C#数字图像处理算法典型实例

赵春江　编著

人 民 邮 电 出 版 社
北　京

内 容 提 要

本书精选数字图像处理领域中的一些应用实例，以理论和实践相结合的方式，系统地介绍了如何使用C#进行数字图像处理。

全书共11章，分别讲述了图像的点运算、几何运算、数学形态学图像处理方法、频率变换、图像平滑与去噪、边缘检测、图像分割、图像压缩编码和彩色图像处理等相关技术。本书的光盘中附有相关章节的实现代码，可供广大的读者参考、阅读。

本书内容丰富，叙述详细，实用性强，适合于数字图像处理工作者阅读参考。

C# 数字图像处理算法典型实例

◆ 编　　著　赵春江
责任编辑　屈艳莲
执行编辑　黄　焱

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷厂印刷

◆ 开本：787×1092　1/16
印张：23
字数：565千字　　2009年1月第1版
印数：1－　000册　　2009年1月北京第1次印刷

ISBN 978-7-115-19358-2/TP

定价：　　元（附光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前　言

写作背景

图像处理是对图像进行分析、加工和处理，使其满足视觉、心理以及其他要求的技术。目前大多数的图像是以数字形式存储，因而图像处理在很多情况下是指数字图像处理。图像处理是信号处理的子类，另外它与计算机科学、人工智能等领域也有密切的关系。自从 20 世纪 60 年代以来，数字图像处理的理论和方法不断完善，已经在宇宙探测、遥感、生物医学、农业生产、军事、公安、办公自动化、视频和多媒体系统等领域得到了广泛应用，并显示出广阔的应用前景，它已成为计算机科学、信息科学、生物学、医学等学科研究的热点。

为了实现和开发数字图像处理的算法，目前主流的应用软件是 C++。在过去的二三十年间，C++已经成为在商业软件的开发领域中使用最广泛的语言。数字图像处理可以被看成是二维数组的运算，应用 C++来完成正是利用了 C++的灵活多变、快速高效的运行能力和面向对象的编程思想等优点。

然而 C++在给程序员带来灵活性的同时，也牺牲了开发效率。用 C++开发应用程序往往需要较长的时间，用它来编写数字图像处理算法尤其如此。对于初学者来说，既要精通图像处理的各种算法，又要熟练掌握该种语言的语法结构，似乎是一件很难的事情。尽管有各种相关书籍可以参考，但对于不是很精通 C++的人来说，仍然会感到一头雾水，无从下手。C++（尤其是 Visual C++）指针的运用、各种自定义的数据类型、函数之间的相互调用、程序的流程走向等问题，始终困扰他们。即使熟悉图像处理的算法，有时也无法用 C++顺利地编写出程序来，不能直观地看出图像处理的效果，更进一步地影响了继续研究数字图像处理算法的信心。

C#是由微软公司开发的一种面向对象的新型编程语言，它是从 C 和 C++中派生出来的，保留了 C/C++原有的强大功能，并且继承了 C/C++的灵活性。同时，由于是 Microsoft 公司的产品，它又同 Visual Basic 一样具有简单的语法结构和高效的开发能力，可以使程序员快速地编写出基于.NET 平台的应用程序。

编写同一段代码，C#与 C++相比不仅开发周期短、代码量小，而且可读性好。C#同样适用于数字图像处理的算法开发，对于那些刚刚接触数字图像处理，对编程语言不是很熟悉，而又急需一种编程语言来实现、验证和开发某种算法的读者来说，C#是一个不错的选择。

然而到目前为止，市面上关于用 C#开发数字图像处理的书籍少之又少，这不能不说是一丶遗憾。但用 C#开发数字图像处理的能力是无需质疑的。许多工程师都开发出了基于 C#的图像处理软件，主要讲解 C#语言的网站都有图像处理的专栏或详细介绍。对于初学者，它的简洁易懂、上手快、开发周期短等优点，具有更大的吸引力，不会因为编程语言上的障碍，而放弃对数字图像处理的研究。

本书系统而全面地介绍如何利用 C#这一高级语言来实现数字图像处理中各种常用的算法，使读者把主要精力集中到图像处理算法本身的研究中去，既快又准确地运用高级语言实现算法。

本书主要内容

本书共分 11 章，分别介绍了数字图像处理中最常用的一些处理方法，如绘制直方图、灰度线性变换、灰度拉伸、直方图均衡化、直方图匹配、图像平移、图像镜像、图像旋转、数学形态学中的腐蚀运算、膨胀运算、开运算与闭运算、击中击不中变换、图像的频域变换、图像的噪声模型、中值滤波、均值滤波、灰度形态学滤波、小波变换滤波、高斯低通滤波、统计方法滤波、一阶导数边缘检测、二阶导数边缘检测、Canny 边缘检测、小波变换边缘检测、灰度形态学边缘检测、金字塔边缘检测、阈值分割法、特征空间聚类分割法、松弛迭代分割法、Hough 变换、哈夫曼编码、香农编码、香农-弗诺编码、行程编码、LZW 编码、预测编码、傅里叶变换编码、小波变换编码、伪彩色处理，彩色图像的直方图均衡化算法、彩色图像平滑处理、彩色图像锐化处理、彩色图像边缘检测和彩色图像分割处理等内容。

读者对象

本书是为那些已经有一定 C#编程基础，并掌握了一些基本的数字图像处理原理，而想进一步利用 C#来进行数字图像处理的读者而编写的。

致谢

本书由赵春江负责编写并统编全部书稿，上海交通大学施文康教授为该书的出版作出了重要的贡献，在此特别感谢。

同时参与编写的还有谭敏、王俊、顾涓涓、胡学友、吉小军、邓勇、张广金、史贤荣、杨得彦、李积俊、黄楠、祁国军、包杰、赵龙、魏延文、卢广平、杨福财、解立龙、郭应世、曹永祥、柴鑫林、冯亮、任俊杰、李成良、何辰、王生成、高广越、韩晓冬、刘志宇、李松等，在此一并表示感谢。

由于笔者水平有限，编写时间仓促，书中难免有疏漏和不足之处，恳请广大读者提出宝贵意见。本书责任编辑的联系方式是 huangyan@ptpress.com.cn，欢迎来信交流。

编 者

2009 年 1 月

目　录

第1章 绪　　论

..1　数字图像处理概述

尽管最近十几年来，数字计算机和通信技术并没有十分重大的突破，但人们还是怀着极大的热情关注信息技术这一领域。这是因为越来越便宜的个人电脑及互联网的广泛应用，使人们获得了海量的及时信息。而大多数的这类信息都被设计成更容易理解的可视的形式，如文本、图像和多媒体。

图像处理就是对图像进行分析和处理的一门很有趣也很重要的学科，它无处不在，从电视到 CT，从摄像到印刷，从机器人到遥感，可以说，数字图像处理技术已经从工业领域、实验室走向商业领域、艺术领域及办公室，甚至走向了人们的日常生活。之所以图像信息在我们生活中几乎所有领域都扮演着重要的角色，那是由于图像的直观、易懂、存储方便和信息量大等特点所决定的。

一般来讲，根据对图像处理的不同目的，数字图像处理可以分为 3 类。

- 改善图像质量：如进行图像的亮度和颜色变换，增强和抑制某些成分，对图像进行几何变换等，以提高图像的视觉效果。
- 提取图像特征：被提取的特征可以包括很多方面，如频域特征、灰度或颜色特征、边缘特征、区域特征、纹理特征、形状特征、拓扑特征和关系结构等，从而为分析图像提供便利。
- 存储传输图像信息：对图像数据进行变换、编码和压缩。

..2　C#概述

1.2.1　C#特点

C#（C Sharp）是由微软公司所开发的一种面向对象，且运行于.NET Framework 之上的高级程序设计语言。C#看似基于 C++写成，但又融入其他语言如 Delphi、Java、VisualBasic 等。

（1）微软公司开发 C#的初衷及 C#的特点如下。

- C#旨在设计成为一种简单、现代、通用以及面向对象的程序设计语言。
- C#语言的实现，应提供对于以下软件工程要素的支持：强类型检查、数组维数检查、未初始化的变量引用检测、自动垃圾收集（一种自动内存释放技术），软件必须做到强大、持久，并具有较强的编程能力。
- C#语言应在分布式环境中的开发提供适用的组件开发应用。
- 为使程序员容易迁移到 C#语言，源代码的可移植性十分重要，尤其是对于那些已熟悉 C 和 C++的程序员而言。
- 对国际化的支持非常重要。
- C#适合为独立和嵌入式的系统编写程序，从使用复杂操作系统的大型系统到特定应用的小型系统均适用。
- 虽然 C#程序在存储和操作能力需求方面具备经济性，但此种语言并不能在性能和尺寸方面与 C 语言或汇编语言相抗衡。

（2）相对于 C 和 C++，C#在许多方面进行了限制和增强。

- 指针只能被用于不安全模式，大多数对象访问通过安全的引用实现，以避免无效的调用，并且有许多算法用于验证溢出，指针只能用于调用值类型以及受垃圾收集控制的托管对象。
- 对象不能被显式释放，而是当不存在被引用时通过垃圾回收器回收。
- 只允许单一继承，但是一个类可以实现多个接口。
- C#比 C++更加类型安全，默认的安全转换是隐含转换，例如由短整型转换为长整型和从派生类转换为基类，而接口同整型，及枚举型同整型不允许隐含转换，非空指针(通过引用相似对象)同用户定义类型的隐含转换必需被显式地确定，不同于 C++的复制构造函数。
- 数组声明语法不同。
- 枚举位于其所在的命名空间中。
- C#中没有模板，但是在 C# 2.0 中引入了泛型，并且支持一些 C++模板不支持的特性，比如泛型参数中的类型约束，另一方面，表达式不能像 C++模板中被用于类型参数。
- 属性支持，使用类似访问成员的方式调用。
- 完整的反射支持。

总之，C#学习起来要更容易，应用起来安全性更高。

C#并不被编译成为能够直接在计算机上执行的二进制本地代码。与 Java 类似，它被编译成为中间代码，然后通过.NET Framework 的虚拟机（被称之为通用语言运行时）执行。所有的.NET 编程语言都被编译成这种被称为 MSIL 的中间代码。因此虽然最终的程序在表面上仍然与传统意义上的可执行文件都具有“.exe”的后缀名。但是实际上，如果计算机上没有安装.NET Framework，那么这些程序将不能被执行。在程序执行时，.NET Framework 将中间代码翻译成为二进制机器码，从而使它得到正确的运行。最终的二进制代码被存储在一个缓冲区中。所以一旦程序使用了相同的代码，那么将会调用缓冲区中的版本。这样如果一个.NET 程序第二次被运行，那么这种翻译不需要进行第二次，速度明显加快。

1.2.2 WinForm 编程

WinForm 是.NET 开发平台中对 Windows Form 的一种称谓。.NET 为开发 WinForm 的应用程序提供了丰富的类库。这些 WinFrom 类库支持 RAD（快速应用程序开发），它们被封装在一个命名空间之中，这个命名空间就是 System.Windows.Forms。在该命名空间中定义了许多类，在开发基于.NET 的 GUI 应用程序的时候，就是通过继承和扩展这些类才使得程序有着多样的用户界面。本书开发的所有程序都是基于 WinForm 的。

一个典型的 Windows 窗体应用程序至少应该包含一个窗体。WinForm 应用程序中通常有一个窗体作为主窗体，它是该应用程序生命期内可能显示的其他窗体的父窗体或所有者，主菜单、工具栏、状态栏等都显示于该窗体内。当主窗体被关闭时，程序应该随即退出。

1.2.3 GDI+

GDI+是与.NET Framework 中的图形设备接口进行交互的入口。它使程序开发人员可以编写出与设备无关的受控应用程序，它的设计目的是要提供较高的性能、方便的使用以及对多语言的支持。如果要编写与监视器、打印机或文件等图形设备进行交互的.NET 应用程序，那么就必须使用 GDI+。

GDI+使得应用程序开发人员在输出屏幕和打印机信息的时候，无需考虑具体显示设备的细节，他们只需调用 GDI+库输出的类的一些方法即可完成图形操作，真正的绘图工作由这些方法交给特定的设备驱动程序来完成，GDI+使得图形硬件和应用程序相互隔离，从而使开发人员编写与设备无关的应用程序变得非常容易。

GDI+在 GDI 的基础上提供了明显地改进。最主要的特点是在 GDI+中，没有了句柄或设备上下文的概念，它被 Graphics 对象取代。Graphics 类提供了绘制不同图形对象的方法和属性，而且更易于使用。

..3 补充说明

- 本书并没有详细地讲解 C#的基本概念，也没有系统地介绍数字图像处理的基本理论和基本技术，而只是应用 C#这种高级语言实现数字图像处理中的几十种常用的算法。
- 本书侧重于实用性，力争用最简洁明了的语言介绍算法。在程序编写的过程中，既考虑效率，又考虑可读性，但更偏重于后者，从而使程序通俗易懂。
- 本书所用到的图像基本上都是图像处理领域内的标准测试图像。图像的文件格式均为 BMP 格式，大小都为 512×512，而且是 24 位彩色图像或者是 8 位灰度图像，即使是二值黑白图像，也仍然使用 8 位灰度图像代替（灰度值只有 0 和 255 两种，0 表示黑色，255 表示白色）。这样做的目的就是为了简化程序，而把精力放到具体算法的实现上，这更有利于初学者。因此，在本书的所有程序中，都默认为满足条件而没有对上述参数进行判断。如果要把本书的程序应用到其他场合，只需添加几条判断图像格式和大小的语句即可。
- 上节已经说过，在本书的其他章节中每一章都是一个完整的 WinForm 程序，图像显

示在主窗体内，通过布置在主窗体内的按钮控件来促发各种图像处理的算法。由于很多算法需要人为地设置各种参数，所以还需要创建一个从窗体用来设置相关参数。处理后的图像替代原图像，同样显示在主窗体内。

- 本书程序是在以下电脑配置下完成的。

硬件	CPU	AMD Athlon 4400+
	内存	DDRⅡ-800 1G
软件	操作系统	Microsoft Windows XP SP2
	开发环境	Visual Studio 2005

第 2 章　C#数字图像处理的 3 种方法

本章通过彩色图像灰度化这一简单的图像处理算法来介绍 C#数字图像处理的 3 种方法。通过编写第一个 C#程序，读者不仅可以了解 C#的特点，也能够掌握 C#图像处理的基本方法，为以后编写更复杂的数字图像处理算法打下基础。

2.1　C#图像处理基础

Bitmap 类、BitmapData 类和 Graphics 类是 C#图像处理中最重要的 3 个类，如果要用 C#进行图像处理，就一定要掌握它们。

2.1.1　Bitmap 类

Bitmap 对象封装了 GDI+中的一个位图，此位图由图形图像及其属性的像素数据组成。因此 Bitmap 是用于处理由像素数据定义的图像的对象。该类的主要方法和属性如下。

- GetPixel 方法和 SetPixel 方法：获取和设置一个图像的指定像素的颜色。
- PixelFormat 属性：返回图像的像素格式。
- Palette 属性：获取或设置图像所使用的颜色调色板。
- Height 属性和 Width 属性：返回图像的高度和宽度。
- LockBits 方法和 UnlockBits 方法：分别锁定和解锁系统内存中的位图像素。在基于像素点的图像处理方法中使用 LockBits 和 UnlockBits 是一个很好的方式，这两种方法可以使我们通过指定像素的范围来控制位图的任意一部分，从而消除了通过循环对位图的像素逐个进行处理的需要。每次调用 LockBits 之后都应该调用一次 UnlockBits。

LockBits 方法的定义如下：

```
public BitmapData LockBits ( Rectangle rect, ImageLockMode flags, PixelFormat format )
```

LockBits 方法使用 3 个类型，分别为 Rectangle、ImageLockMode 枚举和 PixelFormat 枚举的参数，并返回一个类型为 BitmapData 的对象。其中矩形参数定义了要在系统内存中锁定的位图的一部分；ImageLockMode 枚举提供了对数据的访问方式，表 2.1 所示是它的成员。PixelFormat 枚举表示像素的格式，表 2.2 所示是它的主要成员。

表 2.1　　ImageLockMode 的成员

成　员	描　述
ReadOnly	位图的锁定部分只用于读操作
ReadWrite	位图的锁定部分用于读操作和写操作
UserInputBuffer	读取和写入像素数据的缓存由用户分配
WriteOnly	位图的锁定部分只用于写操作

表 2.2　　PixelFormat 的主要成员

成　员	描　述
Format1bppIndexed	每个像素 1 位，使用索引颜色，因此颜色表中有两种颜色
Format4bppIndexed	每个像素 4 位，使用索引颜色
Format8bppIndexed	每个像素 8 位，使用索引颜色
Format16bppGrayScale	每个像素 16 位，共指定 65536 种灰色调
Format24bppRgb	每个像素 24 位，红色、绿色、蓝色分量分别使用 8 位，它们的顺序是蓝、绿、红
Format32bppArgb	每个像素 32 位，Alpha、红色、绿色、蓝色分量分别使用 8 位，这是默认的 GDI+颜色组合
Format64bppArgb	每个像素 64 位，Alpha、红色、绿色、蓝色分量分别使用 16 位
Indexed	索引颜色值，这些值是系统颜色表中颜色的索引，而不是单个颜色值

UnlockBits 方法使用一个由 LockBits 返回的类型为 BitmapData 的参数，它定义为：

```
public void UnlockBits ( BitampData bitmapdata );
```

2.1.2　BitmapData 类

BitmapData 对象指定了位图的属性，如下所示。

- Height 属性：被锁定位图的高度。
- Width 属性：被锁定位图的宽度。
- PixelFormat 属性：数据的实际像素格式。
- Scan0 属性：被锁定数组的首字节地址。如果整个图像被锁定，则是图像的第一个字节地址。
- Stride 属性：步幅，也称为扫描宽度。

如图 2.1 所示，数组的宽度并不一定等于图像像素数组的宽度，还有一部分未用区域。这是为了提高效率，系统要确定每行的字节数必须为 4 的倍数。例如一幅 24 位、宽为 17 个像素的图像，它需要每行占有的空间为 51（3×17）个字节，但

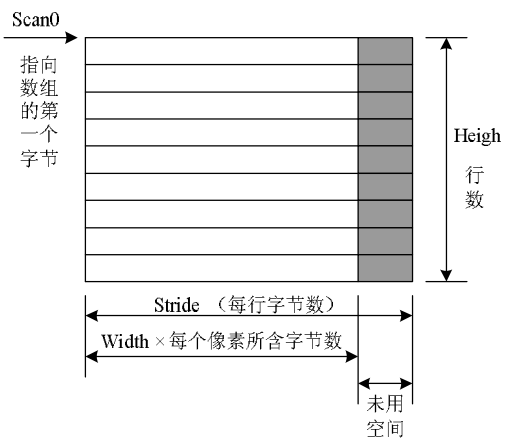


图 2.1　被锁定图像像素数组基本布局

1 不是 4 的倍数，因此还需要补充 1 个字节，从而使每行的字节数扩展为 52（4×13，即 Stride=52）这样就满足了每行字节数是 4 的倍数的条件。需要扩展多少个字节不仅是由图像的宽度决定，而且还由图像像素的格式决定。

由于本书所选择的图像大小都为 512×512，因此无论是 24 位彩色图像，还是 8 位的灰度图像，都满足是 4 的倍数的条件，无需再扩展。如果处理的是任意宽度的图像，那么在进行下一行扫描的时候，就需要把扩展字节去除掉。

2.1.3 Graphics 类

Graphics 对象是 GDI+的关键所在。许多对象都是由 Graphics 类表示的，该类定义了绘制和填充图形对象的方法和属性。一个应用程序只要需要进行绘制或着色，它就必须使用 Graphics 对象。

Graphics 类的方法和属性很多，我们在后面遇到时再进行详细讲解。

2.2 彩色图像灰度化

为加快处理速度，在图像处理算法中，往往需要把彩色图像转换为灰度图像，本书所介绍的算法大多数都是在灰度图像上进行的。况且，在灰度图像上得到验证的算法，很容易移植到彩色图像上。

24 位彩色图像每个像素用 3 个字节表示，每个字节对应着 *R*、*G*、*B* 分量的亮度（红、绿、蓝）。当 *R*、*G*、*B* 分量值不同时，表现为彩色图像；当 *R*、*G*、*B* 分量值相同时，表现为灰度图像，该值就是我们所求的。

一般来说，转换公式有 3 种。第一种转换公式为：

$$Gray(i,j)=\left[R(i,j)+G(i,j)+B(i,j)\right]\div 3 \tag{2.1}$$

其中，$Gray(i,j)$为转换后的灰度图像在(i,j)点处的灰度值。该方法虽然简单，但人眼对颜色的感应是不同的，因此有了第二种转换公式：

$$Gray(i,j)=0.299\times R(i,j)+0.587\times G(i,j)+0.114\times B(i,j) \tag{2.2}$$

我们观察上式，发现绿色所占的比重最大，所以转换时可以直接使用 *G* 值作为转换后的灰度：

$$Gray(i,j)=G(i,j) \tag{2.3}$$

在这里，我们应用最常用的公式(2.2)，并且变换后的灰度图像仍然用 24 位图像表示。

2.3 彩色图像灰度化编程实例

下面我们就详细地讲解本书的第一个 C#图像处理程序。

2.3.1　使用图像

首先使用 Visual Studio 创建一个 Windows 应用程序。它的步骤为：打开 Visual Studio，在主菜单中选择“文件 | 新建 | 项目”选项，然后在“项目类型”下选择“Visual C# Windows”，再在“模板”下选择“Windows 应用程序”，最后为项目选好路径和起好名字后，单击“确定”按钮即可，如图 2.2 所示。

图 2.2　创建项目

在打开的程序主窗体内添加 3 个 Button 控件，如图 2.3 所示，其属性修改如表 2.3 所示

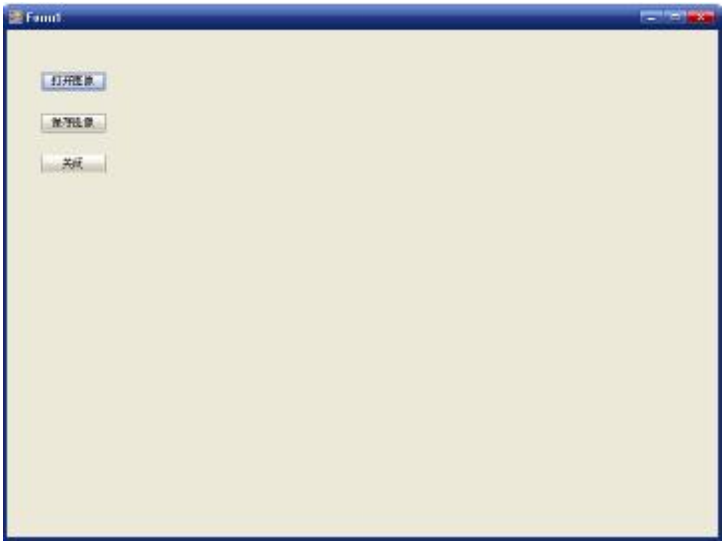


图 2.3　图像处理主窗体

表 2.3　　所修改的属性

控　　件	属　　性	所修改内容
Form1	Size	800，600
Botton1	Name	open

	Text	打开图像
	Location	37，46
Botton2	Name	save

续表

控　　件	属　　性	所修改内容
Botton2	Text	保存图像
	Location	37，92
Botton3	Name	close
	Text	关闭
	Location	37，138

设置完控件，我们就进入程序的编写工作。在应用程序范围中分别定义一个字符串和一个 Bitmap 类型的数据成员。在 Form1 类中添加以下代码：

```
// 文件名
private string curFileName;
// 图像对象
private System.Drawing.Bitmap curBitmap;
```

分别双击在主窗体内所添加的 3 个 Botton 控件，为它们添加 Click 事件，代码如下：

```
// 打开图像文件
private void open_Click(object sender, EventArgs e)
{
     // 创建OpenFileDialog
     OpenFileDialog opnDlg = new OpenFileDialog();
     // 为图像选择一个筛选器
     opnDlg.Filter = "所有图像文件 | *.bmp; *.pcx; *.png; *.jpg; *.gif;"+
       "*.tif; *.ico; *.dxf; *.cgm; *.cdr; *.wmf; *.eps; *.emf|" +
       "位图( *.bmp; *.jpg; *.png;...) | *.bmp; *.pcx; *.png; *.jpg; *.gif; *.tif; *.ico|"
       "矢量图( *.wmf; *.eps; *.emf;...) | *.dxf; *.cgm; *.cdr; *.wmf; *.eps; *.emf";
     // 设置对话框标题
     opnDlg.Title = "打开图像文件";
     // 启用“帮助”按钮
     opnDlg.ShowHelp = true;

     // 如果结果为“打开”，选定文件
     if (opnDlg.ShowDialog() == DialogResult.OK)
     {
          // 读取当前选中的文件名
          curFileName = opnDlg.FileName;
          // 使用Image.FromFile 创建图像对象
          try
          {
               curBitmap = (Bitmap)Image.FromFile(curFileName);
          }
          catch (Exception exp)
          {
               // 抛出异常
               MessageBox.Show(exp.Message);
          }
     }
```

```
        // 对窗体进行重新绘制，这将强制执行paint事件处理程序
        Invalidate();
    }

    // 保存图像文件
    private void save_Click(object sender, EventArgs e)
    {
        // 如果没有创建图像，则退出
        if(curBitmap == null)
            return;
        // 调用SaveFileDialog
        SaveFileDialog saveDlg = new SaveFileDialog();
        // 设置对话框标题
        saveDlg.Title = "保存为";
        // 改写已存在文件时提示用户
        saveDlg.OverwritePrompt = true;
        // 为图像选择一个筛选器
        saveDlg.Filter = "BMP文件 (*.bmp) | *.bmp|" + "Gif文件 (*.gif) | *.gif|" +
            "JPEG文件 (*.jpg) | *.jpg|" + "PNG文件 (*.png) | *.png";
        // 启用“帮助”按钮
        saveDlg.ShowHelp = true;

        // 如果选择了格式，则保存图像
        if( saveDlg.ShowDialog() == DialogResult.OK )
        {
            // 获取用户选择的文件名
            string fileName = saveDlg.FileName;
            // 获取用户选择文件的扩展名
            string strFilExtn = fileName.Remove(0, fileName.Length - 3);

            // 保存文件
            switch (strFilExtn)
            {
                case "bmp":
                    // bmp格式
                    curBitmap.Save(fileName, System.Drawing.Imaging.ImageFormat.Bmp);
                    break;
                case "jpg":
                    // jpg格式
                    curBitmap.Save(fileName, System.Drawing.Imaging.ImageFormat.Jpeg);
                    break;
                case "gif":
                    // gif格式
                    curBitmap.Save(fileName, System.Drawing.Imaging.ImageFormat.Gif);
                    break;
                case "tif":
                    // tif格式
                    curBitmap.Save(fileName, System.Drawing.Imaging.ImageFormat.Tiff);
                    break;
                case "png":
                    // png格式
                    curBitmap.Save(fileName, System.Drawing.Imaging.ImageFormat.Png);
                    break;
                default:
                    break;
```

```
            }
        }
    }

    // 关闭窗体，退出程序
    private void close_Click(object sender, EventArgs e)
    {
        this.Close();
    }
```

当一个应用程序需要进行绘制时，它必须通过 Graphics 对象来执行绘制操作。有多种方式可以获得一个与窗体相关联的 Graphics 对象，如使用窗体的 Paint 事件，重载 OnPaint 方法，使用 CreateGraphics 方法创建 Graphics 对象，及其使用 Graphics 类的静态方法等。在这里，我们使用窗体的 paint 事件的 PaintEventArgs 属性来获取一个与窗体相关联的 Graphics 对象。通过双击主窗体中属性事件列表中的 Paint 选项，进入主窗体 Paint 事件的代码区，为其编写代码，代码如下：

```
    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        // 获取Graphics 对象
        Graphics g = e.Graphics;

        if(curBitmap != null)
        {
            // 使用DrawImage 方法绘制图像
            // 160, 20: 显示在主窗体内，图像左上角的坐标
            // curBitmap.Width, curBitmap.Height: 图像的宽度和高度
            g.DrawImage(curBitmap, 160, 20, curBitmap.Width, curBitmap.Height);
        }
    }
```

编译并运行这段程序，如图 2.4 所示为打开的一幅 24 位彩色图像并把它显示到主窗体上。

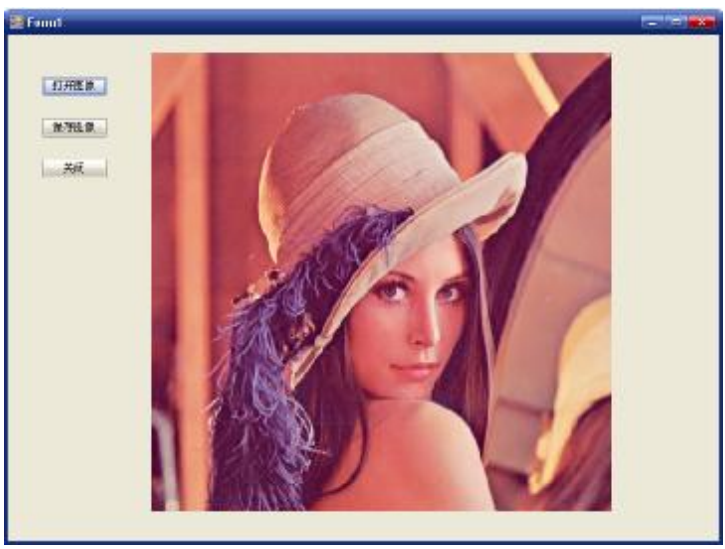


图 2.4 显示图像

2.3.2　图像处理的 3 种方法

在上一节中，我们介绍了如何读取、显示和保存图像，它们是以后各个章节的基础。下面我们就介绍 C#图像处理的 3 种方法：提取像素法、内存法和指针法，并比较它们的各自特点。

首先在主窗体内再添加 3 个 Button 控件，如图 2.5 所示，其属性修改如表 2.4 所示。

图 2.5　3 种方法主窗体

表 2.4　所修改的属性

控　件	属　性	所修改内容
Button1	Name	pixel
	Text	提取像素法
	Location	37，200
Button2	Name	memory
	Text	内存法
	Location	37，246
Button3	Name	pointer
	Text	指针法
	Location	37，292

2.3.2.1　提取像素法

该方法使用的是 GDI+中的 Bitmap.GetPixel 和 Bitmap.SetPixel 方法。为了将位图的颜色设置为灰度或其他颜色，就需要使用 GetPixel 来读取当前像素的颜色，再计算灰度值，最后使用 SetPixel 来应用新的颜色。双击“提取像素法”Button 控件，为该控件添加 Click 事件代码如下：

```
private void pixel_Click(object sender, EventArgs e)
{
```

```
        if(curBitmap != null)
        {
            Color curColor;
            int ret;

            // 二维图像数组循环
            for (int i = 0; i < curBitmap.Width; i++)
            {
                for (int j = 0; j < curBitmap.Height; j++)
                {
                    // 获取该点像素的RGB 颜色值
                    curColor = curBitmap.GetPixel(i,j);
                    // 利用公式(2.2)计算灰度值
                    ret = (int)(curColor.R * 0.299 + curColor.G * 0.587 + curColor.B * 0.114);
                    // 设置该点像素的灰度值，R=G=B=ret
                    curBitmap.SetPixel(i, j, Color.FromArgb(ret, ret, ret));
                }
            }
            // 对窗体进行重新绘制，这将强制执行Paint 事件处理程序
            Invalidate();
        }
    }
```

2.3.2.2 内存法

该方法就是把图像数据直接复制到内存中，这样就使程序的运行速度大大提高。双击“内存法”按钮控件，为该控件添加 Click 事件，代码如下：

```
private void memory_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 位图矩形
        Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
        // 以可读写的方式锁定全部位图像素
        System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
            System.Drawing.Imaging.ImageLockMode.ReadWrite,
            curBitmap.PixelFormat);
        // 得到首地址
        IntPtr ptr = bmpData.Scan0;

        // 24 位bmp 位图字节数
        int bytes = curBitmap.Width * curBitmap.Height * 3;
        // 定义位图数组
        byte[] rgbValues = new byte[bytes];
        // 复制被锁定的位图像素值到该数组内
        System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes);

        // 灰度化
        double colorTemp = 0;
        for (int i = 0; i < rgbValues.Length; i += 3)
        {
            // 利用公式(2.2)计算灰度值
            colorTemp = rgbValues[i + 2] * 0.299 + rgbValues[i + 1] * 0.587 +
                rgbValues[i] * 0.114;
```

```
                // R=G=B
                rgbValues[i] = rgbValues[i + 1] = rgbValues[i + 2] = (byte)colorTemp;
            }

            // 把数组复制回位图
            System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes);
            // 解锁位图像素
            curBitmap.UnlockBits(bmpData);
            // 对窗体进行重新绘制，这将强制执行Paint 事件处理程序
            Invalidate();
        }
    }
```

需要说明的是，本书所采用的图像大小都为 512×512，因此无需考虑被锁定位图数组是否有未用空间（如图 2.1 所示）。如果是任意大小的 24 位彩色图像，则上述程序就不再适用，需要修改，修改后的代码如下：

```
private void memory_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 位图矩形
        Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
        // 以可读写的方式锁定全部位图像素
        System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
            System.Drawing.Imaging.ImageLockMode.ReadWrite,
            curBitmap.PixelFormat);
        // 得到首地址
        IntPtr ptr = bmpData.Scan0;

        // 定义被锁定的数组大小，由位图数据与未用空间组成
        int bytes = bmpData.Stride * bmpData.Height;
        byte[] rgbValues = new byte[bytes];
        // 复制被锁定的位图像素值到该数组内
        System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes);

        // 灰度化
        double colorTemp = 0;
        for (int i = 0; i < bmpData.Height; i++)
        {
            // 只处理每行中是图像像素的数据，舍弃未用空间
            for (int j = 0; j < bmpData.Width * 3; j += 3)
            {
                colorTemp = rgbValues[i * bmpData.Stride + j + 2] * 0.299 +
                    rgbValues[i * bmpData.Stride + j + 1] * 0.587 +
                    rgbValues[i * bmpData.Stride + j] * 0.114;
                rgbValues[i * bmpData.Stride + j] = rgbValues[i * bmpData.Stride + j + 1]
                  rgbValues[i * bmpData.Stride + j + 2] = (byte)colorTemp;
            }
        }

        // 把数组复制回位图
        System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes);
        // 解锁位图像素
        curBitmap.UnlockBits(bmpData);
```

```
            // 对窗体进行重新绘制，这将强制执行Paint 事件处理程序
            Invalidate();
        }
    }
```

上述程序适用于所有的 24 位图像，它为了考虑锁定数组中未用的空间而用到了 Stride 属性。本书为了简化程序，把所有图像的大小都限定为 512×512，所以在以后的程序中，我们都认为被锁定的数组就是全部的图像数据，而不采用上述程序。

2.3.2.3 指针法

该方法与内存法相似，开始都是通过 LockBits 方法来获取位图的首地址。但该方法更简单，直接应用指针对位图进行操作。

为了保持类型安全，在默认情况下，C#是不支持指针运算的，因为使用指针会带来相关的风险。所以 C#只允许在特别标记的代码块中使用指针。通过使用 unsafe 关键字，可以定义可使用指针的不安全上下文。

双击“指针法”按钮控件，为该控件添加 Click 事件，代码如下：

```
private void pointer_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
          // 位图矩形
          Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
          // 以可读写的方式锁定全部位图像素
          System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
              System.Drawing.Imaging.ImageLockMode.ReadWrite,
              curBitmap.PixelFormat);

          byte temp = 0;
          // 启动不安全模式
          unsafe
          {
              // 得到首地址
              byte* ptr = (byte*)(bmpData.Scan0);
              // 二维图像循环
              for (int i = 0; i < bmpData.Height; i++)
              {
                    for (int j = 0; j < bmpData.Width; j++)
                    {
                          // 利用公式(2.2)计算灰度值
                          temp = (byte)(0.299 * ptr[2] + 0.587 * ptr[1] + 0.114 * ptr[0]);
                          // R=G=B
                          ptr[0] = ptr[1] = ptr[2] = temp;
                          // 指向下一个像素
                          ptr += 3;
                    }
                    // 指向下一行数组的首个字节
                    ptr += bmpData.Stride - bmpData.Width * 3;
              }
          }

          // 解锁位图像素
          curBitmap.UnlockBits(bmpData);
          // 对窗体进行重新绘制，这将强制执行Paint 事件处理程序
```

```
            Invalidate();
        }
    }
```

该段代码适用于所有大小的 24 位图像。但由于启动了不安全模式，为了能够顺利地编译该段代码，必须设置相关选项。在主菜单中选择“项目 | gray 属性”，在打开的属性页中选择“生成”属性页，最后选中“允许不安全代码”复选框，如图 2.6 所示。

图 2.6　设置不安全模式

2.3.2.4　3 种方法的比较

从 3 段代码的长度和难易程度来看，提取像素法又短又简单。它直接应用 GDI+中的 Bitmap.GetPixel 方法和 Bitmap.SetPixel 方法，大大减少了代码的长度，降低了使用者的难度，并且可读性好。但衡量程序好坏的标准不是仅仅看它的长度和难易度，而是要看它的效率，尤其是像图像处理这种往往需要处理二维数据的大信息量的应用领域，就更需要考虑效率了。

为了比较这 3 种方法的效率，我们对其进行计时。首先在主窗体内添加一个 Label 控件和 TextBox 控件，如图 2.7 所示，其属性修改如表 2.5 所示。

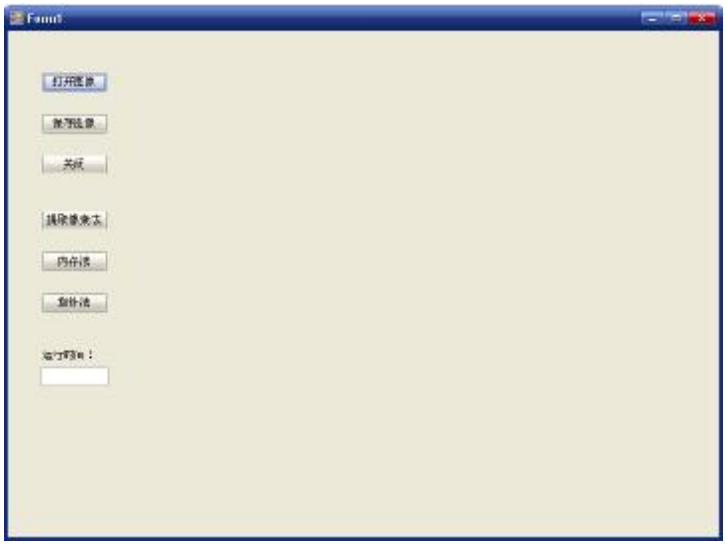


图 2.7 运行时间主窗体

表 2.5　　所修改的属性

控　件	属　性	所修改内容
label1	Text	运行时间：
	Location	37，356
textBox1	Name	timeBox
	TextAlign	Right
	Location	37，375

C#中一共有 4 种常用的计时器。

- System.Timers.Timer 和 System.Windows.Forms.Timer，它的最低识别为 1/18s。
- timeGetTime，它的最低识别能达到 5ms。
- System.Environnent.TickCount，它的最低识别为毫秒级。
- QueryPerformanceCounter，它的最低识别为 1ms。

在这里，我们采用第 4 种方法，具体步骤如下。

（1）新加一个 HiPerfTimer 类。首先，在主菜单中选择“项目 | 添加类”，打开“添加新项”对话框。在该对话框模板内选择“类”，然后在名称内填入“HiPerfTimer.cs”，如图 2.8 所示，最后单击“添加”按钮。

图 2.8　添加类对话框

（2）在 HiPerfTimer.cs 文件中添加如下代码：

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;
using System.Threading;

namespace gray
{
    internal class HiPerfTimer
    {
        // 引用Win32 API 中的QueryPerformanceCounter()方法
        // 该方法用来查询任意时刻高精度计数器的实际值
        [DllImport("Kernel32.dll")]
        private static extern bool QueryPerformanceCounter(
```

```
            out long lpPerformanceCount);

        // 引用Win32 API 中的QueryPerformanceFrequency() 方法
        // 该方法返回高精度计数器每秒的计数值
        [DllImport("Kernel32.dll")]
        private static extern bool QueryPerformanceFrequency(
            out long lpFrequency);

        private long startTime, stopTime;
        private long freq;

        // 构造函数
        public HiPerfTimer()
        {
            startTime = 0;
            stopTime = 0;

            if (QueryPerformanceFrequency(out freq) == false)
            {
                // 不支持高性能计时器
                throw new Win32Exception();
            }
        }

        // 开始计时
        public void Start()
        {
            // 让等待线程工作
            Thread.Sleep(0);

            QueryPerformanceCounter(out startTime);
        }

        // 结束计时
        public void Stop()
        {
            QueryPerformanceCounter(out stopTime);
        }

        // 返回计时结果 (ms)
        public double Duration
        {
            get
            {
                return (double)(stopTime - startTime) * 1000 / (double)freq;
            }
        }
    }
}
```

（3）在 Form1 类内定义 HiPerfTimer 类：

```
private HiPerfTimer myTimer;
```

在构造函数内为其实例化：

```
myTimer = new HiPerfTimer();
```

分别在“提取像素法”、“内存法”和“指针法”Button 控件的 Click 事件程序代码内的判断语句之间的最开始一行添加以下代码：

```
// 启动计时器
myTimer.Start();
```

在上述 3 个单击事件内的 Invalidate()语句之前添加以下代码：

```
// 关闭计时器
myTimer.Stop();
// 在TextBox 内显示计时时间
timeBox.Text = myTimer.Duration.ToString("####.##") +" 毫秒";
```

最后，编译并运行该段程序。我们以如图 2.4 所示的 512×512 的 24 位 BMP 彩色图像为实验对象，在正常运行情况下，分别对 3 种方法的运行时间进行测试，其中图 2.9 所示为单击“指针法”按钮后的结果。表 2.6 所示为 3 种方法的实验结果。

图 2.9　运行结果示意图

表 2.6　　运行时间结果

方　法	提取像素法	内　存　法	指　针　法
平均运行时间（ms）	635	10	9

从表 2.6 中可以明显看出，内存法和指针法比提取像素法要快得多。提取像素法应用 GDI+中的方法，易于理解，方法简单，很适合于 C#的初学者使用，但它的运行速度最慢，效率最低。内存法把图像复制到内存中，直接对内存中的数据进行处理，速度明显提高，程序难度也不大。指针法直接应用指针来对图像进行处理，所以速度最快。但在 C#中是不建议使用指针的，因为使用指针，代码不仅难以编写和调试，而且无法通过 CLR 的内存类型安全检查，不能发挥 C#的特长。只有对 C#和指针有了充分的理解，才能用好该方法。究竟要使用哪种方法，还要看具体情况而定。但 3 种方法都能有效地对图像进行处理。

本书应用的是内存法。

2.4 小结

本章通过编写第一个 C#图像处理程序——彩色图像灰度化，使读者了解 C#图像处理的基本方法，并且还分析和比较了提取像素法、内存法和指针法的各自特点。本章是其他章节的基础，虽然数字图像处理的算法有很多，难易程度也不尽相同，但只要掌握了本章所介绍的知识，就可以较轻松地编写出任意一种图像处理算法来。

第 3 章　点运算及直方图

在数字图像处理中，点运算是一种简单而重要的技术。

点运算是只根据对应像素的输入灰度值来决定该像素输出灰度值的图像处理运算。它有时也被称为对比度增强、对比度拉伸或灰度变换。点运算没有改变图像内的空间关系，它是按照一定的方式改变了图像的灰度直方图。

灰度直方图是一种最简单且最有用的工具，它概括了一幅图像的灰度级内容。

3.1　灰度直方图

任何一幅图像的直方图都包括了可观的信息，某些类型的图像还可以由其直方图完全描述。

3.1.1　灰度直方图定义

灰度直方图是灰度的函数，描述的是图像中具有该灰度级的像素的个数。如果用直角坐标系来表示，则它的横坐标是灰度级，纵坐标是该灰度出现的概率（像素的个数）。

灰度直方图的分布函数为：

$$h(k) = n_k \tag{3.1}$$

其中，k 是指第 k 个灰度级，n_k 是灰度级为 r_k 的像素总和。如果是 8 位灰度图像，k=0、1、…、255。

直方图是很多空间域处理技术的基础，并且它在软件中易于实现，也适用于商用硬件设备，因此，它已成为实时图像处理的一个流行工具。

3.1.2　灰度直方图编程实例

根据灰度直方图的定义，只要遍历二维图像中的所有像素，计算出每个灰度级的像素个数，即可得到其灰度直方图。

本实例实现了绘制主窗体内图像的直方图的功能，并把它显示在从窗体内。

（1）打开 Visual Studio，创建一个 C#的 Windows 应用程序。在主窗体内添加 3 个 Button 控件，其属性修改如表 3.1 所示。

表 3.1　所修改的属性

控　件	属　性	所修改内容
form1	Size	800，600
button1	Name	open
	Text	打开图像
	Location	37，46
button2	Name	close
	Text	关闭
	Location	37，92
button3	Name	histogram
	Text	绘制直方图
	Location	37，150

为“打开图像”按钮和“关闭”按钮的 Click 事件，以及为 form1 的 Paint 事件添加代码的内容和第 2 章介绍的一样。

需要说明的是，在以后的章节中，所有程序都是以此为基础，即每章程序的开始都需添加这些代码，称之为本书的“模板主窗体”。所以在后面我们就不再赘述这些内容，直接进入相关算法代码的编写。

然后为“绘制直方图”按钮的 Click 事件编写代码，该按钮的功能是为所打开的图像绘制直方图，其直方图是在新打开的从窗体内绘制完成的，代码如下：

```
private void histogram_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 定义并实例化新窗体，并把图像数据传递给它
        histForm histoGram = new histForm(curBitmap);
        // 打开从窗体
        histoGram.ShowDialog();
    }
}
```

（2）为了在新窗体内绘制直方图，需要创建一个窗体。在主菜单内选择“项目 | 添加 Windows 窗体”，打开“添加新项”对话框，如图 3.1 所示。为新窗体起好名称后，单击“添加”按钮即可。

图 3.1　添加直方图窗体

在新窗体内添加 Button 控件，其属性修改如表 3.2 所示。

表 3.2　　所修改的属性

控　　件	属　　性	所修改内容
histForm	Text	直方图
	Size	355，340
	ControlBox	False
button1	Name	close
	Text	关闭
	Location	252，265

为“关闭”按钮控件添加 Click 事件，代码如下：

```
private void close_Click(object sender, EventArgs e)
{
     // 关闭窗体
     this.Close();
}
```

在该窗体的类内部，定义如下 3 个数据成员：

```
// 图像数据
private System.Drawing.Bitmap bmpHist;
private int[] countPixel;
// 记录最大的灰度级个数
private int maxPixel;
```

为了实现两个窗体之间的数据传递，需要改写 histForm 窗体的构造函数，代码如下：

```
public histForm(Bitmap bmp)
{
    InitializeComponent();
    // 把主窗体的图像数据传递给从窗体
    bmpHist = bmp;
    // 灰度级计数
    countPixel = new int[256];
}
```

分别为 histForm 窗体添加 Paint 和 Load 事件，Paint 事件用于绘制直方图，Load 事件用于计算各个灰度级所具有的像素个数，代码如下：

```
private void histForm_Paint(object sender, PaintEventArgs e)
{
    // 获取Graphics 对象
    Graphics g = e.Graphics;

    // 创建一个宽度为1 的黑色钢笔
    Pen curPen = new Pen(Brushes.Black, 1);


    // 绘制坐标轴
    g.DrawLine(curPen, 50, 240, 320, 240);
    g.DrawLine(curPen, 50, 240, 50, 30);

    // 绘制并标识坐标刻度
```

```
        g.DrawLine(curPen, 100, 240, 100, 242);
        g.DrawLine(curPen, 150, 240, 150, 242);
        g.DrawLine(curPen, 200, 240, 200, 242);
        g.DrawLine(curPen, 250, 240, 250, 242);
        g.DrawLine(curPen, 300, 240, 300, 242);
        g.DrawString("0", new Font("New Timer", 8), Brushes.Black, new PointF(46, 242));
        g.DrawString("50", new Font("New Timer", 8), Brushes.Black, new PointF(92, 242));
        g.DrawString("100", new Font("New Timer", 8), Brushes.Black, new PointF(139, 242))
        g.DrawString("150", new Font("New Timer", 8), Brushes.Black, new PointF(189, 242))
        g.DrawString("200", new Font("New Timer", 8), Brushes.Black, new PointF(239, 242))
        g.DrawString("250", new Font("New Timer", 8), Brushes.Black, new PointF(289, 242))
        g.DrawLine(curPen, 48, 40, 50, 40);
        g.DrawString("0", new Font("New Timer", 8), Brushes.Black, new PointF(34, 234));
        g.DrawString(maxPixel.ToString(), new Font("New Timer", 8), Brushes.Black,
            new PointF(18, 34));

        // 绘制直方图
        double temp = 0;
        for (int i = 0; i < 256; i++)
        {
              // 纵坐标长度
              temp = 200.0 * countPixel[i] / maxPixel;
              g.DrawLine(curPen, 50 + i, 240, 50 + i, 240 - (int)temp);
        }
        // 释放对象
        curPen.Dispose();
    }

    private void histForm_Load(object sender, EventArgs e)
    {
        // 锁定8位灰度位图
        Rectangle rect = new Rectangle(0, 0, bmpHist.Width, bmpHist.Height);
        System.Drawing.Imaging.BitmapData bmpData = bmpHist.LockBits(rect,
            System.Drawing.Imaging.ImageLockMode.ReadWrite, bmpHist.PixelFormat);
        IntPtr ptr = bmpData.Scan0;
        int bytes = bmpHist.Width * bmpHist.Height;
        byte[] grayValues = new byte[bytes];
        System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

        byte temp = 0;
        maxPixel = 0;
        // 灰度等级数组清零
        Array.Clear(countPixel, 0, 256);
        // 计算各个灰度级的像素个数
        for (int i = 0; i < bytes; i++)
        {
              // 灰度级
              temp = grayValues[i];
              // 计数加1
              countPixel[temp]++;
              if (countPixel[temp] > maxPixel)
            {
                    // 找到灰度频率最大的像素数，用于绘制直方图
                    maxPixel = countPixel[temp];
            }
```

```
        }

        // 解锁
        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
        bmpHist.UnlockBits(bmpData);
    }
```

（3）编译并运行该段程序。打开一幅 8 位灰度图像，单击“绘制直方图”按钮，则在弹出的窗体内完成了主窗体所显示的图像的直方图绘制，如图 3.2 所示。

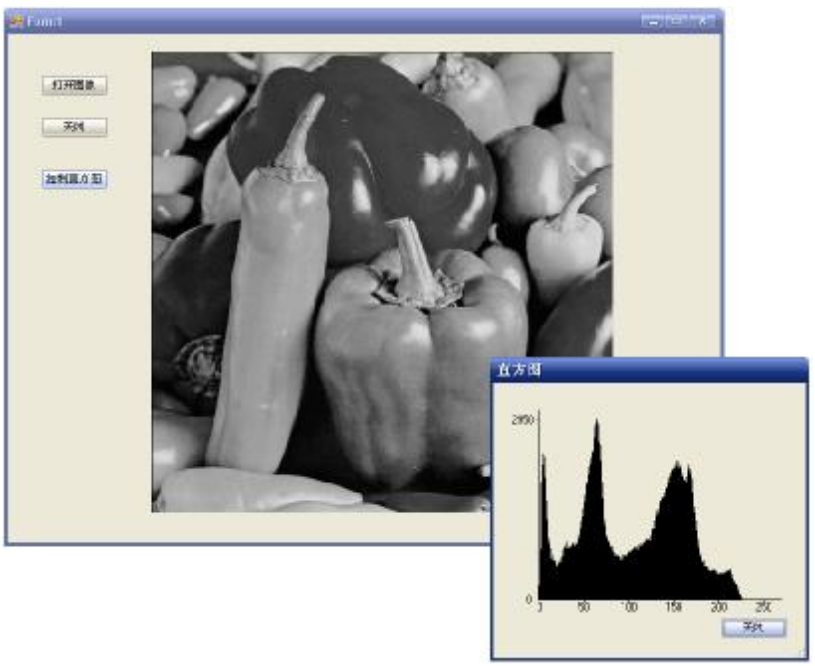


图 3.2　图像的直方图

3.2　线性点运算

灰度图像的点运算可分为线性点运算和非线性点运算两种。

3.2.1　线性点运算定义

线性点运算就是输出灰度级与输入灰度级呈线性关系的点运算。在这种情况下，灰度变换函数的形式为：

$$g(x,y) = pf(x,y) + L \tag{3.2}$$

其中，$f(x,y)$为输入图像在点(x,y)的灰度值，$g(x,y)$为相应的输出点的灰度值。显然，如果 $P=1$ 和 $L=0$，$g(x,y)$就是 $f(x,y)$的复制；如果 $P<1$，输出图像的对比度将增大；如果 P，则对比度将减少；如果 $P=1$ 而 $L\neq 0$，该操作仅使所有像素的灰度值上移或下移，其效果是使整个图像在显示时更暗或更亮；如果 P 为负值，暗区域将变亮，亮区域将变暗，该操作完成了图像求补。

3.2.2　线性点运算编程实例

该实例可以通过调整公式（3.2）中的参数 ***P*** 和 ***L***，实现对灰度图像进行任意线性点运算的功能。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 3.3 所示。

表 3.3　　所修改的属性

控　件	属　性	所修改内容
linearPO	Text	线性点运算
	Name	linearPO
	Location	37，196

（2）创建 1 个名为 linearPOForm 的 Windows 窗体，该窗体用于选择线性点运算的参数 ***P*** 和 ***L***。在该窗体内添加 2 个 Button 控件、3 个 Label 控件和 2 个 TextBox 控件，其属性修改如表 3.4 所示。

表 3.4　　所修改的属性

控　件	属　性	所修改内容
linearPOForm	Text	线性点运算
	ControlBox	False
	Size	260，240
button1	Name	startLinear
	Text	确定
	Location	26，160
button2	Name	close
	Text	退出
	Location	150，160
label1	Text	线性点运算：g(x,y)=Pf(x,y)+L
	Location	24，24
label2	Text	斜率（P）：
	Location	30，69
label3	Text	偏移量（L）：
	Location	24，112
textBox1	Name	scaling
	Location	101，66
	Text	1
textBox2	Name	offset
	Location	101，109
	Text	0

分别为 2 个 Button 控件添加 Click 事件，并为了与主窗体之间传递数据，再添加 2 个 ge

属性访问器，代码如下：

```
private void startLinear_Click(object sender, EventArgs e)
{
     // 设置DialogResult 属性
     this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
     // 关闭窗体
     this.Close();
}

// 设置两个get 访问器
public string GetScaling
{
    get
    {
          // 得到斜率
          return scaling.Text;
    }
}

public string GetOffset
{
     get
    {
          // 得到偏移量
          return offset.Text;
    }
}
```

（3）回到主窗体，为“线性点运算”按钮控件添加 Click 事件，代码如下：

```
private void linearPO_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
          // 实例化linearPOForm 窗体
          linearPOForm linearForm = new linearPOForm();

          // 单击“确定”按钮
          if (linearForm.ShowDialog() == DialogResult.OK)
          {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height)
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] grayValues = new byte[bytes];
                System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes)

                int temp = 0;
                // string 类型转换为double 类型
                // 得到斜率
                double a = Convert.ToDouble(linearForm.GetScaling);
```

```
                    // 得到偏移量
                    double b = Convert.ToDouble(linearForm.GetOffset);

                    for (int i = 0; i < bytes; i++)
                    {
                          // 根据公式(3.2)计算线性点运算
                          // 加0.5表示四舍五入
                          temp = (int) (a * grayValues[i] + b + 0.5);

                          // 灰度值限制在0~255之间
                          // 大于255，则为255；小于0，则为0
                          if (temp > 255)
                                grayValues[i] = 255;
                          else if (temp < 0)
                                grayValues[i] = 0;
                          else
                                grayValues[i] = (byte)temp;
                    }

                    System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                    curBitmap.UnlockBits(bmpData);
              }

              Invalidate();
        }
  }
```

（4）编译并运行该段程序，还是以上一个图像为例，在这里我们想通过线性点运算得到原图像的“负片”。打开图像后，单击“线性点运算”按钮，如图 3.3 所示填写相关数据。然后单击“确定”按钮，就得到了图像的“负片”，如图 3.4 所示。为了更好地看出效果，把它的直方图也一并显示出来。

图 3.3　线性点运算对话框

通过与图 3.2 比较可以看出，负片与原图像的直方图正好相反。

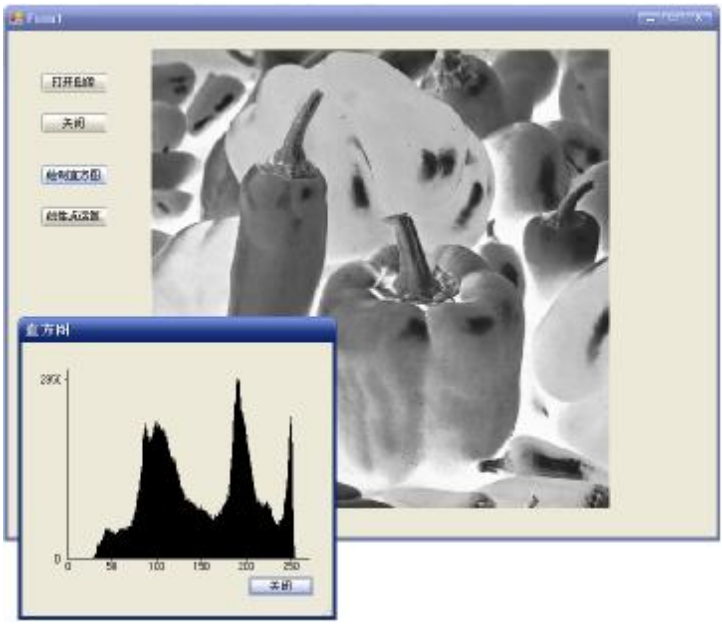


图 3.4 负片结果

3.3　全等级直方图灰度拉伸

灰度拉伸也属于线性点运算的一种，也可以通过上一节的程序得到。但由于它在点运算的特殊性，所以把它单独列出来进行介绍。

3.3.1　灰度拉伸定义

如果一幅图像的灰度值分布在全等级灰度范围内，即在 0～255 之间，那么它更容易被区别确认出来。

灰度拉伸，也称对比度拉伸，是一种简单的线性点运算。它扩展图像的直方图，使其充满整个灰度等级范围内。

设 $f(x,y)$为输入图像，它的最小灰度级 A 和最大灰度级 B 的定义为：

$$A=\min\left[f(x,y)\right] \quad B=\max\left[f(x,y)\right] \tag{3.3}$$

我们的目标是按照公式（3.2），把 A 和 B 分别线性映射到 0 和 255，因此，最终的图像 (x,y)为：

$$g(x,y)=\left(\frac{255}{B-A}\right)\left[f(x,y)-A\right] \tag{3.4}$$

3.3.2　灰度拉伸编程实例

该实例实现了对原图像的全等级直方图灰度拉伸。

（1）在主窗体内添加一个 Button 控件，其属性修改如表 3.5 所示。

表 3.5　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	stretch
	Text	灰度拉伸
	Location	37，242

（2）为该控件添加 Click 事件，代码如下：

```
private void stretch_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
          Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
          System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
              System.Drawing.Imaging.ImageLockMode.ReadWrite,
              curBitmap.PixelFormat);
          IntPtr ptr = bmpData.Scan0;
          int bytes = curBitmap.Width * curBitmap.Height;
          byte[] grayValues = new byte[bytes];
          System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

          byte a = 255, b = 0;
          double p;
```

```
            // 计算最大和最小灰度级
            for (int i = 0; i < bytes; i++)
            {
                  // 最小灰度级
                  if (a > grayValues[i])
                  {
                        a = grayValues[i];
                  }
                  // 最大灰度级
                  if (b < grayValues[i])
                  {
                        b = grayValues[i];
                  }
            }
            // 得到斜率
            p = 255.0 / (b - a);

            // 灰度拉伸
            for (int i = 0; i < bytes; i++)
            {
                  // 公式(3.4)
                  grayValues[i] = (byte)(p * (grayValues[i] - a) + 0.5);
            }

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            curBitmap.UnlockBits(bmpData);

            Invalidate();
      }
}
```

（3）编译并运行该段程序。通过对图 3.5 所示的图像进行灰度拉伸试验发现该图偏暗，通过其直方图可以看出它的灰度级并不是在全体灰度等级上的。

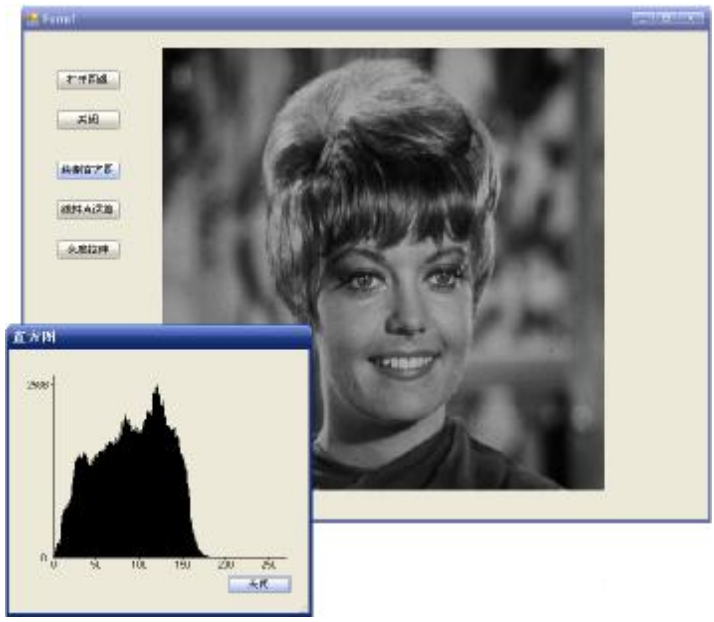


图 3.5　灰度拉伸原图

经过灰度拉伸后，得到了如图 3.6 所示的结果。拉伸后的图像亮度增加，而且通过其直方图可以看出，尽管直方图的分布曲线的形状没有变换，但它的灰度级已经分布在 0～255范围内了。

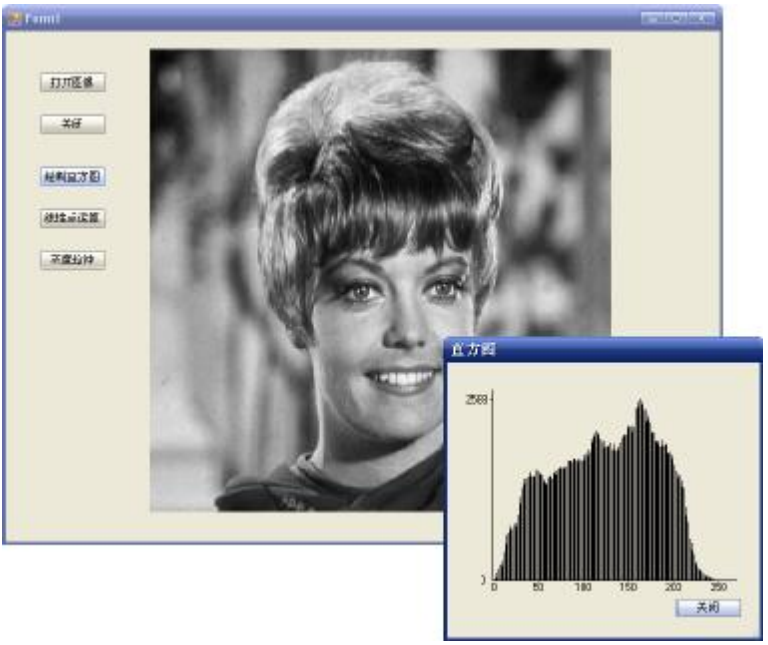


图 3.6　灰度拉伸结果

3.4　直方图均衡化

3.4.1　直方图均衡化定义

直方图均衡化，又称直方图修平，它是一种很重要的非线性点运算。该方法通常用来增加图像的局部对比度，尤其是当图像的有用数据的对比度相当接近的时候。通过这种方法，亮度可以更好地在直方图上分布。

直方图均衡化的基本思想是把原始图像的直方图变换为均匀分布的形式，这样就增加了像素灰度值的动态范围，从而可达到增强图像整体对比度的效果。

它的具体算法为：首先计算图像 f 的各灰度级中像素出现的概率。

$$p(i)=\frac{n_i}{n},\quad i\in 0,1,\mathbf{K},L-1 \tag{3.5}$$

其中 n_i 表示灰度级 i 出现的次数，L 是图像中所有的灰度数。p 实际上是图像的直方图归一化到 0～1 范围内。把 c 作为对应于 p 的累计概率函数，定义为：

$$c(i)=\sum_{j=0}^{i}p(x_j) \tag{3.6}$$

c 是图像的累计归一化直方图。

创建一个形式为 $y=T(x)$的变化，对于原始图像中的每一个值它就产生一个 y，这样 y 的累计概率函数就可以在所有值范围内进行线性化，转换公式定义为：

$$y_i = T(x_i) = c(i) \tag{3.7}$$

这时的 T 是将不同的等级映射到 0～1 范围内。

3.4.2 直方图均衡化编程实例

该实例实现了灰度图像的直方图均衡化。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 3.6 所示。

表 3.6 所修改的属性

控　件	属　性	所修改内容
button1	Name	equalization
	Text	直方图均衡
	Location	37，288

（2）为该控件添加 Click 事件，代码如下：

```
private void equalization_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
        System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
            System.Drawing.Imaging.ImageLockMode.ReadWrite,
            curBitmap.PixelFormat);
        IntPtr ptr = bmpData.Scan0;
        int bytes = curBitmap.Width * curBitmap.Height;
        byte[] grayValues = new byte[bytes];
        System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

        byte temp;
        int[] tempArray = new int[256];
        int[] countPixel = new int[256];
        byte[] pixelMap = new byte[256];

        // 计算各灰度级的像素个数
        for (int i = 0; i < bytes; i++)
        {
            // 灰度级
            temp = grayValues[i];
            // 计数加1
            countPixel[temp]++;
        }

        // 计算各灰度级的累计分布函数
        for (int i = 0; i < 256; i++)
        {
            if (i != 0)
```

```
                {
                        tempArray[i] = tempArray[i - 1] + countPixel[i];
                }
                else
                {
                        tempArray[0] = countPixel[0];
                }
                // 计算累计概率函数，并把值扩展为0～255范围内
                pixelMap[i] = (byte)(255.0 * tempArray[i] / bytes + 0.5);
        }

        // 灰度等级映射转换处理
        for (int i = 0; i < bytes; i++)
        {
                temp = grayValues[i];
                grayValues[i] = pixelMap[temp];
        }

        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
        curBitmap.UnlockBits(bmpData);

        Invalidate();
    }
}
```

（3）编译并运行该段程序。对如图 3.7 所示的图像进行直方图均衡化处理，它的直方图也同时显示出来。从直方图可以看出，它的灰度分布不均衡，主要集中在灰度级的中部。

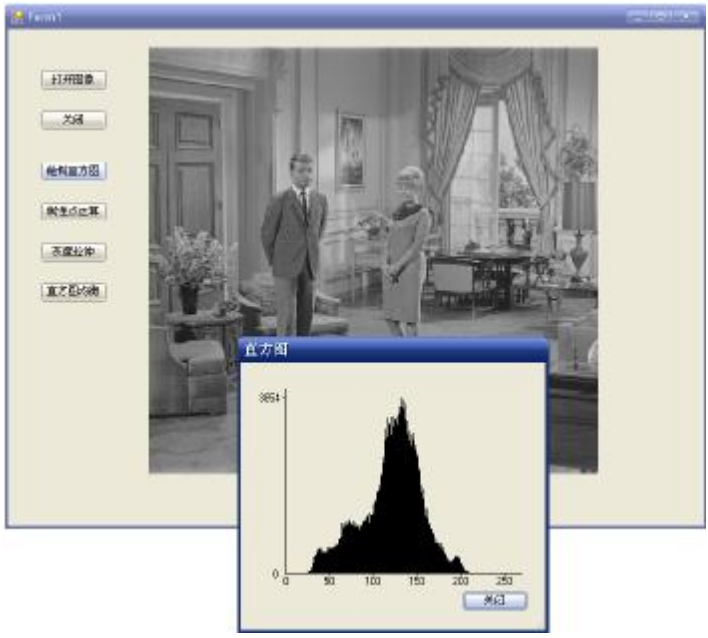


图 3.7　直方图均衡化原图

单击“直方图均衡”按钮，就得到了如图 3.8 所示的输出结果，为了更好地说明问题，把它的直方图也一并显示出来。

从图 3.8 可以明显看出，它的亮度加强了，其直方图也呈现出均匀分布的状态。但需要说明的是，在离散情况下灰度级是不可能作到绝对的一致。

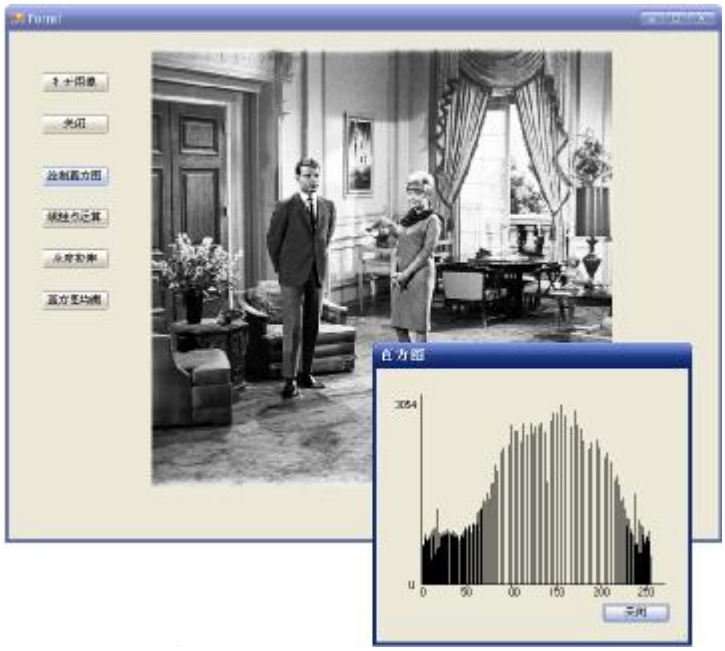


图 3.8　直方图均衡化的结果

3.5　直方图匹配

3.5.1　直方图匹配定义

直方图匹配，又称直方图规定化，是变换原图像的直方图为规定的某种形式的直方图。

能够指定想要的处理后的图像的直方图形状在某些应用中是非常有用的，它属于非线性点运算，3.4 节介绍的直方图均衡化实际上是直方图匹配的一种特例。

把现有的直方图为 $H_a(a_k)$的图像 $a(x, y)$变换到具有某一指定直方图 $H_c(c_k)$的图像 $c(x, y)$一般分两步进行：先把图像 $a(x, y)$变换为具有均衡化直方图的中间图像 $b(x,y)$，而后把 $b(x, y)$变换到 $c(x, y)$。其方法过程可以概括如下。

（1）假定 a_k 和 b_k 的取值范围相同，则分别对 $H_a(a_k)$和 $H_c(c_k)$作均衡化处理，使 a_k 映射成 g_m，c_k 映射成 y_n。g_m 和 y_n 的直方图都应该是近似均匀分布的。这时，查找 g_m 和 y_n 的对应关系，在 $g_m \approx y_n$ 的位置上，找到分别对应于 g_m 和 y_n 的原来灰度级 a_k 和 c_k，于是把此 a_k 映射成 c_k，即 a_k ð c_k。

（2）把两次映射组合一个函数，使得可由 a_k 直接映射成 c_k，若令

$$g_m=T(a_k),\quad y_n=G(c_k) \tag{3.8}$$

式中 $T(\ddot{Y})$和 $G(\ddot{Y})$分别是 a_k ð g_m 和 c_k ð y_n 的变换函数，则在 $g_m \approx y_n$ 处，有

$$c_k=G^{-1}(y_n)=G^{-1}(g_m)=G^{-1}[T(a_k)] \tag{3.9}$$

式中 G^{-1} 是 c_k ð y_n 的反变换函数。由此便得到映射 a_k ð c_k 及其 $H_c(c_i)$。

3.5.2 直方图匹配编程实例

本实例实现了直方图的匹配，待匹配的直方图是由我们任意打开的另一幅图像得到，最终的结果是原图像的直方图和待匹配的直方图相似。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 3.7 所示。

表 3.7 所修改的属性

控　件	属　性	修改内容
button1	Name	shaping
	Text	直方图匹配
	Location	37，334

（2）创建 1 个名为 shapinForm 的 Windows 窗体，该窗体用于显示待匹配的直方图。在该窗体内添加 3 个 Button 控件，其属性修改如表 3.8 所示。

表 3.8 所修改的属性

控　件	属　性	所修改内容
shapinForm	Text	直方图匹配
	ControlBox	False
	Size	370，350
button1	Name	open
	Text	打开文件
	Location	24，270
button2	Name	startShaping
	Text	确定
	Location	149，270
button3	Name	close
	Text	退出
	Location	270，270

在该窗体的应用程序范围内定义数据变量，代码如下：

```
private string shapingFileName;
private System.Drawing.Bitmap shapingBitmap;
private int[] shapingPixel;
private double[] cumHist;
private int shapingSize;
private int maxPixel;
```

在其构造函数内初始化变量，代码如下：

```
shapingPixel = new int[256];
cumHist = new double[256];
```

为 3 个 Button 控件添加 Click 事件，代码如下：

```
private void open_Click(object sender, EventArgs e)
{
```

```
        // 打开所要匹配的图像，得到其直方图
        OpenFileDialog opnDlg = new OpenFileDialog();
        opnDlg.Filter = "所有图像文件 | *.bmp; *.pcx; *.png; *.jpg; *.gif;" +
            "*.tif; *.ico; *.dxf; *.cgm; *.cdr; *.wmf; *.eps; *.emf|" +
            "位图( *.bmp; *.jpg; *.png;...) | *.bmp; *.pcx; *.png; *.jpg; *.gif; *.tif; *.ico|"
            "矢量图( *.wmf; *.eps; *.emf;...) | *.dxf; *.cgm; *.cdr; *.wmf; *.eps; *.emf";
        opnDlg.Title = "打开图像文件";
        opnDlg.ShowHelp = true;
        if (opnDlg.ShowDialog() == DialogResult.OK)
        {
              shapingFileName = opnDlg.FileName;
              try
              {
                  shapingBitmap = (Bitmap)Image.FromFile(shapingFileName);
              }
              Catch (Exception exp)
              {
                  MessageBox.Show(exp.Message);
              }

              // 计算图像灰度等级像素个数
              Rectangle rect = new Rectangle(0, 0, shapingBitmap.Width,
                  shapingBitmap.Height);
              System.Drawing.Imaging.BitmapData bmpData = shapingBitmap.LockBits(rect,
                  System.Drawing.Imaging.ImageLockMode.ReadWrite,
                  shapingBitmap.PixelFormat);
              IntPtr ptr = bmpData.Scan0;
              shapingSize = shapingBitmap.Width * shapingBitmap.Height;
              byte[] grayValues = new byte[shapingSize];
              System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, shapingSize);

              byte temp = 0;
              maxPixel = 0;
              Array.Clear(shapingPixel, 0, 256);
              for (int i = 0; i < shapingSize; i++)
              {
                    temp = grayValues[i];
                    shapingPixel[temp]++;
                    if (shapingPixel[temp] > maxPixel)
                    {
                          maxPixel = shapingPixel[temp];
                    }
              }

              System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, shapingSize);
              shapingBitmap.UnlockBits(bmpData);
        }

        Invalidate();
    }

    private void startShaping_Click(object sender, EventArgs e)
    {
```

```
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}
```

为该窗体添加 1 个 Paint 事件，作用为绘制直方图，并计算它的累计分布函数，代码如下：

```
private void shapingForm_Paint(object sender, PaintEventArgs e)
{
    if (shapingBitmap != null)
    {
        // 在窗体内绘制直方图
        Pen curPen = new Pen(Brushes.Black, 1);
        Graphics g = e.Graphics;
        g.DrawLine(curPen, 50, 240, 320, 240);
        g.DrawLine(curPen, 50, 240, 50, 30);

        g.DrawLine(curPen, 100, 240, 100, 242);
        g.DrawLine(curPen, 150, 240, 150, 242);
        g.DrawLine(curPen, 200, 240, 200, 242);
        g.DrawLine(curPen, 250, 240, 250, 242);
        g.DrawLine(curPen, 300, 240, 300, 242);

        g.DrawString("0", new Font("New Timer", 8), Brushes.Black, new PointF(46, 242));
        g.DrawString("50", new Font("New Timer", 8), Brushes.Black,
            new PointF(92, 242));
        g.DrawString("100", new Font("New Timer", 8), Brushes.Black,
            new PointF(139, 242));
        g.DrawString("150", new Font("New Timer", 8), Brushes.Black,
            new PointF(189, 242));
        g.DrawString("200", new Font("New Timer", 8), Brushes.Black,
            new PointF(239, 242));
        g.DrawString("250", new Font("New Timer", 8), Brushes.Black,
            new PointF(289, 242));

        g.DrawLine(curPen, 48, 40, 50, 40);
        g.DrawString("0", new Font("New Timer", 8), Brushes.Black, new PointF(34, 234));
        g.DrawString(maxPixel.ToString(), new Font("New Timer", 8), Brushes.Black,
            new PointF(18, 34));

        double temp = 0;
        int[] tempArray = new int[256];
        for (int i = 0; i < 256; i++)
        {
            temp = 200 * (double)shapingPixel[i] / (double)maxPixel;
            g.DrawLine(curPen, 50 + i, 240, 50 + i, 240 - (int)temp);

            // 计算该直方图各灰度级的累积分布函数
            if (i != 0)
```

```
                {
                        tempArray[i] = tempArray[i - 1] + shapingPixel[i];
                }
                else
                {
                        tempArray[0] = shapingPixel[0];
                }
                cumHist[i] = (double)tempArray[i] / (double)shapingSize;

           }

           curPen.Dispose();
     }
}
```

为了把该直方图的累积分布函数传递给主窗体，再为该窗体添加 1 个 get 属性访问器，代码如下：

```
public double[] ApplicationP
{
    get
    {
         return cumHist;
    }
}
```

（3）到主窗体，为“直方图匹配”按钮控件添加 Click 事件，代码如下：

```
private void shaping_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化shapingForm 窗体
         shapingForm sForm = new shapingForm();

         if (sForm.ShowDialog() == DialogResult.OK)
         {
               Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
               System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                   System.Drawing.Imaging.ImageLockMode.ReadWrite,
                   curBitmap.PixelFormat);
               IntPtr ptr = bmpData.Scan0;
               int bytes = curBitmap.Width * curBitmap.Height;
               byte[] grayValues = new byte[bytes];
               System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

               byte temp = 0;
               double[] PPixel = new double[256];
               double[] QPixel = new double[256];
               int[] qPixel = new int[256];
               int[] tempArray = new int[256];
               // 计算原图像的各灰度级像素个数
               for (int i = 0; i < grayValues.Length; i++)
               {
```

```
            temp = grayValues[i];
            qPixel[temp]++;
        }

        // 计算该灰度级的累积分布函数
        for (int i = 0; i < 256; i++)
        {
            if (i != 0)
            {
                tempArray[i] = tempArray[i - 1] + qPixel[i];
            }
            else
            {
                tempArray[0] = qPixel[0];
            }

            QPixel[i] = (double)tempArray[i] / (double)bytes;
        }

        // 得到被匹配的直方图的累积分布函数
        PPixel = sForm.ApplicationP;

        double diffA, diffB;
        byte k = 0;
        byte[] mapPixel = new byte[256];
        // 直方图匹配
        for (int i = 0; i < 256; i++)
        {
            diffB = 1;
            for (int j = k; j < 256; j++)
            {
                // 找到两个累计分布函数中最相似的位置
                diffA = Math.Abs(QPixel[i] - PPixel[j]);
                if (diffA - diffB < 1.0E-08)
                {
                    // 记录下差值
                    diffB = diffA;
                    k = (byte)j;
                }
                else
                {
                    // 找到了，记录下位置，并退出内层循环
                    k = (byte)(j - 1);
                    break;
                }
            }

            // 达到最大灰度级，标识未处理灰度级，并退出外循环
            if (k == 255)
            {
                for (int l = i; l < 256; l++)
                {
                    mapPixel[l] = k;
```

```
                        }
                        break;
                    }

                    // 得到映射关系
                    mapPixel[i] = k;
                }

                // 灰度级映射处理
                for (int i = 0; i < bytes; i++)
                {
                    temp = grayValues[i];
                    grayValues[i] = mapPixel[temp];
                }

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运算该段程序。首先打开需要匹配的图像，如图 3.9 所示，为了说明问题，我们把它的直方图也同时显示出来。

然后单击“直方图匹配”按钮，打开直方图匹配对话框。再单击该对话框内的“打开文件”按钮，找到任意一幅想要被匹配的灰度图像，其直方图显示在该对话框内，如图 3.10 所示。

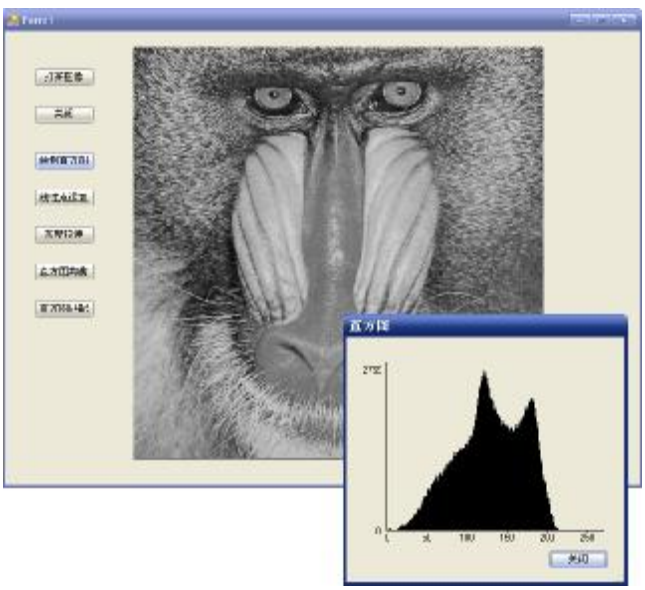


图 3.9　直方图匹配原图

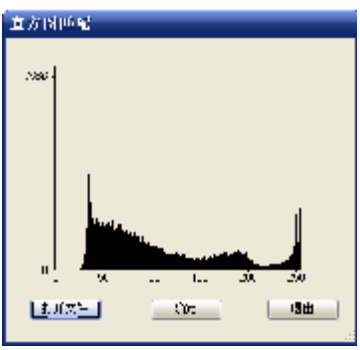


图 3.10　直方图匹配对话框

最后单击“确定”按钮，即完成了直方图的匹配，原图像的直方图按照如图 3.10 所示的

直方图进行匹配，其结果如图 3.11 所示，匹配后的直方图也一并显示出来。

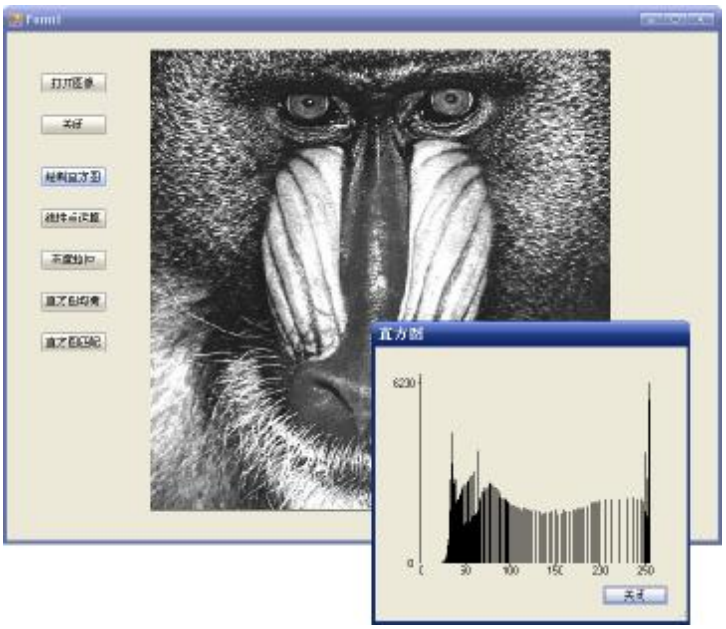


图 3.11　直方图匹配结果

从它的直方图可以看出，与图 3.10 所示的直方图相似，也就是匹配成功。

3.6　小结

本章介绍了利用灰度直方图对图像进行点运算的几种常见方法。点运算应用的是图像中每个独立的像素点，因此无需考虑像素间的相关性。理解、分析和设计图像点运算的基本工具是灰度直方图。通过对直方图的拉伸、均衡化以及匹配等运算，我们就可以得到比较满意的图像。

第 4 章　几何运算

在某种意义上来说，图像的几何运算是与点运算相对立的。它改变了像素之间的空间位置和空间关系，但它没有改变灰度等级值。几何运算需要两个独立的算法：空间变换和灰度值插值。在本章所介绍的几何运算中，应用空间变换算法对图像进行平移、镜像处理，应用空间变换和灰度值插值算法对图像进行缩放和旋转处理。

需要说明的是，在这里，以图像的几何中心作为坐标原点，x 轴由左向右递增，y 轴由上至下递增。因此，在进行图像旋转时，是以图像的几何中心为基准进行旋转的；在进行图像缩放时，也是以图像的几何中心为基准，其上下左右均等地向内收缩或向外扩大的。这种坐标转换会使图像变换更自然。另外，在进行几何运算的时候，保持原图像的尺寸大小不变，如果变换后的图像超出该尺寸，超出部分会被截断，而不足部分会以白色像素填充。

4.1　图像平移

4.1.1　图像平移定义

图像平移就是使图像沿水平方向和垂直方向移动。

具体的算法为：如果把坐标原点$(0, 0)$平移到点(x_0, y_0)处，则变换公式为：

$$x' = x + x_0 \qquad y' = y + y_0 \tag{4.1}$$

(x, y)为原图像坐标，(x', y')为变换后的图像坐标。而图像中的各像素点移动了$\sqrt{x_0^2 + y_0^2}$距离。上式用矩阵形式表示为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4.2}$$

4.1.2 图像平移编程实例

该实例通过设置横向和纵向的平移量，实现了图像的平移。

（1）创建一个“模板主窗体”，并在该窗体内添加一个 Button 控件，其属性修改如表 4 所示。

表 4.1 所修改的属性

控　件	属　性	所修改内容
button1	Name	translation
	Text	图像平移
	Location	37，150

（2）创建 1 个名为 translation 的 Windows 窗体，该窗体用于选择平移量。在该窗体内添加 2 个 Button 控件、2 个 TextBox 控件、2 个 Label 控件和 1 个 GroupBox 控件，其属性修改如表 4.2 所示。

表 4.2 所修改的属性

控　件	属　性	所修改内容
translation	Text	图像平移
	Size	230，220
	ControlBox	False
button1	Name	start
	Text	确定
	Location	24，144
button2	Name	close
	Text	退出
	Location	125，144
TextBox1	Name	xOffset
	Text	0
	Location	121，48
	Size	57，21
TextBox2	Name	yOffset
	Text	0
	Location	121，90
	Size	57，21
label1	Text	沿水平方向：
	Location	27，51
label2	Text	沿垂直方向：
	Location	27，93
groupBox1	Text	图像平移量：
	Location	12，18

	Size	188，112

分别为该窗体内的 2 个 Button 控件添加 Click 事件，为了和主窗体之间传递数据，再添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public string GetXOffset
{
    get
    {
         // 横向平移量
         return xOffset.Text;
    }
}

public string GetYOffset
{
    get
    {
          // 纵向平移量
          return yOffset.Text;
    }
}
```

（3）回到主窗体，为“图像平移”按钮添加 Click 事件，代码如下：

```
private void translation_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
          // 实例化translation 窗体
          translation traForm = new translation();

          if (traForm.ShowDialog() == DialogResult.OK)
          {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] grayValues = new byte[bytes];
                System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

                // 得到两个方向的图像平移量
                int x = Convert.ToInt32(traForm.GetXOffset);
                int y = Convert.ToInt32(traForm.GetYOffset);
```

```
                byte[] tempArray = new byte[bytes];
                // 临时数组初始化为白色（255）像素
                for (int i = 0; i < bytes; i++)
                {
                      tempArray[i] = 255;
                }

                // 平移运算
                for (int i = 0; i < curBitmap.Height; i++)
                {
                      // 保证纵向平移不出界
                      if ((i + y) < curBitmap.Height && (i + y) > 0)
                      {
                            for (int j = 0; j < curBitmap.Width; j++)
                            {
                                  // 保证横向平移不出界
                                  if ((j + x) < curBitmap.Width && (j + x) > 0)
                                  {
                                        // 应用公式(4.1)
                                        tempArray[(j + x) + (i + y) * curBitmap.Width]
                                            grayValues[j + i * curBitmap.Width];
                                  }
                            }
                      }
                }
                //数组复制，返回平移后的图像
                grayValues = (byte[])tempArray.Clone();

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
          }

          Invalidate();
    }
}
```

（4）编译并运行该段程序。首先打开所要平移的图像，如图 4.1 所示。

图 4.1　平移原图像

然后单击“图像平移”按钮，打开图像平移对话框，如图 4.2 所示，填写图像平移量。

单击“确定”按钮，生成平移后的图像，如图 4.3 所示。图像向右平移了 50 个像素单位，向上平移了 80 个像素单位。

图 4.2　平移对话框

图 4.3　平移后图像

需要说明的是，在这里，我们取平移量为整数，如果平移量不是整数，那么就需要运用 4.3 节所介绍的灰度插值法。

.2 图像镜像

4.2.1 图像镜像变换定义

镜像是一个物体相对于一个镜面的复制品。图像镜像分为水平镜像和垂直镜像两种。水平镜像就是将图像左半部分和右半部分以图像垂直中轴线为中心镜像进行对换；垂直镜像就是将图像上半部分和下半部分以图像水平中轴线为中心镜像进行对换。

水平镜像用矩阵形式表示为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & W \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4.3}$$

垂直镜像用矩阵形式表示为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & H \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4.4}$$

其中，W 和 H 分别表示为图像的宽和高。

4.2.2　图像镜像编程实现

该实例实现了图像的水平镜像和垂直镜像。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 4.3 所示。

表 4.3　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	mirror
	Text	图像镜像
	Location	37，196

（2）创建 1 个名为 mirror 的 Windows 窗体，该窗体用于选择是水平镜像还是垂直镜像，在该窗体内添加 2 个 Button 控件、1 个 GroupBox 控件和 2 个 RadioButton 控件，其属性修改如表 4.4 所示。

表 4.4　　所修改的属性

控　　件	属　　性	所修改内容
mirror	Text	图像镜像
	Size	265，260
	ControlBox	False
button1	Name	startMirror
	Text	确定
	Location	28，170
button2	Name	close
	Text	退出
	Location	155，170
gruopBox1	Text	图像镜像
	Size	170，105
	Location	38，34
radioButton1	Name	horMirror
	Text	水平镜像
	Locatio	40，35
	Checked	True
radioButton2	Name	verMirror
	Text	垂直镜像
	Location	40，70

分别为该窗体内的 2 个 Button 控件添加 Click 事件，并再添加 1 个 get 属性访问器，代码如下：

```
private void startMirror_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
```

```
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetMirror
{
     get
     {
         // 得到是水平镜像还是垂直镜像
         return horMirror.Checked;
     }
}
```

（3）回到主窗体，为“图像镜像”按钮添加 Click 事件，代码如下：

```
private void mirror_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化mirror 窗体
         mirror mirForm = new mirror();

         if (mirForm.ShowDialog() == DialogResult.OK)
         {
              Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
              System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                  System.Drawing.Imaging.ImageLockMode.ReadWrite,
                  curBitmap.PixelFormat);
              IntPtr ptr = bmpData.Scan0;
              int bytes = curBitmap.Width * curBitmap.Height;
              byte[] grayValues = new byte[bytes];
              System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

              // 水平中轴
              int halfWidth = curBitmap.Width / 2;
              // 垂直中轴
              int halfHeight = curBitmap.Height / 2;
              byte temp;

              if (mirForm.GetMirror)
              {
                   // 水平镜像处理
                   for (int i = 0; i < curBitmap.Height; i++)
                   {
                        for (int j = 0; j < halfWidth; j++)
                        {
                             // 以水平中轴线为对称轴，两边像素值互换
                             temp = grayValues[i * curBitmap.Width + j];
                             grayValues[i * curBitmap.Width + j] =
                                 grayValues[(i + 1) * curBitmap.Width - 1 - j];
                             grayValues[(i + 1) * curBitmap.Width - 1 - j] = temp;
                        }
                   }
```

```
                }
                else
                {
                    // 垂直镜像处理
                    for (int i = 0; i < curBitmap.Width; i++)
                    {
                        for (int j = 0; j < halfHeight; j++)
                        {
                            // 以垂直中轴线为对称轴，两边像素值互换
                            temp = grayValues[j * curBitmap.Width + i];
                            grayValues[j * curBitmap.Height + i] =
                                grayValues[(curBitmap.Height - j - 1) * curBitmap.Width + i];
                            grayValues[(curBitmap.Height - j - 1) * curBitmap.Width + i] =
                                temp;
                        }
                    }
                }

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运行该段程序。以图 4.1 所示的图像为例。打开图像后，单击“图像镜像”按钮，打开图像镜像对话框，选择“垂直镜像”按钮，如图 4.4 所示。

图 4.4　图像镜像对话框

单击“确定”按钮后，垂直后的图像就显示出来了，如图 4.5 所示。

需要说明的是，本段程序应用的是 512×512 大小的图像，当图像的长或宽不是偶数时，该段程序需要做些改动，不能再应用图像长或宽的 1/2 做为循环的变量。

图 4.5 垂直镜像后的图像

.3　图像缩放

在图像缩放运算和图像旋转运算中，要用到灰度插值算法，因此这里给出灰度插值的两种算法。

4.3.1　图像缩放定义

图像的缩小和放大的定义为：将图像中的某点(x, y)经缩小放大后其位置变为(x', y')，则两者之间的关系是：

$$x' = ax \quad y' = by \tag{4.5}$$

a、b 分别是 x 方向和 y 方向的放大率。a、b 比 1 大时放大；比 1 小时缩小。当 $a = -1$，$b = 1$ 时，会产生一个关于 y 轴对称的镜像；当 $a = 1$，$b = -1$ 时，会产生一个关于 x 轴对称的镜像。用矩阵形式表示为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4.6}$$

4.3.2　灰度插值法

应用公式(4.5)所产生的图像中的像素有可能在原图像中找不到相应的像素点，因为数字图像中的坐标总是整数。这样就必须进行近似处理。一般是应用灰度插值法。它包括最近邻插值和双线性插值。

最近邻插值也称零阶插值，是最简单的插值方法。其做法是令输出像素的灰度值等于离它所映射到的位置最近的输入像素的灰度值。该插值计算虽然十分简单，但它会带来锯齿形的边，图像中也会出现孔洞和重叠。

双线性插值也称一阶插值，该方法是求到相邻的 4 个方格上点的距离之比，用这个比率和 4 个邻点像素的灰度值进行灰度插值，具体方法如下。

对于一个目的像素，设置坐标通过反向变换得到的浮点坐标为$(i+u, j+v)$，其中 i、j 均为非负整数，p、q 为[0, 1)区间的浮点数，则这个像素的值 $f(i+p, j+q)$可由原图像中坐标为(i, j)、$(i+1, j)$、$(i, j+1)$、$(i+1, j+1)$所对应的周围 4 个像素的值决定，即：

$$f(i+p, j+q) = (1-p)(1-q) f(i, j) + (1-p) q f(i, j+1) + p(1-q) f(i+1, j) + pq f(i+1, j+1) \tag{4.7}$$

其中 $f(i, j)$表示源图像(i, j)处的像素值。

双线性插值法计算量大，但缩放后图像质量高，不会出现像素值不连续的情况。由于双线性插值具有低通滤波器的性质，使高频分量受损，所以可能会使图像轮廓在一定程度上变得模糊

4.3.3　图像缩放编程实例

该实例应用最近邻插值法和双线性插值法实现图像的缩放。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 4.5 所示。

表 4.5 所修改的属性

控　件	属　性	所修改内容
button1	Name	zoom
	Text	图像缩放
	Location	37，242

（2）创建 1 个名为 zoom 的 Windows 窗体，该窗体用于选择缩放量及用何种灰度插值法。为该窗体添加 2 个 Button 控件、1 个 GroupBox 控件、2 个 RadioButton 控件、2 个 Label 控件和 2 个 TextBox 控件，其属性修改如表 4.6 所示。

表 4.6 所修改的属性

控　件	属　性	所修改内容
zoom	Text	图像缩放
	Size	390，300
	ControlBox	False
groupBox1	Text	灰度插值
	Location	28，24
	Size	319，92
radioButton1	Name	nearestNeigh
	Text	最近邻插值
	Location	26，41
	Checked	true
radioButton2	Name	bilinear
	Text	双线性插值
	Location	190，41
label1	Text	横向缩放量
	Location	26，149
label2	Text	纵向缩放量
	Location	195，149
textBox1	Name	xZoom
	Text	1
	Location	109，146
	Size	54，21
textBox2	Name	yZoom
	Text	1
	Location	278，146
	Size	54，21
button1	Name	startZoom
	Text	确定

	Location	54，205

续表

控　件	属　性	所修改内容
button2	Name	close
	Text	退出
	Location	247，205

分别为该窗体内的 2 个 Button 控件添加 Click 事件，并再添加 3 个 get 属性访问器，代码如下：

```
private void startZoom_Click(object sender, EventArgs e)
{
    // 缩放量不能为0
    if (xZoom.Text == "0" || yZoom.Text == "0")
    {
        MessageBox.Show("缩放量不能为0! \n请重新正确填写。", "警告",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    else
    {
        this.DialogResult = DialogResult.OK;
    }
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetNearOrBil
{
     get
     {
         // 得到是最近邻插值法还是双线性插值法
         return nearestNeigh.Checked;
     }
}

public string GetXZoom
{
    get
    {
         // 得到横向缩放量
         return xZoom.Text;
    }
}

public string GetYZoom
{
    get
    {
         //得到纵向缩放量
         return yZoom.Text;
```

```
    }
}
```

（3）回到主窗体，为“图像缩放”按钮控件添加 Click 事件代码，代码如下：

```
private void zoom_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化zoom 窗体
        zoom zoomForm = new zoom();

        if (zoomForm.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            // 得到横向和纵向缩放量
            double x = Convert.ToDouble(zoomForm.GetXZoom);
            double y = Convert.ToDouble(zoomForm.GetYZoom);

            // 图像的几何中心
            int halfWidth = (int)(curBitmap.Width / 2);
            int halfHeight = (int)(curBitmap.Height / 2);

            int xz = 0;
            int yz = 0;
            int tempWidth = 0;
            int tempHeight = 0;
            byte[] tempArray = new byte[bytes];

            if (zoomForm.GetNearOrBil == true)
            {
                // 最近邻插值法
                for (int i = 0; i < curBitmap.Height; i++)
                {
                    for (int j = 0; j < curBitmap.Width; j++)
                    {
                        // 以图像的几何中心为坐标原点进行坐标变换
                        // 按逆向映射法得到输入图像的坐标
                        tempHeight = i - halfHeight;
                        tempWidth = j - halfWidth;

                        // 在不同象限进行四舍五入处理
                        if (tempWidth > 0)
                        {
                            xz = (int)(tempWidth / x + 0.5);
                        }
                        else
                        {
```

```
                        xz = (int)(tempWidth / x - 0.5);
                    }
                    if (tempHeight > 0)
                    {
                        yz = (int)(tempHeight / y + 0.5);
                    }
                    else
                    {
                        yz = (int)(tempHeight / y - 0.5);
                    }

                    // 坐标逆变换
                    tempWidth = xz + halfWidth;
                    tempHeight = yz + halfHeight;
                    // 得到输出图像像素值
                    if (tempWidth < 0 || tempWidth >= curBitmap.Width ||
                        tempHeight < 0 || tempHeight >= curBitmap.Height)
                    {
                        // 缩放后留下的空白部分用白色像素代替
                        tempArray[i * curBitmap.Width + j] = 255;
                    }
                    else
                    {
                        tempArray[i * curBitmap.Width + j] =
                            grayValues[tempHeight * curBitmap.Width +
                            tempWidth];
                    }
                }
            }
        }
        else
        {
            // 双线性插值法
            double tempX, tempY, p, q;
            for (int i = 0; i < curBitmap.Height; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    // 以图像的几何中心为坐标原点进行坐标变换
                    // 按逆向映射法得到输入图像的坐标
                    tempHeight = i - halfHeight;
                    tempWidth = j - halfWidth;
                    tempX = tempWidth / x;
                    tempY = tempHeight / y;

                    // 在不同象限进行取整处理
                    if (tempWidth > 0)
                    {
                        xz = (int)tempX;
                    }
                    else
                    {
                        xz = (int)(tempX - 1);
                    }
                    if (tempHeight > 0)
```

```
                                    {
                                          yz = (int)tempY;
                                    }
                                    else
                                    {
                                          yz = (int)(tempY - 1);
                                    }

                                    // 得到公式(4.7)中的变量p和q
                                    p = tempX - xz;
                                    q = tempY - yz;

                                    // 坐标逆变换
                                    tempWidth = xz + halfWidth;
                                    tempHeight = yz + halfHeight;

                                    if (tempWidth < 0 || (tempWidth + 1) >= curBitmap.Width |
                                          tempHeight < 0 || (tempHeight + 1) >=
urBitmap.Height)
                                    {
                                          // 缩放后留下的空白部分用白色像素代替
                                          tempArray[i * curBitmap.Width + j] = 255;
                                    }
                                    else
                                    {
                                          // 应该公式(4.7)得到双线性插值
                                          tempArray[i * curBitmap.Width + j] = (byte)((1.0 - q)
                                             ((1.0 - p) * grayValues[tempHeight * curBitmap
idth +
                                             tempWidth] + p * grayValues[tempHeight *
                                             curBitmap.Width + tempWidth + 1]) + q * ((1.0 - p)
                                             grayValues[(tempHeight + 1) * curBitmap.Width
                                             tempWidth] + p * grayValues[(tempHeight + 1) *
                                             curBitmap.Width + 1 + tempWidth]));
                                    }
                              }
                        }

                  }

                  grayValues = (byte[])tempArray.Clone();

                  System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                  curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
      }
   }
```

（4）编译并运行该段程序。首先打开如图 4.1 所示的图像，然后单击“图像缩放”按钮
了开图像缩放对话框，如图 4.6 所示，填写各个参数。

单击“确定”按钮，得到了用双线性插值法对原图像进行横向放大 2.5 倍、纵向缩小 0.
等的图像，如图 4.7 所示。

为了比较最近邻插值法和双线性插值法的性能，我们分别用这两种方法对如图 4.1 所示的图像进行横向和纵向都放大 5 倍的处理，分别得到了图 4.8 和图 4.9。

通过对比这两幅图像，可以明显地看出，用双线性插值比用最近邻插值处理图像更逼真

图 4.6 图像缩放对话框

图 4.7 缩放处理后图像

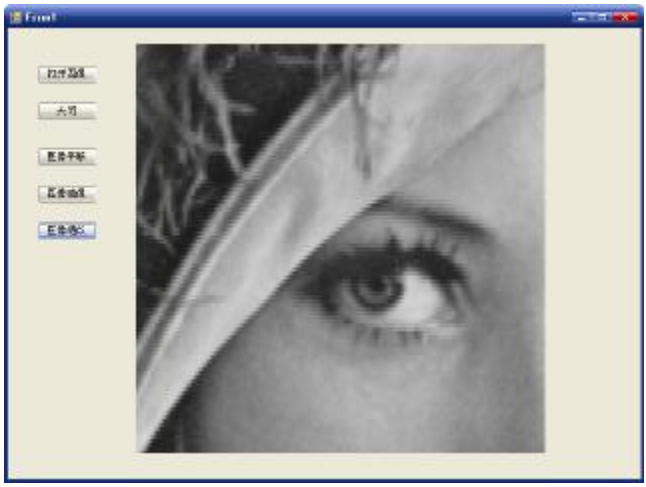


图 4.8 最近邻插值

图 4.9 双线性插值

.4　图像旋转

4.4.1　图像旋转定义

如果平面的所有点绕原点逆时针旋转$\theta°$，则它的变换公式为：

$$x' = x\cos q + y\sin q \quad y' = -x\sin q + y\cos q \tag{4.8}$$

其中，(x, y)为原图坐标，(x', y')为旋转后的坐标。它的逆变换公式为：

$$x = x'\cos q - y'\sin q \quad y = x'\sin q + y'\cos q \tag{4.9}$$

它用矩阵形式表示为：

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos q & -\sin q & 0 \\ \sin q & \cos q & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{4.10}$$

同理，旋转后得到的图像像素也有可能在原图像中找不到相应的像素点，因此旋转处理也要用到插值法。由于双线性插值法在图像处理性能上要好过最近邻插值，因此，我们只应用双线性插值这一种方法对图像进行旋转处理。

4.4.2　图像旋转编程实现

该实例实现了任意角度的图像旋转。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 4.7 所示。

表 4.7　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	rotation
	Text	图像旋转
	Location	37，288

（2）创建 1 个名为 rotation 的 Windows 窗体，该窗体用于选择旋转的角度。为该窗体添加 2 个 Button 控件、1 个 Label 控件和 1 个 TextBox 控件，其属性修改如表 4.8 所示。

表 4.8　　所修改的属性

控　　件	属　　性	所修改内容
rotation	Text	图像旋转
	Size	285，204
	ControlBox	False
button1	Name	startRot
	Text	确定

	Location	39，107

续表

控　　件	属　　性	所修改内容
	Name	close
button2	Text	关闭
	Location	159，107
	Text	旋转角度(°)
label1	Location	37，46
	Name	degree
textBox1	Text	0
	Location	123，43

分别为该窗体内的 2 个 Button 控件添加 Click 事件，并再添加 1 个 get 属性访问器，代码如下：

```
private void button1_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void button2_Click(object sender, EventArgs e)
{
    this.Close();
}

public string GetDegree
{
    get
    {
          // 得到所要旋转的角度
          return textBox1.Text;
    }
}
```

（3）回到主窗体，为“图像旋转”按钮控件添加 Click 事件，代码如下：

```
private void rotation_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化rotation 窗体
         rotation rotForm = new rotation();

         if (rotForm.ShowDialog() == DialogResult.OK)
         {
             Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
             System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                 System.Drawing.Imaging.ImageLockMode.ReadWrite,
                 curBitmap.PixelFormat);
             IntPtr ptr = bmpData.Scan0;
             int bytes = curBitmap.Width * curBitmap.Height;
             byte[] grayValues = new byte[bytes];
             System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
```

```
// 得到所要旋转的角度
int degree = Convert.ToInt32(rotForm.GetDegree);
// 转换为弧度
double radian = degree * Math.PI / 180.0;
// 计算它的正弦和余弦
double mySin = Math.Sin(radian);
double myCos = Math.Cos(radian);
// 图像几何中心
int halfWidth = (int)(curBitmap.Width / 2);
int halfHeight = (int)(curBitmap.Height / 2);
int xz = 0;
int yz = 0;
int tempWidth = 0;
int tempHeight = 0;
byte[] tempArray = new byte[bytes];
double tempX, tempY, p, q;

// 双线性插值旋转
for (int i = 0; i < curBitmap.Height; i++)
{
      for (int j = 0; j < curBitmap.Width; j++)
      {
            // 以图像的几何中心为坐标原点
            tempHeight = i - halfHeight;
            tempWidth = j - halfWidth;
            // 应用公式(4.9)
            tempX = tempWidth * myCos - tempHeight * mySin;
            tempY = tempHeight * myCos + tempWidth * mySin;

            // 在不同象限进行取整处理
            if (tempWidth > 0)
            {
                  xz = (int)tempX;
            }
            else
            {
                  xz = (int)(tempX - 1);
            }
            if (tempHeight > 0)
            {
                  yz = (int)tempY;
            }
            else
            {
                  yz = (int)(tempY - 1);
            }

            // 得到公式(4.7)中的变量p和q
            p = tempX - xz;
            q = tempY - yz;

            // 坐标逆变换
            tempWidth = xz + halfWidth;
            tempHeight = yz + halfHeight;

            if (tempWidth < 0 || (tempWidth + 1) >= curBitmap.Width ||
                  tempHeight < 0 || (tempHeight + 1) >= curBitmap.Height)
```

```
                        {
                            // 旋转后留下的空白部分用白色像素代替
                            tempArray[i * curBitmap.Width + j] = 255;
                        }
                        else
                        {
                            // 应用公式(4.7)得到双线性插值
                            tempArray[i * curBitmap.Width + j] = (byte)((1.0 - q) * ((1.
                                - p) *
                                grayValues[tempHeight * curBitmap.Width + tempWidth] +
                                p * grayValues[tempHeight * curBitmap.Width +
                                tempWidth + 1]) + q * ((1.0 - p) * grayValues[(tempHeight
                                * curBitmap.Width + tempWidth] + p *
                                grayValues[(tempHeight + 1) * curBitmap.Width + 1 +
                                tempWidth]));
                        }

                    }
                }

                grayValues = (byte[])tempArray.Clone();

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运行该段程序。下面仍然以图 4.1 所示的图像为例打开这个图像，然后单击“图像旋转”按钮，打开图像旋转对话框，让图像逆时针旋转 30°，如图 4.10 所示。

图 4.10 图像旋转对话框

然后单击“确定”按钮，得到如图 4.11 所示的界面。

需要说明的是，当图像旋转时可能会引入点噪声，使得图像中出现一些小白点和小黑点。

图 4.11　图像旋转结果

4.5 小结

本章介绍了图像几何算法中的图像平移、图像镜像、图像缩放和图像旋转的定义及编程实现。由于在图像几何运算中需要灰度插值算法，因此，我们也介绍了两种常用的灰度插值最近邻插值和双线性插值。

第 5 章　数学形态学图像处理

数学形态学是一门建立在严格数学理论基础上的学科，它已构成了一种新型的图像处理方法和理论，并成为计算机数字图像处理的一个主要研究领域。形态学图像处理的基本思想是利用一个称作结构元素的“探针”收集图像的信息。当探针在图像中不断移动时，便可考虑图像各个部分间的相互关系，从而了解图像的结构特征。结构元素是最重要、最基本的概念，它在形态变换中的作用相当于信号处理中的“滤波窗口”。对同一幅图像，结构元素不同则处理后的结果也不同。二值图像形态学应用中，结构元素选取的原则往往是具有旋转不变性，或者至少镜像不变性的。也就是说，结构元素的原点在其几何中心处，并且其他像素关于该原点呈对称状。常用到的水平单列、垂直单列、十字形以及方形，如图 5.1 所示。本章所用到的结构元素也是这 4 种。

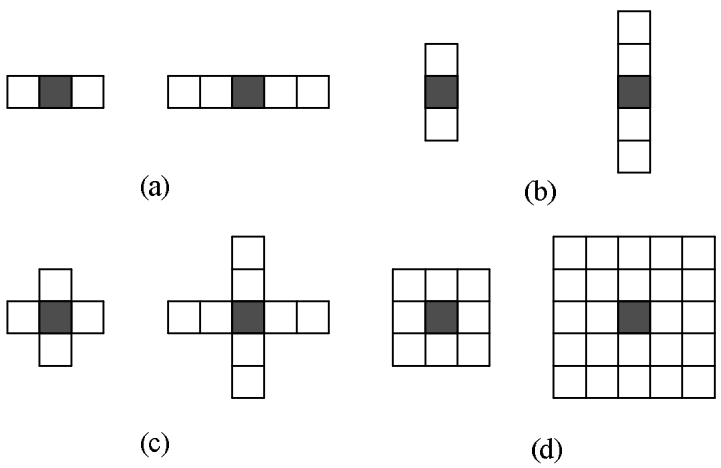


图 5.1　常见的结构元素（灰度区域为该结构元素的原点）

本章主要介绍数学形态学对二值图像的各种处理，如腐蚀运算、膨胀运算、开运算、闭运算和击中击不中变换等。当然，数学形态学也可以对灰度图像进行各种算子运算，这些运算将在以后的章节中结合各种具体应用再进行详细介绍。

需要说明的是，虽然本章应用的是二值图像，但为了使程序保持一致性，本章所应用的

图像仍然是 8 位灰度图像，0 表示黑色，255 表示白色，它代替二值图像中的 1。

5.1 图像腐蚀运算

5.1.1 图像腐蚀运算定义

二值图像腐蚀运算的数学表达式为：

$$g(x,y)=\text{erode}\left[f(x,y),\text{B}\right]=\text{AND}\left[\text{B}f(x,y)\right] \tag{5.1}$$

其中，$g(x,y)$为腐蚀后的二值图像，$f(x,y)$为原二值图像，B 为结构元素。B$f(x,y)$定义为

$$\text{B}f(x,y)=\left\{f(x-bx,y-by),(bx,by)\in \text{B}\right\} \tag{5.2}$$

算子 AND(x_1,⋯,x_n)定义为：当且仅当 x_1=⋯= x_n=1 时，AND(x_1,⋯,x_n)等于 1；否则为 0。

5.1.2 图像腐蚀运算编程实例

该实例应用如图 5.1 所示的结构元素实现图像的腐蚀运算。

（1）创建一个“模板主窗体”，并在该窗体内添加 4 个 Button 控件，由于腐蚀运算、膨胀运算、开运算和闭运算是形态学中的 4 种基本运算，所以把它们的控件也一并添加进来，其属性修改如表 5.1 所示。

表 5.1 所修改的属性

控　件	属　性	所修改内容
button1	Name	erode
	Text	图像腐蚀运算
	Location	37，150
button2	Name	dilate
	Text	图像膨胀运算
	Location	37，196
button3	Name	opening
	Text	开运算
	Location	37，242
button4	Name	closing
	Text	闭运算
	Location	37，288

（2）创建 1 个名为 struction 的 Windows 窗体，该窗体用于选择结构元素形状，它也用形态学的其他 3 种基本运算。在该窗体内添加 2 个 Button 控件、1 个 PictureBox 控件、1 个 Label 控件和 2 个 GroupBox 控件，并分别在这 2 个 GroupBox 控件内添加 4 个和 2 个 RadioButton 控件，其属性修改如表 5.2 所示。

表 5.2 所修改的属性

控　件	属　性	所修改内容
struction	Size	420，380
	ControlBox	False
button1	Name	start
	Text	确定
	Location	193，289
button2	Name	close
	Text	退出
	Location	298，289
label1	Text	结构元素预览
	Location	191，22
pictureBox1	Name	struPic
	Location	193，46
	Size	180，180
	SizeMode	Zoom
groupBox1	Text	结构元素的形状
	Location	29，22
	Size	131，179
groupBox2	Text	结构元素的位数
	Location	29，222
	Size	131，99
radioButton1	Name	row
	Text	水平方向
	Location	6，34
	Checked	True
radioButton2	Name	column
	Text	垂直方向
	Location	6，67
radioButton3	Name	cross
	Text	“十”字形状
	Location	6，100
radioButton4	Name	square
	Text	方形
	Location	6，133
radioButton5	Name	three
	Text	3
	Location	16，34
	Checked	True
radioButton6	Name	five
	Text	5

	Location	16，67

为该窗体内的 2 个 Button 控件添加 Click 事件，为 6 个 radioButton 控件添加 CheckedChanged 事件，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void row_CheckedChanged(object sender, EventArgs e)
{
     // 设置标识变量
     temp = (byte)((temp & 0xf0) | 0x01);
     // 显示结构元素
     showPic(temp);
}

private void column_CheckedChanged(object sender, EventArgs e)
{
     // 设置标识变量
     temp = (byte)((temp & 0xf0) | 0x02);
     // 显示结构元素
     showPic(temp);
}

private void cross_CheckedChanged(object sender, EventArgs e)
{
     // 设置标识变量
     temp = (byte)((temp & 0xf0) | 0x04);
     // 显示结构元素
     showPic(temp);
}

private void square_CheckedChanged(object sender, EventArgs e)
{
     // 设置标识变量
     temp = (byte)((temp & 0xf0) | 0x08);
     // 显示结构元素
     showPic(temp);
}

private void three_CheckedChanged(object sender, EventArgs e)
{
     // 设置标识变量
     temp = (byte)((temp & 0x0f) | 0x10);
     // 显示结构元素
     showPic(temp);
}
```

```
private void five_CheckedChanged(object sender, EventArgs e)
{
     // 设置标识变量
     temp = (byte)((temp & 0x0f) | 0x20);
     // 显示结构元素
     showPic(temp);
}
```

其中 temp 为 byte 型变量，用于标识结构元素，每一位所表示的含义如表 5.3 所示。

表 5.3　　temp变量含义

位	7	6	5	4	3	2	1	0
含义	保留	保留	结构元素位数		结构元素形状			
			5	3	方形	“十”字形	垂直形状	水平形状

showPic()为定义的一个方法，目的是在 pictureBox 控件内显示结构元素的图片（如图 5.
所示），这些图片文件存放在该程序所在目录下的 images 文件夹内，代码如下：

```
/*****************************************
用于显示结构元素形状
pic: 结构元素标识
*****************************************/
private void showPic(byte pic)
{
    byte sPic = pic;

    switch (sPic)
    {
        case 0x11:
            // 3 位水平形状
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\row3.jpg");
            break;
        case 0x12:
            // 3 位垂直形状
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\col3.jpg");
            break;
        case 0x14:
            // 3 位“十”字形
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\cross3.jpg");
            break;
        case 0x18:
            // 3 位方形
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\square3.jpg");
            break;
        case 0x21:
            // 5 位水平形状
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\row5.jpg");
            break;
        case 0x22:
```

```
            // 5 位垂直形状
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\col5.jpg");
            break;
        case 0x24:
            // 5 位十字形
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\cross5.jpg");
            break;
        case 0x28:
            // 5 位方形
            struPic.Image = System.Drawing.Image.FromFile(Application.StartupPath +
                "\\images\\square5.jpg");
            break;
        default:
            break;
    }
}
```

为了和主窗体之间传递所选择的结构元素，再添加一个 get 属性访问器，代码如下：

```
public byte GetStruction
{
    get
    {
         // 得到结构元素标识
         return temp;
    }
}
```

在该窗体的构造函数内为变量和方法初始化，代码如下：

```
// 3 位水平形状
temp = 0x11;
showPic(temp);
```

（3）然后回到主窗体，为该窗体的“图像腐蚀”按钮控件添加 Click 事件，代码如下：

```
private void erode_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
          // 实例化struction
          struction struForm = new struction();
          // 设置从窗体标题
          struForm.Text = "腐蚀运算结构元素";

          if (struForm.ShowDialog() == DialogResult.OK)
          {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height)
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] grayValues = new byte[bytes];
                System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes)

                // 得到结构元素
```

```
byte flagStru = struForm.GetStruction;

byte[] tempArray = new byte[bytes];
for (int i = 0; i < bytes; i++)
{
      tempArray[i] = 255;
}

switch (flagStru)
{
    case 0x11:
       // 3位水平方向结构元素
       for (int i = 0; i < curBitmap.Height; i++)
       {
             for (int j = 1; j < curBitmap.Width - 1; j++)
             {
                   if (grayValues[i * curBitmap.Width + j] == 0 &&
                         grayValues[i * curBitmap.Width + j + 1] == 0 &
                         grayValues[i * curBitmap.Width + j - 1] == 0)
                   {
                         tempArray[i * curBitmap.Width + j] = 0;
                   }
             }
       }
       break;
    case 0x21:
       // 5位水平方向结构元素
       for (int i = 0; i < curBitmap.Height; i++)
       {
             for (int j = 2; j < curBitmap.Width - 2; j++)
             {
                   if (grayValues[i * curBitmap.Width + j] == 0 &&
                         grayValues[i * curBitmap.Width + j + 1] == 0 &
                         grayValues[i * curBitmap.Width + j - 1] == 0 &
                         grayValues[i * curBitmap.Width + j + 2] == 0 &
                         grayValues[i * curBitmap.Width + j - 2] == 0)
                   {
                         tempArray[i * curBitmap.Width + j] = 0;
                   }
             }
       }
       break;
    case 0x12:
       // 3位垂直方向结构元素
       for (int i = 1; i < curBitmap.Height - 1; i++)
       {
             for (int j = 0; j < curBitmap.Width; j++)
             {
                   if (grayValues[i * curBitmap.Width + j] == 0 &&
                         grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                         grayValues[(i + 1) * curBitmap.Width + j] == 0
                   {
                         tempArray[i * curBitmap.Width + j] = 0;
                   }
             }
       }
```

```
                    break;
                case 0x22:
                    // 5 位垂直方向结构元素
                    for (int i = 2; i < curBitmap.Height - 2; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 &&
                                grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i - 2) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 2) * curBitmap.Width + j] == 0
                            {
                                tempArray[i * curBitmap.Width + j] = 0;
                            }
                        }
                    }
                    break;
                case 0x14:
                    // 3 位“十”字形状结构元素
                    for (int i = 1; i < curBitmap.Height - 1; i++)
                    {
                        for (int j = 1; j < curBitmap.Width - 1; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 &&
                                grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                                grayValues[i * curBitmap.Width + j + 1] == 0 &
                                grayValues[i * curBitmap.Width + j - 1] == 0)
                            {
                                tempArray[i * curBitmap.Width + j] = 0;
                            }
                        }
                    }
                    break;
                case 0x24:
                    // 5 位“十”字形状结构元素
                    for (int i = 2; i < curBitmap.Height - 2; i++)
                    {
                        for (int j = 2; j < curBitmap.Width - 2; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 &&
                                grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i - 2) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 2) * curBitmap.Width + j] == 0 &
                                grayValues[i * curBitmap.Width + j + 1] == 0 &
                                grayValues[i * curBitmap.Width + j - 1] == 0 &
                                grayValues[i * curBitmap.Width + j + 2] == 0 &
                                grayValues[i * curBitmap.Width + j - 2] == 0)
                            {
                                tempArray[i * curBitmap.Width + j] = 0;
                            }
                        }
                    }
                    break;
```

```
                        case 0x18:
                            // 3位方形结构元素
                            for (int i = 1; i < curBitmap.Height - 1; i++)
                            {
                                  for (int j = 1; j < curBitmap.Width - 1; j++)
                                  {
                                        if (grayValues[i * curBitmap.Width + j] == 0 &&
                                              grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                                              grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                                              grayValues[i * curBitmap.Width + j + 1] == 0 &
                                              grayValues[i * curBitmap.Width + j - 1] == 0 &
                                              grayValues[(i - 1) * curBitmap.Width + j - 1] == 0 &
                                              grayValues[(i + 1) * curBitmap.Width + j - 1] == 0 &
                                              grayValues[(i - 1) * curBitmap.Width + j + 1] == 0 &
                                              grayValues[(i + 1) * curBitmap.Width + j + 1] == 0
                                        {
                                              tempArray[i * curBitmap.Width + j] = 0;
                                        }
                                  }
                            }
                            break;
                        case 0x28:
                            // 5位方形结构元素
                            for (int i = 2; i < curBitmap.Height - 2; i++)
                            {
                                  for (int j = 2; j < curBitmap.Width - 2; j++)
                                  {
                                        if (grayValues[(i - 2) * curBitmap.Width + j - 2] == 0 &
                                              grayValues[(i - 2) * curBitmap.Width + j - 1] == 0 &
                                              grayValues[(i - 2) * curBitmap.Width + j] == 0 &&
                                              grayValues[(i - 2) * curBitmap.Width + j + 1] == 0 &
                                              grayValues[(i - 2) * curBitmap.Width + j + 2] == 0 &
                                              grayValues[(i - 1) * curBitmap.Width + j - 2] == 0 &
                                              grayValues[(i - 1) * curBitmap.Width + j - 1] == 0 &
                                              grayValues[(i - 1) * curBitmap.Width + j] == 0 &&
                                              grayValues[(i - 1) * curBitmap.Width + j + 1] == 0 &
                                              grayValues[(i - 1) * curBitmap.Width + j + 2] == 0 &
                                              grayValues[i * curBitmap.Width + j - 2] == 0 &
                                              grayValues[i * curBitmap.Width + j - 1] == 0 &
                                              grayValues[i * curBitmap.Width + j] == 0 &&
                                              grayValues[i * curBitmap.Width + j + 1] == 0 &
                                              grayValues[i * curBitmap.Width + j + 2] == 0 &
                                              grayValues[(i + 2) * curBitmap.Width + j - 2] == 0 &
                                              grayValues[(i + 2) * curBitmap.Width + j - 1] == 0 &
                                              grayValues[(i + 2) * curBitmap.Width + j] == 0 &&
                                              grayValues[(i + 2) * curBitmap.Width + j + 1] == 0 &
                                              grayValues[(i + 2) * curBitmap.Width + j + 2] == 0 &
                                              grayValues[(i + 1) * curBitmap.Width + j - 2] == 0 &
                                              grayValues[(i + 1) * curBitmap.Width + j - 1] == 0 &
                                              grayValues[(i + 1) * curBitmap.Width + j] == 0 &&
                                              grayValues[(i + 1) * curBitmap.Width + j + 1] == 0 &
                                              grayValues[(i + 1) * curBitmap.Width + j + 2] == 0
                                        {
                                              tempArray[i * curBitmap.Width + j] = 0;
                                        }
                                  }
```

```
                        }
                        break;
                    default:
                        MessageBox.Show("错误的结构元素! ");
                        break;
                }

                grayValues = (byte[])tempArray.Clone();

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes)
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运行该段程序。打开原二值图像，如图 5.2 所示。

单击“图像腐蚀”按钮，打开结构元素窗体，如图 5.3 所示选择结构元素。

图 5.2 原二值图像

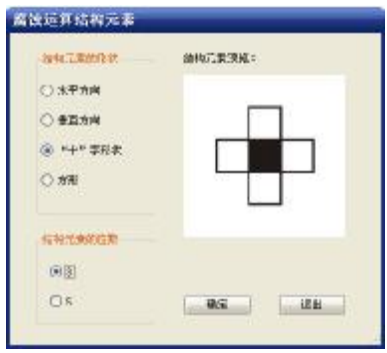


图 5.3 腐蚀运算结构元素

单击“确定”按钮，则经腐蚀运算后的二值图像就显示出来，如图 5.4 所示。

图 5.4　图像腐蚀运算结果

5.2 图像膨胀运算

5.2.1 图像膨胀运算定义

二值图像膨胀运算的数学表达式为：

$$g(x,y)=dilate\left[f(x,y),\mathrm{B}\right]=\mathrm{OR}\left[\mathrm{B}f(x,y)\right] \tag{5.3}$$

其中，$g(x,y)$为膨胀后的二值图像，$f(x,y)$为原二值图像，B 为结构元素。B$f(x,y)$定义为公式(5.2)所示。算子 OR(x_1,⋯,x_n)定义为：当且仅当 x_1=⋯=x_n=0 时，OR(x_1,⋯,x_n)等于 0；否则为 1。

5.2.2 图像膨胀运算编程实例

该实例应用如图 5.1 所示的结构元素实现图像的膨胀运算。

（1）为“图像膨胀”按钮控件添加 Click 事件，代码如下：

```
private void erode_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化struction
         struction struForm = new struction();
         // 设置从窗体标题
         struForm.Text = "膨胀运算结构元素";

         if (struForm.ShowDialog() == DialogResult.OK)
         {
               Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
               System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                   System.Drawing.Imaging.ImageLockMode.ReadWrite,
                   curBitmap.PixelFormat);
               IntPtr ptr = bmpData.Scan0;
               int bytes = curBitmap.Width * curBitmap.Height;
               byte[] grayValues = new byte[bytes];
               System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

               // 得到结构元素
               byte flagStru = struForm.GetStruction;

               byte[] tempArray = new byte[bytes];
               for (int i = 0; i < bytes; i++)
               {
                     tempArray[i] = 255;
               }

               switch (flagStru)
               {
                   case 0x11:
                       // 3 位水平方向结构元素
                       for (int i = 0; i < curBitmap.Height; i++)
```

```
                    {
                        for (int j = 1; j < curBitmap.Width - 1; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 ||
                                grayValues[i * curBitmap.Width + j + 1] == 0 |
                                grayValues[i * curBitmap.Width + j - 1] == 0)
                            {
                                tempArray[i * curBitmap.Width + j] = 0;
                            }
                        }
                    }
                    break;
                case 0x21:
                    // 5位水平方向结构元素
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        for (int j = 2; j < curBitmap.Width - 2; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 ||
                                grayValues[i * curBitmap.Width + j + 1] == 0 |
                                grayValues[i * curBitmap.Width + j - 1] == 0 |
                                grayValues[i * curBitmap.Width + j + 2] == 0 |
                                grayValues[i * curBitmap.Width + j - 2] == 0)
                            {
                                tempArray[i * curBitmap.Width + j] = 0;
                            }
                        }
                    }
                    break;
                case 0x12:
                    // 3位垂直方向结构元素
                    for (int i = 1; i < curBitmap.Height - 1; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 ||
                                grayValues[(i - 1) * curBitmap.Width + j] == 0 |
                                grayValues[(i + 1) * curBitmap.Width + j] == 0
                            {
                                tempArray[i * curBitmap.Width + j] = 0;
                            }
                        }
                    }
                    break;
                case 0x22:
                    // 5位垂直方向结构元素
                    for (int i = 2; i < curBitmap.Height - 2; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
                             if (grayValues[i * curBitmap.Width + j] == 0 ||
                                 grayValues[(i - 1) * curBitmap.Width + j] == 0 |
                                 grayValues[(i + 1) * curBitmap.Width + j] == 0 |
                                 grayValues[(i - 2) * curBitmap.Width + j] == 0 |
                                 grayValues[(i + 2) * curBitmap.Width + j] == 0
```

```
                        {
                              tempArray[i * curBitmap.Width + j] = 0;
                        }
                  }
            }
            break;
      case 0x14:
            // 3 位“十”字形状结构元素
            for (int i = 1; i < curBitmap.Height - 1; i++)
            {
                  for (int j = 1; j < curBitmap.Width - 1; j++)
                  {
                        if (grayValues[i * curBitmap.Width + j] == 0 ||
                              grayValues[(i - 1) * curBitmap.Width + j] == 0 |
                              grayValues[(i + 1) * curBitmap.Width + j] == 0 |
                              grayValues[i * curBitmap.Width + j + 1] == 0 |
                              grayValues[i * curBitmap.Width + j - 1] == 0)
                        {
                              tempArray[i * curBitmap.Width + j] = 0;
                        }
                  }
            }
            break;
      case 0x24:
            // 5 位“十”字形状结构元素
            for (int i = 2; i < curBitmap.Height - 2; i++)
            {
                  for (int j = 2; j < curBitmap.Width - 2; j++)
                  {
                        if (grayValues[i * curBitmap.Width + j] == 0 ||
                              grayValues[(i - 1) * curBitmap.Width + j] == 0 |
                              grayValues[(i + 1) * curBitmap.Width + j] == 0 |
                              grayValues[(i - 2) * curBitmap.Width + j] == 0 |
                              grayValues[(i + 2) * curBitmap.Width + j] == 0 |
                              grayValues[i * curBitmap.Width + j + 1] == 0 |
                              grayValues[i * curBitmap.Width + j - 1] == 0 |
                              grayValues[i * curBitmap.Width + j + 2] == 0 |
                              grayValues[i * curBitmap.Width + j - 2] == 0)
                        {
                              tempArray[i * curBitmap.Width + j] = 0;
                        }
                  }
            }
            break;
      case 0x18:
            // 3 位方形结构元素
            for (int i = 1; i < curBitmap.Height - 1; i++)
            {
                  for (int j = 1; j < curBitmap.Width - 1; j++)
                  {
                        if (grayValues[i * curBitmap.Width + j] == 0 ||
                              grayValues[(i - 1) * curBitmap.Width + j] == 0 |
                              grayValues[(i + 1) * curBitmap.Width + j] == 0 |
                              grayValues[i * curBitmap.Width + j + 1] == 0 |
                              grayValues[i * curBitmap.Width + j - 1] == 0 |
```

```
                                    grayValues[(i - 1) * curBitmap.Width + j - 1] == 0 |
                                    grayValues[(i + 1) * curBitmap.Width + j - 1] == 0 |
                                    grayValues[(i - 1) * curBitmap.Width + j + 1] == 0 |
                                    grayValues[(i + 1) * curBitmap.Width + j + 1] == 0
                              {
                                    tempArray[i * curBitmap.Width + j] = 0;
                              }
                        }
                  }
                  break;
            case 0x28:
                  // 5位方形结构元素
                  for (int i = 2; i < curBitmap.Height - 2; i++)
                  {
                        for (int j = 2; j < curBitmap.Width - 2; j++)
                        {
                              if (grayValues[(i - 2) * curBitmap.Width + j - 2] == 0 |
                                    grayValues[(i - 2) * curBitmap.Width + j - 1] == 0 |
                                    grayValues[(i - 2) * curBitmap.Width + j] == 0 |
                                    grayValues[(i - 2) * curBitmap.Width + j + 1] == 0 |
                                    grayValues[(i - 2) * curBitmap.Width + j + 2] == 0 |
                                    grayValues[(i - 1) * curBitmap.Width + j - 2] == 0 |
                                    grayValues[(i - 1) * curBitmap.Width + j - 1] == 0 |
                                    grayValues[(i - 1) * curBitmap.Width + j] == 0 |
                                    grayValues[(i - 1) * curBitmap.Width + j + 1] == 0 |
                                    grayValues[(i - 1) * curBitmap.Width + j + 2] == 0 |
                                    grayValues[i * curBitmap.Width + j - 2] == 0 |
                                    grayValues[i * curBitmap.Width + j - 1] == 0 |
                                    grayValues[i * curBitmap.Width + j] == 0 ||
                                    grayValues[i * curBitmap.Width + j + 1] == 0 |
                                    grayValues[i * curBitmap.Width + j + 2] == 0 |
                                    grayValues[(i + 2) * curBitmap.Width + j - 2] == 0 |
                                    grayValues[(i + 2) * curBitmap.Width + j - 1] == 0 |
                                    grayValues[(i + 2) * curBitmap.Width + j] == 0 |
                                    grayValues[(i + 2) * curBitmap.Width + j + 1] == 0 |
                                    grayValues[(i + 2) * curBitmap.Width + j + 2] == 0 |
                                    grayValues[(i + 1) * curBitmap.Width + j - 2] == 0 |
                                    grayValues[(i + 1) * curBitmap.Width + j - 1] == 0 |
                                    grayValues[(i + 1) * curBitmap.Width + j] == 0 |
                                    grayValues[(i + 1) * curBitmap.Width + j + 1] == 0 |
                                    grayValues[(i + 1) * curBitmap.Width + j + 2] == 0
                              {
                                    tempArray[i * curBitmap.Width + j] = 0;
                              }
                        }
                  }
                  break;
            default:
                  MessageBox.Show("错误的结构元素! ");
                  break;
      }

      grayValues = (byte[])tempArray.Clone();

      System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
```

```
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（2）编译并运行该段程序。仍然以图 5.2 为例，打开该图像后，单击“图像膨胀”按钮，结构元素窗体被打开，如图 5.5 所示选择结构元素。

单击“确定”按钮，则经膨胀运算后的二值图像如图 5.6 所示。

图 5.5 膨胀运算结构元素

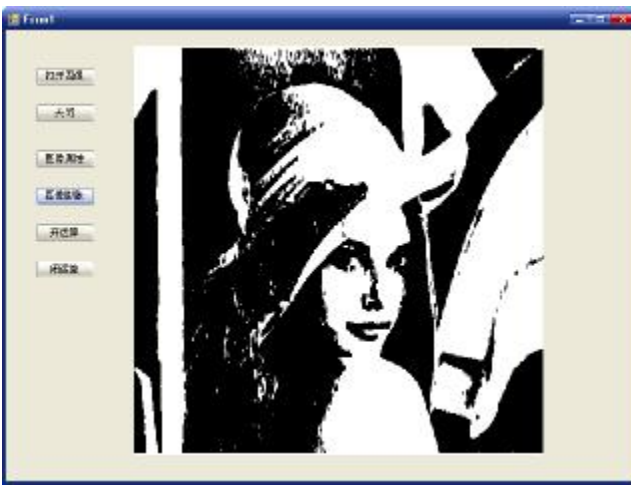


图 5.6 图像膨胀运算结果

5.3 图像开运算与闭运算

5.3.1 图像开运算与闭运算定义

二值图像开运算的数学表达式为：

$$g(x,y)=open\left[f(x,y),\mathrm{B}\right]=dilate\left\{erode\left[f(x,y),\mathrm{B}\right],\mathrm{B}\right\} \tag{5.4}$$

二值图像的开运算事实上就是先作腐蚀运算，再作膨胀运算。

二值图像闭运算的数学表达式为：

$$g(x,y)=close[f(x,y),\mathbf{B}]=erode\{dilate[f(x,y),\mathbf{B}],\mathbf{B}\} \tag{5.5}$$

二值图像的闭运算事实上就是先作膨胀运算，再作腐蚀运算。

5.3.2 图像开运算编程实例

该实例应用如图 5.1 所示的结构元素实现图像的开运算。

因为开运算就是先腐蚀，再膨胀，所以进行开运算的时候，只需单击“图像腐蚀”按钮，然后选择相同的结构元素再单击“图像膨胀”按钮即可。在这里，我们把以上两步合并成一步，用一个“开运算”按钮实现。

为“开运算”按钮控件添加 Click 事件，代码如下：

```
private void opening_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化struction
         struction struForm = new struction();
         // 设置从窗体标题
         struForm.Text = "开运算结构元素";

         if (struForm.ShowDialog() == DialogResult.OK)
         {
              Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
              System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                  System.Drawing.Imaging.ImageLockMode.ReadWrite,
                  curBitmap.PixelFormat);
              IntPtr ptr = bmpData.Scan0;
              int bytes = curBitmap.Width * curBitmap.Height;
              byte[] grayValues = new byte[bytes];
              System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

              // 得到结构元素
              byte flagStru = struForm.GetStruction;

              byte[] temp1Array = new byte[bytes];
              byte[] tempArray = new byte[bytes];
              for (int i = 0; i < bytes; i++)
              {
                   tempArray[i] = temp1Array[i] = 255;
              }

              switch (flagStru)
              {
                  case 0x11:
                      // 3 位水平方向结构元素
                      // 腐蚀运算
                      for (int i = 0; i < curBitmap.Height; i++)
                      {
                           for (int j = 1; j < curBitmap.Width - 1; j++)
                           {
                                if (grayValues[i * curBitmap.Width + j] == 0 &&
                                      grayValues[i * curBitmap.Width + j + 1] == 0 &&
                                      grayValues[i * curBitmap.Width + j - 1] == 0)
                                {
                                      temp1Array[i * curBitmap.Width + j] = 0;
                                }

                           }
                      }
                      // 膨胀运算
                      for (int i = 0; i < curBitmap.Height; i++)
                      {
                           for (int j = 1; j < curBitmap.Width - 1; j++)
                           {
```

```
                    if (temp1Array[i * curBitmap.Width + j] == 0 ||
                        temp1Array[i * curBitmap.Width + j + 1] == 0 ||
                        temp1Array[i * curBitmap.Width + j - 1] == 0)
                    {
                        tempArray[i * curBitmap.Width + j] = 0;
                    }

                }
            }
            break;
        case 0x21:
            // 5 位水平方向结构元素
            // 腐蚀运算
            for (int i = 0; i < curBitmap.Height; i++)
            {
                for (int j = 2; j < curBitmap.Width - 2; j++)
                {
                    if (grayValues[i * curBitmap.Width + j] == 0 &&
                        grayValues[i * curBitmap.Width + j + 1] == 0 &&
                        grayValues[i * curBitmap.Width + j - 1] == 0 &&
                        grayValues[i * curBitmap.Width + j + 2] == 0 &&
                        grayValues[i * curBitmap.Width + j - 2] == 0)
                    {
                        temp1Array[i * curBitmap.Width + j] = 0;
                    }

                }
            }
            // 膨胀运算
            for (int i = 0; i < curBitmap.Height; i++)
            {
                for (int j = 2; j < curBitmap.Width - 2; j++)
                {
                    if (temp1Array[i * curBitmap.Width + j] == 0 ||
                        temp1Array[i * curBitmap.Width + j + 1] == 0 ||
                        temp1Array[i * curBitmap.Width + j - 1] == 0 ||
                        temp1Array[i * curBitmap.Width + j + 2] == 0 ||
                        temp1Array[i * curBitmap.Width + j - 2] == 0)
                    {
                        tempArray[i * curBitmap.Width + j] = 0;
                    }

                }
            }
            break;
        case 0x12:
            // 3 位垂直方向结构元素
            // 腐蚀运算
            for (int i = 1; i < curBitmap.Height - 1; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    if (grayValues[i * curBitmap.Width + j] == 0 &&
                        grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                        grayValues[(i + 1) * curBitmap.Width + j] == 0)
```

```
                    {
                        temp1Array[i * curBitmap.Width + j] = 0;
                    }

                }
            }
            // 膨胀运算
            for (int i = 1; i < curBitmap.Height - 1; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    if (temp1Array[i * curBitmap.Width + j] == 0 ||
                        temp1Array[(i - 1) * curBitmap.Width + j] == 0 |
                        temp1Array[(i + 1) * curBitmap.Width + j] == 0)
                    {
                        tempArray[i * curBitmap.Width + j] = 0;
                    }

                }
            }
            break;
        case 0x22:
            // 5位垂直方向结构元素
            // 腐蚀运算
            for (int i = 2; i < curBitmap.Height - 2; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    if (grayValues[i * curBitmap.Width + j] == 0 &&
                        grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                        grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                        grayValues[(i - 2) * curBitmap.Width + j] == 0 &
                        grayValues[(i + 2) * curBitmap.Width + j] == 0)
                    {
                        temp1Array[i * curBitmap.Width + j] = 0;
                    }

                }
            }
            // 膨胀运算
            for (int i = 2; i < curBitmap.Height - 2; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    if (temp1Array[i * curBitmap.Width + j] == 0 ||
                        temp1Array[(i - 1) * curBitmap.Width + j] == 0 |
                        temp1Array[(i + 1) * curBitmap.Width + j] == 0 |
                        temp1Array[(i - 2) * curBitmap.Width + j] == 0 |
                        temp1Array[(i + 2) * curBitmap.Width + j] == 0)
                    {
                        tempArray[i * curBitmap.Width + j] = 0;
                    }

                }
            }
```

```
                    break;
                case 0x14:
                    // 3 位“十”字形状结构元素
                    // 腐蚀运算
                    for (int i = 1; i < curBitmap.Height - 1; i++)
                    {
                        for (int j = 1; j < curBitmap.Width - 1; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 &&
                                grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                                grayValues[i * curBitmap.Width + j + 1] == 0 &&
                                grayValues[i * curBitmap.Width + j - 1] == 0)
                            {
                                temp1Array[i * curBitmap.Width + j] = 0;
                            }

                        }
                    }
                    // 膨胀运算
                    for (int i = 1; i < curBitmap.Height - 1; i++)
                    {
                        for (int j = 1; j < curBitmap.Width - 1; j++)
                        {
                            if (temp1Array[i * curBitmap.Width + j] == 0 ||
                                temp1Array[(i - 1) * curBitmap.Width + j] == 0 |
                                temp1Array[(i + 1) * curBitmap.Width + j] == 0 |
                                temp1Array[i * curBitmap.Width + j + 1] == 0 ||
                                temp1Array[i * curBitmap.Width + j - 1] == 0)
                            {
                                tempArray[i * curBitmap.Width + j] = 0;
                            }

                        }
                    }
                    break;
                case 0x24:
                    // 5 位“十”字形状结构元素
                    // 腐蚀运算
                    for (int i = 2; i < curBitmap.Height - 2; i++)
                    {
                        for (int j = 2; j < curBitmap.Width - 2; j++)
                        {
                            if (grayValues[i * curBitmap.Width + j] == 0 &&
                                grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i - 2) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 2) * curBitmap.Width + j] == 0 &
                                grayValues[i * curBitmap.Width + j + 1] == 0 &&
                                grayValues[i * curBitmap.Width + j - 1] == 0 &&
                                grayValues[i * curBitmap.Width + j + 2] == 0 &&
                                grayValues[i * curBitmap.Width + j - 2] == 0)
                            {
                                temp1Array[i * curBitmap.Width + j] = 0;
                            }
```

```
                    }
                }
                // 膨胀运算
                for (int i = 2; i < curBitmap.Height - 2; i++)
                {
                    for (int j = 2; j < curBitmap.Width - 2; j++)
                    {
                        if (temp1Array[i * curBitmap.Width + j] == 0 ||
                            temp1Array[(i - 1) * curBitmap.Width + j] == 0 |
                            temp1Array[(i + 1) * curBitmap.Width + j] == 0 |
                            temp1Array[(i - 2) * curBitmap.Width + j] == 0 |
                            temp1Array[(i + 2) * curBitmap.Width + j] == 0 |
                            temp1Array[i * curBitmap.Width + j + 1] == 0 ||
                            temp1Array[i * curBitmap.Width + j - 1] == 0 ||
                            temp1Array[i * curBitmap.Width + j + 2] == 0 ||
                            temp1Array[i * curBitmap.Width + j - 2] == 0)
                        {
                            tempArray[i * curBitmap.Width + j] = 0;
                        }

                    }
                }
                break;
            case 0x18:
                // 3位方形结构元素
                // 腐蚀运算
                for (int i = 1; i < curBitmap.Height - 1; i++)
                {
                    for (int j = 1; j < curBitmap.Width - 1; j++)
                    {
                        if (grayValues[i * curBitmap.Width + j] == 0 &&
                            grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                            grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                            grayValues[i * curBitmap.Width + j + 1] == 0 &&
                            grayValues[i * curBitmap.Width + j - 1] == 0 &&
                            grayValues[(i - 1) * curBitmap.Width + j - 1] == 0 &
                            grayValues[(i + 1) * curBitmap.Width + j - 1] == 0 &
                            grayValues[(i - 1) * curBitmap.Width + j + 1] == 0 &
                            grayValues[(i + 1) * curBitmap.Width + j + 1] == 0
                        {
                            temp1Array[i * curBitmap.Width + j] = 0;
                        }

                    }
                }
                // 膨胀运算
                for (int i = 1; i < curBitmap.Height - 1; i++)
                {
                    for (int j = 1; j < curBitmap.Width - 1; j++)
                    {
                        if (temp1Array[i * curBitmap.Width + j] == 0 ||
                            temp1Array[(i - 1) * curBitmap.Width + j] == 0 |
                            temp1Array[(i + 1) * curBitmap.Width + j] == 0 |
                            temp1Array[i * curBitmap.Width + j + 1] == 0 ||
```

```
                                temp1Array[i * curBitmap.Width + j - 1] == 0 ||
                                temp1Array[(i - 1) * curBitmap.Width + j - 1] == 0 |
                                temp1Array[(i + 1) * curBitmap.Width + j - 1] == 0 |
                                temp1Array[(i - 1) * curBitmap.Width + j + 1] == 0 |
                                temp1Array[(i + 1) * curBitmap.Width + j + 1] == 0
                          {
                                tempArray[i * curBitmap.Width + j] = 0;
                          }

                    }
               }
               break;
          case 0x28:
               // 5 位方形结构元素
               // 腐蚀运算
               for (int i = 2; i < curBitmap.Height - 2; i++)
               {
                    for (int j = 2; j < curBitmap.Width - 2; j++)
                    {
                         if (grayValues[(i - 2) * curBitmap.Width + j - 2] == 0 &
                                grayValues[(i - 2) * curBitmap.Width + j - 1] == 0 &
                                grayValues[(i - 2) * curBitmap.Width + j] == 0 &
                                grayValues[(i - 2) * curBitmap.Width + j + 1] == 0 &
                                grayValues[(i - 2) * curBitmap.Width + j + 2] == 0 &
                                grayValues[(i - 1) * curBitmap.Width + j - 2] == 0 &
                                grayValues[(i - 1) * curBitmap.Width + j - 1] == 0 &
                                grayValues[(i - 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i - 1) * curBitmap.Width + j + 1] == 0 &
                                grayValues[(i - 1) * curBitmap.Width + j + 2] == 0 &
                                grayValues[i * curBitmap.Width + j - 2] == 0 &&
                                grayValues[i * curBitmap.Width + j - 1] == 0 &&
                                grayValues[i * curBitmap.Width + j] == 0 &&
                                grayValues[i * curBitmap.Width + j + 1] == 0 &&
                                grayValues[i * curBitmap.Width + j + 2] == 0 &&
                                grayValues[(i + 2) * curBitmap.Width + j - 2] == 0 &
                                grayValues[(i + 2) * curBitmap.Width + j - 1] == 0 &
                                grayValues[(i + 2) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 2) * curBitmap.Width + j + 1] == 0 &
                                grayValues[(i + 2) * curBitmap.Width + j + 2] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j - 2] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j - 1] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j + 1] == 0 &
                                grayValues[(i + 1) * curBitmap.Width + j + 2] == 0
                          {
                                temp1Array[i * curBitmap.Width + j] = 0;
                          }

                    }
               }
               // 膨胀运算
               for (int i = 2; i < curBitmap.Height - 2; i++)
               {
```

```
                            for (int j = 2; j < curBitmap.Width - 2; j++)
                            {
                                if (temp1Array[(i - 2) * curBitmap.Width + j - 2] == 0 |
                                    temp1Array[(i - 2) * curBitmap.Width + j - 1] == 0 |
                                    temp1Array[(i - 2) * curBitmap.Width + j] == 0 |
                                    temp1Array[(i - 2) * curBitmap.Width + j + 1] == 0 |
                                    temp1Array[(i - 2) * curBitmap.Width + j + 2] == 0 |
                                    temp1Array[(i - 1) * curBitmap.Width + j - 2] == 0 |
                                    temp1Array[(i - 1) * curBitmap.Width + j - 1] == 0 |
                                    temp1Array[(i - 1) * curBitmap.Width + j] == 0 |
                                    temp1Array[(i - 1) * curBitmap.Width + j + 1] == 0 |
                                    temp1Array[(i - 1) * curBitmap.Width + j + 2] == 0 |
                                    temp1Array[i * curBitmap.Width + j - 2] == 0 ||
                                    temp1Array[i * curBitmap.Width + j - 1] == 0 ||
                                    temp1Array[i * curBitmap.Width + j] == 0 ||
                                    temp1Array[i * curBitmap.Width + j + 1] == 0 ||
                                    temp1Array[i * curBitmap.Width + j + 2] == 0 ||
                                    temp1Array[(i + 2) * curBitmap.Width + j - 2] == 0 |
                                    temp1Array[(i + 2) * curBitmap.Width + j - 1] == 0 |
                                    temp1Array[(i + 2) * curBitmap.Width + j] == 0 |
                                    temp1Array[(i + 2) * curBitmap.Width + j + 1] == 0 |
                                    temp1Array[(i + 2) * curBitmap.Width + j + 2] == 0 |
                                    temp1Array[(i + 1) * curBitmap.Width + j - 2] == 0 |
                                    temp1Array[(i + 1) * curBitmap.Width + j - 1] == 0 |
                                    temp1Array[(i + 1) * curBitmap.Width + j] == 0 |
                                    temp1Array[(i + 1) * curBitmap.Width + j + 1] == 0 |
                                    temp1Array[(i + 1) * curBitmap.Width + j + 2] == 0
                                {
                                    tempArray[i * curBitmap.Width + j] = 0;
                                }

                            }
                        }
                        break;
                    default:
                        MessageBox.Show("错误的结构元素! ");
                        break;
                }


                grayValues = (byte[])tempArray.Clone();

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

编译并运行该段程序。首先打开二值图像，仍然以图 5.2 为例。然后单击“开运算”按钮，打开结构元素窗体，如图 5.7 所示选择结构元素。

最后单击“确定”按钮，则经开运算后的二值图像如图 5.8 所示。

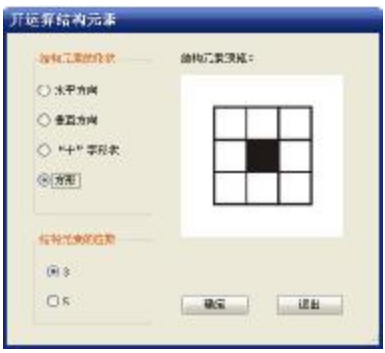


图 5.7 开运算结构元素

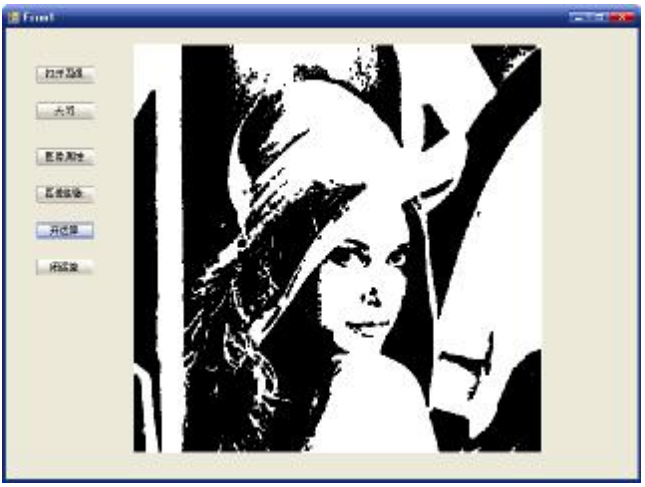


图 5.8 图像开运算结果

5.3.3 图像闭运算编程实例

该实例应用如图 5.1 所示的结构元素实现图像的闭运算。

闭运算是开运算的对偶运算，程序实现与开运算相似，只是先膨胀后腐蚀。在这里就不列出具体程序，只给出用图 5.7 所示的结构元素对图 5.2 的闭运算的结果，如图 5.9 所示。

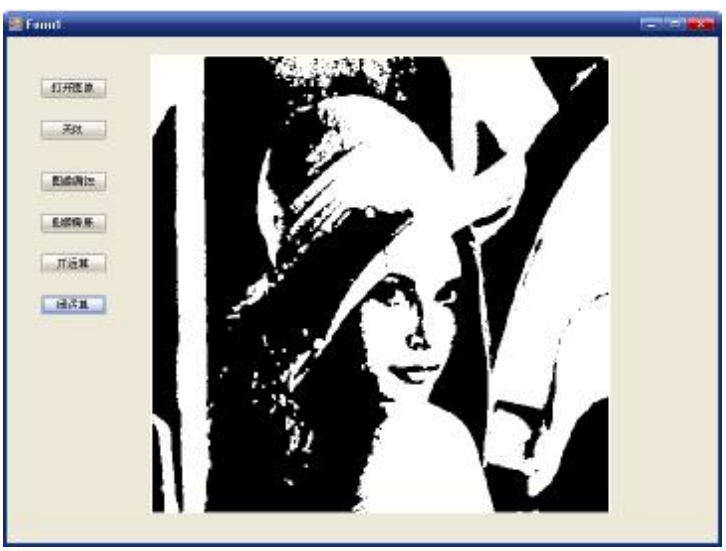


图 5.9 图像闭运算结果

5.4 击中击不中变换

5.4.1 击中击不中变换定义

击中击不中变换（HMT）需要两个结构元素 $\mathbf{B_1}$ 和 $\mathbf{B_2}$，合成一个结构元素对 $\mathbf{B}=(\mathbf{B_1},\mathbf{B_2})$。

一个用于探测图像内部，作为击中部分；另一个用于探测图像外部，作为击不中部分。显然 $\mathbf{B}_1$ 和 $\mathbf{B}_2$ 是不应该相连接的，即 $\mathbf{B}_1 \cap \mathbf{B}_2 = \phi$。击中击不中变换的数学表达式为：

$$g(x, y) = hitmiss[f(x, y), \mathbf{B}] = erode[f(x, y), \mathbf{B}_1] \text{ AND } erode[f^c(x, y), \mathbf{B}_2] \quad (5.6)$$

其中，$f^c(x, y)$表示的是 $f(x, y)$的补集。

5.4.2　击中击不中变换编程实例

该实例应用任意结构元素实现二值图像的击中击不中变换。

（1）在主窗体内添加一个 Button 控件，其属性修改如表 5.4 所示。

表 5.4　　所修改的属性

控　件	属　性	所修改内容
button1	Name	hitMiss
	Text	击中击不中
	Location	37，334

（2）创建 1 个名为 hitmiss 的 Windows 窗体，该窗体用于选择击中击不中变换的 2 个结构元素。在该窗体内添加 20 个 Button 控件和 2 个 GroupBox 控件，其中 9 个 Button 控件组成击中结构元素，另外 9 个 Button 控件组成击不中结构元素，它们分别在 2 个 GroupBox 控件内排列成 3 行 3 列的结构，每个 Button 控件表示一个像素。当单击某个 Button 控件时，表示它被选中作为结构元素中的一个像素，当再次单击它时，表示选中被取消。在这里，我们只用 3×3 的结构元素。其属性修改如表 5.5 所示。

表 5.5　　所修改的属性

控　件	属　性	所修改内容
hitmiss	Size	345，260
	ControlBox	False
button1	Name	start
	Location	55，176
	Text	确定
button2	Name	close
	Location	203，176
	Text	退出
groupBox1	Name	击中结构元素
	Location	25，25
	Size	120，120
groupBox2	Name	击不中结构元素
	Location	190，25
	Size	120，120
button3	Name	hit0
	Location	30，30
	Size	20，20

续表

控　　件	属　　性	所修改内容
button3	FlatStyle	Flat
	Text	
	BackColor	White
其他 8 个 button 类似 button3		
button12	Name	miss0
	Location	30，30
	Size	20，20
	FlatStyle	Flat
	Text	
	BackColor	White
另外 8 个 button 类似 button12		

分别为该窗体内的几个 Button 控件添加 Click 事件，在 9 个按钮组成的击中结构元素中单击某个按钮，则该按钮背景呈现黑色，表示击中结构元素包括该点像素，再次单击该按钮时，则该按钮背景又呈现白色，表示击中结构元素不再包括该点像素。另外 9 个按钮组成的击不中结构元素类似。代码如下：

```
private void start_Click(object sender, EventArgs e)
{
     // 判断两个结构元素是否有相交的部分
     if ((flagHit[0] == true && flagMiss[0] == true) ||
            (flagHit[1] == true && flagMiss[1] == true) ||
            (flagHit[2] == true && flagMiss[2] == true) ||
            (flagHit[3] == true && flagMiss[3] == true) ||
            (flagHit[4] == true && flagMiss[4] == true) ||
            (flagHit[5] == true && flagMiss[5] == true) ||
            (flagHit[6] == true && flagMiss[6] == true) ||
            (flagHit[7] == true && flagMiss[7] == true) ||
            (flagHit[8] == true && flagMiss[8] == true))
     {
            MessageBox.Show("击中与击不中结构元素不应相加! ");
     }
     else
     {
            this.DialogResult = DialogResult.OK;
     }
}

private void close_Click(object sender, EventArgs e)
{
     this.Close();
}

// 击中结构元素组中的第一个按钮控件的Click事件
private void hit0_Click(object sender, EventArgs e)
{
     if (flagHit[0] == false)
     {
```

```
            // 以前没有被选中
            // 设置相关变量标识
            flagHit[0] = true;
            // 按钮背景设置为黑色
            hit0.BackColor = Color.Black;
      }
      else
      {
            // 以前被选中
            // 清除相关变量标识
            flagHit[0] = false;
            // 按钮背景设置为白色
            hit0.BackColor = Color.White;
      }
}
…
// 其他hit按钮控件的Click事件类似
…
// 击不中结构元素组中的第一个按钮控件的Click事件
private void miss0_Click(object sender, EventArgs e)
{
      if (flagMiss[0] == false)
      {
            // 以前没有被选中
            // 设置相关变量标识
            flagMiss[0] = true;
            // 按钮背景设置为黑色
            miss0.BackColor = Color.Black;
      }
      else
      {
            // 以前被选中
            // 清除相关变量标识
            flagMiss[0] = false;
            // 按钮背景设置为白色
            miss0.BackColor = Color.White;
      }
}
…
// 其他miss按钮控件的click事件类似
…
```

其中 flagHit 和 flagMiss 都是 bool 型数组变量，它们在构造函数内被定义并初始化，代码如下：

```
flagHit = new bool[9];
flagMiss = new bool[9];
// 数组清零（置为false）
Array.Clear(flagHit, 0, 9);
Array.Clear(flagMiss, 0, 9);
```

为了和主窗体之间传递这两个数组变量，再添加两个 get 属性访问器，代码如下：

```
public bool[] GetHitStruction
{
    get
```

```
    {
        // 击中结构元素数组
        return flagHit;
    }
}

public bool[] GetMissStruction
{
    get
    {
        // 击不中结构元素数组
        return flagMiss;
    }
}
```

（3）回到主窗体，为“击中击不中”按钮控件添加 Click 事件，代码如下：

```
private void hitMiss_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化hitmiss
         hitmiss hitAndMiss = new hitmiss();

         if (hitAndMiss.ShowDialog() == DialogResult.OK)
         {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] grayValues = new byte[bytes];
                System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

                // 得到击中结构元素
                bool[] hitStru = hitAndMiss.GetHitStruction;
                // 得到击不中结构元素
                bool[] missStru = hitAndMiss.GetMissStruction;

                byte[] tempArray = new byte[bytes];
                byte[] temp1Array = new byte[bytes];
                byte[] temp2Array = new byte[bytes];
                for (int i = 0; i < bytes; i++)
                {
                     // 原图像的补集
                     tempArray[i] = (byte)(255 - grayValues[i]);
                     temp1Array[i] = 255;
                     temp2Array[i] = 255;
                }

                // 应用击中结构元素进行腐蚀运算
                for (int i = 1; i < curBitmap.Height - 1; i++)
                {
```

```
                    for (int j = 1; j < curBitmap.Width - 1; j++)
                    {
                        if ((grayValues[(i - 1) * curBitmap.Width + j - 1] == 0 ||
                                hitStru[0] == false) &&
                            (grayValues[(i - 1) * curBitmap.Width + j] == 0 ||
                                hitStru[1] == false) &&
                            (grayValues[(i - 1) * curBitmap.Width + j + 1] == 0 ||
                                hitStru[2] == false) &&
                            (grayValues[i * curBitmap.Width + j - 1] == 0 ||
                                hitStru[3] == false) &&
                            (grayValues[i * curBitmap.Width + j] == 0 || hitStru[4] =
alse) &&
                            (grayValues[i * curBitmap.Width + j + 1] == 0 ||
                                hitStru[5] == false) &&
                            (grayValues[(i + 1) * curBitmap.Width + j - 1] == 0 ||
                                hitStru[6] == false) &&
                            (grayValues[(i + 1) * curBitmap.Width + j] == 0 ||
                                hitStru[7] == false) &&
                            (grayValues[(i + 1) * curBitmap.Width + j + 1] == 0 ||
                                hitStru[8] == false))
                        {
                            temp1Array[i * curBitmap.Width + j] = 0;
                        }

                    }
                }

                // 应用击不中结构元素进行腐蚀运算
                for (int i = 1; i < curBitmap.Height - 1; i++)
                {
                    for (int j = 1; j < curBitmap.Width - 1; j++)
                    {
                        if ((tempArray[(i - 1) * curBitmap.Width + j - 1] == 0 ||
                                missStru[0] == false) &&
                            (tempArray[(i - 1) * curBitmap.Width + j] == 0 ||
                                missStru[1] == false) &&
                            (tempArray[(i - 1) * curBitmap.Width + j + 1] == 0 ||
                                missStru[2] == false) &&
                            (tempArray[i * curBitmap.Width + j - 1] == 0 ||
                                missStru[3] == false) &&
                            (tempArray[i * curBitmap.Width + j] == 0 ||
                                missStru[4] == false) &&
                            (tempArray[i * curBitmap.Width + j + 1] == 0 ||
                                missStru[5] == false) &&
                            (tempArray[(i + 1) * curBitmap.Width + j - 1] == 0 ||
                                missStru[6] == false) &&
                            (tempArray[(i + 1) * curBitmap.Width + j] == 0 ||
                                missStru[7] == false) &&
                            (tempArray[(i + 1) * curBitmap.Width + j + 1] == 0 ||
                                missStru[8] == false))
                        {
                            temp2Array[i * curBitmap.Width + j] = 0;
                        }
```

```
                    }
                }

            // 两个腐蚀运算的结果再进行“与”操作
            for (int i = 0; i < bytes; i++)
            {
                if(temp1Array[i] == 0 && temp2Array[i] == 0)
                {
                    tempArray[i] = 0;
                }
                else
                {
                    tempArray[i] = 255;
                }
            }

            grayValues = (byte[])tempArray.Clone();

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            curBitmap.UnlockBits(bmpData);
        }

        Invalidate();
    }
}
```

（4）编译并运行该段程序。打开图像，如图 5.10 所示。

单击“击中击不中”按钮，选择结构元素，如图 5.11 所示。

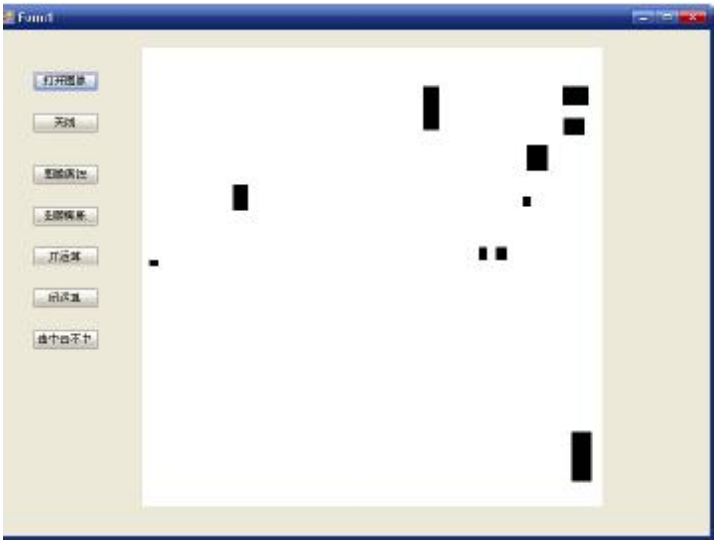


图 5.10 击中击不中原图

图 5.11 击中击不中结构元素

单击“确定”按钮，完成了击中击不中变换，结果如图 5.12 所示，它的每个像素点都是图 5.10 中对应矩形的左上角像素。

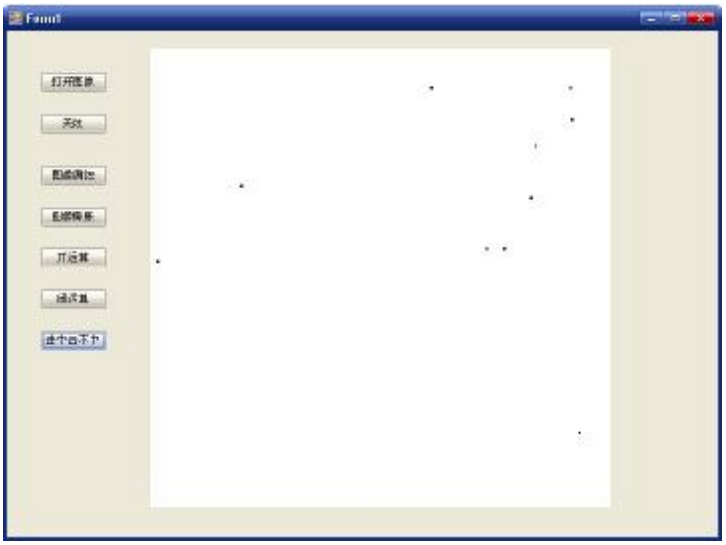


图 5.12　击中击不中变换结果（为便于观察，点已被人为放大）

5.5　小结

本章介绍了数学形态学的 5 种基本运算：腐蚀、膨胀、开运算、闭运算以及击中击不中变换的编程实例。数学形态学的关键是选取结构元素，图 5.1 列出了最重要的 8 种结构元素，其他更复杂的结构元素都可以用上述结构元素组合生成。因此本章开始几个实例是用 switch/case 语句对这常用的 8 种结构元素进行选取；而在击中击不中变换实例中，使用了选取任意形式的结构元素的方法。这两种对结构元素编程实现的方法可以在以后的编程中灵活选用。

第 6 章　频率变换

空间域和频率域是数字图像处理中两个不同的领域。空间域是指图像平面自身，它是以对图像的像素直接处理为基础的；频率域是以修改图像的傅里叶变换为基础的。前面介绍的几章都是在空间域内对图像进行的分析。但有些情况，在空间域内表述十分困难，而在频率域内却变得非常简单。后面章节中介绍的图像去噪、图像分隔、图像编码等许多技术都需要在频率域内进行。

在数字图像处理中，为解决某一问题，往往需要在空间域和频率域之间来回地切换。离散傅里叶变换正是连接这两个不同领域的纽带。本章主要介绍二维离散傅里叶变换，以及基于频域的成分滤波和方位滤波。

6.1　二维离散傅里叶变换

令 $f(x, y)$表示一幅大小为 $M\times N$ 的图像，其中 x=0,1,2,…, $M-1$ 和 y=0,1,2,…, $N-1$。$f(x, y)$ 的二维离散傅里叶变换可表示为 $F(u, v)$，其表达式为：

$$F(u,v)=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y)e^{-j2\pi(ux/M+vy/N)} \tag{6.1}$$

其中 u=0,1,2,…, $M-1$ 和 v=0,1,2,…, $N-1$。离散傅里叶逆变换为：

$$f(x,y)=\frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1}F(u,v)e^{j2\pi(ux/M+vy/N)} \tag{6.2}$$

当以上两式成立时，$f(x, y)$和 $F(u, v)$形成了一个离散傅里叶变换（DFT）对。

离散傅里叶变换的计算量是巨大的。对一幅 $M\times N$ 的图像，它需要 M^2N^2 次复数乘法和加法。例如，M=N=512，则需要 $2^{36}=6.9\times10^{10}$ 次复数乘法/加法。然而值得庆幸的是，离散傅里叶变换的快速算法——快速傅里叶变换（FFT）在许多年以前就已经被提出，并广泛应用到各个领域。它是利用复指数（$W_N=e^{-j2\pi/N}$）的对称性和周期性，通过蝶形结来简化运算量的，它的计算复杂度不超过 $MN\times\log_2(MN)$。

下面介绍一些重要的二维离散傅里叶变换的性质：

（1）可分离性。

$$F(u,v)=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y)e^{-j2\pi(ux/M+vy/N)}=\sum_{x=0}^{M-1}e^{-j2\pi ux/M}\sum_{y=0}^{N-1}f(x,y)e^{-j2\pi vy/N}$$
$$=\sum_{x=0}^{M-1}F(x,v)e^{-j2\pi ux/M} \tag{6.3}$$

其中，$F(x,v)=\sum_{y=0}^{N-1}f(x,y)e^{-j2p vy/N}$，是一维离散傅里叶变换。

可分离性的主要优点是可以由两次一维傅里叶变换来实现二维傅里叶变换。对图像来说，就是先对每一行取变换，再对每一列取变换，或者相反。

（2）可位移性。

由 $f(x, y)$乘以指数（$e^{j2p(u_0x/M+v_0y/N)}$），并取其乘积的傅里叶变换，可以使频率平面的原点位移至(u_0, v_0)。傅里叶逆变换也可以进行原点位移。

当 $u_0= v_0= M/2 = N/2$ 时，用$(-1)^{(x+y)}$ 乘以 $f(x, y)$就可以将 $f(x, y)$的傅里叶变换原点移动到$M\times N$ 频率方阵中心。

之所以要进行坐标位移，是因为这样可以将图像的频谱原点移动到图像的中心，也就是把图像中的能量集中到图像中心，这样做不仅符合人们观察图像的习惯，而且也有利于在频域内对图像进行处理。一般来说，图像中的能量都集中在低频区域。

（3）周期性。

傅里叶变换具有周期性，即：

$$F(u, v)=F(u+aM, v)=F(u, v+bN)=F(u+aM, v+bN) \tag{6.4}$$
$$f(x, y)=f(x+aM, y+bN) \tag{6.5}$$

其中 M 和 N 分别为图像的高和宽，$a,b=0,\pm1,\pm2,\ldots$。

6.2 快速傅里叶变换

6.2.1 快速傅里叶变换概述

在进行快速傅里叶变换时，序列的长度 N 一般都是以 2 为底的整数次幂，即基二算法，只有这样它的效率才比较高，这也是本书所选取图像的大小都是 512×512 的原因。

基二快速傅里叶变换可以分为按时间抽取法和按频率抽取法这两种方法。在这里，我们选用按频率抽取法进行运算，它要求输入序列的顺序是自然顺序，而经过运算后的输出则是以按位反转的规律重排的。因此要想得到正常的自然顺序，还要对输出序列进行重新排序。

用于图像处理的二维离散傅里叶变换是建立在一维离散傅里叶变换基础之上的，即一个二维离散傅里叶变换可以用二次一维离散傅里叶变换来实现。因此可以先对图像中的行序列应用快速傅里叶变换，然后对运算结果再进行列序列的快速傅里叶变换。

离散傅里叶逆变换也可以应用快速傅里叶变换来实现。对一维序列来说，先对序列取共轭变换，然后用快速傅里叶变换计算，最后再对运算结果取一次共轭变换并除以该序列的长度。对二维图像信号来说，如正变换一样，先对行序列处理，再对列序列处理即可。

6.2.2 快速傅里叶变换编程实例

该实例完成了二维傅里叶变换和逆变换的方法，为后面的成分滤波和相位滤波做了准备。

（1）首先创建一个“模板主窗体”。

（2）因为是在频域内，就涉及复数的计算，而 C#中没有将复数设计为一种内部数据类所以要先设计并实现处理复数运算的 C#类 Complex。又因为在这里，没有用到复数的所有功能，因此为了简化程序，只对用到的运算进行定义。

选择主菜单“项目 | 添加类…”，打开添加新项窗口，在“名称”内输入 Complex.cs，然后单击“添加”按钮完成 Complex 类的添加。Complex 类的代码如下：

```
class Complex
{
    // 复数的实部
    private double real = 0.0;
    // 复数的虚部
    private double imaginary = 0.0;

    // 实部的属性
    public double Real
    {
        get
        {
            return real;
        }
        set
        {
            real = value;
        }
    }

    // 虚部的属性
    public double Imaginary
    {
        get
        {
            return imaginary;
        }
        set
        {
            imaginary = value;
        }
    }

    // 基本构造函数
    public Complex()
    {
    }

    // 指定值的构造函数
    public Complex(double dbreal, double dbimag)
    {
        real = dbreal;
```

```
        imaginary = dbimag;
    }

    // 复制构造函数
    public Complex(Complex other)
    {
        real = other.real;
        imaginary = other.imaginary;
    }

    // 重载 + 运算符
    public static Complex operator +(Complex comp1, Complex comp2)
    {
        return comp1.Add(comp2);
    }

    // 重载 - 运算符
    public static Complex operator -(Complex comp1, Complex comp2)
    {
        return comp1.Subtract(comp2);
    }

    // 重载 * 运算符
    public static Complex operator *(Complex comp1, Complex comp2)
    {
        return comp1.Multiply(comp2);
    }

    // 实现复数加法
    public Complex Add(Complex comp)
    {
        double x = real + comp.real;
        double y = imaginary + comp.imaginary;

        return new Complex(x, y);
    }

    // 实现复数减法
    public Complex Subtract(Complex comp)
    {
        double x = real - comp.real;
        double y = imaginary - comp.imaginary;

        return new Complex(x, y);
    }

    // 实现复数乘法
    public Complex Multiply(Complex comp)
    {
        double x = real * comp.real - imaginary * comp.imaginary;
        double y = real * comp.imaginary + imaginary * comp.real;

        return new Complex(x, y);
    }
```

```
        // 求幅度
        public double Abs()
        {
            // 取得实部的绝对值
            double x = Math.Abs(real);
            // 取得虚部的绝对值
            double y = Math.Abs(imaginary);

            // 实部为0
            if (real == 0)
            {
                return y;
            }
            // 虚部为0
            if (imaginary == 0)
            {
                return x;
            }

            // 计算模
            if (x > y)
            {
                return (x * Math.Sqrt(1 + (y / x) * (y / x)));
            }
            else
            {
                return (y * Math.Sqrt(1 + (x / y) * (x / y)));
            }
        }

        // 求相位角
        public double Angle()
        {
            // 实部和虚部都为0
            if (real == 0 && imaginary == 0)
                return 0;

            if (real == 0)
            {
                // 实部为0
                if (imaginary > 0)
                    return Math.PI / 2;
                else
                    return -Math.PI / 2;
            }
            else
            {
                if (real > 0)
                {
                    // 实部大于0
                    return Math.Atan2(imaginary, real);
                }
                else
                {
                    // 实部小于0
                    if (imaginary >= 0)
```

```
                    return Math.Atan2(imaginary, real) + Math.PI;
                else
                    return Math.Atan2(imaginary, real) - Math.PI;
            }
        }
    }

    // 共轭复数
    public Complex Conjugate()
    {
        return new Complex(this.real, -this.imaginary);
    }
}
```

（3）回到主窗体代码区内，编写快速傅里叶变换程序，代码如下：

```
/*******************************************
快速傅里叶变换
sourceData: 待变换的序列
countN: 序列长度
返回变换后的序列
*******************************************/
private Complex[] fft(Complex[] sourceData, int countN)
{
    // fft 的级数
    int r = Convert.ToInt32(Math.Log(countN, 2));

    Complex[] w = new Complex[countN / 2];
    Complex[] interVar1 = new Complex[countN];
    Complex[] interVar2 = new Complex[countN];

    interVar1 = (Complex[])sourceData.Clone();

    // 求加权系数w
    for (int i = 0; i < countN / 2; i++)
    {
        double angle = -i * Math.PI * 2 / countN;
        w[i] = new Complex(Math.Cos(angle), Math.Sin(angle));
    }

    // 蝶形运算
    for (int i = 0; i < r; i++)
    {
        int interval = 1 << i;
        int halfN = 1 << (r - i);
        // 对每级的每一组点循环
        for (int j = 0; j < interval; j++)
        {
            int gap = j * halfN;
            // 对每组的每一点循环
            for (int k = 0; k < halfN / 2; k++)
            {
                // 进行蝶形算法
                interVar2[k + gap] = interVar1[k + gap] + interVar1[k + gap + halfN / 2]
                interVar2[k + halfN / 2 + gap] = (interVar1[k + gap] -
```

```
                        interVar1[k + gap + halfN / 2]) * w[k * interval];
                }
            }
            interVar1 = (Complex[])interVar2.Clone();
        }

        // 按位取反
        for (uint j = 0; j < countN; j++)
        {
            uint rev = 0;
            uint num = j;
            // 重新排序
            for (int i = 0; i < r; i++)
            {
                rev <<= 1;
                rev |= num & 1;
                num >>= 1;
            }
            interVar2[rev] = interVar1[j];
        }
        return interVar2;
    }

    /***********************************
    快速傅里叶逆变换
    sourceData：待变换的序列
    countN：序列长度
    返回变换后的序列
    ***********************************/
    private Complex[] ifft(Complex[] sourceData, int countN)
    {
        // 共轭变换
        for (int i = 0; i < countN; i++)
        {
            sourceData[i] = sourceData[i].Conjugate();
        }

        Complex[] interVar = new Complex[countN];
        // 调用快速傅里叶变换
        interVar = fft(sourceData, countN);

        // 共轭变换，并除以长度
        for (int i = 0; i < countN; i++)
        {
            interVar[i] = new Complex(interVar[i].Real / countN, -interVar[i].Imaginary / countN);
        }

        return interVar;
    }
```

在一维快速傅里叶变换的基础上，继续编写用于图像处理的二维快速傅里叶变换程序，代码如下：

```
    /*****************************************************
    用于图像处理的二维快速傅里叶变换
```

```
    imageData: 图像序列
    imageWidth: 图像的宽度
    imageHeight: 图像的长度
    inv: 标识是否进行坐标位移变换
        true: 进行坐标位移变换
        false: 不进行坐标位移变换
    返回变换后的频域数据
    ***************************************************/
    private Complex[] fft2(byte[] imageData, int imageWidth, int imageHeight, bool inv)
    {
        int bytes = imageWidth * imageHeight;
        byte[] bmpValues = new byte[bytes];
        Complex[] tempCom1 = new Complex[bytes];

        bmpValues = (byte[])imageData.Clone();

        // 赋值: 把实数变为复数, 即虚部为0
        for (int i = 0; i < bytes; i++)
        {
            if (inv == true)
            {
                // 进行频域坐标位移

                if ((i / imageWidth + i % imageWidth) % 2 == 0)
                {
                    tempCom1[i] = new Complex(bmpValues[i], 0);
                }
                else
                {
                    tempCom1[i] = new Complex(-bmpValues[i], 0);
                }
            }
            else
            {
                // 不进行频域坐标位移
                tempCom1[i] = new Complex(bmpValues[i], 0);
            }
        }

        // 水平方向快速傅里叶变换
        Complex[] tempCom2 = new Complex[imageWidth];
        Complex[] tempCom3 = new Complex[imageWidth];
        for (int i = 0; i < imageHeight; i++)
        {
            // 得到水平方向复数序列
            for (int j = 0; j < imageWidth; j++)
            {
                tempCom2[j] = tempCom1[i * imageWidth + j];
            }

            // 调用一维傅里叶变换
            tempCom3 = fft(tempCom2, imageWidth);

            // 把结果赋值回去
            for (int j = 0; j < imageWidth; j++)
```

```
            {
                tempCom1[i * imageWidth + j] = tempCom3[j];
            }
        }

        // 垂直方向快速傅里叶变换
        Complex[] tempCom4 = new Complex[imageHeight];
        Complex[] tempCom5 = new Complex[imageHeight];
        for (int i = 0; i < imageWidth; i++)
        {
            // 得到垂直方向复数序列
            for (int j = 0; j < imageHeight; j++)
            {
                tempCom4[j] = tempCom1[j * imageWidth + i];
            }

            // 调用一维傅里叶变换
            tempCom5 = fft(tempCom4, imageHeight);

            // 把结果赋值回去
            for (int j = 0; j < imageHeight; j++)
            {
                tempCom1[j * imageHeight + i] = tempCom5[j];
            }
        }

        return tempCom1;
    }

    /*************************************************
    用于图像处理的二维快速傅里叶逆变换
    freData: 频域数据
    imageWidth: 图像的宽度
    imageHeight: 图像的长度
    inv: 标识是否进行坐标位移变换，要与二维快速傅里叶正变换一致
        true: 进行坐标位移变换
        false: 不进行坐标位移变换
    返回变换后的空间域数据（即图像数据）
    *************************************************/
    private byte[] ifft2(Complex[] freData, int imageWidth, int imageHeight, bool inv)
    {
        int bytes = imageWidth * imageHeight;
        byte[] bmpValues = new byte[bytes];
        Complex[] tempCom1 = new Complex[bytes];

        tempCom1 = (Complex[])freData.Clone();

        // 水平方向快速傅里叶逆变换
        Complex[] tempCom2 = new Complex[imageWidth];
        Complex[] tempCom3 = new Complex[imageWidth];
        for (int i = 0; i < imageHeight; i++)
        {
            // 得到水平方向复数序列
            for (int j = 0; j < imageWidth; j++)
            {
```

```
            tempCom2[j] = tempCom1[i * imageWidth + j];
        }

        // 调用一维傅里叶变换
        tempCom3 = ifft(tempCom2, imageWidth);

        // 把结果赋值回去
        for (int j = 0; j < imageWidth; j++)
        {
            tempCom1[i * imageWidth + j] = tempCom3[j];
        }
    }

    // 垂直方向快速傅里叶逆变换
    Complex[] tempCom4 = new Complex[imageHeight];
    Complex[] tempCom5 = new Complex[imageHeight];
    for (int i = 0; i < imageWidth; i++)
    {
        // 得到垂直方向复数序列
        for (int j = 0; j < imageHeight; j++)
        {
            tempCom4[j] = tempCom1[j * imageWidth + i];
        }

        // 调用一维傅里叶变换
        tempCom5 = ifft(tempCom4, imageHeight);

        // 把结果赋值回去
        for (int j = 0; j < imageHeight; j++)
        {
            tempCom1[j * imageHeight + i] = tempCom5[j];
        }
    }

    // 赋值：把复数转换为实数，只保留复数的实数部分
    double tempDouble;
    for (int i = 0; i < bytes; i++)
    {
        if (inv == true)
        {
            // 进行坐标位移

            if ((i / curBitmap.Width + i % curBitmap.Width) % 2 == 0)
            {
                tempDouble = tempCom1[i].Real;
            }
            else
            {
                tempDouble = -tempCom1[i].Real;
            }
        }
        else
        {
            // 不进行坐标位移
            tempDouble = tempCom1[i].Real;
```

```
            }

            if (tempDouble > 255)
            {
                bmpValues[i] = 255;
            }
            else
            {
                if (tempDouble < 0)
                {
                    bmpValues[i] = 0;
                }
                else
                {
                    bmpValues[i] = Convert.ToByte(tempDouble);
                }
            }
        }

        return bmpValues;
    }
```

需要说明的是，在图像处理中，对傅里叶变换中的系数并不是很关心，如公式（6.2）中的系数 1/*MN*，因为傅里叶变换的系数就是图像的灰度级，而它的灰度级已经被限制在一定的范围内，完全可以通过灰度级拉伸的办法来增加或减少灰度级，也就是傅里叶变换的系数

6.3　幅度图像和相位图像

一幅空间域图像经过离散傅里叶变换后，会生成频域图像，每个实数像素点会变换为复数，而复数是无法用一幅图像来描述的。然而，我们可以分别在一幅灰度图像中展示它的幅度或者相位，这样频域图像就分别用幅度图像和相位图像来描述出来了。

本实例实现了幅度图像和相位图像的绘制。

（1）在主窗体内添加 2 个 Button 控件，其属性修改如表 6.1 所示。

表 6.1　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	amplitude
	Text	幅度图像
	Location	37，150
button2	Name	phase
	Text	相位图像
	Location	37，196

（2）分别为这 2 个 Button 控件添加 Click 事件，代码如下：

```
private void amplitude_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
```

```
    {
        Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
        System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
            System.Drawing.Imaging.ImageLockMode.ReadWrite,
            curBitmap.PixelFormat);
        IntPtr ptr = bmpData.Scan0;
        int bytes = curBitmap.Width * curBitmap.Height;
        byte[] grayValues = new byte[bytes];
        System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

        Complex[] freDom = new Complex[bytes];
        double[] tempArray = new double[bytes];

        // 调用二维傅里叶变换，需要进行坐标位移
        freDom = fft2(grayValues, curBitmap.Width, curBitmap.Height, true);

        // 变量变换，并取幅度系数
        for (int i = 0; i < bytes; i++)
        {
            tempArray[i] = Math.Log(1 + freDom[i].Abs(), 2);
        }

        // 灰度级拉伸
        double a = 1000.0, b = 0.0;
        double p;
        // 找到最大值和最小值
        for (int i = 0; i < bytes; i++)
        {
            if (a > tempArray[i])
            {
                a = tempArray[i];
            }
            if (b < tempArray[i])
            {
                b = tempArray[i];
            }
        }
        // 得到比例系数
        p = 255.0 / (b - a);
        for (int i = 0; i < bytes; i++)
        {
            grayValues[i] = (byte)(p * (tempArray[i] - a) + 0.5);
        }

        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
        curBitmap.UnlockBits(bmpData);

        Invalidate();
    }
}

private void phase_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
```

```
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            Complex[] freDom = new Complex[bytes];
            double[] tempArray = new double[bytes];

            // 调用二维傅里叶变换，不进行坐标位移
            freDom = fft2(grayValues, curBitmap.Width, curBitmap.Height, false);

            // 取相位系数，并进行了变量变换
            for (int i = 0; i < bytes; i++)
            {
                tempArray[i] = freDom[i].Angle() + 2 * Math.PI;
            }

            // 灰度级拉伸
            double a = 1000.0, b = 0.0;
            double p;
            // 找到最大值和最小值
            for (int i = 0; i < bytes; i++)
            {
                if (a > tempArray[i])
                {
                    a = tempArray[i];
                }
                if (b < tempArray[i])
                {
                    b = tempArray[i];
                }
            }
            // 得到比例系数
            p = 255.0 / (b - a);
            for (int i = 0; i < bytes; i++)
            {
                grayValues[i] = (byte)(p * (tempArray[i] - a) + 0.5);
            }

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            curBitmap.UnlockBits(bmpData);

            Invalidate();
        }
    }
```

需要说明的是，这两段程序在最后赋值的时候都需要进行一次变量变换，这是为了把数据限制在一个较小的范围内，以便可以用灰度图像描述。而且为了使两幅图像看得更清晰，

还用到了第 3 章介绍的灰度拉伸方法。

在得到幅度图像时，进行了坐标位移，而在得到相位图像时，无需坐标位移。

（3）编译并运行该段程序。打开图像，如图 6.1 所示，再分别单击“幅度图像”和“相位图像”按钮后，得到图 6.2 和图 6.3。

比较两幅图像，用户可能会认为相位图像没有揭露任何有用的信息，完全是一些杂乱无序的图像，没有幅度图像那么吸引人。而且任何图像的相位图像都相差不多，仅从相位图像无法区分开来。但是事实上，幅度和相位是同等重要的，它所包括的意义会随着对图像分析的进一步理解，呈现出来的。

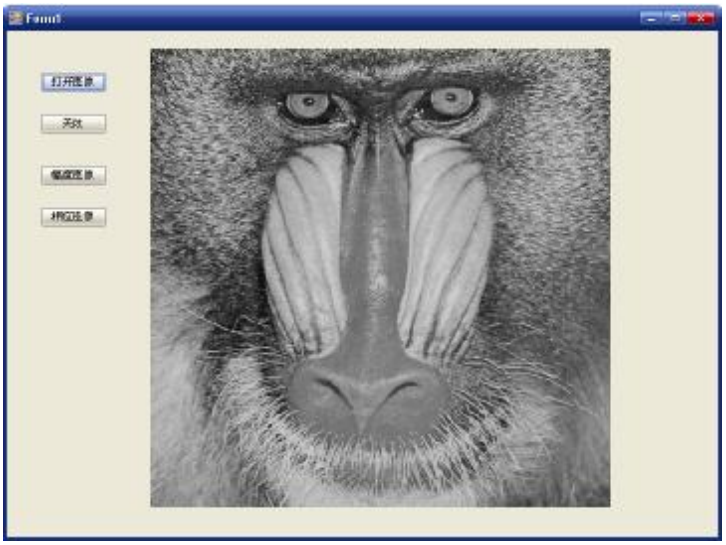


图 6.1 频域变换原图

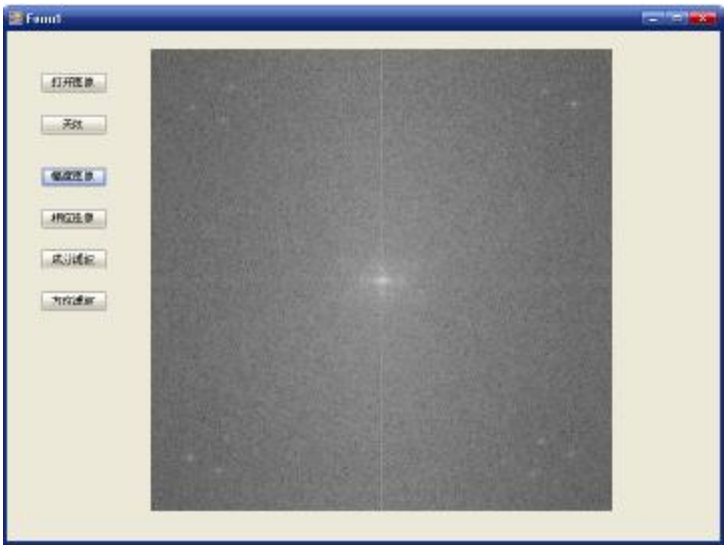


图 6.2 幅度图像（经过坐标位移和灰度级拉伸处理）

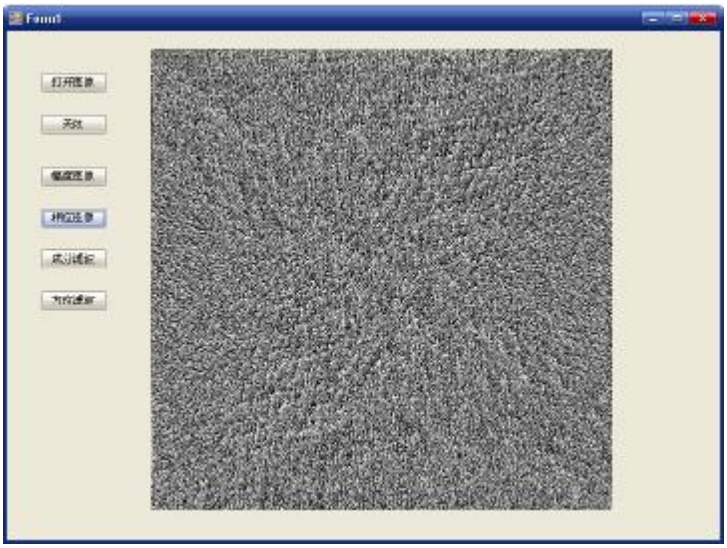


图 6.3　相位图像（经过灰度级拉伸处理）

幅度图像揭示了图像频率的分布和均值，因此当用某一种方法对图像频率进行调整时，可以对幅度图像进行分析处理。如图 6.2 所示，经过坐标位移（即把频谱原点移到图像中心）后的幅度图像，中部较亮的区域表现出来的是低频成分，四周较暗的区域表现出来的是高频成分

6.4　频率成分滤波

6.4.1　频率成分滤波原理

基于 6.3 节对幅度图像的分析，我们可以通过对一幅经过离散傅里叶变换后的图像的掩码方式分别进行高通滤波、低通滤波、带阻滤波和带通滤波，然后再经过离散傅里叶逆变换从而分别得到图像的高频分量、低频分量或中频分量。

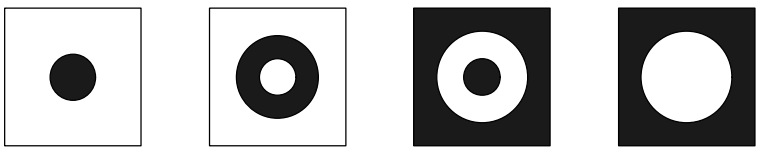


图 6.4　掩码图

图 6.4 为图像频率掩码图，图（a）、（b）、（c）和（d）分别为高通掩码、带阻掩码、带通掩码和低通掩码。图中黑色像素表示“0”，白色像素表示“1”，通过改变掩码圆半径的大小来改变频率成分滤波的效果。在实际滤波过程中，只要将频域图像与同等大小的掩码图相乘即可。我们知道，频率域相乘相当于空间域内进行线性卷积。而进行线性卷积要比单纯的乘法复杂得多。这也是为什么我们要把空间域图像转变为频率域图像的原因之一。

需要说明的是，我们要处理的频率域图像是经过坐标原点位移到图像中心后的频域图像。

6.4.2 频率成分滤波编程实例

该实例通过图 6.4 所示的掩码处理，可以得到不同频率成分的图像

（1）在主窗体内添加一个 Button 控件，其属性修改如表 6.2 所示。

表 6.2 所修改的属性

控 件	属 性	修 改 内 容
button1	Name	freGran
	Location	37，242
	Text	成分滤波

（2）创建 1 个名为 granularity 的 Windows 窗体，该窗体用于选择频率掩码参数，并绘制相应的掩码图。在该窗体内添加 2 个 Button 控件、3 个 Label 控件、4 个 RadioButton 控件和 1 个 NumericUpDown，其属性修改如表 6.3 所示。

表 6.3 所修改的属性

控 件	属 性	所修改内容
granularity	Size	460，430
	Controlbox	False
	Text	频率成分滤波
button1	Name	start
	Text	确定
	Location	80，340
button2	Name	close
	Text	退出
	Location	290，340
radioButton1	Name	low
	Text	低通滤波
	Location	49，59
	Checked	True
radioButton2	Name	midStop
	Text	带阻滤波
	Location	49，106
radioButton3	Name	mid
	Text	带通滤波
	Location	49，153
radioButton4	Name	high
	Text	高通滤波
	Location	49，200
label1	Text	掩码图预览
	Location	179，20

续表

控　件	属　性	所修改内容
label2	Location	47，256
	Text	低通掩码半径（%）
label3	Location	47，295
	Visible	False
	Text	
numericUpDown1	Name	radius1
	Location	196，254
	Size	58，21
	Value	10
numericUpDown2	Name	radius2
	Location	196，293
	Size	58，21
	Visible	False

分别为该窗体内的 2 个 Button 控件添加 Click 事件，为 4 个 radioButton 控件添加 CheckedChanged 事件以及为 2 个 numericUpDown 控件添加 ValueChanged 事件，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void low_CheckedChanged(object sender, EventArgs e)
{
    // 标识变量
    tempFlag = 0;
    // 设置label2控件
    label2.Text = "低通掩码半径（%）：";
    // 设置radius1控件
    radius1.Value = radius[0];
    // 使label3控件和radius2控件无效
    label3.Visible = false;
    radius2.Visible = false;
    // 更新窗体
    Invalidate();
}

private void midStop_CheckedChanged(object sender, EventArgs e)
{
    // 标识变量
    tempFlag = 1;
    // 设置label2和label3控件
```

```
        label2.Text = "带阻掩码外圆半径（%）：";
        label3.Visible = true;
        label3.Text = "带阻掩码内圆半径（%）：";
        // 设置radius1控件和radius2控件
        radius1.Value = radius[1];
        radius2.Visible = true;
        radius2.Value = radius[2];
        // 更新窗体
        Invalidate();
    }

    private void mid_CheckedChanged(object sender, EventArgs e)
    {
        // 标识变量
        tempFlag = 2;
        // 设置label2控件和label3控件
        label2.Text = "带通掩码外圆半径（%）：";
        label3.Visible = true;
        label3.Text = "带通掩码内圆半径（%）：";
        // 设置radius1控件和radius2控件
        radius1.Value = radius[3];
        radius2.Visible = true;
        radius2.Value = radius[4];
        // 更新窗体
        Invalidate();
    }

    private void high_CheckedChanged(object sender, EventArgs e)
    {
        // 标识变量
        tempFlag = 3;
        // 设置label2控件
        label2.Text = "高通掩码半径（%）：";
        // 设置radius1控件
        radius1.Value = radius[5];
        // 使label3控件和radius2控件无效
        label3.Visible = false;
        radius2.Visible = false;
        // 更新窗体
        Invalidate();
    }

    private void radius1_ValueChanged(object sender, EventArgs e)
    {
        byte tempValue = (byte)radius1.Value;
        switch (tempFlag)
        {
            case 0:
                // 低通滤波
                radius[0] = tempValue;
                break;
            case 1:
                // 带阻滤波
                //判断外圆半径是否大于内圆半径
```

```
            if (tempValue <= radius[2])
            {
                radius1.Value = radius[1];
                break;
            }
            radius[1] = tempValue;
            break;
        case 2:
            // 带通滤波
            //判断外圆半径是否大于内圆半径
            if (tempValue <= radius[4])
            {
                radius1.Value = radius[3];
                break;
            }
            radius[3] = tempValue;
            break;
        case 3:
            // 高通滤波
            radius[5] = tempValue;
            break;
        default:
            break;
    }
    Invalidate();
}

private void radius2_ValueChanged(object sender, EventArgs e)
{
    byte tempValue = (byte)radius2.Value;
    switch (tempFlag)
    {
        case 1:
            // 带阻滤波
            // 判断内圆半径是否小于外圆半径
            if (tempValue >= radius[1])
            {
                radius2.Value = radius[2];
                break;
            }
            radius[2] = tempValue;
            break;
        case 2:
            // 带通滤波
            // 判断内圆半径是否小于外圆半径
            if (tempValue >= radius[3])
            {
                radius2.Value = radius[4];
                break;
            }
            radius[4] = tempValue;
            break;
        default:
            break;
```

```
        }
        Invalidate();
    }
```

其中 tempFlag 为 byte 型变量，用于标识是何种滤波形式，radius 为 byte 型数组变量，用于记录滤波半径长度，radius[0]表示低通滤波掩码半径，radius[1]表示带阻掩码外圆半径，radius[2]表示其内圆半径，radius[3]表示带通掩码外圆半径，radius[4]表示其内圆半径，radius[5]表示高通滤波掩码半径，它们的单位都是%。tempFlag 和 radius 在构造函数内被初始化，代码如下：

```
tempFlag = 0;
radius = new byte[6] { 10, 50, 15, 60, 18, 20 };
```

为了更直接地看到掩码图形，还需要在这个窗体内绘出掩码图形，因此要为该窗体添加 Paint 事件，代码如下：

```
private void granularity_Paint(object sender, PaintEventArgs e)
{
    // 实例化白色和黑色画笔
    SolidBrush blackBrush = new SolidBrush(Color.Black);
    SolidBrush whiteBrush = new SolidBrush(Color.White);
    // 实例化黑色钢笔
    Pen blackPen=new Pen(Color.Black,1.5F);
    // 实例化Graphics
    Graphics g = e.Graphics;

    switch(tempFlag)
    {
        case 0:
            // 低通掩码
            g.FillRectangle(blackBrush, 180, 40, 200, 200);
            g.FillEllipse(whiteBrush, 280 - radius[0], 140 - radius[0], 2 * radius[0],
                2 * radius[0]);
            break;
        case 1:
            // 带阻掩码
            g.FillRectangle(whiteBrush, 180, 40, 200, 200);
            g.FillEllipse(blackBrush, 280 - radius[1], 140 - radius[1], 2 * radius[1],
                2 * radius[1]);
            g.FillEllipse(whiteBrush, 280 - radius[2], 140 - radius[2], 2 * radius[2],
                2 * radius[2]);
            g.DrawRectangle(blackPen, 180, 40, 200, 200);
            break;
        case 2:
            // 带通掩码
            g.FillRectangle(blackBrush, 180, 40, 200, 200);
            g.FillEllipse(whiteBrush, 280 - radius[3], 140 - radius[3], 2 * radius[3],
                2 * radius[3]);
            g.FillEllipse(blackBrush, 280 - radius[4], 140 - radius[4], 2 * radius[4],
                2 * radius[4]);
            break;
        case 3:
            // 高通掩码
```

```
                g.FillRectangle(whiteBrush, 180, 40, 200, 200);
                g.FillEllipse(blackBrush, 280 - radius[3], 140 - radius[3], 2 * radius[3],
                    2 * radius[3]);
                g.DrawRectangle(blackPen, 180, 40, 200, 200);
                break;
            default:
                break;
        }

        // 释放对象
        blackBrush.Dispose();
        whiteBrush.Dispose();
        blackPen.Dispose();
    }
```

为了和主窗体之间传递数据，再添加两个 get 属性访问器，代码如下：

```
    public byte[] GetRadius
    {
        get
        {
            // 得到掩码半径数组
            return radius;
        }
    }

    public byte GetFlag
    {
        get
        {
            // 得到应用何种滤波形式
            return tempFlag;
        }
    }
```

（3）回到主窗体，为“成分滤波”按钮控件添加 Click 事件，代码如下：

```
    private void freGran_Click(object sender, EventArgs e)
    {
        if (curBitmap != null)
        {
            // 实例化granularity
            granularity granForm = new granularity();

            if (granForm.ShowDialog() == DialogResult.OK)
            {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] grayValues = new byte[bytes];
                System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
```

```
Complex[] freDom = new Complex[bytes];
byte[] tempArray = new byte[bytes];
byte[] tempRadius = new byte[6];

// 得到掩码半径数组
tempRadius = granForm.GetRadius;
// 得到滤波形式标识
byte flag = granForm.GetFlag;

// 计算掩码半径所对应真实图像的长度
int minLen=Math.Min(curBitmap.Width,curBitmap.Height);
double[] radius = new double[6];
for(int i = 0; i < 6; i++)
{
    radius[i] = tempRadius[i] * minLen / 100;
}

// 调用二维傅里叶变换，需要进行坐标位移
freDom = fft2(grayValues, curBitmap.Width, curBitmap.Height, true);

// 滤波
for (int i = 0; i < curBitmap.Height; i++)
{
    for (int j = 0; j < curBitmap.Width; j++)
    {
        // 当前像素点到图像几何中心的距离
        double distance = (double)((j - curBitmap.Width / 2) *
            (j - curBitmap.Width / 2) + (i - curBitmap.Height / 2) *
            (i - curBitmap.Height / 2));
        distance = Math.Sqrt(distance);

        switch (flag)
        {
            case 0:
                // 低通滤波
                if (distance > radius[0])
                {
                    freDom[i * curBitmap.Width + j] =
                        new Complex(0.0, 0.0);
                }
                break;
            case 1:
                // 带阻滤波
                if (distance < radius[1] && distance > radius[2])
                {
                    freDom[i * curBitmap.Width + j] =
                        new Complex(0.0, 0.0);
                }
                break;
            case 2:
                // 带通滤波
                if (distance > radius[3] || distance < radius[4])
                {
                    freDom[i * curBitmap.Width + j] =
                        new Complex(0.0, 0.0);
```

```
                            }
                            break;
                        case 3:
                            // 高通滤波
                            if (distance < radius[3])
                            {
                                freDom[i * curBitmap.Width + j] =
                                    new Complex(0.0, 0.0);
                            }
                            break;
                        default:
                            MessageBox.Show("无效! ");
                            break;
                    }
                }
            }

            // 调用二维傅里叶逆变换
            tempArray = ifft2(freDom, curBitmap.Width, curBitmap.Height, true);

            // 数组复制
            grayValues = (byte[])tempArray.Clone();

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            curBitmap.UnlockBits(bmpData);
        }

        Invalidate();
    }
}
```

（4）编译并运行该段程序。我们仍然以图 6.1 为例，对其进行带阻滤波。打开图像后，单击“成分滤波”按钮，打开频率成分滤波对话框，如图 6.5 所示设置相关参数。

图 6.5　带阻滤波对话框

单击“确定”按钮，得到带阻滤波后的图像，如图 6.6 所示，原图的中频成分被滤除掉，只留下高频和低频部分。

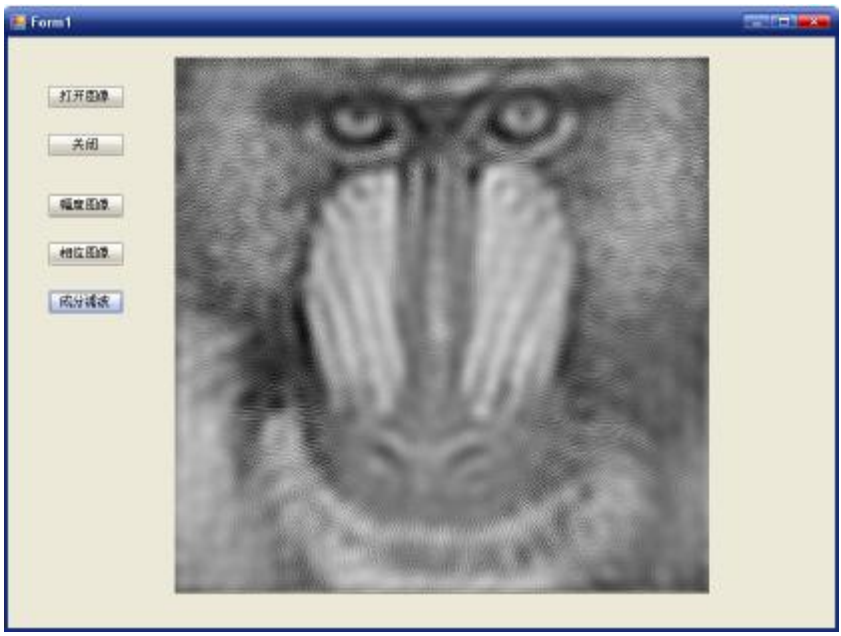
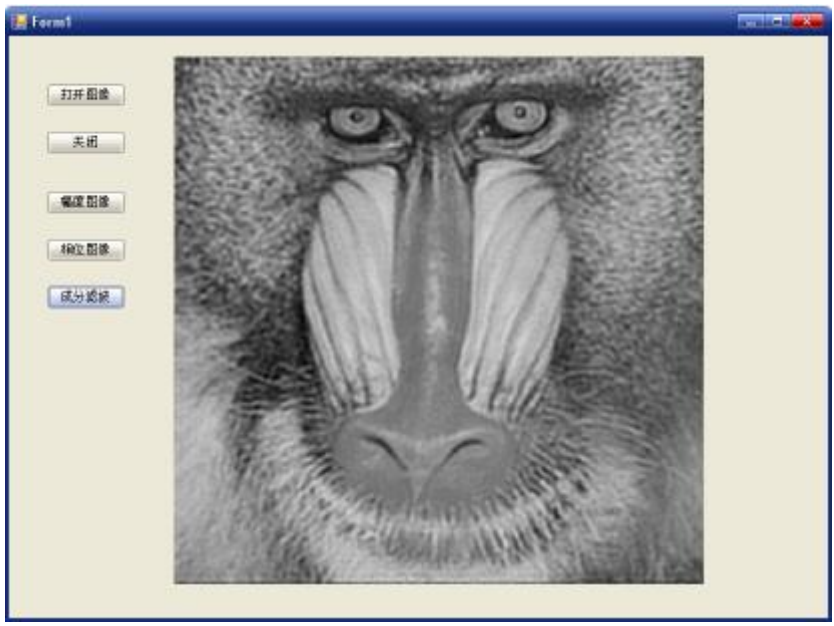
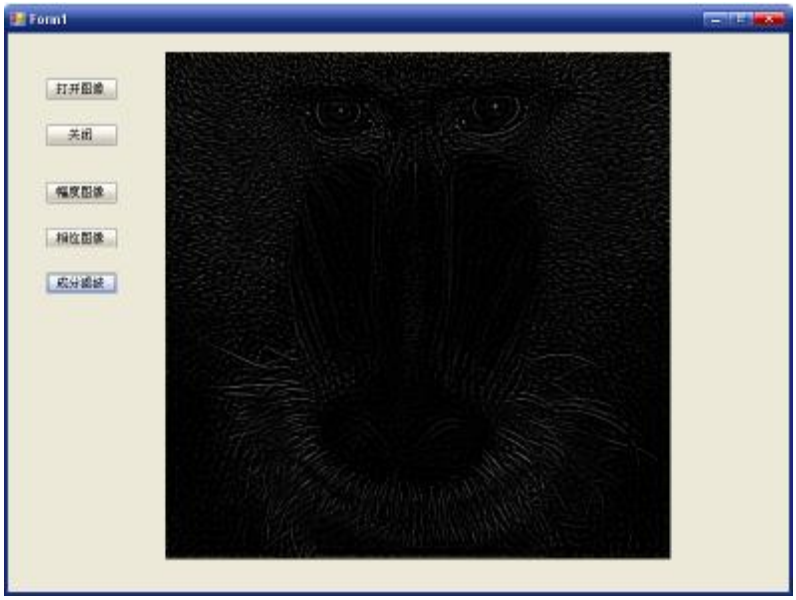


图 6.6　带阻滤波结果

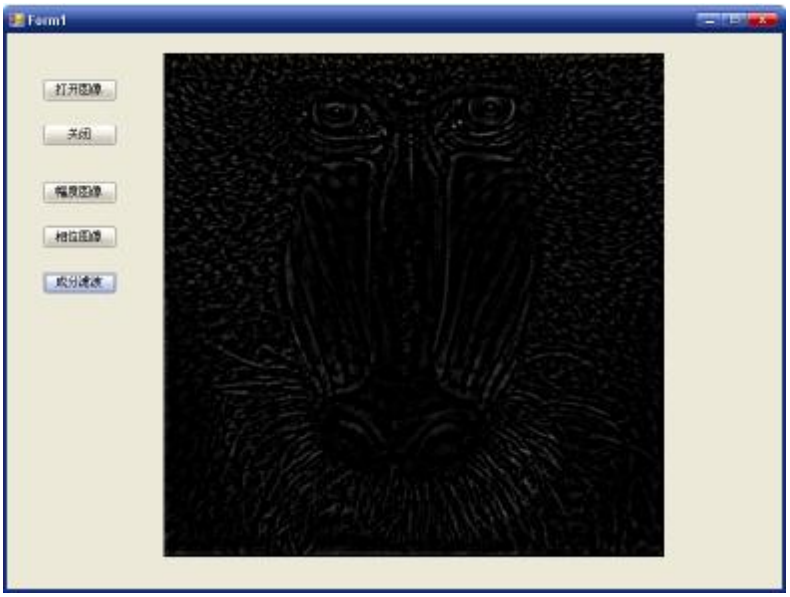
图 6.7 分别给出了低通、高通和带通滤波的结果。其中，低通掩码的半径为 15%，高通掩码的半径为 10%，带通掩码的内径和外径分别为 5%和 20%。



（a）低通滤波



（b）高通滤波



（c）带通滤波

图 6.7　低通、高通、带通滤波结果

6.5　频率方位滤波

6.5.1　频率方位滤波原理

图像频率的方位指的是它的角度。如果一幅图像经过离散傅里叶变换后，它的频谱沿着一条特殊的方位表现出更亮，则说明在该图中，沿着这个方向有很强烈的方位成分。

如上一节一样，我们仍然可以用掩码的方式来滤除掉图像中的某些方向信息。

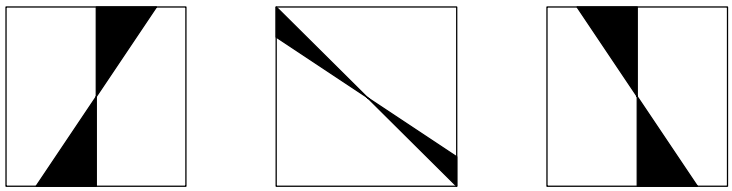


图 6.8 方位滤波掩码图例

图 6.8 为几个方位滤波的例子。与频率成分滤波方法一样，先对图像进行离散傅里叶变换，然后在频域内，与同等大小的方位掩码图像相乘，最后经过离散傅里叶逆变换，就可以滤除掉某些方向角度的信息了。

6.5.2 频率方位滤波编程实例

该实例通过方位掩码实现图像的方位滤波。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 6.4 所示。

表 6.4 所修改的属性

控 件	属 性	修 改 内 容
button1	Name	freOri
	Location	37，288
	Text	方位滤波

（2）创建 1 个名为 orienation 的 Windows 窗体，该窗体用于选择方位掩码图像参数，并绘制方位掩码图像。在该窗体内添加 2 个 Button 控件、3 个 Label 控件和 2 个 NumeriUpDown 控件，这些控件所需要修改的属性如表 6.5 所示。

表 6.5 所修改的属性

控 件	属 性	修 改 内 容
orientation	ControlBox	False
	Size	335，384
	Text	频率方位滤波
button1	Name	start
	Location	45，300
	Text	确定
button2	Name	close
	Location	204，300
	Text	关闭
label1	Location	30，10
	Text	掩码图预览：

label2	Location	21，257

续表

控　件	属　性	修 改 内 容
	Text	起始角度：
label3	Location	164，257
	Text	终止角度：
numeriUpDown1	Name	sOrie
	Location	81，252
	Maximun	135
	Minimum	−45
	Size	55，21
numeriUpDown2	Name	fOrie
	Location	224，252
	Maximum	225
	Minimum	−45
	Size	55，21

分别为 2 个 Button 控件添加 Click 事件，为 2 个 NumeriUpDown 控件添加 ValueChange 事件，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void sOrie_ValueChanged(object sender, EventArgs e)
{
    // 判断起始角度不大于终止角度，并且两者相差不大于90°
    if (sOrie.Value >= orient[1] || orient[1] - sOrie.Value > 90)
    {
        sOrie.Value = orient[0];
        return;
    }
    orient[0] = Convert.ToInt16(sOrie.Value);

    // 计算起始角度的边长，绘图时使用
    if (orient[0] >= -45 && orient[0] <= 45)
    {
        flag = (byte)((flag & 0xf0) | 0x01);
        leng1 = Convert.ToInt16(Math.Tan(orient[0] * (Math.PI / 180)) * 100);
    }
    else
    {
        flag = (byte)((flag & 0xf0) | 0x02);
```

```
            leng1 = Convert.ToInt16(Math.Tan(orient[0] * (Math.PI / 180) - Math.PI / 2) * 100);
        }

        Invalidate();
    }

    private void fOrie_ValueChanged(object sender, EventArgs e)
    {
        // 判断终止角度不小于起始角度，并两者相差不大于90°
        if (fOrie.Value <= orient[0] || fOrie.Value - orient[0] > 90)
        {
            fOrie.Value = orient[1];
            return;
        }
        orient[1] = Convert.ToInt16(fOrie.Value);

        // 计算终止角度的边长，绘图时使用
        if (orient[1] >= -45 && orient[1] <= 45)
        {
            flag = (byte)((flag & 0x0f) | 0x10);
            leng2 = Convert.ToInt16(Math.Tan(orient[1] * (Math.PI / 180)) * 100);
        }
        else
        {
            if (orient[1] > 45 && orient[1] <= 135)
            {
                flag = (byte)((flag & 0x0f) | 0x20);
                leng2 = Convert.ToInt16(Math.Tan(orient[1] *
                    (Math.PI / 180) - Math.PI / 2) * 100);
            }
            else
            {
                flag = (byte)((flag & 0x0f) | 0x40);
                leng2 = Convert.ToInt16(Math.Tan(orient[1] * (Math.PI / 180) - Math.PI) * 100);
            }
        }

        Invalidate();
    }
```

需要说明的是，在程序中，我们限定起始角度与终止角度之间不大于 90°，即掩码角度
不大于 90°。

其中 flag 为 byte 型变量，用来标识起始角度和终止角度的位置状态，该变量后 4 位表示
起始角度，第 0 位置位表示它在−45°～45°之间，第 1 位置位表示在 45°～135°之间；前
4 位表示终止角度，第 4 位置位表示它在−45°～45°之间，第 5 位置位表示在 45°～135°
之间，第 6 位置位表示在其他角度。orient 为 int 型数组变量，用来记录两个角度的度数，leng
和 leng2 为 int 型变量，分别用来记录两个角度的长度，它们在构造函数中被初始化，代码如下

```
    flag = 0x11;
    orient = new int[2] { 45, 90 };
    leng1 = Convert.ToInt16(Math.Tan(orient[0] * (Math.PI / 180)) * 100);
    leng2 = Convert.ToInt16(Math.Tan(orient[1] * (Math.PI / 180) - Math.PI / 2) * 100);
    // 起始角度
    sOrie.Value = orient[0];
```

```
// 终止角度
fOrie.Value = orient[1];
```

为了在该窗体内绘出掩码图，还必须为该窗体添加一个 Paint 事件，并且为了与主窗体传递掩码角度数据，还必须添加一个 get 属性访问器，代码如下：

```
private void orientation_Paint(object sender, PaintEventArgs e)
{
    SolidBrush blackBrush = new SolidBrush(Color.Black);
    SolidBrush whiteBrush = new SolidBrush(Color.White);
    Pen blackPen = new Pen(Color.Black, 1.5F);
    Graphics g = e.Graphics;

    // 绘制掩码图矩形
    g.FillRectangle(whiteBrush, 60, 30, 200, 200);
    g.DrawRectangle(blackPen, 60, 30, 200, 200);

    Point point1, point2, point3, point4;

    // 绘制掩码
    switch (flag)
    {
        case 0x11:
            // 起始角度和终止角度都在-45° ~ 45° 之间
            point1 = new Point(260, 130 - leng1);
            point2 = new Point(260, 130 - leng2);
            point3 = new Point(60, 130 + leng1);
            point4 = new Point(60, 130 + leng2);
            Point[] curvePoints1 = { point1, point2, new Point(160, 130), point3, point4
                new Point(160, 130) };
            g.FillPolygon(blackBrush, curvePoints1);
            break;
        case 0x21:
            // 起始角度在-45° ~ 45° 之间，终止角度在45° ~ 135° 之间
            point1 = new Point(260, 130 - leng1);
            point2 = new Point(160 - leng2, 30);
            point3 = new Point(60, 130 + leng1);
            point4 = new Point(160 + leng2, 230);
            Point[] curvePoints2 = { point1, new Point(260, 30), point2,
                new Point(160, 130), point3, new Point(60, 230), point4,
                new Point(160, 130) };
            g.FillPolygon(blackBrush, curvePoints2);
            break;
        case 0x22:
            // 起始角度和终止角度都在45° ~ 135° 之间
            point1 = new Point(160 - leng1, 30);
            point2 = new Point(160 - leng2, 30);
            point3 = new Point(160 + leng1, 230);
            point4 = new Point(160 + leng2, 230);
            Point[] curvePoints3 = { point1, point2, new Point(160, 130), point3, point4
                new Point(160, 130) };
            g.FillPolygon(blackBrush, curvePoints3);
            break;
        case 0x42:
            // 起始角度在45° ~ 135° 之间，终止角度在135° 以上
            point1 = new Point(160 - leng1, 30);
```

```
                point2 = new Point(60, 130 + leng2);
                point3 = new Point(160 + leng1, 230);
                point4 = new Point(260, 130 - leng2);
                Point[] curvePoints4 = { point1, new Point(60, 30), point2,
                    new Point(160, 130), point3, new Point(260, 230), point4,
                    new Point(160, 130) };
                g.FillPolygon(blackBrush, curvePoints4);
                break;
            default:
                MessageBox.Show("无效! ");
                break;
        }

        blackBrush.Dispose();
        whiteBrush.Dispose();
        blackPen.Dispose();
    }

    public int[] GetOrient
    {
        get
        {
            // 得到起始角度和终止角度
            return orient;
        }
    }
```

（3）回到主窗体，为“方位滤波”按钮控件添加 Click 事件，代码如下：

```
private void freOri_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化orientation
        orientation orieForm = new orientation();

        if (orieForm.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            Complex[] freDom = new Complex[bytes];
            byte[] tempArray = new byte[bytes];

            int[] tempOrient = new int[2];
            // 得到起始角度和终止角度
            tempOrient = orieForm.GetOrient;

            byte flag = 1;
            // 确定所滤波的方位的区间
```

```
if (tempOrient[1] <= 0)
{
    flag = 1;
}
if(tempOrient[0] <= 0 && tempOrient[1] > 0)
{
    flag = 2;
}
if (tempOrient[0] > 0 && tempOrient[1] < 180)
{
    flag = 3;
}
if (tempOrient[1] > 180)
{
    flag = 4;
}

// 调用二维傅里叶变换，需要坐标位移
freDom = fft2(grayValues, curBitmap.Width, curBitmap.Height, true);

double tempD;

for (int i = 0; i < curBitmap.Height; i++)
{
    for (int j = 0; j < curBitmap.Width; j++)
    {
        // 得到当前像素相对于图像中心的角度
        tempD = (Math.Atan2(curBitmap.Height / 2 - i,
            j - curBitmap.Width / 2)) * 180 / Math.PI;

        // 掩码赋值
        switch (flag)
        {
            case 1:
                if ((tempD <= tempOrient[1] && tempD >= tempOrient[0])
                    || (tempD <= (tempOrient[1] + 180)
                    && tempD >= (tempOrient[0] + 180)))
                {
                    freDom[i * curBitmap.Width + j] =
                        new Complex(0.0, 0.0);
                }
                break;
            case 2:
                if ((tempD <= tempOrient[1] && tempD >= tempOrient[0])
                    || tempD <= tempOrient[1] - 180
                    || tempD > tempOrient[0] + 180 )
                {
                    freDom[i * curBitmap.Width + j] =
                        new Complex(0.0, 0.0);
                }
                break;
            case 3:
                if ((tempD <= tempOrient[1] && tempD >= tempOrient[0])
                    || (tempD <= tempOrient[1] - 180
                    && tempD >= tempOrient[0] - 180))
                {
```

```
                                freDom[i * curBitmap.Width + j] =
                                    new Complex(0.0, 0.0);
                            }
                            break;
                        case 4:
                            if (((tempD <= tempOrient[1] - 180)
                                && (tempD >= tempOrient[0] - 180))
                                || tempD <= tempOrient[1] - 360
                                || tempD >= tempOrient[0])
                            {
                                freDom[i * curBitmap.Width + j] =
                                    new Complex(0.0, 0.0);
                            }
                            break;
                        default:
                            MessageBox.Show("无效! ");
                            break;
                    }

                }
            }

            // 调用二维傅里叶逆变换
            tempArray = ifft2(freDom, curBitmap.Width, curBitmap.Height, true);

            grayValues = (byte[])tempArray.Clone();

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            rBitmap.UnlockBits(bmpData);
        }

        Invalidate();
    }
}
```

（4）编译并运行该段程序，仍然对图 6.1 所示的图像进行测试。打开该图，然后单击“立滤波”按钮，打开频率方位滤波窗口，如图 6.9 所示设置相关参数，最后单击“确定”按钮，完成方位滤波功能，结果如图 6.10 所示。

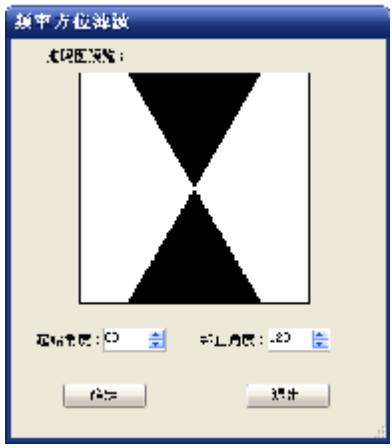


图 6.9　频率方位滤波对话框

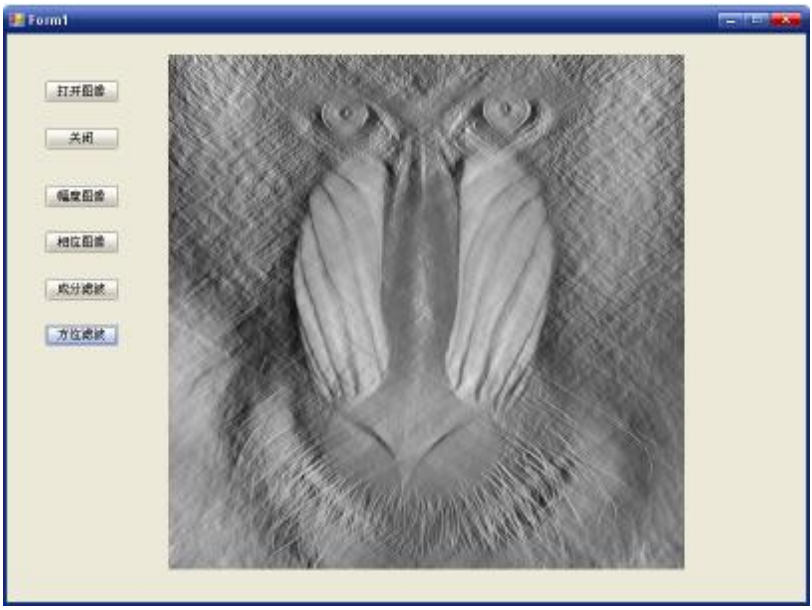


图 6.10 方位滤波

图 6.9 所示设置的参数是要滤除水平方向的成分，从图 6.10 可以看出，水平方向的成分基本被滤波掉了，而与之相对应的垂直方向的成分基本都被保留了下来。

6.6 小结

频率变换（傅里叶变换）是图像处理中最基本的工具之一。本章只是用到了它的最基本的应用，但本章所详述的概念和技术为本书后续章节所要讨论的其他图像处理技术奠定了基础。在以后的章节中，还要应用它做更复杂的处理。

第 7 章　图像平滑与去噪

图像平滑是用于突出图像的宽大区域和主干部分（低频成分）或抑制图像噪声和干扰高频成分），使图像亮度平缓渐变，减小突变梯度，改善图像质量的图像处理方法。它属于图像增强的一部分。这一章主要介绍的图像平滑方法有：均值滤波、中值滤波、灰度形态学滤波、小波滤波、高斯低通滤波以及统计方法滤波。不同的方法能够滤除掉不同形式的噪声。

7.1　噪声模型

在进行平滑去噪之前，必须先介绍噪声模型。根据不同的噪声，应用不同的平滑去噪方法。

7.1.1　噪声概述

图像噪声就是图像中不希望看到的成分，它主要来源于图像的获取和传输过程中。由噪声与图像的关系，噪声一般可分为加性噪声和乘性噪声两种。

$$f(\cdot) = g(\cdot) + q(\cdot) \tag{7.1}$$

$$f(\cdot) = g(\cdot)q(\cdot) \tag{7.2}$$

公式（7.1）和（7.2)分别为加性噪声和乘性噪声的数学表达式，其中 $f(\cdot)$、$g(\cdot)$ 和 $q(\cdot)$ 分别表示图像、希望得到的部分和噪声部分。这两种噪声可以通过取对数和指数相互转换：

$$e^{f} = e^{g+p} = e^{g}e^{q} \tag{7.3}$$

$$\log f = \log(gq) = \log g + \log q \tag{7.4}$$

在这里主要介绍均匀分布噪声、高斯噪声、瑞利噪声、指数噪声和椒盐噪声。前 4 种噪声属于加性噪声，而椒盐噪声既不属于加性噪声也不属于乘性噪声。

噪声是由噪声分量灰度值的统计特性来描述的，它们可以被认为是由概率密度函数（PDF）表示的随机变量，下面就介绍这几种常见的噪声概率密度函数。

均匀分布噪声变量 z 的概率密度函数为：

$$p(z)=\begin{cases}\dfrac{1}{b-a} & a\leqslant z\leqslant b\\ 0 & \text{其他}\end{cases} \tag{7.5}$$

z 表示灰度值，它的期望值和方差分别为$(a+b)/2$ 和$(b-a)^2/12$。

高斯随机变量 z 的概率密度函数为：

$$p(z)=\frac{1}{\sqrt{2\pi}s}e^{-(z-m)^2/2s^2} \tag{7.6}$$

μ表示 z 的期望值，σ表示 z 的标准差。

瑞利噪声的概率密度函数为：

$$p(z)=\begin{cases}\dfrac{2}{b}(z-a)e^{-(z-a)^2/b} & z\geqslant a\\ 0 & z<a\end{cases} \tag{7.7}$$

它的均值和方差分别为 $a+\sqrt{\pi b}/4$ 和 $b(4-\pi)/4$。

指数分布噪声的概率密度函数为：

$$p(z)=\begin{cases}ae^{-az} & z\geq 0\\ 0 & z<0\end{cases} \tag{7.8}$$

它的均值和方差分别为 $1/a$ 和 $1/a^2$，其中 $a>0$。

椒盐噪声又称双极脉冲噪声，它类似于随机分布在图像上的胡椒（黑色）和咸盐（白色）数粒。它的概率密度函数为：

$$p(z)=\begin{cases}P_a & z=a\\ P_b & z=b\\ 0 & \text{其他}\end{cases} \tag{7.9}$$

对于一个 8 位图像，a=0，b=255。

7.1.2　噪声模型编程实例

该实例构造了上述几种常见的噪声模型。均匀分布噪声在实践中应用的很少，它只是作为其他噪声产生的模拟随机数，因此在这里我们只对其他噪声模型进行编程实现。

（1）创建 1 个“模板主窗体”，并在该窗体内添加 1 个 Button 控件，其属性修改如表 7.1 所示。

表 7.1　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	noise
	Text	噪声模型
	Location	37，150

（2）创建 1 个名为 noiseModel 的 Windows 窗体，该窗体用于选择某种噪声模型。在该窗体内添加 2 个 Button 控件、4 个 RadioButton 控件、7 个 Label 和 TextBox 控件，其属性修

如表 7.2 所示。

表 7.2 所修改的属性

控 件	属 性	所修改内容
noiseModel	ControlBox	False
	Size	410，310
	Text	噪声模型
button1	Name	start
	Location	80，225
	Text	确定
button2	Name	close
	Location	250，225
	Text	退出
radioButton1	Name	gaussian
	Location	41，33
	Text	高斯噪声
	Checked	True
radioButton2	Name	rayleigh
	Location	41，79
	Text	瑞利噪声
radioButton3	Name	exponential
	Location	41，125
	Text	指数噪声
radioButton4	Name	saltpepper
	Location	41，171
	Text	椒盐噪声
textBox1	Name	mean
	Location	210，32
	Text	0
textBox2	Name	stanDev
	Location	344，32
	Text	20
textBox3	Name	paraRA
	Location	180，78
	Text	0
	Enabled	False
textBox4	Name	paraRB
	Location	294，78
	Text	250
	Enabled	False
textBox5	Name	paraEA
	Location	214，124
	Text	0.1
	Enabled	False

续表

控　件	属　性	所修改内容
textBox6	Name	pepper
	Location	192，170
	Text	0.02
	Enabled	False
textBox7	Name	salt
	Location	314，170
	Text	0.02
	Enabled	False
label1	Location	133，35
	Text	均值（μ）：
label2	Location	255，35
	Text	均方差（σ）：
label3	Location	133，81
	Text	参数 a：
label4	Location	247，81
	Text	参数 b：
label5	Location	133，127
	Text	参数 a(a>0)：
label6	Location	133，173
	Text	含椒量：
label7	Location	255，173
	Text	含盐量：

分别为 2 个 Button 控件添加 Click 事件，为 4 个 radioButton 控件添加 CheckedChanged 事件，以及 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    switch (flagN)
    {
        case 0:
            // 高斯噪声
            // 得到均值
            paraN[0] = Convert.ToDouble(mean.Text);
            // 得到均方差
            paraN[1] = Convert.ToDouble(stanDev.Text);
            break;
        case 1:
            // 瑞利噪声
            // 得到参数a
            paraN[0] = Convert.ToDouble(paraRA.Text);
            // 得到参数b
            paraN[1] = Convert.ToDouble(paraRB.Text);
            break;
        case 2:
```

```
                // 指数噪声
                // 得到参数a
                paraN[0] = Convert.ToDouble(paraEA.Text);
                // 判断参数是否大于零
                if (paraN[0] <= 0)
                {
                    paraEA.Text = "0.1";
                    return;
                }
                break;
            case 3:
                // 椒盐噪声
                // 含椒量
                paraN[0] = Convert.ToDouble(pepper.Text);
                // 含盐量
                paraN[1] = Convert.ToDouble(salt.Text);
                // 判断两个含量之和是否小于1
                if (paraN[0] +paraN[1] >= 1)
                {
                    pepper.Text = "0.02";
                    salt.Text = "0.02";
                    return;
                }
                break;
            default:
                break;
        }
        this.DialogResult = DialogResult.OK;
    }

    private void close_Click(object sender, EventArgs e)
    {
        this.Close();
    }

    public double[] GetParaN
    {
        get
        {
            // 得到噪声模型的参数
            return paraN;
        }
    }

    public byte GetFlag
    {
        get
        {
            // 得到所用何种噪声模型
            return flagN;
        }
    }

    private void gaussian_CheckedChanged(object sender, EventArgs e)
    {
```

```
        // 标识变量
        flagN = 0;

        mean.Enabled = true;
        stanDev.Enabled = true;
        paraRA.Enabled = false;
        paraRB.Enabled = false;
        paraEA.Enabled = false;
        pepper.Enabled = false;
        salt.Enabled = false;
    }

    private void rayleigh_CheckedChanged(object sender, EventArgs e)
    {
        // 标识变量
        flagN = 1;

        mean.Enabled = false;
        stanDev.Enabled = false;
        paraRA.Enabled = true;
        paraRB.Enabled = true;
        paraEA.Enabled = false;
        pepper.Enabled = false;
        salt.Enabled = false;
    }

    private void exponential_CheckedChanged(object sender, EventArgs e)
    {
        // 标识变量
        flagN = 2;

        mean.Enabled = false;
        stanDev.Enabled = false;
        paraRA.Enabled = false;
        paraRB.Enabled = false;
        paraEA.Enabled = true;
        pepper.Enabled = false;
        salt.Enabled = false;
    }

    private void saltpepper_CheckedChanged(object sender, EventArgs e)
    {
        // 标识变量
        flagN = 3;

        mean.Enabled = false;
        stanDev.Enabled = false;
        paraRA.Enabled = false;
        paraRB.Enabled = false;
        paraEA.Enabled = false;
        pepper.Enabled = true;
        salt.Enabled = true;
    }
```

其中 flagN 和 paraN 分别是 byte 型变量和 double 型数组变量，flagN 用于标识所用何种

噪声模型，paraN 表示噪声模型的参数，它们在构造函数中被初始化，代码如下：

```
flagN = 0;
paraN = new double[2] { 0, 20 };
```

（3）回到主窗体，为“噪声模型”按钮控件添加 Click 事件，代码如下：

```
private void noise_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化noiseModel
        noiseModel noise = new noiseModel();

        if (noise.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            double temp = 0;
            // 得到所用何种噪声模型
            byte flagNoise = noise.GetFlag;
            double[] paraNoise = new double[2];
            // 得到噪声模型参数
            paraNoise = noise.GetParaN;

            // 得到两个均匀随机数
            Random r1, r2;
            r1 = new Random(unchecked((int)DateTime.Now.Ticks));
            r2 = new Random(~unchecked((int)DateTime.Now.Ticks));
            double v1, v2;

            for (int i = 0; i < bytes; i++)
            {
                switch (flagNoise)
                {
                    case 0:
                        // 高斯噪声
                        do
                        {
                            v1 = r1.NextDouble();
                        }
                        while (v1 <= 0.00000000001);
                        v2 = r2.NextDouble();
                        // 应用雅可比变换，直接生成正态分布
                        temp = Math.Sqrt(-2 * Math.Log(v1)) * Math.Cos(2 * Math.PI *
                            v2) * paraNoise[1] + paraNoise[0];
                        break;
                    case 1:
                        // 瑞利噪声
                        do
                        {
                            v1 = r1.NextDouble();
```

```
                        }
                        while (v1 >= 0.9999999999);
                        temp = paraNoise[0] + Math.Sqrt(-1 * paraNoise[1] *
                            Math.Log(1 - v1));
                        break;
                    case 2:
                        // 指数噪声
                        do
                        {
                            v1 = r1.NextDouble();
                        }
                        while (v1 >= 0.9999999999);
                        temp = -1 * Math.Log(1 - v1) / paraNoise[0];
                        break;
                    case 3:
                        // 椒盐噪声
                        v1 = r1.NextDouble();
                        if (v1 <= paraNoise[0])
                            temp = -500;
                        else if (v1 >= (1 - paraNoise[1]))
                            temp = 500;
                        else
                            temp = 0;
                        break;
                    default:
                        MessageBox.Show("无效! ");
                        break;
                }
                // 加性噪声
                temp = temp + grayValues[i];

                if (temp > 255)
                {
                    grayValues[i] = 255;
                }
                else if (temp < 0)
                {
                    grayValues[i] = 0;
                }
                else
                    grayValues[i] = Convert.ToByte(temp);
            }

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            curBitmap.UnlockBits(bmpData);
        }

        Invalidate();
    }
}
```

（4）编译并运行该段程序，仍然以图 4.1 所示的图像为例。打开该图后，单击“噪声模型”按钮，打开噪声模型对话框，选择瑞利噪声模型，并设置相关参数，如图 7.1 所示。

单击“确定”按钮后，受瑞利噪声干扰的图像就显示出来，如图 7.2 所示。

再分别生成其他 3 种噪声模型：高斯噪声、椒盐噪声和指数噪声。高斯噪声的均值为 0，均方差为 15，如图 7.3 所示；椒盐噪声的含椒量和含盐量都是 0.015，如图 7.4 所示；指数噪声的参数 a 为 0.05，如图 7.5 所示。

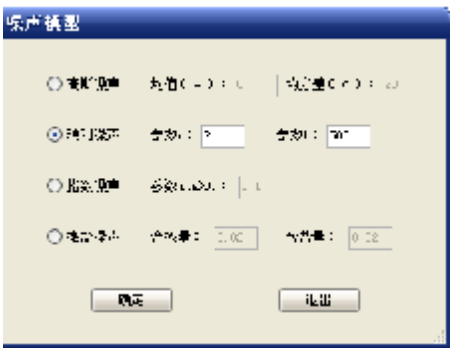


图 7.1　噪声模型

图 7.2　受瑞利噪声干扰的图像

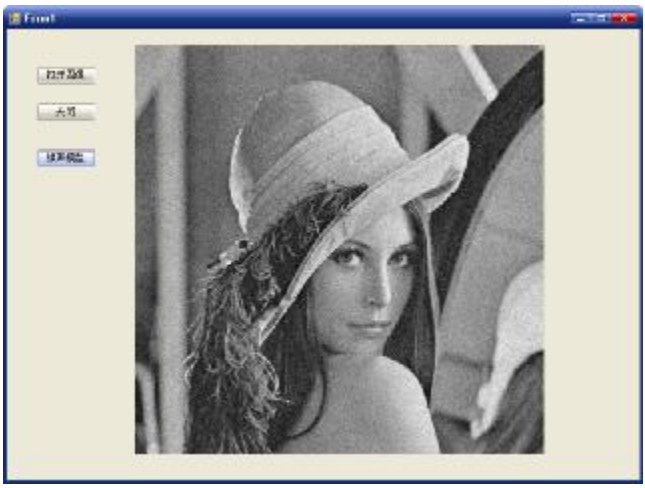


图 7.3　受高斯噪声干扰的图像

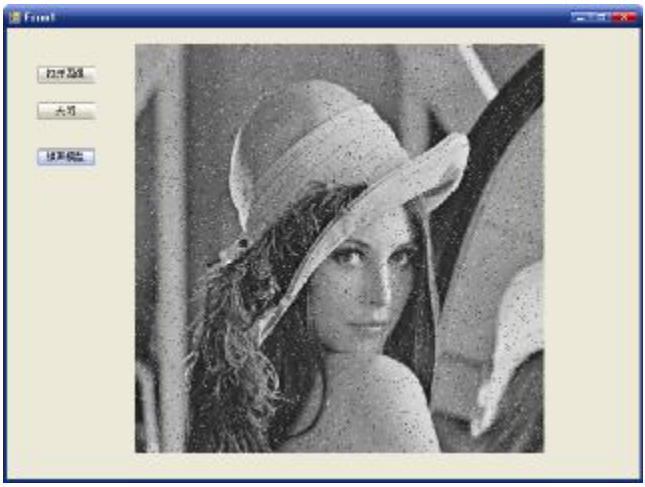


图 7.4　受椒盐噪声干扰的图像

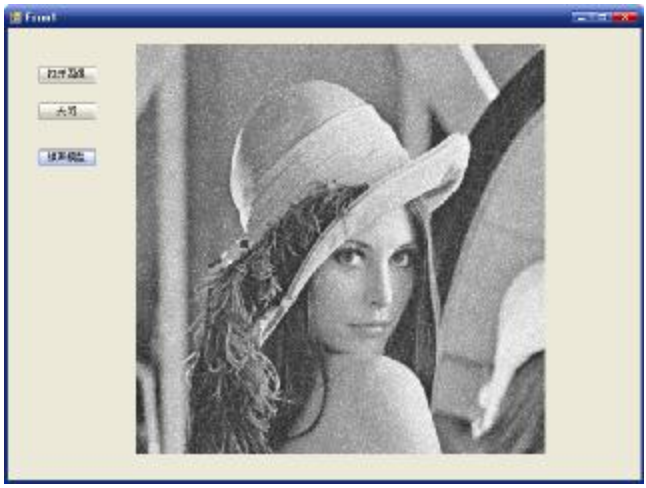


图 7.5 受指数噪声干扰的图像

.2 均值滤波与中值滤波

7.2.1 均值滤波与中值滤波原理

均值滤波属于线性滤波方法，中值滤波属于非线性滤波方法。它们都属于空间域内平滑方法。对于给定的图像 $f(x, y)$中的每个像素点(x, y)，取其邻域 S_{xy}，设 S_{xy} 含有 M 个像素，取其平均值作为处理后所得图像像素点(x, y)处的灰度，该方法称为均值滤波方法；而取其中值作为处理后所得图像像素点（x, y）处的灰度，该方法称为中值滤波。

均值滤波方法能有力地抑制噪声，但同时也由于平均而引起了模糊现象，模糊程度与领域半径成正比。中值滤波方法主要功能是让周围像素灰度值的差比较大的像素改为选取与周围的像素值接近的值，从而可以消除孤立的噪声点，所以中值滤波对于滤除图像的椒盐噪声非常有效。中值滤波可以做到既去除噪声又能保护图像的边缘，从而获得较满意的复原效果。

7.2.2 均值滤波与中值滤波编程实例

该实例实现了均值滤波功能和中值滤波功能。在这里，用模板来表示邻域 S_{xy}，一般来说模板都是中心对称的，所以我们取的模板为正方形，边长为 3、5、7…等奇数像素长，因此 M 分别为 9、25、49…等。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 7.3 所示。

表 7.3 所修改的属性

控 件	属 性	所修改内容
Button1	Name	meanMedian
	Text	均值与中值
	Location	37，196

（2）创建1个名为meanMedian的Windows窗体，该窗体用于选择模板的大小和所用波方法。在该窗体内添加2个Button控件、2个RadioButton控件、1个Label控件和1个NumericUpDown控件，其属性修改如表7.4所示。

表 7.4　　所修改的属性

控　　件	属　　性	所修改内容
meanMedian	Size	282，227
	Text	均值滤波与中值滤波
	ControlBox	False
Button1	Name	start
	Location	27，135
	Text	确定
Button2	Name	close
	Location	156，135
	Text	退出
RadioButton1	Name	mean
	Location	27，40
	Checked	true
	Text	均值滤波
RadioButton2	Name	median
	Location	27，90
	Text	中值滤波
Label1	Location	118，65
	Text	模板边长
NumericUpDown1	Name	length
	Location	189，63
	Maximum	7
	Minimum	3
	Increment	2
	Size	42，21
	Value	3

分别为2个Button控件添加Click事件，并添加2个get属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public byte GetLength
```

```
{
    get
    {
        // 得到模板边长
        return (byte)length.Value;
    }
}

public bool GetFlag
{
    get
    {
        // 得到所用何种滤波方法
        return median.Checked;
    }
}
```

（3）回到主窗体，为“均值与中值”按钮添加 Click 事件，代码如下：

```
private void meanMedian_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化meanMedian
        meanMedian meaAndMed = new meanMedian();

        if (meaAndMed.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            byte[] tempArray = new byte[bytes];

            // 得到模板边长
            byte sideLength = meaAndMed.GetLength;
            byte halfLength = (byte)(sideLength / 2);
            // 得到所用何种方法滤波
            bool flagM = meaAndMed.GetFlag;

            for (int i = halfLength; i < curBitmap.Height - halfLength; i++)
            {
                for (int j = halfLength; j < curBitmap.Width - halfLength; j++)
                {
                    switch (sideLength)
                    {
                        case 3:
                            // 3×3 模板
                            if (flagM == false)
                            {
                                // 均值滤波
```

```
                                // 求取均值
        tempArray[i * curBitmap.Width + j] = (byte)((grayValues[i * curBitmap.Width + j] +
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)]) / 9);
                            }
                            else
                            {
                                // 中值滤波
                                // 定义数组
        byte[] sortArray = new byte[] {grayValues[i * curBitmap.Width + j],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)]};
                                // 调用排序方法
                                Sort(sortArray);
                                // 取中值
                                tempArray[i * curBitmap.Width + j] = sortArray[4];
                            }
                            break;
                        case 5:
                            // 5×5 模板
                            if (flagM == false)
                            {
                                // 均值滤波
                                // 求取均值
        tempArray[i * curBitmap.Width + j] =
            (byte)((grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2)) % curBitmap.Width] +
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1)) % curBitmap.Width] +
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j] +
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
```

```
            (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + j] +
        grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + ((j + 2) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)]) / 25);
                            }
                            else
                            {
                                // 中值滤波
                                // 定义数组
    byte[] sortArray =
        new byte[] {grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width
            (Math.Abs(j - 2)) % curBitmap.Width],
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1)) % curBitmap.Width],
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j],
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)],
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)],
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2) % curBitmap.Width)],
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)],
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j],
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)],
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)],
```

```
            grayValues[i * curBitmap.Width + (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + j],
            grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + ((j + 2) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)]};
                                // 调用排序方法
                                Sort(sortArray);
                                // 取中值
                                tempArray[i * curBitmap.Width + j] = sortArray[12];
                            }
                            break;
                        case 7:
                            // 7×7 模板
                            if (flagM == false)
                            {
                                // 均值滤波
                                // 求取均值
    tempArray[i * curBitmap.Width + j] =
        (byte)((grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2)) % curBitmap.Width] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1)) % curBitmap.Width] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + (Math.Abs(j - 2) % curBitmap.Width)] +
```

```
            grayValues[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)] +
            grayValues[i * curBitmap.Width + j] +
            grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            grayValues[i * curBitmap.Width + ((j + 2) % curBitmap.Width)] +
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width + j] +
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width + j] +
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            grayValues[i * curBitmap.Width + ((j + 3) % curBitmap.Width)] +
            grayValues[i * curBitmap.Width + (Math.Abs(j - 3) % curBitmap.Width)] +
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
```

```
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width + j] +
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)]) / 49);
                            }
                            else
                            {
                                // 中值滤波
                                // 定义数组
        byte[] sortArray =
            new byte[] {grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width
                (Math.Abs(j - 2)) % curBitmap.Width],
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1)) % curBitmap.Width],
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + j],
            grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + ((j + 2) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)],
```

```
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + ((j + 3) % curBitmap.Width)],
            grayValues[i * curBitmap.Width + (Math.Abs(j - 3) % curBitmap.Width)],
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)],
            grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)],
            grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)],
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)],
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)],
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)],
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)],
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width + j],
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)],
            grayValues[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)]};
                            // 调用排序方法
                            Sort(sortArray);
                            // 取中值
                            tempArray[i * curBitmap.Width + j] = sortArray[24];
```

```
                                }
                                break;
                            default:
                                MessageBox.Show("无效! ");
                                break;
                        }
                    }
                }

                grayValues = (byte[])tempArray.Clone();

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }

    /*****************************************
    选择排序法方法
    list: 所要排序的序列
    *****************************************/
    private void Sort(byte[] list)
    {
        int min;
        for (int i = 0; i < list.Length - 1; i++)
        {
            min = i;
            for (int j = i + 1; j < list.Length; j++)
            {
                if (list[j] < list[min])
                    min = j;
            }
            byte t = list[min];
            list[min] = list[i];
            list[i] = t;
        }
    }
```

需要说明的是，在用模板方法对图像进行处理时，都会因图像四周无法应用模板，而出现无法处理到的区域，一般都会用黑色像素或白色像素填补这些区域。就像第 5 章那样，我们没有考虑图像的四周边框，而仅仅用白色像素来填充。但是，在上面所列写的程序中，我们用循环延拓的方法来克服这个不足，从而使整幅图像都可以应用模板进行均值滤波和中值滤波。当然它的代价就是延长了运算时间。

（4）编译并运行该段程序，对图 7.3 所示的受到高斯噪声（均值为 0，均方差为 15）干扰的图像应用均值滤波方法，对图 7.4 所示的受到椒盐噪声（含椒量和含盐量都是 0.015）干扰的图像应用中值滤波方法。单击“均值与中值”按钮，打开

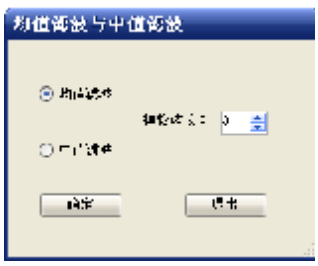


图 7.6　均值与中值滤波对话框

均值与中值滤波对话框，如图 7.6 所示，然后在该窗体内分别设置相应的参数，其结果如图 7.7 和图 7.8 所示。

图 7.7 均值滤波结果（使用 5×5 模板）

图 7.8 中值滤波结果（使用 3×3 模板）

7.3 灰度形态学滤波

7.3.1 灰度形态学原理

第 5 章介绍了二值形态学中，它只能处理二值图像。在这里，我们要对灰度图像进行处理，所以要应用灰度形态学理论。

与二值形态学一样，灰度形态学也包括腐蚀运算、膨胀运算、开运算和闭运算。对灰度图像而言，膨胀运算和腐蚀运算是以像素邻域的最大值和最小值来定义的。

腐蚀运算的数学定义为：

$$g(x,y)=erode[f(x,y),\mathbf{B}]=\min\{f(x+x',y+y')-\mathbf{B}(x',y')|(x',y')\in D_b\} \quad (7.10)$$

其中，$g(x,y)$为腐蚀后的灰度图像，$f(x,y)$为原灰度图像，$\mathbf{B}$ 为结构元素。

膨胀运算的数学定义为：

$$g(x,y)=dilate[f(x,y),\mathbf{B}]=\max\{f(x-x',y-y')+\mathbf{B}(x',y')|(x',y')\in D_b\} \quad (7.11)$$

实际上，在灰度形态学中，一般选择平坦的结构元素。所谓“平坦”，就是指结构元素的高度为零。因此，这样的结构元素，$\mathbf{B}$ 的值在 D_b 的定义域内的所有坐标处均为 0,则上面两个公式可以重写为：

$$g(x,y)=erode[f(x,y),\mathbf{B}]=\min\{f(x+x',y+y')|(x',y')\in D_b\} \quad (7.12)$$

$$g(x,y)=dilate[f(x,y),\mathbf{B}]=\max\{f(x-x',y-y')|(x',y')\in D_b\} \quad (7.13)$$

本书采用的都是平坦结构元素。

灰度图像的开运算和闭运算分别定义为：

$$g(x,y)=open[f(x,y),\mathbf{B}]=dilate\{erode[f(x,y),\mathbf{B}],\mathbf{B}\} \quad (7.14)$$

$$g(x,y)=close[f(x,y),\mathbf{B}]=erode\{dilate[f(x,y),\mathbf{B}],\mathbf{B}\} \quad (7.15)$$

7.3.2　灰度形态学去噪原理

用灰度形态学对图像进行平滑去噪，就是将开运算和闭运算结合起来，因为开运算和闭运算可以分别去除背景和图像本身中比结构元素矩阵小的噪声。根据该方法的特点可以知道它适用的图像类型是图像中的对象尺寸都比较大，且没有细小的细节，对这种类型的图像去噪的效果会比较好。

7.3.3　灰度形态学去噪编程实现

该实例实现了应用灰度形态学去除图像中的噪声的功能。

（1）在主窗体内添加一个 Button 控件，其属性修改如表 7.5 所示。

表 7.5　　所修改的属性

控　件	属　性	所修改内容
button1	Name	grayMor
	Text	灰度形态学
	Location	37，242

（2）创建一个名为 morphologic 的 Windows 窗体，该窗体用于在 5×5 的范围内选择结构元素。在该窗体内添加 27 个 Button 控件和一个 Label 控件，其属性修改如表 7.6 所示。

表 7.6　　所修改的属性

控　件	属　性	所修改内容
morphologic	Size	200，251

	Text	灰度形态学滤波
	ControlBox	False

续表

控　　件	属　　性	所修改内容
button1	Name	start
	Text	确定
	Location	14，172
button2	Name	close
	Text	退出
	Location	107，172
button3	Name	se1
	Text	
	Size	20，20
	Location	50，50
	FlatStyle	Flat
	BackColor	White
其他 24 个 button 类似 button3	…	…
label1	Text	平坦结构元素
	Location	19，24

为这 27 个 Button 控件添加 Click 事件和 1 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

// 把25 个Button 控件排列成5×5 的矩阵
// 第一个Button 控件se1
private void se1_Click(object sender, EventArgs e)
{
    // 判断以前是否被选中
    if (se[0] == 0)
    {
        // 置位
        se[0] = 1;
        // 按钮控件背景置为黑
        se1.BackColor = Color.Black;
    }
    else
    {
        // 清位
        se[0] = 0;
        // 按钮控件背景置为白
```

```
            se1.BackColor = Color.White;
        }
    }

    …
    //其他se按钮控件的Click事件类似
    …

    public byte[] GetStruction
    {
        get
        {
            // 得到结构元素数组
            return se;
        }
    }
```

其中 se 为 byte 型数组变量，它在构造函数内被初始化，代码如下：

```
se = new byte[25];
// 清零
Array.Clear(se, 0, 25);
```

（3）回到主窗体，编写灰度形态学腐蚀运算、膨胀运算、开运算和闭运算方法，代码如下

```
/************************************************
灰度形态学膨胀运算
grayImage：待处理的图像
se：结构元素，限制在5×5的方阵内
tHeight：图像高度
tWidth：图像宽度
返回处理后的图像
************************************************/
private byte[] grayDelation(byte[] grayImage, byte[] se,int tHeight,int tWidth)
{
    byte[] tempImage = new byte[grayImage.Length];

    // 灰度形态学膨胀运算，求结构元素范围内最大值
    for (int i = 0; i < tHeight; i++)
    {
        for (int j = 0; j < tWidth; j++)
        {
    int[] cou = new int[]{grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth +
        (Math.Abs(j - 2)) % tWidth] * se[0],
    grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + (Math.Abs(j - 1)) % tWidth] * se[1]
    grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + j] * se[2],
    grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + ((j + 1) % tWidth)] * se[3],
    grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + ((j + 2) % tWidth)] * se[4],
    grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + (Math.Abs(j - 2) % tWidth)] * se[5]
    grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + (Math.Abs(j - 1) % tWidth)] * se[6]
    grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + j] * se[7],
    grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + ((j + 1) % tWidth)] * se[8],
    grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + ((j + 2) % tWidth)] * se[9],
    grayImage[i * tWidth + (Math.Abs(j - 2) % tWidth)] * se[10],
    grayImage[i * tWidth + (Math.Abs(j - 1) % tWidth)] * se[11],
    grayImage[i * tWidth + j] * se[12],
    grayImage[i * tWidth + ((j + 1) % tWidth)] * se[13],
```

```
        grayImage[i * tWidth + ((j + 2) % tWidth)] * se[14],
        grayImage[((i + 1) % tHeight) * tWidth + (Math.Abs(j - 2) % tWidth)] * se[15],
        grayImage[((i + 1) % tHeight) * tWidth + (Math.Abs(j - 1) % tWidth)] * se[16],
        grayImage[((i + 1) % tHeight) * tWidth + j] * se[17],
        grayImage[((i + 1) % tHeight) * tWidth + ((j + 1) % tWidth)] * se[18],
        grayImage[((i + 1) % tHeight) * tWidth + ((j + 2) % tWidth)] * se[19],
        grayImage[((i + 2) % tHeight) * tWidth + (Math.Abs(j - 2) % tWidth)] * se[20],
        grayImage[((i + 2) % tHeight) * tWidth + (Math.Abs(j - 1) % tWidth)] * se[21],
        grayImage[((i + 2) % tHeight) * tWidth + j] * se[22],
        grayImage[((i + 2) % tHeight) * tWidth + ((j + 1) % tWidth)] * se[23],
        grayImage[((i + 2) % tHeight) * tWidth + ((j + 2) % tWidth)] * se[24]};
                //求最大值
                int maxim = cou[0];
                for (int k = 1; k < 25; k++)
                {
                    if (cou[k] > maxim)
                    {
                        maxim = cou[k];
                    }
                }
                tempImage[i * tWidth + j] = (byte)maxim;
            }
        }

        return tempImage;
    }

    /**********************************************
    灰度形态学腐蚀运算
    grayImage: 待处理的图像
    se: 结构元素，限制在5×5的方阵内
    tHeight: 图像高度
    tWidth: 图像宽度
    返回处理后的图像
    **********************************************/
    private byte[] grayErode(byte[] grayImage, byte[] se,int tHeight,int tWidth)
    {
        byte[] tempImage = new byte[grayImage.Length];

        // 确定5×5的方阵内，为结构元素的像素，不是则赋值为255
        byte[] tSe = new byte[25];
        tempSe = (byte[])se.Clone();
        for (int k = 0; k < 25; k++)
        {
            if (tSe[k] == 0)
                tSe[k] = 255;
        }

        // 灰度形态学腐蚀运算，求结构元素范围内最小值
        for (int i = 0; i < tHeight; i++)
        {
            for (int j = 0; j < tWidth; j++)
            {
        int[] cou = new int[]{grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth +
            (Math.Abs(j - 2)) % tWidth] * tempSe[0],
```

```
        grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + (Math.Abs(j - 1)) % tWidth] * tS
1],
        grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + j] * tSe [2],
        grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + ((j + 1) % tWidth)] * tSe [3],
        grayImage[((Math.Abs(i - 2)) % tHeight) * tWidth + ((j + 2) % tWidth)] * tSe [4],
        grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + (Math.Abs(j - 2) % tWidth)] * tSe [5]
        grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + (Math.Abs(j - 1) % tWidth)] * tSe [6]
        grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + j] * tSe [7],
        grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + ((j + 1) % tWidth)] * tSe [8],
        grayImage[((Math.Abs(i - 1)) % tHeight) * tWidth + ((j + 2) % tWidth)] * tSe [9],
        grayImage[i * tWidth + (Math.Abs(j - 2) % tWidth)] * tSe [10],
        grayImage[i * tWidth + (Math.Abs(j - 1) % tWidth)] * tSe [11],
        grayImage[i * tWidth + j] * tSe [12],
        grayImage[i * tWidth + ((j + 1) % tWidth)] * tSe [13],
        grayImage[i * tWidth + ((j + 2) % tWidth)] * tSe [14],
        grayImage[((i + 1) % tHeight) * tWidth + (Math.Abs(j - 2) % tWidth)] * tSe [15],
        grayImage[((i + 1) % tHeight) * tWidth + (Math.Abs(j - 1) % tWidth)] * tSe [16],
        grayImage[((i + 1) % tHeight) * tWidth + j] * tSe [17],
        grayImage[((i + 1) % tHeight) * tWidth + ((j + 1) % tWidth)] * tSe [18],
        grayImage[((i + 1) % tHeight) * tWidth + ((j + 2) % tWidth)] * tSe [19],
        grayImage[((i + 2) % tHeight) * tWidth + (Math.Abs(j - 2) % tWidth)] * tSe [20],
        grayImage[((i + 2) % tHeight) * tWidth + (Math.Abs(j - 1) % tWidth)] * tSe [21],
        grayImage[((i + 2) % tHeight) * tWidth + j] * tSe [22],
        grayImage[((i + 2) % tHeight) * tWidth + ((j + 1) % tWidth)] * tSe [23],
        grayImage[((i + 2) % tHeight) * tWidth + ((j + 2) % tWidth)] * tSe [24]};
                // 求最小值
                int minimum = cou[0];
                for (int k = 1; k < 25; k++)
                {
                    if (cou[k] < minimum)
                    {
                        minimum = cou[k];
                    }
                }
                tempImage[i * tWidth + j] = (byte)minimum;
            }
        }

        return tempImage;
    }

    /*********************************************
    灰度形态学开运算
    grayImage：待处理的图像
    se：结构元素，限制在5×5的方阵内
    tHeight：图像高度
    tWidth：图像宽度
    返回处理后的图像
    *********************************************/
    private byte[] grayOpen(byte[] grayImage, byte[] se,int tHeight,int tWidth)
    {
        byte[] tempImage = new byte[grayImage.Length];
        // 调用腐蚀运算
        tempImage = grayErode(grayImage, se, tHeight, tWidth);
        // 调用膨胀运算，并返回结果
```

```
    return (grayDelation(tempImage, se, tHeight, tWidth)));
}

/*********************************************
灰度形态学闭运算
grayImage: 待处理的图像
se: 结构元素，限制在5×5 的方阵内
tHeight: 图像高度
tWidth: 图像宽度
返回处理后的图像
**********************************************/
private byte[] grayClose(byte[] grayImage, byte[] se,int tHeight,int tWidth)
{
    byte[] tempImage = new byte[grayImage.Length];
    // 调用膨胀运算
    tempImage = grayDelation(grayImage, se, tHeight, tWidth);
    // 调用腐蚀运算，并返回结果
    return (grayErode(tempImage, se, tHeight, tWidth));
}
```

为“灰度形态学”按钮添加 1 个 Click 事件，代码如下：

```
private void grayMor_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化morphologic
        morphologic grayMor = new morphologic();

        if (grayMor.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                 System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            byte[] tempArray = new byte[bytes];
            byte[] struEle = new byte[25];
            // 得到结构元素
            struEle = grayMor.GetStruction;

            // 调用灰度闭运算
            tempArray = grayClose(grayValues, struEle ,
                curBitmap.Height, curBitmap.Width);
            // 调用灰度开运算
            grayValues = grayOpen(tempArray, struEle ,
                curBitmap.Height, curBitmap.Width);

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            curBitmap.UnlockBits(bmpData);
        }
```

```
            Invalidate();
        }
    }
```

（4）编译并运行该段程序，仍然以图 7.4 所示的受椒盐噪声（含椒量和含盐量都是 0.01
干扰的图像为例进行处理。单击“灰度形态学”按钮，打开灰度形态学滤波对话框，如图 7.
所示设置相应的平坦结构元素。

单击“确定”按钮后，经过该平坦结构元素处理后的图像如图 7.10 所示。

图 7.9　灰度形态学对话框

图 7.10　灰度形态学滤波

需要说明的是，该段程序也同样使用了循环延拓的方法来解决图像四周不能处理的问题。

7.4　小波变换去噪

本书没有把小波理论单独作为一章来介绍，而是把它根据用途进行分类，穿插在各个章节中进行讲解。

7.4.1　小波理论概述

小波分析（wavelet analysis）或小波变换（wavelet transform）是指用有限长或快速衰减的称为母小波（mother wavelet）的震荡波形来表示信号。它与傅里叶变换、窗口傅里叶变换相比，是一个时间和频率的局域变换，因而能有效地从信号中提取信息，通过伸缩和平移等运算功能对函数或信号进行多尺度细化分析（Multiscale Analysis），解决了傅里叶变换不能解决的许多困难问题。小波变换由于其在非平稳图像信号分析方面的灵活性和适应人眼视觉特性的能力，已经成为图像和视频编码的有力工具。

在基于计算机的小波变换中，应用的是离散小波变换。在这里我们应用的是离散小波变换的快速算法。它的正变换算法为：

$$\begin{cases} c_{j,k} = \sum_n h_{n-2k} c_{j-1,n} \\ d_{j,k} = \sum_n g_{n-2k} c_{j-1,n} \end{cases} \tag{7.16}$$

其中 $c_{j,k}$ 是尺度系数，即原信号的平滑信号，$d_{j,k}$ 是小波系数，即细节信号，g_n 是与小波函数相关的高通滤波器的脉冲响应，h_n 是与尺度函数相关的低通滤波器的脉冲响应，它们之间的关系为：$g_n=(-1)^n h_{-n}$。h_n 滤波器的选择也是小波变换的关键问题之一，这里不对它进行研究，只选择目前比较常用的几种即可。下标 j 表示的是级数，即由第（j−1）级的平滑信号生成第 j 级的平滑信号和细节信号。

离散小波变换快速算法的逆变换算法为：

$$c_{j-1,k} = \sum_n h_{k-2n} c_{j,n} + g_{k-2n} d_{j,n} \tag{7.17}$$

上式表示的是从第 j 级的平滑信号和细节信号生成第 j−1 级的平滑信号。图 7.11 为上述两个等式的示意图。

(a)小波正变换　　(b)小波逆变换

图 7.11　小波变换示意图

图像是二维离散信号，与二维离散傅里叶变换一样，二维离散小波变换（分解）首先进行水平方向上的离散小波变换，即把图像的每个行像素作为一维序列信号进行小波变换，结果是把图像分解为两部分，左边的是图像平滑部分，右边的是图像细节部分；然后对其变换结果再进行垂直方向上的小波变换，即把上一步生成的二维信号的每个列信号作为一维序列进行小波变换，结果是把图像分解成了 4 个部分。图 7.12 为小波变换过程图。

在图 7.12 中，d_1^v 表示的是图像中垂直方向上的高频成分，d_1^h 表示的是水平方向上的高频成分，d_1^d 表示的是对角线方向上的高频成分，而 c_1 是图像平滑处理后的低频成分。该图表示的是对图像进行的第一次小波分解，当然可以对 c_1 进行第二次小波分解，以此类推。图 7.13 为由 Haar 小波基（即 h_n）经过两次分解后的图像，该图没有进行灰度拉伸处理，只是简单地进行了阈值处理。

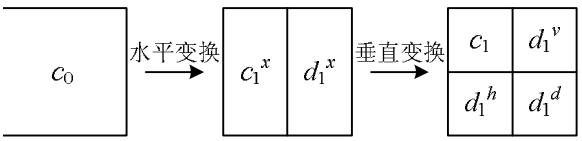


图 7.12　图像的小波分解过程

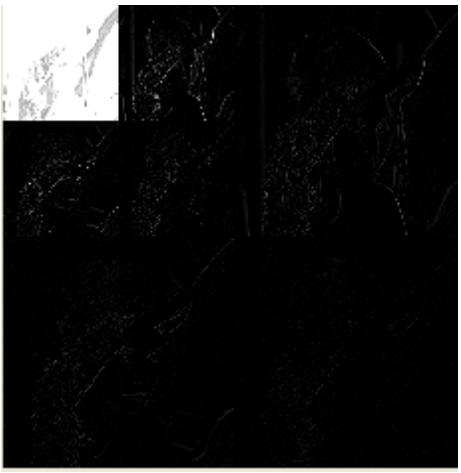


图 7.13　小波两次分解

二维离散小波逆变换（重构）是先进行垂直方向的一维离散小波逆变换，再进行水平方向的一维离散小波逆变换。

7.4.2　小波变换去噪原理

小波变换可以滤除掉图像中的加性噪声。该方法的核心思想是在小波域内，通过设置阈值去除噪声，这是因为小波变换能够分离出信号和噪声。经过小波变换后，信号的能量被压缩到了有很大振幅的一小部分离散小波变换系数当中，而噪声则被扩散到了振幅很小的大部分离散小波变换系数中。因此，再经过阈值处理后，小振幅的系数（噪声）被去掉了，而大振幅的系数（信号）被保留下来。因为引入了阈值处理，所以小波变换去噪方法属于非线性去噪方法。

阈值的选择有两种：软阈值和硬阈值。硬阈值是当小波系数的绝对值小于给定阈值时，令其为 0，而大于阈值时，保持其不变，即：

$$w_l = \begin{cases} w & |w| \geq l \\ 0 & |w| < l \end{cases} \tag{7.18}$$

软阈值是当小波系数的绝对值小于给定阈值时，令其为 0，大于阈值时，令其减去阈值，即

$$w_l = \begin{cases} w - l & w \geq l \\ w + l & w \leq -l \\ 0 & |w| < l \end{cases} \tag{7.19}$$

小波变换去噪的具体步骤为：

（1）对图像进行小波分解；

（2）阈值处理；（3）对图像进行小波重构。

7.4.3　小波变换去噪编程实例

该实例可以对图像进行任意的小波分解及软、硬阈值化，而得到滤波后的图像。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 7.7 所示。

表 7.7　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	wavelet
	Text	小波变换
	Location	37，288

（2）创建 1 个名为 wavelet 的 Windows 窗体，该窗体用于选择小波基、分解级数以及软硬阈值。在该窗体内添加 2 个 Button 控件、1 个 Label 控件、2 个 NumericUpDown 控件、2 个 GroupBox 控件和 8 个 RadioButton 控件，其属性修改如表 7.8 所示。

表 7.8　　所修改的属性

控　　件	属　　性	所修改内容
wavelet	Text	小波变换去噪
	Size	337，427
	ControlBox	False

续表

控　　件	属　　性	所修改内容
button1	Name	start
	Text	确定
	Location	40，345
button2	Name	close
	Text	退出
	Location	215，345
label1	Text	小波变换级数
	Location	63，24
numericUpDown1	Name	waveletSeries
	Size	51，21
	Location	178，22
	Minimum	1
numericUpDown2	Name	thresValue
	Size	51，21
	Location	138，41
	Value	20
	Minimum	3
groupBox1	Location	40，59
	Size	250，95
	Text	阈值
radioButton1	Name	softTh
	Location	25，20
	Text	软阈值
	Checked	True
radioButton2	Name	hardTh
	Location	25，60

	Text	硬阈值
groupBox2	Location	40，177
	Size	250，146
	Text	低通滤波器
radioButton3	Name	haar
	Text	Haar
	Location	25，31
	Checked	True
radioButton4	Name	daubechies2
	Text	Daubechies2
	Location	25，71
radioButton5	Name	daubechies3
	Text	Daubechies3
	Location	25，111

续表

控　　件	属　　性	所修改内容
radioButton6	Name	daubechies4
	Text	Daubechies4
	Location	138，31
radioButton7	Name	daubechies5
	Text	Daubechies5
	Location	138，71
radioButton8	Name	daubechies6
	Text	Daubechies6
	Location	138，111

为 2 个 Button 控件添加 Click 事件，为 8 个 RadioButton 控件添加 CheckedChanged 事件，3 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void softTh_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0x0f);
}

private void hardTh_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0x0f | 0x10);
```

```
}

private void haar_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0xf0);
}

private void daubechies2_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0xf0 | 0x01);
}

private void daubechies3_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0xf0 | 0x02);
}

private void daubechies4_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0xf0 | 0x03);
}

private void daubechies5_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0xf0 | 0x04);
}

private void daubechies6_CheckedChanged(object sender, EventArgs e)
{
    flagV = (byte)(flagV & 0xf0 | 0x05);
}

public byte GetFlagV
{
    get
    {
        // 得到小波基及阈值标识
        return flagV;
    }
}

public byte GetSeries
{
    get
    {
        // 得到小波分解级数
        return (byte)waveletSeries.Value;
    }
}

public byte GetThresholding
{
    get
    {
        // 得到阈值
```

```
            return (byte)thresValue.Value;
        }
    }
```

其中 flagV 都是 byte 型变量，它的前 4 位表示的是选取软阈值还是硬阈值，后 4 位表示的是选取何种小波基。它在构造函数里被初始化，代码如下：

```
flagV = 0x00;
```

（3）回到主窗体，首先为小波变换编写方法，代码如下：

```
/*************************************************
一维离散小波变换，应用公式(7.16)
scl0：小波变换前的尺度系数，即平滑信号
p：与尺度函数相关的低通滤波器
q：与小波函数相关的高通滤波器
scl1：小波变换后的尺度系数
wvl1：小波变换后的小波系数，即细节信号
*************************************************/
private void wavelet1D(double[] scl0, double[] p,double[]q,out double[] scl1,
    out double[] wvl1)
{
    int temp;
    int sclLen = scl0.Length;
    int pLen = p.Length;
    scl1 = new double[sclLen / 2];
    wvl1 = new double[sclLen / 2];

    for (int i = 0; i < sclLen / 2; i++)
    {
        scl1[i] = 0.0;
        wvl1[i] = 0.0;
        for (int j = 0; j < pLen; j++)
        {
            // 循环延拓
            temp = (j + i * 2) % sclLen;
            // 应用公式(7.16)
            scl1[i] += p[j] * scl0[temp];
            wvl1[i] += q[j] * scl0[temp];
        }
    }
}

/*************************************************
一维离散小波逆变换，应用公式(7.17)
scl0：小波逆变换后的尺度系数，即平滑信号
p：与尺度函数相关的低通滤波器
q：与小波函数相关的高通滤波器
scl1：小波逆变换前的尺度系数，即平滑信号
wvl1：小波逆变换前的小波系数，即细节信号
*************************************************/
private void iwavelet1D(out double[] scl0, double[] p,double[]q, double[] scl1, double[] wvl1
{
    int temp;
    int sclLen = scl1.Length;
    int pLen = p.Length;
    scl0 = new double[sclLen * 2];
```

```
    for (int i = 0; i < sclLen; i++)
    {
        scl0[2 * i + 1] = 0.0;
        scl0[2 * i] = 0.0;
        for (int j = 0; j < pLen/2; j++)
        {
            // 循环延拓
            temp = (i - j + sclLen) % sclLen;
            // 公式(7.17)
            scl0[2 * i + 1] += p[2 * j + 1] * scl1[temp] + q[2 * j + 1] * wvl1[temp];
            scl0[2 * i] += p[2 * j] * scl1[temp] + q[2 * j] * wvl1[temp];
        }
    }
}

/*************************************************
二维离散小波变换
dataImage: 二维图像信号
p: 与尺度函数相关的低通滤波器
q: 与小波函数相关的高通滤波器
series: 关于小波变换级数的参数
*************************************************/
private void wavelet2D(ref double[] dataImage, double[] p, double[] q, int series)
{
    double[] s = new double[curBitmap.Width / series];
    double[] s1 = new double[curBitmap.Width / (2 * series)];
    double[] w1 = new double[curBitmap.Width / (2 * series)];

    // 水平方向一维离散小波变换
    for (int i = 0; i < curBitmap.Height / series; i++)
    {
        // 变换前赋值
        for (int j = 0; j < curBitmap.Width / series; j++)
        {
            s[j] = dataImage[i * curBitmap.Width / series + j];
        }

        // 调用一维小波变换
        wavelet1D(s, p,q, out s1, out w1);

        // 变换后赋值
        for (int j = 0; j < curBitmap.Width / series; j++)
        {
            if (j < curBitmap.Width / (2 * series))
                dataImage[i * curBitmap.Width / series + j] = s1[j];
            else
                dataImage[i * curBitmap.Width / series + j] =
                    w1[j - curBitmap.Width / (2 * series)];
        }
    }

    // 垂直方向离散小波变换
    for (int i = 0; i < curBitmap.Width / series; i++)
    {
```

```
            // 变换前赋值
            for (int j = 0; j < curBitmap.Height / series; j++)
            {
                s[j] = dataImage[j * curBitmap.Width / series + i];
            }

            // 调用一维小波变换
            wavelet1D(s, p,q, out s1, out w1);

            // 变换后赋值
            for (int j = 0; j < curBitmap.Height / series; j++)
            {
                if (j < curBitmap.Height / (2 * series))
                    dataImage[j * curBitmap.Width / series + i] = s1[j];
                else
                    dataImage[j * curBitmap.Width / series + i] =
                        w1[j - curBitmap.Height / (2 * series)];
            }
        }
    }

    /*****************************************
    二维离散小波逆变换
    dataImage: 二维图像信号
    p: 与尺度函数相关的低通滤波器
    q: 与小波函数相关的高通滤波器
    series: 关于小波变换级数的参数
    *****************************************/
    private void iwavelet2D(ref double[] dataImage, double[] p, double[] q, int series)
    {
        double[] s = new double[curBitmap.Width / series];
        double[] s1 = new double[curBitmap.Width / (2*series)];
        double[] w1 = new double[curBitmap.Width / (2*series)];

        // 垂直方向离散小波逆变换
        for (int i = 0; i < curBitmap.Width / series; i++)
        {
            // 变换前赋值
            for (int j = 0; j < curBitmap.Height / series; j++)
            {
                if (j < curBitmap.Height / (2*series))
                    s1[j] = dataImage[j * curBitmap.Width / series + i];
                else
                    w1[j - curBitmap.Height / (2 * series)] =
                        dataImage[j * curBitmap.Width / series + i];
            }

            // 调用一维小波逆变换
            iwavelet1D(out s, p,q, s1, w1);

            // 变换后赋值
            for (int j = 0; j < curBitmap.Height / series; j++)
            {
                dataImage[j * curBitmap.Width / series + i] = s[j];
            }
```

```
    }

    // 水平方向离散小波逆变换
    for (int i = 0; i < curBitmap.Height / series; i++)
    {
        // 变换前赋值
        for (int j = 0; j < curBitmap.Width / series; j++)
        {
            if (j < curBitmap.Width / (2*series))
                s1[j] = dataImage[i * curBitmap.Width / series + j];
            else
                w1[j - curBitmap.Width / (2 * series)] =
                    dataImage[i * curBitmap.Width / series + j];
        }

        // 调用一维小波逆变换
        iwavelet1D(out s, p, q,s1, w1);

        // 变换后赋值
        for (int j = 0; j < curBitmap.Width / series; j++)
        {
            dataImage[i * curBitmap.Width / series + j] = s[j];
        }
    }
}
```

然后为“小波变换”按钮添加一个 Click 事件，代码如下：

```
private void wavelet_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化wavelet
        wavelet dwt = new wavelet();

        if (dwt.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            double[] tempA = new double[bytes];
            double[] tempB = new double[bytes];
            for (int i = 0; i < bytes; i++)
            {
                tempA[i] = Convert.ToDouble(grayValues[i]);
            }

            // 得到阈值
            byte thresholding = dwt.GetThresholding;
            // 得到小波分解级数
```

```
byte dwtSeries = dwt.GetSeries;
// 得到应用何种小波基和何种阈值法
int flagFilter = dwt.GetFlagV;

double[] lowFilter = null;
double[] highFilter = null;
// 低通滤波器赋值
switch (flagFilter & 0x0f)
{
    case 0:
        // haar
        lowFilter = new double[] { 0.70710678118655, 0.70710678118655 };
        break;
    case 1:
        // daubechies2
        lowFilter = new double[] { 0.48296291314453, 0.83651630373780,
            0.22414386804201, -0.12940952255126 };
        break;
    case 2:
        // daubechies3
        lowFilter = new double[] { 0.33267055295008, 0.80689150931109,
            0.45987750211849, -0.13501102001025,
            -0.08544127388203, 0.03522629188571 };
        break;
    case 3:
        // daubechies4
        lowFilter = new double[] { 0.23037781330889, 0.71484657055291,
            0.63088076792986, -0.02798376941686,
            -0.18703481171909, 0.03084138183556,
            0.03288301166689, -0.01059740178507 };
        break;
    case 4:
        // daubechies5
        lowFilter = new double[] { 0.16010239797419, 0.60382926979719,
            0.72430852843778, 0.13842814590132,
            -0.24229488706638, -0.03224486958464,
            0.07757149384005, -0.00624149021280,
            -0.01258075199908, 0.00333572528547 };
        break;
    case 5:
        // daubechies6
        lowFilter = new double[] { 0.11154074335011, 0.49462389039845,
            0.75113390802110, 0.31525035170920,
            -0.22626469396544, -0.12976686756726,
            0.09750160558732, 0.02752286553031,
            -0.03158203931849, 0.00055384220116,
            0.00477725751195,    -0.00107730108531 };
        break;
    default:
        MessageBox.Show("无效! ");
        break;
}

// 高通滤波器赋值
highFilter = new double[lowFilter.Length];
```

```
                for (int i = 0; i < lowFilter.Length; i++)
                {
                    highFilter[i] = Math.Pow(-1, i) * lowFilter[lowFilter.Length - 1 - i];
                }

                // 二维离散小波变换
                for (int k = 0; k < dwtSeries; k++)
                {
                    int coef = (int)Math.Pow(2, k);
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        if (i < curBitmap.Height / coef)
                        {
                            for (int j = 0; j < curBitmap.Width; j++)
                            {
                                if (j < curBitmap.Width / coef)
                                {
                                    tempB[i * curBitmap.Width / coef + j] =
                                        tempA[i * curBitmap.Width + j];
                                }
                            }
                        }
                    }

                    // 调用二维小波变换
                    wavelet2D(ref tempB, lowFilter, highFilter, coef);

                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        if (i < curBitmap.Height / coef)
                        {
                            for (int j = 0; j < curBitmap.Width; j++)
                            {
                                if (j < curBitmap.Width / coef)
                                {
                                    tempA[i * curBitmap.Width + j] =
                                        tempB[i * curBitmap.Width / coef + j];
                                }
                            }
                        }
                    }

                    // 阈值处理
                    if ((flagFilter & 0xf0) == 0x10)
                    {
                        // 硬阈值
                        for (int l = 0; l < bytes; l++)
                        {
                            // 公式(7.18)
                            if (tempA[l] < thresholding && tempA[l] > -thresholding)
                                tempA[l] = 0;
                        }
                    }
                    else
                    {
```

```
                // 软阈值
                for (int l = 0; l < bytes; l++)
                {
                    // 公式(7.19)
                    if (tempA[l] >= thresholding)
                        tempA[l] = tempA[l] - thresholding;
                    else
                    {
                        if (tempA[l] <= -thresholding)
                            tempA[l] = tempA[l] + thresholding;
                        else
                            tempA[l] = 0;
                    }
                }
            }
        }

        // 二维离散小波逆变换
        for (int k = dwtSeries - 1; k >= 0; k--)
        {
            int coef = (int)Math.Pow(2, k);
            for (int i = 0; i < curBitmap.Height; i++)
            {
                if (i < curBitmap.Height / coef)
                {
                    for (int j = 0; j < curBitmap.Width; j++)
                    {
                        if (j < curBitmap.Width / coef)
                        {
                            tempB[i * curBitmap.Width / coef + j] =
                                tempA[i * curBitmap.Width + j];
                        }
                    }
                }
            }

            // 调用二维小波逆变换
            iwavelet2D(ref tempB, lowFilter, highFilter, coef);

            for (int i = 0; i < curBitmap.Height; i++)
            {
                if (i < curBitmap.Height / coef)
                {
                    for (int j = 0; j < curBitmap.Width; j++)
                    {
                        if (j < curBitmap.Width / coef)
                        {
                            tempA[i * curBitmap.Width + j] =
                                tempB[i * curBitmap.Width / coef + j];
                        }
                    }
                }
            }
        }
```

```
                for (int i = 0; i < bytes; i++)
                {
                    if (tempA[i] >= 255)
                        tempA[i] = 255;
                    if (tempA[i] <= 0)
                        tempA[i] = 0;
                    grayValues[i] = Convert.ToByte(tempA[i]);
                }

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运行该段程序，这里仍然以如图 7.3 所示的受到高斯噪声（均值为 0，均方差为 15）干扰的图像为例，进行小波变换滤波处理。单击“小波变换”按钮，打开小波变换去噪对话框，如图 7.14 所示设置相关参数。

单击“确定”按钮后，小波变换后的图像就被显示出来，如图 7.15 所示。

图 7.14 小波变换去噪对话框

图 7.15 小波去噪后的图像

从图 7.15 可以看出，虽然小波变换同样能够达到降噪的效果，但它并没有破坏图像的细节部分，因此利用小波分析的理论可以构造一种既能够降低图像噪声，又能保持图像细节信息的方法。

7.5　高斯低通滤波

7.5.1　高斯低通滤波原理

一般来说，噪声都是由高频成分组成的，所以用低通滤波器对图像进行卷积处理，就可以有效地滤除噪声，高斯函数就是一个这样的低通滤波器，它属于线性滤波。高斯函数最主要的特性就是它的傅里叶变换仍然是高斯函数，所以应用快速傅里叶变换可以把空间域内的卷积运算变换为频域内的乘积运算，这样对于半径很大的高斯核来说，大大降低了运算时间，提高了运算速度。这里应用空间域的卷积来处理。

在图像处理中，需要的二维高斯函数，它的定义为：

$$g(x)=e^{-\frac{x^2+y^2}{2s^2}} \tag{7.20}$$

在实际运算中，我们可以把二维高斯函数分解为一维高斯函数，分别对图像的行和列进行卷积，这样的运算效率也得到了提高。

7.5.2　高斯低通滤波编程实例

该实例实现了应用高斯函数在空间域内的去噪功能。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 7.9 所示。

表 7.9 所修改的属性

控　　件	属　　性	所修改内容
button1	Name	gauss
	Text	高斯滤波
	Location	37，334

（2）创建 1 个名为 gauss 的 Windows 窗体，该窗体用于选择高斯函数中的均方差σ。在该窗体内添加 2 个 Button 控件、1 个 Label 控件和 1 个 TextBox 控件，其属性修改如表 7.10 所示。

表 7.10 所修改的属性

控件	属性	所修改内容
gauss	Text	高斯低通滤波
	Size	250，180
	ControlBox	False
button1	Name	start
	Text	确定
	Location	28，90
button2	Name	close
	Text	退出
	Location	131，90
label1	Text	均方差（σ）
	Location	38，36
textBox1	Name	sigmaValue
	Text	2
	Location	133，33
	Size	59，21

在该窗体内，为 2 个 Button 控件添加 Click 事件以及 1 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public double GetSigma
{
    get
    {
        // 得到均方差σ
```

```
            return Convert.ToDouble(sigmaValue.Text);
        }
    }
```

（3）回到主窗体，为“高斯滤波”按钮添加 Click 事件，代码如下：

```
    private void gauss_Click(object sender, EventArgs e)
    {
        if (curBitmap != null)
        {
            // 实例化gauss
            gauss gaussFilter = new gauss();

            if (gaussFilter.ShowDialog() == DialogResult.OK)
            {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] grayValues = new byte[bytes];
                System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

                double[] tempArray;
                double[] tempImage = new double[bytes];
                // 得到均方差σ
                double sigma = gaussFilter.GetSigma;
                for (int i = 0; i < bytes; i++)
                    tempImage[i] = Convert.ToDouble(grayValues[i]);

                // 调用高斯滤波方法
                gaussSmooth(tempImage, out tempArray, sigma);

                for (int i = 0; i < bytes; i++)
                {
                    if (tempArray[i] > 255)
                        grayValues[i] = 255;
                    else if (tempArray[i] < 0)
                        grayValues[i] = 0;
                    else
                        grayValues[i] = Convert.ToByte(tempArray[i]);
                }

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }

    /**********************************
    二维高斯卷积
    inputImage：输入图像
    outputImage：输出图像
```

```
    sigma: 均方差
    **********************************/
    private void gaussSmooth(double[] inputImage, out double[] outputImage, double sigma)
    {
        // 方差
        double std2 = 2 * sigma * sigma;
        // 半径=3σ
        int radius = Convert.ToInt16(Math.Ceiling(3 * sigma));
        int filterWidth = 2 * radius + 1;
        double[] filter = new double[filterWidth];
        outputImage = new double[inputImage.Length];
        // 限定输入的图像为方阵的情况下得到图像的宽度或高度
        int length = Convert.ToInt16(Math.Sqrt(inputImage.Length));
        double[] tempImage = new double[inputImage.Length];

        double sum = 0;
        // 产生一维高斯函数
        for (int i = 0; i < filterWidth; i++)
        {
            int xx = (i - radius) * (i - radius);
            filter[i] = Math.Exp(-xx / std2);
            sum += filter[i];
        }
        // 归一化
        for (int i = 0; i < filterWidth; i++)
        {
            filter[i] = filter[i] / sum;
        }

        // 水平方向滤波
        for (int i = 0; i < length; i++)
        {
            for (int j = 0; j < length; j++)
            {
                double temp = 0;
                for (int k = -radius; k <= radius; k++)
                {
                    // 循环延拓
                    int rem = (Math.Abs(j + k)) % length;
                    // 计算卷积和
                    temp += inputImage[i * length + rem] * filter[k + radius];
                }
                tempImage[i * length + j] = temp;
            }
        }
        // 垂直方向滤波
        for (int j = 0; j < length; j++)
        {
            for (int i = 0; i < length; i++)
            {
                double temp = 0;
                for (int k = -radius; k <= radius; k++)
                {
                    // 循环延拓
                    int rem = (Math.Abs(i + k)) % length;
```

```
                    // 计算卷积和
                    temp += tempImage[rem * length + j] * filter[k + radius];
                }
                outputImage[i * length + j] = temp;
            }
        }
    }
```

（4）编译并运行该段程序，我们仍然以图 7.3 所示的受到高斯噪声（均值为 0，均方差为 15）干扰的图像为例，进行高斯滤波处理。单击“高斯滤波”按钮，打开高斯低通滤波对话框，如图 7.16 所示设置均方差值。

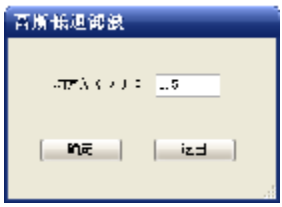


图 7.16　高斯低通滤波对话框

单击“确定”按钮后，高斯滤波后的图像就显示出来，如图 7.17 所示。

图 7.17　高斯滤波后的图像

7.6　统计滤波

7.6.1　统计滤波原理

统计平滑滤波的方法就是利用均值和方差的统计性质来滤波去噪的。它的基本方法是，首先在给定像素 $x(m, n)$的邻域窗 W 内，求出所有像素的均值μ和方差σ^2：

$$m = \frac{1}{N}\sum_{(i,j)\in W} x(i, j) \tag{7.21}$$

$$s^2 = \frac{1}{N}\sum_{(i,j)\in W} [x(i, j) - m]^2 \tag{7.22}$$

其中 N 为邻域窗 W 内的像素个数。在这里，我们只给出 3×3 和 5×5 两种方形邻域窗。

然后，通过给定的阈值 T 求出对应像素 $y(m, n)$：

$$y(m.n)=\begin{cases} x(m,n) & |x(m,n)-m|<sT \\ m & \text{其他} \end{cases} \tag{7.23}$$

7.6.2 统计滤波编程实例

该实例实现了统计方法的滤波。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 7.11 所示。

表 7.11 所修改的属性

控　件	属　性	所修改内容
button1	Name	statistic
	Text	统计方法
	Location	37，380

（2）创建 1 个名为 static 的 Windows 窗体，该窗体用于选择邻域窗的大小和阈值。为该窗体添加 2 个 Button 控件、1 个 GroupBox 控件、两个 RadioButton 控件、1 个 Label 控件和 1 个 TextBox 控件，其属性修改如表 7.12 所示。

表 7.12 所修改的属性

控　件	属　性	所修改内容
static	Text	统计平滑方法
	ControlBox	False
	Size	260，250
button1	Name	start
	Text	确定
	Location	22，167
button2	Name	close
	Text	退出
	Location	147，167
groupBox	Text	邻域窗
	Location	22，26
	Size	200，77
radioButton1	Name	three
	Text	3×3
	Checked	True
	Location	28，33
radioButton2	Name	five
	Text	5×5
	Location	125，33
label1	Text	阈值（T）：
	Location	48，126
textBox1	Name	thresh
	Text	1.5

	Location	139，123
	Size	55，21

为 2 个 Button 控件添加 Click 事件，并添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetWindows
{
    get
    {
        // 得到窗体大小
        return five.Checked;
    }
}

public double GetThresholding
{
    get
    {
        // 得到阈值
        return Convert.ToDouble(thresh.Text);
    }
}
```

（3）回到主窗体，为“统计方法”按钮控件添加 Click 事件，代码如下：

```
private void statistic_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化stati
        stati staticSmooth = new stati();

        if (staticSmooth.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            byte[] tempArray = new byte[bytes];
            // 得到阈值
            double thresholding = staticSmooth.GetThresholding;
            // 得到窗体大小
```

```
            bool flag = staticSmooth.GetWindows;

            if (flag == false)
            {
                // 3×3 邻域窗
                for (int i = 0; i < curBitmap.Height; i++)
                {
                    for (int j = 0; j < curBitmap.Width; j++)
                    {
                        double mu = 0, sigma = 0;
    // 计算公式(7.21)
    mu = (grayValues[i * curBitmap.Width + j] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)]) / 9;

    // 计算公式(7.22)
    sigma = Math.Sqrt((Math.Pow((grayValues[i * curBitmap.Width + j] - mu), 2) +
        Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
            j] - mu), 2) +
        Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width+ j] - mu),2)
        Math.Pow((grayValues[i * curBitmap.Width + ((j+ 1) % curBitmap.Width)] - mu),2)
        Math.Pow((grayValues[i * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] - mu), 2) +
        Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
            ((Math.Abs(j - 1)) % curBitmap.Width)] - mu), 2) +
        Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] - mu), 2) +
        Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
            ((j + 1) % curBitmap.Width)] - mu), 2) +
        Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] - mu), 2)) / 9);

                        // 计算公式(7.23)
                        if (Math.Abs(grayValues[i * curBitmap.Width + j] - mu) <
                                sigma * thresholding)
                            tempArray[i * curBitmap.Width + j] =
                                grayValues[i * curBitmap.Width + j];
                        else
                            tempArray[i * curBitmap.Width + j] = Convert.ToByte(mu);

                    }
                }
            }
            else
            {
```

```
                // 5×5 邻域窗
                for (int i = 0; i < curBitmap.Height; i++)
                {
                    for (int j = 0; j < curBitmap.Width; j++)
                    {
                        double mu = 0, sigma = 0;
    // 计算公式(7.21)
    mu = (grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2)) % curBitmap.Width] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1)) % curBitmap.Width] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + j] +
        grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + ((j + 2) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 2) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            (Math.Abs(j - 1) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 2) % curBitmap.Width)]) / 25;

    // 计算公式(7.22)
    sigma = Math.Sqrt((Math.Pow((grayValues[((Math.Abs(i - 2)) % curBitmap.Height) *
            curBitmap.Width + (Math.Abs(j - 2)) % curBitmap.Width] - mu), 2) +
        Math.Pow((grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width
            (Math.Abs(j - 1)) % curBitmap.Width] - mu), 2) +
        Math.Pow((grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j
```

```
mu), 2) +
            Math.Pow((grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width
                ((j + 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width
                ((j + 2) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                (Math.Abs(j - 2) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                (Math.Abs(j - 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                j] - mu), 2) +
            Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                ((j + 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                ((j + 2) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[i * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[i * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[i * curBitmap.Width + j] - mu), 2) +
            Math.Pow((grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] - mu),2)
            Math.Pow((grayValues[i * curBitmap.Width + ((j + 2) % curBitmap.Width)] - mu),2)
            Math.Pow((grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +j] - mu),2)
            Math.Pow((grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -mu),2)
            Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] - mu), 2) +
            Math.Pow((grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] - mu), 2)) / 25);

                        // 计算公式(7.23)
                        if (Math.Abs(grayValues[i * curBitmap.Width + j] - mu) <
                                sigma * thresholding)
                            tempArray[i * curBitmap.Width + j] =
                                grayValues[i * curBitmap.Width + j];
                        else
                            tempArray[i * curBitmap.Width + j] = Convert.ToByte(mu);

                    }
                }
            }
            grayValues = (byte[])tempArray.Clone();

            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
```

```
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运行该段程序，我们仍然以图 7.3 所示的受到高斯噪声（均值为 0，均方差为 15）干扰的图像为例，进行统计方法平滑滤波。单击“统计方法”按钮，打开统计平滑方法对话框，如图 7.18 所示设置相应参数。

单击“确定”按钮后，统计方法滤波后的图像就显示出来，如图 7.19 所示。

图 7.18　统计平滑对话框

图 7.19　统计方法滤波后的图像

需要说明的是，这段程序也应用了循环延拓的方法。

7.7　小结

本章介绍了图像噪声模型以及几种平滑处理的滤波去噪方法。通过本章内容的介绍，读者可以掌握平滑处理的方法，又可以了解灰度形态学和小波变换，尤其是后者。在熟悉了灰度形态学和小波变换的理论后，就可以把它们应用到其他图像处理领域中，这在本书的余下章节中将会充分地体现出来。

第 9 章　图像分割

在对图像的研究和应用中，人们往往仅对图像中的某些部分感兴趣，这些部分一般称为目标或前景。为了辨识和分析目标，需要将有关区域分离提取出来，在此基础上对目标进一步利用，如进行特征提取和测量。图像分割就是指把图像分成各具特性的区域并提取出感兴趣目标的技术和过程。

边缘检测就是图像分割的一部分，已在第 8 章进行了详细地介绍，本章介绍图像分割的其他算法。

9.1　Hough 变换

Hough 变换是图像处理中从图像中识别几何形状的基本方法之一。Hough 变换的基本原理在于利用点与线的对偶性，将原始图像空间中给定的曲线通过曲线表达形式变为参数空间中的一个点。这样就把原始图像中给定曲线的检测问题转化为寻找参数空间中的峰值问题，也即把检测整体特性转化为检测局部特性，比如直线、椭圆、圆弧线等。

9.1.1　Hough 变换原理

在图像空间 X-Y 中的一点（x, y），经过点-正弦曲线对偶的 Hough 变换：

$$\rho = x\cos\theta + y\sin\theta \tag{9.1}$$

在参数空间θ-ρ中变为一条正弦曲线，其中θ取（0～180°）。可以证明，图像空间 X-Y 中直线上的点经过 Hough 变换后，它们的正弦曲线在参数空间θ-ρ中有一个公共交点。也就是说，参数空间θ-ρ中的一点（θ, ρ），对应于图像空间 X-Y 中的一条直线，而且它们是一一对应的。

因此，为了检测出图像空间 X-Y 中由点所构成的直线，可以将参数空间θ-ρ量化成许多小格。根据图像空间 X-Y 中每个点的坐标（x, y），在θ=0～180°内以小格的步长计算各个ρ值，所得值落在某个小格内，便使该小格的累加计数器加 1。当图像空间中全部的点都变换后，对小格进行检验，计数值最大的小格，其（θ, ρ）值对应于图像空间中所求直线。

参数空间中的θ、ρ与图像空间中直线的斜率 k 和截距 b 的关系为：

$$k = -\mathrm{ctg}(\theta),\quad b = \rho / \sin(\theta) \tag{9.2}$$

9.1.2 Hough 变换编程实例

该实例是对有各种图形的二值图像进行 Hough 变换，在新生成的窗体内绘出 Hough 变换映射图像，并对该映射图像再进行 Hough 反变换，就可得到原图像中的直线图形及其它们的位置。需要说明的是，虽然本节应用的是二值图像，但为了使程序保持一致性，所应用的图像仍然是 8 位灰度图像。

（1）创建 1 个“模板主窗体”，并在该主窗体内添加 1 个 Button 控件，其属性修改如表 9.1 所示。

表 9.1 所修改的属性

控　　件	属　　性	所修改内容
button1	Name	hough
	Text	Hough 变换
	Location	37，150

（2）创建 1 个名为 hough 的 Windows 窗体，该窗体用于绘制 Hough 变换映射图像和反 Hough 变换图像。在该窗体内添加 1 个 Button 控件，其属性修改如表 9.2 所示。

表 9.2 所修改的属性

控　　件	属　　性	所修改内容
hough	Size	470，330
	Text	Hough 变换
	ControlBox	False
button1	Name	close
	Text	关闭
	Location	345，250

在该窗体内定义以下变量：

```
private System.Drawing.Bitmap bmpHough;
// Hough 变换累加器
private int[,] houghMap;
// Hough 变换映累加器最大值
private int maxHough;
```

改写其构造函数为：

```
public hough(Bitmap bmp)
{
    InitializeComponent();
    bmpHough = bmp;
    maxHough = 0;
    // 累加器矩阵大小为180×180
    houghMap = new int[180, 180];
}
```

在该窗体内为“关闭”按钮添加 Click 事件，为窗体添加 Load 事件和 Paint 事件，代码如下：

```
private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

// 得到Hough 变换累加器数据
private void hough_Load(object sender, EventArgs e)
{
    Rectangle rect = new Rectangle(0, 0, bmpHough.Width, bmpHough.Height);
    System.Drawing.Imaging.BitmapData bmpData = bmpHough.LockBits(rect,
        System.Drawing.Imaging.ImageLockMode.ReadWrite, bmpHough.PixelFormat);
    IntPtr ptr = bmpData.Scan0;
    int bytes = bmpHough.Width * bmpHough.Height;
    byte[] grayValues = new byte[bytes];
    System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
    bmpHough.UnlockBits(bmpData);

    // 累加器数组清零
    Array.Clear(houghMap, 0, 32400);

    for (int i = 0; i < bmpHough.Height - 1; i++)
    {
        for (int j = 0; j < bmpHough.Width - 1; j++)
        {
            // 判断白色像素点
            if (grayValues[i * bmpHough.Width + j] == 255)
            {
                // θ取0 ~ 180°，与其范围（0~180）一一对应
                //ρ取-512 × √2 ~ 512 × √2，通过欠采样，使其范围也在0~180
                // 角度分辨率为1°
                for (int thet = 0; thet < 180; thet++)
                {
                    double arc = thet * Math.PI / 180;
                    // 公式(9.1)
                   int rho = Convert.ToInt16((j * Math.Cos(arc) + i * Math.Sin(arc)) / 8
                        + 90;
                    // 计数器加1
                    houghMap[thet, rho]++;
                    // 计算最大累计器值
                    if (maxHough < houghMap[thet, rho])
                        maxHough = houghMap[thet, rho];
                }
            }
        }
    }
}

// 绘制图像
private void hough_Paint(object sender, PaintEventArgs e)
{
    // 定义Hough 变换映射图像
```

```
// 定义位图
Bitmap houghImage = new Bitmap(180, 180,
    System.Drawing.Imaging.PixelFormat.Format8bppIndexed);
// 定义图像调色板，使其为8 为灰度图像
System.Drawing.Imaging.ColorPalette cp = houghImage.Palette;
for (int i = 0; i < 256; i++)
    cp.Entries[i] = Color.FromArgb(i, i, i);
houghImage.Palette = cp;

Rectangle rectHough = new Rectangle(0, 0, 180, 180);
System.Drawing.Imaging.BitmapData houghData = houghImage.LockBits(rectHough,
    System.Drawing.Imaging.ImageLockMode.ReadWrite, houghImage.PixelFormat);
IntPtr ptr = houghData.Scan0;
int bytes = 180 * 180;
byte[] grayHough = new byte[bytes];

// 灰度级拉伸
for (int i = 0; i < 180; i++)
{
    for (int j = 0; j < 180; j++)
    {
        grayHough[i * 180 + j] = Convert.ToByte(houghMap[i, j] * 255 / maxHough);
    }
}

System.Runtime.InteropServices.Marshal.Copy(grayHough, 0, ptr, bytes);
houghImage.UnlockBits(houghData);

// 实例化Graphics
Graphics g = e.Graphics;
// 绘制Hough 变换映射图像
g.DrawImage(houghImage, 40, 20, 180, 180);

// 绘制Hough 反变换图像，大小也为180×180
g.FillRectangle(Brushes.Black, 250, 20, 180, 180);

// 定义阈值大小，用于检测直线
// 阈值的大小决定了被检测到的直线的长度
double thresholding = maxHough * 0.6;
double k, b;
int x1, x2, y1, y2;

for (int i = 0; i < 180; i++)
{
    for (int j = 0; j < 180; j++)
    {
        if (houghMap[i, j] > thresholding)
        {
            if (i == 90)
                g.DrawLine(Pens.White, 250, 200, 250, 20);
            else if (i == 0)
                g.DrawLine(Pens.White, 250, 430, 250, 20);
            else
            {
```

```
                    // Hough 反变换
                    k = (-1 / Math.Tan(i * Math.PI / 180));
                    b = ((j - 90) * 8 / Math.Sin(i * Math.PI / 180)) *180 / 512;

                    // 在180×180 范围内画直线
                    // 确定直线的两个端点
                    if (b >= 0 && b <= 180)
                    {
                        x1 = 0;
                        y1 = Convert.ToInt16(b);
                        double temp = -b / k;
                        if (temp <= 180 && temp >= 0)
                        {
                            x2 = Convert.ToInt16(temp);
                            y2 = 0;
                        }
                        else if (temp >= -180 && temp < 0)
                        {
                            x2 = Convert.ToInt16((180 - b) / k);
                            y2 = 180;
                        }
                        else
                        {
                            x2 = 180;
                            y2 = Convert.ToInt16(180 * k + b);
                        }
                    }
                    else if (b < 0)
                    {
                        x1 = Convert.ToInt16(-b / k);
                        y1 = 0;
                        double temp = k * 180 + b;
                        if (temp >= 0 && temp <= 180)
                        {
                            x2 = 180;
                            y2 = Convert.ToInt16(temp);
                        }
                        else
                        {
                            x2 = Convert.ToInt16((180 - b) / k);
                            y2 = 180;
                        }
                    }
                    else
                    {
                        x1 = Convert.ToInt16((180 - b) / k);
                        y1 = 180;
                        double temp = k * 180 + b;
                        if (temp >= 0 && temp <= 180)
                        {
                            x2 = 180;
                            y2 = Convert.ToInt16(temp);
                        }
                        else
                        {
```

```
                            x2 = Convert.ToInt16(-b / k);
                            y2 = 0;
                        }
                    }

                    g.DrawLine(Pens.White, x1 + 250, y1 + 20, x2 + 250, y2 + 20);
                }
            }
        }
    }
    // 标注图注
    g.DrawString("Hough 变换映射图像", new Font("Arial", 8),     Brushes.Black, 80, 210);
    g.DrawString("Hough 反变换图像", new Font("Arial", 8), Brushes.Black, 290, 210);
}
```

（3）回到主窗体，为“Hough 变换”按钮添加 Click 事件，代码如下：

```
private void hough_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化hough，并把位图数据传递给从窗体
        hough houghtran = new hough(curBitmap);
        // 显示从窗体
        houghtran.ShowDialog();
    }
}
```

（4）编译并运行该段程序。首先打开图像，如图 9.1 所示，对该图像进行 Hough 变换。

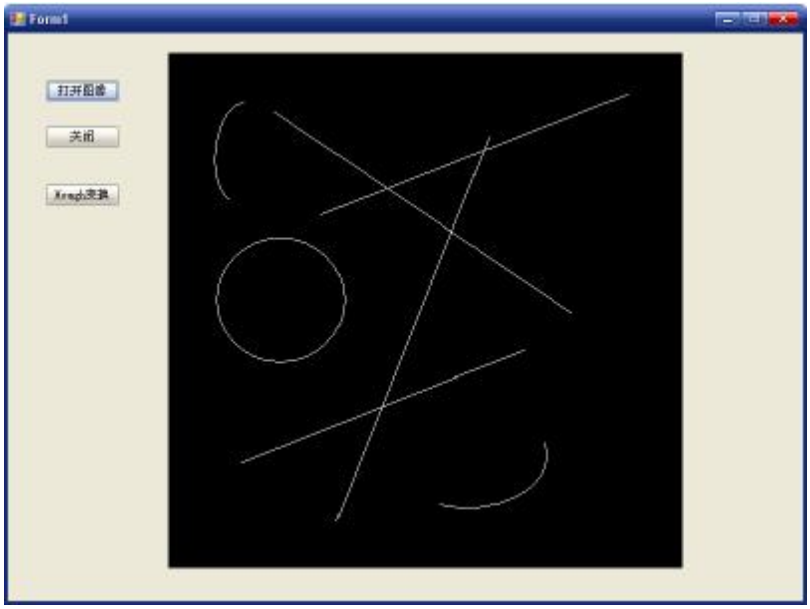


图 9.1 Hough 变换原图

单击“Hough 变换”按钮，则 Hough 变换结果在新打开的窗体内显示出来，如图 9.2 所示。左边为经过灰度级拉伸后的 Hough 变换映射图像，最亮的 4 个点代表原图中的 4 条直线；右边为根据左边图像进行 Hough 反变换后得到的图像。通过与原图比较可以看出，4 条直线不仅能够检测出来，而且它们的位置也准确无误地呈现出来。

图 9.2　Hough 变换结果

需要说明的是，在进行 Hough 反变换时，阈值选取的大小也是检测出直线的关键所在。

9.2　阈值法

阈值分割法是一种基于区域的图像分割技术，其基本原理是：通过设定不同的特征阈值，把图像像素点分为若干类。设原始图像为 f （x, y），按照一定的准则在该图像中找到特征值 T，将图像分割为两个部分，分割后的图像为：

$$g(x,y)=\begin{cases} b_0 & f(x,y)<T \\ b_1 & f(x,y)\geq T \end{cases} \tag{9.3}$$

若取 b_0 为黑，b_1 为白，即为图像的二值化。

阈值分割法是一种最常用也是最简单的图像分割方法，它特别适用于目标和背景占据不同灰度级范围的图像。阈值分割法一般分为人工选择法和自动选择法。人工法就是结合直方图人为地选择阈值。而本节主要介绍自动阈值选择法，其中包括：迭代法、Otsu 法、一维最大熵法、二维最大熵法、简单统计法。

9.2.1　自动阈值选择法原理

1．迭代法

迭代法的基本思想是：开始时选择一个阈值作为初始估计值，然后按照某种策略通过迭代不断地改变这一估计值，直到满足给定的准则为止，其具体步骤如下。

（1）在一幅灰度范围为[0，L-1]的图像中，选择图像灰度的中值作为初始阈值 T_0，其中图像中对应灰度级 i 的像素数为 n_i。

（2）利用阈值 T 把图像分割成两个区域：R_1 和 R_2，用下式计算区域 R_1 和 R_2 的平均灰度 μ_1 和 μ_2。

$$m_1=\frac{\sum_{i=0}^{T_i} in_i}{\sum_{i=0}^{T_i} n_i},\quad m_2=\frac{\sum_{i=T_i}^{L-1} in_i}{\sum_{i=T_i}^{L-1} n_i} \tag{9.4}$$

（3）计算出 μ_1 和 μ_2 后，用下式计算出新的阈值 T_{i+1}。

$$T_{i+1}=\frac{1}{2}(m_1+m_2) \tag{9.5}$$

（4）重复步骤⑵～⑶，直到 T_{i+1} 和 T_i 的差小于某个给定值为止。

2．Otsu 法

Otsu 法是一种使类间方差最大的阈值确定方法，所以也称为最大类间方差法。该方法具有简单、处理速度快等特点，是一种常用的阈值选取方法。其基本思想是：把图像中的像素按灰度值用阈值 T 分成两类 C_1 和 C_2，C_1 由灰度值在[0，T]之间的像素组成，C_2 由灰度值在[T+1，L-1]之间的像素组成，按下式计算两类之间的类间方差：

$$s(t)^2=w_1(t)w_2(t)[m_1(t)-m_2(t)]^2 \tag{9.6}$$

式中，$\omega_1(t)$为 C_1 中包含的像素数，$\omega_2(t)$为 C_2 中包含的像素数，$\mu_1(t)$为 C_1 中所有像素的平均灰度值，$\mu_2(t)$为 C_2 中所有像素的平均灰度值。让 T 在[0，L-1]范围依次取值，使 σ^2 最大的 T 值即为 Otsu 法的最佳阈值。

3．一维最大熵法

熵是平均信息量的表征。把信息熵的概念应用于图像阈值分割的基本思想是：利用图像的灰度分布密度函数定义图像的信息熵，根据假设的不同或视角的不同提出不同的熵准则，最后通过优化该准则得到阈值。其中一维最大熵求阈值的方法如下。

在一幅灰度范围为[0，L-1]的图像中，熵函数定义为：

$$j(t)=\lg p_t(1-p_t)+\frac{H_t}{p_t}+\frac{H_{L-1}-H_t}{1-p_t} \tag{9.7}$$

式中：

$$p_t=\sum_{i=0}^{t} p_i,\quad H_t=-\sum_{i=0}^{t} p_i\lg p_i,\quad H_{L-1}=-\sum_{i=0}^{L-1} p_i\lg p_i$$

p_i 为灰度级 i 出现的概率。

当熵函数取得最大值时，对应的灰度值 T 就是所求的最佳阈值。

4．二维最大熵法

由于灰度一维最大熵是基于图像原始直方图的，它仅仅利用了点灰度信息而未充分利用图像的空间信息，而二维最大熵综合利用了点灰度特征和区域灰度特征，从而较好地表征了图像的信息。它的基本方法是：以原始灰度图像（L 个灰度级）中各像素及其 4 邻域的 4 个

像素为一个区域，计算出区域灰度均值图像，这样原始图像中的每一个像素都对应于一个点灰度-区域灰度均值对。设 $n_{i,j}$ 为图像中点灰度为 i 及其区域灰度均值为 j 的像素点数，$p_{i,j}$ 为点灰度-区域灰度均值对(i, j)发生的概率，则：

$$p_{i,j}=\frac{n_{i,j}}{N\times N} \tag{9.8}$$

其中 N 为图像的大小。

则二维最大熵的判别函数为：

$$f(s,t)=\lg[P_A(1-P_A)]+H_A/P_A+(H_L-H_A)/(1-P_A) \tag{9.9}$$

使 $\phi(s, t)$为最大的阈值 s 和 t 即为所求阈值。其中：

$$P_A=\sum_i\sum_j p_{i,j} \qquad i=0,1,\mathbf{L},s;\ \ j=0,1,\mathbf{L}\,t$$

$$H_A=-\sum_i\sum_j p_{i,j}\lg p_{i,j} \qquad i=0,1,\mathbf{L},s;\ \ j=0,1,\mathbf{L}\,t$$

$$H_L=-\sum_i\sum_j p_{i,j}\lg p_{i,j} \qquad i=0,1,\mathbf{L},L-1;\ \ j=0,1,\mathbf{L}\,L-1$$

5．简单统计法

简单统计法是一种基于简单的图像统计的阈值选取方法。使用该方法，能直接计算一幅图像为 $f(x, y)$的阈值。该方法的计算公式为：

$$T=\frac{\sum_x\sum_y e(x,y)f(x,y)}{\sum_x\sum_y e(x,y)} \tag{9.10}$$

其中：

$$e(x,y)=\max\{|e_x|,|e_y|\}$$

$$e_x=f(x-1,y)-f(x+1,y)$$

$$e_y=f(x,y-1)-f(x,y+1)$$

9.2.2　阈值分割法编程实例

该实例实现了上述 5 种阈值分割方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 9.3 所示。

表 9.3　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	threshold
	Text	阈值法

	Location	37，196

（2）创建 1 个名为 thresholding 的 Windows 窗体，该窗体用于选择上述 5 种阈值分割方法。在该窗体内添加 2 个 Button 控件、1 个 GroupBox 控件和 5 个 RadioButton 控件，其属性修改如表 9.4 所示。

表 9.4　　所修改的属性

控　　件	属　　性	所修改内容
thresholding	Size	247，456
	Text	阈值分割法
	ControlBox	False
button1	Name	start
	Text	确定
	Location	38，363
button2	Name	close
	Text	退出
	Location	127，363
groupBox1	Text	阈值法
	Location	38，28
radioButton1	Name	iteration
	Text	迭代法
	Location	19，34
	Checked	True
radioButton2	Name	otsu
	Text	Otsu 法
	Location	19，82
radioButton3	Name	entropy1D
	Text	一维最大熵法
	Location	19，132
radioButton4	Name	entropy2D
	Text	二维最大熵法
	Location	19，189
radioButton5	Name	statis
	Text	简单统计法
	Location	19，248

分别为 2 个 Button 控件添加 Click 事件，为 5 个 RadioButton 控件添加 CheckedChanged 事件，并再添加 1 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}
```

```
private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void iteration_CheckedChanged(object sender, EventArgs e)
{
    thr = 0;
}

private void otsu_CheckedChanged(object sender, EventArgs e)
{
    thr = 1;
}

private void entropy1D_CheckedChanged(object sender, EventArgs e)
{
    thr = 2;
}

private void entropy2D_CheckedChanged(object sender, EventArgs e)
{
    thr = 3;
}

private void statis_CheckedChanged(object sender, EventArgs e)
{
    thr = 4;
}

public byte GetMethod
{
    get
    {
        // 得到阈值分割方法
        return thr;
    }
}
```

其中 thr 是 byte 型变量，它在构造函数内被初始化为，代码如下：

```
thr = 0;
```

（3）回到主窗体，为“阈值法”按钮添加 Click 事件，代码如下：

```
private void threshold_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化thresholding
        thresholding thrMethod = new thresholding();

        if (thrMethod.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
```

```
        System.Drawing.Imaging.ImageLockMode.ReadWrite,
        curBitmap.PixelFormat);
    IntPtr ptr = bmpData.Scan0;
    int bytes = curBitmap.Width * curBitmap.Height;
    byte[] grayValues = new byte[bytes];
    System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

    // 得到阈值方法
    byte method = thrMethod.GetMethod;

    byte T = 0, S = 0;
    byte[] neighb = new byte[bytes];
    byte temp = 0;
    byte maxGray = 0;
    byte minGray = 255;
    int[] countPixel = new int[256];

    // 计算直方图
    for (int i = 0; i < grayValues.Length; i++)
    {
        temp = grayValues[i];
        countPixel[temp]++;
        if (temp > maxGray)
        {
            // 最大灰度等级
            maxGray = temp;
        }
        if (temp < minGray)
        {
            // 最小灰度等级
            minGray = temp;
        }
    }
    double mu1, mu2;
    int numerator, denominator;
    double sigma;
    double tempMax = 0;
    switch (method)
    {
        case 0:
            // 迭代法
            byte oldT;
            // 初始阈值
            T = oldT = Convert.ToByte((maxGray + minGray) / 2);
            // 迭代过程
            do
            {
                oldT = T;
                numerator = denominator = 0;

                // 公式(9.4)
                for (int i = minGray; i < T; i++)
                {
                    numerator += i * countPixel[i];
                    denominator += countPixel[i];
```

```
                }
                mu1 = numerator / denominator;

                numerator = denominator = 0;
                for (int i = T; i <= maxGray; i++)
                {
                    numerator += i * countPixel[i];
                    denominator += countPixel[i];
                }
                mu2 = numerator / denominator;

                // 公式(9.5)
                T = Convert.ToByte((mu1 + mu2) / 2);
            }
            while (T != oldT);
            break;
        case 1:
            // Otsu 法
            double w1 = 0, w2 = 0;
            double sum = 0;
            numerator = 0;
            for (int i = minGray; i <= maxGray; i++)
            {
                sum += i * countPixel[i];
            }
            for (int i = minGray; i < maxGray; i++)
            {
                w1 += countPixel[i];
                numerator += i * countPixel[i];
                mu1 = numerator / w1;
                w2 = grayValues.Length - w1;
                mu2 = (sum - numerator) / w2;

                // 公式(9.6)
                sigma = w1 * w2 * (mu1 - mu2) * (mu1 - mu2);

                if (sigma > tempMax)
                {
                    tempMax = sigma;
                    T = Convert.ToByte(i);
                }
            }
            break;
        case 2:
            // 一维最大熵法
            double Ht = 0.0, Hl = 0.0, p = 0.0, pt = 0.0;
            for (int i = minGray; i <= maxGray; i++)
            {
                p = (double)countPixel[i] / grayValues.Length;
                if (p < 0.00000000000000001)
                    continue;
                Hl += -p * Math.Log10(p);
            }
            for (int i = minGray; i <= maxGray; i++)
            {
```

```
                p = (double)countPixel[i] / grayValues.Length;
                pt += p;
                if (p < 0.00000000000000001)
                    continue;
                Ht += -p * Math.Log10(p);
                // 公式(9.7)
                sigma = Math.Log10(pt * (1 - pt)) + Ht / pt + (Hl - Ht) / (1 - pt)
                if (sigma > tempMax)
                {
                    tempMax = sigma;
                    T = Convert.ToByte(i);
                }
            }
            break;
        case 3:
            // 二维最大熵
            double[,] pap = new double[256, 256];
            double Hl2D = 0.0, Pa = 0.0, Ha = 0.0;
            // 公式(9.8)
            for (int i = 0; i < bytes; i++)
            {
                neighb[i] = Convert.ToByte((grayValues[(i + 1) % bytes] +
                    grayValues[Math.Abs(i - 1) % bytes] +
                    grayValues[(i + curBitmap.Width) % bytes] +
                    grayValues[Math.Abs(i - curBitmap.Width) % bytes]) / 4);
            }

            // 耗时严重
            for (int i = 0; i < bytes; i++)
            {
                for (int j = 0; j < bytes; j++)
                {
                    int ii = grayValues[i];
                    int jj = neighb[j];
                    pap[ii, jj]++;
                }
            }

            for (int i = 0; i < 256; i++)
            {
                for (int j = 0; j < 256; j++)
                {
                    pap[i, j] = ((double)pap[i, j] / bytes) / bytes;
                    if (pap[i, j] < 0.00000000000000001)
                        continue;
                    Hl2D += -pap[i, j] * Math.Log10(pap[i, j]);
                }
            }
            for (int i = 0; i <= 255; i++)
            {
                for (int j = 0; j <= 255; j++)
                {
                    Pa += pap[i, j];
                    if (pap[i, j] < 0.00000000000000001)
                        continue;
```

```
                        Ha += -pap[i, j] * Math.Log10(pap[i, j]);
                        // 公式(9.9)
                        sigma = Math.Log10(Pa * (1 - Pa)) + Ha / Pa +
                            (Hl2D - Ha) / (1 - Pa);
                        if (sigma > tempMax)
                        {
                            tempMax = sigma;
                            S = Convert.ToByte(i);
                            T = Convert.ToByte(j);
                        }
                    }
                }
                break;
            case 4:
                // 简单统计法
                int[] ee = new int[bytes];
                int ex, ey, ef = 0, esum = 0;
                for (int i = 0; i < bytes; i++)
                {
                    ex = Math.Abs(grayValues[(i + 1) % bytes] -
                        grayValues[Math.Abs(i - 1) % bytes]);
                    ey = Math.Abs(grayValues[(i + curBitmap.Width) % bytes] -
                        grayValues[Math.Abs(i - curBitmap.Width) % bytes]);
                    ee[i] = Math.Max(ex, ey);
                    ef += ee[i] * grayValues[i];
                    esum += ee[i];
                }
                // 公式(9.10)
                T = Convert.ToByte(ef / esum);
                break;
            default:
                break;
        }

        // 二值图像化
        for (int i = 0; i < bytes; i++)
        {
            if (method == 3)
            {
                // 二维最大熵
                if (grayValues[i] < S && neighb[i] < T)
                    grayValues[i] = 0;
                else
                    grayValues[i] = 255;
            }
            else
            {
                // 其他方法
                if (grayValues[i] < T)
                    grayValues[i] = 0;
                else
                    grayValues[i] = 255;
            }
        }
```

```
                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运行该段程序，这里以图 4.1 为例进行阈值分割。打开图像后，单击“阈值法”按钮，按照图 9.3 所示选择 Otsu 法，然后单击“确定”按钮，则完成阈值分割，结果如图 9.4 所示。

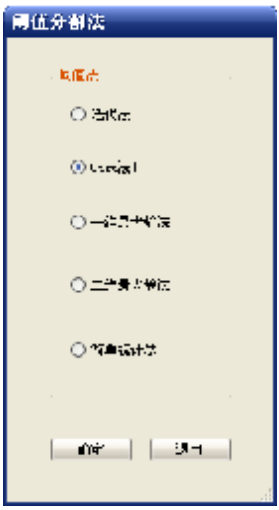


图 9.3 阈值分割法对话框

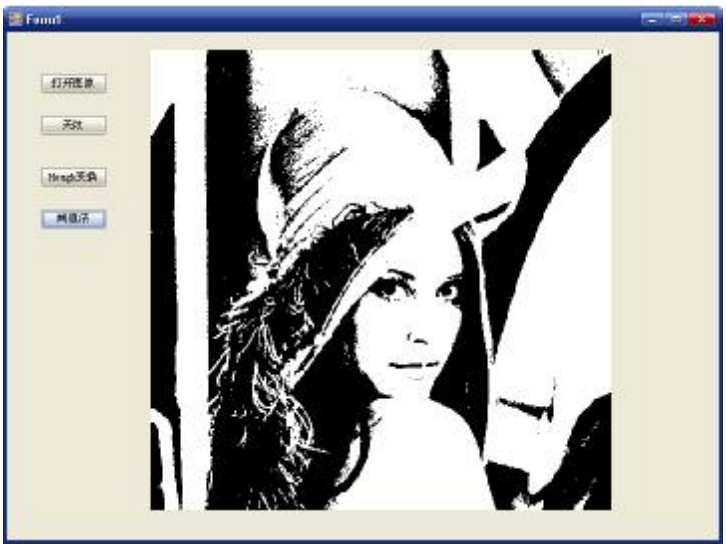


图 9.4 Otsu 法阈值分割结果

通过对这幅图像进行各种阈值分割可以得到，迭代法和 Otsu 法得到的阈值 T 均为 116，一维最大熵法得到的阈值 T 为 121，简单统计法得到的 T 为 114，二维最大熵法得到的两个阈值 S 和 T 分别为 122 和 94。

9.3 特征空间聚类法

特征空间聚类法即根据某些特征将目标点划分到相应的类别中，这种方法是一种比较简单有效的图像分割方法。典型的特征空间聚类方法包括：K-均值聚类法、ISODATA 法、模糊 C-均值法等。在本节，只给出 K-均值聚类法和 ISODATA 法的编程实例。

9.3.1 K-均值聚类法原理

K-均值聚类法可以将一幅图像分割成 K 个区域。设(m,n)代表数字图像像素的坐标，$f(m,n)$代表像素(m,n)的灰度，则 K-均值法的最小化指标为：

$$e^2=\sum_{j=1}^{K}\sum_{(m,n)\in Q_j^{(i)}}|f(m,n)-m_j^{(i+1)}|^2 \tag{9.11}$$

式中：K 表示图像中有 K 个区域，$Q_i^{(i)}$ 表示在第 i 次迭代后图像中第 j 个区域，$m_i^{(i+1)}$ 表示（i+1）次迭代后第 j 个区域的平均值。上式实际上给出了每个像素与其对应区域均值的距离和。

K-均值聚类算法的具体步骤如下。

（1）任意选 K 个初始类均值，$m_1^{(1)}$、$m_2^{(1)}$、$\mathbf{L}$、$m_K^{(1)}$。

（2）在第 i 次迭代时，根据下述距离准则将每个像素都赋给 K 类（区域）之一：

$$(m,n)\in Q_l^{(i)} \quad 若|f(m,n)-m_l^{(i)}|<|f(m,n)-m_j^{(i)}|$$

即将每个像素赋给均值离它最近的类。

（3）对每个类，更新该类的均值：

$$m_j^{(i+1)}=\frac{1}{N_j}\sum_{(m,n)\in Q_j^{(i)}}f(m,n) \tag{9.12}$$

式中 N_j 是更新后的类 $Q_i^{(i+1)}$ 中的像素个数。

（4）如果对所有的类，有 $m_i^{(i+1)}=m_i^{(i)}$，则算法收敛，结束；否则退回步骤⑵继续下一迭代。

9.3.2　ISODATA 聚类法原理

ISODATA 聚类方法是在 K-均值算法上发展起来的，它是在没有先验知识的情况下进行的一种无监督分类。首先，它选择若干样本作为聚类中心，再按照某种聚类准则，使其余样本归入最近的聚类中心，得到初始聚类；然后判断初始聚类结果是否合理，若不合理则按照特定规则将聚类集合进行分裂或合并，以获得新的聚类中心，再判断聚类结果是否符合要求。如此反复迭代，直到聚类划分符合要求为止。

为了简化程序，本节选择的初始聚类数目远远大于预期聚类数目，因此只需考虑合并聚类，而不会出现分裂聚类这种情况。具体的步骤如下：

（1）设置初始聚类数目 C 为预期聚类数目 K 的两倍，并任意选取 C 个初始聚类的均值。

（2）求各个样本到所有聚类中心的距离，按照与聚类中心距离最小的原则将各样本归入最近的聚类；

（3）更新各聚类均值；如果某聚类内的样本数目为零，则抛弃该聚类，并使聚类数 $C\leftarrow C$-1；

（4）计算全部聚类均值之间的距离，找到其最小距离。设聚类 i 和聚类 j 之间的距离最小，则将两个类合并，合并后的聚类均值为：

$$m_l=\frac{1}{n_i+n_j}(n_i m_i+n_j m_j) \tag{9.13}$$

式中：μ_i、μ_j 和 μ_l 分别为合并前聚类 i 和聚类 j 的均值，及其合并后聚类 l 的均值，n_i 和 n

为聚类 i 和聚类 j 内的样本数目。合并后聚类 l 的样本数目 n_l 为 n_i 和 n_j 之和，此时聚类数目 $C \leftarrow C-1$；

（5）如果 C 不大于 K，则结束；否则退回步骤⑵继续下一次迭代。

9.3.3 特征空间聚类法编程实例

该实例实现了通过 K-均值和 ISODATA 法的分割图像方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 9.5 所示。

表 9.5 所修改的属性

控　件	属　性	所修改内容
button1	Name	cluster
	Text	空间聚类法
	Location	37，242

（2）创建 1 个名为 cluster 的 Windows 窗体，该窗体用于选择聚类数以及特征空间聚类法。在该窗体内添加 2 个 Button 控件、1 个 GroupBox 控件、2 个 RadioButton 控件、1 个 Label 控件和 1 个 NumericUpDown 控件，其属性修改如表 9.6 所示。

表 9.6 所修改的属性

控　件	属　性	所修改内容
cluster	Size	340，270
	Text	特征空间聚类法
	ControlBox	False
groupBox1	Size	260，90
	Location	33，25
	Text	聚类方法
radioButton1	Name	kmean
	Text	K-均值聚类法
	Location	17，42
	Checked	True
radioButton2	Name	isodata
	Text	ISODATA 聚类法
	Location	141，42
button1	Name	start
	Text	确定
	Location	33，179
button2	Name	close
	Text	退出
	Location	221，179
label1	Text	分割聚类数

	Location	68，134
numericUpDown1	Name	numClusters
	Location	174，132
	Minimum	2
	Value	10

分别为 2 个 Button 控件添加 Click 事件，为 2 个 RadioButton 控件添加 CheckedChange 事件，并再添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetMethod
{
    get
    {
        // 得到空间聚类方法
        return isodata.Checked;
    }
}

public int GetNumber
{
    get
    {
        // 得到聚类数
        return Convert.ToInt16(numClusters.Value);
    }
}
```

（3）回到主窗体，为“空间聚类法”按钮添加 Click 事件，代码如下：

```
private void clus_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化cluster
        cluster cluMethod = new cluster();

        if (cluMethod.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
```

```
IntPtr ptr = bmpData.Scan0;
int bytes = curBitmap.Width * curBitmap.Height;
byte[] grayValues = new byte[bytes];
System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

// 得到空间聚类方法
bool method = cluMethod.GetMethod;
// 得到聚类数
int numbers = cluMethod.GetNumber;

if (method == false)
{
    // K-均值
    // 聚类内样本数
    int[] kNum = new int[numbers];
    // 聚类均值
    int[] kAver = new int[numbers];
    int[] kOldAver = new int[numbers];
    // 聚类内样本值总和
    int[] kSum = new int[numbers];
    int[] kTemp = new int[numbers];
    // 分割映射图
    byte[] segmentMap = new byte[bytes];

    // 初始化聚类均值
    for (int i = 0; i < numbers; i++)
    {
        kAver[i] = kOldAver[i] = Convert.ToInt16(i * 255 / (numbers - 1));
    }

    while (true)
    {
        int order = 0;
        for (int i = 0; i < numbers; i++)
        {
            kAver[i] = kOldAver[i];
            kNum[i] = 0;
            kSum[i] = 0;
        }
        // 样本归属聚类
        for (int i = 0; i < bytes; i++)
        {
            for (int j = 0; j < numbers; j++)
            {
                kTemp[j] = Math.Abs(grayValues[i] - kAver[j]);
            }
            int temp = 255;

            for (int j = 0; j < numbers; j++)
            {
                if (kTemp[j] < temp)
                {
                    temp = kTemp[j];
```

```
                            order = j;
                        }
                    }
                    kNum[order]++;
                    kSum[order] += grayValues[i];
                    segmentMap[i] = Convert.ToByte(order);
                }
                // 更新聚类均值
                for (int i = 0; i < numbers; i++)
                {
                    if (kNum[i] != 0)
                    kOldAver[i] = Convert.ToInt16(kSum[i] / kNum[i]);
                }

                // 判断迭代结束条件
                int kkk = 0;
                for (int i = 0; i < numbers; i++)
                {
                    if (kAver[i] == kOldAver[i])
                    kkk++;
                }
                // 跳出迭代循环
                if (kkk == numbers)
                    break;
            }

            // 按聚类分割图像
            for (int i = 0; i < bytes; i++)
            {
                for (int j = 0; j < numbers; j++)
                {
                    if (segmentMap[i] == j)
                    {
                        grayValues[i] = Convert.ToByte(kAver[j]);
                    }
                }
            }
        }
        else
        {
            // ISODATA
            int k = 2 * numbers;
            byte[] segmentMap = new byte[bytes];
            List<int> kTemp = new List<int>();
            List<int> kNum = new List<int>();
            List<int> kAver = new List<int>();
            List<int> kSum = new List<int>();
            // 清空列表
            kAver.Clear();
            kNum.Clear();
            kTemp.Clear();
            kSum.Clear();
```

```
// 初始化聚类均值
for (int i = 0; i < k; i++)
{
    kAver.Add(Convert.ToInt16(i * 255 / (k - 1)));
    kNum.Add(0);
    kTemp.Add(0);
    kSum.Add(0);
}

while (true)
{
    int temp;
    // 样本归入聚类
    for (int i = 0; i < bytes; i++)
    {
        kTemp.Clear();
        int order = 0;
        for (int j = 0; j < k; j++)
        {
            kTemp.Add(Math.Abs(grayValues[i] - kAver[j]));
        }
        temp = 255;

        for (int j = 0; j < k; j++)
        {
            if (kTemp[j] < temp)
            {
                temp = kTemp[j];
                order = j;
            }
        }
        int num = kNum[order] + 1;
        kNum.RemoveAt(order);
        kNum.Insert(order, num);
        int sum = kSum[order] + grayValues[i];
        kSum.RemoveAt(order);
        kSum.Insert(order, sum);
        segmentMap[i] = Convert.ToByte(order);
    }

    // 去除没有样本的聚类
    for (int i = 0; i < k; i++)
    {
        if (kNum[i] == 0)
        {
            kNum.RemoveAt(i);
            kAver.RemoveAt(i);
            kSum.RemoveAt(i);
            i--;
            k--;
        }
    }

    // 更新聚类均值
    kAver.Clear();
```

```
                for (int i = 0; i < k; i++)
                {
                    kAver.Add(Convert.ToInt16(kSum[i] / kNum[i]));
                }
                // 跳出迭代循环
                if (k <= numbers)
                    break;
                // 相似聚类合并
                temp = 255;
                int removeI = 0, removeJ = 0;
                for (int i = 0; i < k; i++)
                {
                    for (int j = i + 1; j < k; j++)
                    {
                        int distanceIJ = Math.Abs(kAver[i] - kAver[j]);
                        if (distanceIJ < temp)
                        {
                            temp = distanceIJ;
                            removeI = i;
                            removeJ = j;
                        }
                    }
                }
                k--;
                kNum.Add(kNum[removeI] + kNum[removeJ]);
                kAver.Add(Convert.ToInt16((kNum[removeI] * kAver[removeI] +
                    kNum[removeJ] * kAver[removeJ]) /
                    (kNum[removeI] + kNum[removeJ])));
                kSum.Add(kNum[removeI] * kAver[removeI] + kNum[removeJ] *
                    kAver[removeJ]);
                kNum.RemoveAt(removeI);
                kNum.RemoveAt(removeJ);
                kAver.RemoveAt(removeI);
                kAver.RemoveAt(removeJ);
                kSum.RemoveAt(removeI);
                kSum.RemoveAt(removeJ);
            }

            // 按聚类分割图像
            for (int i = 0; i < bytes; i++)
            {
                for (int j = 0; j < numbers; j++)
                {
                    if (segmentMap[i] == j)
                    {
                        grayValues[i] = Convert.ToByte(kAver[j]);
                    }
                }
            }
        }

        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
        curBitmap.UnlockBits(bmpData);
    }
```

```
            Invalidate();
        }
    }
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开图像后，单击“空间聚类法”按钮，如图 9.5 所示设置相关参数，然后单击“确定”按钮，则完成聚类法分割，结果如图 9.6 所示，它显示了通过 ISODATA 法把原图分割成 8 个聚类的结果。

图 9.5　特征空间聚类法对话框

图 9.6　特征空间聚类法结果

9.4　松弛迭代法

9.4.1　松弛迭代法原理

松弛迭代分割法是以像素为操作对象，借助迭代逐步确定各像素的归类。每次迭代中，需要用到称为相容性的准则。它的基本思想是图像中每一个像素的归属不仅应该由其本身来决定，而且应该受到它的邻域像素的影响。

松弛迭代分割法可以用以下 3 个步骤来完成。

（1）随机初始化标记。

设图像 $f(x, y)$的大小为 $M\times N$，像素为 $A(i)$（i=1,2,⋯, $M\times N$），用阈值法将图像分为 K 类，第 k 类（k=1,2,⋯,K）的均值和方差分别为μ_k和σ_k。因为像素点 i 和第 k 类的马氏距离为

$$d_{ik}=\frac{[m_k-A(i)]^2}{s_k^2} \tag{9.14}$$

则初始概率为：

$$P_i^{(0)}(k)=\frac{1/d_{ik}}{\sum_{l=1}^{K}1/d_{il}} \tag{9.15}$$

（2）规则更新。

对于类 l 和类 k，相容性矩阵 R 定义为：

$$R(l,k)=\begin{cases}1 & l=k\\ 0 & l\neq k\end{cases} \tag{9.16}$$

如果用 $Q_i(k)$表示类 k 对点 i 的相容性因素，用 $V(i)$表示点 i 的邻域，则：

$$Q_i(k)=\frac{1}{n-1}\sum_{j\in V(i)}\sum_{l=1}^{K}R(k,l)P_j(l) \tag{9.17}$$

当只考察 8 点邻域时，则 $n=8$。在第（r+1）步的概率矢量可用下式计算：

$$P_i^{(r+1)}(k)=\frac{P_i^{(n)}(k)\left[1+Q_i^{(n)}(k)\right]}{\sum_{l=1}^{K}P_i^{(n)}(l)\left[1+Q_i^{(n)}(l)\right]} \tag{9.18}$$

（3）迭代终止。

松弛迭代是一种像素标记方法，所以可设定一个量（如百分比）来确定迭代是否达到预目标，并在达到时认为迭代收敛，终止算法。

9.4.2 松弛迭代法编程实例

该实例完成了松弛迭代分割图像。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 9.7 所示。

表 9.7 所修改的属性

控　件	属　性	所修改内容
button1	Name	overRelax
	Text	松弛迭代法
	Location	37，288

（2）创建 1 个名为 ORI 的 Windows 窗体，该窗体用于选择松弛迭代法中的两个参数：分割类数和迭代次数。在该窗体内添加 2 个 Button 控件、2 个 Label 控件和 2 个 NumericUpDown 控件，其属性修改如表 9.8 所示。

表 9.8 所修改的属性

控　件	属　性	所修改内容
ORI	Size	280，235
	Text	松弛迭代分割法
	ControlBox	False
label1	Text	分割类数
	Location	25，36
label2	Text	迭代次数
	Location	25，90
numericUpDown1	Name	segNum
	Location	115，34
	Minimum	2
	Value	8
numericUpDown2	Name	iterNum
	Location	115，88
	Minimum	10

	Value	25

续表

控　件	属　性	所修改内容
button1	Name	start
	Text	确定
	Location	27，143
button2	Name	close
	Text	退出
	Location	160，143

分别为 2 个 Button 控件添加 Click 事件，并再添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public int GetSegNum
{
    get
    {
        // 得到分割类数
        return Convert.ToInt16(segNum.Value);
    }
}

public int GetIterNum
{
    get
    {
        // 得到最大迭代次数
        return Convert.ToInt16(iterNum.Value);
    }
}
```

（3）回到主窗体，为“松弛迭代法”按钮添加 Click 事件，代码如下：

```
private void overRelax_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化ORI
        ORI overRelaxIter = new ORI();

        if (overRelaxIter.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
```

```
        System.Drawing.Imaging.ImageLockMode.ReadWrite,
        curBitmap.PixelFormat);
    IntPtr ptr = bmpData.Scan0;
    int bytes = curBitmap.Width * curBitmap.Height;
    byte[] grayValues = new byte[bytes];
    System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

    // 得到分割类数
    int segNumber = overRelaxIter.GetSegNum;
    // 得到迭代次数
    int iterNumber = overRelaxIter.GetIterNum;

    // 类总和
    int[] kSum = new int[segNumber];
    // 类均值
    double[] kAver = new double[segNumber];
    // 类方差
    double[] kVar = new double[segNumber];
    // 数组清零
    Array.Clear(kAver, 0, segNumber);
    Array.Clear(kSum, 0, segNumber);
    Array.Clear(kVar, 0, segNumber);

    int[] imageMap = new int[bytes];

    // 计算类的总和、均值和方差
    for (int i = 0; i < bytes; i++)
    {
        for (int j = 1; j <= segNumber; j++)
        {
            if (grayValues[i] < Convert.ToByte(255 * j / segNumber))
            {
                imageMap[i] = j - 1;
                kAver[j - 1] += grayValues[i];
                kSum[j - 1] += 1;
                break;
            }
        }
    }
    for (int i = 0; i < segNumber; i++)
    {
        if (kSum[i] != 0)
            kAver[i] /= kSum[i];
    }
    for (int i = 0; i < bytes; i++)
        kVar[imageMap[i]] += Convert.ToInt16(Math.Pow(grayValues[i] -
            kAver[imageMap[i]], 2));
    for (int i = 0; i < segNumber; i++)
    {
        if (kSum[i] != 0)
            kVar[i] /= kSum[i];
    }

    double[] d = new double[bytes * segNumber];
    double[] kProb = new double[bytes * segNumber];
    Array.Clear(d, 0, bytes * segNumber);
```

```
            Array.Clear(kProb, 0, bytes * segNumber);

            // 计算马氏距离
            for (int i = 0; i < bytes; i++)
            {
                for (int j = 0; j < segNumber; j++)
                {
                    if (kVar[j] == 0)
                        kVar[j] = 0.00000005;
                    d[i * segNumber + j] = Math.Pow(kAver[j] - grayValues[i], 2) / kVar[j];
                    if (d[i * segNumber + j] == 0)
                        d[i * segNumber + j] = 0.00000005;
                }
            }

            // 计算初始概率
            double tempSum = 0;
            for (int i = 0; i < bytes; i++)
            {
                tempSum = 0;
                for (int j = 0; j < segNumber; j++)
                tempSum += 1 / d[i * segNumber + j];
                for (int j = 0; j < segNumber; j++)
                    kProb[i * segNumber + j] = 1 / (tempSum * d[i * segNumber + j]);
            }

            // 迭代循环
            while (iterNumber != 0)
            {
                iterNumber--;

                for (int i = 0; i < bytes; i++)
                {
                    for (int j = 0; j < segNumber; j++)
                    {
                        // 8个邻域的概率总和
    tempSum = kProb[(Math.Abs(i + 1 - curBitmap.Width) % bytes) * segNumber + j] +
         kProb[(Math.Abs(i - curBitmap.Width) % bytes) * segNumber + j] +
        kProb[(Math.Abs(i - 1 - curBitmap.Width) % bytes) * segNumber + j] +
        kProb[(Math.Abs(i - 1) % bytes) * segNumber + j] +
        kProb[((i - 1 + curBitmap.Width) % bytes) * segNumber + j] +
        kProb[((i + curBitmap.Width) % bytes) * segNumber + j] +
        kProb[((i + 1 + curBitmap.Width) % bytes) * segNumber + j] +
        kProb[((i + 1) % bytes) * segNumber + j];
                        // 更新每个类的概率
                        d[i * segNumber + j] = tempSum / 8;
                        tempSum = 0;
                        for (int k = 0; k < segNumber; k++)
                            tempSum +=
                                kProb[i * segNumber + k] * (1 + d[i * segNumber + k]);
                        kProb[i * segNumber + j] *=
                            (1 + d[i * segNumber + j]) / tempSum;
                    }
                }
            }
```

```
                // 得到分割后的灰度图
                for (int i = 0; i < bytes; i++)
                {
                    double tempMax = 0;
                    int m = 0;
                    for (int j = 0; j < segNumber; j++)
                    {
                        if (kProb[i * segNumber + j] > tempMax)
                        {
                            tempMax = kProb[i * segNumber + j];
                            m = j;
                        }
                    }
                    grayValues[i] = Convert.ToByte(m * 255 / segNumber);
                }

                 System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                 curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开图像后，单击“松弛迭代法”按钮，出图 9.7 所示设置相关参数。

单击“确定”按钮，则完成松弛迭代分割图像，结果如图 9.8 所示，它显示了通过 40 次迭代后把图像分成 10 类的结果

图 9.7 松弛迭代法对话框

图 9.8 松弛迭代法结果

9.5 小结

图像分割是从图像处理到图像分析的关键步骤，可以说，图像分割结果的好坏直接影响到对图像的理解。因此它是一种重要的图像技术，在理论研究和实际应用中都得到了人们的

泛重视。它的方法和种类也很多。本章只是介绍了众多图像算法中最具代表性的几种。

第 8 章　边缘检测

图像属性中的显著变化通常反映了属性的重要事件和变化，这些包括：深度上的不连续表面方向不连续、物质属性变化和场景照明变化。它们在图像中表现为亮度变化明显的点。边缘检测的目的就是标识这些像素点，它是图像识别的基础和前提，是图像分隔的一部分。它大幅度地减少了数据量，并剔除了可以认为不相关的信息，保留了图像重要的结构属性。因为边缘检测在数字图像处理中有重要地位。

边缘检测就是找到图像中边缘像素点的过程，从而生成一幅边缘图。一般来说，边缘检测分为 3 步：

（1）基于各种原理和方法找到潜在边缘点；

（2）选取阈值，生成二值边缘图；

（3）有些算法还要进行边缘细化、连接等后续处理。

在本章，我们不对第 3 步进行分析处理，而主要介绍如何找到那些潜在的边缘像素点。

8.1　模板算子法

8.1.1　模板算子法原理

边缘是图像中灰度值不连续（或突变）的结果，这种不连续性常可利用求导数的方法方便地检测到，一般常用一阶导数（梯度）和二阶导数（拉普拉斯）来检测边缘，如图 8.1 所示。

图 8.1 为边缘检测在一维连续域 $f_c(x)$内的微分运算，x_0 和 x_1 都是边缘点，$f_c`(x)$和 $f_c``(x)$分别为一阶导数和二阶导数。由上图可知，在一阶导数中，边缘点表现为一个局部极值点，而在二阶导数中，表现为一个过零点。由此可知，图像中目标的边缘可通过求取它们的导数来确定。导数可用微分算子来计算，而数字图像中求导数是利用差分近似微分来进行的。

在一阶导数方法中，对图像中两个正交方向分别求偏导数，然后对这两个偏导数取不同的范数作为边缘强度，即可得到边缘图像。在这里，我们选取最常用的以 2 为模的范数形式，即对这两个偏导数取平方和，再开平方。在实际计算中，求偏导数常常采用的是用小区域模

或进行卷积来近似计算的。一阶导数模板算子最常用的方法有 Roberts 算子法、Prewitt 算子法和 Sobel 算子法。

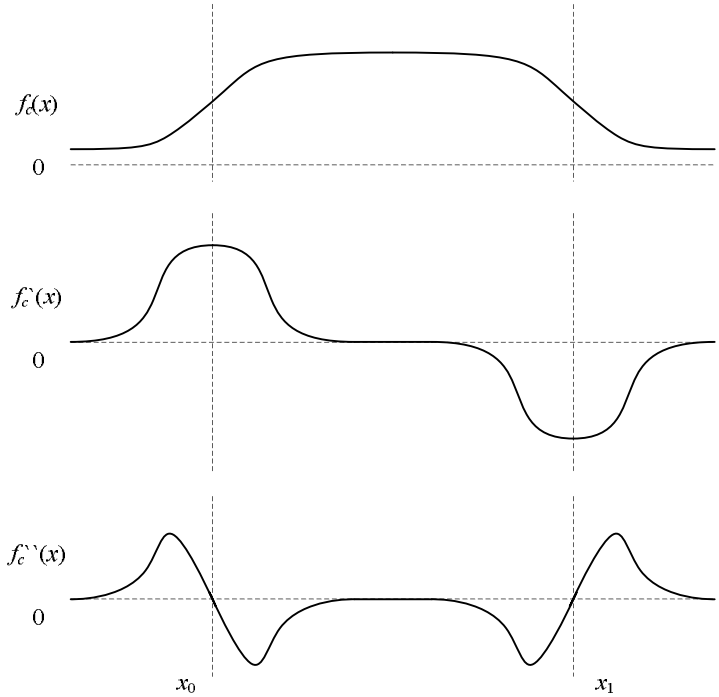


图 8.1　边缘检测的微分运算

Roberts 算子（粗体字为当前所求像素点，下同）：

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{8.1}$$

Prewitt 算子：

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{8.2}$$

Sobel 算子：

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{8.3}$$

二阶导数方法，即拉普拉斯算子法，是无方向的算子，因此它只需要一个模板，以下是三个最常用的拉普拉斯算子模板：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{8.4}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{8.5}$$

$$\begin{bmatrix} -1 & 2 & -1 \\ 2 & -4 & 2 \\ -1 & 2 & -1 \end{bmatrix} \tag{8.6}$$

经过拉普拉斯算子卷积以后的图像，还要判断出它的零交叉点，也就是边缘点。在这里我们用四邻域法，通过比较符号和大小来判断零交叉点，从而得到边缘像素。在实际应用中一方面是因为拉普拉斯算子对图像噪声十分敏感，另一方面就是边缘是双像素宽的，且不能提供边缘方向信息，因此它很少直接用于边缘检测，而是与其他方法相结合，提取出定位精确的边缘。

利用 8 个不同的模板对图像进行卷积处理，每个模板对某个特定边缘方向做最大响应，然后取所有 8 个方向中的最大值作为边缘强度，最终组成边缘图像，该方法就是 Kirsch 算子法。

$$\begin{bmatrix} -5 & 3 & 3 \\ -5 & 0 & 3 \\ -5 & 3 & 3 \end{bmatrix} \begin{bmatrix} 3 & 3 & 3 \\ -5 & 0 & 3 \\ -5 & -5 & 3 \end{bmatrix} \begin{bmatrix} 3 & 3 & 3 \\ 3 & 0 & 3 \\ -5 & -5 & -5 \end{bmatrix} \begin{bmatrix} 3 & 3 & 3 \\ 3 & 0 & -5 \\ 3 & -5 & -5 \end{bmatrix}$$
$$\begin{bmatrix} 3 & 3 & -5 \\ 3 & 0 & -5 \\ 3 & 3 & -5 \end{bmatrix} \begin{bmatrix} 3 & -5 & -5 \\ 3 & 0 & -5 \\ 3 & 3 & 3 \end{bmatrix} \begin{bmatrix} -5 & -5 & -5 \\ 3 & 0 & 3 \\ 3 & 3 & 3 \end{bmatrix} \begin{bmatrix} -5 & -5 & 3 \\ -5 & 0 & 3 \\ 3 & 3 & 3 \end{bmatrix} \tag{8.7}$$

8.1.2 模板算子法编程实例

该实例实现了上述几种边缘检测方法。

（1）创建 1 个“模板主窗体”，在该窗体内添加 1 个 Button 控件，其属性修改如表 8.1 所示。

表 8.1 所修改的属性

控　件	属　性	所修改内容
button1	Name	mask
	Text	模板算子法
	Location	37，150

（2）创建 1 个名为 mask 的 Windows 窗体，该窗体用于选择模板边缘检测的几种方法以及用于二值图像化的阈值。在该窗体内添加 2 个 Button 控件、1 个 GroupBox 控件、7 个 RadioButton 控件、1 个 Label 控件和 1 个 NumericUpDown 控件，其属性修改如表 8.2

所示。

表 8.2 所修改的属性

控　件	属　性	所修改内容
mask	Text	模板算子
	Size	270，520
	ControlBox	False
button1	Name	start
	Text	确定
	Location	40，440
button2	Name	close
	Text	退出
	Location	149，440
groupBox1	Text	边缘检测模板
	Size	184，346
	Location	40，22
radioButton1	Name	roberts
	Text	Roberts
	Location	30，30
	Checked	True
radioButton2	Name	prewitt
	Text	Prewitt
	Location	30，76
radioButton3	Name	sobel
	Text	Sobel
	Location	30，122
radioButton4	Name	laplacian1
	Text	Laplacian1
	Location	30，168
radioButton5	Name	laplacian2
	Text	Laplacian2
	Location	30，214
radioButton6	Name	laplacian3
	Text	Laplacian3
	Location	30，260
radioButton7	Name	kirsch
	Text	Kirsch
	Location	30，306
label1	Text	阈值：
	Location	59，394
numericUpDown1	Name	thresholding
	Location	116，392
	Maximum	500
	Value	100

分别为两个 Button 控件添加 Click 事件，为 RadioButton 控件添加 CheckedChanged 事件并再添加 2 个 get 属性访问器，代码如下：

```
private void close_Click(object sender, EventArgs e)
{
     this.Close();
}

private void start_Click(object sender, EventArgs e)
{
     this.DialogResult = DialogResult.OK;
}

private void roberts_CheckedChanged(object sender, EventArgs e)
{
     operMask = 0;
}

private void prewitt_CheckedChanged(object sender, EventArgs e)
{
     operMask = 1;
}

private void sobel_CheckedChanged(object sender, EventArgs e)
{
     operMask = 2;
}

private void laplacian1_CheckedChanged(object sender, EventArgs e)
{
     operMask = 3;
}

private void laplacian2_CheckedChanged(object sender, EventArgs e)
{
     operMask = 4;
}

private void laplacian3_CheckedChanged(object sender, EventArgs e)
{
     operMask = 5;
}

private void kirsch_CheckedChanged(object sender, EventArgs e)
{
     operMask = 6;
}

public int GetThresholding
{
    get
    {
          // 得到阈值
          return (int)thresholding.Value;
    }
```

```
}

public byte GetMask
{
    get
    {
        // 得到何种边缘检测方法
        return operMask;
    }
}
```

其中 operMask 是 byte 型变量，它在构造函数内被初始化，代码如下：

```
operMask = 0;
```

（3）回到主窗体，为“模板算子法”按钮添加 Click 事件，代码如下：

```
private void mask_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化mask
        mask operatorMask = new mask();

        if (operatorMask.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            // 得到阈值
            int thresholding = operatorMask.GetThresholding;
            // 得到边缘检测方法
            byte flagMask = operatorMask.GetMask;
            double[] tempArray = new double[bytes];
            double gradX, gradY, grad;

            switch (flagMask)
            {
                case 0:
                    // Roberts
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
    gradX = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] - grayValues[i * curBitmap.Width + j];
    gradY = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] -
         grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)];
                            grad = Math.Sqrt(gradX * gradX + gradY * gradY);
                            tempArray[i * curBitmap.Width + j] = grad;
                        }
```

```
                    }
                    break;
                case 1:
                    // Prewitt
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                         for (int j = 0; j < curBitmap.Width; j++)
                         {
gradX = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] +
    grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
    grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        (Math.Abs(j - 1) % curBitmap.Width)] -
    grayValues[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)] -
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        (Math.Abs(j - 1) % curBitmap.Width)];
gradY = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        (Math.Abs(j - 1) % curBitmap.Width)] +
    grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
    grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        (Math.Abs(j - 1) % curBitmap.Width)] -
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)];
                              grad = Math.Sqrt(gradX * gradX + gradY * gradY);
                              tempArray[i * curBitmap.Width + j] = grad;
                          }
                    }
                    break;
                case 2:
                    // Sobel
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                         for (int j = 0; j < curBitmap.Width; j++)
                         {
gradX = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] +
    2 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] –
    grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] -
    2 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)];
gradY = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] +
    2 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
    grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
    grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
```

```
            ((Math.Abs(j - 1)) % curBitmap.Width)] -
        2 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)];
                            grad = Math.Sqrt(gradX * gradX + gradY * gradY);
                            tempArray[i * curBitmap.Width + j] = grad;
                        }
                    }
                    break;
                case 3:
                    // Laplacian1 公式(8.4)
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
    grad = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
        grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] +
        grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
        grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
        4 * grayValues[i * curBitmap.Width + j];
                            tempArray[i * curBitmap.Width + j] = grad;
                        }
                    }
                    break;
                case 4:
                    // Laplacian2 公式(8.5)
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
    grad = grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] +
       grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
       grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)] +
       grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] +
       grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
       grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
           ((Math.Abs(j - 1)) % curBitmap.Width)] +
       grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
       grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
           ((j + 1) % curBitmap.Width)] -
       8 * grayValues[i * curBitmap.Width + j];
                            tempArray[i * curBitmap.Width + j] = grad;
                        }
                    }
                    break;
                case 5:
                    // Laplacian3 公式(8.6)
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
    grad = -1 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
            ((Math.Abs(j - 1)) % curBitmap.Width)] +
```

```
            2 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
            grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            2 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
            2 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] -
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
               ((Math.Abs(j - 1)) % curBitmap.Width)] +
            2 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
               ((j + 1) % curBitmap.Width)] -
            4 * grayValues[i * curBitmap.Width + j];
                                tempArray[i * curBitmap.Width + j] = grad;
                           }
                      }
                      break;
                 case 6:
                      // Kirsch
                      for (int i = 0; i < curBitmap.Height; i++)
                      {
                           for (int j = 0; j < curBitmap.Width; j++)
                           {
                                grad = 0;

gradX = -5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
              ((Math.Abs(j - 1)) % curBitmap.Width)] +
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
              ((j + 1) % curBitmap.Width)] -
            5 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
            3 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
              ((Math.Abs(j - 1)) % curBitmap.Width)] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
              ((j + 1) % curBitmap.Width)];
                                if (gradX > grad)
                                       grad = gradX;

gradX = 3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
              ((Math.Abs(j - 1)) % curBitmap.Width)] +
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
              ((j + 1) % curBitmap.Width)] -
            5 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
            3 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
              ((Math.Abs(j - 1)) % curBitmap.Width)] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
              ((j + 1) % curBitmap.Width)];
                                if (gradX > grad)
                                      grad = gradX;

gradX = 3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
              ((Math.Abs(j - 1)) % curBitmap.Width)] +
```

```
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)] +
            3 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
            3 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((Math.Abs(j - 1)) % curBitmap.Width)] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)];
                                    if (gradX > grad)
                                          grad = gradX;

        gradX = 3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                  ((Math.Abs(j - 1)) % curBitmap.Width)] +
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)] +
            3 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
            5 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((Math.Abs(j - 1)) % curBitmap.Width)] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)];
                                    if (gradX > grad)
                                          grad = gradX;

        gradX = 3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                  ((Math.Abs(j - 1)) % curBitmap.Width)] +
            3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
            5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)] +
            3 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
            5 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((Math.Abs(j - 1)) % curBitmap.Width)] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            5 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)];
                                    if (gradX > grad)
                                          grad = gradX;

        gradX = 3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                  ((Math.Abs(j - 1)) % curBitmap.Width)] -
            5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
            5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)] +
            3 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
            5 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((Math.Abs(j - 1)) % curBitmap.Width)] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                  ((j + 1) % curBitmap.Width)];
```

```
                                if (gradX > grad)
                                      grad = gradX;

gradX = -5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
        ((Math.Abs(j - 1)) % curBitmap.Width)] -
   5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
   5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] +
   3 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
   3 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
   3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] +
   3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
   3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)];
                                if (gradX > grad)
                                      grad = gradX;

gradX = -5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
        ((Math.Abs(j - 1)) % curBitmap.Width)] -
   5 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
   3 * grayValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
   5 * grayValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
   3 * grayValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
   3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] +
   3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
   3 * grayValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)];
                                if (gradX > grad)
                                      grad = gradX;

                                tempArray[i * curBitmap.Width + j] = grad;
                           }
                      }
                      break;
                default:
                      MessageBox.Show("无效! ");
                      break;
          }

          if (thresholding == 0)
          {
                // 不进行阈值处理，即仍然是灰度图像
                for (int i = 0; i < bytes; i++)
                {
                      if (tempArray[i] < 0)
                            grayValues[i] = 0;
                      else
                      {
                            if (tempArray[i] > 255)
                                  grayValues[i] = 255;
                            else
                                  grayValues[i] = Convert.ToByte(tempArray[i]);
```

```
                        }
                    }
                }
                else
                {
                    // 阈值处理，生成二值边缘图像
                    if (flagMask == 3 || flagMask == 4 || flagMask == 5)
                    {
                        // 拉普拉斯方法时，调用零交叉方法找到边缘像素点
                        zerocross(ref tempArray, out grayValues, thresholding);
                    }
                    else
                    {
                        // 其他方法，就是进行简单的阈值化处理
                        for (int i = 0; i < bytes; i++)
                        {
                            if (tempArray[i] > thresholding)
                                grayValues[i] = 255;
                            else
                                grayValues[i] = 0;
                        }
                    }
                }

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }

    /*******************************************
    阈值处理：找到零交叉点
    inputImage：输入图像
    outImage：输出图像
    thresh：阈值
    *******************************************/
    private void zerocross(ref double[] inputImage, out byte[] outImage, double thresh)
    {
        outImage = new byte[curBitmap.Width * curBitmap.Height];
        for (int i = 0; i < curBitmap.Height; i++)
        {
            for (int j = 0; j < curBitmap.Width; j++)
            {
                if (inputImage[i * curBitmap.Width + j] < 0 &&
                    inputImage[((i + 1) % curBitmap.Height) * curBitmap.Width + j] > 0 &
                    Math.Abs(inputImage[i * curBitmap.Width + j] -
                    inputImage[((i + 1) % curBitmap.Height) * curBitmap.Width + j]) > thresh
                {
                    outImage[i * curBitmap.Width + j] = 255;
                }
                else if (inputImage[i * curBitmap.Width + j] < 0 &&
                    inputImage[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width
                    j] > 0 && Math.Abs(inputImage[i * curBitmap.Width + j] -
                    inputImage[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j]
                    > thresh)
                {
```

```
                    outImage[i * curBitmap.Width + j] = 255;
                }
                else if (inputImage[i * curBitmap.Width + j] < 0 &&
                    inputImage[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] > 0 &
                    Math.Abs(inputImage[i * curBitmap.Width + j] -
                    inputImage[i * curBitmap.Width + ((j + 1) % curBitmap.Width)]) > thresh
                {
                    outImage[i * curBitmap.Width + j] = 255;
                }
                else if (inputImage[i * curBitmap.Width + j] < 0 &&
                    inputImage[i * curBitmap.Width + ((Math.Abs(j - 1)) %
                    curBitmap.Width)] > 0 && Math.Abs(inputImage[i * curBitmap.Width + j]
                    inputImage[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]
                    > thresh)
                {
                    outImage[i * curBitmap.Width + j] = 255;
                }
                else if (inputImage[i * curBitmap.Width + j] == 0)
                {
                    if (inputImage[((i + 1) % curBitmap.Height) * curBitmap.Width + j] > 0 &
                        inputImage[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap. Widt
                        + j] < 0 && Math.Abs(inputImage[((Math.Abs(i - 1)) %
                        curBitmap.Height) * curBitmap.Width + j] - inputImage[((i + 1)
                        curBitmap.Height) * curBitmap.Width + j]) > 2 * thresh)
                    {
                        outImage[i * curBitmap.Width + j] = 255;
                    }
                    else if (inputImage[((i + 1) % curBitmap.Height) * curBitmap.Width + j] <
                        && inputImage[((Math.Abs(i - 1)) % curBitmap.Height) *
                        curBitmap.Width + j] > 0 && Math.Abs(inputImage[((Math.Abs(i - 1)
                        % curBitmap.Height) * curBitmap.Width + j] - inputImage[((i + 1)
                        curBitmap.Height) * curBitmap.Width + j]) > 2 * thresh)
                    {
                        outImage[i * curBitmap.Width + j] = 255;
                    }
                    else if (inputImage[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] >
                        && inputImage[i * curBitmap.Width + ((Math.Abs(j - 1)) %
                        curBitmap.Width)] < 0 && Math.Abs(inputImage[i * curBitmap.Width
                        ((j + 1) % curBitmap.Width)] - inputImage[i * curBitmap.Width
                        ((Math.Abs(j - 1)) % curBitmap.Width)]) > 2 * thresh)
                    {
                        outImage[i * curBitmap.Width + j] = 255;
                    }
                    else if (inputImage[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] <
                        && inputImage[i * curBitmap.Width + ((Math.Abs(j - 1)) %
                        curBitmap.Width)] > 0 && Math.Abs(inputImage[i * curBitmap.Width
                        ((j + 1) % curBitmap.Width)] - inputImage[i * curBitmap.Width
                        ((Math.Abs(j - 1)) % curBitmap.Width)]) > 2 * thresh)
                    {
                        outImage[i * curBitmap.Width + j] = 255;
                    }
                    else
                    {
                        outImage[i * curBitmap.Width + j] = 0;
                    }
                }
                else
```

```
                {
                    outImage[i * curBitmap.Width + j] = 0;
                }
            }
        }
    }
```

（4）编译并运行该段程序，本章都是以图 4.1 为例进行边缘检测
了开该图后，单击“模板算子法”按钮，打开模板算子对话框，如图
.2 所示。

图 8.2 模板算子对话

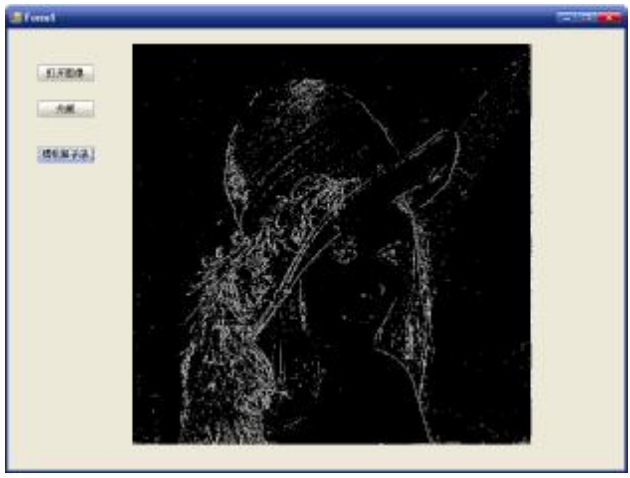
我们分别选择 Roberts 算子法、Prewitt 算子法、Sobel 算子法、
aplacian2 算子法和 Kirsch 算子法，它们的阈值分别为 40、120、150、
30 和 300，边缘检测的结果如图 8.3 所示。

需要说明的是，在上面的程序中，我们使用了循环延拓的方法。
它虽然克服了图像四周无法被模板处理的问题，但它也使程序运行速
度降低，而且使程序的可读性也变差了。

（a）Roberts 边缘图像（阈值为 40）

（b）Prewitt 边缘图像（阈值为 120）

（c）Sobel 边缘图像（阈值为 150）



（d）Laplacian2 边缘图像（阈值为 130）

（e）Kirsch 边缘图像（阈值为 300）

图 8.3　边缘检测结果

8.2 高斯算子

8.2.1 高斯算子原理

从图 8.3（d）可以看出，拉普拉斯算子对图像噪声比较敏感。为了减少噪声的影响，可以用高斯函数对图像先进行平滑卷积滤波处理，然后再运用拉普拉斯算子。高斯滤波器可定义为如下形式：

$$g_c(x,y)=\exp\left(-\frac{x^2+y^2}{2s^2}\right) \tag{8.8}$$

又因为卷积运算和拉普拉斯算子都是线性移不变的，所以它们的运算顺序可以颠倒，即先进行拉普拉斯算子，再进行高斯卷积运算，对运算结果没有丝毫影响。这种将高斯卷积和拉普拉斯算子组合成一个滤波器或算子的方法称为拉普拉斯-高斯（LoG）滤波器或算子，它的脉冲响应函数为：

$$h_c(x,y)=\frac{x^2+y^2-2s^2}{s^4}\exp\left(-\frac{x^2+y^2}{2s^2}\right) \tag{8.9}$$

对上式进行采样，就可构造一个离散形式的响应函数，从而最终能够把它应用到数字图像处理中去。所构造的高斯模板半径一般都为 3σ。

LoG 滤波器（公式（8.9））可表示成两个高斯函数之差，因此也可用差分高斯（DoG）算子对图像进行边缘检测，该滤波器定义为如下形式：

$$h_c(x,y)=g_{c1}(x,y)-g_{c2}(x,y)=\frac{1}{s_1}\exp\left(-\frac{x^2+y^2}{2s_1^2}\right)-\frac{1}{s_2}\exp\left(-\frac{x^2+y^2}{2s_2^2}\right) \tag{8.10}$$

其中，两个高斯函数的均方差为$\sigma_2\approx1.6\sigma_1$。

无论是 LoG，还是 DoG，为了不改变卷积后图像的整体动态范围，还要使模板元素的代数和为零。根据高斯函数的可分离性，可以把二维高斯算子分解为一维高斯算子来提供运算速度。但本节仍然采样二维卷积的方法。

卷积处理后，仍然要用上一节给出的方法得到零交叉边缘点。

8.2.2 高斯算子编程实例

该实例实现了 LoG 和 DoG 滤波边缘检测方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 8.3 所示。

表 8.3　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	gaussian
	Text	高斯算子

	Location	37，196

（2）创建 1 个名为 gaussian 的 Windows 窗体，该窗体用于选择是 LoG 方法还是 DoG 法，以及均方差和阈值。在该窗体内添加两个 Button 控件、1 个 GroupBox 控件、2 个 RadioButton 控件、2 个 Label 控件和 2 个 TextBox 控件，其属性修改如表 8.4 所示。

表 8.4　　所修改的属性

控　件	属　性	所修改内容
gaussian	Text	高斯算子
	ControlBox	False
	Size	250，340
button1	Name	start
	Text	确定
	Location	29，256
button2	Name	close
	Text	退出
	Location	140，256
groupBox1	Text	高斯算子
	Location	29，22
	Size	186，124
radioButton1	Name	log
	Text	拉普拉斯-高斯算子
	Location	21，32
	Checked	True
radioButton2	Name	dog
	Text	差分高斯算子
	Location	21，71
label1	Text	均方差：
	Location	48，167
label2	Text	阈值：
	Location	48，208
textBox1	Name	sigmaValue
	Size	60，21
	Location	115，164
	Text	1.5
textBox2	Name	threshValue
	Size	60，21
	Location	115，205
	Text	40

分别为 2 个 Button 控件添加 Click 事件，并再添加 3 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
```

```
        this.DialogResult = DialogResult.OK;
    }

    private void close_Click(object sender, EventArgs e)
    {
        this.Close();
    }

    public double GetThresholding
    {
        get
        {
            // 得到阈值
            return Convert.ToDouble(thresdValue.Text);
        }
    }

    public double GetSigma
    {
        get
        {
            // 得到均方差
            return Convert.ToDouble(sigmaValue.Text);
        }
    }

    public bool GetFlag
    {
        get
        {
            // 得到使用何种方法
            return dog.Checked;
        }
    }
```

（3）回到主窗体，为“高斯算子”按钮添加 Click 事件，代码如下：

```
private void gaussian_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
          // 实例化gaussian
          gaussian gaussFilter = new gaussian();

          if (gaussFilter.ShowDialog() == DialogResult.OK)
          {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height)
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] grayValues = new byte[bytes];
                System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes)

                // 得到阈值
                double thresholding = gaussFilter.GetThresholding;
```

```
                    // 得到均方差
                    double sigma = gaussFilter.GetSigma;
                    // 得到使用何种方法
                    bool flag = gaussFilter.GetFlag;

                    double[] filt, tempArray;
                    // 调用创建高斯滤波模板方法
                    createFilter(out filt, sigma, flag);
                    // 调用卷积计算方法
                    conv2(ref grayValues, ref filt, out tempArray);
                    // 阈值处理，调用零交叉方法，找到边缘像素点
                    zerocross(ref tempArray, out grayValues, thresholding);

                    System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes)
                    curBitmap.UnlockBits(bmpData);
               }

               Invalidate();
          }
     }

     /*****************************************
     创建高斯滤波模板
     filter：所创建的模板
     sigma：高斯均方差
     lod：判断是LoG（false），还是DoG（true）
     *****************************************/
     private void createFilter(out double[] filter, double sigma, bool lod)
     {
          // 标准方差
          double std2 = 2 * sigma * sigma;
          // 半径=3σ
          int radius = Convert.ToInt16(Math.Ceiling(3 * sigma));
          int filterWidth = 2 * radius + 1;
          filter = new double[filterWidth * filterWidth];
          double sum = 0, average = 0;
          if (lod == false)
          {
               // LoG处理
               // 因为模板是中心对称的，所以先得到模板左上角的值，再赋值到全部模板

               // 计算模板左上角
               for (int i = 0; i < radius; i++)
               {
                    for (int j = 0; j < radius; j++)
                    {
                         int xx = (j - radius) * (j - radius);
                         int yy = (i - radius) * (i - radius);
                         filter[i * filterWidth + j] = (xx + yy - std2) * Math.Exp(-(xx + yy) / std2)
                         sum += 4 * filter[i * filterWidth + j];
                    }
               }
               // 水平和垂直对称轴要单独处理
               for (int i = 0; i < radius; i++)
               {
                    int xx = (i - radius) * (i - radius);
                    filter[i * filterWidth + radius] = (xx - std2) * Math.Exp(-xx / std2);
```

```
                    sum += 2 * filter[i * filterWidth + radius];
               }
               for (int j = 0; j < radius; j++)
               {
                    int yy = (j - radius) * (j - radius);
                    filter[radius * filterWidth + j] = (yy - std2) * Math.Exp(-yy / std2);
                    sum += 2 * filter[radius * filterWidth + j];
               }
               // 中心点
               filter[radius * filterWidth + radius] = -std2;
               // 所有模板数据和
               sum += filter[radius * filterWidth + radius];
               // 计算平均值
               average = sum / filter.Length;

               // 赋值
               for (int i = 0; i < radius; i++)
               {
                    for (int j = 0; j < radius; j++)
                    {
                         filter[i * filterWidth + j] = filter[i * filterWidth + j] - average
                         filter[filterWidth - 1 - j + i * filterWidth] = filter[i * filterWidth + j]
                         filter[j + (filterWidth - 1 - i) * filterWidth] = filter[i * filterWidt
j];
                         filter[filterWidth - 1 - j + (filterWidth - 1 - i) * filterWidth]
                               filter[i * filterWidth + j];
                    }
               }
               // 赋值水平和垂直对称轴
               for (int i = 0; i < radius; i++)
               {
                    filter[i * filterWidth + radius] = filter[i * filterWidth + radius] - average
                    filter[(filterWidth - 1 - i) * filterWidth + radius] = filter[i * filterWidt
radius];
               }
               for (int j = 0; j < radius; j++)
               {
                    filter[radius * filterWidth + j] = filter[radius * filterWidth + j] - average
                    filter[radius * filterWidth + filterWidth - 1 - j] = filter[radius *
ilterWidth + j];
               }
               // 赋值中心点
               filter[radius * filterWidth + radius] = filter[radius * filterWidth + radius
average;
          }
          else
          {
               // DoG 处理
               // 因为模板是中心对称的，所以先得到模板左上角的值，再赋值到全部模板

               // 计算模板左上角
               for (int i = 0; i < radius; i++)
               {
                    for (int j = 0; j < radius; j++)
                    {
                         int xx = (j - radius) * (j - radius);
                         int yy = (i - radius) * (i - radius);
```

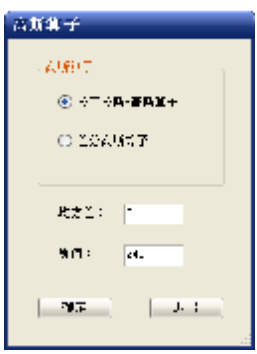
```
                        filter[i * filterWidth + j] = 1.6 * Math.Exp(-(xx + yy) * 1.6 * 1.
std2) / sigma
                               - Math.Exp(-(xx + yy) / std2) / sigma;
                        sum += 4 * filter[i * filterWidth + j];
                    }
               }
               // 水平和垂直对称轴要单独处理
               for (int i = 0; i < radius; i++)
               {
                    int xx = (i - radius) * (i - radius);
                    filter[i * filterWidth + radius] = 1.6 * Math.Exp(-xx * 1.6 * 1.6 / std2) / sigm
                          - Math.Exp(-xx / std2) / sigma;
                    sum += 2 * filter[i * filterWidth + radius];
               }
               for (int j = 0; j < radius; j++)
               {
                    int yy = (j - radius) * (j - radius);
                    filter[radius * filterWidth + j] = 1.6 * Math.Exp(-yy * 1.6 * 1.6 / std2) / sigm
                          - Math.Exp(-yy / std2) / sigma;
                    sum += 2 * filter[radius * filterWidth + j];
               }
               // 中心点
               filter[radius * filterWidth + radius] = 1.6 / sigma - 1 / sigma;
               // 所有模板数据和
               sum += filter[radius * filterWidth + radius];
               // 计算平均值
               average = sum / filter.Length;

               // 赋值
               for (int i = 0; i < radius; i++)
               {
                    for (int j = 0; j < radius; j++)
                    {
                         filter[i * filterWidth + j] = filter[i * filterWidth + j] - average
                         filter[filterWidth - 1 - j + i * filterWidth] = filter[i * filterWidth + j]
                         filter[j + (filterWidth - 1 - i) * filterWidth] = filter[i * filterWidth + j]
                         filter[filterWidth - 1 - j + (filterWidth - 1 - i) * filterWidth]
                               filter[i * filterWidth + j];
                    }
               }
               // 赋值水平和垂直对称轴
               for (int i = 0; i < radius; i++)
               {
                    filter[i * filterWidth + radius] = filter[i * filterWidth + radius] - average
                    filter[(filterWidth - 1 - i) * filterWidth + radius] = filter[i * filterWidt
radius];
               }
               for (int j = 0; j < radius; j++)
               {
                    filter[radius * filterWidth + j] = filter[radius * filterWidth + j] - average
                    filter[radius * filterWidth + filterWidth - 1 - j] = filter[radius *
ilterWidth + j];
               }
               // 赋值中心点
               filter[radius * filterWidth + radius] = filter[radius * filterWidth + radius
average;
          }
```

```
}

/*********************************
卷积计算
inputImage：输入图像
mask：卷积核
outImage：输出图像
*********************************/
private void conv2(ref byte[] inputImage, ref double[] mask, out double[] outImage)
{
     int windWidth = Convert.ToInt16(Math.Sqrt(mask.Length));
     int radius = windWidth / 2;
     double temp;
     outImage = new double[curBitmap.Width * curBitmap.Height];
     // 计算卷积
     for (int i = 0; i < curBitmap.Height; i++)
     {
          for (int j = 0; j < curBitmap.Width; j++)
          {
               temp = 0;
               for (int x = -radius; x <= radius; x++)
               {
                    for (int y = -radius; y <= radius; y++)
                    {
                         // 循环延拓
                         temp += inputImage[((Math.Abs(i + x)) % curBitmap.Height) *
                             curBitmap.Width + (Math.Abs(j + y)) % curBitmap.Width] *
                             mask[(x + radius) * windWidth + y + radius];
                    }
               }
               outImage[i * curBitmap.Width + j] = temp;
          }
     }
}
```

（4）编译并运行该段程序。打开图像后，单击“高斯算子”按钮，打开高斯算子对话框，如图 8.4 所示设置相关参数，则最终的边缘图像如图 8.5 所示。



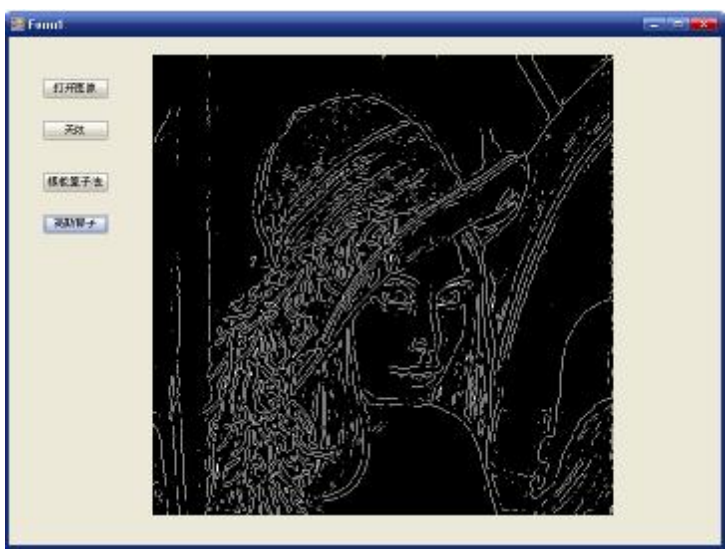


图 8.4 高斯算子对话框

图 8.5 高斯算术边缘检测图像

8.3 Canny 算子

8.3.1 Canny 边缘检测原理

Canny 算子是在边缘检测的 3 个指标（好的检测、好的定位和最小响应）和 3 个准则（信噪比准则、定位精度准则和单边缘响应准则）的基础上发展起来的一种很有效的边缘检测方法。

Canny 边缘检测方法分为 4 步。

（1）用高斯滤波器对图像进行平滑处理。

（2）利用一阶偏导算子（如 Roberts、Prewitt、Sobel 等）找到图像灰度沿着水平方向 G_x 和垂直方向 G_y 的偏导数，并按下列两式求出梯度的幅值 G 和方位 θ。在后面的程序中，一阶偏导算子我们采用的是性能最好的 Sobel 算子。

$$G = \sqrt{G_x^2 + G_y^2} \tag{8.11}$$

$$q = \arctan\left(\frac{G_y}{G_x}\right) \tag{8.12}$$

（3）对梯度幅值进行非极大值抑制，即找到局部梯度最大点。方法是在 3×3 的邻域窗内考察给定像素 p 与沿着梯度线方向的两个像素相比，如果 p 的梯度幅值不比这两个像素的梯度幅值更大，则令 p=0，否则保留原幅值。在这里，我们可以把梯度方向划分为水平方向、垂直方向、和正、－45°方向这 4 种方向来比较梯度幅值的强度。

（4）用双阈值算法检测和连接边缘。方法是凡大于高阈值 T_1 的一定是边缘；凡小于低阈值 T_2 的一定不是边缘；如果检测结果大于低阈值而又小于高阈值，则要看这个像素的邻接像素中是否有大于高阈值的边缘像素，如果有，则是边缘，否则不是。这往往是一个迭代的过程。

8.3.2 Canny 算子编程实例

该实例实现了被公认为性能最好的 Canny 边缘检测方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 8.5 所示。

表 8.5　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	canny
	Text	Canny 算子
	Location	37，242

（2）创建 1 个名为 canny 的 Windows 窗体，该窗体用于选择高斯滤波器的均方差和阈比的高、低两个阈值。在该窗体内添加 2 个 Button 控件、3 个 Label 控件和 3 个 TextBox 控件，其属性修改如表 8.6 所示。

表 8.6　所修改的属性

控　件	属　性	所修改内容
canny	Text	Canny 边缘检测
	ControlBox	False
	Size	276，249
button1	Name	start
	Text	确定
	Location	36，157
button2	Name	close
	Text	退出
	Location	154，157
label1	Text	均方差（σ）
	Location	34，28
label2	Text	高阈值（T1）
	Location	34，68
label3	Text	低阈值（T2）
	Location	34，108
textBox1	Name	sigma
	Text	2
	Location	129，25
	Size	100，21
textBox2	Name	threshHigh
	Text	100
	Location	129，65
	Size	100，21
textBox3	Name	threshLow
	Text	50
	Location	129，105
	Size	100，21

为这 2 个 Button 控件添加 Click 事件，再添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
     thresh[0] = Convert.ToByte(threshHigh.Text);
     thresh[1] = Convert.ToByte(threshLow.Text);
     this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
     this.Close();
}
```

```
public double GetSigma
{
     get
     {
         // 得到均方差
         return Convert.ToDouble(sigma.Text);
     }
}

public byte[] GetThresh
{
     get
     {
         // 得到两个阈值
         return thresh;
     }
}
```

其中 thresh 为 byte 型数组变量，它在构造函数内被初始化，代码如下：

```
thresh = new byte[] { 100, 50 };
```

（3）回到主窗体，为“Canny 算子”按钮添加 Click 事件，代码如下：

```
private void canny_Click(object sender, EventArgs e)
{
     if (curBitmap != null)
     {
           // 实例化canny
           canny cannyOp = new canny();

           if (cannyOp.ShowDialog() == DialogResult.OK)
           {
                 Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height)
                 System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                     System.Drawing.Imaging.ImageLockMode.ReadWrite,
                     curBitmap.PixelFormat);
                 IntPtr ptr = bmpData.Scan0;
                 int bytes = curBitmap.Width * curBitmap.Height;
                 byte[] grayValues = new byte[bytes];
                 System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes)

                 byte[] thresholding = new byte[2];
                 // 得到阈值
                 thresholding = cannyOp.GetThresh;
                 // 得到均方差
                 double sigma = cannyOp.GetSigma;

                 double[] tempArray;
                 double[] tempImage = new double[bytes];
                 double[] grad = new double[bytes];
                 byte[] aLabel = new byte[bytes];
                 double[] edgeMap = new double[bytes];
                 double gradX, gradY, angle;
                 // 高斯模板半径=3σ
                 int rad = Convert.ToInt16(Math.Ceiling(3 * sigma));
```

```
                    for (int i = 0; i < bytes; i++)
                         tempImage[i] = Convert.ToDouble(grayValues[i]);

                    // 调用高斯平滑处理方法
                    gaussSmooth(tempImage, out tempArray, sigma);

                    // Sobel 一阶偏导求梯度幅值和方向
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                         for (int j =0; j < curBitmap.Width; j++)
                         {
        // 水平方向梯度
        gradX = tempArray[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                    ((j + 1) % curBitmap.Width)] +
            2 * tempArray[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            tempArray[((i + 1) % curBitmap.Height) * curBitmap.Width +
                    ((j + 1) % curBitmap.Width)] -
            tempArray[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                    ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * tempArray[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
            tempArray[((i + 1) % curBitmap.Height) * curBitmap.Width +
                    ((Math.Abs(j - 1)) % curBitmap.Width)];

        // 垂直方向梯度
        gradY = tempArray[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                    ((Math.Abs(j - 1)) % curBitmap.Width)] +
            2 * tempArray[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            tempArray[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                    ((j + 1) % curBitmap.Width)] -
            tempArray[((i + 1) % curBitmap.Height) * curBitmap.Width +
                    ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * tempArray[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            tempArray[((i + 1) % curBitmap.Height) * curBitmap.Width +
                    ((j + 1) % curBitmap.Width)];

                         // 梯度和
                         grad[i * curBitmap.Width + j] = Math.Sqrt(gradX * gradX + gradY *
radY);
                              // 梯度方向（弧度）
                              angle = Math.Atan2(gradY, gradX);
                              // 4 个方向量化
                              if ((angle >= -1.178097 && angle < 1.178097) ||
                                        angle >= 2.748894 || angle < -2.748894)
                                   aLabel[i * curBitmap.Width + j] = 0;
                              else if ((angle >= 0.392699 && angle < 1.178097) ||
                                        (angle >= -2.748894 && angle < -1.963495))
                                   aLabel[i * curBitmap.Width + j] = 1;
                              else if ((angle >= -1.178097 && angle < -0.392699) ||
                                        (angle >= 1.963495 && angle < 2.748894))
                                   aLabel[i * curBitmap.Width + j] = 2;
                              else
                                   aLabel[i * curBitmap.Width + j] = 3;
                         }
                    }
```

```
                    // 非最大抑制
                    // 数组清零
                    Array.Clear(edgeMap, 0, bytes);
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                         for (int j = 0; j < curBitmap.Width; j++)
                         {
                             switch (aLabel[i * curBitmap.Width + j])
                             {
                                    case 3:
                                       // 水平方向
        if (grad[i * curBitmap.Width + j] >     grad[((Math.Abs(i - 1))%curBitmap.Height)
                    curBitmap.Width + j] && grad[i * curBitmap.Width + j] >
                    grad[((i + 1)%curBitmap.Height) * curBitmap.Width + j])
            edgeMap[i * curBitmap.Width + j] = grad[i * curBitmap.Width + j];
                                       break;
                                    case 2:
                                       // +45°方向
        if (grad[i * curBitmap.Width + j] > grad[((Math.Abs(i - 1)) % curBitmap.Height) *
                    curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)] &&
                    grad[i * curBitmap.Width + j] > grad[((i + 1) % curBitmap.Height) *
                    curBitmap.Width + ((j + 1) % curBitmap.Width)])
            edgeMap[i * curBitmap.Width + j] = grad[i * curBitmap.Width + j];
                                       break;
                                    case 1:
                                       // -45°方向
        if (grad[i * curBitmap.Width + j] > grad[((Math.Abs(i - 1)) % curBitmap.Height) *
                    curBitmap.Width + ((j + 1) % curBitmap.Width)] && grad[i * curBitmap.Width
                    j] > grad[((i + 1) % curBitmap.Height) * curBitmap.Width + (Math.Abs(j - 1)
                    curBitmap.Width)])
            edgeMap[i * curBitmap.Width + j] = grad[i * curBitmap.Width + j];
                                       break;
                                    case 0:
                                       // 垂直方向
        if (grad[i * curBitmap.Width + j] > grad[i * curBitmap.Width + (Math.Abs(j - 1) %
                    curBitmap.Width)] && grad[i * curBitmap.Width + j] > grad[i *
urBitmap.Width
                    + ((j + 1) % curBitmap.Width)])
            edgeMap[i * curBitmap.Width + j] = grad[i * curBitmap.Width + j];
                                       break;
                                    default:
                                       return;
                             }
                         }
                    }

                    // 双阈值算法
                    Array.Clear(grayValues, 0, bytes);
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                         for (int j =0; j < curBitmap.Width; j++)
                         {
                             if (edgeMap[i * curBitmap.Width + j] > thresholding[0])
```

```
                        {
                            grayValues[i * curBitmap.Width + j] = 255;
                            // 调用边缘点跟踪方法
                            traceEdge(i, j, edgeMap, ref grayValues, thresholding[1])
                        }
                    }
                }

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes)
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }

    /**********************************
    边缘跟踪，递归算法
    k：图像纵坐标
    l：图像横坐标
    inputImage：梯度图像
    outputImage：输出边缘图像
    thrLow：低阈值
    **********************************/
    private void traceEdge(int k, int l, double[] inputImage, ref byte[] outputImage, byte thrLow
    {
        // 8 邻域
        int[] kOffset = new int[] { 1, 1, 0, -1, -1, -1, 0, 1 };
        int[] lOffset = new int[] { 0, 1, 1, 1, 0, -1, -1, -1 };

        int kk, ll;
        for (int p = 0; p < 8; p++)
        {

            kk = k + kOffset[p];
            // 循环延拓
            kk = Math.Abs(kk) % curBitmap.Height;
            ll = l + lOffset[p];
            // 循环延拓
            ll = Math.Abs(ll) % curBitmap.Width;

            if (outputImage[ll * curBitmap.Width + kk] != 255)
            {
                if (inputImage[ll * curBitmap.Width + kk] > thrLow)
                {
                    outputImage[ll * curBitmap.Width + kk] = 255;
                    // 递归调用
                    traceEdge(ll, kk, inputImage, ref outputImage, thrLow);
                }
            }
        }
    }
```

其中方法 gaussSmooth()就是上一章 7.5 节高斯滤波中的方法，在本章不再赘述。

（4）编译并运行该段程序。打开 Canny 算子对话框，如图 8.6 所示设置相应参数，最终的边缘图像如图 8.7 所示。

图 8.6　Canny 算子对话框

图 8.7　Canny 边缘检测结果

8.4　形态学边缘检测

8.4.1　形态学边缘检测原理

形态学边缘检测方法也分为形态学梯度法和形态学拉普拉斯法两种。

$$\text{Gradient}[f(x,y)]=\{\text{dilate}[f(x,y),\mathbf{B}]-\text{erode}[f(x,y),\mathbf{B}]\}/2 \tag{8.13}$$

$$\text{Laplacian}[f(x,y)]=\{\text{dilate}[f(x,y),\mathbf{B}]+\text{erode}[f(x,y),\mathbf{B}]-2f(x,y)\}/2 \tag{8.14}$$

本小节仍然采用的是平坦结构元素。形态学梯度法相当于一阶导数，形态学拉普拉斯法相当于二阶导数。

8.4.2　形态学边缘检测编程实例

该实例实现了形态学梯度法和形态学拉普拉斯法的边缘检测方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 8.7 所示。

表 8.7　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	morph
	Text	灰度形态学
	Location	37，288

（2）创建 1 个名为 morphologic 的 Windows 窗体，该窗体用于选择结构元素以及阈值等其他参数。属性修改如表 7.6 一样，但窗体的 Size 变为（200，400），2 个按钮的位置也都向下平移 143 个像素长。另外还要再添加 1 个 GroupBox 控件、2 个 RadioButton 控件、1 个 Label 控件和 1 个 TextBox 控件，这些控件的属性修改如表 8.8 所示。

表 8.8 所修改的属性

控　件	属　性	所修改内容
label1	Text	阈值：
	Location	19，277
TextBox1	Name	thresh
	Text	25
	Location	74，274
	Size	100，21
groupBox1	Text	形态学方法
	Location	19，162
	Size	155，92
radioButton1	Name	gradient
	Text	形态学梯度
	Location	16，25
	Checked	True
radioButton2	Name	laplacian
	Text	形态学拉普拉斯
	Location	16，59

由 25 个 Button 控件组成的结构元素的 Click 事件及它们的数组变量 se 的 get 属性访问器代码与上一章第 3 节所介绍的一样，这里就不再赘述，其他的控件事件和 get 属性访问器的代码如下：

```
private void start_Click(object sender, EventArgs e)
{
     this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
     this.Close();
}

public bool GetMethod
{
     get
     {
          // 得到使用何种形态学方法
          return laplacian.Checked;
     }
}

public double GetThresh
{
     get
     {
          // 得到阈值
```

```
            return Convert.ToDouble(thresh.Text);
      }
}
```

（3）回到主窗体，为“灰度形态学”按钮控件添加 Click 事件，代码如下：

```
private void morph_Click(object sender, EventArgs e)
{
      if (curBitmap != null)
      {
            // 实例化morphologic
            morphologic grayMor = new morphologic();

            if (grayMor.ShowDialog() == DialogResult.OK)
            {
                  Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height)
                  System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                        System.Drawing.Imaging.ImageLockMode.ReadWrite,
                        curBitmap.PixelFormat);
                  IntPtr ptr = bmpData.Scan0;
                  int bytes = curBitmap.Width * curBitmap.Height;
                  byte[] grayValues = new byte[bytes];
                  System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes)

                  byte[] tempArray1 = new byte[bytes];
                  byte[] tempArray2 = new byte[bytes];
                  // 得到使用何种形态学方法
                  bool flag = grayMor.GetMethod;
                  // 得到阈值
                  double thresh = grayMor.GetThresh;
                  byte[] struEle = new byte[25];
                  // 得到结构元素
                  struEle = grayMor.GetStruction;
                  int temp;

                  // 调用灰度形态学膨胀运算方法
                  tempArray1 = grayDelation(grayValues, struEle,
                        curBitmap.Height, curBitmap.Width);
                  // 调用灰度形态学腐蚀运算方法
                  tempArray2 = grayErode(grayValues, struEle,
                        curBitmap.Height, curBitmap.Width);

                  for (int i = 0; i < bytes; i++)
                  {
                        if (flag == false)
                              // 形态学梯度
                              temp = (tempArray1[i] - tempArray2[i]) / 2;
                        else
                              // 形态学拉普拉斯
                              temp = (tempArray1[i] + tempArray2[i] - 2 * grayValues[i]) / 2

                        if (temp > thresh)
                              grayValues[i] = 255;
                        else
                              grayValues[i] = 0;
                  }
```

```
                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

其中 grayDelation 和 grayErode 分别为灰度形态学的膨胀运算和腐蚀运算方法，它们在.3.3 节已被定义过，这里不再赘述。

（4）编译并运行该段程序，单击“灰度形态学”按钮，打开灰度形态学边缘检测对话框如图 8.8 所示设置平坦结构元素和其他相应参数。

单击“确定”按钮，灰度形态学边缘检测结果就呈现出来，如图 8.9 所示。

图 8.8 灰度形态学边缘检测对话框

图 8.9 灰度形态学边缘检测结果

8.5 小波变换边缘检测

8.5.1 小波变换边缘检测原理

对于某些特殊的小波函数，小波变换的极大值对应于图像的边缘点。

在图像 $f(x,y)$中，设$\theta(x,y)$是一平滑函数，令：

$$\Psi^1(x,y)=\frac{\partial}{\partial x}q(x,y)，\quad \Psi^2(x,y)=\frac{\partial}{\partial y}q(x,y) \tag{8.15}$$

则$\psi^1(x,y)$和$\psi^2(x,y)$可作为小波函数。在尺度 2^j 下的小波变换是

$$W_{2^j}^1 f(x,y)=f*\Psi_{2^j}^1(x,y)，\quad W_{2^j}^2 f(x,y)=f*\Psi_{2^j}^2(x,y) \tag{8.16}$$

由于二维小波变换的两个分量 $W_{2^j}^1 f(x,y)$ 和 $W_{2^j}^2 f(x,y)$ 分别正比于图像 $f(x,y)$ 经 $q_{2^j}(x,y$ 平滑后的沿水平和垂直方向的偏导数，因此二维小波变换矢量就是梯度。当 $\theta(x,y)$ 取为高斯函数时，它与 Canny 算子在同一尺度下是等价的。剩下的部分与 Canny 算子也是一样的，只需沿着梯度方向检测小波变换系数模的局部极大值点，即可得到图像的边缘点，方法就是抑制非最大值和双阈值法。

小波函数的种类很多，也可以自己构造，但用于边缘检测的小波应满足一些条件。在这里我们选择二次样条小波函数，它的低通滤波器系数 $h(n)$ 和高通滤波器系数 $g(n)$ 如表 8.9 所示。

表 8.9　　二次样条小波滤波器系数

n	−1	0	1	2
h(n)	0.125	0.375	0.375	0.125
g(n)		−2	2	

通过对小波函数尺度 $2^j(j=0,1,2)$ 选取的不同，得到的边缘和抑制噪声的能力会有所不同。在尺度 2^j 下，它所对应的滤波器 h_j 和 g_j 分别表示在 h 和 g 的相邻系数之间插入 2^j-1 个零而得到的离散滤波器。在具体编程实现中，用高通滤波器对图像进行卷积得到边缘图像，用低通滤波器对图像进行卷积得到下一个尺度所需要的平滑图像。

8.5.2　小波变换边缘检测编程实例

该实例实现了小波变换的边缘检测方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 8.10 所示。

表 8.10　　所修改的属性

控　件	属　性	所修改内容
button1	Name	wavelet
	Text	小波变换
	Location	37，334

（2）创建 1 个名为 wvl 的 Windows 窗体，该窗体用于选择小波分解级数以及阈值。在窗体内添加 2 个 Button 控件、3 个 Label 控件、2 个 TextBox 控件和 1 个 NumericUpDown 控件，其属性修改如表 8.11 所示。

表 8.11　　所修改的属性

控　件	属　性	所修改内容
wvl	Text	小波变换边缘检测
	Size	270，245
	ControlBox	false
button1	Name	start
	Text	确定
	Location	42，151
button2	Name	close

	Text	退出
	Location	138，151

续表

控　　件	属　　性	所修改内容
numricUpDown1	Name	multiscale
	Size	75，21
	Location	138，34
	Maximum	3
textBox1	Name	highT
	Text	100
	Size	75，21
	Location	138，68
textBox2	Name	lowT
	Text	50
	Size	75，21
	Location	138，103
label1	Text	尺度：
	Location	76，36
label2	Text	高阈值（T1）
	Location	28，71
label3	Text	低阈值（T2）
	Location	28，106

为 2 个 Button 控件添加 Click 事件，并添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
     thresh[0] = Convert.ToByte(highT.Text);
     thresh[1] = Convert.ToByte(lowT.Text);
     this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
     this.Close();
}

public byte GetScale
{
     get
     {
          // 得到小波分解级数
          return Convert.ToByte(multiscale.Value);
     }
}

public byte[] GetThresh
{
```

```
        get
        {
            // 得到高、低两个阈值
            return thresh;
        }
    }
```

其中 thresh 为 byte 型数组变量，它在构造函数内被初始化，代码如下：

```
thresh = new byte[] { 100, 50 };
```

（3）回到主窗体，为“小波变换”按钮添加 Click 事件，代码如下：

```
private void wavelet_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化wvl
        wvl wavelet = new wvl();

        if (wavelet.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height)
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes)

            double[] tempArray1 = new double[bytes];
            double[] tempArray2 = new double[bytes];
            double[] tempArray3 = new double[bytes];
            double[] gradX = new double[bytes];
            double[] gradY = new double[bytes];

            // 得到小波分解级数
            byte multiscale = wavelet.GetScale;
            byte[] thresholding = new byte[2];
            // 得到双阈值
            thresholding = wavelet.GetThresh;

            for (int i = 0; i < bytes; i++)
                tempArray1[i] = Convert.ToDouble(grayValues[i]);

            for (int z = 0; z <= multiscale; z++)
            {
                double[] p = null;
                double[] q = null;
                // 确定各尺度的低通和高通滤波器系数
                switch (z)
                {
                    case 0:
                        p = new double[] { 0.125, 0.375, 0.375, 0.125 };
                        q = new double[] { -2, 2 };
                        break;
```

```
            case 1:
                p = new double[] { 0.125, 0, 0.375, 0, 0.375, 0, 0.125 };
                q = new double[] { -2, 0, 2 };
                break;
            case 2:
                p = new double[]     { 0.125, 0, 0, 0, 0.375, 0, 0, 0,
                    0.375, 0, 0, 0, 0.125 };
                q = new double[] { -2, 0, 0, 0, 2 };
                break;
            case 3:
                p = new double[]     { 0.125, 0, 0, 0, 0, 0, 0, 0, 0.375, 0
                   0, 0, 0, 0, 0, 0, 0.375, 0, 0, 0, 0, 0, 0, 0, 0.125 };
                q = new double[] { -2, 0, 0, 0, 0, 0, 0, 0, 2 };
                break;
            default:
                return;
        }

        int coff = Convert.ToInt16(Math.Pow(2, z) - 1);
        // 小波行变换
        for (int i = 0; i < curBitmap.Height; i++)
        {
            for (int j = 0; j < curBitmap.Width; j++)
            {
                double[] scl = new double[curBitmap.Width];
                double[] wvl = new double[curBitmap.Width];
                int temp;

                scl[j] = 0.0;
                wvl[j] = 0.0;
                // 低通滤波
                for (int x = -2 - 2 * coff; x < p.Length - 2 - 2 * coff; x++
                {
                    temp = (Math.Abs(j + x)) % curBitmap.Width;
                    scl[j] += p[1 + coff - x] *
                          tempArray1[i * curBitmap.Width + temp];
                }
                // 高通滤波
                for (int x = -1 - coff; x < q.Length - 1 - coff; x++)
                {
                    temp = (Math.Abs(j + x)) % curBitmap.Width;
                    wvl[j] += q[-x] * tempArray1[i * curBitmap.Width + temp]
                }

                // 平滑系数
                tempArray2[i * curBitmap.Width + j] = scl[j];
                // 水平梯度系数
                gradX[i * curBitmap.Width + j] = wvl[j];
            }
        }

        // 小波列变换
        for (int i = 0; i < curBitmap.Width; i++)
        {
```

```
                for (int j = 0; j < curBitmap.Height; j++)
                {
                    double[] scl = new double[curBitmap.Height];
                    double[] wvl = new double[curBitmap.Height];
                    int temp;

                    scl[j] = 0.0;
                    wvl[j] = 0.0;
                    // 低通滤波
                    for (int x = -2 - 2 * coff; x < p.Length - 2 - 2 * coff; x++
                    {
                        temp = (Math.Abs(j + x)) % curBitmap.Height;
                        scl[j] += p[1 + coff - x] *
                              tempArray2[temp * curBitmap.Width + i];
                    }
                    // 高通滤波
                    for (int x = -1 - coff; x < q.Length - 1 - coff; x++)
                    {
                        temp = (Math.Abs(j + x)) % curBitmap.Height;
                        wvl[j] += q[-x] * tempArray1[temp * curBitmap.Width + i]
                    }

                    // 平滑系数
                    tempArray3[j * curBitmap.Width + i] = scl[j];
                    // 垂直梯度系数
                    gradY[j * curBitmap.Width + i] = wvl[j];
                }
            }

            tempArray1 = (double[])tempArray3.Clone();
        }

        // 非最大值抑制
        double angle;
        for (int i = 0; i < curBitmap.Height; i++)
        {
            for (int j = 0; j < curBitmap.Width; j++)
            {
                // 梯度和
                tempArray1[i * curBitmap.Width + j] =
                    Math.Sqrt(gradX[i * curBitmap.Width + j] *
                    gradX[i * curBitmap.Width + j] + gradY[i * curBitmap.Width + j]
                    gradY[i * curBitmap.Width + j]);
                // 梯度方向（弧度）
                angle = Math.Atan2(gradY[i * curBitmap.Width + j],
                    gradX[i * curBitmap.Width + j]);
                if ((angle >= -1.178097 && angle < 1.178097) ||
                        angle >= 2.748894 || angle < -2.748894)
                    tempArray2[i * curBitmap.Width + j] = 0;
                else if ((angle >= 0.392699 && angle < 1.178097) ||
                        (angle >= -2.748894 && angle < -1.963495))
                    tempArray2[i * curBitmap.Width + j] = 1;
                else if ((angle >= -1.178097 && angle < -0.392699) ||
                        (angle >= 1.963495 && angle < 2.748894))
                    tempArray2[i * curBitmap.Width + j] = 2;
```

```
                    else
                        tempArray2[i * curBitmap.Width + j] = 3;
                }
            }
            Array.Clear(tempArray3, 0, bytes);
            for (int i = 0; i < curBitmap.Height; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    switch (Convert.ToInt16(tempArray2[i * curBitmap.Width + j]))
                    {
                        case 3:
                            // 水平方向
 if (tempArray1[i * curBitmap.Width + j] > tempArray1[((Math.Abs(i - 1)) %
       curBitmap.Height) * curBitmap.Width + j] && tempArray1[i * curBitmap.Width +
       j] > tempArray1[((i + 1) % curBitmap.Height) * curBitmap.Width + j])
    tempArray3[i * curBitmap.Width + j] = tempArray1[i * curBitmap.Width + j];
                            break;
                        case 1:
                            // +45°方向
 if (tempArray1[i * curBitmap.Width + j] > tempArray1[((Math.Abs(i - 1)) %
       curBitmap.Height) * curBitmap.Width + (Math.Abs(j - 1) %
       curBitmap.Width)] && tempArray1[i * curBitmap.Width + j] >
       tempArray1[((i + 1) % curBitmap.Height) * curBitmap.Width + ((j + 1) %
       curBitmap.Width)])
    tempArray3[i * curBitmap.Width + j] = tempArray1[i * curBitmap.Width + j];
                            break;
                        case 2:
                            // -45°方向
if (tempArray1[i * curBitmap.Width + j] > tempArray1[((Math.Abs(i - 1)) %
         curBitmap.Height) * curBitmap.Width + ((j + 1) % curBitmap.Width)] &&
         tempArray1[i * curBitmap.Width + j] > tempArray1[((i + 1) %
         curBitmap.Height) * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)]
    tempArray3[i * curBitmap.Width + j] = tempArray1[i * curBitmap.Width + j];
                            break;
                        case 0:
                            // 垂直方向
if (tempArray1[i * curBitmap.Width + j] > tempArray1[i * curBitmap.Width +
         (Math.Abs(j - 1) % curBitmap.Width)] && tempArray1[i * curBitmap.Width + j]
         tempArray1[i * curBitmap.Width + ((j + 1) % curBitmap.Width)])
      tempArray3[i * curBitmap.Width + j] = tempArray1[i * curBitmap.Width + j];
                            break;
                        default:
                            return;
                    }
                }
            }

            // 双阈值法
            Array.Clear(grayValues, 0, bytes);
            for (int i = 0; i < curBitmap.Height; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    if (tempArray3[i * curBitmap.Width + j] > thresholding[0])
```

```
                        {
                            grayValues[i * curBitmap.Width + j] = 255;
                            // 调用边缘跟踪办法
                            traceEdge(i, j, tempArray3, ref grayValues, thresholding[1]);
                        }
                    }
                }

                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

该段程序中的 traceEdge 方法在 Canny 算子一节中已给出，这里不再赘述。

（4）编译并运行该段程序，在打开的小波变换边缘检测对话框中，如图 8.10 所示设置好的参数。

单击“确定”按钮，最终所得到的边缘图像如图 8.11 所示。

图 8.10　小波变换边缘检测对话框

图 8.11　小波变换边缘检测结果

8.6　金字塔方法

8.6.1　金字塔方法原理

金字塔方法是多分辨率图像处理的一个最基本的方法，小波变换的多分辨率分析就是在高斯-拉普拉斯金字塔方法的启发下提出来的。

图像处理的金字塔方法是将原始图像分解成不同空间分辨率的子图像，高分辨率（尺度

较大）的子图像放在下层，低分辨率（尺度较小）的图像放在上层，从而形成一个金字塔的形状，如图 8.12 所示。

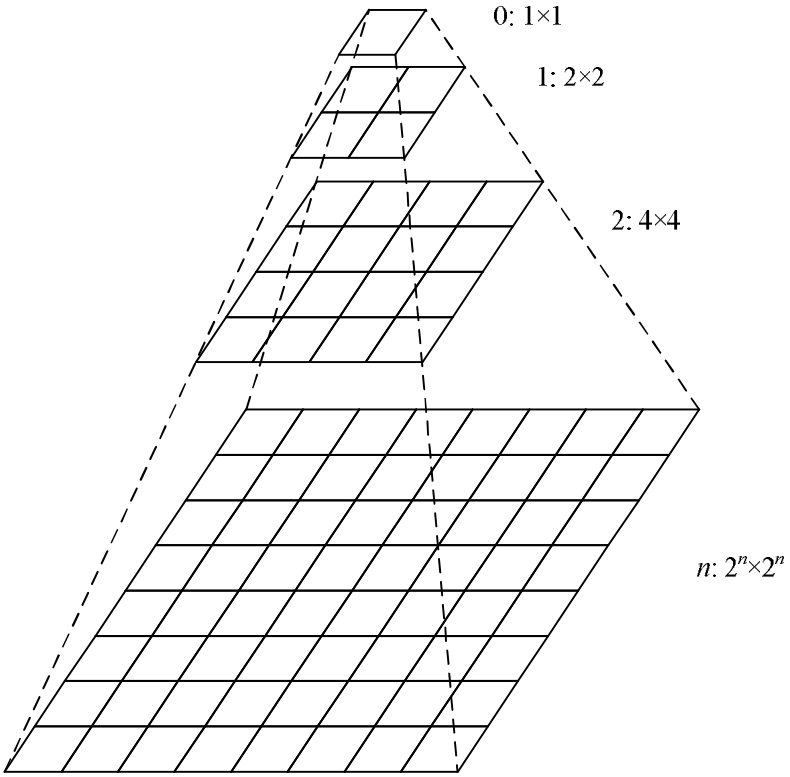


图 8.12　图像处理的金字塔模型

高斯-拉普拉斯金字塔方法也可用于图像的边缘检测。它的基本原理是，先对图像 $f_l(i,j)$ 进行高斯卷积，$g(i,j)$为高斯卷积核，下标 l 表示的是金字塔的层数：

$$f'_{l+1}(i,j) = g(i,j) * f_l(i,j) \tag{8.17}$$

然后求出两个图像之间的差：

$$h_{l+1}(i,j) = f_l(i,j) - f'_{l+1}(i,j) \tag{8.18}$$

再对卷积后的图像进行隔行隔列采样，以起到降低分辨率的作用：

$$f_{l+1}(i,j) = f'_{l+1}(2i,2j) \tag{8.19}$$

该过程称为高斯塔分解，如图 8.13 所示。

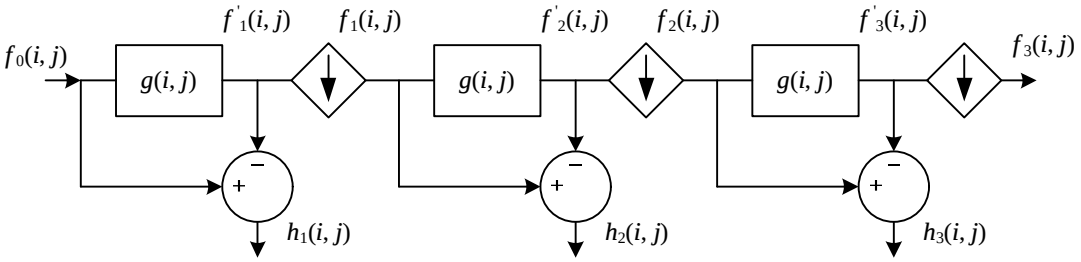


图 8.13　高斯塔分解模型

然后由高斯塔重构拉普拉斯塔，它的过程就是高斯塔分解的逆过程。首先内插放大图像

$$f_l'(i,j)=\begin{cases} f_l(i/2,j/2) & i,j=0,2,4,\mathbf{L} \\ 0 & \text{其他} \end{cases} \tag{8.20}$$

然后再进行高斯卷积计算：

$$f_{l-1}(i,j)=g(i,j)*f_l'(i,j)+h_l(i,j) \tag{8.21}$$

拉普拉斯塔重构如图 8.14 所示，最终所得到的图像经过阈值处理后即为边缘图像。

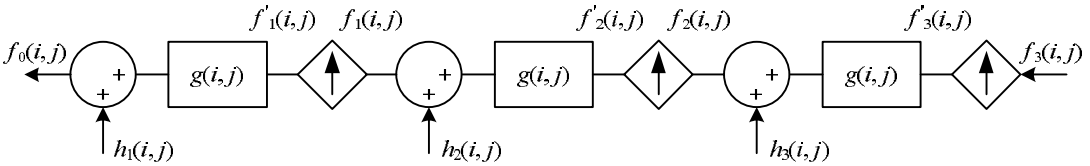


图 8.14 拉普拉斯塔重构模型

8.6.2 金字塔方法编程实例

该实例实现了金字塔方法的边缘检测方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 8.12 所示。

表 8.12 所修改的属性

控 件	属 性	所修改内容
button1	Name	pyramid
	Text	金字塔
	Location	37，380

（2）创建 1 个名为 glp 的 Windows 窗体，该窗体用于选择金字塔分解级数、高斯均方以及阈值。在该窗体内添加 2 个 Button 控件、3 个 Label 控件、2 个 TextBox 控件和 1 个 NumericUpDown 控件，其属性修改如表 8.13 所示。

表 8.13 所修改的属性

控 件	属 性	所修改内容
glp	Text	金字塔边缘检测
	ControlBox	False
	Size	236，255
button1	Name	start
	Text	确定
	Location	23，168
button2	Name	close
	Text	退出
	Location	116，168
label1	Text	分解层次：

	Location	33，30

续表

控　件	属　性	所修改内容
label2	Text	高斯均方差：
	Location	21，77
label3	Text	阈值：
	Location	57，124
numericUpDown1	Name	level
	Location	113，28
	Size	78，21
	Value	3
	Minimum	1
textBox1	Name	sigma
	Text	1.2
	Location	113，74
	Size	78，21
textBox2	Name	thresh
	Text	25
	Location	113，121
	Size	78，21

为 2 个 Button 控件添加 Click 事件，添加 3 个 get 属性访问器，并改写构造函数，代码如下：

```
public glp(int maxLevel)
{
     InitializeComponent();
     // 最大分解级数
     level.Maximum = maxLevel;
}

private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public byte GetLevel
{
    get
    {
         // 得到分解级数
         return Convert.ToByte (level.Value);
    }
```

```
}

public double GetThresh
{
    get
    {
         // 得到阈值
         return Convert.ToDouble (thresh.Text);
    }
}

public double GetSigma
{
    get
    {
         // 得到均方差
         return Convert.ToDouble(sigma.Text);
    }
}
```

构造函数中的形参是主窗体传给该窗体的参数，用来定义 level 控件的一个参数。

（3）回到主窗体，为“金字塔”按钮添加 Click 事件，代码如下：

```
private void pyramid_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        int series = Convert.ToInt16(Math.Log(curBitmap.Width, 2));
        // 实例化glp，并把最大金字塔分解级数传递给子窗体
        glp pyramid = new glp(series);

        if (pyramid.ShowDialog() == DialogResult.OK)
        {
             Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
             System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                 System.Drawing.Imaging.ImageLockMode.ReadWrite,
                 curBitmap.PixelFormat);
             IntPtr ptr = bmpData.Scan0;
             int bytes = curBitmap.Width * curBitmap.Height;
             byte[] grayValues = new byte[bytes];
             System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

             // 得到阈值
             double thresh = pyramid.GetThresh;
             // 得到分解级数
             byte level = pyramid.GetLevel;
             // 得到均方差
             double sigma = pyramid.GetSigma;

             // 定义变量
             double[][] pyramidImage = new double[level + 1][];
             double[][] passImage = new double[level + 1][];
             int levelBytes = bytes;
             for (int k = 0; k < level + 1; k++)
             {
                  passImage[k] = new double[levelBytes];
```

```
            pyramidImage[k] = new double[levelBytes];
            levelBytes = levelBytes / 4;
        }
        for (int i = 0; i < bytes; i++)
            pyramidImage[0][i] = Convert.ToDouble(grayValues[i]);

        // 高斯塔分解
        for (int k = 0; k < level; k++)
        {
            double[] tempImage = null;
            // 调用高斯平滑方法
            gaussSmooth(pyramidImage[k], out tempImage, sigma);
            int coff = pyramidImage[k].Length;
            for (int i = 0; i < coff; i++)
            {
                passImage[k][i] = pyramidImage[k][i] - tempImage[i];
                int div = i / Convert.ToInt16(Math.Sqrt(coff));
                int rem = i % Convert.ToInt16(Math.Sqrt(coff));
                // 隔行隔列抽取采样点
                if (div % 2 == 0 && rem % 2 == 0)
                {
                    int j = (int)((div / 2) * Math.Sqrt(pyramidImage[k + 1].Length)
                          rem / 2);
                    pyramidImage[k + 1][j] = tempImage[i];
                }
            }
        }

        // 拉普拉斯塔重构
        for (int k = level - 1; k >= 0; k--)
        {
            int coff = pyramidImage[k].Length;
            // 内插采样点
            for (int i = 0; i < coff; i++)
            {
                int div = i / Convert.ToInt16(Math.Sqrt(coff));
                int rem = i % Convert.ToInt16(Math.Sqrt(coff));
                int j = (int)((div / 2) * Math.Sqrt(pyramidImage[k + 1].Length)
                      rem / 2);
                if (div % 2 == 0 && rem % 2 == 0)
                      pyramidImage[k][i] = pyramidImage[k + 1][j];
                else
                      pyramidImage[k][i] = 0;
            }
            double[] tempImage = null;
            // 调用高斯平滑方法
            gaussSmooth(pyramidImage[k], out tempImage, 1);
            for (int i = 0; i < coff; i++)
            pyramidImage[k][i] = tempImage[i] + passImage[k][i];
        }

        // 调用零交叉点方法，提取边缘点
        zerocross(ref pyramidImage[0], out grayValues,thresh);

        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
```

```
                curBitmap.UnlockBits(bmpData);
            }

            Invalidate();
        }
    }
```

其中 gaussSmooth 方法和 zerocross 方法这里就不再赘述。

（4）编译并运算该段程序，单击“金字塔”按钮，按照如图 8.15 所示设置相关参数。单击“确定”按钮，最终得到的图像如图 8.16 所示。

图 8.15 金字塔边缘检测对话框

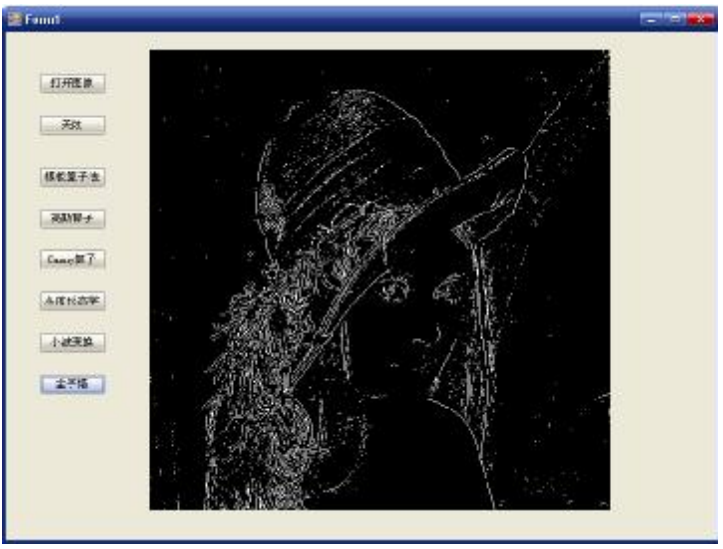


图 8.16 金字塔边缘检测结果

8.7 小结

边缘检测虽然是图像分隔的一部分，但它一直是图像处理中的研究热点问题之一，因此它单独作为一章来进行介绍是很有必要的。本章所介绍的这几种方法都是目前最常用的边缘检测方法。与其他图像处理方法类似，没有一种边缘检测方法可以应用于所有的图像边缘提取中，只有根据具体情况，选择某种方法，才能得到最佳的效果。

第 10 章　图像压缩编码

近年来，随着计算机通信技术的迅速发展，特别是多媒体网络技术的兴起，图像压缩编码已受到了人们越来越多的关注。

图像压缩是数据压缩技术在数字图像上的应用，它的目的是减少图像数据中的冗余信息，从而用更加高效的格式存储和传输数据。

图像压缩可以是有损数据压缩，也可以是无损数据压缩。对于如绘制的技术图、图表或者漫画则优先使用无损压缩，这是因为有损压缩方法，尤其是在低的位数条件下将会带来压缩失真；如医疗图像或者用于存档的扫描图像等这些有价值的内容的压缩也尽量选择无损压缩方法。有损方法非常适合于自然的图像，因为这些图像的微小损失是可以接受的（有时是无法感知的），这样就可以大幅度地减小位数。

.0.1　哈夫曼编码

哈夫曼编码（Huffman Coding）是一种用于无损数据压缩的熵编码算法。

设图像像素灰度级集合为$\{d_1,d_2,\cdots,d_m\}$，其对应的概率分别为 $p(d_1), p(d_2),\cdots, p(d_m)$，则图像熵为：

$$H(d)=-\sum_{i=1}^{m} p(d_i)\log_2 p(d\ ) \tag{10.1}$$

平均码长度是衡量任何特定码性能的准则，它定义为：

$$l=\sum_{i=1}^{m} l_i\, p(d_i) \tag{10.2}$$

其中 l_i 为灰度级 d_i 所对应的码字的长度。

编码效率通常用下式表示：

$$h=\frac{H(d)}{l}\times 100\% \tag{10.3}$$

10.1.1　哈夫曼编码原理

哈夫曼编码是一种长度不均匀的、平均码率可以接近信息源熵值的一种编码，它是一种有效的编码方法。它的基本思想是：对于出现概率大的信息，采用短字长的码，对于出现概率小的信息用长字长的码，以达到缩短平均码上，从而实现数据压缩的目的。

它的过程如下。

（1）统计出图像中每个灰度值出现的概率，并按照从小到大的顺序排列。

（2）每一次选出概率最小的两个值，将它们相加，形成的新频率值和其他频率值形成一个新的频率集合。

（3）重复第（2）步，直到最后得到频率和为 1。

（4）分配码字，对上述步骤反过来逐步向前进行编码，每一步有两个分支各赋予一个二进制码，对概率大的赋予码元 0，对概率小的赋予码元 1（或相反）。

10.1.2　哈夫曼编码编程实例

该实例是对所显示的图像进行哈夫曼编码计算，并把编码结果显示在打开的对话框内。

（1）创建 1 个“模板主窗体”，在该窗体内添加 1 个 Button 控件，其属性修改如表 10.1 所示。

表 10.1　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	huffman
	Text	哈夫曼编码
	Location	37，150

（2）创建 1 个名为 huffmamCode 的 Windows 窗体，该窗体用于显示哈夫曼编码结果。在该窗体内添加 1 个 Button 控件、1 个 ListView 控件和 6 个 Label 控件，其属性修改如表 10.2 所示。

表 10.2　　所修改的属性

控　　件	属　　性	所修改内容
HuffmanCode	Size	468，408
	Text	哈夫曼编码
	ControlBox	False
listView1	Name	huffmanData
	Size	391，222
	Location	32，24
	View	Details
label1	Text	图像熵：
	Location	30，274
label2	Text	平均码长

	Location	239，274

续表

控　　件	属　　性	所修改内容
label3	Text	编码效率
	Location	30，321
label4	Name	Entropy
	Location	101，274
	BackColor	HighlightText
	BorderStyle	Fixed3D
label5	Name	AverLength
	Location	310，274
	BackColor	HighlightText
	BorderStyle	Fixed3D
label6	Name	Efficiency
	Location	101，321
	BackColor	HighlightText
	BorderStyle	Fixed3D
button1	Name	Close
	Text	退出
	Location	324，321

为 Button 控件添加 Click 事件，为该窗体添加 Load 事件，并改写构造函数，代码如下

```
public huffmanCode(Bitmap bmp)
{
    InitializeComponent();
    bmpHuffman = bmp;
    // 定义列表的表头
    huffmanData.Columns.Add("灰度值", 50);
    huffmanData.Columns.Add("出现概率", 130, HorizontalAlignment.Center);
    huffmanData.Columns.Add("哈夫曼编码", 130, HorizontalAlignment.Center);
    huffmanData.Columns.Add("码字长", 60, HorizontalAlignment.Center);
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void huffmanCode_Load(object sender, EventArgs e)
{
    Rectangle rect = new Rectangle(0, 0, bmpHuffman.Width, bmpHuffman.Height);
    System.Drawing.Imaging.BitmapData bmpData = bmpHuffman.LockBits(rect,
         System.Drawing.Imaging.ImageLockMode.ReadWrite, bmpHuffman.PixelFormat);
    IntPtr ptr = bmpData.Scan0;
    int bytes = bmpHuffman.Width * bmpHuffman.Height;
    byte[] grayValues = new byte[bytes];
    System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

    double[] hist = new double[256];
```

```
double[] feq = new double[256];
int[] hMap = new int[256];
string[] strCode = new string[256];
double entr = 0.0;
double codeLeng = 0.0;

// 赋初值
for (int i = 0; i < 256; i++)
{
     hist[i] = 0.0;
     hMap[i] = i;
}

// 计算图像的直方图
for (int i = 0; i < bytes; i++)
{
     hist[grayValues[i]] += 1;
}

// 计算图像灰度值的出现概率
for (int i = 0; i < 256; i++)
{
     hist[i] /= (double)bytes;
     feq[i] = hist[i];
}

// 按概率大小进行冒泡法排序，并记录下排序后的映射关系
double temp = 0;
for (int i = 0; i < 255; i++)
{
     for (int j = 0; j < 255 - i; j++)
     {
         if (hist[j] > hist[j + 1])
          {
               temp = hist[j];
               hist[j] = hist[j + 1];
               hist[j + 1] = temp;

               for (int k = 0; k < 256; k++)
               {
                    if (hMap[k] == j)
                        hMap[k] = j + 1;
                    else if (hMap[k] == j + 1)
                        hMap[k] = j;
               }
          }
     }
}

// 哈夫曼编码
for (int i = 0; i < 255; i++)
{
     // 找到第一个不为零的概率所对应的灰度值
     if (hist[i] == 0.0)
          continue;

     for (int j = i; j < 255; j++)
```

```
        {
            for (int k = 0; k < 256; k++)
            {
                // 编码
                if (hMap[k] == j)
                    strCode[k] = "1" + strCode[k];
                else if (hMap[k] == j + 1)
                    strCode[k] = "0" + strCode[k];
            }

            // 合并最小的两个概率值
            hist[j + 1] += hist[j];

            // 更新映射关系
            for (int k = 0; k < 256; k++)
            {
                if (hMap[k] == j)
                    hMap[k] = j + 1;
            }

            // 对余下的概率再重新进行排序
            for (int m = j + 1; m < 255; m++)
            {
                if (hist[m] > hist[m + 1])
                {
                    temp = hist[m];
                    hist[m] = hist[m + 1];
                    hist[m + 1] = temp;

                    for (int k = 0; k < 256; k++)
                    {
                        if (hMap[k] == m)
                            hMap[k] = m + 1;
                        else if (hMap[k] == m + 1)
                            hMap[k] = m;
                    }
                }
                else
                    break;
            }
        }

        // 跳出循环
        break;
    }

    // 填写表单
    for (int i = 0; i < 256; i++)
    {
        ListViewItem li = new ListViewItem();
        // 灰度级
        li.Text = i.ToString();
        // 出现概率
        li.SubItems.Add(feq[i].ToString("F10"));
        // 编码
        li.SubItems.Add(strCode[i]);
        if (strCode[i] == null)
            li.SubItems.Add("0");
```

```
            else
            {
                // 码长
                li.SubItems.Add(strCode[i].Length.ToString());
                // 计算平均码长
                codeLeng += feq[i] * strCode[i].Length;
                // 计算图像熵
                entr -= feq[i] * Math.Log(feq[i], 2);
            }
            huffmanData.Items.Add(li);
        }

        entropy.Text = entr.ToString("F8");
        averLength.Text = codeLeng.ToString("F8");
        // 编码效率
        double codeEff = entr / codeLeng * 100;
        efficiency.Text = codeEff.ToString("F4");

        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
        bmpHuffman.UnlockBits(bmpData);
    }
```

其中 bmpHuffman 为 System.Drawing.Bitmap 型变量。

（3）回到主窗体，为“哈夫曼编码”按钮添加 Click 事件，代码如下：

```
    private void huffman_Click(object sender, EventArgs e)
    {
        if (curBitmap != null)
        {
            // 实例化huffmanCode，并把位图数据传递给从窗体
            huffmanCode huffmanCoding = new huffmanCode(curBitmap);
            huffmanCoding.ShowDialog();
        }
    }
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开该图后，单击“哈夫曼编程”按钮，打开哈夫曼编码对话框，如图 10.1 所示，该窗体不仅显示出了哈夫曼编码，而且为了说明该编码的效率，也显示出了图像熵、平均码长和编码效率。

图 10.1 哈夫曼编码结果

.0.2　香农编码

10.2.1　香农编码原理

香农（Shannon）编码也是基于熵编码方法。它按下列步骤进行。

（1）将图像灰度级按出现的概率由大到小顺序排列。

（2）按下式计算出各概率对应的码字长度 t_i：

$$-\log_2 P_i \leqslant t_i < -\log_2 P_i + 1 \qquad (10.4$$

其中 P_i 为灰度级为 i 的出现概率。

（3）计算各概率对应的累加概率 a_i，即：

$$a_1=0$$
$$a_2=P_1$$
$$a_3=P_2+a_2=P_2+P_1$$
$$\cdots$$
$$a_i=P_{i-1}+P_{i-2}+\cdots+P_1$$
$$\cdots$$

（4）把各个累加概率由十进制转换成二进制数。

（5）将二进制表示的累加概率去掉多于⑵步中计算的 t_i 的尾数，即获得各个灰度级的码字

10.2.2　香农编码编程实例

该实例如上一个实例一样，在打开的对话框内显示香农编码。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 10.3 所示。

表 10.3　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	shannon
	Text	香农编码
	Location	37，196

（2）创建 1 个名为 shannonC 的 Windows 窗体，该窗体用于显示香农编码结果。在该窗体内添加 1 个 Button 控件、1 个 ListView 控件和 6 个 Label 控件，其属性修改如表 10.4 所示

表 10.4　　所修改的属性

控　　件	属　　性	所修改内容
shannonC	Size	468，408
	Text	香农编码
	ControlBox	False
listView1	Name	shannonData
	Size	391，222
	Location	32，24

	View	Details

续表

控　　件	属　　性	所修改内容
label1	Text	图像熵：
	Location	30，274
label2	Text	平均码长
	Location	239，274
label3	Text	编码效率
	Location	30，321
label4	Name	entropy
	Location	101，274
	BackColor	HighlightText
	BorderStyle	Fixed3D
label5	Name	averLength
	Location	310，274
	BackColor	HighlightText
	BorderStyle	Fixed3D
label6	Name	efficiency
	Location	101，321
	BackColor	HighlightText
	BorderStyle	Fixed3D
button1	Name	close
	Text	退出
	Location	324，321

为 Button 控件添加 Click 事件，为该窗体添加 Load 事件，并改写构造函数，代码如下

```
public shannonC(Bitmap bmp)
{
    InitializeComponent();
    bmpShannon = bmp;
    // 定义列表的表头
    shannonData.Columns.Add("灰度值", 50);
    shannonData.Columns.Add("出现概率", 130, HorizontalAlignment.Center);
    shannonData.Columns.Add("香农编码", 130, HorizontalAlignment.Center);
    shannonData.Columns.Add("码字长", 60, HorizontalAlignment.Center);
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void shannonC_Load(object sender, EventArgs e)
{
    Rectangle rect = new Rectangle(0, 0, bmpShannon.Width, bmpShannon.Height);
    System.Drawing.Imaging.BitmapData bmpData = bmpShannon.LockBits(rect,
        System.Drawing.Imaging.ImageLockMode.ReadWrite, bmpShannon.PixelFormat);
```

```
IntPtr ptr = bmpData.Scan0;
int bytes = bmpShannon.Width * bmpShannon.Height;
byte[] grayValues = new byte[bytes];
System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

double[] hist = new double[256];
double[] feq = new double[256];
int[] hMap = new int[256];
string[] strCode = new string[256];
double entr = 0.0;
int[] codeLeng = new int[256];
double averCL = 0.0;
double[] accP = new double[256];

// 赋初值
for (int i = 0; i < 256; i++)
{
     hist[i] = 0.0;
     hMap[i] = i;
}

// 计算图像的直方图
for (int i = 0; i < bytes; i++)
{
     hist[grayValues[i]] += 1;
}

// 计算图像灰度值的出现概率及其码字长度
for (int i = 0; i < 256; i++)
{
     hist[i] /= (double)bytes;
     feq[i] = hist[i];
     if (hist[i] > 0.0)
          codeLeng[i] = Convert.ToInt16(1 - Math.Log(hist[i], 2));
     else
          codeLeng[i] = 0;
}

double fTemp = 0.0;
int iTemp = 0;
// 冒泡排序法，按概率由大到小排序，并记录下排序后的映射关系
for (int i = 0; i < 255; i++)
{
     for (int j = 0; j < 255 - i; j++)
     {
          if (hist[j] < hist[j + 1])
          {
              fTemp = hist[j];
              hist[j] = hist[j + 1];
              hist[j + 1] = fTemp;

              iTemp = hMap[j];
              hMap[j] = hMap[j + 1];
              hMap[j + 1] = iTemp;
          }
```

```
            }
        }

        // 香农编码
        for (int i = 0; i < 255; i++)
        {
            // 找到第一个为零的概率所对应的灰度值
            if (hist[i] != 0.0)
                continue;

            // 计算第一个累加概率及其编码
            accP[0] = 0.0;
            for (int k = 0; k < codeLeng[hMap[0]]; k++)
                strCode[hMap[0]] += "0";

            // 计算其余的编码
            for (int j = 1; j < i; j++)
            {
                // 计算累加概率
                for (int k = 0; k < j; k++)
                {
                    accP[j] += hist[k];
                }

                // 把十进制小数转换为二进制，从而得到其编码
                for (int k = 0; k < codeLeng[hMap[j]]; k++)
                {
                    accP[j] *= 2;
                    if (accP[j] >= 1)
                    {
                        strCode[hMap[j]] += "1";
                        accP[j] -= 1;
                    }
                    else
                        strCode[hMap[j]] += "0";
                }
            }

            // 跳出循环
            break;
        }

        // 填写表单
        for (int i = 0; i < 256; i++)
        {
            ListViewItem li = new ListViewItem();
            // 灰度级
            li.Text = i.ToString();
            // 出现概率
            li.SubItems.Add(feq[i].ToString("F10"));
            // 编码
            li.SubItems.Add(strCode[i]);
            if (strCode[i] == null)
                li.SubItems.Add("0");
            else
```

```
            {
                // 码长
                li.SubItems.Add(codeLeng[i].ToString());
                // 计算平均码长
                averCL += feq[i] * strCode[i].Length;
                // 计算图像熵
                entr -= feq[i] * Math.Log(feq[i], 2);
            }
            shannonData.Items.Add(li);
        }

        entropy.Text = entr.ToString("F8");
        averLength.Text = averCL.ToString("F8");
        // 编码效率
        double codeEff = entr / averCL * 100;
        efficiency.Text = codeEff.ToString("F4");

        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
        bmpShannon.UnlockBits(bmpData);
    }
```

其中 bmpShannon 为 System.Drawing.Bitmap 型变量。

（3）回到主窗体，为“香农编码”按钮添加 Click 事件，代码如下：

```
private void shannon_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化shanonC，并把位图数据传递给从窗体
        shannonC shannonCoding = new shannonC(curBitmap);
        shannonCoding.ShowDialog();
    }
}
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开该图后，单击“香农编码”按钮，打开香农编码对话框，如图 10.2 所示。

图 10.2 香农编码结果

.0.3　香农-弗诺编码

10.3.1　香农-弗诺编码原理

香农-弗诺编码同上两种方法一样，属于熵编码方法的一种，它的步骤如下。

（1）计算出每个灰度级出现的概率，并按由小到大顺序排列。

（2）从序列中某个位置将序列分成两个子序列，并尽量使这两个序列概率和近似相等，给前面一个子序列赋值为 1，后面一个子序列赋值为 0；

（3）重复步骤⑵，直到各个子序列不能再分为止；

（4）分配码字，将每个像素所属子序列的值串起来，这样就得到了各个像素的香农-弗编码。

需要说明的是，如果各个灰度值出现的概率正好为 2^{-N_i} 时，采用香农-弗诺编码，它的效率可以高达 100%。

10.3.2　香农-弗诺编码编程实例

该实例与前两种实例实现的功能相同。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 10.5 所示。

表 10.5　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	shanFan
	Text	香-弗编码
	Location	37，242

（2)创建 1 个名为 shannonFannon 的 Windows 窗体，该窗体用于显示香农-弗诺编码结果在该窗体内添加 1 个 Button 控件、1 个 ListView 控件和 6 个 Label 控件，其属性修改如表 10.6 所示。

表 10.6　　所修改的属性

控　　件	属　　性	所修改内容
shannonFannon	Size	468，408
	Text	香农-弗诺编码
	ControlBox	False
listView1	Name	sfData
	Size	391，222
	Location	32，24
	View	Details
label1	Text	图像熵
	Location	30，274

续表

控　件	属　性	所修改内容
label2	Text	平均码长
	Location	239，274
label3	Text	编码效率
	Location	30，321
label4	Name	entropy
	Location	101，274
	BackColor	HighlightText
	BorderStyle	Fixed3D
label5	Name	averLength
	Location	310，274
	BackColor	HighlightText
	BorderStyle	Fixed3D
label6	Name	efficiency
	Location	101，321
	BackColor	HighlightText
	BorderStyle	Fixed3D
button1	Name	close
	Text	退出
	Location	324，321

为 Button 控件添加 Click 事件，为该窗体添加 Load 事件，并改写构造函数，代码如下

```
public shannonFannon(Bitmap bmp)
{
    InitializeComponent();
    bmpSF = bmp;
    // 定义列表的表头
    sfData.Columns.Add("灰度值", 50);
    sfData.Columns.Add("出现概率", 130, HorizontalAlignment.Center);
    sfData.Columns.Add("香农-弗诺编码", 130, HorizontalAlignment.Center);
    sfData.Columns.Add("码字长", 60, HorizontalAlignment.Center);
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void shannonFannon_Load(object sender, EventArgs e)
{
    Rectangle rect = new Rectangle(0, 0, bmpSF.Width, bmpSF.Height);
    System.Drawing.Imaging.BitmapData bmpData = bmpSF.LockBits(rect,
        System.Drawing.Imaging.ImageLockMode.ReadWrite, bmpSF.PixelFormat);
    IntPtr ptr = bmpData.Scan0;
    int bytes = bmpSF.Width * bmpSF.Height;
    byte[] grayValues = new byte[bytes];
    System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
```

```
double[] hist = new double[256];
double[] feq = new double[256];
int[] hMap = new int[256];
string[] strCode = new string[256];
double entr = 0.0;
double codeLeng = 0.0;
bool[] bFlag = new bool[256];
double feqSum = 0.0;
double feqTotal = 0.0;

// 赋初值
for (int i = 0; i < 256; i++)
{
     hist[i] = 0.0;
     hMap[i] = i;
     bFlag[i] = false;
}

// 计算图像的直方图
for (int i = 0; i < bytes; i++)
{
     hist[grayValues[i]] += 1;
}

// 计算图像灰度值的出现概率
for (int i = 0; i < 256; i++)
{
     hist[i] /= (double)bytes;
     feq[i] = hist[i];
     feqTotal += feq[i];
}

double fTemp = 0;
int iTemp = 0;
// 冒泡排序法，按概率由小到大排序，并记录下排序后的映射关系
for (int i = 0; i < 255; i++)
{
     for (int j = 0; j < 255 - i; j++)
     {
          if (hist[j] > hist[j + 1])
          {
              fTemp = hist[j];
              hist[j] = hist[j + 1];
              hist[j + 1] = fTemp;

              iTemp = hMap[j];
              hMap[j] = hMap[j + 1];
              hMap[j + 1] = iTemp;
          }
     }
}

// 香农-弗诺编码
for (int i = 0; i < 255; i++)
```

```
{
     // 找到第一个不为零的概率所对应的灰度值
     if (hist[i] == 0.0)
          continue;

     int gCount = 0;

     while (gCount < 256)
     {
          gCount = i;

          for (int j = i; j < 256; j++)
          {
              if (bFlag[j] == false)
              {
                    // 该灰度值尚未完成全部编码

                   feqSum += hist[j];
                   if (feqSum > feqTotal / 2)
                       // 某段的前部分赋为0
                       strCode[hMap[j]] = strCode[hMap[j]] + "0";
                   else
                       // 某段的后部分赋为1
                       strCode[hMap[j]] = strCode[hMap[j]] + "1";

                   // 完成某段编码后的处理
                   if (feqTotal == feqSum)
                   {
                       feqSum = 0.0;
                       feqTotal = 0.0;
                       int k;

                       // 判断是否完成某一循环的全部编码
                       // 是，则重新开始循环编码
                       // 否，则继续下一段的编码
                       if (j == 255)
                            k = i;
                       else
                            k = j + 1;
                       int m;
                       // 重新计算累加概率
                       for (m = k; m < 256; m++)
                       {
     // 判断是否是同一段内的编码
     if ((strCode[hMap[m]].Substring((strCode[hMap[m]].Length) - 1) !=
           (strCode[hMap[k]].Substring((strCode[hMap[k]].Length) - 1))||
           (strCode[hMap[m]].Length) != (strCode[hMap[k]].Length)))
                                break;
                         feqTotal += hist[m];
                       }
                       // 判断某灰度值是否已完成编码，即段长为1
                       if (k + 1 == m)
                            bFlag[k] = true;
                   }
              }
```

```
                else
                {
                    // 该灰度值已完成全部编码，则从下一个灰度值开始重新分段计算

                    // 用于跳出循环
                    gCount++;

                    feqSum = 0.0;
                    feqTotal = 0.0;
                    int k;
                    if (j == 255)
                        k = i;
                    else
                        k = j + 1;
                    int m;
                    for (m = k; m < 256; m++)
                    {
if ((strCode[hMap[m]].Substring((strCode[hMap[m]].Length) - 1) !=
    (strCode[hMap[k]].Substring((strCode[hMap[k]].Length) - 1))||
    (strCode[hMap[m]].Length) != (strCode[hMap[k]].Length)))
                            break;
                        feqTotal += hist[m];
                    }
                    if (k + 1 == m)
                        bFlag[k] = true;
                }
            }
        }
        break;
    }

    // 填写表单
    for (int i = 0; i < 256; i++)
    {
        ListViewItem li = new ListViewItem();
        // 灰度值
        li.Text = i.ToString();
        // 出现概率
        li.SubItems.Add(feq[i].ToString("F10"));
        // 编码
        li.SubItems.Add(strCode[i]);
        if (strCode[i] == null)
            li.SubItems.Add("0");
        else
        {
            // 码长
            li.SubItems.Add(strCode[i].Length.ToString());
            // 计算平均码长
            codeLeng += feq[i] * strCode[i].Length;
            // 图像熵
            entr -= feq[i] * Math.Log(feq[i], 2);
        }
        sfData.Items.Add(li);
    }
```

```
        entropy.Text = entr.ToString("F8");
        averLength.Text = codeLeng.ToString("F8");
        // 编码效率
        double codeEff = entr / codeLeng * 100;
        efficiency.Text = codeEff.ToString("F4");

        System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
        bmpSF.UnlockBits(bmpData);
    }
```

其中 bmpSF 为 System.Drawing.Bitmap 型变量。

（3）回到主窗体，为“香-弗编码”按钮添加 Click 事件，代码如下：

```
    private void shanFan_Click(object sender, EventArgs e)
    {
        if (curBitmap != null)
        {
            // 实例化shannonFannon，并把位图数据传递给从窗体
            shannonFannon shFCoding = new shannonFannon(curBitmap);
            shFCoding.ShowDialog();
        }
    }
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开该图后，单击“香-弗编码”按钮，打开香农-弗诺编码对话框，如图 10.3 所示。

图 10.3 香农-弗诺编码结果

10.4 行程编码

10.4.1 行程编码原理

行程编码（Run Length Encoding，RLE）是一种利用空间冗余度压缩图像的方法。对某

是具有相同灰度级成片连续出现的图像，行程编码是一种高效的编码方法。特别对二值图像效果尤为显著。

假定有一幅灰度图像，某一行的像素灰度值为：00000000111888882222555，用 RLE 编码方法得到的代码为：**8**0**3**1**5**8**4**2**3**5。代码中用黑体表示的数字是行程长度，黑体字后面的数字代表像素的灰度值。RLE 所能获得的压缩主要是取决于图像本身特点，如果图像中具有相同灰度值（或颜色）的图像块越大，图像块数目越小，获得的压缩比就越高，反之，压缩比就越小。因此，RLE 对颜色丰富的自然图像直接进行编码，效果并不理想，如果直接使用 RLE 编码方法，不仅不能压缩图像数据，反而可能使原来的图像数据变得更大。

早期的 PCX 图像格式所采样的压缩方法就是基于行程编码方法。对于灰度图像，它的编码原则如下。

- 图像数据以字节为单位进行编码。
- 对于连续重复的像素值，统计其连续出现的次数 *n*（最大取值为 63），先存入长度信息（*n* | 0xC0，“|”表示逻辑与运算），然后再存入像素灰度值。如果连续次数超过 63 次，则必须分多次处理。例如，连续 132 个 0x66，编码时必须分 3 次处理，处理结果为：0xFF0x660xFF0x660xC60x66。
- 当遇到不重复的灰度值时，如果该灰度值小于 0xC0，则直接存入该灰度值。否则首先存入一个 0xC1，然后再存入该灰度值。这样做是为了避免该灰度值被误认为是数据长度。

行程解码的思路正好与编码相反，首先读取一个字节，如果该字节不大于 0xC0，则是图像的像素值，否则表示的是行程信息，它的低 6 位表示后面连续出现的字节数量 *n*，读取下一个字节并重复 *n* 次作为图像像素的 *n* 个像素值。

10.4.2 行程编码编程实例

本实例借鉴 PCX 图像格式的编码方法，自定义一种新的图像格式（zrle），把经过行程编码后的数据放入该文件内。它只对 8 位灰度图像进行处理，因此该图像格式的头文件很简单，只需要 6 位字节即可。第一位用于识别 zrle 文件格式，必须为 0x0B；第二位用于表示是 8 位灰度图像，必须为 0x08；第三、四位表示图像的高；后两位表示图像的宽。

该实例完成了行程编码和行程解码。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 10.7 所示。

表 10.7 所修改的属性

控　件	属　性	所修改内容
button1	Name	rle
	Text	行程运算
	Location	37，288

（2）创建 1 个名为 rleCode 的 Windows 窗体，该窗体用于选择是行程编码（生成 zrle 文件）还是行程解码（把 zrle 文件恢复成原图像）。在该窗体内添加 2 个 Button 控件、1 个 GroupBox 控件和 2 个 RadioButton 控件，其属性修改如表 10.8 所示。

表 10.8 所修改的属性

控　件	属　性	所修改内容
rleCode	Size	357，231
	Text	行程运算
	ControlBox	False
button1	Name	start
	Text	确定
	Location	61，141
button2	Name	close
	Text	退出
	Location	196，141
groupBox1	Text	行程编、解码
	Size	260，81
	Location	42，30
radioButton1	Name	rleCoding
	Text	行程编码
	Location	23，33
	Checked	True
radioButton2	Name	rleDecoding
	Text	行程解码
	Location	154，33

为 2 个 Button 控件添加 Click 事件，并添加 1 个 get 属性访问器，代码如下：

```
private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

public bool GetDorC
{
    get
    {
        // 得到是编码还是解码
        return rleDecoding.Checked;
    }
}
```

（3）回到主窗体，为“行程运算”按钮添加 Click 事件，代码如下：

```
private void rle_Click(object sender, EventArgs e)
{
    // 实例化rleCode
    rleCode rleDC = new rleCode();
```

```
if (rleDC.ShowDialog() == DialogResult.OK)
{
     // 得到是编码还是解码
     bool rleDorC = rleDC.GetDorC;

     if (rleDorC == false)
     {
          // 行程编码

          if (curBitmap == null)
          {
              MessageBox.Show("请先打开图像! ");
              return;
          }

          Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
          System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
              System.Drawing.Imaging.ImageLockMode.ReadWrite,
              curBitmap.PixelFormat);
          IntPtr ptr = bmpData.Scan0;
          int bytes = curBitmap.Width * curBitmap.Height;
          byte[] grayValues = new byte[bytes];
          System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
          curBitmap.UnlockBits(bmpData);

          // 预留行程编码空间
          char[] rleData = new char[bytes * 2];
          int rleLength = 0;
          int position = 0;
          char oldData, newData;
          byte sameCount = 0;
          // 头文件
          char[] rleHeader = new char[6];

          // zrle 文件标识
          rleHeader[0] = Convert.ToChar(0x0B);
          // 8 位灰度图像标识
          rleHeader[1] = Convert.ToChar(0x08);
          // 图像高
          rleHeader[2] = Convert.ToChar(curBitmap.Height / 0xFF);
          rleHeader[3] = Convert.ToChar(curBitmap.Height % 0xFF);
          // 图像宽
          rleHeader[4] = Convert.ToChar(curBitmap.Width / 0xFF);
          rleHeader[5] = Convert.ToChar(curBitmap.Width % 0xFF);

          // 以每行为单位进行编码
          for (int i = 0; i < curBitmap.Height; i++)
          {
              position = curBitmap.Width * i;
              oldData = Convert.ToChar(grayValues[position]);
              // 相同个数计数
              sameCount = 1;
              for (int j = 1; j < curBitmap.Width; j++)
              {
```

```
                position++;
                newData = Convert.ToChar(grayValues[position]);

                // 前后灰度值相同，并且相同个数是小于63
                if ((newData == oldData) && (sameCount < 63))
                    sameCount++;
                else
                {
                // 前后灰度值相同并且相同个数不小于63，或者灰度值不小于0xC0
                    if ((sameCount > 1) || (oldData >= 0xC0))
                    {
                        // 保存码长，即灰度值重复的个数
                        rleData[rleLength] = Convert.ToChar(sameCount | 0xC0);
                        // 保存像素的灰度值
                        rleData[rleLength + 1] = oldData;
                        rleLength += 2;
                    }
                    // 其他情况
                    else
                    {
                        // 保存像素的灰度值
                        rleData[rleLength] = oldData;
                        rleLength++;
                    }

                    // 更新数据
                    oldData = newData;
                    sameCount = 1;
                }
            }

            // 计算每行的最后一部分编码
            if ((sameCount > 1) || (oldData >= 0xC0))
            {
                rleData[rleLength] = Convert.ToChar(sameCount | 0xC0);
                rleData[rleLength + 1] = oldData;
                rleLength += 2;
            }
            else
            {
                rleData[rleLength] = oldData;
                rleLength++;
            }
        }

        // 保存文件对话框
        SaveFileDialog sf = new SaveFileDialog();
        // 设置标题
        sf.Title = "保存文件";
        // 设置文件保存类型（zrle）
        sf.Filter = "自定义图像格式(*.zrle)|*.zrle";
        // 如果用户没有输入扩展名，自动追加后缀
        sf.AddExtension = true;
```

```
            // 如果用户单击了保存按钮
            if (sf.ShowDialog() == DialogResult.OK)
            {
                // 实例化一个文件流
                FileStream fs = new FileStream(sf.FileName, FileMode.Create);
                // 实例化一个StreamWriter
                StreamWriter sw = new StreamWriter(fs);

                // 开始写入
                // 头文件
                sw.Write(rleHeader, 0, 6);
                // 行程编码数据
                sw.Write(rleData, 0, rleLength);

                // 清空缓冲区
                sw.Flush();
                // 关闭流
                sw.Close();
                fs.Close();
            }
        }
        else
        {
            // 行程解码

            // 打开文件对话框
            OpenFileDialog of = new OpenFileDialog();
            // 设置标题
            of.Title = "打开文件";
            // 设置文件保存类型（zrle）
            of.Filter = "自定义图像格式(*.zrle)|*.zrle";
            // 如果用户单击了保存按钮
            if (of.ShowDialog() == DialogResult.OK)
            {
                // 实例化一个文件流
                FileStream fs = new FileStream(of.FileName, FileMode.Open);
                // 实例化一个StreamWrit
                StreamReader sr = new StreamReader(fs);

                // 读取文件类型zrle 标识
                int rleTemp = sr.Read();
                if (rleTemp != 0x0B)
                {
                    //不是
                    MessageBox.Show("不是zrle 文件格式! ");
                    sr.Close();
                    fs.Close();
                    return;
                }
                // 读取8 位灰度图像标识
                rleTemp = sr.Read();
                if (rleTemp != 0x08)
                {
```

```
                    MessageBox.Show("本实例只能处理8 位灰度图像! ");
                    sr.Close();
                    fs.Close();
                    return;
               }

               int heightRle = sr.Read();
               rleTemp = sr.Read();
               // 图像高
               heightRle = heightRle * 255 + rleTemp;
               int widthRle = sr.Read();
               rleTemp = sr.Read();
               // 图像宽
               widthRle = widthRle * 255 + rleTemp;

               // 读取行程编码数据
               string rleString = sr.ReadToEnd();
               // 设置图像大小空间
               byte[] rleData = new byte[heightRle * widthRle];

               int count = 0;
               int sameCount = 0;
               int totalCount = 0;
               char bPix;
               // 以图像行为单位开始解码
               for (int i = 0; i < heightRle; i++)
               {
                    // 行计数
                    count = 0;
                    while (count < widthRle)
                    {
                         // 读取一个字节
                         bPix = rleString[totalCount];
                         // 行程编码数据个数加1
                         totalCount++;
                         if ((bPix & 0xC0) == 0xC0)
                         {
                              // 有重复的数据

                              // 解码重复像素的个数
                              sameCount = bPix & 0x3F;
                              // 解码像素的灰度值
                              bPix = rleString[totalCount];
                              totalCount++;
                              // 保存sameCount 个像素
                              for (int j = 0; j < sameCount; j++)
                              {
                                  rleData[i * widthRle + count + j] =
onvert.ToByte(bPix);

                              }
                              count += sameCount;
                         }
                         else
```

```
                    {
                        // 没有重复的数据

                        // 保存一个像素
                        rleData[i * widthRle + count] = Convert.ToByte(bPix);
                        count++;
                    }
                }
            }

            // 把解码后的图像显示在主窗体内
            curBitmap = new Bitmap(widthRle, heightRle,
                System.Drawing.Imaging.PixelFormat.Format8bppIndexed);
            // 设置为8位灰度图像
            System.Drawing.Imaging.ColorPalette cp = curBitmap.Palette;
            for (int i = 0; i < 256; i++)
            {
                cp.Entries[i] = Color.FromArgb(i, i, i);
            }
            curBitmap.Palette = cp;

            Rectangle rectRle = new Rectangle(0, 0, widthRle, heightRle);
            System.Drawing.Imaging.BitmapData rData =curBitmap.LockBits(rectRle,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = rData.Scan0;
            // 赋值
            System.Runtime.InteropServices.Marshal.Copy((byte[])rleData, 0,
                ptr, widthRle * heightRle);
            curBitmap.UnlockBits(rData);
            // 关闭流
            sr.Close();
            fs.Close();

            Invalidate();
        }
    }
}
```

为能够成功编译上段程序，还必须在该文件的头部添加一个用于读写文件的命名空间，代码如下：

```
using System.IO;
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开该图后，单击“行程运算”按钮，打开行程运算对话框，选择“行程编码”按钮，如图 10.4 所示，单击“确定”按钮后，会出现保存文件”对话框，选择好路径和文件名后，就完成了把所打开的图像经过行程编码后保存为 zrle 文件的过程。

图 10.4　“行程运算”对话框

在任何情况下，单击“行程运算”按钮，打开如图 10.4 所示的对话框，选择“行程解码”按钮后，单击“确定”按钮，则打开了“打开文件”对话框，找到刚才保存的那个 zrle 文件，则 zrle 文件被解码成图像并显示在主窗体内，如图 10.5 所示。可以

看出，原图被不失真地还原了回来。

图 10.5　行程解码结果

.0.5　LZW 编码

LZW 算法是一种应用最为广泛的压缩算法，它是由 3 位发明、改进和实现该算法的学者的
生氏的首个字母命名的。目前，GIF、PDF 等众多文件格式中都使用了 LZW 算法或其衍生算法

10.5.1　LZW 编码原理

LZW 编码的基本原理是首先建立一个字符串表，把每一个第一次出现的字符串放入串表
中，并用一个数字来表示，这个数字与此字符串在串表中的位置有关，并将这个数字存入压
缩文件中，如果这个字符串再次出现时，即可用表示它的数字来代替，并再次将这个数字存
入文件中，压缩完成后将串表丢弃。它的适用范围是原始数据串中最好有大量的子串多次重
复出现，重复的越多，压缩效果越好。反之则越差，甚至有可能不减反增。

LZW 编码算法示例：

如果 8 位图像的灰度值为：255,24,54,255,24,255,255,24,5,123…

首先初始化串表为：0 1 2 3 …254 255。

然后对图像的灰度值编码。

（1）读取第一个灰度值为 255，在串表里查找，255 已经存在，不做处理。

（2)读取第二个灰度值 24,此时的前缀为上一个读取的灰度值 255,并形成了当前的 Ent
为（255，24）（Entry 为前缀+后缀（即当前值)），在串表中不存在，那么把它存入到串表中
并标记为 256，然后输出前缀，后缀 24 作为下一次的前缀。

（3）读取第三个灰度值 54，当前 Entry=（24，54），串表内没有该值，记录该 Entry 到
串表内，并标记为 257，输出 24，后缀变前缀。

（4）读取第四个灰度值 255，Entry=（54，255），串表内没有该值，记录该 Entry，并
记为 258，输出 54，更新前缀。

（5）读取第五个灰度值 24，Entry=（255，24），串表内有该值，并且该值在串表内被已为 256，则此时不输出任何值，并把 Entry 作为前缀。

⑹ 读取第六个灰度值 255，Entry=（255，24，255），串表内没有该值，记录该 Entry，并标记为 259，输出 256，更新前缀。

一直处理到最后一个灰度值为止。

LZW 解码过程和编码过程正好相反，其原则是将所有编码后的码字转换成对应的字符串，重新生成串表，然后依次输出对应的字符串即可。需要说明的是：Entry=（前缀+后缀）当码字对应的字符串在串表内时，输出该字符串在串表内对应的字符，并把 Entry 存入串表此时的前缀为上一次读取的字符串，后缀为当前读取的字符串的第一个字符；当码字对应的字符串不在串表内时，则把 Entry 输出，并存入串表内，此时前缀为上一次读取的字符串，而后缀为前缀的第一个字符。

无论是编码还是解码，串表都是在编码或解码过程中动态生成的。

10.5.2 LZW 编码编程实例

该实例与上一个实例相似，自定义一个新的图像格式（zlzw），把经过 LZW 编码后的数据放入该文件内。它只对 8 位灰度图像进程处理，因此该图像格式的头文件也很简单，只需要 5 个部分，第一个部分用于表示为 zlzw 文件，第二个部分用于表示是 8 位灰度图像，第三、四部分用于表示为图像的高和宽，第五部分用于表示该文件后面数据的大小。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 10.9 所示。

表 10.9 所修改的属性

控　　件	属　　性	所修改内容
button1	Name	lzw
	Text	LZW 运算
	Location	37，334

（2）创建一个名为 lzwCode 的 Windows 窗体，该窗体用于选择是 LZW 编码（生成 zlzw 文件）还是 LZW 解码（把 zlzw 文件恢复成原图像）。在该窗体内添加 2 个 Button 控件、1 个 GroupBox 控件和 2 个 RadioButton 控件，其属性修改如表 10.10 所示。

表 10.10 所修改的属性

控　　件	属　　性	所修改内容
lzwCode	Size	357，231
	Text	LZW 运算
	ControlBox	False
button1	Name	start
	Text	确定
	Location	61，141
button2	Name	close
	Text	退出

	Location	196，141

续表

控　件	属　性	所修改内容
groupBox1	Text	LZW 编、解码
	Size	260，81
	Location	42，30
radioButton1	Name	lzwCoding
	Text	LZW 编码
	Location	23，33
	Checked	True
radioButton2	Name	lzwDecoding
	Text	LZW 解码
	Location	154，33

为两个 Button 控件添加 Click 事件，并添加 1 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetDorC
{
    get
    {
        // 得到是编码还是解码
        return lzwDecoding.Checked;
    }
}
```

（3）回到主窗体，为“LZW 运算”按钮添加 Click 事件，代码如下：

```
private void lzw_Click(object sender, EventArgs e)
{
    // 实例化lzwCode
    lzwCode lzwDC = new lzwCode();

    if (lzwDC.ShowDialog() == DialogResult.OK)
    {
        // 得到是编码还是解码
        bool lzwDorC = lzwDC.GetDorC;

        if (lzwDorC == false)
        {
            // LZW 编码

            if (curBitmap == null)
```

```
{
    MessageBox.Show("请先打开图像! ");
    return;
}

Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
    System.Drawing.Imaging.ImageLockMode.ReadWrite,
    curBitmap.PixelFormat);
IntPtr ptr = bmpData.Scan0;
int bytes = curBitmap.Width * curBitmap.Height;
byte[] grayValues = new byte[bytes];
System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
curBitmap.UnlockBits(bmpData);

// 定义数组列表和哈希表
// 输出压缩数据的数组列表
ArrayList codingData = new ArrayList();
// 串表
Hashtable codingDic = new Hashtable();

// 初始化
codingData.Clear();
codingDic.Clear();

// 哈希表中的键值是由灰度值与“|”组成的字符串，“|”用于识别每个不同像
// 素；映射为字符串所对应的索引号
for (int i = 0; i < 256; i++)
    codingDic.Add(i.ToString() + "|", i.ToString());
// 索引号
int valueDic = 256;

// 前缀
string prefix = null;
// 后缀
string suffix = null;
// Entry
string fullcode = null;
// 前缀序号
string prefixIndex = null;

// 开始编码
prefix = grayValues[0].ToString() + "|";
for (int i = 1; i < bytes; i++)
{
    suffix = grayValues[i].ToString() + "|";
    fullcode = prefix + suffix;
    int dicLength = codingDic.Count;

    if (codingDic.ContainsKey(fullcode))
    {
        // 串表内有Entry
        // 更新前缀
        prefix += suffix;
    }
```

```
            else
            {
                // 串表内没有Entry
                // 从串表内取出索引号
                prefixIndex = codingDic[prefix].ToString();
                // 添加到输出数组
                codingData.Add(prefixIndex);
                // 添加Entry到串表
                codingDic.Add(fullcode, valueDic.ToString());
                // 索引号加1
                valueDic++;
                // 更新前缀
                prefix = suffix;
            }
        }
        // 最后一个灰度值的处理
        prefixIndex = codingDic[prefix].ToString();
        codingData.Add(prefixIndex);

        // 保存文件对话框
        SaveFileDialog sf = new SaveFileDialog();
        // 设置标题
        sf.Title = "保存文件";
        // 设置文件保存类型
        sf.Filter = "自定义图像格式(*.zlzw)|*.zlzw";
        // 如果用户没有输入扩展名，自动追加后缀
        sf.AddExtension = true;
        // 如果用户单击了保存按钮
        if (sf.ShowDialog() == DialogResult.OK)
        {
            // 实例化一个文件流
            FileStream fs = new FileStream(sf.FileName, FileMode.Create);
            // 实例化一个BinaryWriter
            BinaryWriter bw = new BinaryWriter(fs);

            // 开始写入
            int dataLength = codingData.Count;
            // zlzw文件格式标识
            bw.Write("zlzw");
            // 8位灰度图像标识
            bw.Write("8b");
            // 图像高
            bw.Write(curBitmap.Height.ToString());
            // 图像宽
            bw.Write(curBitmap.Width.ToString());
            // 数据量大小
            bw.Write(dataLength.ToString());
            // 写入编码数据
            for (int i = 0; i < dataLength; i++)
                bw.Write(codingData[i].ToString());

            // 清空缓冲区
            bw.Flush();
            // 关闭流
            bw.Close();
```

```
                fs.Close();
            }
        }
        else
        {
            // LZW 解码

            // 打开文件对话框
            OpenFileDialog of = new OpenFileDialog();
            // 设置标题
            of.Title = "打开文件";
            // 设置文件保存类型
            of.Filter = "自定义图像格式(*.zlzw)|*.zlzw";
            // 如果用户单击了保存按钮
            if (of.ShowDialog() == DialogResult.OK)
            {
                // 实例化一个文件流
                FileStream fs = new FileStream(of.FileName, FileMode.Open);
                // 实例化一个BinaryReader
                BinaryReader br = new BinaryReader(fs);

                // 定义数组列表和哈希表
                // 压缩数据解码后的输出数组
                ArrayList deCodingData = new ArrayList();
                // 串表
                Hashtable deCodingDic = new Hashtable();

                // 初始化
                deCodingData.Clear();
                deCodingDic.Clear();
                // 哈希表中的映射为由灰度值与“|”组成的字符串，“|”用于识别每个不
                // 同的像素；键值为字符串所对应的索引号
                for (int i = 0; i < 256; i++)
                deCodingDic.Add(i.ToString(), i.ToString() + "|");
                // 键值
                int valueDic = 256;

                // 前缀
                string prefix = null;
                // 后缀
                string suffix = null;
                // Entry
                string fullcode = null;
                string readData, readOldData;

                // 定义读取文件的起始位置
                br.BaseStream.Position = 0;
                readData = br.ReadString();
                // 判断是否是zlzw 文件格式
                if (readData != "zlzw")
                {
                    MessageBox.Show("不是zlzw 文件格式! ");
                    return;
                }
                readData = br.ReadString();
```

```
// 判断是否是8位灰度图像
if (readData != "8b")
{
     MessageBox.Show("本实例只能处理8位灰度图像! ");
     return;
}

int heightLzw, widthLzw;
// 图像高
heightLzw = Convert.ToInt32(br.ReadString());
// 图像宽
widthLzw = Convert.ToInt32(br.ReadString());
// 数据量大小
int dataL = Convert.ToInt32(br.ReadString());

// 开始解码
readData = br.ReadString();
// 输出该值对应的字符串到解码列表
deCodingData.Add(deCodingDic[readData]);
readOldData = readData;
for (int i = 1; i < dataL; i++)
{
     readData = br.ReadString();
     if (deCodingDic.ContainsKey(readData))
     {
          // 串表内有该值

          // 输出该值对应的字符串到解码列表
          deCodingData.Add(deCodingDic[readData]);
          // 赋前缀
          prefix = deCodingDic[readOldData].ToString();
          // 赋后缀
          string tempFix = deCodingDic[readData].ToString();
          // 通过判断“|”，在字符串中找到第一个字符给后缀
          for (int j = 0; j < tempFix.Length; j++)
          {
               if (tempFix[j] == '|')
               {
                    suffix = tempFix.Substring(0, j + 1);
                    break;
               }
          }
          fullcode = prefix + suffix;
          // 添加串表
          deCodingDic.Add(valueDic.ToString(), fullcode);
          // 键值加1
          valueDic++;
     }
     else
     {
          // 串表内没有该值

          // 赋前缀
          prefix = deCodingDic[readOldData].ToString();
          // 赋后缀，通过判断“|”，在字符串中找到第一个字符给后缀
```

```
                for (int j = 0; j < prefix.Length; j++)
                {
                    if (prefix[j] == '|')
                    {
                        suffix = prefix.Substring(0, j + 1);
                        break;
                    }
                }
                fullcode = prefix + suffix;
                // 输出该值对应的字符串到解码列表
                deCodingData.Add(fullcode);
                // 添加串表
                deCodingDic.Add(valueDic.ToString(), fullcode);
                // 键值加1
                valueDic++;
            }
            // 更新数据
            readOldData = readData;
        }

        // 把解码后的图像显示在主窗体内
        curBitmap = new Bitmap(widthLzw, heightLzw,
            System.Drawing.Imaging.PixelFormat.Format8bppIndexed);
        // 设置为8位灰度图像
        System.Drawing.Imaging.ColorPalette cp = curBitmap.Palette;
        for (int i = 0; i < 256; i++)
        {
            cp.Entries[i] = Color.FromArgb(i, i, i);
        }
        curBitmap.Palette = cp;

        Rectangle rectLzw = new Rectangle(0, 0, widthLzw, heightLzw);
        System.Drawing.Imaging.BitmapData lData = curBitmap.LockBits
            (rectLzw, System.Drawing.Imaging.ImageLockMode.ReadWrite,
            curBitmap.PixelFormat);
        IntPtr ptr = lData.Scan0;
        int bytesD = widthLzw * heightLzw;
        byte[] grayLzw = new byte[bytesD];

        // 通过判断“|”，分离出解码列表中的字符串成各个像素灰度值
        int k = 0;
        for (int i = 0; i < deCodingData.Count; i++)
        {
            string decodingStr = deCodingData[i].ToString();
            int m = 0;
            for (int j = 0; j < decodingStr.Length; j++)
            {
                if (decodingStr[j] == '|')
                {
                    int subL = j - m;
                    grayLzw[k] = Convert.ToByte(decodingStr.Substring
                     (m, subL));
                    m = j + 1;
                    k++;
                }
```

```
                    }
                }

                System.Runtime.InteropServices.Marshal.Copy(grayLzw, 0, ptr, bytesD)
                curBitmap.UnlockBits(lData);
                // 关闭流
                br.Close();
                fs.Close();

                Invalidate();
            }
        }
    }
}
```

为了能够使用哈希表这个数据结构，还必须在该文件的头部添加一个命名空间，代码如下

```
using System.Collections;
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开该图后，单击“LZW 运算”按钮，打开 LZW 运算对话框，选择“LZW 编码”按钮，如图 10.6 所示，单击“确定”按钮后，会出现“保存文件”对话框，选择好路径和文件名后，就完成了把所打开的图像经过 LZW 编码后保存为 zlzw 文件的过程。

图 10.6　“LZW 运算”对话框

在任何情况下，单击“LZW 运算”按钮，打开如图 10.6 所示的对话框，选择“LZW 解码”按钮后，单击“确定”按钮，则打开了“打开文件”对话框，找到刚才保存的那个 zlzw 文件，此时 zlzw 文件被解码成图像并显示在主窗体内，如图 10.7 所示。可以看出，原图被不失真地还原了回来。


图 10.7　LZW 解码结果

.0.6 预测编码

预测编码数据压缩技术建立在信号数据的相关性上。它根据某一模型，利用以前的样本值对新样本进行预测，以此减少数据在时间和空间上的相关性，从而达到压缩数据的目的。预测编码可分为线性预测和非线性预测两类，它可以在一幅图像内进行，也可以在多幅图像之间进行。线性预测编码通常称为差分脉冲编码调制法（DPCM），本节重点介绍该方法。

10.6.1 DPCM 原理

DPCM 是利用相邻像素间具有相近的值的性质，把少许的差异信息一个接一个地传递下去进行编码。

如图 10.8 所示为图像中某一小块像素，X 为当前所要处理的像素，X_1、X_2、X_3、X_4、X_5、X_6 等为其相邻的像素。DPCM 就是利用这些相邻的像素值来预测 X 的值。DPCM 是对预测值 X' 与实际值 X 间的差 $\triangle X$（$=X-X'$）进行编码，这个差越小，预测值越接近于实际值，越能准确地获取信息。

X_6	X_2	X_3	X_4	X_7
X_5	X_1	X		

图 10.8 图像像素的编号示意图

经常使用的预测方法有：

$$X'=X_1$$
$$X'=(X_1+X_3)/2$$
$$X'=X_3-X_2+X_1$$
$$X'=(X_1+X_4)/2$$

具体采用哪种预测方法，要根据图像的种类、处理内容的不同而定。

10.6.2 预测编码编程实例

该实例对上述 4 种预测方法进行编程，并把 $\triangle X$ 保存到文件中，而且能够对该文件进行解码还原出原图像来。在实际应用中，还需要对 $\triangle X$ 进行量化，并通过哈夫曼编码处理等步骤，但该实例省略这些处理内容。

与上一个实例相似，该实例自定义一个新的图像格式（zdpcm），把经过 DPCM 编码后的数据放入该文件内。它只对 8 位灰度图像进程处理，因此该图像格式的头文件也很简单，只需要 5 个部分，第一个部分用于表示为 zdpcm 文件，第二个部分用于表示是 8 位灰度图像，第三部分用于表示使用的哪种预测方法，第四、五部分用于表示为图像的高和宽。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 10.11 所示。

表 10.11 所修改的属性

控　件	属　性	所修改内容
button1	Name	dpcm
	Text	预测编码
	Location	37，380

（2）创建 1 个名为 dpcmCode 的 Windows 窗体，该窗体用于选择是 DPCM 编码（生成

dpcm 文件），还是 DPCM 解码（把 zdpcm 文件恢复成原图像），并且编码时选择是哪种预测方法，在解码时无需选择预测方法，程序会自动判断预测方法。在该窗体内添加 2 个 Button 控件、1 个 GroupBox 控件、2 个 RadioButton 控件、1 个 Label 控件和 1 个 ListBox 控件，其属性修改如表 10.12 所示。

表 10.12 所修改的属性

控　件	属　性	所修改内容
dpcmCode	Size	332，226
	Text	DPCM
	ControlBox	False
button1	Name	start
	Text	确定
	Location	59，136
button2	Name	close
	Text	退出
	Location	185，136
groupBox1	Text	DPCM
	Size	112，78
	Location	39，27
radioButton1	Name	dpcmEncoding
	Text	DPCM 编码
	Location	20，21
	Checked	True
radioButton2	Name	dpcmDecoding
	Text	DPCM 解码
	Location	20，54
label1	Text	预测方法：
	Location	183，30
listBox1	Name	methodBox
	Size	99，52
	Location	183，53

为 2 个 Button 控件添加 Click 事件，为 2 个 RadioButton 控件添加 CheckedChanged 事件并添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    //得到ListBox 被选项
    methodDPCM = Convert.ToByte(methodBox.SelectedIndex);
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetDorC
{
    get
```

```
        {
            // 得到是编码还是解码
            return DorC;
        }
    }

    public byte GetMethod
    {
        get
        {
            // 得到运用哪种预测方法
            return methodDPCM;
        }
    }

    private void dpcmEncoding_CheckedChanged(object sender, EventArgs e)
    {
        DorC = false;
        //使ListBox有效
        methodBox.Enabled = true;
    }

    private void dpcmDecoding_CheckedChanged(object sender, EventArgs e)
    {
        DorC = true;
        //使ListBox无效
        methodBox.Enabled = false;
    }
```

其中 DorC 和 methodDPCM 分别为 bool 和 byte 型变量，它们在构造函数内被初始化，并且在构造函数内还为 LixtBox 控件添加了一些 Item 项，代码如下：

```
    DorC = false;
    methodDPCM = 0;
    // 清空ListBox
    methodBox.Items.Clear();
    // 为ListBox添加选项
    methodBox.Items.Add("· X'=X1");
    methodBox.Items.Add("· X'=(X1+X3)/2");
    methodBox.Items.Add("· X'=X3-X2+X1");
    methodBox.Items.Add("· X'=(X1+X4)/2");
    methodBox.SelectedIndex = 0;//设置初始被选项
```

（3）回到主窗体，为“预测编码”按钮添加 Click 事件，代码如下：

```
    private void dpcm_Click(object sender, EventArgs e)
    {
        // 实例化dpcmCode
        dpcmCode dpcmDC = new dpcmCode();

        if (dpcmDC.ShowDialog() == DialogResult.OK)
        {
            // 得到编码还是解码
            bool dpcmDorC = dpcmDC.GetDorC;

            if (dpcmDorC == false)
            {
```

```
// DPCM 编码

if (curBitmap == null)
{
    MessageBox.Show("请先打开图像! ");
    return;
}

Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
    System.Drawing.Imaging.ImageLockMode.ReadWrite,
    curBitmap.PixelFormat);
IntPtr ptr = bmpData.Scan0;
int bytes = curBitmap.Width * curBitmap.Height;
byte[] grayValues = new byte[bytes];
System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
curBitmap.UnlockBits(bmpData);

// DPCM 编码输出数据
int[] dpcmData = new int[bytes];
// 得到预测方法
byte methodD = dpcmDC.GetMethod;
int tempV = 0;

switch (methodD)
{
    case 0:
         // 预测方法1: X`=X1
         for (int i = 0; i < curBitmap.Height; i++)
         {
             for (int j = 0; j < curBitmap.Width; j++)
             {
                if (j == 0)
                   // 不在编码范围的数据需要单独处理
                   // 人为设定某一值
                   tempV = 128;
                else
                   tempV = grayValues[i * curBitmap.Width + j - 1];
                dpcmData[i * curBitmap.Width + j] =
                   grayValues[i * curBitmap.Width + j] - tempV;
             }
         }
         break;
    case 1:
         // 预测方法2: X`=(X1+X3)/2
         for (int i = 0; i < curBitmap.Height; i++)
         {
             if (i == 0)
             {
                  // 不在编码范围的数据
                  for (int j = 0; j < curBitmap.Width; j++)
                  {
                   if (j == 0)
                        tempV = 128;
                   else
```

```
                                    tempV = (grayValues[i * curBitmap.Width + j - 1]
                                        128) / 2;
                                dpcmData[i * curBitmap.Width + j] =
                                    grayValues[i * curBitmap.Width + j] - tempV;
                            }
                        }
                        else
                        {
                            for (int j = 0; j < curBitmap.Width; j++)
                            {
                              if (j == 0)
                                // 不在编码范围的数据
                                tempV = (128 + grayValues[(i - 1) *
                                    curBitmap.Width + j]) / 2;
                              else
                                tempV = (grayValues[(i - 1) * curBitmap.Width + j]
                                    grayValues[i * curBitmap.Width + j - 1]) / 2;
                              dpcmData[i * curBitmap.Width + j] =
                                grayValues[i * curBitmap.Width + j] - tempV;
                            }
                        }
                    }
                    break;
                case 2:
                    // 预测方法3: X`=X3-X2+X1
                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        if (i == 0)
                        {
                            // 不在编码范围的数据
                            for (int j = 0; j < curBitmap.Width; j++)
                            {
                             if (j == 0)
                                tempV = 64;
                             else
                                tempV =
                                    grayValues[i * curBitmap.Width + j - 1] + 64;
                             dpcmData[i * curBitmap.Width + j] =
                                grayValues[i * curBitmap.Width + j] - tempV;
                            }
                        }
                        else
                        {
                            for (int j = 0; j < curBitmap.Width; j++)
                            {
                              if (j == 0)
                                // 不在编码范围的数据
                                tempV = 64 + grayValues[(i - 1) *
                                    curBitmap.Width + j];
                              else
                                tempV = grayValues[(i - 1) * curBitmap.Width + j]
                                    grayValues[i * curBitmap.Width + j - 1] -
                                    grayValues[(i - 1) * curBitmap.Width + j - 1]
                              dpcmData[i * curBitmap.Width + j] =
                                grayValues[i * curBitmap.Width + j] - tempV;
```

```
                        }
                    }
                }
                break;
            case 3:
                // 预测方法4: X`=(X1+X4)/2
                for (int i = 0; i < curBitmap.Height; i++)
                {
                    if (i == 0)
                    {
                        // 不在编码范围的数据
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
                         if (j == 0)
                             tempV = 128;
                         else
                             tempV = (grayValues[i * curBitmap.Width + j - 1] +
                                 128) / 2;
                         dpcmData[i * curBitmap.Width + j] =
                             grayValues[i * curBitmap.Width + j] - tempV;
                        }
                    }
                    else
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
                          if (j == 0)
                            // 不在编码范围的数据
                            tempV = (128 + grayValues[(i - 1) *
                                curBitmap.Width + j + 1]) / 2;
                          else if (j == curBitmap.Width - 1)
                            // 不在编码范围的数据
                            tempV = (128 + grayValues[i * curBitmap.Width +
                                j - 1]) / 2;
                          else
                            tempV = (grayValues[(i - 1) * curBitmap.Width + j
                                1] + grayValues[i * curBitmap.Width + j - 1]) / 2
                          dpcmData[i * curBitmap.Width + j] =
                            grayValues[i * curBitmap.Width + j] - tempV;
                        }
                    }
                }
                break;
           default:
                break;
        }

        // 保存文件对话框
        SaveFileDialog sf = new SaveFileDialog();
        // 设置标题
        sf.Title = "保存文件";
        // 设置文件保存类型
        sf.Filter = "自定义图像格式(*.zdpcm)|*.zdpcm";
        // 如果用户没有输入扩展名，自动追加后缀
        sf.AddExtension = true;
```

```
            // 如果用户单击了保存按钮
            if (sf.ShowDialog() == DialogResult.OK)
            {
                // 实例化一个文件流
                FileStream fs = new FileStream(sf.FileName, FileMode.Create);
                // 实例化一个BinaryWriter
                BinaryWriter bw = new BinaryWriter(fs);

                // 开始写入
                //zdpcm 文件标识
                bw.Write("zdpcm");
                // 8 位灰度图像
                bw.Write("8b");
                // 预测方法
                bw.Write(methodD);
                // 图像高
                bw.Write(Convert.ToInt16(curBitmap.Height));
                // 图像宽
                bw.Write(Convert.ToInt16(curBitmap.Width));
                // 写入编码数据
                for (int i = 0; i < bytes; i++)
                    bw.Write(Convert.ToInt16(dpcmData[i]));

                // 清空缓冲区
                bw.Flush();
                // 关闭流
                bw.Close();
                fs.Close();
            }
        }
        else
        {
            // DPCM 解码

            // 打开文件对话框
            OpenFileDialog of = new OpenFileDialog();
            // 设置标题
            of.Title = "打开文件";
            // 设置文件保存类型
            of.Filter = "自定义图像格式(*.zdpcm)|*.zdpcm";
            // 如果用户单击了保存按钮
            if (of.ShowDialog() == DialogResult.OK)
            {
                // 实例化一个文件流
                FileStream fs = new FileStream(of.FileName, FileMode.Open);
                // 实例化一个BinaryReader
                BinaryReader br = new BinaryReader(fs);

                // 定义读取文件的起始位置
                br.BaseStream.Position = 0;
                // 读取zdpcm 文件标识
                string readData = br.ReadString();
                if (readData != "zdpcm")
                {
                    MessageBox.Show("不是zdpcm 文件格式! ");
```

```
            return;
        }
        // 读取8 位灰度图像标识
        readData = br.ReadString();
        if (readData != "8b")
        {
            MessageBox.Show("本实例只能处理8 位灰度图像! ");
            return;
        }
        // 预测方法
        byte methodOFDpcm = br.ReadByte();
        // 图像高
        int heightDpcm = br.ReadInt16();
        // 图像宽
        int widthDpcm = br.ReadInt16();
        // 数据大小
        int bytesD = widthDpcm * heightDpcm;
        // 定义图像数据
        byte[] grayDpcm = new byte[bytesD];
        // 定义解码输出数据
        int[] dpcmDecode = new int[bytesD];
        // 读取解码数据
        for (int i = 0; i < bytesD; i++)
            dpcmDecode[i] = br.ReadInt16();

        int tempDV = 0;
        switch (methodOFDpcm)
        {
            case 0:
                // 预测方法1: X'=X1
                for (int i = 0; i < heightDpcm; i++)
                {
                    for (int j = 0; j < widthDpcm; j++)
                    {
                        if (j == 0)
                            tempDV = 128;
                        else
                            tempDV = grayDpcm[i * widthDpcm + j - 1];
                        grayDpcm[i * widthDpcm + j] =
                            (byte)(dpcmDecode[i * widthDpcm + j] + tempDV);
                    }
                }
                break;
            case 1:
                // 预测方法2: X'=(X1+X3)/2
                for (int i = 0; i < heightDpcm; i++)
                {
                    if (i == 0)
                    {
                        for (int j = 0; j < widthDpcm; j++)
                        {
                            if (j == 0)
                                tempDV = 128;
                            else
                                tempDV = (grayDpcm[i * widthDpcm + j - 1] +
```

```
                                        128) / 2;
                                grayDpcm[i * widthDpcm + j] = (byte)
                                    (dpcmDecode[i * widthDpcm + j] + tempDV);
                            }
                        }
                        else
                        {
                            for (int j = 0; j < widthDpcm; j++)
                            {
                                if (j == 0)
                                    tempDV = (128 + grayDpcm[(i - 1) *
                                        widthDpcm + j]) / 2;
                                else
                                    tempDV = (grayDpcm[i * widthDpcm + j - 1] +
                                        grayDpcm[(i - 1) * widthDpcm + j]) / 2;
                                grayDpcm[i * widthDpcm + j] = (byte)
                                    (dpcmDecode[i * widthDpcm + j] + tempDV);
                            }
                        }
                    }
                    break;
                case 2:
                    // 预测方法3: X'=X3-X2+X1
                    for (int i = 0; i < heightDpcm; i++)
                    {
                        if (i == 0)
                        {
                            for (int j = 0; j < widthDpcm; j++)
                            {
                                if (j == 0)
                                    tempDV = 64;
                                else
                                    tempDV = grayDpcm[i * widthDpcm + j - 1] + 64
                                grayDpcm[i * widthDpcm + j] = (byte)
                                    (dpcmDecode[i * widthDpcm + j] + tempDV);
                            }
                        }
                        else
                        {
                            for (int j = 0; j < widthDpcm; j++)
                            {
                                if (j == 0)
                                    tempDV = 64 +
                                        grayDpcm[(i - 1) * widthDpcm + j];
                                else
                                    tempDV = grayDpcm[i * widthDpcm + j - 1] +
                                        grayDpcm[(i - 1) * widthDpcm + j] -
                                        grayDpcm[(i - 1) * widthDpcm + j - 1];
                                grayDpcm[i * widthDpcm + j] = (byte)
                                    (dpcmDecode[i * widthDpcm + j] + tempDV);
                            }
                        }
                    }
                    break;
                case 3:
```

```
                            // 预测方法4: X'=(X1+X4)/2
                            for (int i = 0; i < heightDpcm; i++)
                            {
                                if (i == 0)
                                {
                                   for (int j = 0; j < widthDpcm; j++)
                                   {
                                      if (j == 0)
                                          tempDV = 128;
                                      else
                                          tempDV = (grayDpcm[i * widthDpcm + j - 1] +
                                              128) / 2;
                                      grayDpcm[i * widthDpcm + j] = (byte) (dpcmDecode[
widthDpcm + j] + tempDV);
                                   }
                                }
                                else
                                {
                                   for (int j = 0; j < widthDpcm; j++)
                                   {
                                      if (j == 0)
                                          tempDV = (128 + grayDpcm[(i - 1) *
                                              widthDpcm + j + 1]) / 2;
                                      else if (j == widthDpcm - 1)
                                          tempDV = (128 + grayDpcm[i * widthDpcm + j
                                              - 1]) / 2;
                                      else
                                          tempDV = (grayDpcm[i * widthDpcm + j - 1] +
                                              grayDpcm[(i - 1) * widthDpcm + j + 1]) / 2
                                      grayDpcm[i * widthDpcm + j] = (byte)
                                          (dpcmDecode[i * widthDpcm + j] + tempDV);
                                   }
                                }
                            }
                            break;
                        default:
                            break;
                    }

                    // 把解码后的图像显示在主窗体内
                    curBitmap = new Bitmap(widthDpcm, heightDpcm,
                        System.Drawing.Imaging.PixelFormat.Format8bppIndexed);
                    // 设置为8位灰度图像
                    System.Drawing.Imaging.ColorPalette cp = curBitmap.Palette;
                    for (int i = 0; i < 256; i++)
                    {
                        cp.Entries[i] = Color.FromArgb(i, i, i);
                    }
                    curBitmap.Palette = cp;

                    Rectangle rectDpcm = new Rectangle(0, 0, widthDpcm, heightDpcm);
                    System.Drawing.Imaging.BitmapData dData = curBitmap.LockBits
                        (rectDpcm, System.Drawing.Imaging.ImageLockMode.ReadWrite,
                        curBitmap.PixelFormat);
                    IntPtr ptr = dData.Scan0;
```

```
                    System.Runtime.InteropServices.Marshal.Copy(grayDpcm,0,ptr, bytesD);
                    curBitmap.UnlockBits(dData);
                    // 关闭流
                    br.Close();
                    fs.Close();

                    Invalidate();
                }
            }
        }
    }
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开该图后，单击“预测编码”按钮，打开预测编码对话框，如图 10.9 所示设置相关参数，然后单击“确定”按钮后，会出现“保存文件”对话框，选择好路径和文件名后，就完成了把所打开的图像经过预测编码中的某一种方法处理后保存为 zdpcm 文件的过程。

图 10.9　预测编码对话框

在任何情况下，单击“预测编码”按钮，则打开如图 10.9 所示的对话框，这时选择“DPCM 解码”按钮，再单击“确定”按钮，则打开了“打开文件”对话框，找到刚才保存的那个 zdpcm 文件，此时 zdpcm 文件被解码成图像并显示在主窗体内，如图 10.10 所示。如果没有量化误差，预测压缩编码方法是一种无损压缩方法，正如该实例所呈现出来的那样。

图 10.10　预测解码结果

10.7　傅里叶变换编码

变换编码不直接对空间域图像数据进行编码，而是首先将空间域图像数据映射变换到另一个正交向量空间，得到一组变换系数。因为是对变换系数进行压缩编码，因此往往比直接

对图像数据本身进行压缩更容易获得高的效率。实践证明无论对单色图像、彩色图像、静止图像还是运动图像，变换编码都是非常有效的方法。

变换编码的基本方法是将数字图像分成一定大小的子图像块，用某种正交变换对子图像块进行变换，得到变换域中的系数矩阵，然后选用其中的主要系数进行量化和编码。图像显示时再经过逆变换即可重构原图像。

正交变换的种类很多，比如傅里叶变换、沃尔什-哈达玛变换、哈尔变换、斜变换、余弦变换、正弦变换，还有基于统计特性的 K−L 变换。本节介绍傅里叶变换。

10.7.1 傅里叶变换编码原理

图像经过二维傅里叶变换后，其变换系数矩阵为二维频谱。若变换矩阵的原点设在中心则其频谱能量集中分布在变换系数矩阵的中心附近，而且中心数据占据总能量很大比例。根据图像信号二维傅里叶变换系数分布的这些特点，采用区域抽样，再以区域编码，也即只对数值较大的变换系数进行编码，就可以达到图像压缩的目的。

傅里叶变换编码的具体步骤是：将图像分割成若干个子图像，对每个子图像进行 FFT，得到不同区域的变换系数，分别在每个区域内对该区域内的变换系数进行排序，根据压缩比舍去小的变换系数，就可实现数据的压缩。它的解码就是它的逆变换：对变换系数进行傅里叶逆变换，得到每个子图像，再把这些子图像合并成一幅最终的完整图像。一般子图像的大小选择 8×8 较为常见，也有用 16×16、4×4 的。

需要说明的是，由于相邻子图像边界的不连续性，会造成所谓的 Gibbs 现象，它表现为当复原子图像构成的整幅图像将呈现隐约可见的以子图像尺寸为单位的方块状结构，影响整个图像质量，当子图像尺寸较小时更为严重。

10.7.2 傅里叶变换编码编程实例

该实例通过选择不同的压缩比和子图像的大小，来完成傅里叶变换编、解码，并把解码后复原图像呈现出来。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 10.13 所示。

表 10.13　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	transform
	Text	傅里叶变换
	Location	37，426

（2）创建 1 个名为 transCode 的 Windows 窗体，该窗体用于选择子图像大小及压缩比。在该窗体内添加 2 个 Button 控件、2 个 GroupBox 控件、6 个 RadioButton 控件，其属性修改如表 10.14 所示。

表 10.14　　所修改的属性

控　　件	属　　性	所修改内容
transCode	Size	365，300

	Text	傅里叶变换编码
	ControlBox	False

续表

控　件	属　性	所修改内容
button1	Name	start
	Text	确定
	Location	51，218
button2	Name	close
	Text	退出
	Location	225，218
groupBox1	Text	子图像尺寸大小
	Location	26，25
	Size	125，164
radioButton1	Name	size4
	Text	4×4
	Location	25，34
	Checked	True
radioButton2	Name	size8
	Text	8×8
	Location	25，78
radioButton3	Name	size16
	Text	16×16
	Location	25，122
groupBox2	Text	压缩比
	Location	204，25
	Size	125，164
radioButton4	Name	ratio2
	Text	2：1
	Location	30，34
	Checked	True
radioButton5	Name	ratio4
	Text	4：1
	Location	30，78
radioButton6	Name	ratio8
	Text	8：1
	Location	30，122

为 2 个 Button 控件添加 Click 事件，为 6 个 RadioButton 控件添加 CheckedChanged 事件件添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
```

```
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void size4_CheckedChanged(object sender, EventArgs e)
{
    sizeNum = 4;
}

private void size8_CheckedChanged(object sender, EventArgs e)
{
    sizeNum = 8;
}

private void size16_CheckedChanged(object sender, EventArgs e)
{
    sizeNum = 16;
}

private void ratio2_CheckedChanged(object sender, EventArgs e)
{
    ratioNum = 2;
}

private void ratio4_CheckedChanged(object sender, EventArgs e)
{
    ratioNum = 4;
}

private void ratio8_CheckedChanged(object sender, EventArgs e)
{
    ratioNum = 8;
}

public byte GetSize
{
    get
    {
         // 得到子图像大小
         return sizeNum;
    }
}

public byte GetRatio
{
    get
    {
         // 得到压缩比
         return ratioNum;
    }
}
```

其中 sizeNum 和 ratioNum 都是 byte 型变量，它们在构造函数内被初始化，代码如下：

```
sizeNum = 4;
ratioNum = 2;
```

（3）回到主窗体，为傅里叶变换编码编写代码。如第 6 章介绍的那样，首先要添加一个用于表示复数的类，然后编写 FFT 方法代码。这些在 6.2 节已给出了完整的代码，只需把它们复制到本实例代码内即可，这里就不再赘述。完成代码复制工作后，为“变换编码”按钮添加 Click 事件，代码如下：

```
private void transform_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化transCode
         transCode fftCode = new transCode();

         if (fftCode.ShowDialog() == DialogResult.OK)
         {
              Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
              System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                  System.Drawing.Imaging.ImageLockMode.ReadWrite,
                  curBitmap.PixelFormat);
              IntPtr ptr = bmpData.Scan0;
              int bytes = curBitmap.Width * curBitmap.Height;
              byte[] grayValues = new byte[bytes];
              System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

              // 得到子图像尺寸
              byte sizeN = fftCode.GetSize;
              // 得到压缩比
              byte ratioN = fftCode.GetRatio;

              // 子图像大小
              int sizeSI = sizeN * sizeN;
              int col = curBitmap.Width / sizeN;
              int row = curBitmap.Height / sizeN;
              byte[,] splitImage = new byte[col * row, sizeSI];
              // 根据压缩比确定变换系数中变为零的个数
              int snum = sizeSI - sizeSI / ratioN;
              int a, b, c, d;

              // 分割成子图像
              for (int i = 0; i < curBitmap.Height; i++)
              {
                  for (int j = 0; j < curBitmap.Width; j++)
                  {
                       a = j / sizeN;
                       b = j % sizeN;
                       c = i / sizeN;
                       d = i % sizeN;
                       splitImage[c * col + a, d * sizeN + b] =
                           grayValues[i * curBitmap.Width + j];
                  }
              }
```

```
Complex[] freDom = new Complex[sizeSI];
byte[] tempImage = new byte[sizeSI];
double[] temp1D = new double[sizeSI];

// 计算不同的子图像
for (int i = 0; i < col * row; i++)
{
    for (int j = 0; j < sizeSI; j++)
        tempImage[j] = splitImage[i, j];
    // 调用二维傅里叶变换
    freDom = fft2(tempImage, sizeN, sizeN, false);
    // 复数求模
    for (int j = 0; j < sizeSI; j++)
        temp1D[j] = freDom[j].Abs();

    double tempD = 0;

    // 冒泡排序法
    for (int m = 0; m < sizeSI - 1; m++)
    {
        for (int n = 0; n < sizeSI - 1 - m; n++)
        {
            if (temp1D[n] > temp1D[n + 1])
            {
                tempD = temp1D[n];
                temp1D[n] = temp1D[n + 1];
                temp1D[n + 1] = tempD;
            }
        }
    }

    // 根据阈值把变换系数赋值为零，实现压缩
    tempD = temp1D[snum - 1];
    for (int j = 0; j < sizeSI; j++)
    {
        if (freDom[j].Abs() <= tempD)
            freDom[j] = new Complex(0.0, 0.0);
    }
    // 调用二维傅里叶逆变换
    tempImage = ifft2(freDom, sizeN, sizeN, false);
    for (int j = 0; j < sizeSI; j++)
        splitImage[i, j] = tempImage[j];
}

// 把各个子图像复原成完整的图像
for (int i = 0; i < curBitmap.Height; i++)
{
    for (int j = 0; j < curBitmap.Width; j++)
    {
        a = j / sizeN;
        b = j % sizeN;
        c = i / sizeN;
        d = i % sizeN;
        grayValues[i * curBitmap.Width + j] =
```

```
                                splitImage[c * col + a, d * sizeN + b];
                        }
                    }

                    System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                    curBitmap.UnlockBits(bmpData);

                    Invalidate();
                }
            }
        }
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开图像后，单击“变换编码”按钮，打开傅里叶变换编码对话框，如图 10.11 所示，并按照该图设置相关参数，单击“确定”按钮后，在主窗体内会出现经过傅里叶变换后的还原图像，如图 10.12 所示。

图 10.11　“傅里叶变换编码”对话框

图 10.12　傅里叶变换编码结果

图 10.12 是子图像尺寸为 8×8、压缩比为 8：1 的压缩图像，从图中可明显看出有与子图像尺寸大小相同的方块状结构，这就是变换编码中最令人头痛的方块效应。克服这种失真的方法有交叠分块法和后滤波法，在这里就不再介绍。

10.8　小波变换编码

小波变换编码也属于变换编码，因此同其他变换编码一样，小波变换编码分为小波变换、量化和熵编码 3 个步骤。同上面介绍的一样，本部分只涉及小波变换内容。

10.8.1　小波变换编码原理

图像的小波变换编码首先是对原始图像实行二维小波变换，得到小波变换系数。由于小波变换能将原始图像的能量集中到少部分小波系数上，且分解后的小波系数在 3 个方向的分量有高度的局部相关性，这为进一步量化提供了有利条件。因此应用小波编码可得到较

弱的压缩比。

在基于小波变换压缩编码的方法中，最简单的方法是利用小波分解，去掉图像的高频部分而保留低频部分。

10.8.2　小波变换编码编程实例

本实例通过选择小波分解级数及小波基来完成小波变换，经过去除高频分量处理后，再对解码图像显示在主窗体内，从而完成小波变换编码、解码的过程。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 10.15 所示。

表 10.15　　所修改的属性

控　　件	属　　性	所修改内容
button1	Name	wavelet
	Text	小波变换
	Location	37，472

（2）创建 1 个名为 wlTrans 的 Windows 窗体，该窗体用于选择小波变换级数及低通滤波器。在该窗体内添加 2 个 Button 控件、1 个 Label 控件、1 个 NumericUpDown 控件、1 个 GroupBox 控件和 6 个 RadioButton 控件，其属性修改如表 10.16 所示。

表 10.16　　所修改的属性

控　　件	属　　性	所修改内容
wlTrans	Text	小波变换编码
	Size	260，410
	ControlBox	False
button1	Name	start
	Text	确定
	Location	25，332
button2	Name	close
	Text	退出
	Location	150，332
label1	Text	小波变换级数
	Location	34，25
numericUpDown1	Name	waveletSeries
	Size	77，21
	Location	129，23
	Minimum	1
	Value	2
groupBox1	Location	25，69
	Size	200，240
	Text	低通滤波器
radioButton1	Name	haar
	Text	Haar
	Location	40，29
	Checked	True

radioButton2	Name	daubechies2

续表

控　　件	属　　性	所修改内容
radioButton2	Text	Daubechies2
	Location	40，64
radioButton3	Name	daubechies3
	Text	Daubechies3
	Location	40，99
radioButton4	Name	daubechies4
	Text	Daubechies4
	Location	40，134
radioButton5	Name	daubechies5
	Text	Daubechies5
	Location	40，169
radioButton6	Name	daubechies6
	Text	Daubechies6
	Location	40，204

分别为 2 个 Button 控件添加 Click 事件，6 个 RadioButton 控件添加 CheckedChanged 事件，并添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void haar_CheckedChanged(object sender, EventArgs e)
{
    wlBase = 0;
}

private void daubechies2_CheckedChanged(object sender, EventArgs e)
{
    wlBase = 1;
}

private void daubechies3_CheckedChanged(object sender, EventArgs e)
{
    wlBase = 2;
}

private void daubechies4_CheckedChanged(object sender, EventArgs e)
{
    wlBase = 3;
}
```

```
private void daubechies5_CheckedChanged(object sender, EventArgs e)
{
    wlBase = 4;
}

private void daubechies6_CheckedChanged(object sender, EventArgs e)
{
    wlBase = 5;
}

public byte GetSeries
{
    get
    {
        // 得到小波变换级数
        return (byte)waveletSeries.Value;
    }
}

public byte GetBase
{
    get
    {
        // 得到小波基（低通滤波器）
        return wlBase;
    }
}
```

其中 wlBase 是 byte 型变量，它在构造函数内被初始化，代码如下：

```
wlBase = 0;
```

（3）回到主窗体。为了实现小波变换，还必须编写二维小波变换方法，这在第 7.4 节已经作了详细介绍，并给出了具体代码，在这里直接复制这些代码即可。复制完成后，为“小波变换”按钮添加 Click 事件，代码如下：

```
private void wavelet_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化wlTrans
        wlTrans waveletCode = new wlTrans();

        if (waveletCode.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);

            // 得到小波变换级数
```

```
                byte wlSeries = waveletCode.GetSeries;
                // 得到小波基
                byte waveletBase = waveletCode.GetBase;

                double[] tempA = new double[bytes];
                double[] tempB = new double[bytes];
                for (int i = 0; i < bytes; i++)
                    tempA[i] = Convert.ToDouble(grayValues[i]);

                double[] lowFilter = null;
                double[] highFilter = null;

                // 确定低通滤波器
                switch (waveletBase)
                {
                    case 0:
                        // haar
                        lowFilter = new double[] { 0.70710678118655, 0.70710678118655 }
                        break;
                    case 1:
                        // daubechies2
                        lowFilter = new double[] { 0.48296291314453, 0.83651630373780,
                            0.22414386804201, -0.12940952255126 };
                        break;
                    case 2:
                        // daubechies3
                        lowFilter = new double[] { 0.33267055295008, 0.80689150931109,
                                0.45987750211849, -0.13501102001025,
                                -0.08544127388203, 0.03522629188571 };
                        break;
                    case 3:
                        // daubechies4
                        lowFilter = new double[] { 0.23037781330889, 0.71484657055291,
                            0.63088076792986, -0.02798376941686, -0.18703481171909,
                            0.03084138183556, 0.03288301166689, -0.01059740178507 };
                        break;
                    case 4:
                        // daubechies5
                        lowFilter = new double[] { 0.16010239797419, 0.60382926979719,
                            0.72430852843778, 0.13842814590132,
0.24229488706638,-0.03224486958464,     0.07757149384005, -0.00624149021280,
                            -0.01258075199908, 0.00333572528547 };
                        break;
                    case 5:
                        // daubechies6
                        lowFilter = new double[] { 0.11154074335011, 0.49462389039845,
                            0.75113390802110, 0.31525035170920, -0.22626469396544,
                            -0.12976686756726, 0.09750160558732, 0.02752286553031,
                            -0.03158203931849, 0.00055384220116, 0.00477725751195,
                            -0.00107730108531 };
                        break;
                    default:
                        MessageBox.Show("无效! ");
                        break;
                }
```

```
// 确定高通滤波器
highFilter = new double[lowFilter.Length];
for (int i = 0; i < lowFilter.Length; i++)
    highFilter[i] = Math.Pow(-1, i) * lowFilter[lowFilter.Length - 1 - i]

// 二维小波变换
for (int k = 0; k < wlSeries; k++)
{
    int coef = (int)Math.Pow(2, k);

    // 变换前更新数据
    for (int i = 0; i < curBitmap.Height; i++)
    {
        if (i < curBitmap.Height / coef)
        {
            for (int j = 0; j < curBitmap.Width; j++)
            {
                if (j < curBitmap.Width / coef)
                {
                 tempB[i * curBitmap.Width / coef + j] =
                     tempA[i * curBitmap.Width + j];
                }
            }
        }
    }

    // 调用二维小波变换
    wavelet2D(ref tempB, lowFilter, highFilter, coef);

    // 变换后更新数据
    for (int i = 0; i < curBitmap.Height; i++)
    {
        if (i < curBitmap.Height / coef)
        {
            for (int j = 0; j < curBitmap.Width; j++)
            {
                if (j < curBitmap.Width / coef)
                {
                   tempA[i * curBitmap.Width + j] =
                       tempB[i * curBitmap.Width / coef + j];
                }
            }
        }
    }
}

// 保留低频部分，高频部分置为零
int col = curBitmap.Width / (int)Math.Pow(2, wlSeries);
int row = curBitmap.Height / (int)Math.Pow(2, wlSeries);
for (int i = 0; i < curBitmap.Height; i++)
{
    for (int j = 0; j < curBitmap.Width; j++)
    {
        if (i >= row || j >= col)
```

```
                tempA[i * curBitmap.Width + j] = 0.0;
        }
    }

    // 二维小波逆变换
    for (int k = wlSeries - 1; k >= 0; k--)
    {
        int coef = (int)Math.Pow(2, k);

        // 变换前更新数据
        for (int i = 0; i < curBitmap.Height; i++)
        {
            if (i < curBitmap.Height / coef)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    if (j < curBitmap.Width / coef)
                    {
                     tempB[i * curBitmap.Width / coef + j] =
                         tempA[i * curBitmap.Width + j];
                    }
                }
            }
        }

        // 调用二维小波逆变换
        iwavelet2D(ref tempB, lowFilter, highFilter, coef);

        // 变换后更新数据
        for (int i = 0; i < curBitmap.Height; i++)
        {
            if (i < curBitmap.Height / coef)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
                    if (j < curBitmap.Width / coef)
                    {
                       tempA[i * curBitmap.Width + j] =
                           tempB[i * curBitmap.Width / coef + j];
                    }
                }
            }
        }
    }

    for (int i = 0; i < bytes; i++)
    {
        if (tempA[i] >= 255)
            tempA[i] = 255;
        if (tempA[i] <= 0)
            tempA[i] = 0;
        grayValues[i] = Convert.ToByte(tempA[i]);
    }

    System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
```

```
                curBitmap.UnlockBits(bmpData);

                Invalidate();
            }
        }
    }
```

（4）编译并运行该段程序，仍以图 4.1 为例。打开图像后，单击“小波变换”按钮，打开小波变换对话框，如图 10.13 所示设置相关参数，单击“确定”按钮后，小波变换后的解码图像如图 10.14 所示。

图 10.13　“小波变换编码”对话框

图 10.14　小波变换编码结果

10.9　小结

图像压缩编码有几十种之多，本章只介绍了几种比较常见的方法。需要说明的是，本章介绍的几种方法中，有的只给出了具体的编、解码方法，而并没有真正地实现数据的压缩。例如在变换编码时，在实际应用中，还需对变换系数做量化处理以及限定数据字节的长度，然后应用熵编码等技术实现真正的数据压缩。同时这也解释了本章所保存的文件中有些文件要比源文件大的原因。

第 11 章　彩色图像处理

在大千世界里，物体是五彩斑斓的，大多数图像都具有丰富多彩的色彩。彩色图像提供了比灰度图像更丰富的信息，而且人眼对彩色图像的视觉感受要比对黑白或灰度图像的感受丰富得多。然而，在彩色图像的处理过程中不仅要考虑位置、灰度，还要考虑彩色信息。因此它比对灰度图像处理要复杂。

彩色图像处理技术比灰度图像处理技术的研究要晚一些，但随着彩色图像应用领域的扩大，它将会得到迅速发展。目前，彩色图像处理技术分为两类：一类是分别处理彩色图像中的多个分量图像，然后将分别处理后的分量图像合成一幅彩色图像，这样就可以直接应用对灰度图像的处理方法；另一类是直接对彩色图像像素进行处理。

本章首先介绍彩色空间各个分量，然后给出具体的彩色图像处理方法。

11.1　彩色空间

在进行图像处理时，人们并不对自然界的所有颜色一一处理，而是根据色度学理论建立彩色模型，再基于彩色模型进行处理。在这些彩色模型中，除了介绍过的 RGB 彩色空间（即彩色模型）外，还包括 NTSC、YCbCr、HSV、CMY、CMYK 和 HSI 彩色空间。在这里，我们只给出 RGB 彩色空间和 HSI 彩色空间这两种最常用的彩色空间的定义，以及它们之间的转换关系。

11.1.1　RGB 彩色空间和 HSI 彩色空间

RGB 彩色空间是目前常用的一种彩色信息表达方式，它用红、绿、蓝三原色的亮度来定量表示颜色。

HSI 彩色空间通过色度 H、饱和度 S 和亮度 I 来表示物体的颜色。其中 H 定义了颜色的波长，它反映了颜色最接近哪种光谱波长，即光的不同颜色，如红、绿、蓝等。S 表示颜色的深浅程度，S 越高，颜色越深。I 表示强度，是指光波作用与感受器所发生的效应，其大小由物体反射系数来决定。HSI 彩色空间反映了人的视觉对色彩的感觉。H 和 S 包含了颜色信息，而 I 则与颜色信息无关。

若给出一幅 RGB 彩色格式的图像，并且 RGB 的值已归一化到范围[0，1]，则每一个 RGB 像素的 H 分量可用下面的方程得到：

$$H=\begin{cases} q & B\leqslant G \\ 360-q & B>G \end{cases} \tag{11.1}$$

其中，

$$q=\arccos\left\{\frac{[(R-G)+(R-B)]/2}{[(R-G)^2+(R-B)(G-B)]^{1/2}}\right\} \tag{11.2}$$

S 分量由下式给出：

$$S=1-\frac{3}{(R+G+B)}[\min(R,G,B)] \tag{11.3}$$

I 分量由下式给出：

$$I=\frac{R+G+B}{3} \tag{11.4}$$

此时，将等式（11.1）的结果除以 360，则 H 可归一化到范围[0，1]，而 S 和 I 自然是在[0，1]内。

同理，假设 H、S、I 的值在[0，1]之间，R、G、B 的值也在[0，1]之间，则由 HSI 转换为 RGB 的公式需要依据颜色落在色环的哪个扇区来选用不同的转换公式。

（1）当 0°≤H<120°：

$$R=I\left[1+\frac{S\cdot\cos H}{\cos(60-H)}\right]$$

$$\begin{aligned} B&=I(1-S) \\ G&=3I-R-B \end{aligned} \tag{11.5}$$

（2）当 120°≤H<240°：

$$G=I\left[1+\frac{S\cdot\cos(H-120)}{\cos(180-H)}\right]$$

$$\begin{aligned} R&=I(1-S) \\ B&=3I-R-G \end{aligned} \tag{11.6}$$

（3）当 240°≤H<360°：

$$B=I\left[1+\frac{S\cdot\cos(H-240)}{\cos(300-H)}\right]$$

$$\begin{aligned} G&=I(1-S) \\ R&=3I-G-B \end{aligned} \tag{11.7}$$

11.1.2 彩色空间转换编程实例

该实例的目的就是把彩色图像的各个分量（R、G、B、H、S、I）单独显示出来。

（1）创建1个“模板主窗体”，在该窗体内添加1个Button控件，其属性修改如表11.1所示

表 11.1 所修改的属性

控　件	属　性	所修改内容
button1	Name	tranSpace
	Text	空间转换
	Location	37，150

（2）创建1个名为colorSpace的Windows窗体，该窗体用于选择RGB空间和HSI空间的分量。在该窗体内添加2个Button控件、1个GroupBox控件、6个RadioButton控件，其属性修改如表11.2所示。

表 11.2 所修改的属性

控　件	属　性	所修改内容
colorSpace	Text	彩色空间
	Size	336，270
	ControlBox	False
button1	Name	start
	Text	确定
	Location	50，194
button2	Name	close
	Text	退出
	Location	204，194
groupBox1	Text	彩色空间分量
	Location	26，21
	Size	274，153
radioButton1	Name	redC
	Text	红（R）
	Location	24，30
	Checked	True
radioButton2	Name	greenC
	Text	绿（G）
	Location	24，73
radioButton3	Name	blueC
	Text	蓝（B）
	Location	24，116
radioButton4	Name	hueC
	Text	色度（H）

	Location	145，30

续表

控　　件	属　　性	所修改内容
radioButton5	Name	satC
	Text	饱和度（S）
	Location	145，73
radioButton6	Name	intC
	Text	亮度（I）
	Location	145，116

分别为 2 个 Button 控件添加 Click 事件，为 6 个 RadioButton 控件添加 CheckedChanged 事件和 1 个 get 属性访问器，代码如下：

```
private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void redC_CheckedChanged(object sender, EventArgs e)
{
    component = 0;
}

private void greenC_CheckedChanged(object sender, EventArgs e)
{
    component = 1;
}

private void blueC_CheckedChanged(object sender, EventArgs e)
{
    component = 2;
}

private void hueC_CheckedChanged(object sender, EventArgs e)
{
    component = 3;
}

private void satC_CheckedChanged(object sender, EventArgs e)
{
    component = 4;
}

private void intC_CheckedChanged(object sender, EventArgs e)
{
    component = 5;
}
```

```
public byte GetCom
{
    get
    {
        // 得到颜色分量
        return component;
    }
}
```

其中 component 为 byte 型变量，它在构造函数内被初始化为 0。

（3）回到主窗体，为“空间转换”按钮添加 Click 事件，代码如下：

```
private void tranSpace_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化colorSpace
        colorSpace clS = new colorSpace();

        if (clS.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] rgbValues = new byte[bytes * 3];
            System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes * 3)
            curBitmap.UnlockBits(bmpData);

            // 得到所要显示的颜色分量
            byte colorCom = clS.GetCom;
            // 图像数组
            byte[] grayValues = new byte[bytes];
            byte tempB;
            double tempD;
            switch (colorCom)
            {
                case 0:
                    // 红色分量
                    for (int i = 0; i < bytes; i++)
                        grayValues[i] = rgbValues[i * 3 + 2];
                    break;
                case 1:
                    // 绿色分量
                    for (int i = 0; i < bytes; i++)
                        grayValues[i] = rgbValues[i * 3 + 1];
                    break;
                case 2:
                    // 蓝色分量
                    for (int i = 0; i < bytes; i++)
                        grayValues[i] = rgbValues[i * 3];
                    break;
                case 3:
                    // 色度分量
                    double theta;
```

```
                    for (int i = 0; i < bytes; i++)
                    {
                        theta = Math.Acos(0.5 * ((rgbValues[i * 3 + 2] -
                            rgbValues[i * 3 + 1]) + (rgbValues[i * 3 + 2] -
                            rgbValues[i * 3])) / Math.Sqrt((rgbValues[i * 3 + 2] -
                            rgbValues[i * 3 + 1]) * (rgbValues[i * 3 + 2] -
                            rgbValues[i * 3 + 1]) + (rgbValues[i * 3 + 2] -
                            rgbValues[i * 3]) * (rgbValues[i * 3 + 1] -
                            rgbValues[i * 3]))) / (2 * Math.PI);
                        tempD = (rgbValues[i * 3] <= rgbValues[i * 3 + 1]) ?
                            theta : (1 - theta);
                         grayValues[i] = (byte)(tempD * 255);
                    }
                    break;
                case 4:
                    // 饱和度分量
                    for (int i = 0; i < bytes; i++)
                    {
                        tempB = Math.Min(rgbValues[i * 3], rgbValues[i * 3 + 1]);
                        tempB = Math.Min(tempB, rgbValues[i * 3 + 2]);
                        tempD = 1.0 - 3.0 * tempB / (rgbValues[i * 3] +
                            rgbValues[i * 3 + 1] + rgbValues[i * 3 + 2]);
                        grayValues[i] = (byte)(tempD * 255);
                    }
                    break;
                case 5:
                    // 亮度分量
                    for (int i = 0; i < bytes; i++)
                        grayValues[i] = (byte)((rgbValues[i * 3] + rgbValues[i * 3 + 1]
                            rgbValues[i * 3 + 2]) / 3);
                    break;
                default:
                    break;
            }

            // 用灰度图像显示结果
            curBitmap = new Bitmap(curBitmap.Width, curBitmap.Height,
                System.Drawing.Imaging.PixelFormat.Format8bppIndexed);
            System.Drawing.Imaging.ColorPalette cp = curBitmap.Palette;
            for (int i = 0; i < 256; i++)
            {
                cp.Entries[i] = Color.FromArgb(i, i, i);
            }
            curBitmap.Palette = cp;
            bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            ptr = bmpData.Scan0;
            System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
            curBitmap.UnlockBits(bmpData);

            Invalidate();
        }
    }
}
```

（4）编译并运行该段程序，以图 2.4 为例。打开该图后，单击“空间转换”按钮，打开

彩色空间对话框，如图 11.1 所示，设置要显示的分量是“饱和度”，单击“确定”按钮后，主窗体内就显示出了饱和度分量的灰度图像，如图 11.2 所示。

图 11.1 “彩色空间”对话框

图 11.2 饱和度分量

11.1.3 彩色空间分量调整编程实例

该实例通过分别调整 RGB 空间的 R、G、B 分量和 HSI 空间的 H、S、I 分量来改变图像的颜色。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 11.3 所示。

表 11.3 所修改的属性

控　件	属　性	所修改内容
button1	Name	chCom
	Text	调整分量
	Location	37，196

（2）创建 1 个名为 changeCom 的 Windows 窗体，该窗体用于调整 R、G、B 和 H、S、I 分量值。在该窗体内添加 1 个 Button 控件、TabControl 控件（内含 2 个 TabPage）、6 个 Label 控件、6 个 NumericUpDown 控件和 6 个 TrackBar 控件，其中把 label1、label2、label3、trackBar1、trackBar2、trackBar3 和 numericUpDown1、numericUpDown2、numericUpDown3 放到 tabPage1 内，其余的 Label 控件、NumericUpDown 控件和 TrackBar 控件放到 tabPage2 内。其属性修改如表 11.4 所示。

表 11.4 所修改的属性

控　件	属　性	所修改内容
changCom	Text	彩色分量调整
	Size	505，390

	ControlBox	False

续表

控　　件	属　　性	所修改内容
button1	Name	close
	Text	退出
	Location	350，305
tabControl1	Size	406，258
	Location	45，25
tabPage1	Text	RGB 空间
tabPage2	Text	HSI 空间
label1	Text	红色（R）：
	Location	6，30
label2	Text	绿色（G）：
	Location	6，98
label3	Text	蓝色（B）：
	Location	6，166
trackBar1	Name	redTB
	Location	83，24
	Maximum	100
	Minimum	−100
	Size	229，45
trackBar2	Name	greenTB
	Location	83，92
	Maximum	100
	Minimum	−100
	Size	229，45
trackBar3	Name	blueTB
	Location	83，160
	Maximum	100
	Minimum	−100
	Size	229，45
numericUpDown1	Name	redUD
	Location	327，28
	Size	50，21
	Minimum	−100
numericUpDown2	Name	greenUD
	Location	327，96
	Size	50，21
	Minimum	−100
numericUpDown3	Name	blueUD
	Location	327，164
	Size	50，21
	Minimum	−100

label4	Text	色度（H）
	Location	6，30

续表

控　　件	属　　性	所修改内容
label5	Text	饱和度（S）
	Location	6，98
label6	Text	亮度（I）：
	Location	6，166
trackBar4	Name	hueTB
	Location	83，24
	Maximum	180
	Minimum	−180
	Size	229，45
trackBar5	Name	satTB
	Location	83，92
	Maximum	100
	Minimum	−100
	Size	229，45
trackBar6	Name	intTB
	Location	83，160
	Maximum	100
	Minimum	−100
	Size	229，45
numericUpDown4	Name	hueUD
	Location	327，28
	Size	50，21
	Maximum	180
	Minimum	−180
numericUpDown5	Name	satUD
	Location	327，96
	Size	50，21
	Minimum	−100
numericUpDown6	Name	intUD
	Location	327，164
	Size	50，21
	Minimum	−100

在该窗体内为 Button 控件添加 Click 事件，为 TrackBar 添加 Scroll 事件，为 NumericUpDown 添加 ValueChanged 事件，为 TabControl 添加 SelectedIndexChanged 事件，并改写构造函数，代码如下：

```
// 改写构造函数，用于传递主窗体参数
public changeCom(Form1 pF)
```

```
{
    InitializeComponent();
    masterF = pF;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

// 红色TrackBar
private void redTB_Scroll(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    redUD.Value = redTB.Value;
    // 调用主窗体函数
    masterF.adjustCom(0, redTB.Value, greenTB.Value, blueTB.Value);
}

// 红色NumericUpDown
private void redUD_ValueChanged(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    redTB.Value = (int)redUD.Value;
    // 调用主窗体函数
    masterF.adjustCom(0, redTB.Value, greenTB.Value, blueTB.Value);
}

// 绿色TrackBar
private void greenTB_Scroll(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    greenUD.Value = greenTB.Value;
    // 调用主窗体函数
    masterF.adjustCom(0, redTB.Value, greenTB.Value, blueTB.Value);
}

// 绿色NumericUpDown
private void greenUD_ValueChanged(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    greenTB.Value = (int)greenUD.Value;
    // 调用主窗体函数
    masterF.adjustCom(0, redTB.Value, greenTB.Value, blueTB.Value);
}

// 蓝色TrackBar
private void blueTB_Scroll(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    blueUD.Value = blueTB.Value;
    // 调用主窗体函数
    masterF.adjustCom(0, redTB.Value, greenTB.Value, blueTB.Value);
```

```
}

// 蓝色NumericUpDown
private void blueUD_ValueChanged(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    blueTB.Value = (int)blueUD.Value;
    // 调用主窗体函数
    masterF.adjustCom(0, redTB.Value, greenTB.Value, blueTB.Value);
}

// 色度TrackBar
private void hueTB_Scroll(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    hueUD.Value = hueTB.Value;
    // 调用主窗体函数
    masterF.adjustCom(1, hueTB.Value, satTB.Value, intTB.Value);
}

// 色度NumericUpDown
private void hueUD_ValueChanged(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    hueTB.Value = (int)hueUD.Value;
    // 调用主窗体函数
    masterF.adjustCom(1, hueTB.Value, satTB.Value, intTB.Value);
}

// 饱和度TrackBar
private void satTB_Scroll(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    satUD.Value = satTB.Value;
    // 调用主窗体函数
    masterF.adjustCom(1, hueTB.Value, satTB.Value, intTB.Value);
}

// 饱和度NumericUpDown
private void satUD_ValueChanged(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    satTB.Value = (int)satUD.Value;
    // 调用主窗体函数
    masterF.adjustCom(1, hueTB.Value, satTB.Value, intTB.Value);
}

// 亮度TrackBar
private void intTB_Scroll(object sender, EventArgs e)
{
    // TrackBar 和NumericUpDown 同步改变
    intUD.Value = intTB.Value;
    // 调用主窗体函数
```

```
        masterF.adjustCom(1, hueTB.Value, satTB.Value, intTB.Value);
    }

    // 亮度NumericUpDown
    private void intUD_ValueChanged(object sender, EventArgs e)
    {
        // TrackBar 和NumericUpDown 同步改变
        intTB.Value = (int)intUD.Value;
        // 调用主窗体函数
        masterF.adjustCom(1, hueTB.Value, satTB.Value, intTB.Value);
    }

    private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
    {
        // 翻页时，以前调整的数据仍然有效
        switch (this.tabControl1.SelectedIndex)
        {
            case 0:
                // RGB 空间
                masterF.adjustCom(0, redTB.Value, greenTB.Value, blueTB.Value);
                break;
            case 1:
                // HSI 空间
                masterF.adjustCom(1, hueTB.Value, satTB.Value, intTB.Value);
                break;
        }
    }
```

其中 masterF 定义为：

```
    private Form1 masterF;
```

adjustCom 是主窗体内的方法，用于调整彩色分量值。

（3）回到主窗体，为“调整分量”按钮添加 Click 事件，并编写 adjustCom 方法，代码如下：

```
    private void chCom_Click(object sender, EventArgs e)
    {
        if (curBitmap != null)
        {
            // 实例化changeCom，并传递主窗体参数
            changeCom adjCom = new changeCom(this);
            // 用于初始化tempArray 数组，
            adjustCom(255,0,0,0);
            // 打开从窗体
            adjCom.Show();
        }
    }

    /***********************************************
    用于调整各个彩色空间中的分量
    changTab：标识作用
              =255：初始化tempArray 数组
              =0：调整RGB 空间
```

```
          =1：调整HSI 空间
numCom1、numCom2、numCom3：各个空间中的3 个分量
*******************************************/
public void adjustCom(byte changTab, int numCom1, int numCom2, int numCom3)
{
     Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
     System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
          System.Drawing.Imaging.ImageLockMode.ReadWrite, curBitmap.PixelFormat);
     IntPtr ptr = bmpData.Scan0;
     int bytes = curBitmap.Width * curBitmap.Height;
     byte[] rgbValues = new byte[bytes * 3];
     System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes * 3);

     // 初始化tempArray 数组
     if (changTab == 255)
     {
          tempArray = new byte[bytes * 3];
          tempArray = (byte[])rgbValues.Clone();
          System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes * 3);
          curBitmap.UnlockBits(bmpData);
          return;
     }

     double valueCom1 = numCom1 / 100.0;
     double valueCom2 = numCom2 / 100.0;
     double valueCom3 = numCom3 / 100.0;
     double hue, sat, inten;
     double r, g, b;
     if (changTab == 0)
     {
          // RGB 空间

          for (int i = 0; i < bytes; i ++)
          {
               // 调整红色分量
               if (valueCom1 <= 0)
                    // 分量成分减小
                    rgbValues[i * 3 + 2] = (byte)(tempArray[i * 3 + 2] * (1.0 + valueCom1));
               else
                    // 分量成分增加
                    rgbValues[i * 3 + 2] = (byte)(tempArray[i * 3 + 2] +
                         (255.0 - tempArray[i * 3 + 2]) * valueCom1);

               // 调整绿色分量
               if (valueCom2 <= 0)
                    // 分量成分减小
                    rgbValues[i * 3 + 1] = (byte)(tempArray[i * 3 + 1] * (1.0 + valueCom2));
               else
                    // 分量成分增加
                    rgbValues[i * 3 + 1] = (byte)(tempArray[i * 3 + 1] +
                         (255.0 - tempArray[i * 3 + 1]) * valueCom2);

               // 调整蓝色分量
```

```
                if (valueCom3 <= 0)
                    // 分量成分减小
                    rgbValues[i * 3] = (byte)(tempArray[i * 3] * (1.0 + valueCom3));
                else
                    // 分量成分增加
                    rgbValues[i * 3] = (byte)(tempArray[i * 3] +
                        (255.0 - tempArray[i * 3]) * valueCom3);
            }
        }
        else
        {
            // HSI 空间

            // 重新定义色度范围
            valueCom1 = valueCom1 * Math.PI / 1.8;
            for (int i = 0; i < bytes; i++)
            {
                r = tempArray[i * 3 + 2] / 255.0;
                g = tempArray[i * 3 + 1] / 255.0;
                b = tempArray[i * 3] / 255.0;

                // RGB 空间转换为HSI 空间
                double theta = Math.Acos(0.5 * ((r - g) + (r - b)) / Math.Sqrt((r - g) * (r - g)
                    (r - b) * (g - b) + 0.0000001)) / (2 * Math.PI);

                hue = (b <= g) ? theta : (1 - theta);

                sat = 1.0 - 3.0 * Math.Min(Math.Min(r, g), b) / (r + g + b + 0.0000001);

                inten = (r + g + b) / 3.0;

                // 通过选择角度来调整色度分量
                hue = hue * 2 * Math.PI;
                hue = (hue + valueCom1 + 2 * Math.PI) % (2 * Math.PI);

                // 调整饱和度分量
                if (valueCom2 <= 0)
                    // 分量值减小
                    sat = sat * (1.0 + valueCom2);
                else
                    // 分量值增加
                    sat = sat + (1.0 - sat) * valueCom2;

                // 调整亮度分量
                if (valueCom3 <= 0)
                    // 分量值减小
                    inten = inten * (1.0 + valueCom3);
                else
                {
                    // 分量值增加
                    inten = inten + (1.0 - inten) * valueCom3;
                    if (sat == 1.0)
                        // 避免饱和度为1 时，调整亮度无意义
```

```
                    sat = sat * (1 - valueCom3);
                }

                if (sat == 0.0)
                    hue = 0;

                // HSI 空间转换为RGB 空间
                if (hue >= 0 && hue < 2 * Math.PI / 3)
                {
                    b = inten * (1 - sat);
                    r = inten * (1 + sat * Math.Cos(hue) / Math.Cos(Math.PI / 3 - hue));
                    g = 3 * inten - (r + b);
                }
                else if (hue >= 2 * Math.PI / 3 && hue < 4 * Math.PI / 3)
                {
                    r = inten * (1 - sat);
                    g = inten * (1 + sat * Math.Cos(hue - 2 * Math.PI / 3)
                        / Math.Cos(Math.PI - hue));
                    b = 3 * inten - (r + g);
                }
                else //if (h >= 4 * Math.PI / 3 && h <= 2 * Math.PI)
                {
                    g = inten * (1 - sat);
                    b = inten * (1 + sat * Math.Cos(hue - 4 * Math.PI / 3)
                        / Math.Cos(5 * Math.PI / 3 - hue));
                    r = 3 * inten - (g + b);
                }

                // 限制RGB 范围
                if (r > 1)
                    r = 1;
                if (g > 1)
                    g = 1;
                if (b > 1)
                    b = 1;

                rgbValues[i * 3 + 2] = (byte)(r * 255);
                rgbValues[i * 3 + 1] = (byte)(g * 255);
                rgbValues[i * 3] = (byte)(b * 255);

            }
        }

        System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes * 3);
        curBitmap.UnlockBits(bmpData);

        Invalidate();
    }
```

其中 tempArray 为 byte 型数组变量，用于保留图像的原始数据。

（4）编译并运行该段程序，仍以图 2.4 为例。打开该图后，单击“调整分量”按钮，打开彩色分量调整对话框。如图 11.3 所示，设置相关参数。在调整过程中，主窗体内的图像会

随着参数的改变而变化，最后的结果如图 11.4 所示。

然后再单击彩色分量调整对话框中的“HSI 空间”标签，则显示 HSI 空间分量调整页面如图 11.5 所示，设置相关参数，调整后的结果如图 11.6 所示。

图 11.3 “彩色分量调整”对话框

图 11.4 RGB 空间分量调整结果

图 11.5 HSI 空间分量调整

图 11.6 HSI 空间分量调整结果

11.2 伪彩色处理

11.2.1 伪彩色处理原理

伪彩色处理是指将灰度图像转化为彩色图像。由于人眼对彩色的分辨能力远远高于对灰度的分辨能力，所以将灰度图像转化成彩色表示，就可以提高对图像细节的辨别力。因此，伪彩色处理的主要目的是为了提高人眼对图像的细节分辨能力，以达到图像增强的目的。

在空间域内，伪彩色处理的主要方法有：强度分层法和灰度级-彩色变换法。

（1）强度分层法是伪彩色处理技术中最简单的一种。

设一幅灰度图像 $f(x, y)$，在某一个灰度级如 $f(x, y)=L_i$ 上设置一个平行于 x–y 平面的切

平面，切割平面下面的，即灰度级小于 L_i 的像素分配给一种颜色，相应的切割平面上面的，即灰度级大于 L_i 的像素分配给另一种颜色。这样切割结果就可以将灰度图像变为只有两个颜色的伪彩色图像。若将灰度图像的灰度级用 M 个切割平面去切割，就会得到 M 个不同灰度级的区域。对这 M 个区域中的像素人为地分配给 M 个不同颜色，这样就可以得到具有 M 种颜色的伪彩色图像。这种方法虽然简单，但视觉效果不理想，量化噪声也大。

（2）灰度级-彩色变换法可以将灰度图像变为具有多种颜色渐变的连续彩色图像。

它是将灰度图像送入具有不同变换特性的红、绿、蓝 3 个变换器，然后再将 3 个变换器的不同输出分别送到彩色显像管的红、绿、蓝枪，再合成某种颜色。同一灰度由于 3 个变换器对其实施不同变换，使 3 个变换器输出不同，从而不同大小灰度级可以合成不同的颜色。一组典型的灰度级-彩色变换的传递函数如图 11.7 所示，其中图（a）、（b）、（c）分别表示红色、绿色、蓝色的传递函数，图（d）是 3 种彩色传递函数组合在一起的情况。

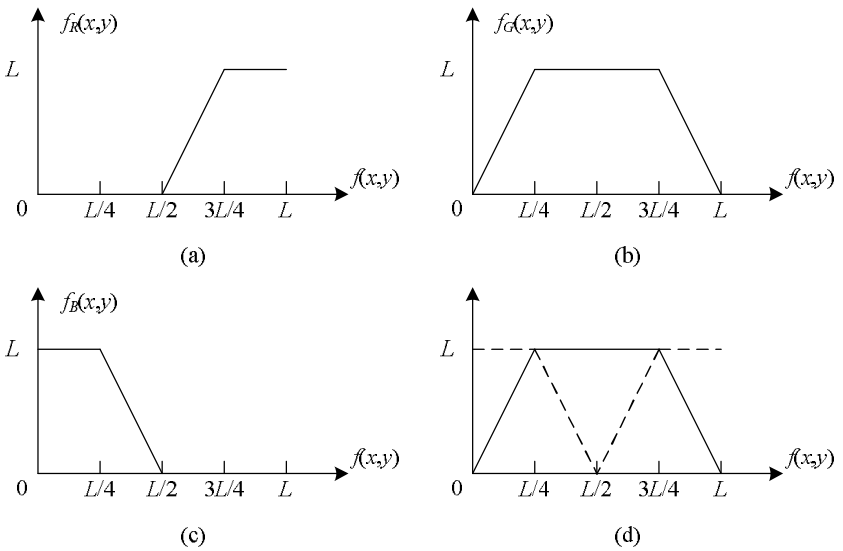


图 11.7　典型的传递函数

11.2.2　伪彩色处理编程实例

该实例实现了强度分层法和灰度级-彩色变换法的伪彩色图像处理。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 11.5 所示。

表 11.5　　所修改的属性

控　件	属　性	所修改内容
button1	Name	pseudoC
	Text	伪彩色处理
	Location	37，242

（2）创建 1 个名为 pColor 的 Windows 窗体，该窗体用于选择应用何种伪彩色图像处理方法。在该窗体内添加 2 个 Button 控件、一个 GroupBox 控件、2 个 RadioButton 控件、1 个

abel 控件和 1 个 ComboBox 控件，其属性修改如表 11.6 所示。

表 11.6　所修改的属性

控　件	属　性	所修改内容
pColor	Text	伪彩色处理
	Size	370，270
	ControlBox	False
button1	Name	start
	Text	确定
	Location	62，187
button2	Name	close
	Text	退出
	Location	218，187
groupBox1	Text	伪彩色处理方法
	Location	31，23
	Size	303，137
radioButton1	Name	intSeg
	Text	强度分层法
	Location	31，35
	Checked	True
radioButton2	Name	gcTrans
	Text	灰度级-彩色变换法
	Location	31，90
label1	Text	分层数：
	Location	134，37
comboBox1	Name	intNum
	Location	208，33
	Size	71.20
	DropDownStyle	DropDownList

为 2 个 Button 控件添加 Click 事件，为 RadioButton 控件添加 CheckedChanged 事件，并手添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    numSeg = (byte)(Math.Pow(2, 2 + intNum.SelectedIndex));
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

private void intSeg_CheckedChanged(object sender, EventArgs e)
{
    pseMethod = false;
    intNum.Enabled = true;
}
```

```
private void gcTrans_CheckedChanged(object sender, EventArgs e)
{
    pseMethod = true;
    intNum.Enabled = false;
}

public bool GetMethod
{
    get
    {
        // 得到使用何种方法
        return pseMethod;
    }
}

public byte GetSeg
{
    get
    {
        // 得到强度分层法中的分层数
        return numSeg;
    }
}
```

其中 pseMethod 和 numSeg 分别为 bool 型和 byte 型变量，它们在构造函数内被初始化，并且在构造函数内再为 ComboBox 控件添加相关的可选项，代码如下：

```
pseMethod = false;
numSeg = 4;
intNum.Items.Add("分 4 层");
intNum.Items.Add("分 8 层");
intNum.Items.Add("分 16 层");
intNum.Items.Add("分 32 层");
// 默认被选项
intNum.SelectedIndex = 0;
```

（3）回到主窗体，为“伪彩色处理”按钮添加 Click 事件，代码如下：

```
private void pseudoC_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化pColor
        pColor pc = new pColor();

        if (pc.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] grayValues = new byte[bytes];
            System.Runtime.InteropServices.Marshal.Copy(ptr, grayValues, 0, bytes);
            curBitmap.UnlockBits(bmpData);

            // 得到何种方法进行伪彩色处理
            bool method = pc.GetMethod;
```

```
// 得到强度分层法中的分层数
byte seg = pc.GetSeg;
byte[] rgbValues = new byte[bytes * 3];
// 清零
Array.Clear(rgbValues, 0, bytes * 3);
byte tempB;

if (method == false)
{
    // 强度分层法

    for (int i = 0; i < bytes; i++)
    {
        byte ser = (byte)(256 / seg);
        tempB = (byte)(grayValues[i] / ser);
        // 分配任意一种指定的颜色
        rgbValues[i * 3 + 1] = (byte)(tempB * ser);
        rgbValues[i * 3] = (byte)((seg - 1 - tempB) * ser);
        rgbValues[i * 3 + 2] = 0;
    }
}
else
{
    // 灰度级-彩色变换法

    for (int i = 0; i < bytes; i++)
    {
        if (grayValues[i] < 64)
        {
            // 小于L/4
            rgbValues[i * 3 + 2] = 0;
            rgbValues[i * 3 + 1] = (byte)(4 * grayValues[i]);
            rgbValues[i * 3] = 255;
        }
        else if (grayValues[i] < 128)
        {
            // 大于L/4，而小于L/2
            rgbValues[i * 3 + 2] = 0;
            rgbValues[i * 3 + 1] = 255;
            rgbValues[i * 3] = (byte)(-4 * grayValues[i] + 2 * 255);
        }
        else if (grayValues[i] < 192)
        {
            // 大于L/2，而小于3L/4
            rgbValues[i * 3 + 2] = (byte)(4 * grayValues[i] - 2 * 255);
            rgbValues[i * 3 + 1] = 255;
            rgbValues[i * 3] = 0;
        }
        else
        {
            // 大于3L/4
            rgbValues[i * 3 + 2] = 255;
            rgbValues[i * 3 + 1] = (byte)(-4 * grayValues[i] + 4 * 255)
            rgbValues[i * 3] = 0;
        }
    }
}

curBitmap = new Bitmap(curBitmap.Width, curBitmap.Height,
    System.Drawing.Imaging.PixelFormat.Format24bppRgb);
```

```
                bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                ptr = bmpData.Scan0;
                System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes * 3)
                curBitmap.UnlockBits(bmpData);

                Invalidate();
            }
        }
    }
```

（4）编译并运行该段程序，以图 4.1 为例。打开该图后，单击“伪彩色处理”按钮，打开伪彩色处理对话框，如图 11.8 所示。图 11.9 为用强度分层法，分层数为 16 的伪彩色图像。图 11.10 为用灰度级-彩色变换法处理的伪彩色图像。

图 11.8 “伪彩色处理”对话框

图 11.9 强度分层法

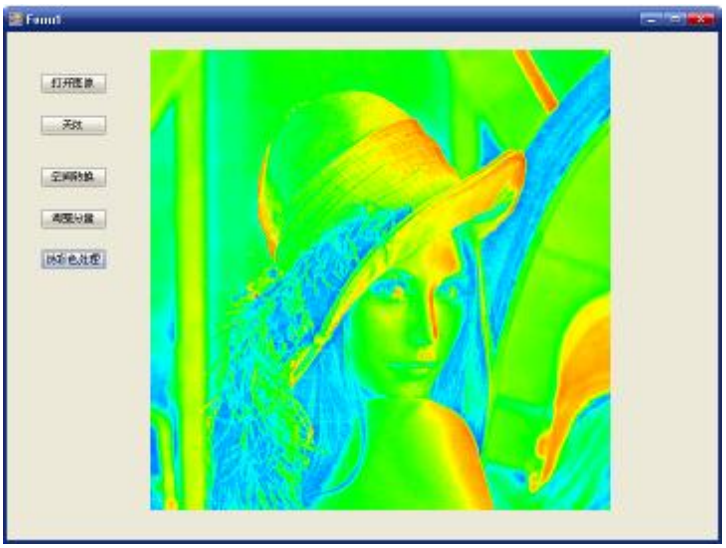


图 11.10 灰度级-彩色变换法

11.3 彩色图像直方图均衡化

11.3.1 彩色图像直方图均衡化原理

在第 3 章介绍了灰度图像的直方图均衡化方法，它的目的是产生具有均匀的灰度直方图。在彩色图像处理中，直方图均衡化有两种方法：

（1）在 RGB 空间内独立地进行每个彩色图像分量的直方图均衡化，但这种方法的处理效果不好，会产生错误的颜色；

（2）在 HSI 空间内仅对亮度进行直方图均衡化处理，而保存色度和饱和度不变，该方法要比第 1 种方法更好一些。

11.3.2 彩色图像直方图均衡化编程实例

该实例对上述两种方法都通过编程来实现。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 11.7 所示。

表 11.7　　所修改的属性

控　件	属　性	所修改内容
button1	Name	equC
	Text	直方图均衡
	Location	37，288

（2）创建 1 个名为 eColor 的 Windows 窗体，该窗体用于选择哪种彩色图像直方图均衡化方法。在该窗体内添加 2 个 Button 控件、2 个 RadioButton 控件，其属性修改如表 11.8 所示。

表 11.8　　所修改的属性

控　件	属　性	所修改内容
eColor	Text	彩色图像直方图均衡化
	Size	234，200
	ControlBox	False
button1	Name	start
	Text	确定
	Location	23，117
button2	Name	close
	Text	退出
	Location	119，117
radioButton1	Name	rgbEqu
	Text	RGB 空间分量均衡化法
	Location	33，34
	Checked	True
radioButton2	Name	intEqu
	Text	HSI 空间亮度均衡化法

	Location	33，75

为该窗体内的两个 Button 添加 Click 事件，并添加一个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetMethod
{
    get
    {
         // 得到均衡化方法
         return intEqu.Checked;
    }
}
```

（3）回到主窗体，为“直方图均衡”按钮添加 Click 事件，并如第 3 章所介绍的那样编写直方图均衡化方法，以便程序调用，代码如下：

```
private void equC_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化eColor
         eColor equC = new eColor();

         if (equC.ShowDialog() == DialogResult.OK)
         {
              Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
              System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                  System.Drawing.Imaging.ImageLockMode.ReadWrite,
                  curBitmap.PixelFormat);
              IntPtr ptr = bmpData.Scan0;
              int bytes = curBitmap.Width * curBitmap.Height;
              byte[] rgbValues = new byte[bytes * 3];
              System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes * 3)

              // 得到方法
              bool method = equC.GetMethod;
              if (method == false)
              {
                  // RGB 分量法

                  byte[] rValues = new byte[bytes];
                  byte[] gValues = new byte[bytes];
                  byte[] bValues = new byte[bytes];

                  // 取值
                  for (int i = 0; i < bytes; i++)
```

```
        {
            rValues[i] = rgbValues[i * 3 + 2];
            gValues[i] = rgbValues[i * 3 + 1];
            bValues[i] = rgbValues[i * 3];
        }

        // 调用灰度图像均衡化方法
        rValues = equalization(rValues);
        gValues = equalization(gValues);
        bValues = equalization(bValues);

        // 赋值
        for (int i = 0; i < bytes; i++)
        {
            rgbValues[i * 3 + 2] = rValues[i];
            rgbValues[i * 3 + 1] = gValues[i];
            rgbValues[i * 3] = bValues[i];
        }
    }
    else
    {
        // 亮度法

        double[] hue = new double[bytes];
        double[] sat = new double[bytes];
        byte[] inten = new byte[bytes];
        double r, g, b;

        // RGB 空间转换为HSI 空间
        for (int i = 0; i < bytes; i++)
        {
            r = rgbValues[i * 3 + 2];
            g = rgbValues[i * 3 + 1];
            b = rgbValues[i * 3];

            double theta = Math.Acos(0.5 * ((r - g) + (r - b)) /
                Math.Sqrt((r - g) * (r - g) + (r - b) * (g - b) + 1)) / (2 * Math.PI)

            hue[i] = ((b <= g) ? theta : (1 - theta));

            sat[i] = 1.0 - 3.0 * Math.Min(Math.Min(r, g), b) / (r + g + b + 1)

            inten[i] = (byte)((r + g + b) / 3);
        }

        // 仅对亮度进行均衡化处理
        inten = equalization(inten);

        // HSI 空间转换为RGB 空间
        for (int i = 0; i < bytes; i++)
        {
            r = rgbValues[i * 3 + 2];
            g = rgbValues[i * 3 + 1];
            b = rgbValues[i * 3];
```

```
                        hue[i] = hue[i] * 2 * Math.PI;
                        if (hue[i] >= 0 && hue[i] < 2 * Math.PI / 3)
                        {
                            b = inten[i] * (1 - sat[i]);
                            r = inten[i] * (1 + sat[i] * Math.Cos(hue[i]) /
                                 Math.Cos(Math.PI / 3 - hue[i]));
                            g = 3 * inten[i] - (r + b);
                        }
                        else if (hue[i] >= 2 * Math.PI / 3 && hue[i] < 4 * Math.PI / 3)
                        {
                            r = inten[i] * (1 - sat[i]);
                            g = inten[i] * (1 + sat[i] * Math.Cos(hue[i] - 2 * Math.PI / 3)
                                 Math.Cos(Math.PI - hue[i]));
                            b = 3 * inten[i] - (r + g);
                        }
                        else //if (h >= 4 * Math.PI / 3 && h <= 2 * Math.PI)
                        {
                            g = inten[i] * (1 - sat[i]);
                            b = inten[i] * (1 + sat[i] * Math.Cos(hue[i] - 4 * Math.PI / 3)
                                 Math.Cos(5 * Math.PI / 3 - hue[i]));
                            r = 3 * inten[i] - (g + b);
                        }
                        if (r > 255)
                            r = 255;
                        if (g > 255)
                            g = 255;
                        if (b > 255)
                            b = 255;

                        rgbValues[i * 3 + 2] = (byte)r;
                        rgbValues[i * 3 + 1] = (byte)g;
                        rgbValues[i * 3] = (byte)b;
                    }
                }

                System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes * 3)
                curBitmap.UnlockBits(bmpData);

                Invalidate();
            }
        }
    }

    /*********************************************
    灰度图像均衡化方法
    colorArray: 输入byte 型灰度值数组
    输出为已灰度均衡化的byte 型数组
    *********************************************/
    private byte[] equalization(byte[] colorArray)
    {
        byte[] comValues = new byte[colorArray.Length];
        comValues = (byte[])colorArray.Clone();
        int[] countPixel = new int[256];
        byte temp;
        int[] tempArray = new int[256];
```

```
        Array.Clear(tempArray, 0, 256);
        byte[] pixelMap = new byte[256];
        for (int i = 0; i < comValues.Length; i++)
        {
            temp = comValues[i];
            countPixel[temp]++;
        }

        // 计算各灰度级的累计分布函数
        for (int i = 0; i < 256; i++)
        {
            if (i != 0)
            {
                tempArray[i] = tempArray[i - 1] + countPixel[i];
            }
            else
            {
                tempArray[0] = countPixel[0];
            }

            pixelMap[i] = (byte)(255.0 * tempArray[i] / comValues.Length + 0.5);
        }

        // 灰度等级映射处理
        for (int i = 0; i < comValues.Length; i++)
        {
            temp = comValues[i];
            comValues[i] = pixelMap[temp];
        }
        return comValues;
    }
```

（4）编译并运行该段程序，仍以图 2.4 为例。打开该图后，单击“直方图均衡”按钮，打开彩色图像直方图均衡化对话框，选择“HSI 空间亮度均衡化法”，如图 11.11 所示。单击“确定”按钮后，则完成彩色图像直方图均衡化，其结果如图 11.12 所示。

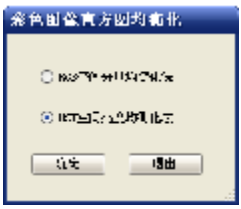


图 11.11　“彩色图像直方图均衡化”对话框

图 11.12　彩色图像直方图均衡化结果

11.4 彩色图像平滑处理

11.4.1 彩色图像平滑处理原理

第 7 章介绍的灰度图像的平滑处理可以看做是对图像的滤波处理，而用得最多的是均值滤波。把这一概念扩展到彩色图像处理中，所不同的是用彩色空间分量代替灰度值。

在 RGB 空间内，平滑处理的计算公式为：

$$\bar{C}(x,y)=\begin{bmatrix}\dfrac{1}{k}\displaystyle\sum_{(x,y)\in S_{xy}}R(x,y)\\ \dfrac{1}{k}\displaystyle\sum_{(x,y)\in S_{xy}}G(x,y)\\ \dfrac{1}{k}\displaystyle\sum_{(x,y)\in S_{xy}}B(x,y)\end{bmatrix} \tag{11.8}$$

其中，$\bar{C}(x,y)$ 为平滑后的彩色图像，$R(x,y)$、$G(x,y)$、$B(x,y)$分别表示 R、G、B 彩色分量，k 为邻域 S_{xy} 中的像素点数。

在 HSI 空间内，只需对亮度分量进行平滑处理即可。

11.4.2 彩色图像平滑处理编程实例

本实例可以在 RGB 空间和 HSI 空间内，通过选取不同的邻域来对彩色图像进行平滑处理。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 11.9 所示。

表 11.9 所修改的属性

控　　件	属　　性	所修改内容
button1	Name	smoC
	Text	平滑处理
	Location	37，334

（2）创建 1 个名为 smoothColor 的 Windows 窗体，该窗体用于选择是在 RGB 空间还是在 HSI 空间内进行平滑处理，还有处理所需要的窗体大小。在该窗体内添加 2 个 Button 控件、2 个 RadioButton 控件、1 个 Label 控件和 1 个 NumericUpDown 控件，其属性修改如表 11.10 所示。

表 11.10 所修改的属性

控　　件	属　　性	所修改内容
smoothColor	Text	彩色图像平滑处理
	Size	391，242
	ControlBox	False
button1	Name	start
	Text	确定

	Location	57，157

续表

控　　件	属　　性	所修改内容
button2	Name	close
	Text	退出
	Location	244，157
radioButton1	Name	rgbS
	Text	RGB 空间平滑处理
	Location	27，42
	Checked	True
radioButton2	Name	hsiS
	Text	HSI 空间平滑处理
	Location	33，75
numericUpDown1	Name	lengths
	Location	267，69
	Size	57，21
	Maximum	7
	Minimum	3
	Value	3
label1	Text	领域模板边长
	Location	168，71

为 2 个 Button 控件添加 Click 事件，并添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetMethod
{
    get
    {
        // 得到方法
        return hsiS.Checked;
    }
}

public byte GetLength
{
    get
    {
        // 得到窗体大小
        return (byte)lengthS.Value;
```

```
    }
}
```

（3）回到主窗体，为“平滑处理”按钮添加 Click 事件，并如第 7 章所介绍的那样编写均值平滑处理方法，以便程序调用，代码如下：

```
private void smoC_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
        // 实例化smoothColor
        smoothColor smoothC = new smoothColor();

        if (smoothC.ShowDialog() == DialogResult.OK)
        {
            Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
            System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                System.Drawing.Imaging.ImageLockMode.ReadWrite,
                curBitmap.PixelFormat);
            IntPtr ptr = bmpData.Scan0;
            int bytes = curBitmap.Width * curBitmap.Height;
            byte[] rgbValues = new byte[bytes * 3];
            System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes * 3)

            // 得到何种方法
            bool method = smoothC.GetMethod;
            // 得到邻域模板边长
            byte sideL = smoothC.GetLength;

            if (method == false)
            {
                // RGB 空间

                byte[] rValues = new byte[bytes];
                byte[] gValues = new byte[bytes];
                byte[] bValues = new byte[bytes];

                for (int i = 0; i < bytes; i++)
                {
                    rValues[i] = rgbValues[i * 3 + 2];
                    gValues[i] = rgbValues[i * 3 + 1];
                    bValues[i] = rgbValues[i * 3];
                }

                // RGB 三个分量调用平滑函数
                rValues = smooth(rValues, sideL);
                gValues = smooth(gValues, sideL);
                bValues = smooth(bValues, sideL);

                for (int i = 0; i < bytes; i++)
                {
                    rgbValues[i * 3 + 2] = rValues[i];
                    rgbValues[i * 3 + 1] = gValues[i];
                    rgbValues[i * 3] = bValues[i];
                }
            }
```

```
                else
                {
                    // HSI空间

                    double[] hue = new double[bytes];
                    double[] sat = new double[bytes];
                    byte[] inten = new byte[bytes];
                    double r, g, b;

                    // RGB空间转换为HSI空间
                    for (int i = 0; i < bytes; i++)
                    {
                        r = rgbValues[i * 3 + 2];
                        g = rgbValues[i * 3 + 1];
                        b = rgbValues[i * 3];

                        double theta = Math.Acos(0.5 * ((r - g) + (r - b)) /
                            Math.Sqrt((r - g) * (r - g) + (r - b) * (g - b) + 1)) / (2 * Math.PI)

                        hue[i] = ((b <= g) ? theta : (1 - theta));

                        sat[i] = 1.0 - 3.0 * Math.Min(Math.Min(r, g), b) / (r + g + b + 1)

                        inten[i] = (byte)((r + g + b) / 3);
                    }

                    // 亮度分量调用平滑方法
                    inten = smooth(inten, sideL);

                    // HSI空间转换为RGB空间
                    for (int i = 0; i < bytes; i++)
                    {
                        r = rgbValues[i * 3 + 2];
                        g = rgbValues[i * 3 + 1];
                        b = rgbValues[i * 3];

                        hue[i] = hue[i] * 2 * Math.PI;
                        if (hue[i] >= 0 && hue[i] < 2 * Math.PI / 3)
                        {
                            b = inten[i] * (1 - sat[i]);
                            r = inten[i] * (1 + sat[i] * Math.Cos(hue[i]) /
                                Math.Cos(Math.PI / 3 - hue[i]));
                            g = 3 * inten[i] - (r + b);
                        }
                        else if (hue[i] >= 2 * Math.PI / 3 && hue[i] < 4 * Math.PI / 3)
                        {
                            r = inten[i] * (1 - sat[i]);
                            g = inten[i] * (1 + sat[i] * Math.Cos(hue[i] - 2 * Math.PI / 3)
                                Math.Cos(Math.PI - hue[i]));
                            b = 3 * inten[i] - (r + g);
                        }
                        else //if (h >= 4 * Math.PI / 3 && h <= 2 * Math.PI)
                        {
                            g = inten[i] * (1 - sat[i]);
                            b = inten[i] * (1 + sat[i] * Math.Cos(hue[i] - 4 * Math.PI / 3)
```

```
                                Math.Cos(5 * Math.PI / 3 - hue[i]));
                        r = 3 * inten[i] - (g + b);
                    }
                    if (r > 255)
                        r = 255;
                    if (g > 255)
                        g = 255;
                    if (b > 255)
                        b = 255;

                    rgbValues[i * 3 + 2] = (byte)r;
                    rgbValues[i * 3 + 1] = (byte)g;
                    rgbValues[i * 3] = (byte)b;
                }
            }

            System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes * 3);
            curBitmap.UnlockBits(bmpData);

            Invalidate();
        }
    }
}

/*********************************************
灰度值平滑函数
comValues: 输入待处理的分量数组数据
sideLength: 输入邻域模板边长
输出平滑后的分量数组数据
*********************************************/
private byte[] smooth(byte[] comValues, byte sideLength)
{
    byte halfLength = (byte)(sideLength / 2);
    byte[] comS = new byte[comValues.Length];
    comS = (byte[])comValues.Clone();
    byte[] comD = new byte[comS.Length];
    Array.Clear(comD, 0, comD.Length);

    switch (sideLength)
    {
        case 3:
            // 3×3 模板
            for (int i = 0; i < curBitmap.Height; i++)
            {
                for (int j = 0; j < curBitmap.Width; j++)
                {
    comD[i * curBitmap.Width + j] = (byte)((comS[i  * curBitmap.Width + j] +
        comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
        comS[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
        comS[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
        comS[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] +
        comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] +
        comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] +
```

```
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)]) / 9);
                    }
                }
                break;
            case 5:
                // 5×5 模板
                for (int i = 0; i < curBitmap.Height; i++)
                {
                    for (int j = 0; j < curBitmap.Width; j++)
                    {
        comD[i * curBitmap.Width + j] =
            (byte)((comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2)) % curBitmap.Width] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1)) % curBitmap.Width] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[i * curBitmap.Width + (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[i * curBitmap.Width + j] +
            comS[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            comS[i * curBitmap.Width + ((j + 2) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
```

```
                ((j + 2) % curBitmap.Width)]) / 25);
                    }
                }
                break;
            case 7:
                // 7×7 模板
                for (int i = 0; i < curBitmap.Height; i++)
                {
                    for (int j = 0; j < curBitmap.Width; j++)
                    {
    comD[i * curBitmap.Width + j] =
            (byte)((comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2)) % curBitmap.Width] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1)) % curBitmap.Width] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[i * curBitmap.Width + (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[i * curBitmap.Width + (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[i * curBitmap.Width + j] +
            comS[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            comS[i * curBitmap.Width + ((j + 2) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
```

```
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            comS[((Math.Abs(i - 3)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            comS[i * curBitmap.Width + ((j + 3) % curBitmap.Width)] +
            comS[i * curBitmap.Width + (Math.Abs(j - 3) % curBitmap.Width)] +
            comS[((i + 1) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            comS[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            comS[((i + 2) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            comS[((Math.Abs(i - 2)) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)] +
            comS[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            comS[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 2) % curBitmap.Width)] +
            comS[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 2) % curBitmap.Width)] +
            comS[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 1) % curBitmap.Width)] +
            comS[((i + 3) % curBitmap.Height) * curBitmap.Width + j] +
            comS[((i + 3) % curBitmap.Height) * curBitmap.Width +
                ((j + 3) % curBitmap.Width)] +
            comS[((i + 3) % curBitmap.Height) * curBitmap.Width +
                (Math.Abs(j - 3) % curBitmap.Width)]) / 49);
                    }
                }
                break;
            default:
                break;
        }
        return comD;
    }
```

（4）编译并运行该段程序，仍以图 2.4 为例。打开该图后，单击“平滑处理”按钮，打

开彩色图像平滑处理对话框，如图 11.13 所示设置相关参数，单击“确定”按钮后，经过 5×5 模板对亮度分量进行平滑处理后的结果便呈现出来，如图 11.14 所示。

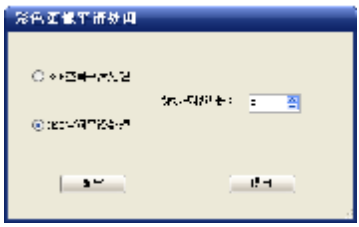


图 11.13　“彩色图像平滑处理”对话框

图 11.14　彩色图像平滑处理结果

11.5　彩色图像锐化处理

11.5.1　彩色图像锐化处理原理

锐化处理与平滑处理相反，它的目的是使图像的细节部分更加突出，最常用的是拉普拉斯方法。与平滑处理相似，它可以对 RGB 空间中的 R、G、B 三个分量分别进行锐化处理，也可以对 HSI 空间中的亮度分量单独进行锐化处理。拉普拉斯锐化处理公式为：

$$g(x,y) = 5f(x,y) - [f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1)] \qquad (11.9)$$

其中，$f(x,y)$和 $g(x,y)$分别为点(x,y)在锐化处理前、后的灰度值。

11.5.2　彩色图像锐化处理编程实例

该实例实现了彩色图像的锐化处理功能。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 11.11 所示。

表 11.11　所修改的属性

控　件	属　性	所修改内容
button1	Name	shaC
	Text	锐化处理
	Location	37，380

（2）创建 1 个名为 sharpColor 的 Windows 窗体，该窗体用于选择锐化方法。在该窗体添加 2 个 Button 控件、2 个 RadioButton 控件，其属性修改如表 11.12 所示。

表 11.12 所修改的属性

控　件	属　性	所修改内容
sharpColor	Text	彩色图像锐化处理
	Size	234，200
	ControlBox	False
button1	Name	start
	Text	确定
	Location	23，117
button2	Name	close
	Text	退出
	Location	119，117
radioButton1	Name	rgbSha
	Text	RGB 空间锐化处理
	Location	33，34
	Checked	True
radioButton2	Name	intSha
	Text	HSI 空间锐化处理
	Location	33，75

为该窗体内的两个 Button 添加 Click 事件，并添加 1 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public bool GetMethod
{
    get
    {
        // 得到锐化方法
        return hsiSha.Checked;
    }
}
```

（3）回到主窗体，为“锐化处理”按钮添加 Click 事件，并编写锐化处理方法，以便程序调用，代码如下：

```
private void shaC_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
```

```
            // 实例化sharpColor
            sharpColor sharpC = new sharpColor();

            if (sharpC.ShowDialog() == DialogResult.OK)
            {
                Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
                System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                    System.Drawing.Imaging.ImageLockMode.ReadWrite,
                    curBitmap.PixelFormat);
                IntPtr ptr = bmpData.Scan0;
                int bytes = curBitmap.Width * curBitmap.Height;
                byte[] rgbValues = new byte[bytes * 3];
                System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes * 3)

                // 得到应用何种方法
                bool method = sharpC.GetMethod;

                if (method == false)
                {
                    // RGB空间

                    byte[] rValues = new byte[bytes];
                    byte[] gValues = new byte[bytes];
                    byte[] bValues = new byte[bytes];

                    for (int i = 0; i < bytes; i++)
                    {
                        rValues[i] = rgbValues[i * 3 + 2];
                        gValues[i] = rgbValues[i * 3 + 1];
                        bValues[i] = rgbValues[i * 3];
                    }

                    // 调用锐化处理方法
                    rValues = sharp(rValues);
                    gValues = sharp(gValues);
                    bValues = sharp(bValues);

                    for (int i = 0; i < bytes; i++)
                    {
                        rgbValues[i * 3 + 2] = rValues[i];
                        rgbValues[i * 3 + 1] = gValues[i];
                        rgbValues[i * 3] = bValues[i];
                    }
                }
                else
                {
                    // HSI空间

                    double[] hue = new double[bytes];
                    double[] sat = new double[bytes];
                    byte[] inten = new byte[bytes];
                    double r, g, b;

                    // RGB空间转换为HSI空间
                    for (int i = 0; i < bytes; i++)
                    {
                        r = rgbValues[i * 3 + 2];
```

```
                g = rgbValues[i * 3 + 1];
                b = rgbValues[i * 3];

                double theta = Math.Acos(0.5 * ((r - g) + (r - b)) /
                    Math.Sqrt((r - g) * (r - g) + (r - b) * (g - b) + 1)) / (2 * Math.PI)

                hue[i] = ((b <= g) ? theta : (1 - theta));

                sat[i] = 1.0 - 3.0 * Math.Min(Math.Min(r, g), b) / (r + g + b + 1)

                inten[i] = (byte)((r + g + b) / 3);
            }

            // 调用锐化函数
            inten = sharp(inten);

            // HSI 空间转换为RGB 空间
            for (int i = 0; i < bytes; i++)
            {
                r = rgbValues[i * 3 + 2];
                g = rgbValues[i * 3 + 1];
                b = rgbValues[i * 3];

                hue[i] = hue[i] * 2 * Math.PI;
                if (hue[i] >= 0 && hue[i] < 2 * Math.PI / 3)
                {
                    b = inten[i] * (1 - sat[i]);
                    r = inten[i] * (1 + sat[i] * Math.Cos(hue[i]) /
                         Math.Cos(Math.PI / 3 - hue[i]));
                    g = 3 * inten[i] - (r + b);
                }
                else if (hue[i] >= 2 * Math.PI / 3 && hue[i] < 4 * Math.PI / 3)
                {
                    r = inten[i] * (1 - sat[i]);
                    g = inten[i] * (1 + sat[i] * Math.Cos(hue[i] - 2 * Math.PI / 3)
                         Math.Cos(Math.PI - hue[i]));
                    b = 3 * inten[i] - (r + g);
                }
                else //if (h >= 4 * Math.PI / 3 && h <= 2 * Math.PI)
                {
                    g = inten[i] * (1 - sat[i]);
                    b = inten[i] * (1 + sat[i] * Math.Cos(hue[i] - 4 * Math.PI / 3)
                         Math.Cos(5 * Math.PI / 3 - hue[i]));
                    r = 3 * inten[i] - (g + b);
                }
                if (r > 255)
                    r = 255;
                if (g > 255)
                    g = 255;
                if (b > 255)
                    b = 255;

                rgbValues[i * 3 + 2] = (byte)r;
                rgbValues[i * 3 + 1] = (byte)g;
                rgbValues[i * 3] = (byte)b;
            }
        }
```

```
                System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes * 3)
                curBitmap.UnlockBits(bmpData);

                Invalidate();
            }
        }
    }

    /***********************************************
    灰度值锐化函数
    comArray: 输入待处理的分量数组数据
    输出锐化后的分量数组数据
    ***********************************************/
    private byte[] sharp(byte[] comArray)
    {
        byte[] comValues = new byte[comArray.Length];

        for (int i = 0; i < curBitmap.Height; i++)
        {
            for (int j = 0; j < curBitmap.Width; j++)
            {
                // 公式(11.2)
        comValues[i * curBitmap.Width + j] = (byte)(5 * comArray[i * curBitmap.Width + j]
                (comArray[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            comArray[((i + 1) % curBitmap.Height) * curBitmap.Width + j] +
            comArray[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            comArray[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)]));
            }
        }

        return comValues;
    }
```

（4）编译并运行该段程序，对上一节平滑后的图 11.14 做锐化处理。单击“锐化处理”按钮后，打开彩色图像锐化处理对话框，如图 11.15 所示选择“HSI 锐化处理”，单击“确定”按钮后，锐化处理后的结果便呈现出来，如图 11.16 所示。

图 11.15 “锐化处理”对话框

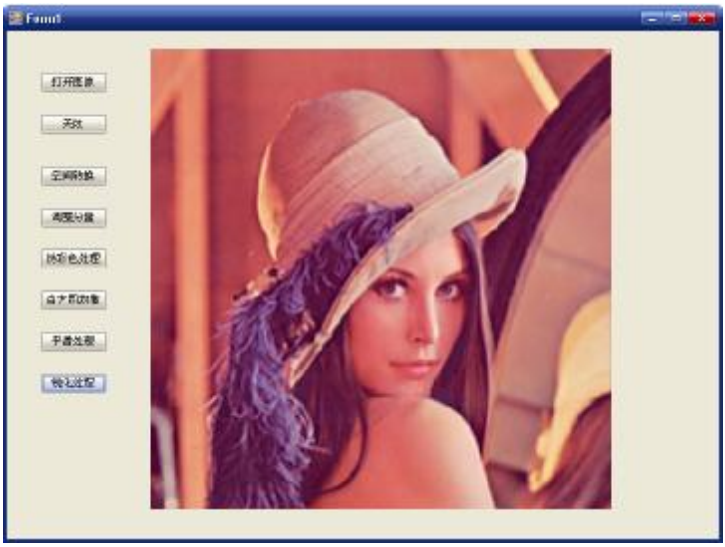


图 11.16 锐化处理结果

11.6　彩色图像边缘检测

11.6.1　彩色图像边缘检测原理

彩色图像的边缘检测比灰度图像复杂，因为它不仅要考虑亮度，还要考虑各种不同颜色E这里介绍 3 种彩色图像的边缘检测方法。

1．向量法Ⅰ

彩色图像中每一点像素由 RGB 三部分组成，则可以把它看成是一个向量。标量函数 f(x)的梯度是一个在坐标(x, y)处 f 的最大变化率方向上的向量。

令 **r**、**g**、**b** 是 RGB 彩色空间沿 R、G、B 轴的单位向量，并定义向量为：

$$\begin{aligned} \mathrm{u} &= \frac{\partial R}{\partial x}\mathrm{r} + \frac{\partial G}{\partial x}\mathrm{g} + \frac{\partial B}{\partial x}\mathrm{b} \\ \mathrm{v} &= \frac{\partial R}{\partial y}\mathrm{r} + \frac{\partial G}{\partial y}\mathrm{g} + \frac{\partial G}{\partial y}\mathrm{b} \end{aligned} \tag{11.10}$$

令 g_{xx}，g_{yy} 和 g_{xy} 是这些向量的点积：

$$\begin{aligned} g_{xx} &= \left|\frac{\partial R}{\partial x}\right|^2 + \left|\frac{\partial G}{\partial x}\right|^2 + \left|\frac{\partial B}{\partial x}\right|^2 \\ g_{yy} &= \left|\frac{\partial R}{\partial y}\right|^2 + \left|\frac{\partial G}{\partial y}\right|^2 + \left|\frac{\partial B}{\partial y}\right|^2 \\ g_{xy} &= \frac{\partial R}{\partial x}\frac{\partial R}{\partial y} + \frac{\partial G}{\partial x}\frac{\partial G}{\partial y} + \frac{\partial B}{\partial x}\frac{\partial B}{\partial y} \end{aligned} \tag{11.11}$$

那么最大变化率的方向角度为：

$$q(x, y) = \frac{1}{2}\arctan\left[\frac{2g_{xy}}{(g_{xx} - g_{yy})}\right] \tag{11.12}$$

则梯度值为：

$$F_q(x, y) = \sqrt{\frac{1}{2}\left[(g_{xx} + g_{xy}) + (g_{xx} - g_{yy})\cos 2q + 2g_{xy}\sin 2q\right]} \tag{11.13}$$

由于反正切方程在[0，π]范围内提供两个相隔 90° 的值，因此需要应用上式计算两次梯度后，取较大者为最终的梯度图像。计算偏导时可以应用 Sobel 算子。

2．向量法Ⅱ

对于灰度图像，大多数边缘检测方法是基于局部梯度信息的。而对于彩色图像，就要应用距离测度（一般是指欧氏距离）来定义彩色梯度。设向量 v_1 和 v_2 代表彩色图像中的两个像素点，则这两点之间的欧氏距离 $D_E(v_1, v_2)$定义为：

$$D_E(v_1, v_2) = \sqrt{(v_{1,R} - v_{2,R})^2 + (v_{1,G} - v_{2,G})^2 + (v_{1,B} - v_{2,B})^2} \tag{11.14}$$

其中，下标 R、G、B 表示 RGB 彩色空间中的 3 个分量。

有了上述定义后，就可以应用第 8 章介绍的灰度图像的边缘检测方法对彩色图像进行处理。在这里，仍然以最常用的 Sobel 方法为例，它在彩色图像的定义为：

$$E_R = \max(D_E(v_a, v_b), D_E(v_c, v_d)) \tag{11.15}$$

其中向量 v_a、v_b、v_c、v_d 表示图像中 3 个像素的组合：

$$v_a = [v(x-1, y-1), 2 \times v(x-1, y), v(x-1, y+1)]$$

$$v_b = [v(x+1, y-1), 2 \times v(x+1, y), v(x+1, y+1)]$$

$$v_c = [v(x-1, y-1), 2 \times v(x, y-1), v(x+1, y-1)]$$

$$v_d = [v(x-1, y+1), 2 \times v(x, y+1), v(x+1, y+1)]$$

3. RGB 分量直接梯度法

该方法是分别对 R、G、B 3 个彩色分量进行灰度边缘检测，然后再组合成一幅完整的边缘图像。

11.6.2 彩色图像边缘检测编程实例

该实例对上述 3 种方法进行编程实现，通过人为选择阈值的方法生成二值边缘图像，也可以不选择阈值，直接生成灰度边缘图像。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 11.13 所示。

表 11.13　　所修改的属性

控　件	属　性	所修改内容
Button1	Name	edgeC
	Text	边缘检测
	Location	37，426

（2）创建 1 个名为 edgeColor 的 Windows 窗体，该窗体用于选择边缘检测方法和二值化阈值。在该窗体内添加 2 个 Button 控件、2 个 RadioButton 控件、1 个 Label 控件和 1 个 NumericUpDown 控件，其属性修改如表 11.14 所示。

表 11.14　　所修改的属性

控　件	属　性	所修改内容
edgeColor	Text	彩色图像边缘检测
	Size	317，262
	ControlBox	False
button1	Name	start
	Text	确定
	Location	38，168
button2	Name	close

	Text	退出

续表

控　　件	属　　性	所修改内容
button2	Location	190，168
radioButton1	Name	vectEdge1
	Text	向量法 I
	Location	38，26
	Checked	True
radioButton2	Name	vectEdge2
	Text	矢量梯度法
	Location	38，72
radioButton3	Name	rgbEdge
	Text	RGB 分量直接梯度法
	Location	38，118
numericUpDown1	Name	threshUD
	Location	217，72
	Size	49，21
	Maximum	255
	Value	75
label1	Text	阈值：
	Location	170，74

为 2 个 Button 控件添加 Click 事件，并添加 2 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    if (vectEdge1.Checked == true)
          methodF = 0;
    else if (vectEdge2.Checked == true)
          methodF = 1;
    else
          methodF = 2;
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public byte GetMethod
{
    get
    {
          // 得到边缘检测方法
          return methodF;
    }
}
```

```
public int GetThresholding
{
    get
    {
          // 得到阈值
          return (int)threshUD.Value;
    }
}
```

其中 methodF 为 byte 型变量，它在构造函数内被初始化，代码如下：

```
methodF = 0;
```

（3）回到主窗体，为“边缘检测”按钮添加 Click 事件，代码如下：

```
private void edgeC_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化edgeColor
         edgeColor edgedetC = new edgeColor();

         if (edgedetC.ShowDialog() == DialogResult.OK)
         {
              Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height);
              System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                  System.Drawing.Imaging.ImageLockMode.ReadWrite,
                  curBitmap.PixelFormat);
              IntPtr ptr = bmpData.Scan0;
              int bytes = curBitmap.Width * curBitmap.Height;
              byte[] rgbValues = new byte[bytes * 3];
              System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes * 3)

              // 得到何种方法
              byte method = edgedetC.GetMethod;
              // 得到阈值
              int thresh = edgedetC.GetThresholding;

              byte[] grayValues = new byte[bytes];
              double[] tempArray = new double[bytes];
              byte[] rValues = new byte[bytes];
              byte[] gValues = new byte[bytes];
              byte[] bValues = new byte[bytes];
              // 得到R、G、B三个分量的灰度值
              for (int i = 0; i < bytes; i++)
              {
                   rValues[i] = rgbValues[i * 3 + 2];
                   gValues[i] = rgbValues[i * 3 + 1];
                   bValues[i] = rgbValues[i * 3];
              }
              double maxV = 0.0;
              double minV = 1000.0;

              switch(method)
              {
                   case 0:
```

```
                    // 向量法 I

                    double grh, ggh, gbh;
                    double grv, ggv, gbv;
                    double gxx, gyy, gxy;
                    double ge1, ge2;
                    double theta;

                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                        for (int j = 0; j < curBitmap.Width; j++)
                        {
                            // 利用Sobel求偏导
grh = rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] +
    2 * rValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
    rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
    rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] -
    2 * rValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
    rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)];
ggh = gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] +
    2 * gValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
    gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
    gValues[((Math.Abs(i - 1)) % curBitmap.Height) *     curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] -
    2 * gValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
    gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)];
gbh = bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] +
    2 * bValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
    bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
    bValues[((Math.Abs(i - 1)) % curBitmap.Height) *    curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] -
    2 * bValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
    bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)];

grv = rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] +
    2 * rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
    rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)] -
    rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((Math.Abs(j - 1)) % curBitmap.Width)] –
    2 * rValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] –
    rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
        ((j + 1) % curBitmap.Width)];
ggv = gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
```

```
            ((Math.Abs(j  - 1)) % curBitmap.Width)] +
        2 * gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
        gValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
            ((j + 1) % curBitmap.Width)] -
        gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] -
        2 * gValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
        gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)];
    gbv = bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j  - 1)) % curBitmap.Width)] +
        2 * bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
        bValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
            ((j + 1) % curBitmap.Width)] -
        bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((Math.Abs(j - 1)) % curBitmap.Width)] -
        2 * bValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
        bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
            ((j + 1) % curBitmap.Width)];

                                // 向量点积
                                gxx = grh * grh + ggh * ggh + gbh * gbh;
                                gyy = grv * grv + ggv * ggv + gbv * gbv;
                                gxy = grh * grv + ggh * ggv + gbh * gbv;

                                // 最大变化率的方向角度
                                theta = Math.Atan(2 * gxy / (gxx - gyy + 0.000000001)) / 2
                                // 梯度
                                ge1 = ((gxx + gyy) + (gxx - gyy) * Math.Cos(2 * theta)
                                    2 * gxy * Math.Sin(2 * theta)) / 2;
                                // 方向角度加90°
                                theta = theta + Math.PI / 2;
                                ge2 = ((gxx + gyy) + (gxx - gyy) * Math.Cos(2 * theta)
                                    2 * gxy * Math.Sin(2 * theta)) / 2;
                                // 最终的梯度值
                                tempArray[i * curBitmap.Width + j] =
                                   Math.Max(Math.Sqrt(ge1), Math.Sqrt(ge2));

                                // 确定灰度范围
                                if (tempArray[i * curBitmap.Width + j] > maxV)
                                      maxV = tempArray[i * curBitmap.Width + j];
                                else if (tempArray[i * curBitmap.Width + j] < minV)
                                     minV = tempArray[i * curBitmap.Width + j];

                            }
                        }

                        // 灰度值拉伸到[0, 255]
                        for (int i = 0; i < bytes; i++)
                        {
                            grayValues[i] =
                                (byte)((tempArray[i] - minV) * 255 / (maxV - minV));
                        }
                        break;
                    case 1:
                        // 向量法Ⅱ
                        double gr, gg, gb;
```

```
                    double de1, de2;

                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                         for (int j = 0; j < curBitmap.Width; j++)
                         {
                              // 结合Sobel法，计算欧氏距离
    gr = rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)] +
         2 * rValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
         rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)] -
         rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)] -
         2 * rValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
         rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)];
    gg = gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)] +
         2 * gValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
         gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)] -
         gValues[((Math.Abs(i - 1)) % curBitmap.Height) *     curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)] -
         2 * gValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
         gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)];
    gb = bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)] +
         2 * bValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
         bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)] -
         bValues[((Math.Abs(i - 1)) % curBitmap.Height) *     curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)] -
         2 * bValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
         bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)];

                              de1 = Math.Sqrt(gr * gr + gg * gg + gb * gb);

    gr = rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)] +
         2 * rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
         rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)] -
         rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)] -
         2 * rValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
         rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((j + 1) % curBitmap.Width)];
    gg = gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)] +
         2 * gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
         gValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
             ((j + 1) % curBitmap.Width)] -
         gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
             ((Math.Abs(j - 1)) % curBitmap.Width)] -
         2 * gValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
```

```
            gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)];
        gb = bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] +
            2 * bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            bValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
                ((j + 1) % curBitmap.Width)] -
            bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * bValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)];

                                de2 = Math.Sqrt(gr * gr + gg * gg + gb * gb);

                                // 最终梯度值
                                tempArray[i * curBitmap.Width + j] = Math.Max(de1, de2);

                                // 确定范围
                                if (tempArray[i * curBitmap.Width + j] > maxV)
                                      maxV = tempArray[i * curBitmap.Width + j];
                                else if (tempArray[i * curBitmap.Width + j] < minV)
                                      minV = tempArray[i * curBitmap.Width + j];
                           }
                      }

                      // 灰度值拉伸到[0, 255]
                      for (int i = 0; i < bytes; i++)
                      {
                           grayValues[i] =
                                (byte)((tempArray[i] - minV) * 255 / (maxV - minV));
                      }
                      break;
                 case 2:
                    // RGB 分量直接梯度法
                    double[] rvg = new double[bytes];
                    double[] gvg = new double[bytes];
                    double[] bvg = new double[bytes];
                    double gh, gv;
                    double[] maxValue = new double[3] { 0.0, 0.0, 0.0 };
                    double[] minValue = new double[3] { 1000.0, 1000.0, 1000.0 };

                    for (int i = 0; i < curBitmap.Height; i++)
                    {
                         for (int j = 0; j < curBitmap.Width; j++)
                         {
                              // 红色分量的Sobel 法
        gh = rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            2 * rValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] -
            rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * rValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
```

```
            rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)];
        gv = rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] +
            2 * rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            rValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] -
            rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * rValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            rValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)];

                                rvg[i * curBitmap.Width + j] = Math.Sqrt(gh * gh + gv * gv)
                                if (rvg[i * curBitmap.Width + j] > maxValue[0])
                                      maxValue[0] = rvg[i * curBitmap.Width + j];
                                else if (rvg[i * curBitmap.Width + j] < minValue[0])
                                     minValue[0] = rvg[i * curBitmap.Width + j];

                                // 绿色分量的Sobel法
        gh = gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            2 * gValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] -
            gValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * gValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
            gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)];
        gv = gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] +
            2 * gValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
            gValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
                ((j + 1) % curBitmap.Width)] -
            gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * gValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
            gValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)];

                                gvg[i * curBitmap.Width + j] = Math.Sqrt(gh * gh + gv * gv)
                                if (gvg[i * curBitmap.Width + j] > maxValue[1])
                                      maxValue[1] = gvg[i * curBitmap.Width + j];
                                else if (gvg[i * curBitmap.Width + j] < minValue[1])
                                     minValue[1] = gvg[i * curBitmap.Width + j];

                                // 蓝色分量的Sobel法
        gh = bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] +
            2 * bValues[i * curBitmap.Width + ((j + 1) % curBitmap.Width)] +
            bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)] -
            bValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] -
            2 * bValues[i * curBitmap.Width + ((Math.Abs(j - 1)) % curBitmap.Width)] -
            bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)];
```

```
        gv = bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] +
             2 * bValues[((Math.Abs(i - 1)) % curBitmap.Height) * curBitmap.Width + j] +
             bValues[((Math.Abs(i - 1)) % curBitmap.Height) *   curBitmap.Width +
                ((j + 1) % curBitmap.Width)] -
             bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((Math.Abs(j - 1)) % curBitmap.Width)] -
             2 * bValues[((i + 1) % curBitmap.Height) * curBitmap.Width + j] -
             bValues[((i + 1) % curBitmap.Height) * curBitmap.Width +
                ((j + 1) % curBitmap.Width)];

                                    bvg[i * curBitmap.Width + j] = Math.Sqrt(gh * gh + gv * gv)
                                    if (bvg[i * curBitmap.Width + j] > maxValue[2])
                                          maxValue[2] = bvg[i * curBitmap.Width + j];
                                    else if (bvg[i * curBitmap.Width + j] < minValue[2])
                                         minValue[2] = rvg[i * curBitmap.Width + j];
                                }
                            }

                            // 灰度值拉伸到[0, 255]
                            for (int i = 0; i < bytes; i++)
                            {
        rgbValues[i * 3 + 2] = (byte)((rvg[i] - minValue[0]) * 255 / (maxValue[0] - minValue[0]));
        rgbValues[i * 3 + 1] = (byte)((gvg[i] - minValue[1]) * 255 / (maxValue[1] - minValue[1]));
        rgbValues[i * 3] = (byte)((bvg[i] - minValue[2]) * 255 / (maxValue[2] - minValue[2]));
        gh = Math.Max((rvg[i] - minValue[0]) * 255 / (maxValue[0] - minValue[0]),
            (gvg[i] - minValue[1]) * 255 / (maxValue[1] - minValue[1]));
        gv = Math.Max(gh, (bvg[i] - minValue[2]) * 255 / (maxValue[2] - minValue[2]));
                               grayValues[i] = (byte)(gv);
                            }
                            break;
                         default:
                            break;
                    }
                    // 阈值处理成二值边缘图像，如果为0，则保留为灰度边缘图像
                    if (thresh != 0)
                    {
                        for (int i = 0; i < bytes; i++)
                        {
                             if (grayValues[i] > thresh)
                                   grayValues[i] = 255;
                             else
                                   grayValues[i] = 0;
                        }
                    }

                    curBitmap = new Bitmap(curBitmap.Width, curBitmap.Height,
                        System.Drawing.Imaging.PixelFormat.Format8bppIndexed);
                    System.Drawing.Imaging.ColorPalette cp = curBitmap.Palette;
                    for (int i = 0; i < 256; i++)
                    {
                          cp.Entries[i] = Color.FromArgb(i, i, i);
                    }
                    curBitmap.Palette = cp;
                    bmpData = curBitmap.LockBits(rect,
                       System.Drawing.Imaging.ImageLockMode.ReadWrite,
                       curBitmap.PixelFormat);
                    ptr = bmpData.Scan0;
```

```
                System.Runtime.InteropServices.Marshal.Copy(grayValues, 0, ptr, bytes);
                curBitmap.UnlockBits(bmpData);

                Invalidate();
            }
        }
    }
```

（4）编译并运行该段程序，仍以图 2.4 为例。打开该图后，单击“边缘检测”按钮，打开彩色图像边缘检测对话框，如图 11.17 所示设置相关参数，单击“确定”按钮，则经过向量法 II 处理并且阈值为 56 的二值边缘图像就显示出来，如图 11.18 所示。

图 11.17　“彩色图像边缘检测”对话框

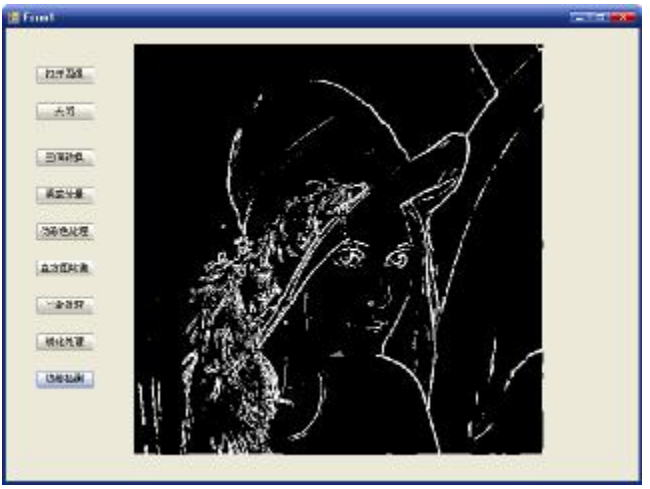


图 11.18　彩色图像边缘检测结果

11.7　彩色图像分割

11.7.1　彩色图像分割原理

彩色图像分割可以看成是灰度图像分割技术在各种颜色空间上的应用。在这里，给出基于 K-均值聚类的彩色图像分割。

彩色图像的 K-均值聚类法与灰度图像的 K-均值聚类法的步骤相似，如下所示。

（1）确定 K 个初始聚类：

$$C_1, C_2, \cdots, C_k \qquad (C_i=[R_i, G_i, B_i] \quad i=1, 2, \cdots, k)$$

（2）根据欧氏距离计算各个像素与聚类之间的距离，并归入距离最小的那个类。

（3）更新聚类：

$$C_{jR}(n+1)=\frac{1}{N_j(n)}\sum_{x\in K_j(n)}x_R$$
$$C_{jG}(n+1)=\frac{1}{N_j(n)}\sum_{x\in K_j(n)}x_G \qquad (11.16)$$
$$C_{jB}(n+1)=\frac{1}{N_j(n)}\sum_{x\in K_j(n)}x_B$$

其中 $N_j(n)$为第 n 次迭代后聚类 K_j 中像素的个数，x_R、x_G、x_B 分别为 R、G、B 分量值。

（4）若所有聚类中的 R、G、B 分量有 $C_i(n+1)=C_i(n)$，则终止迭代；否则回到⑵。

11.7.2 彩色图像分割编程实例

该实例实现了彩色图像的 K-均值聚类分割方法。

（1）在主窗体内添加 1 个 Button 控件，其属性修改如表 11.15 所示。

表 11.15 所修改的属性

控　件	属　性	所修改内容
button1	Name	segC
	Text	图像分割
	Location	37，472

（2）创建 1 个名为 segColor 的 Windows 窗体，该窗体用于选择所要分割的层数。在窗体内添加 2 个 Button 控件、1 个 Label 控件和 1 个 NumericUpDown 控件，其属性修改如表 11.16 所示。

表 11.16 所修改的属性

控　件	属　性	所修改内容
segColor	Text	彩色图像 K-均值聚类分割
	Size	277，177
	ControlBox	False
button1	Name	start
	Text	确定
	Location	41，92
button2	Name	close
	Text	退出
	Location	153，92
numericUpDown1	Name	numClusters
	Location	172，35
	Size	49，21
	Maximum	20
	Minimum	2

	Value	8
label1	Text	K-均值聚类分割数
	Location	37，37，74

为 2 个 Button 控件添加 Click 事件，并添加 1 个 get 属性访问器，代码如下：

```
private void start_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.OK;
}

private void close_Click(object sender, EventArgs e)
{
    this.Close();
}

public byte GetNum
{
    get
    {
         // 得到分割层数
         return (byte)numClusters.Value;
    }
}
```

（3）回到主窗体，为“图像分割”按钮添加 Click 事件，代码如下：

```
private void segC_Click(object sender, EventArgs e)
{
    if (curBitmap != null)
    {
         // 实例化segColor
         segColor segmentationC = new segColor();

         if (segmentationC.ShowDialog() == DialogResult.OK)
         {
               Rectangle rect = new Rectangle(0, 0, curBitmap.Width, curBitmap.Height)
               System.Drawing.Imaging.BitmapData bmpData = curBitmap.LockBits(rect,
                   System.Drawing.Imaging.ImageLockMode.ReadWrite,
                   curBitmap.PixelFormat);
               IntPtr ptr = bmpData.Scan0;
               int bytes = curBitmap.Width * curBitmap.Height;
               byte[] rgbValues = new byte[bytes * 3];
               System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes * 3)

               // 得到所要分割的层数
               byte numbers = segmentationC.GetNum;

               int[] kNum = new int[numbers];
               int[] kAver = new int[numbers * 3];
               int[] kOldAver = new int[numbers * 3];
               int[] kSum = new int[numbers * 3];
               double[] kTemp = new double[numbers];
               byte[] segmentMap = new byte[bytes * 3];

               // 初始化聚类均值
               for (int i = 0; i < numbers; i++)
               {
    kAver[i * 3 + 2] = kOldAver[i * 3 + 2] = Convert.ToInt16(i * 255 / (numbers - 1));
```

```
        kAver[i * 3 + 1] = kOldAver[i * 3 + 1] = Convert.ToInt16(i * 255 / (numbers - 1));
        kAver[i * 3] = kOldAver[i * 3] = Convert.ToInt16(i * 255 / (numbers - 1));
                }

                int count = 0;
                while (true)
                {
                    int order = 0;
                    for (int i = 0; i < numbers; i++)
                    {
                         kNum[i] = 0;
                         kSum[i * 3 + 2] = kSum[i * 3 + 1] = kSum[i * 3] = 0;
                         kAver[i * 3 + 2] = kOldAver[i * 3 + 2];
                         kAver[i * 3 + 1] = kOldAver[i * 3 + 1];
                         kAver[i * 3] = kOldAver[i * 3];
                    }

                    // 归属聚类
                    for (int i = 0; i < bytes; i++)
                    {
                         for (int j = 0; j < numbers; j++)
                         {
                              // 计算欧氏距离
                              kTemp[j] = Math.Pow(rgbValues[i * 3 + 2] - kAver[j * 3 + 2], 2) +
                                 Math.Pow(rgbValues[i * 3 + 1] - kAver[j * 3 + 1], 2) +
                                 Math.Pow(rgbValues[i * 3] - kAver[j * 3], 2);
                         }
                         double temp = 1000000;

                         for (int j = 0; j < numbers; j++)
                         {
                              if (kTemp[j] < temp)
                              {
                                   temp = kTemp[j];
                                   order = j;
                              }
                         }
                         kNum[order]++;
                         kSum[order * 3 + 2] += rgbValues[i * 3 + 2];
                         kSum[order * 3 + 1] += rgbValues[i * 3 + 1];
                         kSum[order * 3] += rgbValues[i * 3];
                         segmentMap[i] = Convert.ToByte(order);
                    }

                    // 更新聚类
                    for (int i = 0; i < numbers; i++)
                    {
                         if (kNum[i] != 0)
                         {
                              kOldAver[i * 3 + 2] = Convert.ToInt16(kSum[i * 3 + 2] / kNum[i]);
                              kOldAver[i * 3 + 1] = Convert.ToInt16(kSum[i * 3 + 1] / kNum[i]);
                              kOldAver[i * 3] = Convert.ToInt16(kSum[i * 3] / kNum[i]);
                         }
                    }

                    // 终止迭代
                    int kkk = 0;
                    count++;
                    for (int i = 0; i < numbers; i++)
                    {
```

```
                        if (kAver[i * 3 + 2] == kOldAver[i * 3 + 2] &&
                                kAver[i * 3 + 1] == kOldAver[i * 3 + 1] &&
                                kAver[i * 3] == kOldAver[i * 3])
                            kkk++;
                    }
                    if (kkk == numbers || count == 100)
                        break;
                }

                // 聚类赋值
                for (int i = 0; i < bytes; i++)
                {
                    for (int j = 0; j < numbers; j++)
                    {
                        if (segmentMap[i] == j)
                        {
                            rgbValues[i * 3 + 2] = Convert.ToByte(kAver[j * 3 + 2]);
                            rgbValues[i * 3 + 1] = Convert.ToByte(kAver[j * 3 + 1]);
                            rgbValues[i * 3] = Convert.ToByte(kAver[j * 3]);
                        }
                    }
                }

                System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes * 3);
                curBitmap.UnlockBits(bmpData);

                Invalidate();
            }
        }
    }
```

（4）编译并运行该段程序，仍以图 2.4 为例。打开该图后，单击“图像分割”按钮，打开彩色图像 K-均值聚类分割对话框，如图 11.19 所示设置相关参数，单击“确定”按钮，则原图被分割成 8 类，如图 11.20 所示。

图 11.19　“彩色图像 K-均值聚类分割”对话框

图 11.20　彩色图像分割结果

11.8 小结

本章对彩色图像处理中最常见的几种算法作了简单的介绍。随着计算机技术的不断发展，彩色图像处理受到越来越多的重视，它不仅仅是灰度图像处理的延伸，它还有自己一套算法理论，这需要人们进一步地开发利用。

参 考 文 献

1.（美）罗宾逊（Robinson，S.），（美）内格尔（Nagel，C.）著；李敏波译．C#高级编程（第 3 版）．北京：清华大学出版社，2005.6

2.（美）塞尔斯（Sells，C.）著；荣耀，蒋贤哲译．Windows Forms 程序设计．北京：人民邮电出版社，2004.9

3.（美）昌德（Chand，M.）著；韩江等译．GDI+图形程序设计．北京：电子工业出版社，2005.3

4．周长发著．C#数值计算算法编程．北京：电子工业出版社，2007.1

5.（美）博维克（Bovik，A.）著．图像与视频处理手册，上册（第二版）（英文版）．北京：电子工业出版社，2006.3

6.（美）博维克（Bovik，A.）著．图像与视频处理手册，下册（第二版）（英文版）．北京：电子工业出版社，2006.3

7.（美）卡斯尔曼（Castleman，K.R.）著；朱志刚等译．数字图像处理．北京：电子工业出版社，2002.2

8.（美）冈萨雷斯（Gonzalez，R.C.）等著．数字图像处理（第二版）（英文版）．北京：电子工业出版社，2002.7

9．余松煜，周源华，张瑞著．数字图像处理．上海：上海交通大学出版社，2007.2

10．陈书海，傅录祥著．实用数字图像处理．北京：科学出版社，2005.6

11．夏良正，李久贤著．数字图像处理．南京：东南大学出版社，2005.8

12．姚敏等著．数字图象处理．北京：机械工业出版社，2006.1

13．章毓晋著．图象分割．北京：科学出版社，2001.2

14．崔屹著．图象处理与分析——数学形态学方法及应用．北京：科学出版社，2000.4

15.（美）冈萨雷斯（Gonzalez，R.C.）等著；阮秋琦等译．数字图像处理（MATLAB 版）。北京：电子工业出版社，2005.9

16．余成波著．数字图像处理及 MATLAB 实现．重庆：重庆大学出版社，2003.6

17．钟志光，卢君，刘伟荣著．Visual C++.NET 数字图像处理实例与解析．北京：清华大学出版社，2003.6

18．何斌著．Visual C++数字图像处理．北京：人民邮电出版社，2001.4

19．陈兵旗，孙明著．Visual C++实用图像处理专业教程．北京：清华大学出版社，2004.

20．靳济芳著．Visual C++小波变换技术与工程实践．北京：人民邮电出版社，2004.1