取和描述，关系到目标物识别、分类与理解及整个图像处理与分析系统的结果。

图像分割的应用非常广泛，几乎应用在有关图像处理的所有领域。例如，应用于医学领域，图像分割是病变区域提取、临床试验、特定组织测量以及实现三维重建的基础，可以帮助医生分析病情，进行组织器官的重建等；应用于交通领域，可用于桥梁裂缝检测、智能车辆导航、红外行人检测等；应用于军事领域，可以为军事目标的识别和跟踪提供特征参数，为导航与制导提供依据；应用于服装领域，可从服装图像中快速提取服装款式信息、结构元素，提高款式设计和制版效率，促进服装行业智能化、一体化发展。遥感图像可以提供真实、丰富的地面信息和资料。应用于遥感图像，可以进行城乡的建设与规划、地图的绘制与更新、考古和旅游资源的开发、森林资源及环境的监测与管理、农产品长势的检测与产量估计、

海岸区域的环境监测等。

图像分割算法发展几十年来，一直备受关注，借助各种理论，至今已提出了上千种各种类型的分割算法，但目前尚无普遍适用的最优分割算法，现有算法都是针对具体问题的。

图像分割方法主要分为以下几类：基于阈值的分割方法、基于边缘的分割方法、基于区域的分割方法以及基于特定理论的分割方法等。

针对单色图像的分割算法，通常基于灰度值的两个基本性质，即不连续性和相似性。不连续性是基于灰度的不连续变化分割图像，如图像边缘；相似性是依据一组预定义的准则将图像分割为相似区域，如阈值处理、区域生成、区域分裂合并等。这两类方法是互补的，有时需要结合使用，以求得更好的分割效果。

5.2 阈值分割原理与实现

阈值分割是一种常用的传统图像分割方法，因其处理直观、实现简单、计算速度快、性能较稳定而成为图像分割中最基本和应用最广泛的分割技术。

阈值分割的基本思想是利用图像中灰度值的差异，通过把图像中每个像素的灰度值与设置阈值进行比较，实现对像素的划分。如果只设置一个阈值，就把像素分为两类，高于阈值的分为一类，其他的分为另一类，也就是将图像分成两部分，即前景区和背景区。这种对整幅图像采用统一的阈值进行分割的方法叫全局阈值分割，适用于目标与背景有较强对比的图像。分割后习惯上将图像设置为黑白两色，就是所谓的图像二值化。当阈值在一幅图像上不再单一而是有改变时，就称为可变阈值分割。可变阈值分割适合于较复杂的图像情况。

基于阈值的分割算法中最核心的问题是如何找到合适的阈值，这一步骤直接影响分割的准确性以及由此产生的图像描述、分析的正确性。

阈值分割的一般步骤可总结如下。

(1) 确定阈值。

(2) 图像中像素的灰度值和阈值做比较。

(3) 像素划分。

本节重点介绍全局阈值分割的常用方法，包括固定阈值法、直方图双峰法、最大类间方差法、迭代法、最大熵法等。可变阈值分割在本节中不做重点介绍，仅在最后介绍一种自适应法。

5.2.1 固定阈值法

固定阈值法就是不使用相关算法去计算阈值，而是由用户自行设定一个阈值作为划分像素的依据。根据各像素点的灰度值与阈值的关系，将图像的灰度值分为两类。习惯上，一类灰度值为 0，另一类灰度值为 255，使整幅图像呈现出黑白效果。

OpenCV 提供的阈值处理函数为 ret，dst＝cv2. threshold(src，thresh，maxval，type)。其中，src 是源图像；thresh 是进行分类的阈值；maxval 是图像中像素的灰度值大于(或小于等于，根据 type 决定)阈值时赋予的新值(习惯上设置为 255，可以不为 255)；type 是一

个方法选择参数，常用的方法有以下几个。

① cv2.THRESH_BINARY(当前灰度值大于阈值时，设置为 maxval；否则设置为 0)。

② cv2.THRESH_BINARY_INV(当前灰度值大于阈值时，设置为 0；否则设置为 maxval)。

③ cv2.THRESH_TRUNC (当前灰度值大于阈值时，设置为阈值；否则不改变)。

④ cv2.THRESH_TOZERO(当前灰度值大于阈值时，不改变；否则设置为 0)。

⑤ cv2.THRESH_TOZERO_INV(当前灰度值大于阈值时，设置为 0；否则不改变)。

该函数有两个返回值，第一个是输入的阈值 ret，第二个是阈值化后的图像 dst。

【例 5.1】 利用 cv2.threshold()函数完成固定阈值法的阈值分割。

图 5.1 是对一幅灰度图像使用 cv2.threshold()函数进行阈值分割，type 分别取 cv2.THRESH_BINARY 和 cv2.THRESH_BINARY_INV 时图像效果图及直方图，此程序中阈值设置为 100。从图中可见，灰度图像被划分成了黑白二值图像，cv2.THRESH_BINARY_INV 是 cv2.THRESH_BINARY 的反相效果。

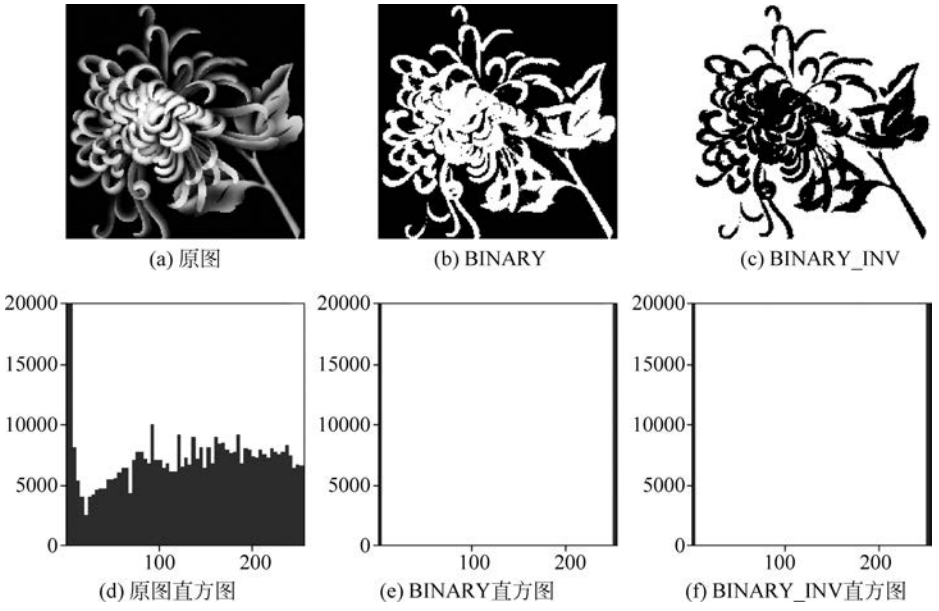


图 5.1 threshold()函数进行阈值分割

程序参考代码如下：

```
import cv2                                  #导入 opencv 库
import numpy as np                          #导入 numpy 库
import matplotlib.pyplot as plt             #导入 matplotlib 库的 pyplot 模块
#以灰度模式读入原始图像
original = cv2.imread("flower.jpg",0)
#threshold 函数,type 值不同
ret,dst1 = cv2.threshold(original,100,255,cv2.THRESH_BINARY)
ret,dst2 = cv2.threshold(original,100,255,cv2.THRESH_BINARY_INV) #反相效果
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
plt.rcParams.update({"font.size": 10})
titles1 = ["Original","BINARY","BINARY_INV"]
```

```
titles2 = ["Original 直方图","BINARY 直方图","BINARY_INV 直方图"]
images = [original,dst1,dst2]
plt.figure(figsize = (9,6))
#显示图像
for i in range(3):
    plt.subplot(2,3,i + 1)
    plt.imshow(images[i],"gray")
    plt.title(titles1[i])
    plt.axis("off")
#显示直方图
for i in range(3,6):
   plt.subplot(2,3,i + 1)
   a = np.array(images[i - 3]).flatten()
   plt.hist(a,bins = 64,color = "blue")
   plt.title(titles2[i - 3])
   plt.xlim(0,255)
   plt.ylim(0,20000)
plt.show()
```

有关 type 参数的其他效果参见实训 4。

5.2.2 直方图双峰法

1996 年,Prewitt 提出了直方图双峰法。该方法的基本思想是假设图像中有明显的目标和背景,则其灰度直方图呈双峰分布。当灰度级直方图具有双峰特性时,选取两峰之间的谷底对应的灰度级作为阈值,如图 5.2 所示。该方法不适合直方图中双峰差别很大或双峰间的谷比较宽广且平坦的图像以及单峰直方图的情况。对于有多个峰值的直方图,可以选择多个阈值,要根据实际情况具体分析。

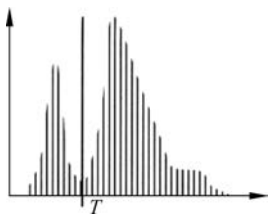


图 5.2 具有双峰分布的直方图

5.2.3 最大类间方差法

最大类间方差法是由日本学者大津(Nobuyuki Otsu)于 1979 年提出,是一种不需人为设定其他参数,自动选择阈值的方法,也称为 Otsu 算法或大津法。该算法计算简单,不受图像亮度和对比度的影响,被认为是阈值分割算法中阈值选取的最佳方法。

最大类间方差法的基本思想是通过阈值将图像分为前景和背景两部分,当取最佳阈值时,前景和背景之间的差别应该是最大的。方差是灰度分布均匀性的一种度量,衡量差别的标准采用最大类间方差。类间方差越大,说明构成图像的前景和背景之间的差别越大。若某些像素被错分,都会导致两部分差别变小。当使用所取阈值进行阈值分割,使类间方差最大就意味着错分概率最小。该算法的缺点是对图像噪声敏感,只能针对单一目标分割,当目标和背景大小比例悬殊时效果不理想。

设图像 $f(x,y)$的灰度范围是 $[0,L-1]$,灰度值为 i 的像素数为 n_i,图像的总像素数为 N,各灰度值出现的概率 p_i 为

$$p_i = \frac{n_i}{N}, \quad \text{且有} \sum_{i=0}^{L-1} p_i = 1$$

阈值 T 将图像像素按灰度值与 T 的关系分为两部分，即 C_0 和 C_1，C_0 由图像中灰度值在 $[0,T]$ 内的所有像素组成，C_1 由灰度值在 $[T+1,L-1]$ 内的所有像素组成，两部分的概率分别为 P_0 和 P_1，即

$$P_0=\sum_{i=0}^{T}p_i,\quad P_1=\sum_{i=T+1}^{L-1}p_i=1-P_0$$

两部分的平均灰度分别为 μ_0 和 μ_1，整幅图像的总平均灰度为 μ，有

$$\mu_0=\frac{1}{P_0}\sum_{i=0}^{T}ip_i,\quad \mu_1=\frac{1}{P_1}\sum_{i=T+1}^{L-1}ip_i$$

$$\mu=\sum_{i=0}^{L-1}ip_i=\sum_{i=0}^{T}ip_i+\sum_{i=T+1}^{L-1}ip_i=P_0\mu_0+P_1\mu_1$$

两部分的类间方差 g 为

$$g=P_0(\mu_0-\mu)^2+P_1(\mu_1-\mu)^2$$

将 μ 的表达式代入上式，可得简化公式为

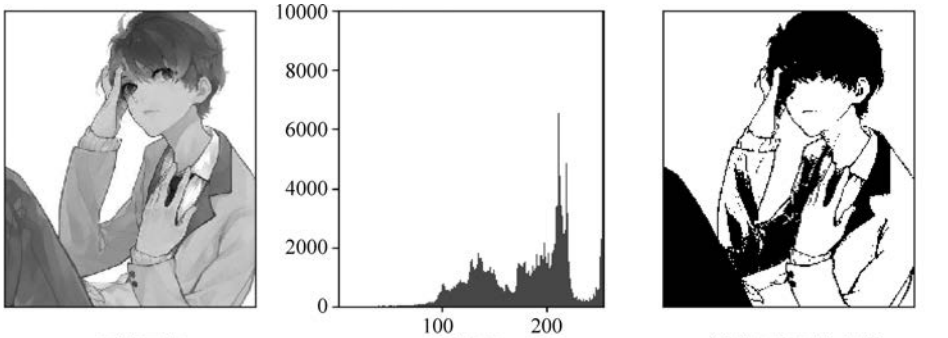
$$g=P_0P_1(\mu_0-\mu_1)^2$$

T 在 $[0,L-1]$ 内依次取值，使 g 最大的值就是需要的阈值。

OpenCV 提供的阈值处理函数 cv2.threshold(src, thresh, maxval, type)，type 取 THRESH_OTSU，thresh 取 0(屏蔽 thresh)时，就会使用 Otsu 算法得到的阈值(函数第一个返回值)分割图像。type 取 THRESH_OTSU 时也可以同时搭配 THRESH_BINARY、THRESH_BINARY_INV、THRESH_TRUNC、THRESH_TOZERO 及 THRESH_TOZERO_INV 使用。

【例 5.2】 利用 cv2.threshold()函数完成 Otsu 算法的阈值分割。

图 5.3 是利用 cv2.threshold()函数对一幅灰度图像用 Otsu 算法进行分割的效果图和直方图，并显示了分割阈值。



(a) 原始图像　(b) 直方图　(c) Otsu算法分割图像(阈值187.0)

图 5.3　Otsu 算法进行阈值分割

程序参考代码如下：

```
import cv2
from matplotlib import pyplot as plt
original = cv2.imread("boy.jpg",0)
ret1,dst = cv2.threshold(original,0,255,cv2.THRESH_OTSU)
plt.rcParams["font.sans-serif"] = ["SimHei"]
plt.figure(figsize = (9,3))
```

```
plt.subplot(131)
plt.imshow(original,"gray")
plt.title("原始图像")
plt.xticks([])
plt.yticks([])
plt.subplot(132)
plt.hist(original.ravel(),256)     # original.ravel()将数组拉成一维数组
plt.title("直方图")
plt.xlim(0,255)
plt.ylim(0,10000)
plt.subplot(133)
plt.imshow(dst,"gray")
plt.title("Otsu算法分割图像,阈值" + str(ret1))
plt.xticks([])
plt.yticks([])
plt.show()
```

如需得到分割后的反相效果,只需使用语句:

```
ret1,dst = cv2.threshold(original,0,255,cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
```

效果如图 5.4 所示。

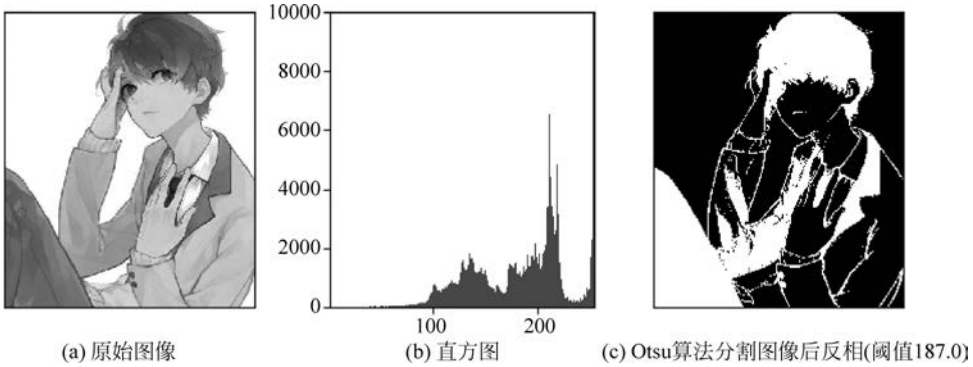


图 5.4　Otsu 算法进行阈值分割后反相效果

5.2.4　迭代法

迭代法的基本思想是设定一个阈值 T 作为初始估计值,然后根据某种规则不断更新这一估计值,直到满足给定条件。迭代算法的优劣取决于迭代规则的确定。好的迭代规则既能快速收敛又能产生优于上次迭代的结果。迭代法适合直方图有明显波谷的图像。

下面是一种应用较广泛的迭代算法,步骤如下。

(1) 用图像的最大灰度值和最小灰度值的平均值作为初始估计阈值 T。

(2) 将图像像素按灰度值与阈值 T 的关系分为两部分,即 C_0 和 C_1,两部分的平均灰度分别为 μ_0 和 μ_1。

(3) 计算新的阈值 T,$T=\frac{1}{2}(\mu_0+\mu_1)$。

(4) 重复步骤(2)和(3),直到连续迭代中的 T 值间的差小于一个预定义参数为止。

【例 5.3】 利用迭代法对图像进行阈值分割。

图 5.5 是利用迭代法对图 5.3 所示的原始图像进行阈值分割的效果。

(a) 原始图像　　(b) 迭代法分割图像(阈值186.0)

图 5.5　迭代法阈值分割效果

程序参考代码如下：

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
def iteration_threshold(gray):
    gray = np.array(gray).astype(np.float32)
    zmax = np.max(gray)
    zmin = np.min(gray)
    t0 = (zmax + zmin)/2                    #设置初始阈值
    m,n = gray.shape
    fnum = 0                                #初始化前景像素个数
    bnum = 0                                #初始化背景像素个数
    fsum = 0                                #初始化前景像素灰度值的和
    bsum = 0                                #初始化背景像素灰度值的和
    for i in range(m):
        for j in range(n):
            tmp = gray[i][j]
            if tmp > t0:
                fnum = fnum + 1             #前景像素的个数
                fsum = fsum + int(tmp)      #前景像素灰度值的总和
            else:
                bnum = bnum + 1             #背景像素的个数
                bsum = bsum + int(tmp)      #背景像素灰度值的总和
    #计算前景和背景的平均灰度
    zf = int(fsum/fnum)
    zb = int(bsum/bnum)
    t = (zf + zb)/2 #求出新的阈值
    while(abs(t - t0)> 1):                  #设置两次阈值差值不大于 1
        t0 = t
        fnum = 0
        bnum = 0
        fsum = 0
        bsum = 0
        for i in range(m):
            for j in range(n):
```

```
                tmp = gray[i][j]
                if tmp > t0:
                    fnum = fnum + 1
                    fsum = fsum + int(tmp)
                else:
                    bnum = bnum + 1
                    bsum = bsum + int(tmp)
            zf = int(fsum/fnum)
            zb = int(bsum/bnum)
            t = (zf + zb)/2
        return t
original = cv2.imread("boy.jpg",0)
threshold = iteration_threshold(original)
ret,dst = cv2.threshold(original,threshold,255,cv2.THRESH_BINARY)
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
plt.subplot(121)
plt.imshow(original,"gray")
plt.title("原始图像")
plt.xticks([])
plt.yticks([])
plt.subplot(122)
plt.imshow(dst,"gray")
plt.title("迭代法分割图像,阈值" + str(ret))
plt.xticks([])
plt.yticks([])
plt.show()
```

5.2.5 最大熵法

熵是信息论中的一个术语,是所研究对象平均信息量的表征。假设离散随机变量 x 的概率分布是 $p(x)$,则其熵为

$$H(p) = -\sum_{x} p(x)\log p(x)$$

式中,log 的底数可以取 2、e 或 10。取不同的底数,熵的单位不一样。熵的定义使随机变量的不确定性得到了度量。熵越大,随机变量越不确定,也就是随机变量越随机,意味着添加的约束和假设越少,这时求出的分布越自然、偏差越小、越具有均匀分布。

最大熵算法就是找出一个最佳阈值把图像分成前景和背景两部分,使得两部分熵之和最大。从信息论角度来说,这样选择的阈值分割出的图像信息量最大,最不确定,分布偏差最小。

设图像 $f(x,y)$的灰度范围是 $[0,L-1]$,灰度值为 i 的像素数为 n_i,图像的总像素数为 N,各灰度值出现的概率 p_i 为

$$p_i = \frac{n_i}{N}, \quad \text{且有} \sum_{i=0}^{L-1} p_i = 1$$

以灰度值 T 分割图像,图像中不大于灰度值 T 的像素点构成背景 B,高于灰度值 T 的像素点构成目标物体 O,各灰度在本区的概率分布为

B 区: $\frac{p_i}{p_T}, i = 0,1,2,\cdots,T$

O 区：$\frac{p_i}{1-p_T}, i=T+1, T+2, \cdots, L-1$

其中

$$p_T = \sum_{i=0}^{T} p_i$$

背景区域和目标区域的熵分别为

$$H_B = -\sum_{i=0}^{T} \frac{p_i}{p_T} \lg \frac{p_i}{p_T}$$

$$H_O = -\sum_{i=T+1}^{L-1} \frac{p_i}{1-p_T} \lg \frac{p_i}{1-p_T}$$

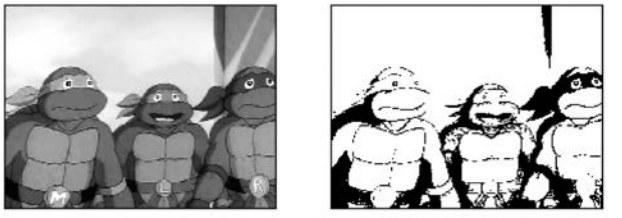
图像总熵为

$$H = H_B + H_O$$

使 H 最大的 T 值就是寻找的最佳阈值。

【例 5.4】 利用最大熵法对图像进行阈值分割。

图 5.6 是一幅灰度图像使用最大熵法实现阈值分割的效果图。



(a) 原始图像　　(b) 最大熵法分割图像(阈值123.0)

图 5.6　最大熵法阈值分割效果

程序参考代码如下：

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
def calentropy(T):
#初始化
    PT = 0                                                    #概率和
    entropyB = 0                                              #背景熵
    entropyO = 0                                              #前景熵
    for i in range(T + 1):
        PT += p[i]                                            #0～T 灰度级的概率和
    for i in range(256):
        if i <= T:
            if p[i]!= 0:
                entropyB -= (p[i]/PT) * np.log10(p[i]/PT)     #背景熵
        else:
            if p[i]!= 0:
                entropyO -= (p[i]/(1 - PT)) * np.log10(p[i]/(1 - PT))   #前景熵
    entropy = entropyB + entropyO                             #背景熵和前景熵的和
    return entropy
def maxentropy(image):
    Entropy = []
```

```
    for i in range(256):
        entropy = calentropy(i)                    #将返回的熵赋给 entropy
        Entropy.append(entropy)                    #各个 entropy 形成列表,列表索引为各灰度级
    return Entropy.index(max(Entropy))             #Entropy 列表中最大值的索引号就是所求阈值
original = cv2.imread("turtles.jpg",0)
hist = cv2.calcHist([original],[0],None,[256],[0,255])   #返回直方图各个灰度级的像素个数
m,n = original.shape
sum = m * n
p = []
for i in range(256):
    p.append(hist[i]/sum)                          #计算概率
threshold = maxentropy(original)
ret,dst = cv2.threshold(original,threshold,255,cv2.THRESH_BINARY)
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
plt.figure(figsize = (6,3))
plt.subplot(121)
plt.imshow(original,"gray")
plt.title("原始图像")
plt.xticks([])
plt.yticks([])
plt.subplot(122)
plt.imshow(dst,"gray")
plt.title("最大熵法分割图像,阈值" + str(ret))
plt.xticks([])
plt.yticks([])
plt.show()
```

5.2.6 自适应法

在绝大多数情况下,目标和背景的对比度在图像中各处不是完全一样的,很难用一个统一的阈值将目标和背景区分开,这时往往需要通过控制阈值选取范围的方法实现局部分割阈值的选择。自适应法对于图像不同区域,能够自适应计算不同的阈值。

OpenCV 提供 dst=cv2.adaptiveThreshold(src, maxValue,adaptiveMethod,thresholdType,blockSize,C)函数,该函数可以通过计算某个邻域(局部)的均值、高斯加权平均来确定阈值。

① src:表示源图像,8 位单通道图像。

② dst:表示输出图像,与源图像大小一致。

③ maxValue:表示大于(小于等于)阈值时赋予的新值。

④ adaptiveMethod:表示在一个邻域内计算阈值所采用的算法,有以下两个取值。

- ADAPTIVE_THRESH_MEAN_C 的计算方法是计算出邻域的平均值再减去常量 *C* 的值作为阈值;
- ADAPTIVE_THRESH_GAUSSIAN_C 的计算方法是计算出邻域的高斯均值再减去常量 *C* 的值作为阈值。

⑤ thresholdType:表示阈值类型,只有两个取值,即 THRESH_BINARY 和 THRESH_BINARY_INV。

⑥ blockSize:表示计算阈值的像素邻域大小,可取 3、5、7 等奇数,取值越大,结果越表现为阈值分割效果(可取 21、31、41),取值越小,结果越表现为边缘检测效果。

⑦ C：表示偏移值调整量，值越大越能抑制噪声。用均值或高斯均值计算阈值后，再减这个值就是最终阈值。

【例 5.5】 利用 cv2.adaptiveThreshold()函数体会自适应法进行阈值分割。

图 5.7 是一幅灰度图像使用自适应法实现阈值分割的效果图。

图 5.7　自适应法阈值分割效果

程序参考代码如下：

```
import cv2
import Matplotlib.pyplot as plt
original = cv2.imread("characters.jpg",0)
dst = cv2.adaptiveThreshold(original,255,cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BINARY,41,
3)
plt.rcParams["font.sans-serif"] = ["SimHei"]
plt.subplot(121)
plt.imshow(original,"gray")
plt.title("原始图像")
plt.xticks([])
plt.yticks([])
plt.subplot(122)
plt.imshow(dst,"gray")
plt.title("自适应法分割图像")
plt.xticks([])
plt.yticks([])
plt.show()
```

有关全局阈值分割(选用 Otsu 算法)与可变阈值分割(选用自适应法)对同一幅图像进行阈值分割的效果对比参见实训 4。

5.3　边缘分割原理与实现

边缘检测是图像分割的一种重要途径。图像边缘是图像最基本也是最重要的特征。边缘指图像局部特性不连续，总是以某种图像特征所对应的数值发生突变的形式出现，如灰度值、颜色分量、纹理结构的突变都会形成图像边缘，它标志着一个区域的终结和另一个区域的开始。根据灰度变换的特点，常见的边缘可分为阶跃型、斜坡型和屋顶型，如图 5.8 所示。

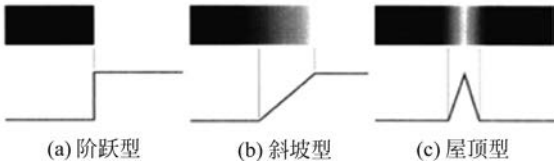


图 5.8　常见的边缘类型

图像边缘有方向和幅度两个属性，沿边缘方向像素变化平缓，垂直于边缘方向像素变化剧烈。边缘上的这种变化可以用微分算子检测出来，在第 4 章中曾讲过微分算子在图像锐化处理中的应用，本节将它应用于图像分割。基于微分算子的边缘检测是经典的边缘检测方法，通常用一阶或二阶导数来实现。一阶导数的最大值对应图像边缘，二阶导数以过零点对应图像边缘。在数字图像处理中，经证实，一阶导数和二阶导数具有以下性质：一阶导数通常在图像中产生较粗的边缘；二阶导数对精细细线，如细线、孤立点和噪声有较强的响应；二阶导数在灰度斜坡和灰度台阶过渡处会产生双边缘响应；二阶导数的符号可用于确定边缘的过渡是从亮到暗还是从暗到亮。由于边缘和噪声都是灰度不连续点，检测边缘的同时噪声会对其产生影响，因此用微分算子检测边缘前要对图像进行平滑滤波。

基于边缘检测的图像分割主要通过以下 4 步实现。

(1) 图像平滑。对图像进行平滑处理以降低噪声，但是降低噪声的平滑能力越强，边缘强度的损失越大。这一步通过平滑滤波实现。

(2) 边缘点检测。将邻域中灰度有显著变化的点突出显示，这些点是边缘点的候选者。这一步通过锐化滤波实现。

(3) 边缘判断。根据具体情况，选择边缘点，去除没有意义的边缘点。实际工程中，这一步通过阈值化实现。

(4) 边缘连接。前 3 步操作后得到的仅是边缘像素点，这些边缘像素点很少能完整地描绘实际的一条边缘，因此需要紧接着使用连接方法将这些边缘像素点形成有意义的边缘。

5.3.1 常用边缘检测算子

目前常用的边缘检测算子主要有 Roberts 算子、Prewitt 算子、Sobel 算子、LoG 算子和 Canny 算子等，其中 Roberts 算子、Prewitt 算子和 Sobel 算子都是基于一阶导数的，都是梯度算子；LoG 是基于二阶导数的算子。有关梯度算子的知识已在 4.4.2 小节空间邻域锐化中介绍过，本节简单回顾。

1. 梯度算子

图像在计算机中以数值矩阵的形式存在，形成了离散的数值信号。图像灰度函数 $f(x,y)$ 在点 (x,y) 的梯度是一个具有方向和大小的矢量。梯度的方向就是函数 $f(x,y)$ 最大变化率的方向，梯度幅值作为变化率大小的度量，可以反映图像灰度局部变化的强弱，因此可以作为检测边缘点的依据。

梯度表示为

$$\nabla \boldsymbol{f} = \mathbf{grad}(\boldsymbol{f}) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}$$

式中，g_x 和 g_y 分别为 x 方向和 y 方向的梯度。

梯度幅值为

$$M(x,y) = \mathrm{mag}(\nabla \boldsymbol{f}) = \sqrt{g_x^2 + g_y^2} = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

可以用下式近似表示，即

$$M(x,y)\approx |g_x|+|g_y|=\left|\frac{\partial f}{\partial x}\right|+\left|\frac{\partial f}{\partial y}\right|$$

梯度的方向角为(相对于 x 轴而言)

$$\alpha(x,y)=\arctan\left[\frac{g_y}{g_x}\right]$$

将$\frac{\partial f}{\partial x}$和$\frac{\partial f}{\partial y}$用差分近似表示，图像的梯度幅值就可以通过对原始图像进行空间滤波求得。图 5.9 是一个 3×3 空间滤波器，滤波的过程就是求图像中各像素点的新像素值的过程，模板系数与被该模板覆盖区域的灰度值的乘积之和作为模板覆盖区域下图像中心点处的新灰度值。图 5.10 是一幅图像被模板覆盖的 3×3 区域，z_k 表示各灰度值，z_5 的新值 $=w_1z_1+w_2z_2+\cdots+w_9z_9=\sum_{k=1}^{9}w_kz_k$。对原始图像进行滤波可分别求得 g_x 和 g_y，通过 $M(x,y)\approx|g_x|+|g_y|$，求得 $M(x,y)$。

常用的梯度算子如下。

(1) Roberts 算子。

对于图 5.10 中的 3×3 区域，Roberts 算子以求对角像素之差为基础，有

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

图 5.9 3×3 空间滤波器

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

图 5.10 图像的 3×3 区域

$$g_x=\frac{\partial f}{\partial x}=z_9-z_5$$

$$g_y=\frac{\partial f}{\partial y}=z_8-z_6$$

g_x 和 g_y 分别使用模板 $\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}$ 和 $\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}$ 对图像滤波求得。

Roberts 算子常用来处理具有陡峭边缘的低噪声图像，当边缘接近±45°时，处理效果更好，但其边缘定位较差，对噪声敏感，提取的边缘比较粗，常需对检测出的边缘图像做细化处理。

(2) Prewitt 算子。

还是对于图 5.10 中的 3×3 区域，有

$$g_x=\frac{\partial f}{\partial x}=(z_7+z_8+z_9)-(z_1+z_2+z_3)$$

$$g_y=\frac{\partial f}{\partial y}=(z_3+z_6+z_9)-(z_1+z_4+z_7)$$

g_x 和 g_y 分别使用模板 $\begin{bmatrix}-1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1\end{bmatrix}$(水平)和 $\begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix}$(垂直)对图像滤波

求得。

这两种算子可通过调用 OpenCV 的 filter2D()函数实现。该函数在 4.4.2 小节中已有介绍,此处不再赘述。

(3) Sobel 算子。

同样对于图 5.10 中的 3×3 区域,有

$$g_x=\frac{\partial f}{\partial x}=(z_7+2z_8+z_9)-(z_1+2z_2+z_3)$$

$$g_y=\frac{\partial f}{\partial y}=(z_3+2z_6+z_9)-(z_1+2z_4+z_7)$$

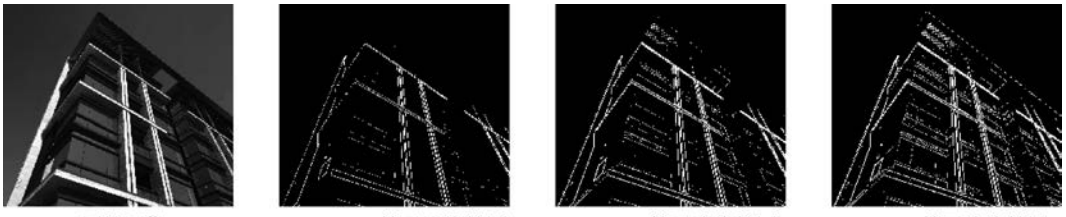
g_x 和 g_y 分别使用模板 $\begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}$(水平)和 $\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}$(垂直)对图像滤波求得。

Sobel 算子在 OpenCV 中用 Sobel()函数实现。该函数在 4.4.2 小节中已有介绍,此处不再赘述。

Prewitt 算子、Sobel 算子因考虑到邻域信息,相当于对图像先做加权平滑处理,再做微分运算,因此对噪声有一定的抑制能力,适合具有较多噪声且灰度渐变图像的分割。

【例 5.6】 用梯度算子进行边缘检测。

图 5.11 是对一幅灰度图像用不同梯度算子进行边缘检测的效果图。处理时先进行高斯平滑;然后进行边缘检测;最后进行阈值处理。



(a) 原始图像 (b) Roberts算子边缘检测 (c) Prewitt算子边缘检测 (d) Sobel算子边缘检测

图 5.11 不同梯度算子边缘检测效果

程序参考代码如下:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#读取图像并进行高斯平滑处理
original = cv2.imread("building.jpg",0)
original1 = cv2.GaussianBlur(original,(3,3),0)
#Roberts 算子
kernelxr = np.array([[-1,0],[0,1]])
kernelyr = np.array([[0,-1],[1,0]])
xr = cv2.filter2D(original1,cv2.CV_16S,kernelxr)
yr = cv2.filter2D(original1,cv2.CV_16S,kernelyr)
#转回 CV_8U
absXr = cv2.convertScaleAbs(xr)
absYr = cv2.convertScaleAbs(yr)
#融合
```

```
Roberts = cv2.addWeighted(absXr,0.5,absYr,0.5,0)
#Prewitt 算子
kernelxp = np.array([[-1,-1,-1],[0,0,0],[1,1,1]])
kernelyp = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
xp = cv2.filter2D(original1,cv2.CV_16S,kernelxp)
yp = cv2.filter2D(original1,cv2.CV_16S,kernelyp)
#转回 CV_8U
absXp = cv2.convertScaleAbs(xp)
absYp = cv2.convertScaleAbs(yp)
#融合
Prewitt = cv2.addWeighted(absXp,0.5,absYp,0.5,0)
#Sobel 算子
xs = cv2.Sobel(original1,cv2.CV_16S,1,0)
ys = cv2.Sobel(original1,cv2.CV_16S,0,1)
#转回 CV_8U
absXs = cv2.convertScaleAbs(xs)
absYs = cv2.convertScaleAbs(ys)
#融合
Sobel = cv2.addWeighted(absXs,0.5,absYs,0.5,0)
#阈值化
Roberts_ret1,Roberts1 = cv2.threshold(Roberts,0,255,cv2.THRESH_OTSU)
Prewitt_ret1,Prewitt1 = cv2.threshold(Prewitt,0,255,cv2.THRESH_OTSU)
Sobel_ret1,Sobel1 = cv2.threshold(Sobel,0,255,cv2.THRESH_OTSU)
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
titles = ["原始图像","Roberts 算子边缘检测","Prewitt 算子边缘检测","Sobel 算子边缘检测"]
images = [original,Roberts1,Prewitt1,Sobel1]
plt.figure(figsize = (14,3))
for i in range(4):
    plt.subplot(1,4,i + 1)
    plt.imshow(images[i],"gray")
    plt.title(titles[i])
    plt.axis("off")
plt.show()
```

2. LoG 算子

图像函数 $f(x,y)$的拉普拉斯算子是根据数字图像中阶跃型边缘点对应二阶导数的过零点而设计的一种与方向无关的边缘检测算子,定义为

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

其中

$$\frac{\partial^2 f}{\partial x^2} = f(x+1,y) + f(x-1,y) - 2f(x,y)$$

$$\frac{\partial^2 f}{\partial y^2} = f(x,y+1) + f(x,y-1) - 2f(x,y)$$

故

$$\nabla^2 f(x,y) = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

拉普拉斯算子的缺点是对噪声非常敏感,而且会产生双边缘,因此一般不用其原始形式作边缘检测。

针对拉普拉斯算子抗噪声能力差的缺点，Marr 和 Hildreth 提出先对图像进行高斯平滑，再用拉普拉斯算子进行边缘检测的改进方法。这一过程可表示为

$$\nabla^2[G(x,y)\bigstar f(x,y)]$$

式中：★表示卷积；$f(x,y)$为图像；$G(x,y)$为高斯函数，即

$$G(x,y)=\frac{1}{2\pi\sigma^2}\mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}}$$

σ 是标准差，直接影响检测结果。一般来说 σ 越大，抗噪声能力越强，但是会导致一些变化细微的边缘检测不到。

线性系统中，卷积与微分的次序可以交换，先卷积后微分和先微分后卷积是相等的，所以

$$\nabla^2[G(x,y)\bigstar f(x,y)]=\nabla^2G(x,y)\bigstar f(x,y)$$

即先对高斯算子做拉普拉斯变换，得到$\nabla^2G(x,y)$，再与图像卷积，等同于先对图像进行高斯平滑，再用拉普拉斯算子进行边缘检测。

$$\begin{aligned}\nabla^2G(x,y)&=\frac{\partial^2G(x,y)}{\partial x^2}+\frac{\partial^2G(x,y)}{\partial y^2}\\&=\frac{1}{2\pi\sigma^4}\left[\frac{x^2}{\sigma^2}-1\right]\mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}}+\frac{1}{2\pi\sigma^4}\left[\frac{y^2}{\sigma^2}-1\right]\mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}}\\&=-\frac{1}{\pi\sigma^4}\left[1-\frac{x^2+y^2}{2\sigma^2}\right]\mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}}\end{aligned}$$

将$\nabla^2G(x,y)$称为 LoG(Laplacian of Gaussian，高斯-拉普拉斯)算子。

LoG 算子形如草帽，也称墨西哥草帽算子，$\begin{bmatrix}0&0&-1&0&0\\0&-1&-2&-1&0\\-1&-2&16&-2&-1\\0&-1&-2&-1&0\\0&0&-1&0&0\end{bmatrix}$(实践中，用该模板的负模板)是一个对 LoG 算子近似的 5×5 模板，这种近似并不是唯一的。任意尺寸的模板可以通过对$\nabla^2G(x,y)$取样，并标定系数以使系数之和为零来生成。LoG 算子本质的形状是：一个正的中心项由紧邻的负区域包围着，中心项的值以距原点的距离为函数而增大，而外层区域的值为零。系数之和为零，以便在灰度级恒定区域中模板的响应为零。在应用 LoG 算子时，标准差参数 σ 的值对边缘检测效果有很大的影响，图像不同，选择的参数值不同。

LoG 是一个对称滤波器，所以使用相关或卷积的空间滤波将产生相同的结果。关于 LoG 的参考文献都采用卷积说法，为保持一致，也使用卷积来表示线性滤波。

在具体实现时，直接用 LoG 算子与图像卷积或先高斯平滑图像，再求平滑后图像的拉普拉斯这两种方法都是可以的，但是后者要进行两次卷积，图像较大时一般不建议采用。

另外，Scipy 库的 ndimage 子模块专门用于图像处理，提供了 gaussian_laplace( )函数用于进行 LoG 边缘检测。

【例 5.7】 用 LoG 算子进行边缘检测。

图 5.12 是对一幅灰度图像用 LoG 算子进行边缘检测的效果图。本例先进行高斯平滑；然后用拉普拉斯算子进行边缘点检测；最后进行阈值处理。阈值处理时以 LoG 边缘点图像最大灰度值的 50%为阈值，保留了大多数的主要边缘点，过滤掉了一些“无关”的特征。

(a) 原始图像　(b) LoG算子得到的边缘点图像　(c) LoG算子边缘检测

图 5.12　LoG 算子边缘检测效果

程序参考代码如下：

```
import cv2
import matplotlib.pyplot as plt
#读取图像并进行高斯平滑处理
original = cv2.imread("building.jpg",0)
original1 = cv2.GaussianBlur(original,(3,3),0)
#拉普拉斯算子
Laplace = cv2.Laplacian(original1,cv2.CV_16S,ksize = 3)
#转回原来的 uint8 形式；否则将无法显示图像，而只是一幅灰色的窗口图像
LoG = cv2.convertScaleAbs(Laplace)
#阈值化
m,n = LoG.shape
max = 0
for i in range(m):
    for j in range(n):
        if LoG[i][j]> max:
            max = LoG[i][j]
ret,LoG1 = cv2.threshold(LoG,max * 0.5,255,cv2.THRESH_BINARY)
#显示
plt.rcParams["font.sans - serif"] = ["SimHei"]
titles = ["原始图像","LoG 算子得到的边缘点图像","LoG 算子边缘检测"]
images = [original,LoG,LoG1]
plt.figure(figsize = (12,3))
for i in range(3):
    plt.subplot(1,3,i + 1)
    plt.imshow(images[i],"gray")
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```

3. Canny 算子

Canny 边缘检测算法是一种多级检测算法，它在抑制噪声和边缘精确定位之间取得了较好的平衡，是边缘检测中最经典、最先进的算法之一。它基于 3 个基本目标。

(1) 低错误率。

检测算法应该尽可能检测出图像的真实边缘，尽可能减少漏检和误检。

(2) 最优定位。

检测的边缘点应该精确地定位于边缘的中心。

(3) 单边缘响应。

对于同一边缘要有尽可能低的响应次数,即对单边缘最好只有一个响应。

Canny 边缘检测算法较为复杂,主要通过以下 4 步实现。

(1) 高斯滤波。用高斯滤波器平滑图像,以降低错误率。

(2) 计算图像的梯度幅值和方向。用一阶偏导的有限差分计算梯度的幅值和方向。

(3) 非极大值抑制。为确定边缘,需将非局部极大值点置零以得到细化的边缘。

(4) 双阈值筛选边缘。使用两个阈值 T_1 和 T_2($T_1<T_2$),如果梯度值比 T_1 小,就认为不是边缘点。如果梯度值比 T_2 大,就认为是边缘点。如果梯度值介于 $T_1\sim T_2$,并且该点与大于 T_2 的点是 8 连通的,就认为是边缘点;否则就认为不是边缘点。推荐的高、低阈值比值为 $T_2:T_1=2:1$ 或 3∶1。

在 Python 中,Canny 算子可通过调用 OpenCV 的 cv2.Canny()函数实现,具体使用方法为:

```
edge = cv2.Canny(image,threshold1,threshold2[,apertureSize[,L2gradient]])
```

① edge:输出的边缘图,8 位单通道黑白图。

② image:输入图像。

③ threshold1:阈值 T_1。

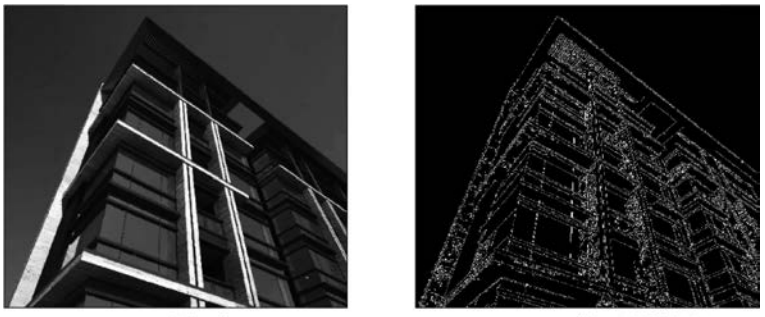
④ threshold2:阈值 T_2。

⑤ apertureSize:应用 Sobel 算子的孔径大小,默认值为 3。

⑥ L2gradient:用来设定求梯度大小的方程,如果设为 True,就使用 $M(x,y)=\sqrt{g_x^2+g_y^2}$;否则使用 $M(x,y)=|g_x|+|g_y|$,默认值为 False 。

【例 5.8】 用 Canny 算子进行边缘检测。

图 5.13 是对一幅灰度图像用 Canny 算子进行边缘检测的效果。



(a) 原始图像　　(b) Canny算子边缘检测

图 5.13　Canny 算子边缘检测

程序参考代码如下:

```
import cv2
import matplotlib.pyplot as plt
#读取图像
original = cv2.imread("building.jpg",0)
```

```
#Canny 算子
result = cv2.Canny(original,50,150)
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
titles = ["原始图像","Canny 算子边缘检测"]
images = [original,result]
plt.figure(figsize = (14,6))
for i in range(2):
    plt.subplot(1,2,i + 1)
    plt.imshow(images[i],"gray")
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```

5.3.2 霍夫变换

在实际应用中，由于不均匀照明和噪声等因素的影响，通过 5.3.1 小节边缘检测方法获得的边缘点往往是不连续的，因此一般在边缘检测后紧跟连接算法，将边缘像素点连接成有意义的边缘或区域边界。

霍夫变换(Hough Transform)是在应用中使用的非常有代表性的边缘连接技术，主要用来从图像中识别几何形状，如直线、圆等。它将在一个空间中具有相同形状的直线或曲线映射到另一个空间的一个点上形成峰值，从而把检测任意形状的问题转化为统计峰值问题。霍夫变换的前提是已经做了边缘检测，得到了边缘二值图像。

1. 直线检测

在图像 $x-y$ 坐标空间，经过点(x_1,y_1)的直线可表示为

$$y_1 = kx_1 + q$$

式中，k、q 为参数，k 为斜率，q 为截距。

通过点(x_1,y_1)的直线有无数条，且对应不同的 k 和 q 值。

如果把(x_1,y_1)视为常数，参数 k、q 视为变量，就可以把 $y_1=kx_1+q$ 写成

$$q = -x_1k + y_1$$

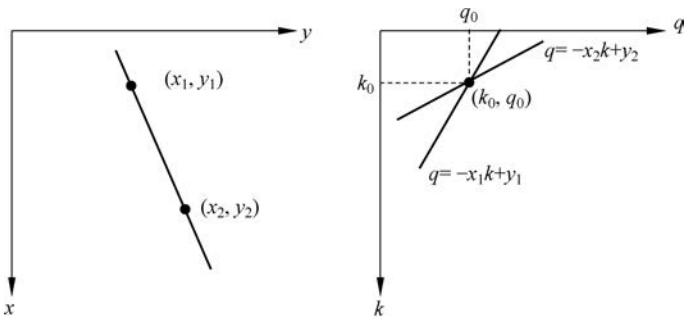
就变换到了参数空间 $k-q$，这个变换就是点(x_1,y_1)的霍夫变换。这条直线是图像坐标空间中点(x_1,y_1)在参数空间的唯一方程。

再来看图像坐标空间中的另一个点(x_2,y_2)，它在参数空间中也有一条相应的直线，即

$$q = -x_2k + y_2$$

这条直线与点(x_1,y_1)在参数空间的直线交于点(k_0,q_0)，其中 k_0 和 q_0 分别是图像坐标空间中点(x_1,y_1)和(x_2,y_2)形成的直线的斜率和截距。对应的变换如图 5.14 所示。

事实上，图像坐标空间中点(x_1,y_1)和点(x_2,y_2)形成的直线上的每一点在参数空间 $k-q$ 上各自对应一条直线，这些直线都相交于点(k_0,q_0)；反之，在参数空间相交于同一点的所有直线，在图像坐标空间都有共线的点与之对应。通过参数空间相交点的坐标，就可以确定图像坐标空间共线的点形成的直线的斜率和截距。依据这一特性，将图像坐标空间的各个边缘点投影到参数空间，看参数空间有无聚集点，聚集点就对应了图像坐标空间的直线。这就是霍夫变换的原理。



(a) 图像坐标空间　　(b) 参数空间

图 5.14　霍夫变换

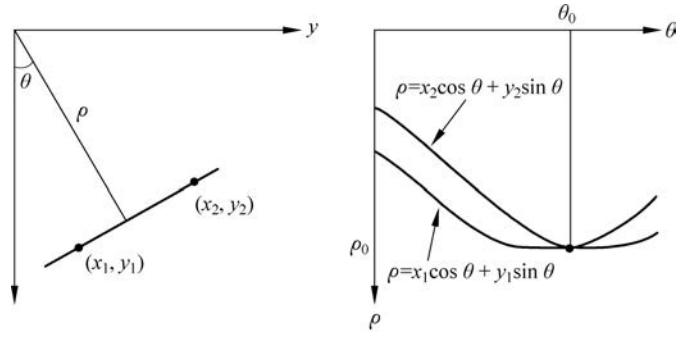
具体实现时，先创建一个二维数组，将其设置为 0。第一维的范围是图像坐标空间中直线斜率的可能范围，第二维的范围是图像坐标空间中直线截距的可能范围。然后对图像坐标空间上每一边缘点(x_i, y_i)，根据不同的 k，求得对应的 q，并将每对(k, q)对应的数组元素加 1。最后找数组元素的最大值，其对应的 k、q 就是图像坐标空间共线点数目最多的直线的参数。简单讲就是对图像坐标空间上每一边缘点(x_i, y_i)，求出参数空间对应的直线，把对应直线上所有的点(k, q)对应的数组元素都加 1，最后找到参数空间最大点的位置，这个位置就是图像坐标空间上直线的参数。如果图像坐标空间上有两条直线，那么在参数空间就会看到两个峰值点。

但在实际应用中，在图像坐标空间使用 $y=kx+q$ 表示直线的方法，不能表示斜率为无穷大或接近无穷大的直线，因此采用下面参数表示图像坐标空间中的直线，即

$$\rho = x\cos\theta + y\sin\theta$$

式中，ρ、θ 为参数，ρ 表示直线到原点的垂直距离，θ 表示 x 轴与直线垂线的夹角。

虽然在图像坐标空间中直线的表示参数发生了改变，但霍夫变换的原理并没有改变。不同的是，在将图像坐标空间中的点变换到参数空间时，图像坐标空间中的一个点对应到参数 ρ-θ 空间上的不再是一条直线，而是一条正弦曲线。其他的还是一样。图像坐标空间中共线的点变换到参数空间后，在参数空间相交于点(ρ_0, θ_0)，(ρ_0, θ_0)就是所求直线的参数，如图 5.15 所示。



(a) 直线的ρ、θ参数表示　　(b) 参数空间

图 5.15　用 ρ、θ 参数表示的共线点的霍夫变换

最后，总结一下，在参数空间，越多对应线（直线或正弦曲线）交于一点就意味着这个交

点表示的直线由越多的点组成。可以通过设置图像中直线上点的阈值来定义多少条对应线交于一点时，就认为检测到了一条直线。霍夫变换就是追踪图像中每个边缘点在参数空间中对应线的交点，如果交于一点的对应线数量超过设置的阈值，就可以认为这个交点是图像中一条直线的参数。

在 OpenCV 中，提供了 3 种不同的霍夫变换。

(1) 标准霍夫变换。

(2) 多尺度霍夫变换：和标准霍夫变换类似。

(3) 累计概率霍夫变换：是一种执行起来效率更高的霍夫变换。

① 标准霍夫变换和多尺度霍夫变换的函数：

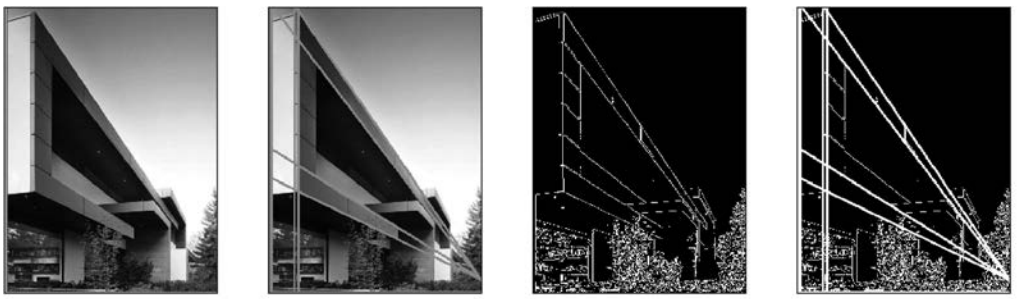
```
lines = cv2.HoughLines(image,rho,theta,threshold[,srn = 0[,stn = 0]])
```

- lines：返回值。三维的直线集合，第一维为某条直线，第二维和第三维为对应的 ρ 和 θ。
- image：输入图像，8 位单通道二值图像。
- rho：以像素为单位的距离精度。另一种形容方式是直线搜索时的步进尺寸的单位半径。
- theta：以弧度为单位的角度精度。另一种形容方式是直线搜索时的步进尺寸的单位角度。
- threshold：设定的阈值，大于此阈值的交点，才会被认为是一条直线。
- srn：默认值为 0，此时为标准霍夫变换，距离精度等于参数 rho。不为 0 时为多尺度霍夫变换，距离精度为 rho 除以 srn。
- stn：默认值为 0，此时为标准霍夫变换，角度精度等于参数 theta。不为 0 时为多尺度霍夫变换，角度精度为 theta 除以 stn。

【注意】 当参数 srn 和 stn 同时为 0 时，为标准霍夫变换；否则为多尺度霍夫变换。

【例 5.9】 用霍夫变换检测图像中的直线。

图 5.16 是对一幅彩色图像进行标准霍夫变换后的效果。



(a) 原始图像　(b) 检测到的直线画到原始图像后　(c) Canny算子边缘检测结果图　(d) 检测到的直线画到Canny算子边缘检测结果图上

图 5.16　标准霍夫变换效果

程序参考代码如下：

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```
#读入原始图像
original = cv2.imread("building - 1.jpg")
#复制原始图像
original_hough = original.copy()
#将原始图像转为灰度图
gray = cv2.cvtColor(original,cv2.COLOR_BGR2GRAY)
#Canny 算子进行边缘检测
edges = cv2.Canny(gray,50,150)
#复制检测结果图
edges_hough = edges.copy()
#进行标准霍夫变换
lines = cv2.HoughLines(edges,1,np.pi/180,220)
for line in lines:
    rho,theta = line[0][0],line[0][1]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a * rho
    y0 = b * rho
    x1 = int(x0 + 1000 * ( - b))
    y1 = int(y0 + 1000 * (a))
    x2 = int(x0 - 1000 * ( - b))
    y2 = int(y0 - 1000 * (a))
    cv2.line(original_hough,(x1,y1),(x2,y2),(0,255,0),4)#在原始图像副本上用绿色 4 像素画检测到的线
    cv2.line(edges_hough,(x1,y1),(x2,y2),(255,255,255),4)
#显示
plt.figure(figsize = (14,4))
plt.rcParams["font.sans - serif"] = ["SimHei"]
#显示原始图像及画线后的副本
plt.subplot(141),plt.imshow(cv2.cvtColor(original,cv2.COLOR_BGR2RGB)),plt.title("原始图像")
plt.xticks([])
plt.yticks([])
plt.subplot(142)
plt.imshow(cv2.cvtColor(original_hough,cv2.COLOR_BGR2RGB))
plt.title("检测到的直线画到原始图像后")
plt.xticks([]),plt.yticks([])
#显示 Canny 算子边缘检测结果图及画线后的副本
titles = ["Canny 算子边缘检测结果图","检测到的直线画到 Canny 算子边缘检测结果图上"]
images = [edges,edges_hough]
for i in range(2):
    plt.subplot(1,4,i + 3)
    plt.imshow(images[i],"gray")
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```

② 累计概率霍夫变换函数：

```
lines = cv2.HoughLinesP(image,rho,theta,threshold[,minLineLength = 0[,maxLineGap = 0]])
```

- lines：返回值。检测到的线段，每条线段由 4 个参数组成，即(x_1,y_1,x_2,y_2)，分别对应线段的起始点坐标和终止点坐标。

- image：输入图像，8 位单通道二值图像。
- minLineLength：最短线段长度，比这个设定参数短的线段不被显现，默认值为 0。
- maxLineGap：同一方向上两条线段判定为一条线段的最大允许间隔，两条线段的间隔如果小于这个值则为一条线段，默认值为 0。

使用累计概率霍夫变换函数检测图像中线段的过程参考实训 5。

2. 圆检测

霍夫变换检测圆形的原理和检测直线的原理差别不大。直线检测是将图像 $x-y$ 坐标空间的各个边缘点投影到 ρ-θ 参数空间查看聚集点，而圆的表达式为 $(x-a)^2+(y-b)^2=r^2$，有 3 个参数，即圆心坐标 (a,b) 和半径 r，所以圆的检测就是将图像坐标空间的各个边缘点投影到 $a-b-r$ 三维参数空间。如果图像坐标空间中两个不同点映射到 $a-b-r$ 参数空间后相交，即它们有一组公共的 (a,b,r)，就意味这两个点在同一个圆上，(a,b,r) 就是这个圆的圆心和半径。这就把检测圆的问题转换为在三维参数空间寻找峰值点 (a,b,r) 参数对上。以上是标准霍夫圆变换实现算法，但参数空间由二维变到三维，意味着需要更多的计算量。OpenCV 霍夫圆变换对标准霍夫圆变换做了运算上的优化，它采用霍夫梯度法，减少计算量。该算法主要分为两个阶段，从而减小参数空间的维数。第一阶段用于检测圆心，第二阶段从圆心推导出圆半径。圆心和圆半径都得到后，就能确定圆形了。

在 OpenCV 中，霍夫圆变换函数如下：

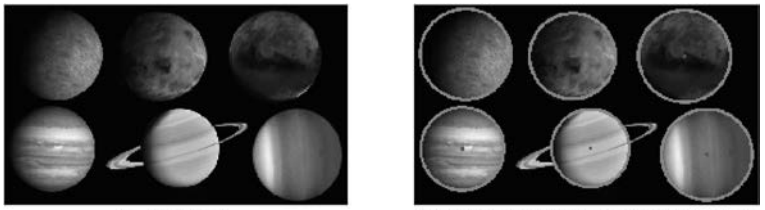
```
circles = cv2. HoughCircles ( image, method, dp, minDist [, param1 [, param2 [, minRadius [, maxRadius]]]])
```

- circles：返回值。检测到的圆的输出向量，每个向量由包含了 3 个元素的浮点向量表示，即每个向量由圆心横坐标、圆心纵坐标和圆半径表示。
- image：输入图像，8 位单通道灰度图像。
- method：检测圆时使用的算法，目前唯一支持的方法是霍夫梯度法，它的标识符为 cv2. HOUGH_GRADIENT。
- dp：累加面分辨率(大小)＝原始图像分辨率(大小)×1/dp。默认 dp＝1 时，两者分辨率相同。
- minDist：检测到的圆中圆心之间的最小距离。如果两个圆之间的距离小于给定的 minDist，则认为是同一个圆。
- param1：是参数 method 设置的霍夫梯度法对应的参数，表示传递给 Canny 边缘检测算子的高阈值，低阈值为高阈值的一半，默认值为 100。
- param2：是 method 设置的霍夫梯度法对应的参数，表示在检测阶段圆心的累加器阈值。它越小，越可以检测到更多根本不存在的圆，而它越大，能通过检测的圆就越接近完美的圆形，默认值为 100。
- minRadius：检测的最小圆半径，单位为像素，默认值为 0。
- maxRadius：检测的最大圆半径，单位为像素，默认值为 0。

【例 5.10】 使用霍夫圆变换函数检测图像中的圆。

图 5.17 是对一幅彩色图像使用霍夫圆变换函数检测圆的效果。

程序参考代码如下：



(a) 原始图像　　(b) 检测到的圆及圆心画到原始图像后

图 5.17　霍夫圆变换函数检测圆效果

```
import cv2
import matplotlib.pyplot as plt
original = cv2.imread("planets.jpg")
original_hough = original.copy()
gray = cv2.cvtColor(original,cv2.COLOR_BGRA2GRAY)
#中值平滑模糊,过滤椒盐噪声
gray1 = cv2.medianBlur(gray,7)                                    #卷积核为 7
#进行霍夫圆检测
circles = cv2.HoughCircles(gray1,cv2.HOUGH_GRADIENT,1,50,param1 = 100,param2 = 30)
                          circles = np.unitlb(np.arond(circles))
circles1 = circles[0]
for circle in circles1:
  cv2.circle(original_hough,(circle[0],circle[1]),circle[2],(0,255,0),3)
      #画圆,轮廓为绿色 3 像素
  cv2.circle(original_hough,(circle[0],circle[1]),3,(0,0,255), -1)
      #画红色圆心、半径为 3 的实心圆
#显示设置
plt.figure(figsize = (7,2))
plt.rcParams["font.sans - serif"] = ["SimHei"]
#显示原始图像及检测到的圆及圆心
b,g,r = cv2.split(original)                                       #通道拆分
original = cv2.merge((r,g,b))                                     #通道融合
b,g,r = cv2.split(original_hough)
original_hough = cv2.merge((r,g,b))
plt.subplot(121),plt.imshow(original),plt.title("原始图像")
plt.xticks([]),plt.yticks([])
plt.subplot(122),plt.imshow(original_hough),plt.title("检测到的圆及圆心画到原始图像后")
plt.xticks([]),plt.yticks([])
plt.show()
```

5.3.3　轮廓检测和绘制

通过边缘检测算子得到边缘点后,也可采用轮廓提取函数 cv2.findContours()查找检测轮廓,再通过 cv2.drawContours()函数将轮廓绘制出来。

1. 轮廓检测

cv2.findContours()函数的作用是寻找一个二值图像的轮廓。黑色表示背景,白色表示目标,即在黑色背景里寻找白色目标的轮廓。

在 OpenCV 中,轮廓提取函数如下:

```
image,contours,hierarchy = cv2.findContours(image,mode,method[,offset])
```

① image：8 位单通道二值图像。

② mode：轮廓检索的方式，有 4 种。

- cv2.RETR_EXTERNAL，只检测外部轮廓；
- cv2.RETR_LIST，检测所有轮廓且不建立层次结构；
- cv2.RETR_CCOMP，检测所有轮廓，建立两级层次结构，上面的一层为外边界，里面的一层为内孔的边界信息；
- cv2.RETR_TREE，检测所有轮廓，建立完整的层次结构。

③ method：轮廓近似的方法，有 4 种。

- cv2.CHAIN_APPROX_NONE，存储所有的轮廓点；
- cv2.CHAIN_APPROX_SIMPLE，压缩水平、垂直和对角线方向的元素，只保留该方向的终点坐标，如一个矩形轮廓只需 4 个点来保存轮廓信息；
- cv2.CHAIN_APPROX_TC89_L1，使用 teh-Chini chain 近似算法；
- cv2.CHAIN_APPROX_TC89_KCOS，使用 teh-Chini chain 近似算法。

④ offset：可选参数。轮廓点的偏移量，格式为元组，如(－5，5)表示轮廓点沿 X 轴负方向偏移 5 个像素点，沿 Y 轴正方向偏移 5 个像素点。

返回值含义如下。

① image：返回与第一个参数相同的图像。

② contours：检测到的轮廓，列表格式，每个元素为一个三维数组(其形状为(n，1，2)，其中 n 表示轮廓点个数，2 表示像素点坐标)，表示一个轮廓。

③ hierarchy：轮廓间的层次关系，为三维数组，形状为(1，n，4)，其中 n 表示轮廓总个数，4 指的是用 4 个数表示各轮廓间的相互关系。第一个数表示同级轮廓的下一个轮廓编号，第二个数表示同级轮廓的上一个轮廓编号，第三个数表示该轮廓下一级轮廓编号，第四个数表示该轮廓的上一级轮廓编号。

【注意】 OpenCV3 返回上述 3 个值，OpenCV2 返回后两个值。

2. 轮廓绘制

得到轮廓之后，通过 cv2.drawContours()函数在图像上绘制轮廓。在 OpenCV 中，该函数为：

```
cv2.drawContours(image, contours, contourIdx, color[, thickness[, lineType[, hierarchy[,
maxLevel[,offset]]]]])
```

① image：需要绘制轮廓的图像。

② contours：上述 cv2.findContours()函数检测到的轮廓。

③ contourIdx：轮廓的索引，表示绘制第几个轮廓，－1 表示绘制所有轮廓。

④ color：绘制轮廓的颜色。

⑤ thickness：可选参数，绘制轮廓线的宽度，－1 表示填充。

⑥ lineType：可选参数，绘制轮廓线型，包括 cv2.LINE_4、cv2.LINE_8(默认)和 cv2.LINE_AA，分别表示 4 邻域线、8 邻域线和抗锯齿线(可以更好地显示曲线)。

⑦ hierarchy：可选参数，层级结构，上述 cv2.findContours()函数返回的轮廓间的层次关系，配合 maxLevel 参数使用，只有绘制部分轮廓时才会用到。

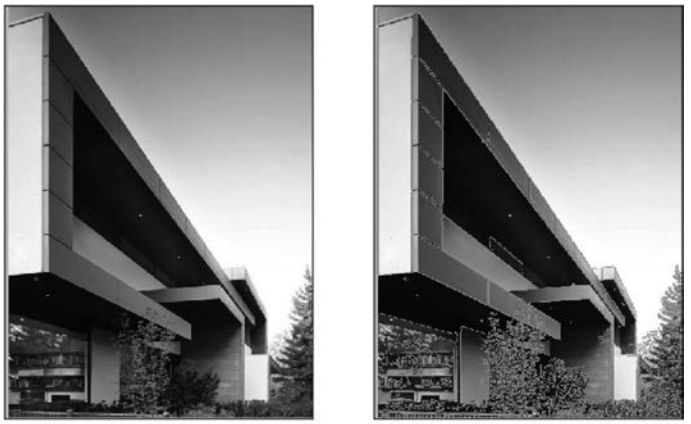
⑧ maxLevel：可选参数，只有 hierarchy 有效时才有效。等于 0 表示只绘制指定的轮廓，等于 1 表示绘制指定轮廓及其下一级子轮廓，等于 2 表示绘制指定轮廓及其所有子轮廓。

⑨ offset：可选参数，轮廓点的偏移量。

【注意】 该函数直接在参数 image 上绘制轮廓。

【例 5.11】 利用 cv2.findContours()和 cv2.drawContours()函数检测图像中的轮廓并绘制。

图 5.18 是对彩色图像 building-1.jpg 检测并绘制轮廓的效果图。



(a) 原始图像　　(b) 检测并绘制轮廓的图像

图 5.18　检测并绘制轮廓效果

程序参考代码如下：

```
import cv2
import matplotlib.pyplot as plt
original = cv2.imread("building - 1.jpg")
original1 = original.copy()
gray = cv2.Canny(original,50,150)
contours,hierarchy = cv2.findContours(gray,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
cv2.drawContours(original1,contours, - 1,(0,255,0),1) # 以绿色 1 像素,绘制检测到的所有轮廓
plt.rcParams["font.sans - serif"] = ["SimHei"]
titles = ["原始图像","检测并绘制轮廓的图像"]
images = [original,original1]
plt.figure(figsize = (8,5))
for i in range(2):
    plt.subplot(1,2,i + 1)
    plt.imshow(cv2.cvtColor(images[i],cv2.COLOR_BGR2RGB))
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```

思考：如何利用 cv2.findContours()和 cv2.drawContours()函数完成车牌识别？

5.4　区域分割原理与实现

区域分割是将图像按照相似性准则分成不同的区域，主要包括区域生长法、区域分裂合并法和分水岭算法等几种方法。

5.4.1 区域生长法

区域生长法的基本思想是将具有相似性质的像素点合并到一起。选定一组种子像素(对每一个要划分的区域各指定一个种子作为生长起点,多个要划分的区域就指定多个种子),对组中的每个种子完成以下操作:从一个种子像素开始,将种子像素邻域里满足预先定义的生长准则的像素与种子像素合并,再将新合并进来的像素作为新的种子继续进行合并,直到找不到符合条件的新像素为止。生长准则指种子像素与邻域像素满足的相似性,判据可以是灰度值、纹理、颜色等。

区域生长法的三要素如下。

(1) 选择合适的种子像素(种子的选取需要具体情况具体分析,可人工选择,也可通过一些方法自动选取)。

(2) 确定在生长过程中能将相邻像素包含进来的准则(一般用灰度差值小于某个阈值表示,不同的判定准则可能会产生不同的分割结果)。

(3) 确定生长停止的条件。

区域生长法实现的步骤如下。

(1) 获取一个种子。

(2) 以该种子像素为中心,考察它的邻域像素是否满足生长准则(如生长准则为灰度差小于阈值 T),将邻域内的像素逐个与它比较,如果满足,则将它们合并。

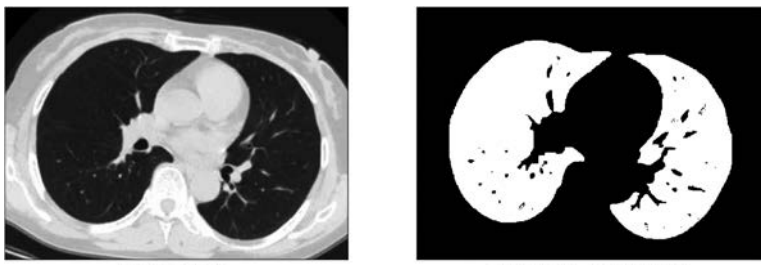
(3) 以新合并的像素为种子,返回步骤(2)检查新像素的邻域,直到区域无法进一步扩张。

(4) 获取下一个种子,重复步骤(2)和(3)直到所有种子全部完成生长。

【注意】 当区分多目标时,选取多个种子,然后将每个种子得到的分割区域合并,即可分割出多目标。

【例 5.12】 采用区域生长法将一幅灰度图像分割为二值图像。

本例题采用人工选择种子,目标是得到左、右肺。根据肺在图像中的位置,设定两个种子点(左、右肺各一个)。以灰度值作为像素点的特征,生长准则是灰度差值小于阈值 7。分割出的前景区域用白色表示,背景区域用黑色表示。效果见图 5.19。



(a) 原始图像　　(b) 区域生长结果图像

图 5.19 区域生长效果图

程序参考代码如下:

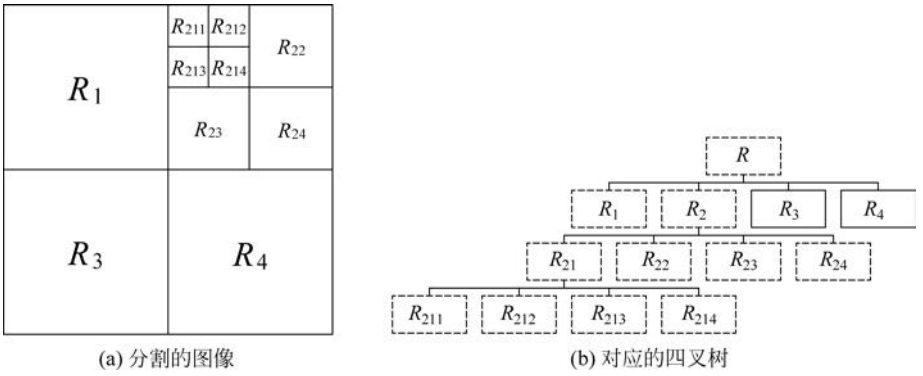
```
import cv2
import numpy as np
```

```
import matplotlib.pyplot as plt
#区域生长函数
def regionGrow(img,seeds,threshold,p):
    height,weight = img.shape
    seedMark = np.zeros(img.shape)        #创建与原始图像大小相同的图像,作为结果图像。初始时像素值都是0
    seedList = []
    for seed in seeds:
        seedList.append(seed)
        seedMark[seed[0],seed[1]] = 255
#8邻域
        if p == 1:
            connects = [(-1,-1),(0,-1),(1,-1),(-1,0),(1,0),(-1,1),(0,1),(1,1)]
#4邻域
        else:
            connects = [(0,-1),(-1,0),(0,1),(1,0)]
#seedList内无种子时停止生长
        while(len(seedList)>0):
#取出第一个种子
            currentPoint = seedList.pop(0)
#检测种子点8邻域内满足生长准则的情况
            for i in range(8):
                tmpX = currentPoint[0] + connects[i][0]
                tmpY = currentPoint[1] + connects[i][1]
#如果出边界就检测下一点
                if tmpX < 0 or tmpX >= height or tmpY < 0 or tmpY >= weight:
                    continue
#没出边界的就检测是否满足生长准则并且没被标记过
                if abs(int(img[currentPoint[0],currentPoint[1]]) - int(img[tmpX,tmpY]))<threshold and seedMark[tmpX,tmpY] == 0:
#满足条件的设为白色并将其作为种子加入到检测列表中
                    seedMark[tmpX,tmpY] = 255
                    seedList.append((tmpX,tmpY))
    return seedMark
original = cv2.imread("lung window.jpg",0)
#初始种子坐标
seeds = [(180,100),(300,370)]
#调用regionGrow()函数
threshold = 7
neighbourhood = 1
result = regionGrow(original,seeds,threshold,neighbourhood)
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
titles = ["原始图像","区域生长结果图像"]
images = [original,result]
plt.figure(figsize = (14,6))
for i in range(2):
    plt.subplot(1,2,i + 1)
    plt.imshow(images[i],"gray")
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```

5.4.2 区域分裂合并法

区域分裂合并法的基本思想是，先确定一个分裂合并的相似性准则，然后通过分裂将不同特征的区域分离，再通过合并将相同特征的区域并到一起，从而实现图像的分割。

算法的具体过程为：令 R 表示整幅图像区域，P 表示某种相似性准则。假设以像素的灰度值为特征，制定相似性准则 P 为灰度值差小于阈值 T。如果 $P(R)=\text{False}$，即区域 R 不满足相似性准则，就将图像等分为 4 个区域。分别检测分得的 4 个区域的 $P(R_i)$ 值，如果某个区域的值是 False，就将该区域再次分为 4 个区域，如此不断继续下去。直到对任何区域 R_i，有 $P(R_i)=\text{True}$，表示区域 R_i 已经满足相似性准则或是已经为单个像素，此时分裂结束。该过程可以用四叉树形式表示，树的根对应于整幅图像区域，每个节点对应于划分的子区域，如图 5.20 所示，图中只对 R_2 进行了细分。分裂后，如果有满足相似性准则的邻近区域，即 $P(R_j \cup R_k)=\text{True}$ 时，两个相邻的区域 R_j 和 R_k 就进行合并。合并的区域可以大小不同。当无法再进行合并时操作停止。



(a) 分割的图像　　(b) 对应的四叉树

图 5.20　四叉树算法示意图

总结，区域分裂合并法主要有以下 4 个步骤。

(1) 制定相似性准则 P。

(2) 分裂：满足 $P(R_i)=\text{False}$ 的任何区域 R_i，分裂为 4 个相等的不相交区域，直到不能进一步分裂为止。

(3) 合并：对满足条件的 $P(R_j \cup R_k)=\text{True}$ 的任意两个相邻的区域 R_j 和 R_k 进行合并，直到无法进一步合并为止。

(4) 结束。

图 5.21 是一个分裂合并的实例。假设相似性准则 P 为区域内灰度值相等。对于整幅图像，如图 5.21(a)所示，有黑色和白色，灰度值不相等，不满足相似性准则，进行第一次分裂，把图像等分成了 4 个区域，如图 5.21(b)所示；分得的 4 个区域，每个区域都不满足灰度值相等这一准则，所以每一区域又都等分为 4 个区域，如图 5.21(c)所示；对前述分得的区域，发现只有右侧中间两个区域不满足相似性准则，需要继续分裂，其余都满足，都不需要再进行分裂，如图 5.21(d)所示；对不满足相似性准则的两个区域 4 等分之后，发现所有区域都满足相似性准则了，则分裂结束。接下来开始进行合并，对满足相似性准则的相邻区域进行合并，得到图像分割的结果，如图 5.21(e)所示。

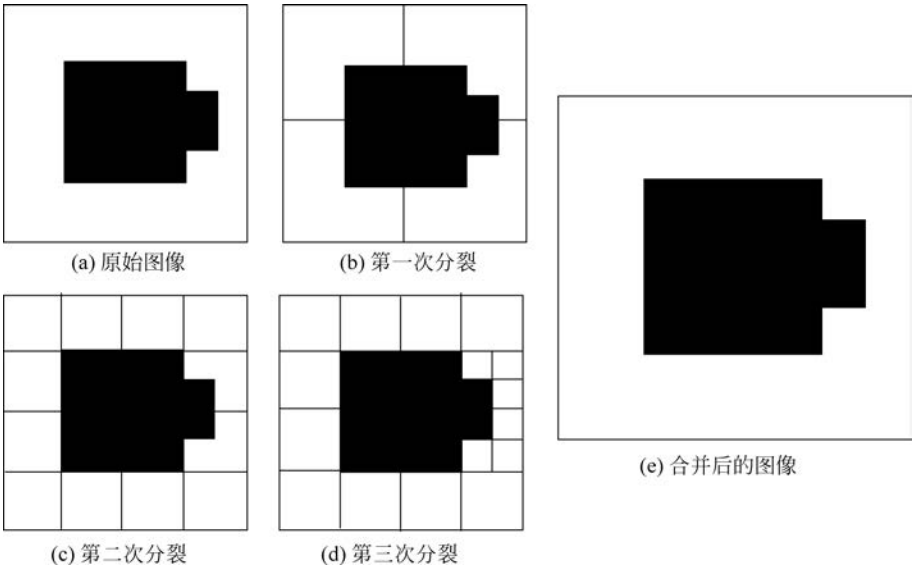


图 5.21　区域分列合并实例

区域分裂合并算法的难点在于分裂与合并的准则不易确定，选用合适的准则 P 对于提高图像分割质量十分重要。目前，均方误差最小、F 检测等是最常用的准则。

5.4.3　分水岭算法

分水岭算法是一种基于拓扑理论的数学形态学的分割方法，它根据分水岭的构成来考虑图像的分割。分水岭算法把图像看作测地学上的拓扑地貌，图像中每一点像素的灰度值表示该点的海拔高度，每个局部极小值及其影响区域称为集水盆，分水岭就是集水盆的边界形成的，对应图像边缘。分水岭变换可以保证分割区域的连续性和封闭性。

下面采用模拟浸入过程来说明分水岭的概念和形成。在每一个局部极小值表面，打一个小孔，然后让模型慢慢浸入水中，随着浸入的加深，水淹没极小值及周围的区域，各个极小值及其影响域就是相应的集水盆，两个集水盆相遇的界限，就是分水岭。

图 5.22 显示了形成分水岭的过程。图 5.22(a)是水刚刚从一个极小值处涌出，在第一个集水盆慢慢上升，图 5.22(d)是其俯视图。图 5.22(b)是水涌出到一定程度，在两个集水盆中上升，图 5.22(e)是其俯视图。图 5.22(c)是两个集水盆将连未连时的截面图，图 5.22(f)是其俯视图。此时两个集水盆的边界就是分水岭，就是希望得到的原始图像的边缘。

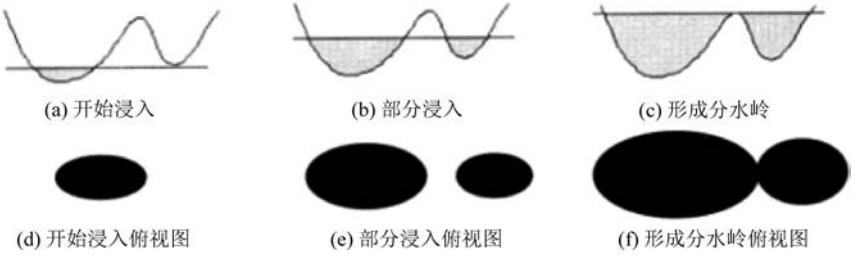


图 5.22　分水岭形成示意图

传统的分水岭算法对噪声等非常敏感，图像中的噪声点或其他干扰因素等的存在，常产生过度分割现象。过度分割就是原本完整的物体被分割成很多细小的区域，从而导致分割后的图像不能将图像中有意义的区域表示出来。

为了减少过度分割造成的影响，OpenCV 采用了一个基于掩模的分水岭算法，使用 cv2.watershed(img,markers)函数实现。img 表示输入 8 位 3 通道图像；markers 参数表示与原始图像大小相同的标记图，用于标记图像不同区域，每个区域有一个自己唯一的编号，确定区域的编号用正整数值 1、2、3、……分别表示，作为后续区域分割的种子，而不确定的区域用 0 表示。在使用分水岭函数之前，必须先通过一定的操作得到第二个参数 markers，它是 OpenCV 分水岭算法的关键。函数依据种子信息，对图像上的像素点根据分水岭算法规则进行判断，并对每个像素点的区域归属进行划定，直到处理完图像上所有像素点。函数运行后，返回值是一个与输入大小相同的图像，其中灰度值为－1 的像素点就是想要的边界。

利用 OpenCV 分水岭算法实现图像自动分割的步骤如下。

(1) 加载原始图像。

(2) 阈值分割，将图像分割为黑白两部分。

(3) 对图像进行开运算，即先腐蚀再膨胀(简单理解腐蚀就是使目标区域“变小”，膨胀就是使目标区域“变大”)，目的是消除噪声。

(4) 对上一步的结果进行膨胀，得到确定背景区域。

(5) 通过距离变换 cv2.distanceTransform( )函数，获取确定前景区域。距离变换的结果是一幅灰度级图像，即距离图像，图像中每个像素的灰度值为该像素与距其最近的背景像素间的距离。由于目标中心像素距离背景最远，所以值最大，就最亮。再通过设定合理的阈值对距离变换后的图像进行二值化处理，就可获得确定前景区域。

(6) 背景区域和前景区域相减，得到不确定是背景还是前景的区域，即未知区域，要求的分水岭就在这一区域。

(7) 得到 OpenCV 分水岭函数需要的 markers。OpenCV 分水岭函数需要的 markers 对确定分类的区域(无论是前景还是背景)使用不同的正整数标记，对不确定的区域使用 0 标记。而利用 cv2.connectedComponents( )函数得到的标记图会把背景标记为 0，其他的对象用从 1 开始的正整数标记，所以要通过操作将其转换为 OpenCV 分水岭函数需要的 markers 的标记形式。

(8) 最后使用分水岭函数 cv2.watershed(img,markers)。返回图像中灰度值为－1 的像素点就是所求边界。

【例 5.13】 利用分水岭算法实现图像分割。

图 5.23 是对一幅彩色图像利用分水岭算法实现图像分割的效果图。图中用红色标出了图像中两部分的分割线。

图 5.23 分水岭算法效果

程序参考代码如下：

```
import cv2
import numpy as np
```

```
original = cv2.imread("landscape.jpg")
#转换为灰度图
gray = cv2.cvtColor(original,cv2.COLOR_BGR2GRAY)
#阈值分割
ret,dst = cv2.threshold(gray,0,255,cv2.THRESH_OTSU)
#对图像进行"开运算"(先腐蚀再膨胀),迭代3次,目的是消除噪声
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(dst,cv2.MORPH_OPEN,kernel,iterations = 3)
#对"开运算"的结果进行膨胀,得到确定背景区域
sure_bg = cv2.dilate(opening,kernel,iterations = 5)
#通过distanceTransform,得到确定前景区域
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)
ret,sure_fg = cv2.threshold(dist_transform,0.2 * dist_transform.max(),255,cv2.THRESH_
BINARY)
#用sure_bg与sure_fg相减,得到未知区域,边界就在这一区域
sure_fg = np.uint8(sure_fg)
unknow = cv2.subtract(sure_bg,sure_fg)
#得到OpenCV分水岭函数需要的markers
ret,markers_connect = cv2.connectedComponents(sure_fg,connectivity = 8)
markers_OpenCV = markers_connect + 1        #转换为OpenCV分水岭函数使用的markers形式
markers_OpenCV[unknow == 255] = 0
#调用分水岭函数
markers_watershed = cv2.watershed(original,markers_OpenCV)
original[markers_watershed == -1] = [0,0,255] #markers_watershed中灰度值为-1的像素点即
边界,标为红色
#显示效果图
cv2.imshow("Result",original)
cv2.waitKey(0)
cv2.destroyAllWindows()                     #销毁所有窗口
```

也可设置提取目标,如想要提取图5.23中的草地和树木,效果图及参考代码见实训6。

5.5 基于特定理论的图像分割

随着各学科新理论和新方法的提出,出现了与一些特定理论、方法相结合的图像分割方法,主要有基于聚类分析的图像分割方法、基于模糊集理论的图像分割方法、基于神经网络的图像分割方法、基于图论的图像分割方法、基于遗传算法的图像分割方法、基于小波变换的图像分割方法等。

5.5.1 基于聚类分析的图像分割方法

聚类分析是数据挖掘的一个重要研究领域。在众多的分割算法中,基于聚类分析的图像分割方法是图像分割领域中一类极其重要和应用相当广泛的算法。长久以来,专家学者们关于聚类算法及应用的探讨从未间断。聚类算法是将一组分布未知的数据进行分类,使得同一类中的数据相似度尽可能大,而不同类的数据相似度尽可能小。聚类算法主要分为两大类,即硬聚类和软聚类。硬聚类是指数据集中每个样本只能划分到一个簇的算法,最具代表性的如 K 均值(K-Means)算法。软聚类也称模糊聚类,指算法可以将一个样本划分到一个或者多个簇。常见的算法是模糊 C 均值(Fuzzy C-Means,FCM)算法。软聚类建立了

样本对类别的不确定描述，更能真实反映客观世界，成为聚类分析的主流。

K 均值算法的基本思想是以空间中的 K 个点为中心进行聚类，将空间中的每个点归类到离自己距离最近的中心所在的类，形成 K 个类。采用迭代方式，逐次更新各聚类中心的值，直至得到最好的聚类结果。最终实现各聚类本身尽可能紧凑，而各聚类之间尽可能分开。具体步骤如下。

(1) 从 n 个数据对象任选 K 个对象作为初始聚类中心。

(2) 对于所剩其他对象，根据它们与这些聚类中心的距离，分别将它们归类给与其距离最近的聚类。

(3) 计算每个所获新聚类的聚类中心(新聚类中所有点的坐标平均值)。

(4) 如果新的聚类中心与老的聚类中心的距离小于某一阈值，就表示新的聚类中心位置变化不大，收敛稳定，认为聚类已达到了期望的结果，算法终止。

(5) 否则就认为新的聚类中心与老的聚类中心变化很大，就继续迭代步骤(2)～(4)。

该算法的优点是原理简单、实现容易；处理大数据集时非常高效且伸缩性较好。缺点主要是 K 值难以估计；不同的初始聚类中心可能导致完全不同的聚类结果；结果只能保证局部最优；对于离群点和孤立点敏感；对于非凸数据集或类别规模差异太大的数据效果不好；需样本存在均值(限定数据种类)等。

模糊 C 均值算法是在模糊数学基础上对 K 均值算法的推广，是一种柔性的模糊划分。它不像 K 均值聚类那样非此即彼地认为每个点只能属于某一类，而是赋予每个点一个对各类的隶属度，通过隶属度值大小将样本点归类。具体操作时通过不断迭代优化各点与所有类中心相似性的目标函数，得到每个数据点对所有类中心的隶属度，用隶属度获取局部极小值，确定每个数据点属于哪个聚类，从而得到最优聚类。

模糊 C 均值算法因算法简单、收敛速度快、能处理大数据集、解决问题范围广、易于应用计算机实现等特点，在图像分割领域得到广泛应用。但在应用 FCM 时，聚类个数和模糊指数的选取至关重要，只有选取正确才能得到好的聚类效果。

5.5.2 基于模糊集理论的图像分割方法

模糊集合论作为描述和处理具有不确定性(即模糊性)事物和现象的一种数学手段，更接近人类真实思维和决策，更符合客观实际，非常适合处理图像分割问题。这是由于人的视觉特性和数字图像本身所具有的不确定性使得图像分割问题是典型的结构不良问题，将模糊集理论应用于图像分割是针对图像不确定性的非常有效的方法，能得到更好的分割效果。

模糊技术在图像分割中常常和现有的许多图像分割技术相结合，形成基于模糊理论的图像分割方法，主要包括模糊阈值分割方法、模糊聚类分割方法、模糊神经网络分割方法等。

模糊阈值分割法通过计算图像的模糊率或模糊熵来选取图像分割阈值。模糊程度由模糊率函数确定，当模糊率最低时，分割效果最好。其中模糊率与隶属函数相关，模糊数学的基本思想是隶属度的思想。应用模糊数学方法建立数学模型的关键是建立符合实际的隶属函数。利用不同的 S 形隶属度函数来定义模糊目标，通过优化过程最后选择一个具有最小模糊率的 S 形函数。用该函数增强目标及属于该目标的像素之间的关系，这样得到的 S 形函数的交叉点为阈值分割需要的阈值。传统的模糊熵阈值分割通过计算图像的最大模糊熵确定阈值。然而，将图像分为黑白两色很多时候并不能很好地区分这两色内有区别的目标，这时就需要多阈值分割。

5.5.3 基于神经网络的图像分割方法

人工神经网络简称为神经网络或连接模型，是对人脑或自然神经网络若干基本特性的抽象和模拟。人工神经网络以对大脑的生理研究成果为基础，其目的在于模拟大脑的某些机理与机制，实现某个方面的功能。近年来，人工神经网络识别技术引起广泛关注，并应用于图像分割。基于神经网络的图像分割方法的基本思想是通过训练多层感知机得到线性决策函数，然后用决策函数对像素进行分类来达到分割图像的目的。神经网络存在巨量连接，容易引入空间信息，解决了图像中的噪声和不均匀问题。

基于神经网络的分割算法主要有 BP(Back Propagation)神经网络算法、RBF(Radial Basis Function)神经网络算法、Hopfield 神经网络算法等。

5.5.4 基于图论的图像分割方法

图论理论能够充分考虑图像像素之间的空间位置关系，可以兼顾边界和区域两方面的信息，减少图像离散化造成的误差，因而能够得到良好的分割效果。近年来，基于图论的图像分割方法受到广泛关注。该方法的基本思想是将图像表示成图论中的网络图，利用图论的相关特性并结合有关技术，通过对网络图进行一系列操作和计算处理，完成对图像的分割。网络图和图像之间有很好的对应关系，网络图的节点对应图像的像素，网络图的边对应图像中像素的相邻性，网络图边的权对应图像中相邻像素的关系，可以表示相邻两个像素的相似性，也可表示相邻两个像素间边的连接强度。图论的相关特性与图像分割具有某种共性，因此可以基于图论的特性对图像进行分割。图论中节点间的最短路径可完成类似基于边缘的图像分割，图论中的最小分割可以实现类似基于区域的图像分割效果。

5.5.5 基于遗传算法的图像分割方法

遗传算法又叫基因算法，是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法，具有简单、实用性强、稳定性好、适用于并行处理等显著特点，在很多领域得到广泛应用，成功地解决了各种类型的优化问题。在分割复杂图像时，人们往往采用多参量进行信息融合，在多参量参与最优值的求取过程中，优化计算是最重要的。把自然进化的特征应用到计算机算法中，能解决很多困难。遗传算法的出现为解决这类问题提供了有效的方法，它不仅可以得到全局最优解，而且大大缩短了计算时间。

5.5.6 基于小波变换的图像分割方法

小波变换是一种时、频两域分析工具，它在时间域和频率域都具有良好的局部特性，能有效地从信号中提取信息。小波变换具有多分辨率(也叫多尺度)特性，能够在不同分辨率上对信号进行分析，因此在图像处理和分析等许多方面得到应用。

基于小波变换的图像阈值分割方法的基本思想是首先由二进制小波变换将图像的直方图分解为不同层次的小波系数；然后依据给定的分割准则和小波系数选择阈值；最后利用阈值标出图像分割的区域。基于小波变换的边缘检测图像分割方法的基本思想是取小波函数作为平滑函数的一阶导数或二阶导数，利用信号的小波变换的模值在信号突变点处取局

部极大值或过零点的性质来提取信号的边缘点。

实训 4 阈值分割

（1）通过编写程序体会 OpenCV 提供的阈值处理函数 cv2.threshold( )的 type 参数，观察 type 参数不同时得到的图像效果。type 分别取 cv2.THRESH_BINARY、cv2.THRESH_BINARY_INV、cv2.THRESH_TRUNC、cv2.THRESH_TOZERO 和 cv2.THRESH_TOZERO_INV。效果如图 5.24 所示。

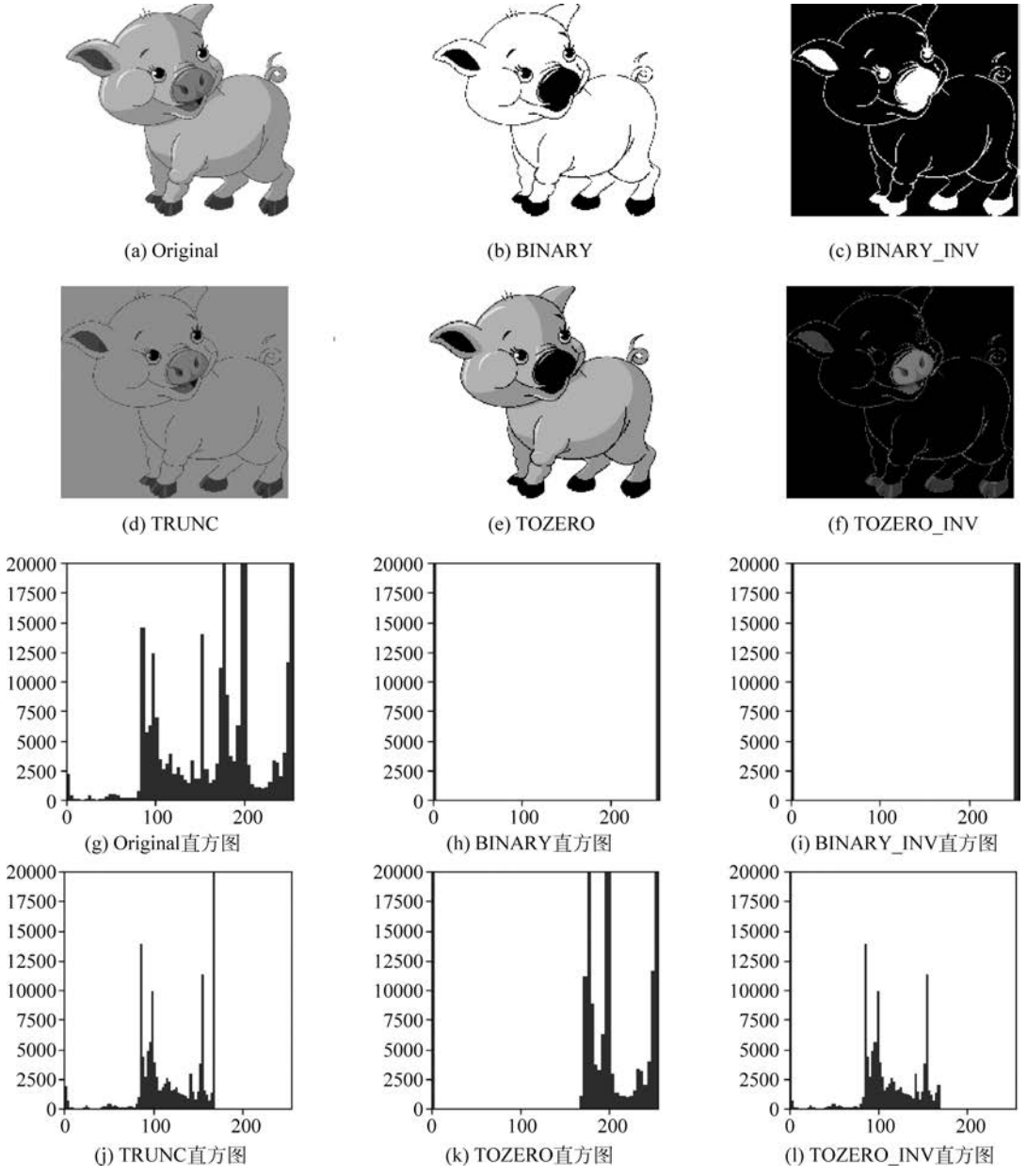


图 5.24 threshold()函数的 type 参数取不同值时的效果图和直方图

程序参考代码如下：

```
import cv2                          #导入 opencv 库
import numpy as np                  #导入 numpy 库
import matplotlib.pyplot as plt #导入 matplotlib 库的 pyplot 模块
from matplotlib.colors import NoNorm
#以灰度模式读入原始图像
original = cv2.imread("pig.jpg",0)
#threshold 函数,type 值不同
ret,dst1 = cv2.threshold(original,170,255,cv2.THRESH_BINARY)
ret,dst2 = cv2.threshold(original,170,255,cv2.THRESH_BINARY_INV)
ret,dst3 = cv2.threshold(original,170,255,cv2.THRESH_TRUNC)
ret,dst4 = cv2.threshold(original,170,255,cv2.THRESH_TOZERO)
ret,dst5 = cv2.threshold(original,170,255,cv2.THRESH_TOZERO_INV)
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
plt.rcParams.update({"font.size": 8})
titles1 = ["Original","BINARY","BINARY_INV","TRUNC","TOZERO","TOZERO_INV"]
titles2 = ["Original 直方图","BINARY 直方图","BINARY_INV 直方图","TRUNC 直方图","TOZERO 直
方图","TOZERO_INV 直方图"]
images = [original,dst1,dst2,dst3,dst4,dst5]
plt.figure(figsize = (18,6))
#显示图像
for i in range(6):
    plt.subplot(2,6,i + 1)
    plt.imshow(images[i],"gray",norm = NoNorm())
    plt.title(titles1[i])
    plt.axis("off")
#显示直方图
for i in range(6,12):
    plt.subplot(2,6,i + 1)
    a = np.array(images[i - 6]).flatten()
    plt.hist(a,bins = 64,color = "blue")
    plt.title(titles2[i - 6])
    plt.xlim(0,255)
    plt.ylim(0,20000)
plt.show()
```

(2) 利用 Otsu 算法和自适应法对同一图像进行阈值分割，体会全局阈值分割和可变阈值分割的效果对比。

Otsu 算法进行阈值分割时，只有一个阈值，把图像中每个像素的灰度值与该阈值进行比较，这样往往会过滤掉很多信息，导致细节不明显。而自适应法对每一个像素点单独计算阈值，即每个像素点的阈值都是不同的，就是将该像素点周围 blockSize * blockSize 区域内的像素取均值或高斯加权平均，然后减去一个常数 C，从而得到该点的阈值，这样能避免整张图像亮度分布不均，能保留更多的细节性信息。效果如图 5.25 所示。

程序参考代码如下：

```
import cv2
import matplotlib.pyplot as plt
original = cv2.imread("basket.jpg",0)
ret1,dst1 = cv2.threshold(original,0,255,cv2.THRESH_OTSU + cv2.THRESH_BINARY_INV)
dst2 = cv2.adaptiveThreshold(original,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY_INV,
```

(a) 原始图像　(b) Otsu算法阈值分割后　(c) 自适应法阈值分割后

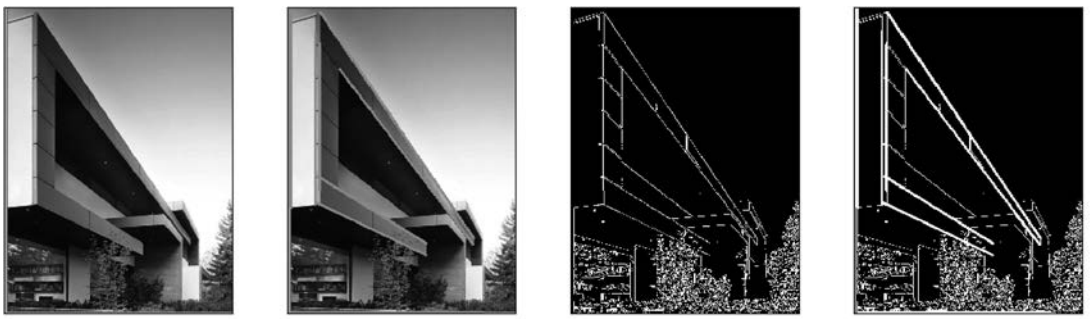
图 5.25　Otsu 算法和自适应法阈值分割效果对比

```
41,16)
#显示
plt.rcParams["font.sans-serif"] = ["SimHei"]
image = [original,dst1,dst2]
title = ["原始图像","Otsu 算法阈值分割后","自适应法阈值分割后"]
plt.figure(figsize = (14,4))
for i in range(3):
    plt.subplot(1,3,i + 1)
    plt.imshow(image[i],"gray")
    plt.title(title[i])
    plt.axis("off")
plt.show()
```

实训 5　边缘分割

利用累计概率霍夫变换函数检测彩色图像 building-1.jpg 中的线段。

图 5.26 是检测后的效果。



(a) 原始图像　(b) 检测到的线段画到原始图像后　(c) Canny算子边缘检测结果　(d) 检测到的线段画到Canny算子边缘检测结果图上

图 5.26　累计概率霍夫变换效果

程序参考代码如下：

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#读入原始图像
original = cv2.imread("building-1.jpg")
```

```
#复制原始图像
original_hough = original.copy()
#将原始图像转为灰度图
gray = cv2.cvtColor(original,cv2.COLOR_BGR2GRAY)
#Canny 算子进行边缘检测
edges = cv2.Canny(gray,50,150)
#复制检测结果图
edges_hough = edges.copy()
#进行累计概率霍夫变换
lines = cv2.HoughLinesP(edges,1,np.pi/180,200,minLineLength = 200,maxLineGap = 10)
for line in lines:
    x1,y1,x2,y2 = line[0]
    cv2.line(original_hough,(x1,y1),(x2,y2),(0,255,0),4)#在原始图像副本上用绿色 4 像素画检测到的线段
    cv2.line(edges_hough,(x1,y1),(x2,y2),(255,255,255),4)
#显示
plt.figure(figsize = (14,4))
plt.rcParams["font.sans - serif"] = ["SimHei"]
#显示原始图像及画线段后的副本
plt.subplot(141),plt.imshow(cv2.cvtColor(original,cv2.COLOR_BGR2RGB)),plt.title("原始图像")
plt.xticks([]),plt.yticks([])
plt.subplot(142),plt.imshow(cv2.cvtColor(original_hough,cv2.COLOR_BGR2RGB)),plt.title("检测到的线段画到原始图像后")
plt.xticks([]),plt.yticks([])
#显示 Canny 算子边缘检测结果图及画线段后的副本
titles = ["Canny 算子边缘检测结果图","检测到的线段画到 Canny 算子边缘检测结果图上"]
images = [edges,edges_hough]
for i in range(2):
    plt.subplot(1,4,i + 3)
    plt.imshow(images[i],"gray")
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```

实训 6　区域分割

利用分水岭算法实现图像中目标的提取。原始图像是一幅风景图，假设要提取草地和树木，只需先使用分水岭算法分割图像，对于非目标区域用其他颜色填充即可。程序代码与利用分水岭算法实现图像分割基本相同，只需在最后利用 original[markers_watershed==2]=[255,255,255]语句将背景区设成白色或其他颜色即可，效果如图 5.27 所示。

程序参考代码如下：

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
original = cv2.imread("landscape.jpg")
original_copy = original.copy()
#转换为灰度图
gray = cv2.cvtColor(original,cv2.COLOR_BGR2GRAY)
```

(a) 原始图像　　(b) 提取目标后的图像

图 5.27　分水岭算法提取目标

```
#阈值分割
ret,dst = cv2.threshold(gray,0,255,cv2.THRESH_OTSU)
#对图像进行"开运算"(先腐蚀再膨胀),迭代 3 次,目的是消除噪声
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(dst,cv2.MORPH_OPEN,kernel,iterations = 3)
#对"开运算"的结果进行膨胀,得到确定背景区域
sure_bg = cv2.dilate(opening,kernel,iterations = 5)
#通过 distanceTransform 得到确定前景区域
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)
ret,sure_fg = cv2.threshold(dist_transform,0.2 * dist_transform.max(),255,cv2.THRESH_
BINARY)
#用 sure_bg 与 sure_fg 相减,得到未知区域,边界就在这一区域
sure_fg = np.uint8(sure_fg)
unknow = cv2.subtract(sure_bg,sure_fg)
#得到 OpenCV 分水岭函数需要的 markers
ret,markers_connect = cv2.connectedComponents(sure_fg,connectivity = 8)
markers_OpenCV = markers_connect + 1          #转换为 OpenCV 分水岭函数使用的 markers 形式
markers_OpenCV[unknow == 255] = 0
#调用分水岭函数
markers_watershed = cv2.watershed(original,markers_OpenCV)
original[markers_watershed == 2] = [255,255,255]
#显示
plt.figure(figsize = (12,4))
plt.rcParams["font.sans - serif"] = ["SimHei"]
titles = ["原始图像","提取目标后的图像"]
images = [original_copy,original]
for i in range(2):
    plt.subplot(1,2,i + 1)
    plt.imshow(cv2.cvtColor(images[i],cv2.COLOR_BGR2RGB))
    plt.title(titles[i])
    plt.xticks([])
    plt.yticks([])
plt.show()
```