数字图像处理——Java 语言实现

王　宏　赵海滨　编著

东北大学出版社

·沈　阳·

图书在版编目（CIP）数据

数字图像处理——Java 语言实现 ／王宏，赵海滨编著 .— 沈阳 ：东北大学出版社，2005.4
ISBN 7-81102-139-0

Ⅰ. 数… Ⅱ. ①王… ②赵… Ⅲ. ①数字图像处理 ②Java 语言—程序设计
Ⅳ. ①TN919.8 ②TP312

中国版本图书馆 CIP 数据核字（2005）第 043383 号

出 版 者：东北大学出版社
地址：沈阳市和平区文化路 3 号巷 11 号
邮编：110004
电话：024—83687331（市场部）　83680267（社务室）
传真：024—83680180（市场部）　83680265（社务室）
E-mail：neuph @ neupress.com
http：//www.neupress.com
印 刷 者：铁岭新华印刷有限公司
发 行 者：东北大学出版社
幅面尺寸：184mm × 260mm
印　　张：13
字　　数：324 千字
出版时间：2005 年 4 月第 1 版
印刷时间：2005 年 4 月第 1 次印刷
责任编辑：李毓兴
特邀编辑：李　艳
封面设计：唐敏智
责任校对：尔　顺
责任出版：秦　力

定　　价：28.00 元　（附赠一张光盘）

前　言

数字图像处理是随着现代计算机技术和超大规模集成电路技术的发展而产生、发展和不断成熟起来的一个新技术领域。目前，数字图像处理技术已经应用到了生物医学、遥感图像、军事图像和工业图像处理等领域。特别是近年来随着 Internet 和通讯技术的快速发展，数字图像已经广泛地进入了人们的日常生活。人们可以在 PC 机上浏览和处理数字图像，可以通过网络传输、下载、上载和处理数字图像等。数字图像处理技术引起了人们的广泛关注，并在许多领域中起到了很大的作用。据统计，人的视觉信息占人类获取信息的 80% 左右，所以图像信息无论是在社会进步，还是在人们的日常生活中，都占有极其重要的地位。

本书作为数字图像处理技术的入门书，由浅入深、详细地介绍了数字图像处理的基本原理和方法，并给出了具体而生动的例子。本书共分 8 章。第 1 章介绍了计算机数字图像处理的发展历史和它包含的主要内容；第 2 章介绍了计算机数字图像处理中所使用的数学变换；第 3 章介绍了将模拟图像转换成数字图像的基本原理；第 4 章介绍了图像增强；第 5 章介绍了图像复原；第 6 章介绍了图像的编码与压缩；第 7 章介绍了图像的分割与描述；第 8 章讲述了数字图像处理在医学中的应用。

本书中的 Java 程序由王宏和赵海滨共同完成，附录由赵海滨辑录，其余内容均由王宏编著。在编著本书的过程中得到了庞小飞、王志宇、叶柠、汤龙海、袁正华的帮助，本书的出版得到了国家自然科学基金的资助，在此表示深深的感谢。

由于编著者水平所限，书中如有不妥之处，恳切欢迎读者批评指正。

编著者

2005 年 4 月

目　　录

第 1 章　引　　言

1.1　数字图像处理的发展概况

数字图像处理就是用计算机对图像进行分析和处理，它是一门跨学科的技术。数字图像处理始于 20 世纪 50 年代。特别是在 1964 年，美国喷射推进实验室使用计算机对太空船送回地面的大批月球照片进行处理后，得到了清晰逼真的图像，使这门技术受到了广泛的关注，它成为这门技术发展的重要里程碑，此后数字图像处理技术在空间研究方面得到了广泛的应用。20 世纪 70 年代初，由于大量的研究和应用，数字图像处理已具备了自己的技术特色，并形成了较完善的学科体系，从而成为一门独立的新学科。目前，数字图像处理在生物医学、通信、流通领域，产业界、文件处理领域，军事、公安、遥感、宇宙探险及日常生活中被广泛应用，已经成为当代不可缺少的一门技术。

⑴ 生物医学。生物医学数字图像处理技术大约是 20 世纪 80 年代初在生物医学上得到广泛应用的，随着现代医学特别是数字化医疗技术的不断发展，数字图像处理技术显得更为重要。如 X 光对人体组织有损害，在临床上为了减少这种生物副效应，同时又能得到比较理想的病人的 X 光片，可以用强度较低的 X 光对病人进行照相，然后通过图像处理技术得到清晰的图像，这就是 X 光图像的处理。此外，数字图像处理技术还应用到对超声图像的处理 (图 1-1)、激光显微图像的处理、CT 图像的处理 (图 1-2)、磁共振图像的处理、PET 图像的处理等。目前，数字图像处理技术在现代医学中不仅用于图像的加工和处理，同时还用于信息的存储和传输。

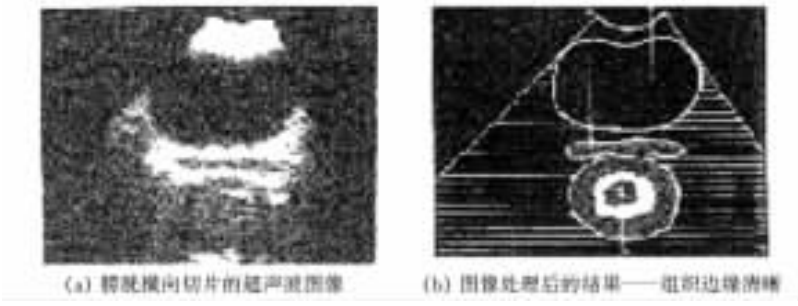


图 1-1　超声波图像的处理 (选自 T. Lehmann et al.)

⑵ 通信。在多媒体网络通讯中，对电视和电话等传输的图像进行数据压缩和处理等。

⑶ 流通领域、产业界、文件处理领域。数字图像处理技术在文件处理、机器人视觉、地质、海洋、气象、农业、灾害治理、货物检测、邮政编码、金融、银行、工矿企业、冶金、渔业、机械、交通、电子商务等领域被广泛应用。

⑷ 军事和公安。对现场照片、指纹和手迹等图像进行分析和处理。

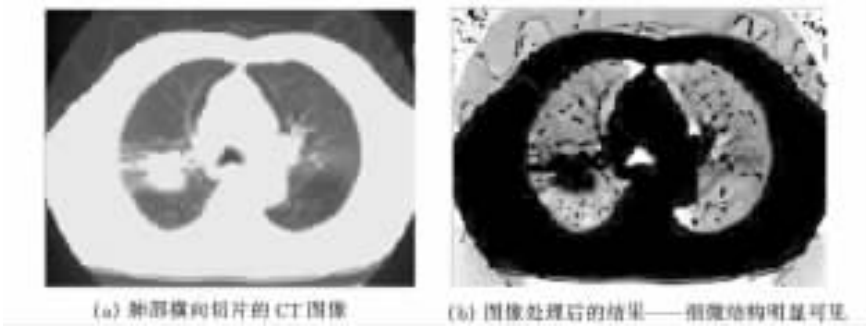


图 1-2 CT 图像的处理

⑸ 遥感。遥感是用不同光源和技术获得大量的遥感图像，这些图像需要用数字图像处理技术加工处理并提取有用的信息。

1.2 数字图像和数字图像处理

1.2.1 图 像

1. 景象

众所周知，人类的视觉系统能够看到五彩缤纷的世界，这是人类的生理功能。具体地说，它是由于光线照射到物体上，经过不发光物体形成的反射光线或发光物体发出的光线射入人们的眼睛内，这些光信号在视网膜上转化成神经电信号，然后神经电信号通过神经纤维传送到大脑皮层中，使人们看到外部的世界，这时所看到的外部世界称为景象。

2. 图像

直到人类文明发展到一定程度时，人们才意识到景象的存在，并想出一些方法将景象记录下来，这种记录下来的景象称为图像。所以说图像是对视觉信息的记录和展现。一幅照片就是一幅图像，如图 1-1 和图 1-2 所示。

要形成一幅图像，必须有两个因素：一个是景象，另一个是电磁波。电磁波谱包括可见光、X 射线和微波等（如图 1-3 所示）。电磁波以 3×10^8km/s 的速度传播，所以事件的成像是在瞬间进行的。电磁波辐射决定了一幅图像的亮度。电磁波与物体的相互作用决定了图像中呈现出的物体的几何形状。由于电磁波的波长不同，电磁波与物质的相互作用形式也不同。在可见光（波长在 400～700nm）区内呈现彩色图像，其中 550nm 波长呈现绿色，700nm 波长呈现红色。在短波区，电磁波具有很大的能量。如在 X 射线区电磁波具有很强的穿透能

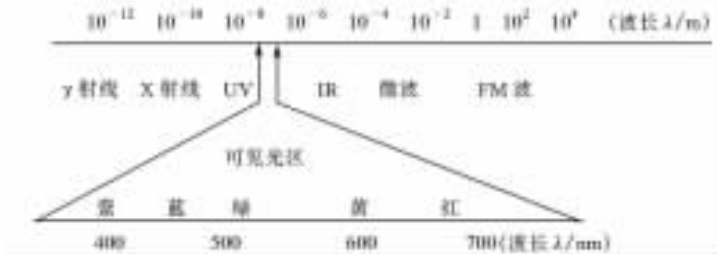


图 1-3 电磁波分布示意图

力，所以 X 射线成像常用于观测物体的内部结构 (如图 1-4 所示)。在更短的波长区域 (如 γ 射线区) 电磁波具有更大的能量，所以具有更强的穿透能力，γ 射线成像在医学上也广泛应用着。X 射线可以提供物体内部的结构信息，而 γ 射线成像可以提供内部的生理功能信息。在长电磁波区域，如红外线成像和微波 (雷达) 成像也被广泛地应用。关于图像的例子很多，这里就不一一列举了。

图 1-4　1896 年 1 月第一张 X 光片 (选自 H. Breuer)

根据电磁波长的不同，常采用不同的记录方式 (如照相机、摄像机、X 光成像等) 来形成各种不同的图像。通常将图像分为两种，一种是可以为视觉系统直接感受到的图像，如照片和图画等，称为可见图像；另一种是不可以为视觉系统直接感受到的图像，这类图像必须经过某些数学和物理的转换才能为视觉系统所感受，如 X 光片等，称为不可见图像。从另一个角度，若记录的图像是随时间变化的，则称为时变图像，如电影；若记录的图像是不随时间变化的，则称为静止图像，如图画。

通常用数学函数 $Q(x, y, z, \lambda, t)$ 表示一幅图像。对于平面图像，只有变量 x 和 y，可用数学函数 $f(x, y, \lambda, t)$ 表示。

若只考虑光的能量，而不考虑波长，则图像只有黑白深浅之分，这样的图像称为黑白图像。对于平面黑白图像的数学表达式为

$$f(x, y, t) = \int Q(x, y, \lambda, t)\, v(\lambda)\, \mathrm{d}\lambda \tag{1-1}$$

式中　$v(\lambda)$ 为 λ 的品质函数。

这样，$f(x, y, t)$ 就表示了一幅随时间变化的平面黑白图像，即动态平面黑白图像。若图像函数与时间无关，则代表静止图像，对于静止的平面黑白图像可表示为

$$f(x, y) = \int f(x, y, t)\, \mathrm{d}t \tag{1-2}$$

图像函数 $f(x, y)$ 在某一点的值称为灰度，它与图像在该点的亮度是相对应的，即

灰度值大——→亮

灰度值小——→暗

灰度是一个非负的连续函数，即

$$0 \leqslant f(x, y) \leqslant B_{\mathrm{m}} \tag{1-3}$$

式中　B_{m}——图像中灰度的最大值。

1.2.2　数字图像

上面提到的图像叫模拟图像或连续图像，它们是不能直接送到计算机中进行处理的，为了在计算机中对图像进行处理，需要将现有的连续图像转换成计算机能识别的形式——数字图像。所谓数字图像，就是将连续图像进行数字化后，用一个矩阵表示的图像。图 1-5 示出了一幅连续图像的数字化过程，图 1-5 (b) 是该图像的矩阵表达。

这时从广义上来说，静止、平面、黑白的数字图像就可用下面的数学形式表达

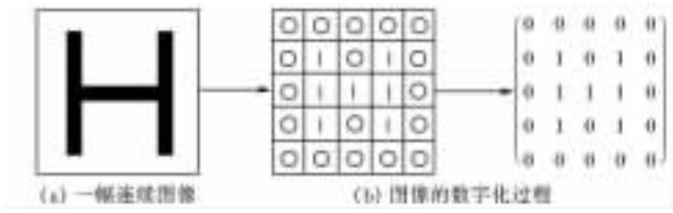


图 1-5 连续图像的数字化过程

$$f(x,y)=\begin{pmatrix} f(0,0) & f(1,0) & \cdots & f(M-1,0) \\ f(0,1) & f(1,1) & \cdots & f(M-1,1) \\ \cdots & \cdots & & \cdots \\ f(0,N-1) & f(1,N-1) & \cdots & f(M-1,N-1) \end{pmatrix} \tag{1-4}$$

式中 $x=0,1,2,\cdots,M-1$；

$y=0,1,2,\cdots,N-1$；

$f(x,y)$——点(x,y)处的灰度值。

这样就将一幅图像表示成为矩阵的形式——数字图像。

1.2.3 数字图像处理

一幅图像用矩阵表示后就可用计算机对数字图像的矩阵进行各种运算了，这就是数字图像处理。图像处理的目的是改善图像的质量，使它更便于人们观察，适合机器识别。

通常分为两大类图像处理方式：

(1) 光学图像处理，它是用光学的方法对图像作某些特殊的处理。

(2) 数字图像处理，它是用计算机对图像进行加工和处理，主要有以下几个方面：

1) 对图像进行增强处理。这里是增强有用的信息，抑制无用的信息，从而改变图像的灰度分布，使图像更易于人们视觉系统观看。在图 1-6 (a) 中图像比较暗，经过增强处理后，得到了图 1-6 (b)，这时图像更清晰了。

图 1-6 图像增强的例子

2) 对图像进行复原处理。有些图像由于在拍摄的曝光时间内，景物与照相机之间产生了相对移动，使图像模糊了 (如图 1-7 (a) 所示)，应用图像复原技术可以改善这种图像的质量 (如图 1-7 (b) 所示)。

图 1-7　图像复原的例子

3) 提取、分析和描述图像中所包含的特殊信息。在图 1-8 (a) 中将图像中的边缘提取出来，得到了图 1-8 (b)。

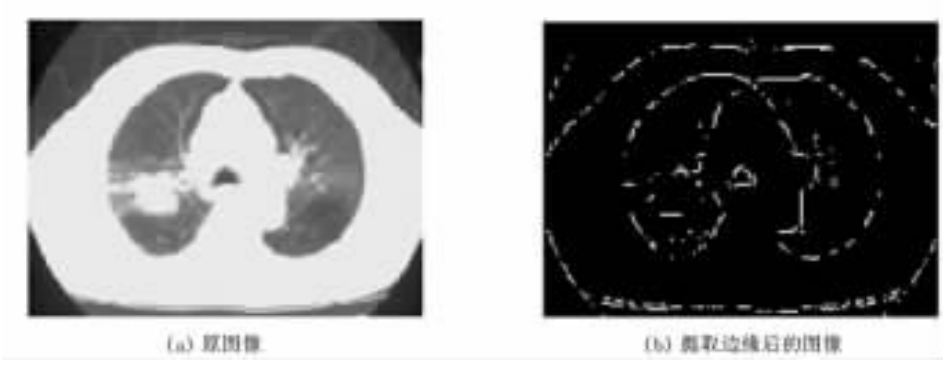


图 1-8　图像边缘提取的例子

1.3　数字图像处理系统

能够完成图像处理和分析任务的系统就是数字图像处理系统。数字图像处理系统主要有：图像输入设备，执行分析与处理图像的计算机及图像处理机；输出设备及存储系统中的图像数据库等。计算机数字图像处理系统的结构框图如图 1-9 所示。

数字图像处理系统与其他数据处理系统的不同之处是其庞大的数据处理量和存储量。因此，无论从硬件的配置还是软件的环境上，数字图像处理系统都有别于其他的计算机系统，从而形成了专门的数字图像处理系统。可以说，图像处理技术是以计算机为核心的技术，因此，图像处理系统的发展是随着计算机技术的提高而发展的。从系统的层次上，数字图像处理系统可分为高、中、低三个档次。

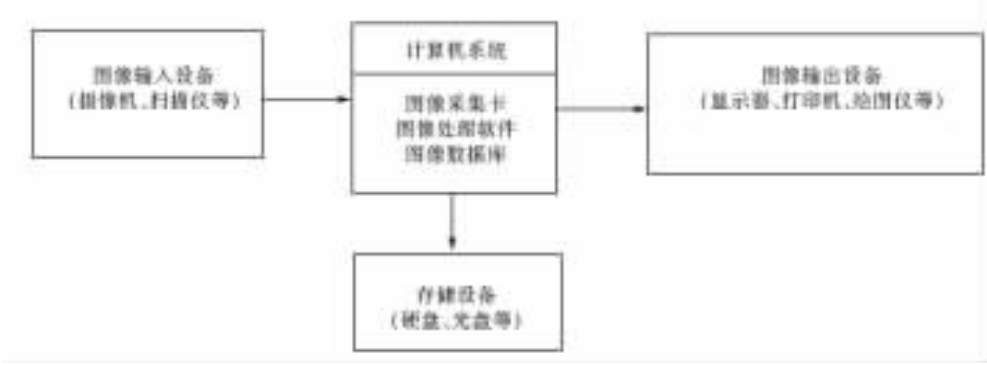


图 1-9 数字图像处理系统示意图

⑴ 高档图像处理系统。它是采用高速芯片、适合图像处理特有规律的并行阵列图像处理机。这种系统采用多个 CPU 或多机结构，可以以并行或流水线的方式工作。

⑵ 中档图像处理工作站。它是由主控计算机和图像处理器构成。其主控机是小型机或工作站。这类系统有较强的交互处理能力，在系统环境下具有较好的再开发能力。

⑶ 低档的数字图像处理系统。由计算机加上图像采集部件构成，其结构简单，是一种便于普及和推广的图像处理系统，它是本书重点介绍的系统。

此外，从图像传感器的敏感区域看，又可分为可见光、红外线、近红外线、X 射线、雷达、γ 射线、超声波等图像处理系统；从采集部件与景物的距离上来说，还可分成遥感、宏观和微观图像处理系统；就应用场所而言，又能分成通用图像处理系统和专用图像处理系统。通用图像处理系统一般用于研究开发；而专用图像处理系统一般用于特殊用途，它是在通用系统研究基础上研制开发的为实现某一个或几个功能的商用系统。

1.3.1 数字图像处理硬件系统

图像处理硬件系统主要由图像采集系统、图像处理系统和图像输出系统三部分组成。

1. 数字图像采集系统

原始的图像数据是通过图像采集部件进入计算机的，即图像采集部件是采集原始的模拟图像数据，并将其转换成数字信号，计算机接受到图像的数字信号后，将其存入内存储区。

数字图像处理系统常用的图像采集部件有：

⑴ 摄像机和视频图像采集卡。该图像处理系统采用的摄像机需要和视频图像采集卡配合使用，使用时要考虑两者参数的优化问题。视频图像采集卡可以将摄像机摄取的模拟图像转化为数字图像。

⑵ 图像扫描仪。它是近年来出现的数字化产品，将图像采集和数字化部件集成在同一个机器上，使其输出的信号能直接为计算机所接受。

⑶ 数码摄像机。数码摄像机也是近几年来出现的数字化产品，将图像采集和数字化部件集成在同一机器上，使其输出的信号能直接为计算机所接受。数码摄像机使图像的采集部件和主机的连接更具有通配性，而且由于其携带方便，有相应的存储器，因此更适用于现场数据采集。

2. 数字图像处理系统

在数字图像处理系统中，图像处理工作是由计算机完成的，计算机的扩展槽上插有带帧

存储器的采集卡，图像处理的过程通常包含从帧存储器提取数据到计算机内存、处理内存中的图像数据和送数据回图像帧存储器三个步骤。对于直接使用内存的采集卡，则只需和内存进行数据交换，计算机的内存越大，CPU 的运算速度越快，图像处理的速度也就越快。

3. 数字图像输出系统

图像的输出是图像处理的最终目的。图像输出有两种基本形式：一种是根据图像处理的结果作出判断，例如质量检测中的合格与不合格，输出不一定以图像作为最终形式，而只需作出提示供人或机器进行选择。这种提示可以是计算机屏幕信息，或是电平信号的高低，这样的输出往往用于成熟研究的应用上。另一种则是以图像为输出形式，它包括中间过程的监视以及结果图像的输出。图像输出方式有屏幕输出、打印输出和视频硬拷贝输出。

⑴ 屏幕输出。用屏幕输出处理结果是最直观、最简单的方法，并可获得高质量图像。根据硬件的不同，可分为单屏显示和分屏显示两种形式。单屏显示是指图像处理的过程与结果都在同一计算机的显示器上显示，一屏两用，比较经济。分屏显示是指图像处理的结果或中间过程由专门的监视器显示，加上计算机本身的显示器，这样的系统可以称作双屏系统。由于图像部分和程序执行过程互不干扰，因此处理过程比较直观。

⑵ 打印输出。打印输出的设备为打印机，按打印效果分成黑白、彩色两种，有点阵式、喷墨式、激光式、热敏式打印机等。

⑶ 视频硬拷贝输出。视频硬拷贝输出采用专用的拷贝和复制纸，得到高质量输出图像。视频拷贝机分成模拟式和数字式两种。模拟式拷贝机需连接视频信号，数字式拷贝机则可以直接和计算机相连，视频硬拷贝输出形式能长时间地保存图像。

1.3.2 数字图像处理软件系统——Java 语言

进行处理图像的应用软件很多，如 Photoshop，Fireworks 等。但这些图像处理软件只是为了改善用户的某些视觉效果，不利于针对某个具体任务进行特殊的图像处理。在实际应用中进行的图像处理应根据高级程序语言来编写自己的图像处理软件。Java 就是一种新型的可用于图像处理的高级语言。

众所周知，随着科技的发展，数字图像和网络都进入了人们的日常生活中。人们可以在计算机上浏览和处理数字图像，也可以通过网络传输、下载、上载和处理数字图像，这样就需要能将网络与数字图像联系起来的高级语言。Java 就是近年来很流行的一种网络编程语言，它提供了对图像的支持。在 Java 出现以前，Internet 上的信息内容都是一些乏味死板的 HTML 文档。这对于那些迷恋于 WEB 浏览的人们来说简直不可容忍。他们迫切希望能在 WEB 中看到一些交互式的内容，开发人员也极希望能够在 WEB 上创建一类无需考虑软硬件平台就可以执行的应用程序，当然这些程序还要有极大的安全保障。对于用户的这种要求，传统的编程语言显得无能为力。1995 年，当 SUN 公司正式以 Java 这个名字推出一套新的语言的时候，几乎所有的 WEB 开发人员都想到：噢，这正是我想要的。于是，全世界的目光都被这种神奇的语言所吸引了。

Java 是一种简单的、面向对象的、分布式的、解释的、安全的、可移植的、性能很优异的、多线程的、动态的语言。Java 能运行于不同的平台，使用 Java 编写的程序能在世界范围内共享。Java 对程序提供了安全管理器，防止程序的非法访问，避免病毒通过指针侵入系统。Java 建立在扩展 TCP/IP 网络平台上，其类库提供了用 HTTP 和 FTP 等协议传送和接受信息的方法。这就使得程序员使用网络上的文件和使用本机文件一样容易。

我们知道，早先的 WWW 仅可以传送文本和图片，Java 的出现实现了互动的页面，这是一次伟大的革命。Java 并不只是为 Internet 的 WWW 而设计的，它也可以用来编写独立的应用程序。Java 程序和它的浏览器提供了程序在浏览器中运行的方法。比如，浏览器可直接播放声音，播放动画，提供友好的交互式界面。

Java 图像处理常用的图像格式是 GIF 和 JPEG。GIF 图像文件格式是为了方便用户在网络上传输数据而制定的。JPEG 图像文件格式可用不同的比例对文件格式进行压缩，它采用最少的磁盘空间，而得到较好的图像质量。

Java 程序分为两种：Java Application 和 Java Applet。Java Application 可以独立运行，Java Applet 不能独立运行，可以使用 Applet Viewer 或其他支持 Java 虚拟机的浏览器运行。为了使读者对 Java 语言有一个初步的印象，下面将给出对图像进行提取和显示的简单例子。

在图像处理开始时，需要将图像载入，可以使用 Toolkit 类的 getImage () 方法，

Image getIamge (URL url) ——从由 URL 对象指定的因特网地址上的图形文件中创建一个 Image 对象。

Image getIamge (String filename) ——从图形文件中创建一个带有指定名字的 Image 对象。

下面的语句是创建一个名字为 fileimg.jpg 的 Image 对象。

Image img = Toolkit.getDefaultToolkit().getImage("fileimg.jpg")。

然后，若希望将读入的图像显示出来，需要使用 Graphics 类中的 drawImage () 方法。程序清单 1-1 和 1-2 中示出了一个使用Java Applet 读取和显示图像的例子。

程序清单 1-1　　LoadFromApple. Java 的源代码

```
//LoadFromApple.java
import java.awt.*;
import java.awt.Image.*;
import java.applet.Applet;

public class LoadFromApple extends Applet{
Image imgObj;
// init()方法,加载图像 filename.jpg
public void init()
{
imgObj=getImage(getDocumentBase(),"filename.jpg");
}
// paint()方法,显示图像信息
public void paint(Graphics g)
{
g.drawImage(imgObj,0,0,this);
}
}
```

上面的程序若使用 Applet 运行，还需要一个 HTML 文件 (如清单 1-2 所示)。

程序清单 1-2　　LoadFromApple.html **的源代码**

```
<html>
<head>
</head>
<body>
<applet code = LoadFromApple.class width=500 height=400></applet>
</body>
</html>
```

为了观察上述程序的显示效果，可以将 LoadFromApple.class **和** LoadFromApple.html **复制在同一个目录下，以便浏览器对这两个文件进行读取，图** 1-10 **给出了根据程序清单** 1-1 **和程序清单** 1-2 **得出的效果。**

图 1-10　用 Java Applet 读取和显示图像的例子

Java **程序也可以独立运行，这就是** Java Application**。程序清单** 1-3 **中给出了一个使用** Java Application **读取和显示图像的例子，图** 1-11 **示出了该程序运行的结果。**

程序清单 1-3　　LoadFromAppli. Java **的源代码**

```
//LoadFromAppli.java
import java.awt.*;
import java.awt.event.*;

public class LoadFromAppli extends Frame
{
Image im;
//LoadFromAppli 的构造方法,加载图像 filename.jpg
```

```
public LoadFromAppli()
{
im=Toolkit.getDefaultToolkit().getImage("filename.jpg");
//添加窗口监听事件
addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
}
//LoadFromAppli 的 paint()方法,显示图像信息
public void paintGraphics g){
g.drawImage(im,30,30,this);
}
//定义 main 方法,设置窗口的大小,显示窗口
public static void main(String[] args) {
LoadFromAppli f = new LoadFromAppli();
f.setSize(200,200);
f.show();
}
}
```

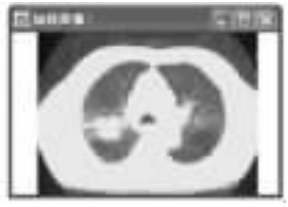


图 1-11 用 Java Application 读取和显示图像的例子

练 习 题

1 **当一个人照全身的 X 光片时，从图像上能看到什么现象？为什么？**

2 **如果你有一本像册，你可以看看所有的照片是否都很清晰，将不清晰的照片通过扫描仪输入到计算机中，试用 Java 语言将其显示在屏幕上，并存入计算机内，以备后续处理。**

第 2 章　数字图像处理的数学基础

在第一章中已经介绍了，所谓数字图像就是指代表图像的矩阵。数字图像处理就是对图像矩阵进行各种数学运算。这就意味着在进行图像处理时需要一些数学基础，主要包括线性系统、傅立叶变换、沃尔什变换和小波变换等。

2.1　图像的线性处理系统

所谓系统就是若干相互作用和相互依赖的事物组合而成的、具有特定功能的整体。在数字图像处理中，为了数学上计算的方便，需要对图像处理系统予以近似，如下所述。

2.1.1　成像系统模型的划分

1. 空间位置连续系统和空间位置离散系统

若系统的输入信号和输出信号都是空间位置的连续信号，则该系统称为空间位置连续系统，这样的系统通常用数学解析方法进行处理。若系统的输入信号和输出信号都是空间位置的离散信号，则该系统称为空间位置离散系统，这样的系统可用向量与矩阵理论进行处理。

2. 线性系统与非线性系统

若输入图像函数为 $f(x, y)$，而 $f(x, y)$ 进行 H 运算或称为作 H 处理，则处理后的图像函数为

$$g(x, y) = Hf(x, y) \tag{2-1}$$

这个运算关系可以用图表示如下

如果有两个图像 $f_1(x, y)$ 和 $f_2(x, y)$ 进入 H 系统中，系统的输入和输出分别为

输入　　输出

$$f_1(x, y) \longrightarrow Hf_1(x, y)$$

$$f_2(x, y) \longrightarrow Hf_2(x, y)$$

当下面条件满足时

$$H(f_1(x, y) + f_2(x, y)) = Hf_1(x, y) + Hf_2(x, y) \tag{2-2}$$

称此系统具有叠加性。

若系统满足

$$H(kf(x, y)) = k(Hf(x, y)) \tag{2-3}$$

时，称此系统具有奇次性，这里为 k 常数。

如果系统有 N 个输入图像，根据式 (2-2) 与式 (2-3)，若有

$$H\sum_{i=1}^{N}k_i f_i(x,y)=\sum_{i=1}^{N}k_i Hf_i(x,y) \tag{2-4}$$

表明这个系统具有叠加性和奇次性，这种系统称为线性系统。而不具有叠加性与奇次性的系统称为非线性系统。

3. 位移不变系统

如果输入图像函数为 $f(x,y)$，根据式 (2-1) 其输出图像函数应为

$$g(x,y)=Hf(x,y) \tag{2-5}$$

当空间位置变化时，即输入图像函数为 $f(x-x_0,y-y_0)$ 时，其输出图像函数为

$$g(x-x_0,y-y_0)=Hf(x-x_0,y-y_0) \tag{2-6}$$

可见成像系统的参数没有随空间位置的变化而发生变化，这种成像系统的参数不随空间位置而发生变化的系统称为位移不变系统。在图像处理中，通常将成像系统描述成为线性、位移不变的系统。

2.1.2 点光源和 δ 函数

1. δ 函数的定义

在图像处理的线性系统分析中，常引入点光源的概念。任一图像 $f(x,y)$ 都可以看作是许多点光源的集合，那么有关点光源的运算规则可以应用在图像函数的运算中。点光源是一种抽象的物理概念，即它的面积趋近于零，亮度趋近于无限大。通过下面的例子可以进一步了解图像函数 $f(x,y)$ 与点光源的关系。

【例 2-1】 假设某一函数定义如下，

$$\delta(x-x_0,y-y_0)=\begin{cases}\dfrac{1}{\varepsilon} & |x-x_0|\leqslant\dfrac{\varepsilon}{2},\ |y-y_0|\leqslant\dfrac{\varepsilon}{2}\\ 0 & \text{其他}\end{cases}$$

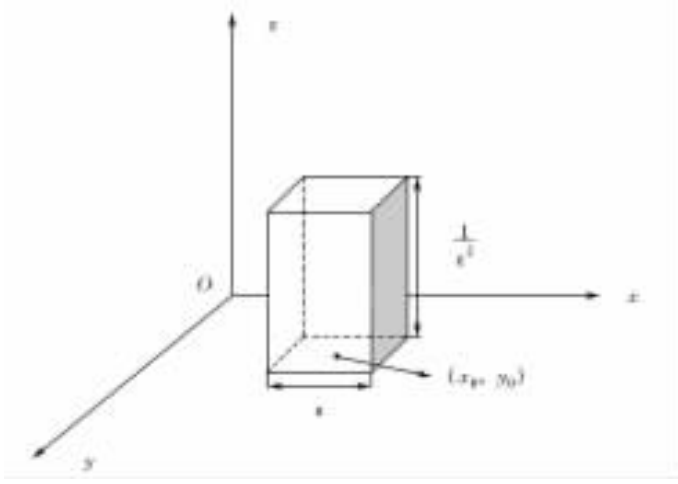


图 2-1 $\delta(x-x_0,y-y_0)$ 函数示意图

从图 2-1 中可见，当 ε 减少时，长方体的底面积减少，高度增加；当 ε 趋近于零时，长方体的底面积趋近于零，高度趋近于无限大，这样一来图 2-1 就变为了图 2-2 所示的形式。

图 2-2 可用下面的数学形式表达

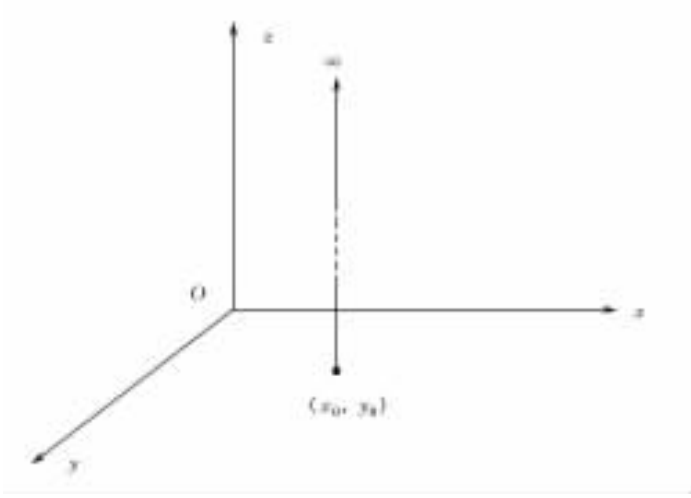


图 2-2　δ 函数示意图

$$\iint_{-\infty}^{+\infty} \delta(x - x_0, y - y_0)\,\mathrm{d}x\mathrm{d}y = 1 \tag{2-7}$$

$$\delta(x - x_0, y - y_0)_{x \neq x_0,\ y \neq y_0} = 0 \tag{2-8}$$

$$\delta(x - x_0, y - y_0)_{x = x_0,\ y = y_0} = \infty \tag{2-9}$$

式 (2-7)、(2-8) 和 (2-9) 所表示的函数称为二维 δ 函数。可见，δ 函数类似于非常短暂的、非常强烈的单位脉冲，常用来表示点光源和点电荷等。这样对于图像 $f(x, y)$ 在 (x_0, y_0) 位置上点光源的亮度可记为 $f(x, y)\,\delta(x - x_0, y - y_0)$，并将图像函数看成是由无数个点光源集合而成的。

2．δ 函数的若干性质

(1) 筛选性质。

若 $F(x, y)$ 在 $x = x_0$，$y = y_0$ 处连续，则有

$$\iint_{-\infty}^{+\infty} f(x, y)\,\delta(x - x_0, y - y_0)\,\mathrm{d}x\mathrm{d}y = f(x_0, y_0) \tag{2-10}$$

式 (2-10) 称为 δ 函数的筛选性质。

(2) δ 函数是偶函数

δ 函数满足偶函数的性质，即

$$\delta(x - x_0, y - y_0) = \delta(x_0 - x, y_0 - y) \tag{2-11}$$

(3) 卷积性质

根据 δ 函数的筛选性质，即式 (2-10)

$$\iint_{-\infty}^{+\infty} f(x, y)\,\delta(x - x_0, y - y_0)\,\mathrm{d}x\mathrm{d}y = f(x_0, y_0)$$

如果假设 $x = \alpha$，$y = \beta$，$x = x_0$，$y = y_0$，则上式变为

$$\iint_{-\infty}^{+\infty} f(\alpha, \beta)\,\delta(\alpha - x, \beta - y)\,\mathrm{d}\alpha\mathrm{d}\beta = f(x, y) \tag{2-12}$$

因为 δ 函数是偶函数，所以有

$$\delta(\alpha - x, \beta - y) = \delta(x - \alpha, y - \beta) \tag{2-13}$$

将式(2-13)代入式(2-12)得

$$\iint_{-\infty}^{+\infty} f(\alpha, \beta)\,\delta(x - \alpha, y - \beta)\,\mathrm{d}\alpha\mathrm{d}\beta = f(x, y) \tag{2-14}$$

根据卷积的定义，式(2-14)是 $f(x, y)$ 与 $\delta(x, y)$ 的卷积，即

$$f(x, y) * \delta(x, y) = f(x, y) \tag{2-15}$$

可见，函数 $f(x, y)$ 与 $\delta(x, y)$ 的卷积再次产生原函数。

(4) 可分离性

δ 函数是可分离的，即

$$\delta(x - x_0, y - y_0) = \delta(x - x_0)\,\delta(y - y_0) \tag{2-16}$$

2.2 傅立叶变换

数字图像处理就是用计算机对图像矩阵进行各种运算，运算既可以在空间域内进行，也可以在频率域内进行。如果图像处理需要在频率域内进行，就需要将图像函数从空间域内变换到频率域内，即进行图像变换，傅立叶变换就是最常用的一种变换。

2.2.1 连续函数的傅立叶变换

1. 一维连续函数的傅立叶变换

令 $f(x)$ 是实变量 x 的连续、可积的函数，则 $f(x)$ 的一维傅立叶变换为

$$F(u) = \int_{-\infty}^{+\infty} f(x)\exp(-\mathrm{j}2\pi ux)\,\mathrm{d}x \tag{2-17}$$

式中 $\mathrm{j} = \sqrt{-1}$。

若 $F(u)$ 已知，可以通过下面的傅立叶逆变换

$$f(x) = \int_{-\infty}^{+\infty} F(u)\exp(\mathrm{j}2\pi ux)\,\mathrm{d}u \tag{2-18}$$

得到 $f(x)$，式(2-17)和式(2-18)称为傅立叶变换对。

因为 $f(x)$ 是实函数，所以它的傅立叶变换可表示为

$$F(u) = R(u) + \mathrm{j}I(u) \tag{2-19}$$

这里 $R(u)$ 和 $I(u)$ 分别是 $F(u)$ 的实部和虚部。式(2-19)可以表示成指数的形式，即

$$F(u) = |F(u)|\exp(\mathrm{j}\Phi(u)) \tag{2-20}$$

$|F(u)|$ 称为 $f(x)$ 的傅立叶谱，它的值为

$$|F(u)| = \sqrt{R^2(u) + I^2(u)} \tag{2-21}$$

并且

$$\Phi(u) = \arctan\left[\frac{I(u)}{R(u)}\right] \tag{2-22}$$

称为相角。傅立叶谱的平方

$$E(u) = |F(u)|^2 \tag{2-23}$$

称为 $f(x)$ 的能量谱。

根据 Euler 公式，傅立叶变换的指数形式 $\exp[-\mathrm{j}2\pi ux]$ 可以表示成

$$\exp(-j2\pi ux) = \cos 2\pi ux - j\sin 2\pi ux \tag{2-24}$$

可见式(2-24)中的 u 与正弦和余弦函数的频率变量相对应，所以，u 称为频率变量。

【例 2-2】　函数 $f(x)$ 如图 2-3(a)所示，它的傅立叶变换为

$$\begin{aligned} F(u) &= \int_{-\infty}^{+\infty} f(x)\exp(-j2\pi ux)\,dx = \int_0^X A\exp(-j2\pi ux)\,dx \\ &= \frac{-A}{j2\pi u}[\exp(-j2\pi ux)]_0^X = \frac{-A}{j2\pi u}[\exp(-j2\pi uX) - 1] \\ &= \frac{A}{j2\pi u}[\exp(j\pi uX) - \exp(-j\pi uX)]\exp(-j\pi uX) = \frac{A}{\pi u}\sin(\pi uX)\exp(-j\pi uX) \end{aligned}$$

它的傅立叶谱为

$$|F(u)| = \frac{A}{\pi u}|\sin(\pi uX)|\,|\exp(-j\pi uX)| = AX\left|\frac{\sin(\pi uX)}{\pi uX}\right|$$

在图 2-3(b)中示出了 $|F(u)|$ 的图形。

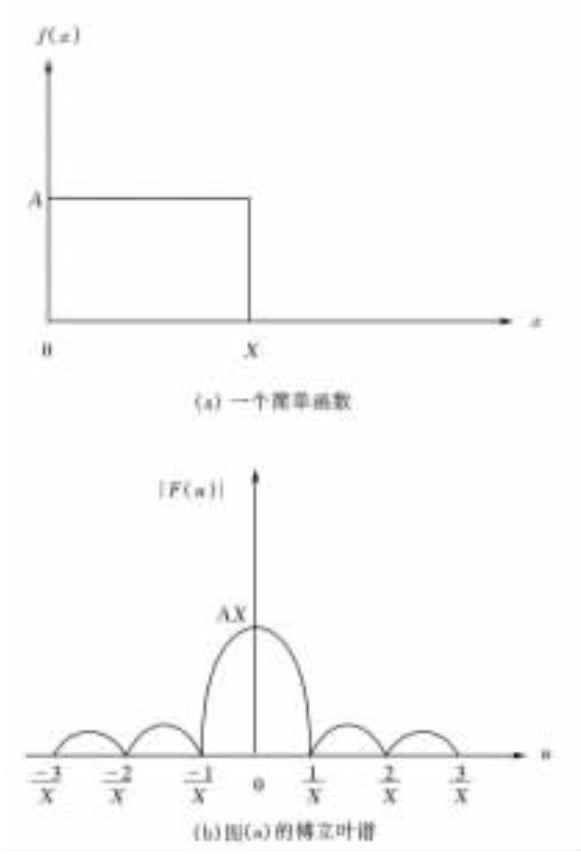


图 2-3　一个简单函数和它的傅立叶谱

2. 二维连续函数的傅立叶变换

如果二维函数 $f(x, y)$ 是连续可积的，则它的傅立叶变换对为

$$F(u, v) = \iint_{-\infty}^{+\infty} f(x, y)\exp(-j2\pi(ux + vy))\,dx\,dy \tag{2-25}$$

并且

$$f(x, y) = \iint_{-\infty}^{+\infty} F(u, v) \exp[j2\pi(ux + vy)] \mathrm{d}u \mathrm{d}v \tag{2-26}$$

式中 u 和 v 是频率变量。

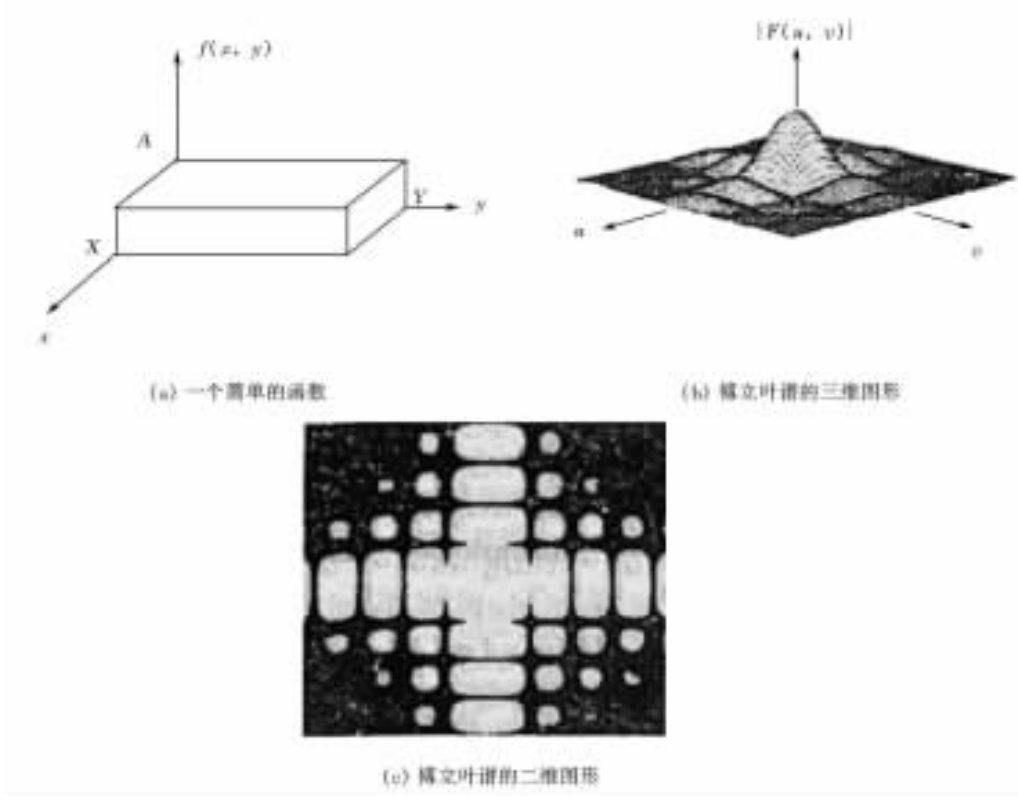
与一维傅立叶变换相类似，二维傅立叶变换的傅立叶谱、相角和能量谱分别为

$$|F(u, v)| = \sqrt{R^2(u, v) + I^2(u, v)} \tag{2-27}$$

$$\Phi(u, v) = \arctan\left[\frac{I(u, v)}{R(u, v)}\right] \tag{2-28}$$

$$E(u, v) = R^2(u, v) + I^2(u, v) \tag{2-29}$$

【例 2-3】 函数 $f(x, y)$ 如图 2-4 (a) 所示，它的傅立叶变换为



(a) 一个简单的函数　　(b) 傅立叶谱的三维图形

(c) 傅立叶谱的二维图形

图 2-4 一个简单的函数和它的傅立叶谱 (选自 C. Rafeal et al.)

$$\begin{aligned}
F(u, v) &= \iint_{-\infty}^{+\infty} f(x, y) \exp(-j2\pi(ux + vy)) \mathrm{d}x \mathrm{d}y \\
&= A\int_0^X \exp(-j2\pi ux) \mathrm{d}x \int_0^Y \exp(-j2\pi vy) \mathrm{d}y \\
&= A\left[\frac{\exp(-j2\pi ux)}{-j2\pi u}\right]_0^X \left[\frac{\exp(-j2\pi vy)}{-j2\pi v}\right]_0^Y \\
&= \frac{A}{-j2\pi u}[\exp(-j2\pi uX) - 1] \frac{-1}{j2\pi v}[\exp(-j2\pi vY) - 1] \\
&= AXY\left[\frac{\sin(\pi uX)\exp(-j\pi uX)}{\pi uX}\right]\left[\frac{\sin(\pi vY)\exp(-j\pi vY)}{\pi vY}\right]
\end{aligned}$$

它的傅立叶谱为

$$|F(u,v)| = AXY\left|\frac{\sin(\pi uX)}{\pi uX}\right|\left|\frac{\sin(\pi vY)}{\pi vY}\right|$$

傅立叶谱如图 2-4 (b) 和 2-4 (c) 所示。

2.2.2　离散函数的傅立叶变换

1. 一维离散函数的傅立叶变换

如果将一连续函数 $f(x)$ 离散化，如图 2-5 所示，则 $f(x)$ 可被表示为

$$\{f(x_0), f(x_0+\Delta x), f(x_0+2\Delta x), f(x_0+3\Delta x), \cdots, f(x_0+(N-1)\Delta x)\}$$

这里 N 为取样点，Δx 为样点间距。这样 $f(x)$ 就变为了离散函数 $f(n)$，即

$$f(n) = f(x_0 + n\Delta x) \tag{2-30}$$

$n=1, 2, 3, \cdots, N-1$，为离散值。

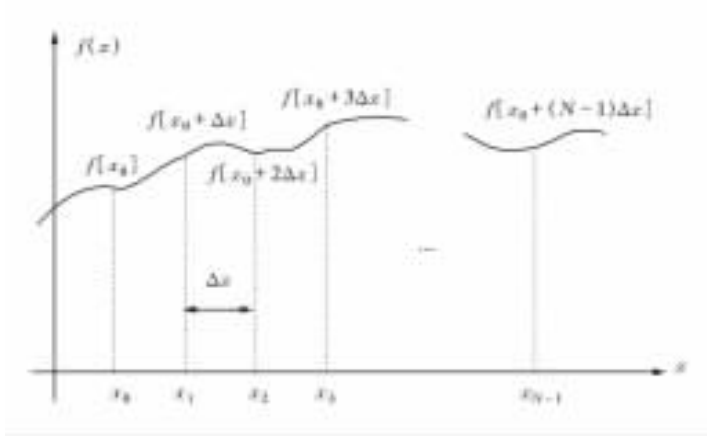


图 2-5　连续函数的离散化

这样一维离散函数 $f(n)$ 的傅立叶变换对为

$$F(u) = \frac{1}{\sqrt{N}}\sum_{n=0}^{N-1} f(n)\exp(-j2\pi un/N) \quad (u = 0, 1, 2, \cdots, N-1) \tag{2-31}$$

$$f(u) = \frac{1}{\sqrt{N}}\sum_{u=0}^{N-1} F(n)\exp(j2\pi un/N) \quad (n = 0, 1, 2, \cdots, N-1) \tag{2-32}$$

式中，系数 $\frac{1}{\sqrt{N}}$ 是为了计算方便而引入的，它不影响问题的本质。

【例 2-4】　函数如图 2-6 (a) 所示，在 $x_0=0.5$，$x_1=0.75$，$x_2=1.00$，$x_3=1.25$ 处取样，将连续函数转化为离散函数，如图 2-6 (b) 所示。

根据式 (2-31)，四个取样点，即 $N=4$，其傅立叶变换为

$$\begin{aligned} F(0) &= \frac{1}{2}\sum_{n=0}^{3} f(n)\exp(0) \\ &= \frac{1}{2}(f(0) + f(1) + f(2) + f(3)) \\ &= \frac{1}{2}(2 + 3 + 4 + 4) \\ &= 6.5 \end{aligned}$$

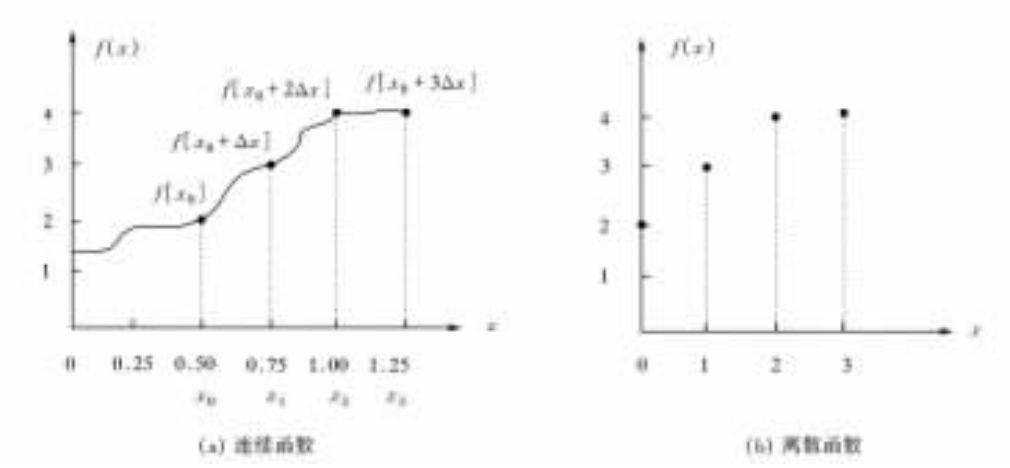


图 2-6 一维连续函数的离散化

$$
\begin{aligned}
F(1) &= \frac{1}{2}\sum_{n=0}^{3} f(n)\exp(-\mathrm{j}2\pi n/4) \\
&= \frac{1}{2}(2\mathrm{e}^{0} + 3\mathrm{e}^{-\mathrm{j}\pi/2} + 4\mathrm{e}^{-\mathrm{j}\pi} + 4\mathrm{e}^{-\mathrm{j}3\pi/2}) \\
&= \frac{1}{2}(-2 + \mathrm{j}) \\
F(2) &= \frac{1}{2}\sum_{n=0}^{3} f(n)\exp(-\mathrm{j}4\pi n/4) \\
&= \frac{1}{2}(2\mathrm{e}^{0} + 3\mathrm{e}^{-\mathrm{j}\pi} + 4\mathrm{e}^{-\mathrm{j}2\pi} + 4\mathrm{e}^{-\mathrm{j}3\pi}) \\
&= -\frac{1}{2}(1 + \mathrm{j}0) \\
F(3) &= \frac{1}{2}\sum_{n=0}^{3} f(n)\exp(-\mathrm{j}6\pi n/4) \\
&= \frac{1}{2}(2\mathrm{e}^{0} + 3\mathrm{e}^{-\mathrm{j}3\pi/2} + 4\mathrm{e}^{-\mathrm{j}3\pi} + 4\mathrm{e}^{-\mathrm{j}9\pi/2}) \\
&= -\frac{1}{2}(2 + \mathrm{j})
\end{aligned}
$$

为了用矩阵的形式表示，令

$$W_N = \exp(-\mathrm{j}2\pi/N)$$

这样式 (2-31) 可以写成

$$F(u) = \frac{1}{\sqrt{N}}\sum_{n=0}^{N-1} f(n)(W_N)^{un}$$

当 $N=4$，$u=0$，1，2，3 **时，根据上式可以得出**

$$
\begin{aligned}
F(0) &= \frac{1}{2}[(W_4)^0 f(0) + (W_4)^0 f(1) + (W_4)^0 f(2) + (W_4)^0 f(3)] \\
F(1) &= \frac{1}{2}[(W_4)^0 f(0) + (W_4)^1 f(1) + (W_4)^2 f(2) + (W_4)^3 f(3)] \\
F(2) &= \frac{1}{2}[(W_4)^0 f(0) + (W_4)^2 f(1) + (W_4)^4 f(2) + (W_4)^6 f(3)]
\end{aligned}
$$

$$F(3)=\frac{1}{2}\left[(W_4)^0 f(0)+(W_4)^3 f(1)+(W_4)^6 f(2)+(W_4)^9 f(3)\right]$$

上面四个式子用矩阵表示为

$$\begin{pmatrix}F(0)\\F(1)\\F(2)\\F(3)\end{pmatrix}=\frac{1}{2}\begin{pmatrix}W_4^0 & W_4^0 & W_4^0 & W_4^0\\W_4^0 & W_4^1 & W_4^2 & W_4^3\\W_4^0 & W_4^2 & W_4^4 & W_4^6\\W_4^0 & W_4^3 & W_4^6 & W_4^9\end{pmatrix}\begin{pmatrix}F(0)\\F(1)\\F(2)\\F(3)\end{pmatrix}$$

它的傅立叶谱为

$$|F(0)|=6.5$$

$$|F(1)|=\frac{\sqrt{5}}{2}$$

$$|F(2)|=\frac{1}{2}$$

$$|F(3)|=\frac{\sqrt{5}}{2}$$

2. 二维离散函数的傅立叶变换及性质

(1) 二维离散函数的傅立叶变换。

将一维离散的傅立叶变换直接推广到二维，假设某二维函数为 $f(m,n)$，它的傅立叶变换对为

$$F(u,v)=\frac{1}{\sqrt{MN}}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}f(m,n)\exp\left[-j2\pi(um/M+vn/N)\right] \tag{2-33}$$

$u=0,1,2,\cdots,M-1$；$v=0,1,2,\cdots,N-1$。

$$f(m,n)=\frac{1}{\sqrt{MN}}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1}F(u,v)\exp\left[j2\pi(um/M+vn/N)\right] \tag{2-34}$$

$m=0,1,2,\cdots,M-1$；$n=0,1,2,\cdots,N-1$。

在图像处理中，表示数字图像的二维矩阵通常选择方阵的形式，即 $M=N$，这样二维傅立叶变换对为

$$F(u,v)=\frac{1}{N}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}f(m,n)\exp\left[-j2\pi(um+vn)/N\right] \tag{2-35}$$

$u,v=0,1,2,\cdots,N-1$，并且

$$f(m,n)=\frac{1}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}F(u,v)\exp\left[j2\pi(um+vn)/N\right] \tag{2-36}$$

$m,u=0,1,2,\cdots,N-1$。

一维与二维离散傅立叶变换的傅立叶谱、相角和能量谱与连续函数的形式一样，只是将连续变量变为离散变量。为了给读者一个图像傅立叶变换的初步印象，在图 2-7 中示出了傅立叶变换图谱的例子。

(2) 二维离散函数傅立叶变换的性质

①可分离性

二维离散傅立叶变换对可分离为两部分之积，即

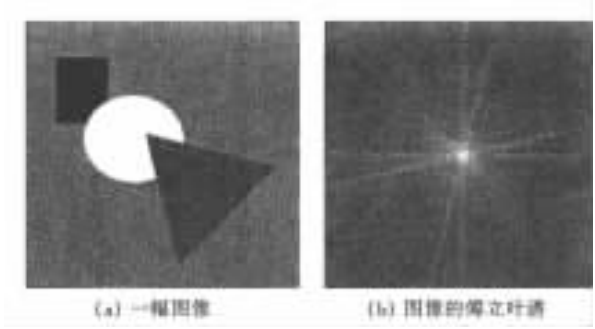


图 2-7 傅立叶变换图谱的例子(选自 N.Efford)

$$F(u,v)=\frac{1}{N}\sum_{m=0}^{N-1}\exp(-j2\pi um/N)\sum_{n=0}^{N-1}f(m,n)\exp(-j2\pi vn/N) \tag{2-37}$$

$u,v=0,1,2,\cdots,N-1$。

$$f(m,n)=\frac{1}{N}\sum_{u=0}^{N-1}\exp(j2\pi um/N)\sum_{v=0}^{N-1}F(u,v)\exp(j2\pi vn/N) \tag{2-38}$$

$m,n=0,1,2,\cdots,N-1$。

可见，一个二维离散的傅立叶变换可以分解为两个一维离散的傅立叶变换之积，即计算二维离散的傅立叶变换时可进行两次一维变换运算。

②线性

令 $F_1(u,v)$ 和 $F_2(u,v)$ 分别为二维离散函数 $f_1(m,n)$ 和 $f_2(m,n)$ 的傅立叶变换，则有

$$F(af_1(m,n)+bf_2(m,n))=aF_1(u,v)+bF_2(u,v) \tag{2-39}$$

这里 a，b 为常数。

③平移性

以一维情况为例，若 $f(n)$ 向右(或左)移动 k 位，则有

$$\begin{aligned}f(n-k)&=\frac{1}{\sqrt{N}}\sum_{u=0}^{N-1}F(u)\exp(j2\pi(n-k)u/N)\\&=\frac{1}{\sqrt{N}}\sum_{u=0}^{N-1}F(u)\exp(j2\pi nu/N)\exp(-j2\pi ku/N)\\&=\frac{1}{\sqrt{N}}\sum_{u=0}^{N-1}[F(u)\exp(-j2\pi ku/N)]\exp(j2\pi nu/N)\end{aligned} \tag{2-40}$$

可见，在空间域内函数平移 k 时，其对应的傅立叶变换要乘上一个指数项 $\exp(-j2\pi ku/N)$。

推广到二维的情况，当平面图像在空间域内平移 m_0，n_0 时，其对应的傅立叶变换要乘上一个指数项 $\exp(-j2\pi(um_0+vn_0)/N)$，即

$$f(m-m_0,n-n_0)\Leftrightarrow F(u,v)\exp[-j2\pi(um_0+vn_0)/N] \tag{2-41}$$

同理，若在频域内平移 u_0，v_0 时，其对应的 $f(m,n)$ 要乘上一个指数项 $\exp[j2\pi(mu_0+nv_0)/N]$，即

$$F(u-u_0,v-v_0)\Leftrightarrow f(m,n)\exp[j2\pi(mu_0+nv_0)/N] \tag{2-42}$$

平移性表明，当空间域内函数 $f(m,n)$ 产生平移时，在频域内只发生相移，而傅立叶变换

的幅值不变，即

$$\left| F(u, v) \exp\left[-\mathrm{j}2\pi(um_0 + vn_0)/N\right] \right| = \left| F(u, v) \right| \tag{2-43}$$

④周期性

以一维情况为例，离散傅立叶变换具有下面的性质，即

$$\begin{aligned} f(n - N) &= \frac{1}{\sqrt{N}} \sum_{u=0}^{N-1} F(u) \exp\left[\mathrm{j}2\pi(n + N)u/N\right] \\ &= \frac{1}{\sqrt{N}} \sum_{u=0}^{N-1} F(u) \exp(\mathrm{j}2\pi nu/N) \exp(\mathrm{j}2\pi u) \\ &= \frac{1}{\sqrt{N}} \sum_{u=0}^{N-1} F(u) \exp(\mathrm{j}2\pi nu/N) \\ &= f(n) \end{aligned} \tag{2-44}$$

同理

$$F(u) = F(u + N) \tag{2-45}$$

推广到二维情况，

$$f(m, n) = f(m + M, n + N) \tag{2-46}$$

$$F(u, v) = F(u + M, v + N) = F(u, v + N) = F(u + M, v) \tag{2-47}$$

这里 M，N 为取样点。可见离散傅立叶变换对具有周期性，其周期为取样点 M 和 N。

2.2.3　快速傅立叶变换

2.2.2 中介绍的离散傅立叶变换有一个缺点，即计算量较大，下面以一维离散傅立叶变换为例对此加以说明。根据例 2-4 有

$$F(u) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} f(n) (W_N)^{un} \tag{2-48}$$

这里 $W_N = \exp(-\mathrm{j}2\pi/N)$，$u = 0, 1, 2, \cdots, N-1$。

将式 (2-48) 展开 (这里为了方便讨论，不考虑系数$\frac{1}{\sqrt{N}}$) 得

$$F(0) = f(0) W_N^{00} + f(1) W_N^{01} + f(2) W_N^{02} + \cdots + f(N-1) W_N^{0(N-1)} \tag{2-49}$$

$$F(1) = f(0) W_N^{10} + f(1) W_N^{11} + f(2) W_N^{12} + \cdots + f(N-1) W_N^{1(N-1)} \tag{2-50}$$

$$F(2) = f(0) W_N^{20} + f(1) W_N^{21} + f(2) W_N^{22} + \cdots + f(N-1) W_N^{2(N-1)} \tag{2-51}$$

…

$$F(N-1) = f(0) W_N^{(N-1)0} + f(1) W_N^{(N-1)1} + f(2) W_N^{(N-1)2} + \cdots + f(N-1) W_N^{(N-1)(N-1)} \tag{2-52}$$

上述式子一共要进行多少次乘法和加法运算呢？对于每一个频率分量 u，$f * W$ 共有 N 次，各项相加有 $N-1$ 次，共有 N 个方程式，所以最后要进行 $N * N = N^2$ 次乘法，$N * (N-1)$ 次加法。若 N 很大，运算量也很大，运算时间较长。这样在某种程度上限制了它的使用范围，为了提高运算速度，人们提出了一种快速算法——快速傅立叶变换 (FFT，Fast Fourier Transform)，FFT 的基本原理和计算公式与前面介绍的离散傅立叶变换一样，它是一种运算次数减少的离散傅立叶变换。这里只介绍一种称为按时间抽取的 FFT 算法。

1. 基本原理

它是采用将长序列分解成短序列的策略，这里以一维情况为例，将 $f(n)$ 分解成短序

列，然后求短序列的傅立叶变换，以减少运算次数。

令

$$N=2^k \tag{2-53}$$

这里 k 为正整数。

将 $f(n)$ 分解成两部分，一个是偶数部分 $f(2n)$，另一个是奇数部分 $f(2n+1)$。这样傅立叶变换（为了计算方便，不考虑系数$\frac{1}{\sqrt{N}}$）可表示为

$$\begin{aligned} F(u) &= \sum_{n=0}^{N-1} f(n) W_N^{un} \\ &= \sum_{n=0}^{N/2-1} f(2n) W_N^{u(2n)} + \sum_{n=0}^{N/2-1} f(2n+1) W_N^{u(2n+1)} \end{aligned} \tag{2-54}$$

$u=0, 1, 2, \cdots, N-1$，$W_N=\exp(-j2\pi/N)$。

因为

$$W_N^{2nu} = W_{N/2}^{nu} \tag{2-55}$$

$$W_N^{u(2n+1)} = W_N^u W_{N/2}^{nu} \tag{2-56}$$

所以式（2-54）可写成

$$F(u) = \sum_{n=0}^{N/2-1} f(2n) W_{N/2}^{un} + W_N^u \sum_{n=0}^{N/2-1} f(2n+1) W_{N/2}^{un} \tag{2-57}$$

令

$$F_{偶}(u) = \sum_{n=0}^{N/2-1} f(2n) W_{N/2}^{un} \tag{2-58}$$

$$F_{奇}(u) = \sum_{n=0}^{N/2-1} f(2n+1) W_{N/2}^{un} \tag{2-59}$$

则式（2-57）变为

$$F(u) = F_{偶}(u) + W_N^u F_{奇}(u) \qquad \left(u=0, 1, 2, \cdots, \frac{N}{2}-1\right) \tag{2-60}$$

$F_{偶}(u)$ 和 $F_{奇}(u)$ 均为以$\frac{N}{2}$ 为周期的函数。

根据傅立叶变换的周期性，

$$F_{偶}\left(u+\frac{N}{2}\right) = F_{偶}(u) \tag{2-61}$$

$$F_{奇}\left(u+\frac{N}{2}\right) = F_{奇}(u) \tag{2-62}$$

根据式（2-60）得

$$F\left(u+\frac{N}{2}\right) = F_{偶}\left(u+\frac{N}{2}\right) + W_N^{u+N/2} F_{奇}\left(u+\frac{N}{2}\right) \tag{2-63}$$

因为

$$W_N^{u+N/2} = -W_N^u \tag{2-64}$$

所以

$$F\left(u+\frac{N}{2}\right) = F_{偶}(u) - W_N^u F_{奇}(u) \qquad \left(u=0, 1, 2, \cdots, \frac{N}{2}-1\right) \tag{2-65}$$

分析一下式（2-60）和式（2-65）会发现，这样表示的离散傅立叶变换使运算过程简化，即将 N 点的离散傅立叶变换分为两部分，式（2-60）是用于求出 $u=0, 1, 2, \cdots, \frac{N}{2}-1$ 点的傅

立叶变换；式 (2-65) 是用于求出 $u=\frac{N}{2}$，$\frac{N}{2}+1$，$\frac{N}{2}+2$，…，$N-1$ 点的傅立叶变换。在运算中，只需要计算$\frac{N}{2}$个点的复指数运算，其余运算只是加减法了，可见这样做使乘法的次数大大减少。

由此可见，FFT 就是将 N 个点的序列依次分解成短序列，然后求这些短序列的傅立叶变换，即第一次将 N 分解成$\frac{N}{2}$的傅立叶变换，经过第二次抽取时将$\frac{N}{2}$分解成$\frac{N}{4}$的傅立叶变换，依此类推，直到只剩下两点为止。式 (2-60) 和式 (2-65) 称为蝶形运算。

2. 信号流图

信号流图可以形象地表示出 FFT 的整体运算过程，因此，它成为分析和表示 FFT 的重要方法，式 (2-60) 和式 (2-65) 的信号流图为

在图 2-8 中，W_N^u 为加权因子。

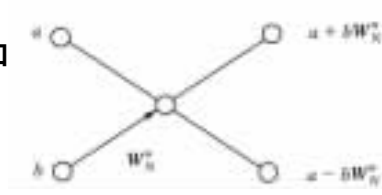


图 2-8　蝶形运算表达

【例 2-5】　画出 $N=8$ 时的信号流图，并用式 (2-60) 和式 (2-65) 表示出 $F(u)$ 的计算流程。

因为 $N=8$，即由 8 个样点组成序列

$\{f(0), f(1), f(2), f(3), f(4), f(5), f(6), f(7)\}$

首先将它们分解成四个偶数点 $\{f(0), f(2), f(4), f(6)\}$ 和四个奇数点 $\{f(1), f(3), f(5), f(7)\}$。然后进行第二次分解，即对于四个样点 $\{f(0), f(2), f(4), f(6)\}$ 组成的序列再进一步分解成两个偶数点 $\{f(0), f(4)\}$ 和两个奇数点 $\{f(2), f(6)\}$；同理分解四个点 $\{f(1), f(3), f(5), f(7)\}$ 成两个偶数点 $\{f(1), f(5)\}$ 和两个奇数点 $\{f(3), f(7)\}$。再进行第三次分解，将 $\{f(0), f(4)\}$ 分成偶数点 $\{f(0)\}$ 和奇数点 $\{f(4)\}$；将 $\{f(2), f(6)\}$ 分成偶数点 $\{f(2)\}$ 和奇数点 $\{f(6)\}$；将 $\{f(1), f(5)\}$ 分成偶数点 $\{f(1)\}$ 和奇数点 $\{f(5)\}$；将 $\{f(3), f(7)\}$ 分成偶数点 $\{f(3)\}$ 和奇数点 $\{f(7)\}$。上述过程可以用图 2-9 表示出，其信号流图如图 2-10 所示。下面以 $F(1)$ 的计算为例，计算图 2-10 所示的信号流图。

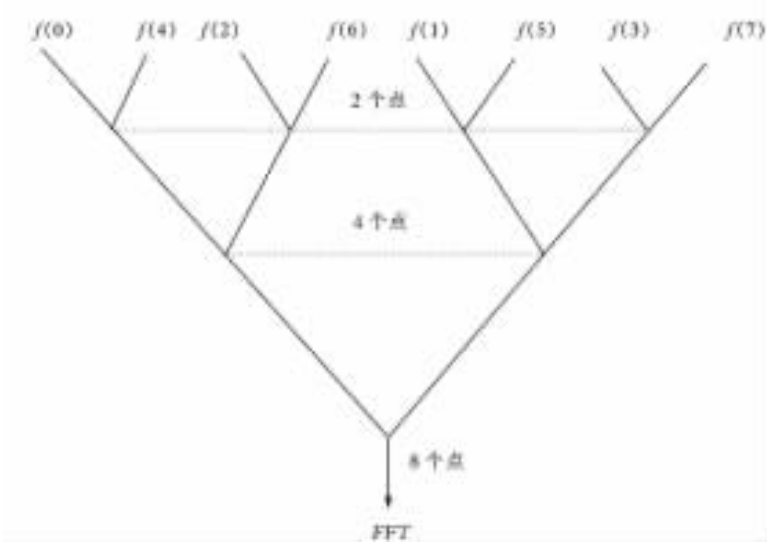


图 2-9　分解过程图示

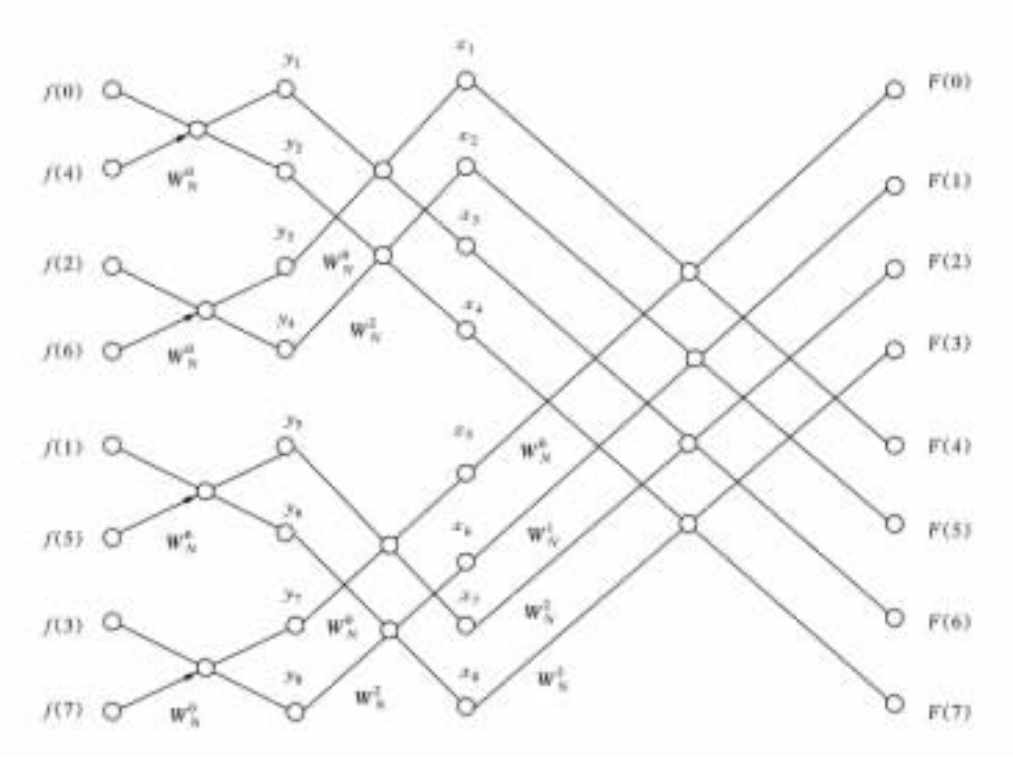


图 2-10　信号流图

$$F(1) = x_2 + W_N^1 x_6 = [y_2 + W_N^2 y_4] + W_N^1 [y_6 + W_N^2 y_8]$$
$$= [f(0) - W_N^0 f(4)] + W_N^2 [f(2) - W_N^0 f(6)] + W_N^1 \{[f(1) - W_N^0 f(5)] + W_N^2 [f(3) - W_N^0 f(7)]\}$$

因为　$W_N^0 = -W_N^4$

所以

$$F(1) = f(0) + W_N^4 f(4) + W_N^2 f(2) + W_N^{2+4} f(6) + W_N^1 f(1) + W_N^{1+4} f(5) + W_N^{1+2} f(3) + W_N^{1+2+4} f(7)$$
$$= f(0) + W_N^1 f(1) + W_N^2 f(2) + W_N^3 f(3) + W_N^4 f(4) + W_N^5 f(5) + W_N^6 f(6) + W_N^7 f(7)$$

3．Java **语言实现** FFT **变换**

在 CD **盘的** FFT **文件夹中示出了用** Java **语言编写的** FFT **变换程序，这里建立了** FFT **类、**Complex **类和** OneFft **类。在进行** FFT **变换时，**Complex **类实现复数的加、减、乘等运算；**OneFft **类进行一维的** FFT **运算，核心部分是蝶形算法；**FFT **类需要调用** Complex **类和** OneFft **类，首先对每一行进行一维** FFT **运算，然后进行位倒序，再对每一列进行一维** FFT **运算，最后，将变换的结果计算为频谱，并显示出来。清单** 2-1 **示出了** FFT **类的代码，**Complex **类和** OneFft **类的代码请见** CD **盘中** FFT **文件夹。在图** 2-11 **示出了程序运行的例子。**

程序清单 2-1　　FFT.java **代码**

```
//FFT.java

import java.awt.*;
```

```
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class FFT extends Frame {
    Image im, tmp;
    int iw, ih;
    int[] pixels;
    int [] newPixels;
    boolean flagLoad = false;

    OneFft of;

    //构造方法
    public FFT(){
        super("傅立叶变换");
        Panel pdown;
        Button load, run, quit;
        //添加窗口监听事件
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });

        pdown = new Panel();
        pdown.setBackground(Color.lightGray);

        load = new Button("装载图像");
        run = new Button("傅立叶变换");
        quit = new Button("退出");

        this.add(pdown, BorderLayout.SOUTH);

        pdown.add(load);
        pdown.add(run);
        pdown.add(quit);

        load.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
```

```
                jLoad_ActionPerformed(e);
            }
        });

        run.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jRun_ActionPerformed(e);
            }
        });

        quit.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });
    }

    public void jLoad_ActionPerformed(ActionEvent e){
        //利用 MediaTracker 跟踪图像的加载
        MediaTracker tracker = new MediaTracker(this);
        im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
        tracker.addImage(im,0);

        //等待图像的完全加载
        try{
        tracker.waitForID(0);
        }catch(InterruptedException e2){ e2.printStackTrace();}

        //获取图像的宽度 iw 和高度 ih
        iw=im.getWidth(this);
        ih=im.getHeight(this);
        pixels=new int[iw*ih];

        //获取图像的像素 pixels
        try{
        PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }
```

```
        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        flagLoad = true;
        repaint();
    }

    public void jRun_ActionPerformed(ActionEvent e){
        //必须先加载图像,然后才可以进行 FFT 变换
        if(flagLoad){

        //对图像进行傅立叶变换
        ColorModel cm = ColorModel.getRGBdefault();

        // 赋初值
        int w = 1;
        int h = 1;
        int wp = 0;
        int hp = 0;

        //计算进行傅立叶变换的宽度和高度(2 的整数次方)
        while(w * 2 <= iw)
        {
            w * = 2;
            wp++;
        }
        while(h * 2 <= ih)
        {
            h * = 2;
            hp++;
        }

        //分配内存
        Complex [] td = new Complex[h * w];
        Complex [] fd = new Complex[h * w];

        newPixels = new int[h * w];

        //初始化 newPixels
```

```
for(int i=0;i<h;i++)
{
    for(int j=0;j<w;j++)
    {
        newPixels[i*w+j]=pixels[i*iw+j]&0xff;
    }
}

//初始化 fd,td
for(int i=0;i<h;i++)
{
    for(int j=0;j<w;j++)
    {
        fd[i*w+j]=new Complex();
        td[i*w+j]=new Complex(newPixels[i*w+j],0);
    }
}

//初始化中间变量
Complex [] tempW1=new Complex[w];
Complex [] tempW2=new Complex[w];
for(int j=0;j<w;j++)
{
    tempW1[j]=new Complex(0,0);
    tempW2[j]=new Complex(0,0);
}

//在 y 方向上进行快速傅立叶变换
for(int i=0;i<h;i++)
{
    //每一行做傅立叶变换
    for(int j=0;j<w;j++)
    {
        tempW1[j]=td[i*w+j];
    }

    //进行一维 FFT 变换
    of=new OneFft();
    of.setData(tempW1,wp);
    tempW2=of.getData();
```

```
        for(int j=0;j<w;j++)
        {
            fd[i*w+j]=tempW2[j];
        }
    }

    //保存变换结果
    for(int i=0;i<h;i++)
    {
        for(int j=0;j<w;j++)
        {
            td[j*h+i]=fd[i*w+j];
        }
    }

    //初始化中间变量
    tempW1=new Complex[h];
    tempW2=new Complex[h];
    for(int j=0;j<h;j++)
    {
        tempW1[j]=new Complex(0,0);
        tempW2[j]=new Complex(0,0);
    }

    //对 x 方向进行傅立叶变换
    for(int i=0;i<w;i++)
    {
        //每一列做傅立叶变换
        for(int j=0;j<h;j++)
        {
            tempW1[j]=td[i*h+j];
        }

        //进行一维 FFT 变换
        of=new OneFft();
        of.setData(tempW1,hp);
        tempW2=of.getData();

        for(int j=0;j<h;j++)
```

```
            }
                fd[i * h + j] = tempW2[j];
            }
        }

        //计算频谱
        for(int i = 0;i < h;i++)
        {
            for(int j = 0;j < w;j++)
            {
                double re = fd[j * h + i].re;
                double im = fd[j * h + i].im;

                int ii = 0, jj = 0;
                int temp = (int)(Math.sqrt(re * re + im * im)/100);
                if(temp > 255) { temp = 255; }

                //第 i 行,j 列,变为:ii 行,第 jj 列
                if(i < h/2) { ii = i + h/2; } else { ii = i - h/2; }
                if(j < w/2) { jj = j + w/2; } else { jj = j - w/2; }

                newPixels[ii * w + jj] = temp;
            }
        }

        for(int i = 0;i < w * h;i++)
        {
            int alpha = cm.getAlpha(pixels[i]);
            int x = newPixels[i];
            newPixels[i] = alpha << 24 | x << 16 | x << 8 | x;
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(w, h, newPixels, 0, w);
        tmp = createImage(ip);

        repaint();
    }else{
        JOptionPane.showMessageDialog(null, "请先打开一幅图片!",
                "Alert", JOptionPane.WARNING_MESSAGE);
```

```
        }
    }
    public void jQuit_ActionPerformed(ActionEvent e){
        System.exit(0);
    }

    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flagLoad){
            g.drawImage(tmp,10,20,this);
        }else { }
    }

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[] args) {
        FFT fft = new FFT();
        fft.setSize(400,350);
        fft.show();
    }
}
```

图 2-11　Java 语言实现 FFT 变换的例子

2.3　沃尔什变换

2.2 中介绍的傅立叶变换要用到复数乘法，尽管用了 FFT，但运算的时间还是比较长的，在某些领域需要更为简便的变换方法，沃尔什变换就是其中的一种。沃尔什变换的基函数是由“+1”和“-1”组成的二值正交函数。

2.3.1　一维离散沃尔什变换

当取样点 $N=2^n$（n 为整数）时，函数 $f(x)$ 的离散沃尔什变换为

$$W(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x) \prod_{i=0}^{n-1} (-1)^{b_i(x)b_{n-1-i}(u)} \qquad (u = 0,1,2,\cdots,N-1) \tag{2-66}$$

其中

$$g(x,u) = \frac{1}{N}\prod_{i=0}^{n-1} (-1)^{b_i(x)b_{n-1-i}(u)} \tag{2-67}$$

称为沃尔什变换核。这里 $b_k(z)$ **表示** z **变量二进制第** k **位的值。如，** $z=6$**，它的二进制为** 110**，则** $b_0(z)=0$**，** $b_1(z)=1$**，** $b_2(z)=1$**。表 2-1 示出了部分沃尔什变换核的值。**

表 2-1　　N = 2，4，8 时的沃尔什变换核的值

u \ N	$N=2$		$N=4$				$N=8$							
u \ x	0	1	0	1	2	3	0	1	2	3	4	5	6	7
0	+	+	+	+	+	+	+	+	+	+	+	+	+	+
1	+	−	+	+	−	−	+	+	+	+	−	−	−	−
2			+	−	+	−	+	+	−	−	+	+	−	−
3			+	−	−	+	+	+	−	−	−	−	+	+
4							+	−	+	−	+	−	+	−
5							+	−	+	−	−	+	−	+
6							+	−	−	+	+	−	−	+
7							+	−	−	+	−	+	+	−

注：这里“+”代表+1；“−”代表−1；只计算沃尔什变换核中 $\prod_{i=0}^{n-1} (-1)^{b_i(x)b_{n-1-i}(u)}$ **的值。**

离散沃尔什逆变换为

$$f(x) = \sum_{u=0}^{N-1} W(u) \prod_{i=0}^{n-1} (-1)^{b_i(x)b_{n-1-i}(u)} \qquad (x = 0,1,2,\cdots,N-1) \tag{2-68}$$

其中

$$h(x,u) = \prod_{i=0}^{n-1} (-1)^{b_i(x)b_{n-1-i}(u)} \tag{2-69}$$

称为沃尔什逆变换核。

【例 2-6】　求 $N=4$ **时的沃尔什变换。**

根据式 (2-66) 有

$$W(u) = \frac{1}{4}\sum_{x=0}^{3} f(x) \prod_{i=0}^{1} (-1)^{b_i(x)b_{1-1-i}(u)} \qquad (u,x = 0,1,2,3)$$

可得

$$W(0) = \frac{1}{4}[f(0) + f(1) + f(2) + f(3)]$$

$$W(1) = \frac{1}{4}[f(0) + f(1) - f(2) - f(3)]$$

$$W(2) = \frac{1}{4}[f(0) - f(1) + f(2) - f(3)]$$

$$W(3) = \frac{1}{4}[f(0) - f(1) - f(2) + f(3)]$$

上述用矩阵表示为

$$\begin{bmatrix} W(0) \\ W(1) \\ W(2) \\ W(3) \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \end{bmatrix}$$

可见，上式等号右侧第一个矩阵为 $N=4$ 时的沃尔什变换核矩阵，所以沃尔什变换可以用矩阵表示为

$$\boldsymbol{W}(u) = \frac{1}{N}\boldsymbol{G}\boldsymbol{f}(x) \tag{2-70}$$

这里，$\boldsymbol{G}$ 为沃尔什变换核矩阵

2.3.2　二维离散沃尔什变换

将一维离散沃尔什变换推广到二维的情况，则二维离散沃尔什变换对为

$$W(u,v) = \frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\prod_{i=0}^{n-1}(-1)^{[b_i(x)b_{n-1-i}(u)+b_i(y)b_{n-1-i}(v)]} \tag{2-71}$$

$$(u,v=0,1,2,\cdots,N-1)$$

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} W(u,v)\prod_{i=0}^{n-1}(-1)^{[b_i(x)b_{n-1-i}(u)+b_i(y)b_{n-1-i}(v)]} \tag{2-72}$$

$$(x,y=0,1,2,\cdots,N-1)$$

其中，沃尔什正变换核为

$$g(x,y,u,v) = \frac{1}{N^2}\prod_{i=0}^{n-1}(-1)^{[b_i(x)b_{n-1-i}(u)+b_i(y)b_{n-1-i}(v)]} \tag{2-73}$$

沃尔什逆变换核为

$$h(x,y,u,v) = \prod_{i=0}^{n-1}(-1)^{[b_i(x)b_{n-1-i}(u)+b_i(y)b_{n-1-i}(v)]} \tag{2-74}$$

沃尔什变换核具有可分离性和对称性，即

$$g(x,y,u,v) = g(x,u)g(y,v) \tag{2-75}$$

$$h(x,y,u,v) = h(x,u)h(y,v) \tag{2-76}$$

可见，二维沃尔什变换可分解为二次一维的沃尔什变换。与一维沃尔什变换类似，二维沃尔什变换对的矩阵表达式为

$$\boldsymbol{W}(u,v) = \frac{1}{N^2}\boldsymbol{G}\boldsymbol{f}(x,y)\boldsymbol{G} \tag{2-77}$$

$$\boldsymbol{f}(x,y) = \boldsymbol{G}\boldsymbol{W}(u,v)\boldsymbol{G} \tag{2-78}$$

2.4　哈达玛变换

哈达玛变换是一种特殊排列的沃尔什变换，它的最大优点是变换核矩阵具有简单的递推关系，即高阶矩阵可以由低阶矩阵直接求得。

2.4.1　一维离散哈达玛变换

当取样点 $N=2^n$（n 为整数）时，函数 $f(x)$ 的离散哈达玛变换为

$$H(u)=\frac{1}{N}\sum_{x=0}^{N-1}f(x)(-1)^{\sum_{i=0}^{n-1}b_i(x)b_i(u)} \qquad (u=0,1,2,\cdots,N-1) \tag{2-79}$$

其中

$$g(x,u)=\frac{1}{N}(-1)^{\sum_{i=0}^{n-1}b_i(x)b_i(u)} \tag{2-80}$$

称为一维哈达玛变换核。

离散哈达玛逆变换为

$$f(x)=\sum_{u=0}^{N-1}H(u)(-1)^{\sum_{i=0}^{n-1}b_i(x)b_i(u)} \tag{2-81}$$

哈达玛逆变换核为

$$h(x,u)=(-1)^{\sum_{i=0}^{n-1}b_i(x)b_i(u)} \qquad (x=0,1,2,\cdots,N-1) \tag{2-82}$$

这里 $b_k(z)$ 表示 z 变量二进制第 k 位的值。表 2-2 示出了部分哈达玛变换核的值。

令

$$G_{H_2}=\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{2-83}$$

表示 $N=2$ 时的哈达玛变换核矩阵，从表 2-2 可见，哈达玛变换核矩阵具有下面的递推关系

$$G_{H_4}=\begin{bmatrix}G_{H_2} & G_{H_2}\\ G_{H_2} & -G_{H_2}\end{bmatrix} \tag{2-84}$$

表 2-2　　$N=2$，4，8 时的哈达玛变换核的值

N / x / U	$N=2$		$N=4$				$N=8$							
	0	1	0	1	2	3	0	1	2	3	4	5	6	7
0	+	+	+	+	+	+	+	+	+	+	+	+	+	+
1	+	−	+	−	+	−	+	−	+	−	+	−	+	−
2			+	+	−	−	+	+	−	−	+	+	−	−
3			+	−	−	+	+	−	−	+	+	−	−	+
4							+	+	+	+	−	−	−	−
5							+	−	+	−	−	+	−	+
6							+	+	−	−	−	−	+	+
7							+	−	−	+	−	+	+	−

注：这里“+”代表 +1；“−”代表 −1；只计算沃尔什变换核中 $(-1)^{\sum_{i=0}^{n-1}b_i(x)b_i(u)}$ 的值。

将式 (2-84) 加以推广，得

$$G_{H_{2N}}=\begin{bmatrix}G_{H_N} & G_{H_N}\\ G_{H_N} & -G_{H_N}\end{bmatrix} \tag{2-85}$$

式 (2-85) 给出了哈达玛变换核矩阵的递推关系。

2.4.2　二维离散哈达玛变换

将一维离散哈达玛变换推广到二维的情况，则二维离散哈达玛变换对为

$$H(u,v)=\frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)(-1)^{\sum_{i=0}^{n-1}[b_i(x)b_i(u)+b_i(y)b_i(v)]} \tag{2-86}$$

$$(u,v=0,1,2,\cdots,N-1)$$

$$f(x,y)=\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}H(u,v)(-1)^{\sum_{i=0}^{n-1}[b_i(x)b_i(u)+b_i(y)b_i(v)]} \tag{2-87}$$

$$(x,y=0,1,2,\cdots,N-1)$$

其中，哈达玛正变换核为

$$g(x,y,u,v)=\frac{1}{N^2}(-1)^{\sum_{i=0}^{n-1}[b_i(x)b_i(u)+b_i(y)b_i(v)]} \tag{2-88}$$

哈达玛逆变换核为

$$h(x,y,u,v)=(-1)^{\sum_{i=0}^{n-1}[b_i(x)b_i(u)+b_i(y)b_i(v)]} \tag{2-89}$$

哈达玛变换核具有可分离性和对称性，即

$$g(x,y,u,v)=g(x,u)g(y,v) \tag{2-90}$$

$$h(x,y,u,v)=h(x,u)h(y,v) \tag{2-91}$$

2.5　小波变换

前面介绍的傅立叶变换的正交基函数是正弦曲线，并且变换函数是在两个方向无限扩展的，从而导致了傅立叶变换的一个缺点，它反映的是信号或函数的整体特征，是一种全局的变换，无法表述信号的时频局域性质。而在实际问题中，有时仅关心信号在局部范围中的特征，如函数在某给定点附近的性质等。为了弥补这一缺陷，人们提出了一种变换称为小波变换 (WT，Wavelet Transform)，小波变换不同于其他变换的地方是它的基函数是由有限宽度的波——母小波 (Mother Wavelet) ——通过平移和伸缩产生的。小波变换是一种信号的时间-频率分析方法，它具有多分辨率分析的特点，而且在时频两域都具有表征信号局部特征的能力，即在低频部分具有较高的频率分辨率和较低的时间分辨率，在高频部分具有较高的时间分辨率和较低的频率分辨率，被誉为分析信号的数学显微镜。

2.5.1　连续小波变换

设 $\psi(x)$ 是一个实值函数，其傅立叶变换为 $\hat{\psi}(\omega)$，当 $\hat{\psi}(\omega)$ 满足条件

$$C_\psi=\int_{-\infty}^{\infty}\frac{|\hat{\psi}(\omega)|^2}{|\omega|}\mathrm{d}\omega<\infty \tag{2-92}$$

时，我们称 $\psi(x)$ 为一个基本小波或母小波。通过伸缩和平移基本小波 $\psi(x)$ 生成一组小波基函数 $\{\psi_{a,b}(x)\}$，

$$\psi_{a,b}(x)=\frac{1}{\sqrt{a}}\psi\left(\frac{x-b}{a}\right) \tag{2-93}$$

这里 a，b 均为实数，且 $a>0$，a 为尺度参数，它反映一个特定基函数的尺度；b 为平移参

数，指明沿 x 轴的平移位置。

一维函数 $f(x)$ 以小波 $\psi_{a,b}(x)$ 为基函数的连续小波变换为

$$W_f(a,b) = \langle f, \psi_{a,b} \rangle = \int_{-\infty}^{\infty} f(x)\,\psi_{a,b}(x)\,\mathrm{d}x \tag{2-94}$$

小波逆变换为

$$f(x) = \frac{1}{C_\psi}\iint_{-\infty}^{\infty}\frac{1}{a^2}W_f(a,b)\,\psi_{a,b}(x)\,\mathrm{d}a\,\mathrm{d}b \tag{2-95}$$

将一维小波变换推广到二维的情况，则二维函数 $f(x,y)$ 的连续小波变换为

$$W_f(a,b_x,b_y) = \iint_{-\infty}^{\infty} f(x,y)\,\psi_{a,b_x,b_y}(x,y)\,\mathrm{d}x\,\mathrm{d}y \tag{2-96}$$

其中，b_x，b_y 表示在两个方向上的平移。

二维连续小波逆变换为

$$f(x,y) = \iiint_{-\infty}^{\infty} W_f(a,b_x,b_y)\,\psi_{a,b_x,b_y}(x,y)\,\mathrm{d}a\,\mathrm{d}b_x\,\mathrm{d}b_y \tag{2-97}$$

式中 $$\psi_{a,b_x,b_y}(x,y) = \frac{1}{|a|}\psi\left(\frac{x-b_x}{a},\frac{y-b_y}{a}\right) \tag{2-98}$$

是一组二维小波基函数，而 $\psi(x,y)$ 是一个二维基本小波。

2.5.2 离散小波变换

这里提到的离散小波变换与以前习惯的离散化不同，它不是针对时间的离散化，而是针对连续的尺度参数 a 和连续平移参数 b 的离散化，对此要加以注意。通常，连续小波变换中的尺度参数 a 和连续平移参数 b 的离散形式为

$$a = a_0^m \quad (a_0 > 1) \tag{2-99}$$

$$b = nb_0a_0^m \ (b_0 > 0) \tag{2-100}$$

式中 a_0 和 b_0 为常数，m，n 为整数。

这样连续小波基函数 $\psi_{a,b}(x,y)$ 变为离散小波基函数 $\psi_{m,n}(x)$，即

$$\psi_{m,n}(x) = \frac{1}{\sqrt{a_0^m}}\psi\left(\frac{x-nb_0a_0^m}{a_0^m}\right) \tag{2-101}$$

一维函数 $f(x)$ 以小波 $\psi_{m,n}(x)$ 为基函数的离散小波变换为

$$W_f(m,n) = \langle f, \psi_{m,n} \rangle = \int_{-\infty}^{\infty} f(x)\,\psi_{m,n}(x)\,\mathrm{d}x \tag{2-102}$$

由于连续变量 a，b 离散化，所以离散小波逆变换为

$$f(x) = \frac{1}{C_\psi}\sum_m\sum_n W_f(m,n)\,\psi_{m,n}(x) \tag{2-103}$$

若 $a_0=2$，$b_0=1$，尺度参数 $a=2^m$，连续平移参数 $b=2^mn$，这样小波基函数为

$$\psi_{m,n}(x) = 2^{-m/2}\psi(2^{-m}x-n) \tag{2-104}$$

称为二进小波。二进小波对信息的分析具有变聚焦的作用。若某一放大倍数为 $a=2^{-m}$，它对应观测信号的某部分信息。若要观测更小的信息，就需要增加放大倍数，即减小 m 值；若要观测更粗糙的信息，就需要减小放大倍数，即加大 m 值。这样小波变换就具有显微镜特性，所以小波变换常被称为数学显微镜。

【例 2-7】 如下定义离散函数 $f(x)$，求 $f(x)$ 的小波变换矩阵。

$$f(x)=\begin{cases}9 & 0\leqslant x<\dfrac{1}{4}\\ 1 & \dfrac{1}{4}\leqslant x<\dfrac{1}{2}\\ 2 & \dfrac{1}{2}\leqslant x<\dfrac{3}{4}\\ 0 & \dfrac{3}{4}\leqslant x\leqslant 1\end{cases}$$

这里选择 Haar 小波作为母小波，即

$$\psi(x)=\begin{cases}1 & 0\leqslant x<\dfrac{1}{2}\\ -1 & \dfrac{1}{2}\leqslant x<1\\ 0 & \text{其他}\end{cases}\tag{2-105}$$

根据式 (2-105)，对母小波进行伸缩后得函数

$$\psi(2x)=\begin{cases}1 & 0\leqslant x<\dfrac{1}{4}\\ -1 & \dfrac{1}{4}\leqslant x<\dfrac{1}{2}\\ 0 & \text{其他}\end{cases}\tag{2-106}$$

根据式 (2-106)，再对母小波进行平移后得函数

$$\psi(2x-1)=\begin{cases}1 & \dfrac{1}{2}\leqslant x<\dfrac{3}{4}\\ -1 & \dfrac{3}{4}\leqslant x<1\\ 0 & \text{其他}\end{cases}\tag{2-107}$$

这样式 (2-105)、式 (2-106) 和式 (2-107) 就构成了一组小波基函数。当然也可以选择其他的尺度。在小尺度上，细节更加清楚，而较大的特征却不见了；在大尺度上，较大的特征明显而细节不清楚。可以用一个通式来表示该小波基函数，即

$$\psi_{m,n}(x)=\psi(2^m x-n)\tag{2-108}$$

式中　m，n 为整数，$m\geqslant 0$，$0\leqslant n\leqslant 2^m$。

为了构造正交小波基，需要引入一个尺度函数 $\Phi(x)$

$$\Phi(x)=\begin{cases}1 & 0\leqslant x\leqslant 1\\ 0 & \text{其他}\end{cases}\tag{2-109}$$

这里要求尺度函数 $\Phi(x)$ 满足尺度函数的容许条件，即

$$\int_{-\infty}^{\infty}\Phi(x)\,\mathrm{d}x=1\tag{2-110}$$

满足尺度函数与小波基函数的正交，即

$$<\Phi(x),\psi(x)>=0\tag{2-111}$$

根据式 (2-103)，代入尺度函数 $\Phi(x)$ 和小波基函数 $\psi_{m,n}(x)$，得

$$f(x)=3\Phi(x)+2\psi(x)+4\psi(2x)+1\psi(2x-1)\tag{2-112}$$

即

$$\begin{pmatrix}9\\1\\2\\0\end{pmatrix}=3\cdot\begin{pmatrix}1\\1\\1\\1\end{pmatrix}+2\cdot\begin{pmatrix}1\\1\\-1\\-1\end{pmatrix}+4\cdot\begin{pmatrix}1\\-1\\0\\0\end{pmatrix}+1\cdot\begin{pmatrix}0\\0\\1\\-1\end{pmatrix}$$

上式可进一步写为

$$\begin{pmatrix}9\\1\\2\\0\end{pmatrix}=\begin{pmatrix}1&1&1&0\\1&1&-1&0\\1&-1&0&1\\1&-1&0&-1\end{pmatrix}\cdot\begin{pmatrix}3\\2\\4\\1\end{pmatrix}$$

这样小波变换矩阵为

$$\begin{pmatrix}1&1&1&0\\1&1&-1&0\\1&-1&0&1\\1&-1&0&-1\end{pmatrix}$$

这个例子的思路可以在图像处理中借用。为了给读者一个图像小波变换的初步印象，在图 2-12 示出了一个 X 射线图像的小波变换应用实例。

图 2-12 X 射线图像的小波变换（选自 T Lehmann et al.）

练 习 题

1 某一数字图像为

$$\begin{pmatrix}1&1&1&1\\1&3&3&1\\1&3&3&1\\1&1&1&1\end{pmatrix}$$

求它的二维离散傅立叶变换（DFT）和二维离散沃尔什变换（DWT）。

2 将第一章练习 2 中存入计算机内的图像用 Java 语言对其进行 FFT 变换。

第 3 章　图像的数字化

3.1　概　　述

日常生活中所接触到的图画和照片等图像称为模拟图像，这种图像的灰度是空间位置的连续函数，它们是不能直接用计算机来处理的。为了使图像能在计算机内进行处理，首先必须将这种图像转化为数字图像。将模拟图像转化为数字图像的过程称为图像的数字化，它是将代表图像的连续信号转换成离散信号的过程。所谓图像的数字化，就是把图像分割成如图 3-1 所示的 $M \times N$ 个小区域，每个区域称为一个像素，这个过程称为取样；由于每一像素所对应的灰度幅值可以取任何值，即取样后的灰度幅值仍然是连续的，所以还应将灰度幅值分为 K 个小区间，每个区间只用一个整数值来表示某个像素的灰度值，这个过程称为量化。所以说图像的数字化是由取样和量化两部分组成的。

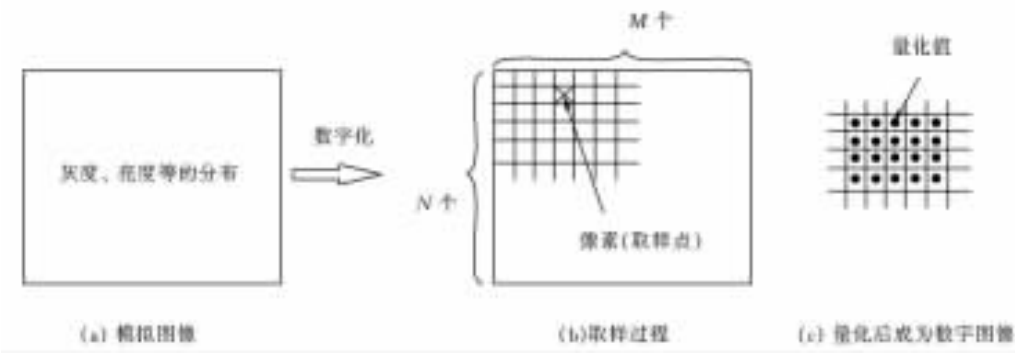


图 3-1　图像的数字化

3.1.1　取　样

取样就是使图像的灰度在空间坐标上离散化，即把一幅图像在空间上划分成许多个小区域，每个小区域称为一个样点、像素、样本或像元。把图像分割成像素的方法可以是多种多样的，如每个像素所占的小区域可以是正方形的、六角形的或三角形的，如图 3-2 所示。与之相对应的像素所构成的点阵则分别为正方形网格点阵、正三角形网格点阵和正六角形网格点阵。在上述各像素分割方案中，由于正方形网格点阵规范，易于在图像输入和输出设备上实现，从而被绝大多数图像采集、处理系统所采用，即正方形网格点阵是实际应用中常使用的像素分割方案。

在图 3-3 中，采用正方形网格点阵的方式，把一幅图像在空间上分成了 $M \times N$ 个像素，即这一幅图像在空间上用 $M \times N$ 个取样点来表示了，这 $M \times N$ 个取样点可以组成一个二维数组，在二维数组中，数组元素 (i, j) 表示第 i 列、第 j 行的像素，这个过程就是取样。

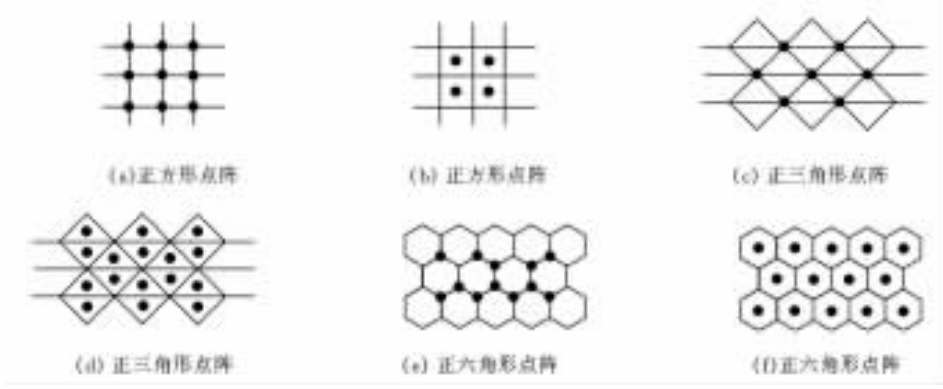


图 3-2 几种形式的图像取样点阵

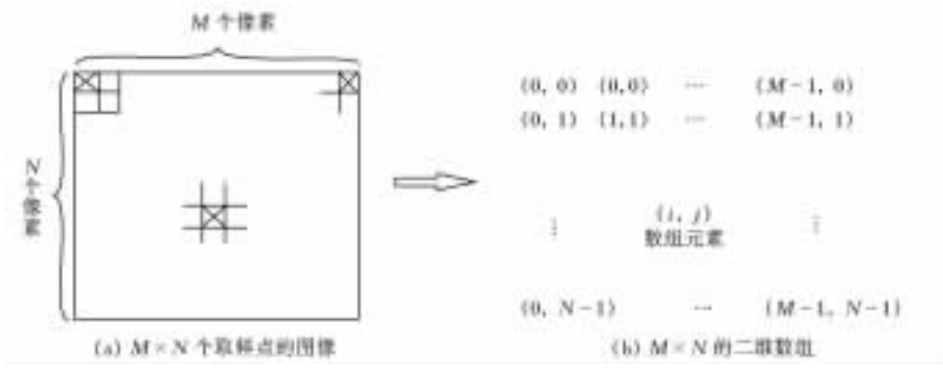


图 3-3 图像存储在二维数组中

可见，对图像完成取样后，得到的图像函数在空间坐标上是离散的。那么，一幅图像应取多少个样点呢？对此有一个约束条件，即由这些取样点采用某些方法能够正确重建原图像。通常取样的方法分为两类：⑴ 直接对表示图像的二维函数进行取样，所得的结果就是一个取样点阵，所以称为点阵取样；⑵ 先将图像函数进行某种正交变换，对其变换系数进行取样，称为正交系数取样。

3.1.2 量　化

经过取样后的图像只是在空间上被离散化了，但每个像素的灰度幅值还是一个无穷多个取值的连续变量，必须将其也转化成为有限个离散变量，即将其灰度的幅值分为 K 个离散的区间，每个区间只用一个整数值来表示某个像素的灰度值，这个过程就是量化。量化有两种方法：⑴ 均匀量化，它将像素灰度的幅值等间隔分挡取整；⑵ 非均匀量化，它将像素灰度的幅值不等间隔分挡取整。在图 3-4 中，将图像的灰度值分成了 256 个区间，每一个像素的灰度幅值就由这 256 个区间中一个区间所对应的一个量化值 q_i 来表示。

完成上述转化后，图像中的每个像素就有两个属性：位置和灰度量化值。位置是由二维数组的列和行决定的，灰度量化值是表示样点亮暗程度的整数。这样一幅图像就可用一个二维矩阵来表示了，这个二维矩阵就是数字图像，如图 3-5 所示。图 3-5 (b) 中，$f(x, y)$ 表示 x 列、y 行处像素的灰度值。

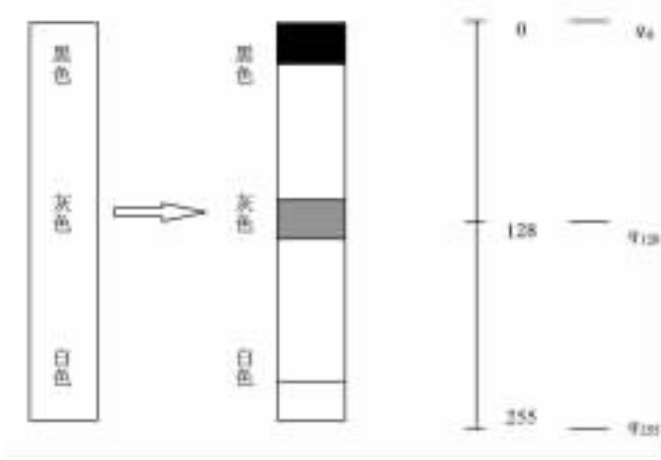


图 3-4　量化过程

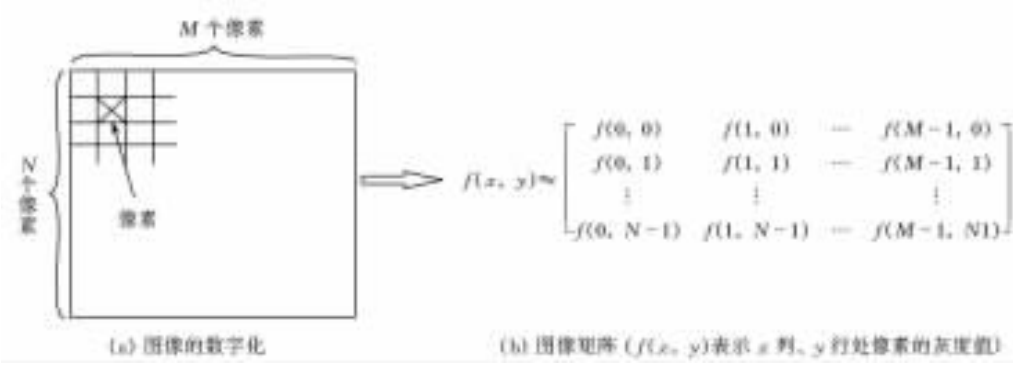


图 3-5　数字图像

3.1.3　Java 语言实现对图像像素的获取

从 3.1.1 和 3.1.2 中介绍的内容可知，在进行图像处理之前，必须进行取样和量化。即首先获取图像的像素并存入数组中，然后对二维数组中的像素灰度值进行量化，最后形成图像矩阵。在 Java 中，用 PixelGrabber 类来实现获取图像像素并存入数组中的功能，用 MemoryImageSource 类来实现在像素数组中形成图像矩阵。程序清单 3-1 示出了用 Java Application 获取图像的像素并保存在一个数组中，成为图像矩阵，然后又显示该图像。

程序清单 3-1　　ImagePixel.java **代码**

```
//ImagePixel.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
```

```
public class ImagePixel extends Frame {
Image im, tmp;
int i, iw, ih;
int[] pixels;

//ImagePixel 的构造方法,添加窗口监听事件
public ImagePixel(){
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    //利用 MediaTracker 跟踪图像的加载
    MediaTracker tracker = new MediaTracker(this);
    im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
    tracker.addImage(im,0);

    //等待图像的完全加载
    try{
    tracker.waitForID(0);
    }catch(InterruptedException e){ e.printStackTrace();}

    //获取图像的宽度 iw 和高度 ih
    iw=im.getWidth(this);
    ih=im.getHeight(this);

    pixels=new int[iw*ih];

    try{
    PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
    pg.grabPixels();
    }catch (InterruptedException e) {
        e.printStackTrace();
    }

    //对 RGB 值和 Alpha 值进行重新计算和赋值
    ColorModel cm=ColorModel.getRGBdefault();
    for(i=0;i<iw*ih;i++)
    {
```

```
            int alpha = 100;
            int red = cm.getRed(pixels[i]);
            int green = cm.getGreen(pixels[i]);
            int blue = cm.getBlue(pixels[i]);
            pixels[i] = alpha<<24 | red<<16 | green<<8 | blue;
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
    }

    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        g.drawImage(im, 10, 20, this);
        g.drawImage(tmp, iw + 50, 20, this);
    }

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[] args) {
        ImagePixel ip = new ImagePixel();
        ip.setSize(300, 300);
        ip.show();
    }
}
```

3.1.4　取样、量化与图像的质量

令一幅图像取 $M \times N$ 个样点，对每个样点灰度值的量化级数目为 K。如果 $K = 2^m$，并在量化过程中使用自然二进制码，那么一个样点就可以用 m 比特二进制表示。这样表示一幅图像的比特数为 $M \times N \times m$。现在的问题是，对于某一特定的比特数，如何选取 M、N 和 K 才能使重建后的图像信息损失最小呢？对此没有统一的模式，它完全取决于图像的具体情况。图 3-6 表示了具有相同的比特数，但不同的 M、N 和 K 值的两幅图像。在图 3-6 (a) 中，$M = N = 128$，$K = 64$；而在图 3-6 (b) 中，$M = N = 256$，$K = 16$；重建后两者的效果是不同的。图 3-7 所示的情况与上述相类似，在图 3-7 (a) 中，$M = N = 128$，$K = 64$；而在图 3-7 (b) 中，$M = N = 256$，$K = 16$；重建后两者的效果也不相同。

下面再进一步看看不同的 N (这里 $N = M$) 和 K 对图像效果的影响。在图 3-8 中，$K = 256$ 保持不变，N 取不同的值，即 $N = 256$ (图 3-8 (a))，$N = 128$ (图 3-8 (b))，$N = 64$ (图 3-8 (c))，$N = 32$ (图 3-8 (d))，$N = 16$ (图 3-8 (e))，$N = 8$ (图 3-8 (f))；可见，随着 N 的减少，重建图像失真严重。图 3-8 的效果是根据程序清单 3-2 用 Java Application 实现的，程序清单如下所示。

图 3-6 M，N 和 K 对图像质量的影响 (选自 C.Rafael et al.)
(a) 和 (b) 的比特数相同，但 M，N 和 K 的值不同；

图 3-7 M、N 和 K 对图像质量的影响 (选自 C.Rafael et al.)
(a) 和 (b) 具有相同的比特数，不同的 M，N 和 K 值；

程序清单 3-2　　Sample.java **代码**

```
//Sample.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
```

图 3-8　*K* = 256 保持不变，*N* 取不同值的效果

```
public class Sample extends Frame {
Image im, tmp;
int iw, ih;
int[ ] pixels;
boolean flag = false;
```

```
//构造方法
public Sample(){
    super("Sample");
    Panel pdown;
    Button load, run, quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    pdown = new Panel();
    pdown.setBackground(Color.lightGray);
    load=new Button("装载图像");
    run = new Button("进行采样");
    quit=new Button("退出");
    this.add(pdown, BorderLayout.SOUTH);
    pdown.add(load);
    pdown.add(run);
    pdown.add(quit);

    load.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jLoad_ActionPerformed(e);
        }
    });

    run.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jRun_ActionPerformed(e);
        }
    });

    quit.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jQuit_ActionPerformed(e);
        }
    });
```

```
}

public void jLoad_ActionPerformed(ActionEvent e){
    //利用 MediaTracker 跟踪图像的加载
    MediaTracker tracker = new MediaTracker(this);
    im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
    tracker.addImage(im,0);

    //等待图像的完全加载
    try{
    tracker.waitForID(0);
        }catch(InterruptedException e2){ e2.printStackTrace();}

        //获取图像的宽度 iw 和高度 ih
    iw=im.getWidth(this);
    ih=im.getHeight(this);
    pixels=new int[iw*ih];

    try{
    PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
    pg.grabPixels();
    }catch (InterruptedException e3) {
        e3.printStackTrace();
    }

    //将数组中的像素产生一个图像
    ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
    tmp=createImage(ip);
    flag=true;
    repaint();
}

public void jRun_ActionPerformed(ActionEvent e){

    try{
    PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
    pg.grabPixels();
    }catch (InterruptedException e3) {
        e3.printStackTrace();
    }
```

```
        //设定
        int grey=iw;
        String s=JOptionPane.showInputDialog(null,"请输入 N 值(256/128/64/32/16/8):");

        if(s!=null){
            grey=Integer.parseInt(s);
        }

        //检查输入是否正确
        if((grey>256)|(grey<8)){
            grey=256;
            JOptionPane.showMessageDialog(null,"输入不正确,请重新输入!");
        }

        //对图像进行采样
        ColorModel cm=ColorModel.getRGBdefault();
        for(int i=0;i<iw*ih-1;i=i+(int)(256/grey))
        {
            int red,green,blue;
            int alpha=cm.getAlpha(pixels[i]);

            red=cm.getRed(pixels[i]);
            green=cm.getGreen(pixels[i]);
            blue=cm.getBlue(pixels[i]);

            for(int j=0;j<(int)(256/grey);j++)
            {
                pixels[i+j]=alpha<<24|red<<16|green<<8|blue;
            }
        }
        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        flag=true;
        repaint();
    }

    public void jQuit_ActionPerformed(ActionEvent e)
    {
        System.exit(0);
```

```
}

//调用 paint()方法,显示图像信息。
public void paint(Graphics g){
    if(flag){
        g.drawImage(tmp,10,20,this);
    }else { }
}
//定义 main 方法,设置窗口的大小,显示窗口
public static void main(String[] args) {
    Sample sample = new Sample();
    sample.setLocation(300,200);
    sample.setSize(350,350);
    sample.show();
}
}
```

在图 3-9 中，$N = 512$ 保持不变，K 取不同的值，即 $K = 256$（图 3-9 (a)），$K = 128$（图 3-9 (b)），$K = 64$（图 3-9 (c)），$K = 32$（图 3-9 (d)），$K = 16$（图 3-9 (e)），$K = 8$（图 3-9 (f)），$K = 4$（图 3-9 (g)），$K = 2$（图 3-9 (h)）；随着 K（m）的减少，重建图像失真严重。图 3-9 的效果是根据程序清单 3-3 用 Java Application 实现的，运行程序如下所示。

程序清单 3-3　　GreyLevel.java 代码

```
//GreyLevel.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class GreyLevel extends Frame {
Image im,tmp;
int i,iw,ih;
int[] pixels;
boolean flag=false;

//构造方法
public GreyLevel(){
    super("GreyLevel");
    Panel pdown;
    Button load,run,quit;
```

图 3-9　*N* = 512 保持不变，*K* 取不同值的效果

```
//添加窗口监听事件
addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
            System.exit(0);
      }
});

pdown = new Panel();
pdown.setBackground(Color.lightGray);
load=new Button("装载图像");
run = new Button("进行量化");
quit=new Button("退出");
```

```
        this.add(pdown,BorderLayout.SOUTH);
        pdown.add(load);
        pdown.add(run);
        pdown.add(quit);

        load.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jLoad_ActionPerformed(e);
            }
        });

        run.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jRun_ActionPerformed(e);
            }
        });

        quit.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });

    }

    public void jLoad_ActionPerformed(ActionEvent e){
        //利用 MediaTracker 跟踪图像的加载
        MediaTracker tracker = new MediaTracker(this);
        im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
        tracker.addImage(im,0);

        //等待图像的完全加载
        try{
        tracker.waitForID(0);
        }catch(InterruptedException e2){ e2.printStackTrace();}

        //获取图像的宽度 iw 和高度 ih
        iw=im.getWidth(this);
        ih=im.getHeight(this);
        pixels=new int[iw*ih];
```

```
        try{
        PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        flag = true;
        repaint();
    }

    public void jRun_ActionPerformed(ActionEvent e){

        try{
        PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //设定默认值为 256 级灰度
        int level = 256;
            String s = JOptionPane.showInputDialog(null, "请输入量化级 K 值
(256/128/64/32/16/8/4/2):");

            if(s! = null){
                level = Integer.parseInt(s);
            }

        //检查输入是否正确
        if((level > 256)|(level < 0))
        {
            level = 256;
            JOptionPane.showMessageDialog(null, "输入不正确,请重新输入!");
        }

        int greyLevel = 256/level;
```

```
            //对图像进行量化处理
            ColorModel cm = ColorModel.getRGBdefault();
            for(i = 0;i < iw * ih;i ++ )
            {
                int alpha = cm.getAlpha(pixels[i]);
                int grey = cm.getRed(pixels[i]);
                int temp = grey/greyLevel;

                grey = temp * greyLevel;
                pixels[i] = alpha << 24 | grey << 16 | grey << 8 | grey;
            }

            //将数组中的像素产生一个图像
            ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
            tmp = createImage(ip);
            flag = true;
            repaint();
        }

public void jQuit_ActionPerformed(ActionEvent e)
{
    System.exit(0);
}

//调用 paint()方法,显示图像信息。
public void paint(Graphics g){
    if(flag){
        g.drawImage(tmp, 10, 20, this);
    }else { }
}

//定义 main 方法,设置窗口的大小,显示窗口
public static void main(String[ ] args) {
    GreyLevel gl = new GreyLevel();
    gl.setLocation(300, 200);
    gl.setSize(350, 350);
    gl.show();
}
}
```

由上述结果可见，图像的质量分别与 M、N 和 K 有关，M、N 和 K 的取值越大，图像的失真越小。但 M、N 和 K 增加时，图像所占的比特数增加，即图像的存储量加大。在实际应用中，M、N 和 K 如何取值才能得到满意的结果，很难讲出一个统一的方案，通常根据图像的内容和应用要求以及系统本身的技术指标来选定。如供人眼观察的图像 m 取 5～8 就可以了。而对于卫星片和航海片，为了区别图像中灰度变化不大的目标，往往 m 取 8～12。

3.2 点阵取样原理

若对图像函数 $f(x, y)$ 沿 x 方向取 M 个样点，沿 y 方向取 N 个样点，便得到了一个 $M \times N$ 的二维取样数组。现在的主要问题是应该取多少个样点，即取样密度应多大才能使重建的图像没有信息损失。对此，点阵取样原理有一个基本要求，即取样后不丢失信息而能完整地恢复原来的图像。对理想的、无限大的图像函数 $f(x, y)$ 点阵取样的数学处理方法是引入一个由函数组成的空间抽样函数 $s(x, y)$，将 $s(x, y)$ 函数与 $f(x, y)$ 函数相乘，所得的结果就是取样后的函数 $f_s(x, y)$，如图 3-10 所示。

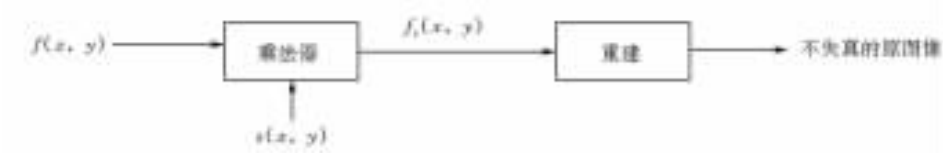


图 3-10 点阵取样基本过程

图 3-10 可用公式表示如下

$$f_s(x, y) = f(x, y)\, s(x, y) \tag{3-1}$$

下面首先从一维点阵取样原理介绍起，然后再推广到二维的情况。

3.2.1 一维点阵取样

假设一函数 $f(x)$ 的定义域为 $(-\infty, +\infty)$，并且 $f(x)$ 的傅立叶变换 $F(u)$ 的频率变量 u 的定义域是 $[-U_c, U_c]$，如图 3-11 (a) 和 3-11 (b) 所示，这样的函数称为有限带宽函数。

引入一维空间抽样函数 $s(x)$，如图 3-11 (c) 所示，并

$$s(x) = \sum_{i=-\infty}^{i=\infty} \delta(x - i\Delta x) \tag{3-2}$$

由卷积定理可知，$f(x)$ 与 $s(x)$ 在 x 域乘积的傅立叶变换等于它们各自的傅立叶变换在频域中的卷积——$F(u) * S(u)$，这样就得到了 $f(x)s(x)$ 积的傅立叶变换，如图 3-11 (f) 所示。注意 $F(u) * S(u)$ 是周期为$\frac{1}{\Delta x}$的函数，并且各个波形是可以重叠的。比如，在第一个周期内，如果$\frac{1}{2\Delta x}$小于 U_c，将发生波形的重叠。为了避免重叠的发生，要求取样间隔满足

$$\frac{1}{\Delta x} \geqslant 2U_c \tag{3-3}$$

或

$$\Delta x \leqslant \frac{1}{2U_c} \tag{3-4}$$

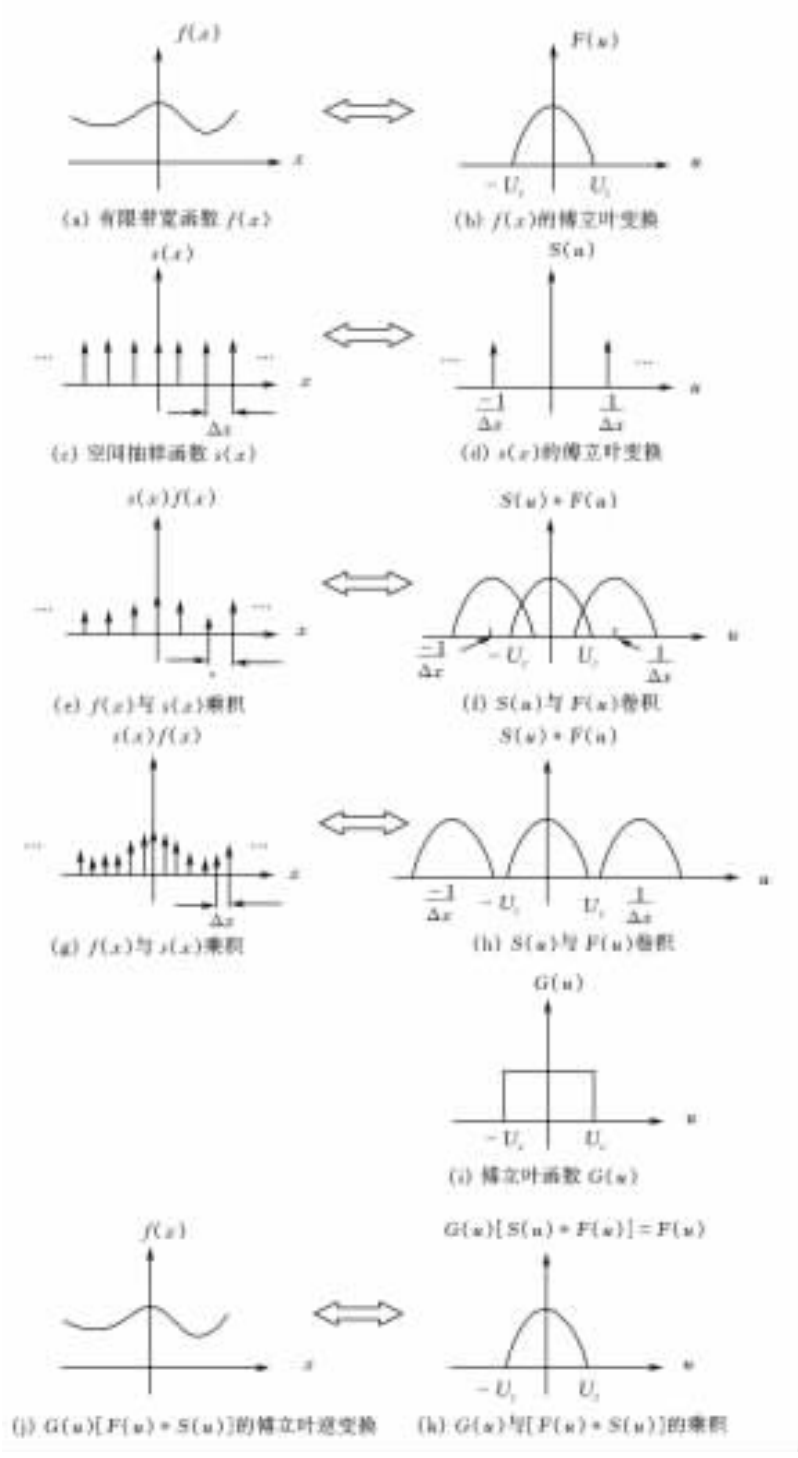


图 3-11　一维点阵取样示意图

这就是一维函数的点阵取样定理。

如果 Δx 减少，在频域中的周期函数就没有重叠，如图 3-11 (g) 和 3-11 (h) 所示。如果

在频域中引入函数（图 3-11（i）），

$$G(u) = \begin{cases} 1 & -U_c \leqslant u \leqslant U_c \\ 0 & \text{其他} \end{cases} \tag{3-5}$$

则 $G(u)$ 与 $F(u) * S(u)$ 的乘积如图 3-11（k）所示，这个乘积的傅立叶逆变换就是函数 $f(x)$（图 3-11（j））。

将上述结果加以推广，在图 3-12 中的取样区间为 [0，X]，引入一个函数

$$h(x) = \begin{cases} 1 & 0 \leqslant x \leqslant X \\ 0 & \text{其他} \end{cases} \tag{3-6}$$

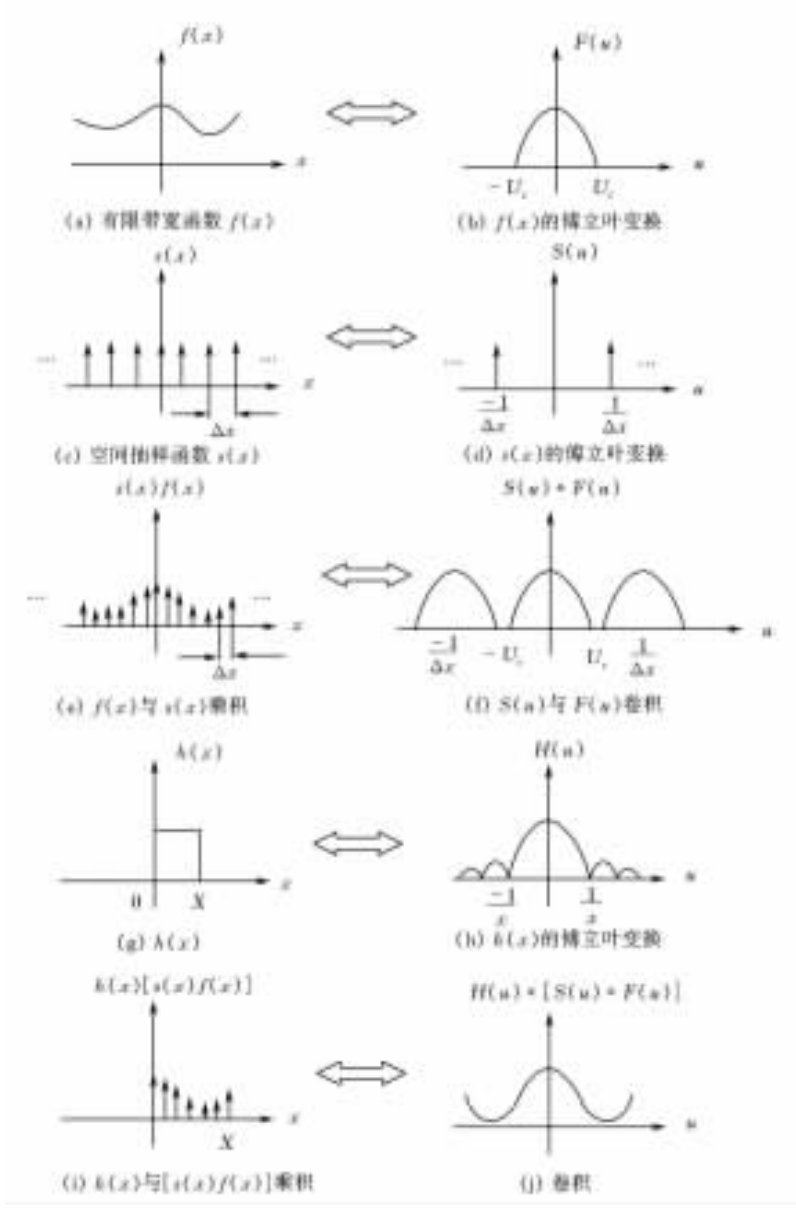


图 3-12 取样示意图

将函数 $h(x)$ 乘以 $f(x)s(x)$，其结果如图 3-12 (i) 所示，它们的傅立叶变换如图 3-12 (j) 所示。根据图 3-12 的情况，如果函数 $f(x)$ 在 $[0, X]$ 内取 N 个样点，样点的间距为 Δx (如图 3-13 (a) 所示)，则

$$N\Delta x = X \tag{3-7}$$

根据前面介绍的运算思路，从图 3-13 (a) 就可以得到图 3-13 (e) 和图 3-13 (f)。从图 3-13 (f) 可见，在频域中有

$$N\Delta u = \frac{1}{\Delta x} \tag{3-8}$$

这样就得到了空间域的取样间隔 Δx 与频域中对应样点间距 Δu 间的关系：

$$\Delta u = \frac{1}{N\Delta x} \tag{3-9}$$

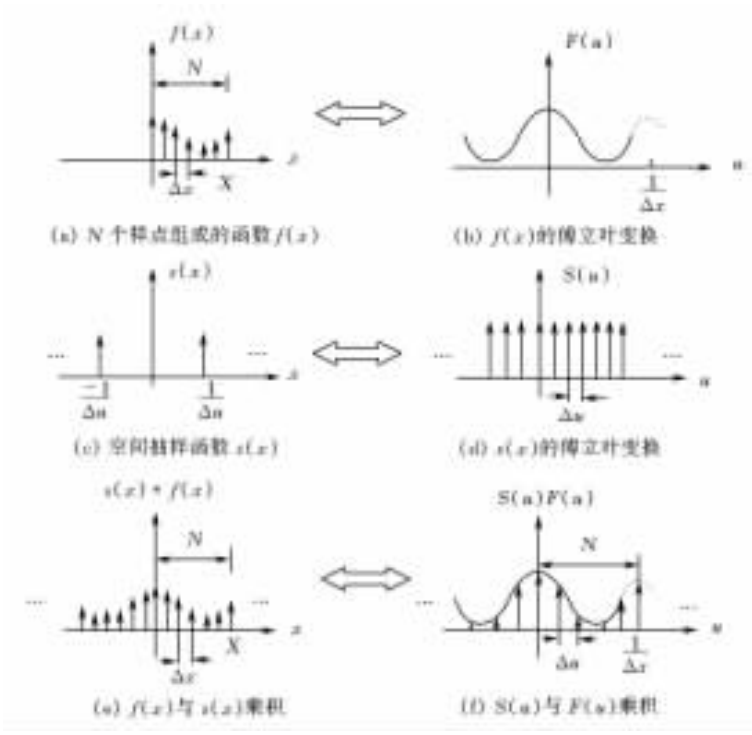


图 3-13　离散傅立叶变换

3.2.2　二维点阵取样

上面介绍的取样定理可以推广二维函数的情况。引入二维空间抽样函数 (如图 3-14 所示) 为

$$s(x, y) = \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x - i\Delta x, y - j\Delta y) \tag{3-10}$$

令 $f(x, y)$ 为图像函数，取样后的函数为 $f_s(x, y) = s(x, y)f(x, y)$。$S(u, v)$ 是 $s(x, y)$ 的傅立叶变换，$F(u, v)$ 是 $f(x, y)$ 的傅立叶变换。在频域中，$s(x, y)f(x, y)$ 的傅立叶变换是 $S(u, v)$ 与 $F(u, v)$ 的卷积。如果 $f(x, y)$ 是一个有限带宽的函数，则 $S(u, v)$ 与 $F(u, v)$ 的卷积的结果如图 3-15 所示。

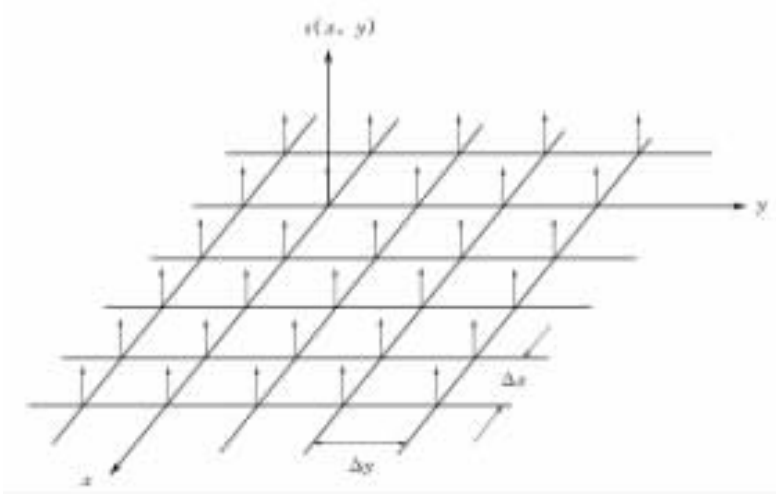


图 3-14 空间抽样函数

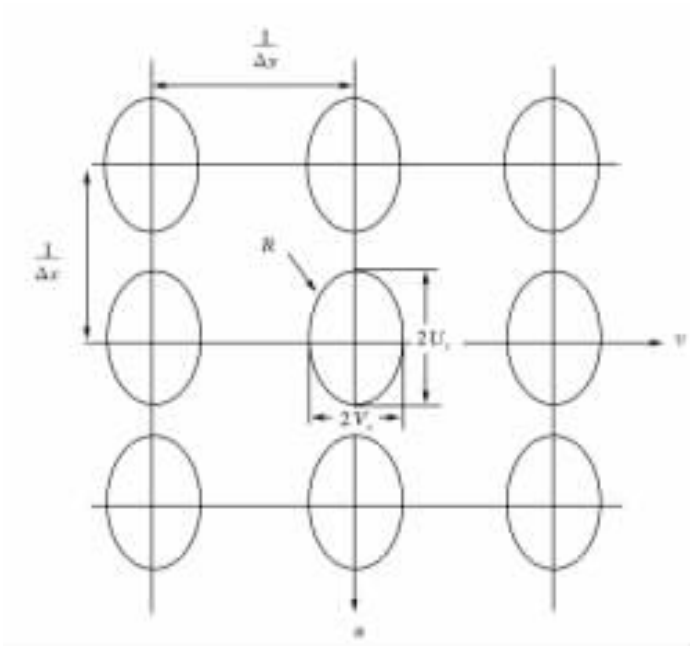


图 3-15 二维有限带宽取样函数在频域中的表示

这里 $2U_c$ 和 $2V_c$ 分别表示频域中在 U 和 V 方向 R 的边长。与一维函数相类似，如果 $\frac{1}{\Delta x} \geqslant 2U_c$ 和 $\frac{1}{\Delta y} \geqslant 2V_c$，并在频域中引入函数

$$G(u, v) = \begin{cases} 1 & \text{在 } R \text{ 范围内} \\ 0 & \text{其他} \end{cases} \tag{3-11}$$

则 $G(u, v)[S(u, v) * F(u, v)]$ 的傅立叶逆变换就是函数 $f(x, y)$。这样就得到了二维点阵取样定理

$$\Delta x \leqslant \frac{1}{2U_c} \tag{3-12a}$$

$$\Delta y \leqslant \frac{1}{2V_c} \tag{3-12b}$$

与一维情况类似，对于 $N \times N$ 的图像，在空间域与频域中取样间隔的关系为

$$\Delta u = \frac{1}{N\Delta x} \tag{3-13a}$$

$$\Delta v = \frac{1}{N\Delta y} \tag{3-13b}$$

3.3　最佳量化

在数字图像处理中，取样完成后还要进行量化。即对图像 $f(x, y)$ 取样完成后，得到函数 $f_s(x, y)$，但每个样本的灰度幅值还是一个无穷多个取值的连续变量，必须将其也转化为有限个离散值。也就是说，将样本的灰度取值范围分成若干个区间，然后用单个值来代表这一区间内的所有可能的值。若将连续的灰度幅值变量分成 K 个区间，则称为 K 级量化。

下面是一个量化过程的例子，如图 3-16 所示。这里将样本值的取值范围分成 $2^6 = 64$ 个区间，$\{q_i\}$ 为一组输出的量化值，q_i 代表一个整数。当图像信号样本幅度与一组判别层比较时，若样本幅度落在两个判别层之间，则该样本便被量化在这两个判别层之间。比如，一样本幅度值为 31.4，介于 31 与 32 之间，这一样点幅度取值被量化在 31～32 区间内，该区间的输出量化值为 q_{31}，因此量化后就用 q_{31} 代表该像素的灰度值。

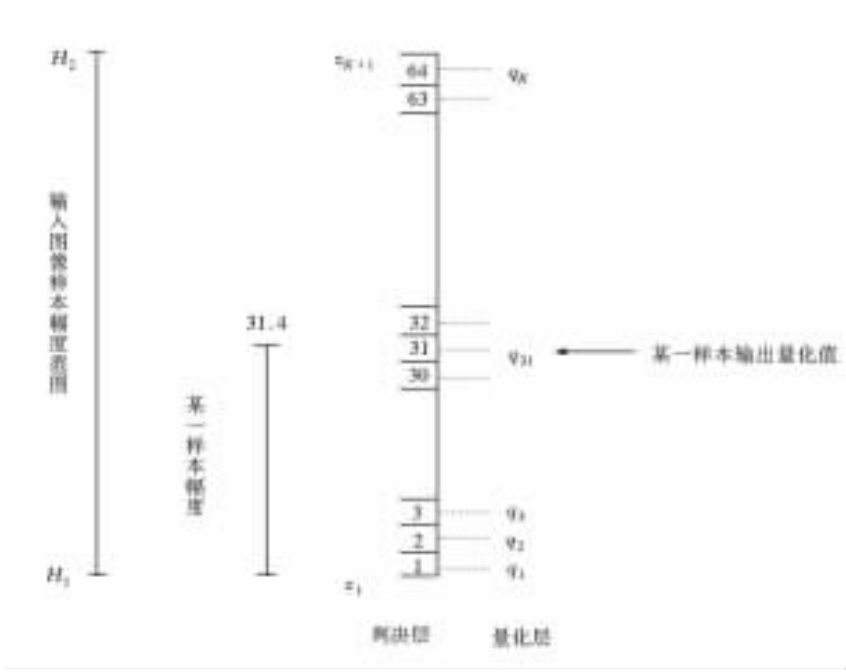


图 3-16　量化过程

可见，当输入图像样本幅度在任意两个判别层 z_i 和 z_{i+1}之间的任何位置时，其量化输出都是 q_i，这必然引起量化输出和输入之间的误差，这种误差称为量化误差。

为了减少量化误差，下面介绍一种量化方法——最佳量化。所谓最佳量化，是使量化误差为最小的量化方法。这里首先假设样点的灰度取值范围是 $H_1 - H_2$ (如图 3-17 所示)，灰度样点被量化成 K 个区间，每个区间的量化输出值为 q_1，q_2，…，q_k，输入图像样本灰度值的判别层用 z_1，z_2，…，z_{k+1}表示。

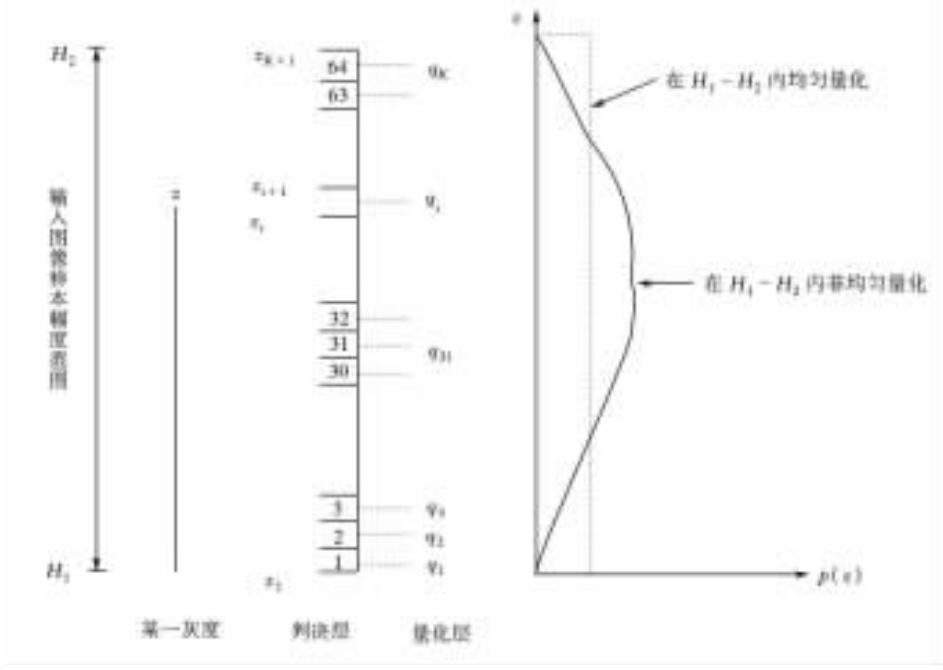


图 3-17　量化过程

令

δ_q^2 表示某一输入和输出之间的均方量化误差，即

$$\delta_q^2 = \sum_{i=1}^{K} \int_{z_i}^{z_{i+1}} (z - q_i)^2 p(z) \, dz \tag{3-14}$$

式中　$p(z)$ ——图像样本灰度 z 出现的概率。

下面要确定：

(1) 各个量化子区间对应的量化值 q_i。

(2) 各个量化子区间 $[z_i, z_{i+1}]$。

要使量化误差为最小，即 δ_q^2 为最小，需要将 δ_q^2 对 z_i 和 q_i 求微分，并令其导数为 0，根据式 (3-14) 得

$$\frac{\partial}{\partial z_i}(\delta_q^2) = (z_i - q_{i-1})^2 p(z_i) - (z_i - q_i)^2 p(z_i) = 0 \quad (i = 1, 2, \cdots, K) \tag{3-15}$$

$$\frac{\partial}{\partial q_i}(\delta_q^2) = -2\int_{z_i}^{z_{i+1}} (z - q_i) p(z) \, dz = 0 \quad (i = 1, 2, \cdots, K) \tag{3-16}$$

这样由式 (3-15) 和式 (3-16) 就得出了最佳量化条件

$$z_i = \frac{q_{i-1} + q_i}{2} \quad (i = 2, 3, \cdots, K) \tag{3-17}$$

$$q_i = \frac{\int_{z_i}^{z_{i+1}} z p(z)\,\mathrm{d}z}{\int_{z_i}^{z_{i+1}} p(z)\,\mathrm{d}z} \quad (i = 1, 2, \cdots, K) \tag{3-18}$$

若为均匀量化，即 $p(z) = 1$，则最佳量化条件为

$$z_i = \frac{q_{i-1} + q_i}{2} \quad (i = 2, 3, \cdots, K) \tag{3-19}$$

$$q_i = \frac{z_i + z_{i+1}}{2} \quad (i = 1, 3, \cdots, K) \tag{3-20}$$

练 习 题

1　设灰度分布满足

$$p(z) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-z^2}{2}\right)$$

$$z_1 = -\infty,\ z_{K+1} = \infty$$

取量化级数 $K = 16$，求最佳量化的判别层和输出的量化值。

2　将第一章练习 2 中存入计算机内的图像用 Java 语言获取像素，并存入数组中形成图像矩阵，改变 M，N 和 K 值，看看图像的效果如何。

第 4 章 图像增强

图像增强技术是数字图像处理的主要内容之一，它通过改变图像的灰度分布来改善图像的质量。通过图像增强可以突出一幅图像中的某些信息，削弱或去除某些不需要的信息，从而使图像更适合于人的视觉特性或机器识别系统。图像增强技术主要分为两部分，即空间域方法和频率域方法。空间域方法是在空间域内对图像样本灰度值直接进行运算处理，若 $f(x, y)$ 表示增强处理前的图像函数，$g(x, y)$ 表示增强处理后的图像函数，$h(x, y)$ 表示空间运算函数，则

$$g(x, y) \Leftrightarrow h(x, y) f(x, y) \tag{4-1}$$

式 (4-1) 可用图 4-1 表示。

频率域方法是在图像的某种变换域内对图像的变换值进行运算，如 $f(x, y)$ 表示增强处理前的图像函数，$h(x, y)$ 表示空间运算函数，增强处理后的图像函数 $g(x, y)$ 是由 $f(x, y)$ 和 $h(x, y)$ 的卷积得到的，即

图 4-1 空间域方法示意图

$$g(x, y) = h(x, y) * f(x, y) \tag{4-2}$$

根据卷积理论，在频域中有下面的变换关系

$$G(u, v) = H(u, v) F(u, v) \tag{4-3}$$

这里 G、H 和 F 分别表示 g、h 和 f 的傅立叶变换，$H(u, v)$ 称为传递函数。

在实际应用中，可根据需要先对图像函数进行傅立叶变换，并选定传递函数 $H(u, v)$，然后由式 (4-3) 计算出 $G(u, v)$，最后通过傅立叶逆变换得出增强处理后的图像函数 $g(u, v)$，即

$$g(x, y) = F^{-1}(H(u, v) F(u, v)) \tag{4-4}$$

上述过程可用图 4-2 表示。

图 4-2 频率域方法示意图

4.1 灰度变换

4.1.1 灰度变换

如果一幅图像灰度的对比度差，图像的质量就不好。为了改善图像灰度的对比度，可以对图像中样点的灰度进行刻度尺方面的改变——灰度变换。假设 $f(x, y)$ 和 $g(x, y)$ 分别

表示原始图像及增强处理后图像像素的灰度。这样使原始图像的像素灰度 $f(x, y)$ 转换成增强后图像对应像素的灰度 $g(x, y)$ 变换关系的一般表达式为

$$g(x, y) = T(f(x, y)) \tag{4-5}$$

下面讨论如何通过改变灰度分布来改变图像的效果。如果在原始图像 $f(x, y)$ 各个样点的灰度值上加一个常数 b，可得

$$g(x, y) = f(x, y) + b \tag{4-6}$$

若 $b>0$，则整个图像的亮度增加，如图 4-3 所示；若 $b<0$，则整个图像的亮度降低。

为了改变图像灰度的对比度，可对原始图像 $f(x, y)$ 各个样点的灰度值乘上一个常数 a，即

$$g(x, y) = af(x, y) \tag{4-7}$$

如果 $a>1$，对比度增强，如图 4-4 所示；如果 $a<1$，对比度减弱。

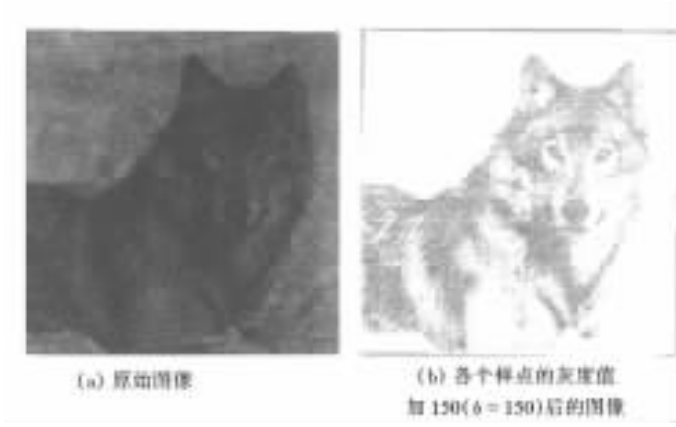


图 4-3　改变亮度的灰度变换

图 4-4　改变对比度的灰度变换

将式 (4-6) 和式 (4-7) 结合起来，可以得到线性灰度变换的一般表达式

$$g(x, y) = af(x, y) + b \tag{4-8}$$

假设要将原始图像样点的灰度取值范围 从 $[f_1, f_2]$ 变成 $[g_1, g_2]$，可通过下面的变换实现

$$g(x, y) = g_1 + \left(\frac{g_2 - g_1}{f_2 - f_1}\right)[f(x, y) - f_1] \tag{4-9}$$

图 4-5 示出了与方程 (4-9) 对应的变换直线。

这里有两种特殊的情况：一是如图 4-6 所示，即灰度值小于 f_1 的样点变换后灰度值变为 0，使与这些样点对应的图像区域变得更暗了；而灰度值大于 f_1 的样点变换后灰度值变为 255，与这些样点对应的图像区域变得更亮了，使图像的对比度十分明显，成为二值图像。第二种情况是 $a=-1$，$b=255$，如图 4-7 所示，形成了原始图像的负版。

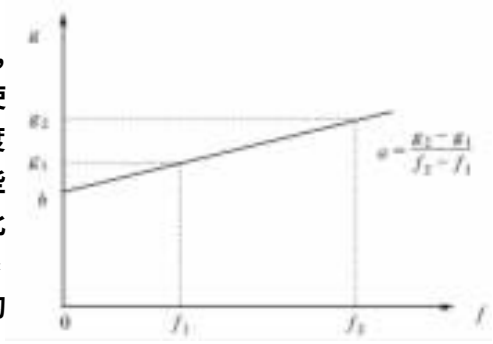


图 4-5 线性灰度变换 (选自 N.Efford)

在灰度变换中，可以根据需要选择变换的形式。若要详细勘察图像中某一区域的细节，应将此区域扩展；若对图像的某些区域不需要详细勘察，可将该区域压缩，但在压缩中会有一定的信息损失。下面举一个简单的例子。

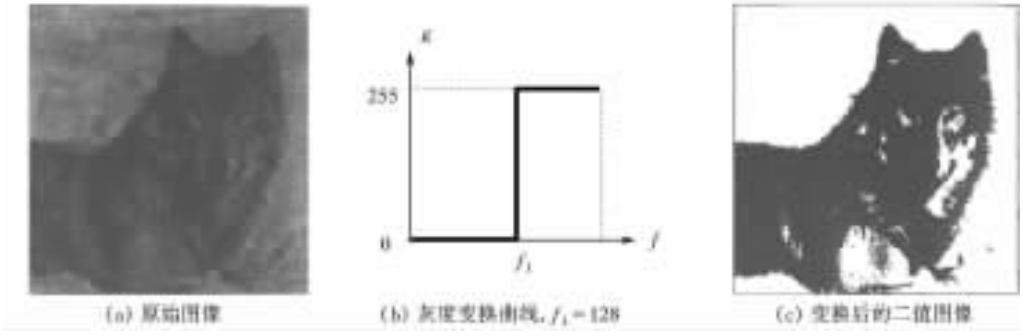


图 4-6 图像变换 (选自 N.Efford)

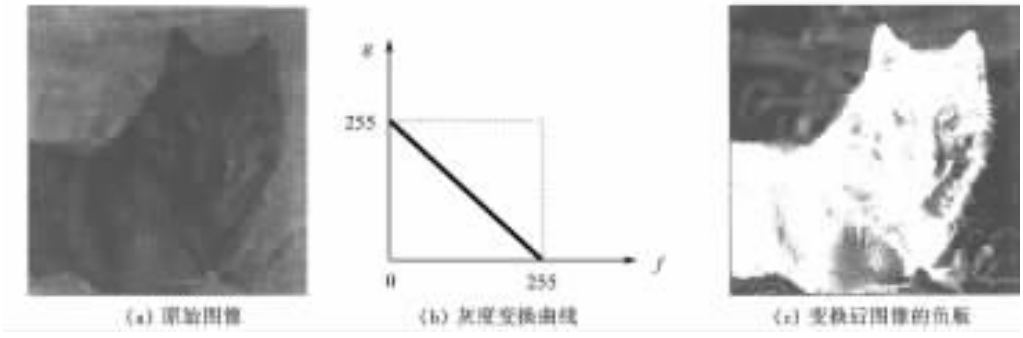


图 4-7 图像变换 (选自 N.Efford)

【例 4-1】 假设某图像的灰度分布范围是 [0，30]，进行如下的变换

$$g=\begin{cases}\dfrac{f}{2} & 0\leqslant f\leqslant 10\\ 2f-15 & 10<f<20\\ \dfrac{f}{2}+15 & 20\leqslant f\leqslant 30\end{cases}$$

在区间 [0，10] 和 [20，30] 内，由于乘一个常数 $\frac{1}{2}$，所以这两个区间被压缩；而在区间 (10，20) 内，由于乘一个常数 2，此区间被扩展，图像会更清楚。图 4-8 中示出了对原始图像扩展后的效果。

图 4-8　图像扩展变换 (选自何斌等)

对图像灰度的变换不只限于线性变换，也可进行非线性变换。在实际应用中，非线性变换也是很有用的。图 4-9 示出了一个非线性变换的曲线，在原始图像中有两个灰度区域 Δf_1 和 Δf_2。经过变换后，Δf_1 变成了 Δg_1，Δf_2 变成了 Δg_2。并 $\Delta g_1 > \Delta f_1$，表明与 Δf_1 对应的图像区域对比度增强了；而 $\Delta g_2 < \Delta f_2$，表明与 Δf_2 对应的图像区域的对比度减弱了。在实际应用中可以根据需要来改变变换曲线，如在图 4-10 中，为了看清汽车的牌号，通过非线性变换来改变牌号区域对比度，使牌号区域图像更加清晰。

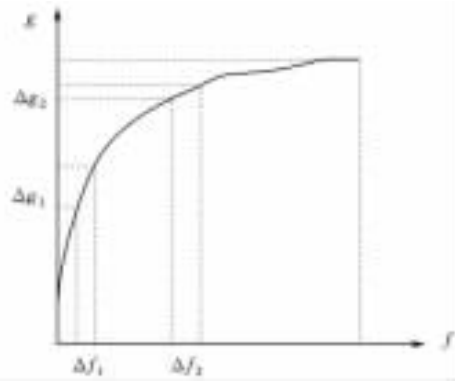


图 4-9　非线性灰度变换曲线 (选自 N.Efford)

图 4-10　图像的非线性变换 (选自 N.Efford)

4.1.2　Java 语言实现线性灰度变换

根据式 (4-8)，取 $a = 1.1$，$b = 30$，用 Java Application 实现对图像灰度进行的线性变换如程序清单 4-1 所示，图 4-11 示出了程序运行结果的例子。

程序清单 4-1 LineGrey.java **源代码**

```
//LineGrey.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
//import javax.swing.*;

public class LineGrey extends Frame {
Image im, tmp;
int i, iw, ih;
int[] pixels;
boolean flag = false;

//ImagePixel 的构造方法
public LineGrey(){
    Panel pdown;
    Button load, run, quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    pdown = new Panel();
    pdown.setBackground(Color.lightGray);

    load = new Button("装载图像");
    run = new Button("线性扩展");
    quit = new Button("退出");

    this.add(pdown, BorderLayout.SOUTH);

    pdown.add(load);
    pdown.add(run);
    pdown.add(quit);

    load.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
```

```
            jLoad_ActionPerformed(e);
        }
    });

    run.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jRun_ActionPerformed(e);
        }
    });

    quit.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jQuit_ActionPerformed(e);
        }
    });
}

    public void jLoad_ActionPerformed(ActionEvent e){
        //利用 MediaTracker 跟踪图像的加载
        MediaTracker tracker = new MediaTracker(this);
        im=Toolkit.getDefaultToolkit().getImage("filename.jpg");
        tracker.addImage(im,0);

        //等待图像的完全加载
        try{
        tracker.waitForID(0);
        }catch(InterruptedException e2){ e2.printStackTrace();}

        //获取图像的宽度 iw 和高度 ih
        iw=im.getWidth(this);
        ih=im.getHeight(this);
        pixels=new int[iw*ih];

        try{
        PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }
```

```
        //获得图像的 RGB 值和 Alpha 值
        ColorModel cm = ColorModel.getRGBdefault();
        for(i = 0;i < iw * ih;i ++ )
        {
            int alpha = cm.getAlpha(pixels[i]);
            int red = cm.getRed(pixels[i]);
            int green = cm.getGreen(pixels[i]);
            int blue = cm.getBlue(pixels[i]);
            pixels[i] = alpha << 24 | red << 16 | green << 8 | blue;
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        flag = true;
        repaint();
    }

    public void jRun_ActionPerformed(ActionEvent e){

        try{
        PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //对图像进行进行线性拉伸, Alpha 值保持不变
        ColorModel cm = ColorModel.getRGBdefault();
        for(i = 0;i < iw * ih;i ++ )
        {
            int alpha = cm.getAlpha(pixels[i]);
            int red = cm.getRed(pixels[i]);
            int green = cm.getGreen(pixels[i]);
            int blue = cm.getBlue(pixels[i]);

            //增加了图像的亮度
            red = (int)(1.1 * red + 30);
            green = (int)(1.1 * green + 30);
            blue = (int)(1.1 * blue + 30);
```

```
                if(red>=255)
                {
                    red=255;
                }
                    if(green>=255)
                {
                    green=255;
                }
                    if(blue>=255)
                {
                    blue=255;
                }
                pixels[i]=alpha<<24|red<<16|green<<8|blue;
            }

            //将数组中的像素产生一个图像
            ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
            tmp=createImage(ip);
            flag=true;
            repaint();
        }
        public void jQuit_ActionPerformed(ActionEvent e){
            System.exit(0);
        }

//调用 paint()方法,显示图像信息。
public void paint(Graphics g){
    if(flag){
        g.drawImage(tmp,10,20,this);
    }else { }
}

//定义 main 方法,设置窗口的大小,显示窗口
public static void main(String[] args) {
    LineGrey lg = new LineGrey();
    lg.setSize(300,300);
    lg.show();
}
}
```

图 4-11 Java Application 对图像进行线性变换

4.2 直方图均匀化处理

直方图均匀化处理是图像处理中最常用的方法之一，它借助灰度直方图来改善图像的灰度分布。

4.2.1 灰度直方图

灰度直方图是反映一幅图像中的灰度级与出现这种灰度的概率之间关系的图形，其横坐标为灰度级 f，纵坐标为图像中出现某个灰度级的概率 $p(f)$，如图 4-12 所示。

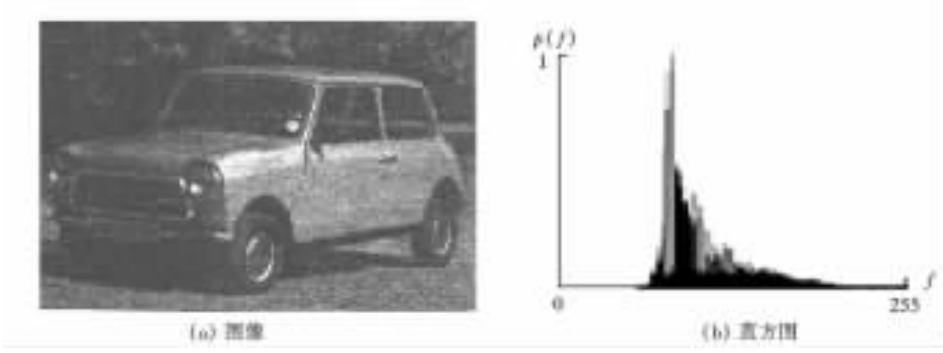


图 4-12 图像和它的直方图

图 4-13 是一个只有两个灰度级 f_1 和 f_2 的直方图，并且它们的概率密度 $p(f)$ 是相同的。与这个直方图对应的图像可以有多种形式，如图 4-14 所示。

可见，一个直方图可能对应几幅图像，但一幅图像只有一个直方图。在数字图像处理

图 4-13　直方图

图 4-14　与图 4-13 所示直方图对应的四幅图像

中，直方图是最简单、最有用的工具。它描述了一幅图像中的信息，如根据直方图可预测一幅图像的亮暗程度，根据直方图可以确定检测出图像背景和目标的灰度阈值等。

4.2.2　直方图修正技术

直方图修正技术是图像增强最常用的方法之一。那么，什么是直方图修正技术呢？下面通过两个例子加以说明。图 4-15 (a) 是一幅过曝光的照片，它的直方图如图 4-15 (b) 所示，可见其灰度级都集中在高亮度区域内 (即 f 值大的区域内)，由于各样点间的灰度级差小，所以图像不清楚。

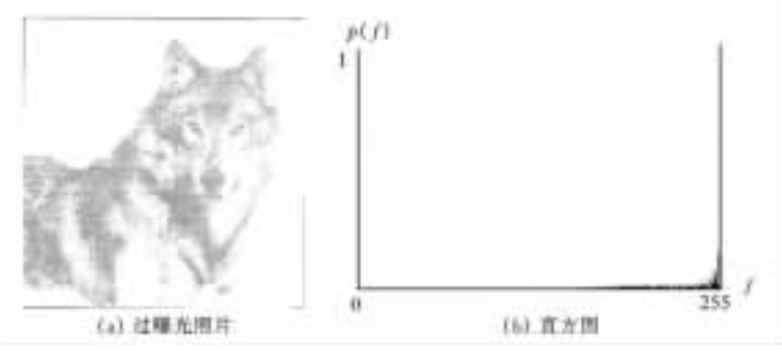


图 4-15　过曝光照片的直方图

图 4-16 (a) 是一幅曝光不足的照片，它的直方图如图 4-16 (b) 所示，可见其灰度级都集中在低亮度区域内 (即 f 值较小的区域内)，由于各样点间的灰度级差小，所以图像也不清楚。

可见，通过直方图可以粗略地评价图像的质量，如果对图 4-15 和图 4-16 的直方图作如下的处理，即将图 4-15 (b) 和图 4-16 (b) 的直方图变成图 4-17 所示的类似直方图形式，这样样点间的灰度级差变大，图像的清晰度将会增加。这种根据直方图来改变灰度级分布的方法就是直方图修正技术。

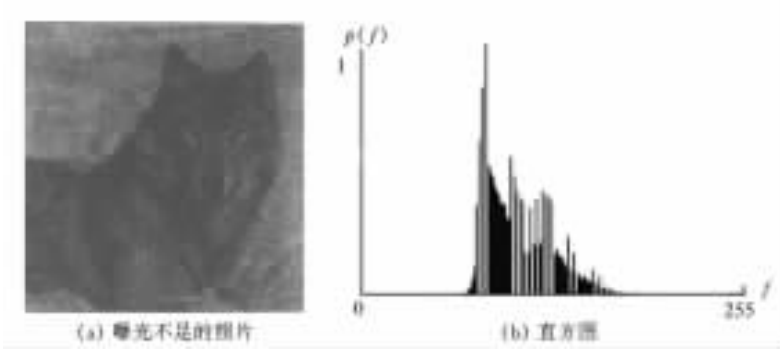


图 4-16 曝光不足照片的直方图

直方图修正技术在实际应用中经常使用，如在医学方面为了改善操作 X 射线人员的工作条件，可以使用低强度的 X 射线曝光，以减少 X 射线对工作人员的损伤，但这样获得的 X 光片灰度集中在较暗的区域，许多图像细节无法看清楚，这时可以使用直方图修正技术对该图像进行修正，使灰度级分布在人眼合适的亮度区域，这样就可以使 X 光片中的细节清楚可见。

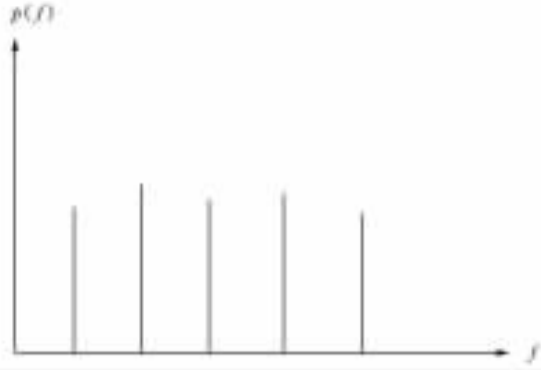


图 4-17 直方图

直方图修正技术是通过某种变换来改变图像灰度的概率分布。假设 f 表示增强处理前图像样点的灰度，为了研究方便，将灰度级归一化，即

$$0 \leqslant f \leqslant 1 \tag{4-10}$$

式中 $f=0$ 表示黑色，$f=1$ 表示白色。

对于在区间 [0，1] 内的任意 f 值满足变换关系

$$g = T(f) \tag{4-11}$$

式中 g 表示相对应的增强处理后图像中样点的灰度级。

这里要求变换函数满足下列条件：

(1) 在 $0 \leqslant f \leqslant 1$ 区间内，$T(f)$ 单值单调增加。

(2) 对应于 $0 \leqslant f \leqslant 1$，有关系式 $0 \leqslant T(f) \leqslant 1$。

条件 (1) 是使变换后的灰度值保持从黑到白的次序；条件 (2) 保证变换后的样点灰度级仍在允许的范围内。上述可用图 4-18 表示。

从 g 到 f 的逆变换为

$$f = T^{-1}(g) \tag{4-12}$$

要求 $T^{-1}(g)$ 也满足条件 (1) 和 (2)。

假设 $p_f(f)$ 表示原始图像灰度级 f 出现的概率，$p_g(g)$ 表示增强处理后图像灰度级 g 出现的概率。如果 $p_f(f)$ 和 $g = T(f)$ 为已知，且 $T^{-1}(g)$ 满足条件 (1)，由概率论知，灰度变换的概率满足下面的关系式

$$p_g(g) = \left[p_f(f)\frac{\mathrm{d}f}{\mathrm{d}g}\right]_{f=T^{-1}(g)} \tag{4-13}$$

可见，经处理后图像灰度级的概率函数 $p_g(g)$ 是由原始图像灰度级的概率 $p_f(f)$ 及变换函数 $T(f)$ 决定的，而 $T(f)$ 是根据需要人为选定的，所以选择合适的 $T(f)$ 对改变图像的灰度分布、改善图像的质量是很有意义的。

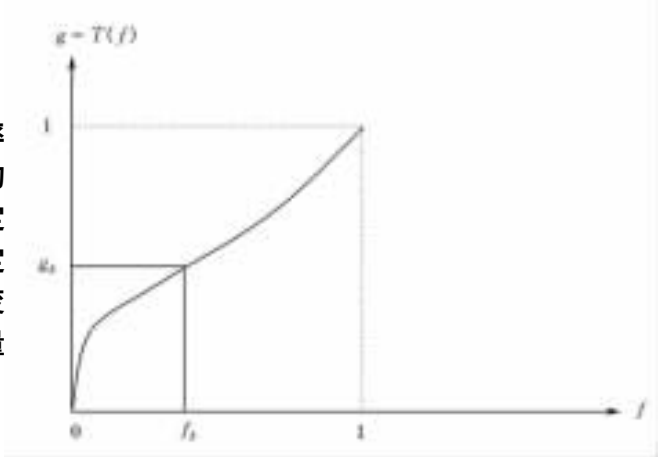


图 4-18　灰度变换函数

4.2.3　直方图均匀化处理

直方图均匀化处理使经过变换函数 $T(f)$ 变换后的图像的灰度分布是均匀的。为了达到均匀化处理的目的，引入变换函数的形式如下

$$g = T(f) = \int_0^f p_f(\omega)\,\mathrm{d}\omega \quad (0 \leqslant f \leqslant 1) \tag{4-14}$$

式中　ω——积分变量。

根据式 (4-13)，首先将式 (4-14) 中的 g 对 f 求导数，得

$$\frac{\mathrm{d}g}{\mathrm{d}f} = p_f(f) \tag{4-15}$$

则

$$\frac{\mathrm{d}f}{\mathrm{d}g} = \frac{1}{p_f(f)} \tag{4-16}$$

然后将式 (4-16) 代入式 (4-13) 中，得

$$p_g(g) = \left[p_f(f)\frac{1}{p_f(f)}\right]_{f=T^{-1}(g)} = 1 \tag{4-17}$$

可见，若变换函数选式 (4-14) 的形式，变换后的灰度变量 g 在定义域内的概率分布是均匀的。所以这样的变换函数可以产生一幅灰度级分布具有均匀概率的图像。

上述是针对连续变量的情况，而对于一幅数字图像必须引入离散形式的公式。假设灰度值是离散的，某一灰度出现的概率为

$$p_f(f_i) = \frac{n_i}{n} \quad (0 \leqslant f_i \leqslant 1,\ i = 0, 1, 2, \cdots, K-1) \tag{4-18}$$

这里 K 是量化区间数，$p_f(f_i)$ 是第 i 个灰度级出现的概率，n_i 是在图像中具有灰度 f_i 的样点数目，n 是图像中的样点总数目。由 $p_f(f_i)$ 和 f_i 组成的图形就是直方图，获得均匀分布直方图的技术就是直方图均匀化处理。

根据式 (4-14) 和式 (4-18)，变换函数的离散形式为

$$g_i = T(f_i) = \sum_{j=0}^{i}\frac{n_j}{n} = \sum_{j=0}^{i}p_f(f_i) \quad (0 \leqslant f_i \leqslant 1,\ i = 0, 1, 2, \cdots, K-1) \tag{4-19}$$

并

$$f_i = T^{-1}(g_i) \quad (0 \leqslant g_i \leqslant 1) \tag{4-20}$$

这里 $T(f_i)$ 和 $T^{-1}(g_i)$ 均满足前面提到的条件(1)和(2)。

【例 4-2】 假设某一图像的取样点为 64×64，灰度量化区间为 8 个，如表 4-1 所示，试对其进行直方图的均匀化处理。

表 4-1 灰度分布

i	f_i	n_i	$p_f(f_i)$
0	0	790	0.19
1	$\frac{1}{7}$	1023	0.25
2	$\frac{2}{7}$	850	0.21
3	$\frac{3}{7}$	656	0.16
4	$\frac{4}{7}$	329	0.08
5	$\frac{5}{7}$	245	0.06
6	$\frac{6}{7}$	122	0.03
7	1	81	0.02

这一图像的直方图如图 4-19(a)所示。根据式(4-19)，不同值的变换函数为

当 $i=0$ 时，

$$g_0 = T(f_0) = \sum_{j=0}^{0} p_f(f_j) = p_f(f_0) = 0.19$$

当 $i=1$ 时，

$$g_1 = T(f_1) = \sum_{j=0}^{1} p_f(f_j) = p_f(f_0) + p_f(f_1) = 0.44$$

依此类推，得

$$g_2 = 0.65 \qquad g_3 = 0.81$$
$$g_4 = 0.89 \qquad g_3 = 0.95$$
$$g_6 = 0.98 \qquad g_7 = 1.00$$

因为对原图像只取了 8 个等间隔的灰度级，变换后的 g 值也只能选择最靠近的一个灰度级的值，因此对上述计算值加以修正，

$$g_0 \approx \frac{1}{7} \qquad g_1 \approx \frac{3}{7}$$
$$g_2 \approx \frac{5}{7} \qquad g_3 \approx \frac{6}{7}$$
$$g_4 \approx \frac{6}{7} \qquad g_5 \approx 1$$
$$g_6 \approx 1 \qquad g_7 \approx 1$$

从上述数值可见，新图像将只有 5 个独立的灰度级，对此重新定义如下：

$$g_0 \approx \frac{1}{7} \qquad g_1 \approx \frac{3}{7}$$
$$g_2 \approx \frac{5}{7} \qquad g_3 \approx \frac{6}{7}$$

$$g_4 \approx 1$$

根据式 (4-19) 计算出新图像的灰度概率分布，如表 4-2 所示。

表 4-2　新的灰度分布

i	g_i	n_i	$p_g(g_i)$
0	$\frac{1}{7}$	790	0.19
1	$\frac{3}{7}$	1023	0.25
2	$\frac{5}{7}$	850	0.21
3	$\frac{6}{7}$	656 + 329	0.24
4	1	245 + 122 + 81	0.11

根据表 (4-2)，新的灰度直方图如图 4-19 (b) 所示。对比图 4-19 (a) 和图 4-19 (b) 可见，处理后的直方图分布更加均匀了。图 4-20 示出了一个直方图均匀化处理的实际例子，图 4-20 (a) 为原始图像，图 4-20 (b) 是图 4-20 (a) 的直方图，图 4-20 (c) 是均匀化处理后的图像，图 4-20 (d) 是图 4-20 (c) 的直方图，可见处理后的图像清晰度增加了。

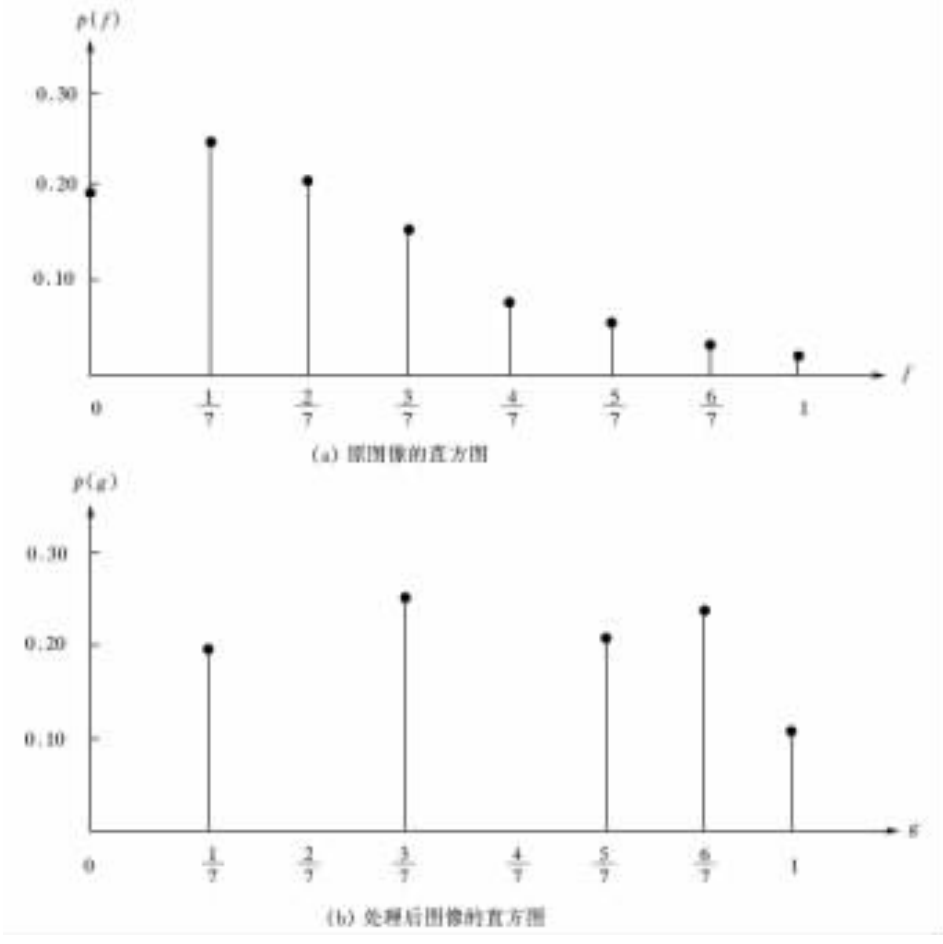


图 4-19　直方图均匀化处理

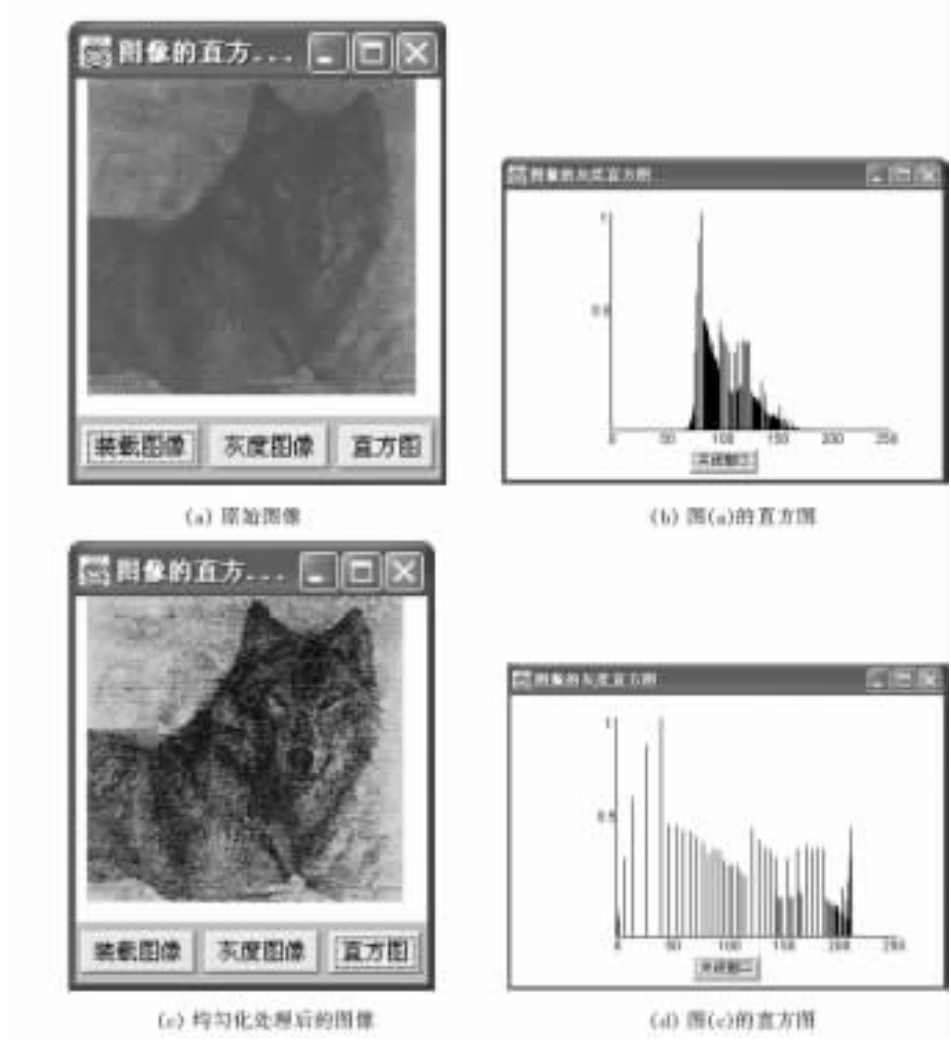


图 4-20　直方图均匀化处理

程序清单 4-2 **给出了图** 4-20 **所示进行直方图均匀化处理的** Java Application **程序。**

程序清单 4-2　　Histogram.java **源代码**

```
//Histogram.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class Histogram extends Frame {
Image im, tmp;
```

```
int i,iw,ih;
int[] pixels;
boolean flagLoad=false;
boolean flagGrey=false;

//ImagePixel 的构造方法
public Histogram(){
    this.setTitle("图像的直方图均匀化");

    Panel pdown;
    Button load,grey,hist,run,save,quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    pdown = new Panel();
    pdown.setBackground(Color.lightGray);

    load=new Button("装载图像");
    grey=new Button("灰度图像");
    hist=new Button("直方图");
    run=new Button("均匀化");
    save=new Button("保存");
    quit=new Button("退出");

    this.add(pdown,BorderLayout.SOUTH);

    pdown.add(load);
    pdown.add(grey);
    pdown.add(hist);
    pdown.add(run);
    pdown.add(save);
    pdown.add(quit);

    load.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jLoad_ActionPerformed(e);
```

```
        }
    });

    hist.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jHist_ActionPerformed(e);
        }
    });

    grey.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jGrey_ActionPerformed(e);
        }
    });

    run.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jRun_ActionPerformed(e);
        }
    });

    quit.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jQuit_ActionPerformed(e);
        }
    });
    save.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jSave_ActionPerformed(e);
        }
    });
}

public void jLoad_ActionPerformed(ActionEvent e){
    //利用 MediaTracker 跟踪图像的加载
    MediaTracker tracker = new MediaTracker(this);
    im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
    tracker.addImage(im,0);

    //等待图像的完全加载
```

```
        try{
        tracker.waitForID(0);
            }catch(InterruptedException e2){ e2.printStackTrace();}

            //获取图像的宽度 iw 和高度 ih
        iw=im.getWidth(this);
        ih=im.getHeight(this);
        pixels=new int[iw*ih];

        try{
        PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //获得图像的 RGB 值和 Alpha 值
        ColorModel cm=ColorModel.getRGBdefault();
        for(i=0;i<iw*ih;i++)
        {
            int alpha=cm.getAlpha(pixels[i]);
            int red=cm.getRed(pixels[i]);
            int green=cm.getGreen(pixels[i]);
            int blue=cm.getBlue(pixels[i]);
            pixels[i]=alpha<<24|red<<16|green<<8|blue;
        }

        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        flagLoad=true;
        repaint();
}

public void jGrey_ActionPerformed(ActionEvent e){
        //必须保证图像进行了加载,然后才可以进行灰度化
        if(flagLoad){
        try{
        PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
        pg.grabPixels();
```

```
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        ColorModel cm=ColorModel.getRGBdefault();
        for(i=0;i<ih;i++)
        {
            for(int j=0;j<iw;j++)
            {
                int alpha=cm.getAlpha(pixels[i*iw+j]);
                int red=cm.getRed(pixels[i*iw+j]);
                int green=cm.getGreen(pixels[i*iw+j]);
                int blue=cm.getBlue(pixels[i*iw+j]);

                int grey=(int)(0.3*red+0.59*green+0.11*blue);

                pixels[i*iw+j]=alpha<<24|grey<<16|grey<<8|grey;
            }
        }
        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        flagGrey=true;
        repaint();

    }else{
        JOptionPane.showMessageDialog(null,"请先打开一幅图片!",
                    "Alert",JOptionPane.WARNING_MESSAGE);
        }
    }

    public void jHist_ActionPerformed(ActionEvent e){

        //显示图像的直方图
        Hist h=new Hist();

        //传送数据
        h.getData(pixels,iw,ih);

        h.setSize(480,300);
```

```
        h.show();
    }

    //对图像进行直方图均匀化处理
    public void jRun_ActionPerformed(ActionEvent e){
        //图像已经加载,并且变成了灰度图像
        if(flagLoad && flagGrey){

        //获取图像的直方图
        int [] histogram=new int [256];
        ColorModel cm=ColorModel.getRGBdefault();
        for(i=0;i<ih;i++)
        {
            for(int j=0;j<iw;j++)
            {
                int grey=pixels[i*iw+j]&0xff;
                histogram[grey]++;
            }
        }
        //直方图均匀化处理
        double a=(double)255/(iw*ih);
        double [] c=new double [256];
        c[0]=(a*histogram[0]);
        for(i=1;i<256;i++)
        {
            c[i]=c[i-1]+(int)(a*histogram[i]);
        }
        for(i=0;i<ih;i++)
        {
            for(int j=0;j<iw;j++)
            {
                int alpha=cm.getAlpha(pixels[i*iw+j]);
                int grey=pixels[i*iw+j]&0x0000ff;
                int hist=(int)c[grey];

                pixels[i*iw+j]=alpha<<24|hist<<16|hist<<8|hist;

            }
        }
        //将数组中的像素产生一个图像
```

```
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        repaint();
    }else{
        JOptionPane.showMessageDialog(null, "请先打开一幅图片,或者把该图片变为灰度
图像!",
                        "Alert", JOptionPane.WARNING_MESSAGE);
        }
    }
    public void jSave_ActionPerformed(ActionEvent e){
        //保存该图像,还不能实现 .
        if(flagLoad){
        ImageIcon ii = new ImageIcon("copyColor.jpg");
        ii.setImage(tmp);
    }else{
        JOptionPane.showMessageDialog(null, "请先打开一幅图片!",
                        "Alert", JOptionPane.WARNING_MESSAGE);
        }
    }

    public void jQuit_ActionPerformed(ActionEvent e){
        System.exit(0);
    }
    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flagLoad){
            g.drawImage(tmp, 10, 20, this);
        }else { }
    }

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[] args) {
        Histogram hist = new Histogram();
        hist.setSize(400, 300);
        hist.show();
    }
    }
```

4.3　平滑化处理

图像平滑化处理的目的是减少噪声，这种处理方法可以分为两类：一是在空间域内处理，二是在频率域内处理。

4.3.1　邻域平均法

邻域平均法是一种简单的在空间域内进行处理的方法，它易于实现，效果也较好。邻域平均法的基本思想是：由于噪声使图像上一些样点的灰度造成突变，那么就可以以这样的样点为中心取一个邻域，用邻域内其他样点的灰度平均值来代替要处理的样点的灰度，其结果对亮度突变的点产生了“平滑”的效果。假设图 4-21 是在某一图像中取的一个邻域，其中 e 点认为是噪声点，那么就以 e 点为中心取了这样一个邻域，在处理后图像中 e 点的灰度值为

$$e' = \frac{1}{8}(a + b + c + d + f + g + h + i) \qquad (4\text{-}21)$$

式中　a，b，c，d，f，g，h 和 i 分别为邻域内各个样点的灰度值。

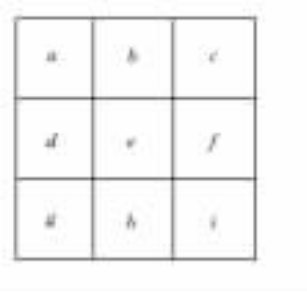


图 4-21　以 e 为中心的一个邻域

对上述加以推广，若已知一幅 $N \times N$ 的图像 $f(n_1, n_2)$，以图像中某一样点 (m_1, m_2) 为中心取一邻域进行平滑化处理，处理后该样点 (m_1, m_2) 的新灰度值为

$$g(m_1, m_2) = \frac{1}{M} \sum_{n_1, n_2 \in S} f(n_1, n_2) \qquad (4\text{-}22)$$

式中　$g(m_1, m_2)$ —— (m_1, m_2) 点平滑后的灰度值；

S —— (m_1, m_2) 点的邻域中样点的集合，但不包括 (m_1, m_2) 点；

M ——邻域内样点的总数，但不包括 (m_1, m_2) 点；

$f(n_1, n_2)$ ——邻域内其他样点的灰度值。

图 4-22 和图 4-23 示出了两种邻域的取法。在图 4-22 中是以 R 点为中心，以样点间距 ΔX 为半径做圆，则 R 点的邻域就是圆内和边界上的点的集合。在图 4-23 中是以 R 点为中心，以样点间距 $\sqrt{2}\Delta X$ 为半径做圆，则 R 点的邻域就是圆内和边界上的点的集合。

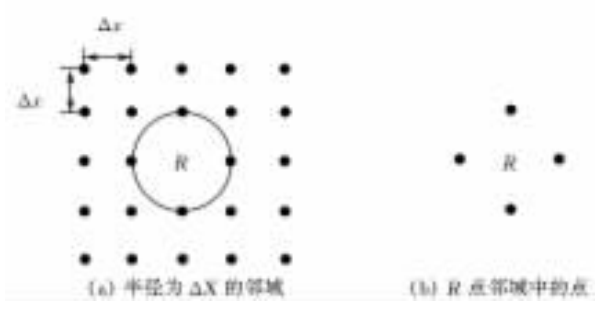


图 4-22　半径为 Δx 的邻域

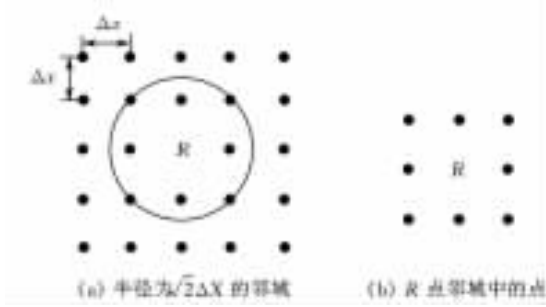


图 4-23 半径为$\sqrt{2}\Delta x$ 的邻域

图 4-24 示出了经邻域平均法处理后图像的效果，可见，这样处理后使图像中的噪声得到了平滑，但图像中的某些细节 (灰度突变区域) 也变模糊了。为了弥补这一缺点，通常选择一个或几个阈值来减少图像信息在处理中的损失，如可以设一个非负的阈值，这样式 (4-22) 变为

$$g(m_1, m_2) = \begin{cases} \dfrac{1}{M}\sum\limits_{n_1, n_2 \in S} f(n_1, n_2) & \text{当} \left| f(m_1, m_2) - \dfrac{1}{M}\sum\limits_{n_1, n_2 \in S} f(n_1, n_2) \right| > T \text{ 时} \\ f(m_1, m_2) & \text{其他} \end{cases} \tag{4-23}$$

从式 (4-23) 可以看出，当一些样点和它的邻域内的点的灰度的平均值的差不超过阈值 T 时，仍然保留其原始的灰度值不变；如果超过阈值 T，就用它们的平均值来代替该点的灰度值，这样就可以大大减少模糊的程度了，当然也可以取其他形式的阈值。

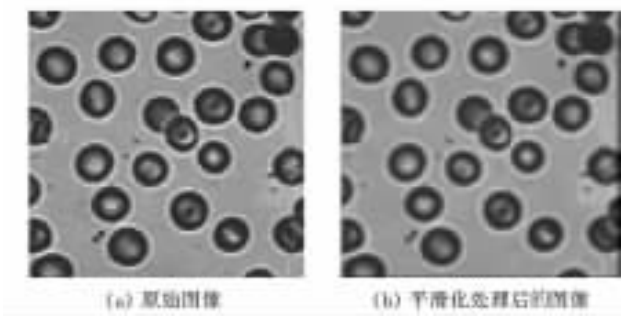
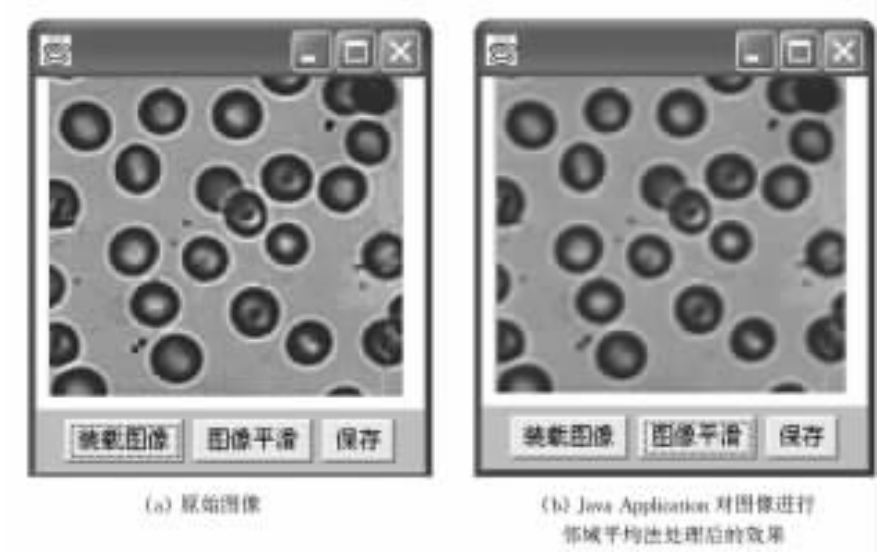


图 4-24 平滑化处理

在程序清单 4-3 中选择了如下形式的邻域

$$\begin{Bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{Bmatrix}$$

对图像进行邻域平均法的平滑化处理，程序运行结果如图 4-25 所示。



(a) 原始图像　(b) Java Application 对图像进行邻域平均法处理后的效果

图 4-25　平滑化处理

程序清单 4-3　　SmoothImage.java **源代码**

```
//SmoothImage.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class SmoothImage extends Frame {
Image im, tmp;
int i, iw, ih;
int[] pixels;
boolean flag = false;

//ImagePixel 的构造方法
public SmoothImage(){
    Panel pdown;
    Button load, run, save, quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });
```

```
        pdown = new Panel();
        pdown.setBackground(Color.lightGray);

        load=new Button("装载图像");
        run = new Button("图像平滑");
        save=new Button("保存");
        quit=new Button("退出");

        this.add(pdown,BorderLayout.SOUTH);

        pdown.add(load);
        pdown.add(run);
        pdown.add(save);
        pdown.add(quit);

        load.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jLoad_ActionPerformed(e);
            }
        });

        run.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jRun_ActionPerformed(e);
            }
        });

        quit.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });
        save.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jSave_ActionPerformed(e);
            }
        });
    }
```

```
public void jLoad_ActionPerformed(ActionEvent e){
    //利用 MediaTracker 跟踪图像的加载
    MediaTracker tracker = new MediaTracker(this);
    im=Toolkit.getDefaultToolkit().getImage("Color.jpg");
    tracker.addImage(im,0);

    //等待图像的完全加载
    try{
    tracker.waitForID(0);
        }catch(InterruptedException e2){ e2.printStackTrace();}

        //获取图像的宽度 iw 和高度 ih
    iw=im.getWidth(this);
    ih=im.getHeight(this);
    pixels=new int[iw*ih];

    try{
    PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
    pg.grabPixels();
    }catch (InterruptedException e3) {
        e3.printStackTrace();
    }

    //获得图像的 RGB 值和 Alpha 值
    ColorModel cm=ColorModel.getRGBdefault();
    for(i=0;i<iw*ih;i++)
    {
        int alpha=cm.getAlpha(pixels[i]);
        int red=cm.getRed(pixels[i]);
        int green=cm.getGreen(pixels[i]);
        int blue=cm.getBlue(pixels[i]);
        pixels[i]=alpha<<24|red<<16|green<<8|blue;
    }

    //将数组中的像素产生一个图像
    ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
    tmp=createImage(ip);
    flag=true;
    repaint();
}
```

```
public void jRun_ActionPerformed(ActionEvent e){

    try{
    PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
    pg.grabPixels();
    }catch (InterruptedException e3) {
        e3.printStackTrace();
    }

    //对图像进行平滑化处理,Alpha 值保持不变
    ColorModel cm = ColorModel.getRGBdefault();
    for(i = 1;i < ih - 1;i++)
    {
        for(int j = 1;j < iw - 1;j++)
        {
            int alpha = cm.getAlpha(pixels[i * iw + j]);
            int red = cm.getRed(pixels[i * iw + j]);
            int green = cm.getGreen(pixels[i * iw + j]);
            int blue = cm.getBlue(pixels[i * iw + j]);

            //对图像进行平滑
            int red1 = cm.getRed(pixels[(i - 1) * iw + j - 1]);
            int red2 = cm.getRed(pixels[(i - 1) * iw + j]);
            int red3 = cm.getRed(pixels[(i - 1) * iw + j + 1]);
            int red4 = cm.getRed(pixels[i * iw + j - 1]);
            int red6 = cm.getRed(pixels[i * iw + j + 1]);
            int red7 = cm.getRed(pixels[(i + 1) * iw + j - 1]);
            int red8 = cm.getRed(pixels[(i + 1) * iw + j]);
            int red9 = cm.getRed(pixels[(i + 1) * iw + j + 1]);
            int averageRed = (red1 + red2 + red3 + red4 + red6 + red7 + red8 + red9)/8;

            int green1 = cm.getGreen(pixels[(i - 1) * iw + j - 1]);
            int green2 = cm.getGreen(pixels[(i - 1) * iw + j]);
            int green3 = cm.getGreen(pixels[(i - 1) * iw + j + 1]);
            int green4 = cm.getGreen(pixels[i * iw + j - 1]);
            int green6 = cm.getGreen(pixels[i * iw + j + 1]);
            int green7 = cm.getGreen(pixels[(i + 1) * iw + j - 1]);
            int green8 = cm.getGreen(pixels[(i + 1) * iw + j]);
            int green9 = cm.getGreen(pixels[(i + 1) * iw + j + 1]);
```

```
                int averageGreen = (green1 + green2 + green3 + green4 + green6 + green7 +
green8 + green9)/8;

                int blue1 = cm.getBlue(pixels[(i - 1) * iw + j - 1]);
                int blue2 = cm.getBlue(pixels[(i - 1) * iw + j]);
                int blue3 = cm.getBlue(pixels[(i - 1) * iw + j + 1]);
                int blue4 = cm.getBlue(pixels[i * iw + j - 1]);
                int blue6 = cm.getBlue(pixels[i * iw + j + 1]);
                int blue7 = cm.getBlue(pixels[(i + 1) * iw + j - 1]);
                int blue8 = cm.getBlue(pixels[(i + 1) * iw + j]);
                int blue9 = cm.getBlue(pixels[(i + 1) * iw + j + 1]);
                 int averageBlue = (blue1 + blue2 + blue3 + blue4 + blue6 + blue7 + blue8 +
blue9)/8;

                pixels[i * iw + j] = alpha << 24 | averageRed << 16 | averageGreen << 8 | av-
erageBlue;
            }
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        flag = true;
        repaint();
    }
    public void jSave_ActionPerformed(ActionEvent e){
        //保存该图像,还不能实现 .
        ImageIcon ii = new ImageIcon("copyColor.jpg");
        ii.setImage(tmp);
    }

    public void jQuit_ActionPerformed(ActionEvent e){
            System.exit(0);
    }

    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flag){
            g.drawImage(tmp, 10, 20, this);
        }else { }
```

```
}
//定义 main 方法,设置窗口的大小,显示窗口
public static void main(String[] args) {
    SmoothImage si = new SmoothImage();
    si.setSize(300,300);
    si.show();
}
}
```

4.3.2 低通滤波法

所谓低通滤波，是在频率域内对图像进行的平滑化处理。在一幅图像中，图像的边缘和灰度突变的区域(如噪声区域)位于频域的高频区，而图像的背景位于低频区。这样就可以用滤波的方法滤去高频分量，保留低频分量，从而达到去掉噪声、使图像得到平滑处理，所以这种方法叫低通滤波法。

根据式(4-3)有

$$G(u,v) = H(u,v)F(u,v) \tag{4-24}$$

式中 $F(u,v)$——处理前图像函数的傅立叶变换；

$H(u,v)$——传递函数，它的作用是使 $F(u,v)$ 的高频分量得到衰减，而低通分量可以通过，即具有低通滤波的特性；

$G(u,v)$——平滑处理后图像函数的傅立叶变换，再通过傅立叶逆变换就可以得到平滑处理后的图像 $g(x,y)$。

低通滤波平滑化处理的过程可用图 4-26 表示。

图 4-26 低通滤波示意图

综上所述可见，低通滤波的效果主要取决于传递函数 $H(u,v)$，常用的传递函数有以下几种形式。

(1) 理想低通滤波。理想低通滤波的传递函数的形式如下

$$H(u,v) = \begin{cases} 1 & \text{当 } D(u,v) \leqslant D_0 \\ 0 & \text{当 } D(u,v) > D_0 \end{cases} \tag{4-25}$$

式中 D_0——一个非负的量，称为截止频率；

$D(u,v)$——频率平面上的点 (u,v) 到原点的距离，即

$$D(u,v) = \sqrt{u^2 + v^2} \tag{4-26}$$

在图 4-27 中示出了传递函数 $H(u,v)$ 的三维图形和二维截面图形，可见在以 D_0 为半径的圆内的所有频率对应的传递函数 $H(u,v) = 1$，而在截止频率之外的传递函数 $H(u,v) = 0$。根据式(4-24)可知，选择这样的传递函数可以使高于截止频率 D_0 的图像信息全部滤掉；而低于截止频率 D_0 的图像信息全部通过，起到了低通滤波的作用。

(2) Butterworth 低通滤波。n 阶 Butterworth 低通滤波的传递函数满足下面的关系

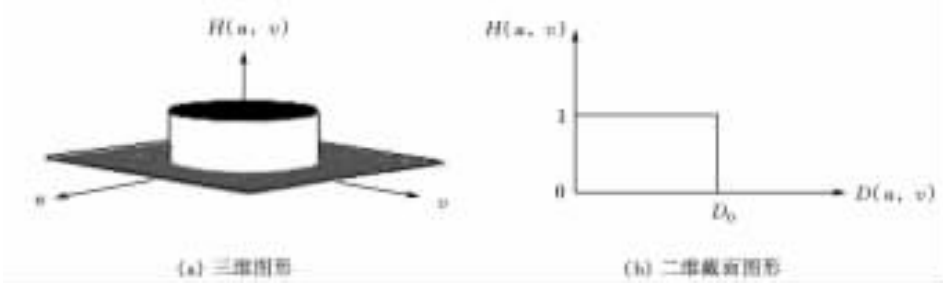


图 4-27　理想低通滤波的传递函数

$$H(u, v) = \frac{1}{1 + \left[\dfrac{D(u, v)}{D_0}\right]^{2n}} \tag{4-27}$$

式中　$D(u, v)$——频率平面上的点 (u, v) 到原点的距离，即

$$D(u, v) = \sqrt{u^2 + v^2} \tag{4-28}$$

式中　D_0——截止频率，把 $H(u, v)$ 下降到原来值 $\frac{1}{2}$ 的 $D(u, v)$ 值定为截止频点 D_0。

在图 4-28 示出了 $n = 1$ 时的传递函数 $H(u, v)$ 的三维图形和二维截面图形。与理想低通滤波相比，Butterworth 低通滤波的传递函数的衰减缓慢，这就使得在滤波过程中部分高频信息被保留，所以滤波处理后图像的模糊程度要小于理想低通滤波。

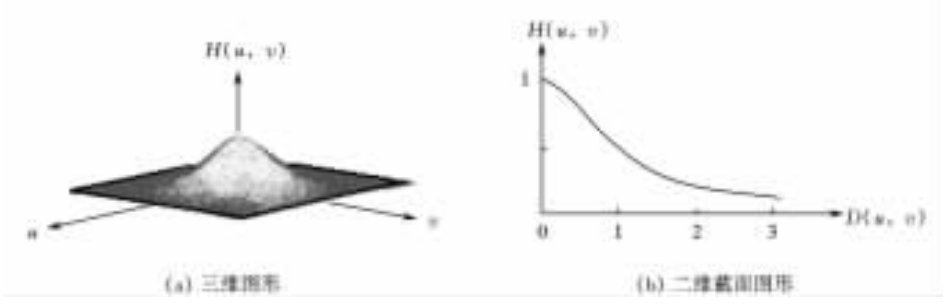


图 4-28　Butterworth 低通滤波的传递函数

⑶ 指数低通滤波。指数低通滤波也是在图像处理中常使用的滤波方法，它的传递函数的形式为

$$H(u, v) = \exp\left\{-\left[\frac{D(u, v)}{D_0}\right]^n\right\} \tag{4-29}$$

式中　D_0——一个非负的量，称为截止频率；

$D(u, v)$——频率平面上的点 (u, v) 到原点的距离，即

$$D(u, v) = \sqrt{u^2 + v^2} \tag{4-30}$$

当 $n = 1$、$H(u, v) = \frac{1}{\mathrm{e}}$ 时，$D(u, v) = D_0$，n 控制着指数函数的衰减速度。传递函数 $H(u, v)$ 的三维图形和二维截面图形如图 4-29 所示。可见指数低通滤波传递函数的衰减程度介于理想低通滤波和 Butterworth 低通滤波之间，但衰减率快于 Butterworth 低通滤波，所以其图像比经 Butterworth 低通滤波的图像稍模糊一些。

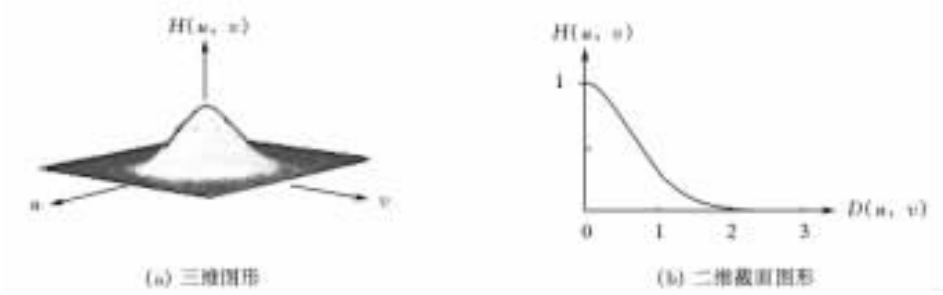


图 4-29 指数低通滤波的传递函数

(4) 梯形低通滤波。梯形低通滤波的传递函数满足下面的关系：

$$H(u,v)=\begin{cases}1 & \text{当 } D(u,v)<D_0\\ \dfrac{1}{D_0-D_1}[D(u,v)-D_1] & \text{当 } D_0\leqslant D(u,v)\leqslant D_1\\ 0 & \text{当 } D(u,v)>D_1\end{cases} \tag{4-31}$$

其中

$$D(u,v)=\sqrt{u^2+v^2} \tag{4-32}$$

式中 D_0——截止频率，要求 $D_0<D_1$。

传递函数 $H(u,v)$ 的三维图形和二维截面图形如图 4-30 所示。

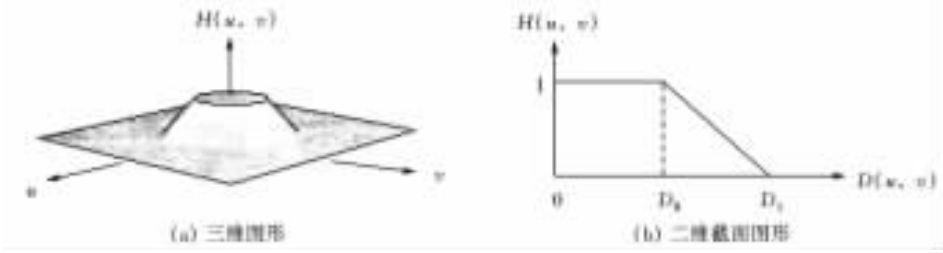


图 4-30 梯形低通滤波的传递函数

由上述可见，在平滑化处理中，虽然衰减了噪声的强度，但也使图像模糊了，所以平滑化处理是以牺牲图像清晰度为代价的。

4.4 尖锐化处理

图像尖锐化处理的目的是使模糊图像变得清晰些，即提高图像中边缘区域的清晰度。这种处理方法可以分为两类：一是在空间域内进行处理，二是在频率域内进行处理。

4.4.1 微分尖锐化处理

从 4.3.1 中可见，对图像求和取平均的效果是使图像的细节变模糊。因为求和算法在数学上是与积分算法相对应的，因此可以说积分处理的效果是使图像细节模糊；而微分与积分是相对立的，那么微分处理的效果可能与积分处理的效果相反，即微分处理将使图像的细节(灰度突变区域)清晰。根据上述思想引入了微分尖锐化处理方法。梯度是一种微分运算，在

微分尖锐化处理方法中最常用的就是梯度法，它的基本思想是：设图像函数为 $f(x, y)$，它的梯度为 $\boldsymbol{G}(f(x, y))$，数字图像某样点 (x, y) 处的梯度值和邻近样点间的灰度差成正比，因此在图像灰度变化平缓区域 $\boldsymbol{G}(f(x, y))$ 小，在线条轮廓处等灰度变化快的区域 $\boldsymbol{G}(f(x, y))$ 大，这样就可以用梯度值来代替 (x, y) 处的灰度值作图，经过这样变换后的图像在线条轮廓等灰度突变处更明显，从而达到尖锐化的目的。

某一函数 $f(x, y)$ 的梯度定义为

$$\boldsymbol{G}(f(x, y)) = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix} \tag{4-33}$$

梯度有两个重要的性质：

(1) 梯度的方向在函数 $f(x, y)$ 变化率最大的方向上。

(2) 梯度的幅值 (梯度的模) 用 $G(f(x, y))$ 表示，即

$$G(f(x, y)) = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{4-34}$$

可见，$G(f(x, y))$ 就是在梯度方向上每单位距离 $f(x, y)$ 的最大增加率。

对于数字图像，式 (4-34) 要采用离散的形式，为此可用差分运算来代替微分运算，即

$$G(f(x, y)) \approx \sqrt{[f(x, y) - f(x+1, y)]^2 + [f(x, y) - f(x, y+1)]^2} \tag{4-35}$$

为了便于计算机的计算，通常使用绝对值来代替式 (4-35)，即

$$G(f(x, y)) \approx |[f(x, y) - f(x+1, y)| + |f(x, y) - f(x, y+1)| \tag{4-36}$$

关于梯度处理还有一种近似方法 (一种经验公式) 叫 Robert 梯度，即

$$G(f(x, y)) \approx \sqrt{[f(x, y) - f(x+1, y+1)]^2 + [f(x+1, y) - f(x, y+1)]^2} \tag{4-37}$$

式 (4-37) 用绝对值近似表示为

$$G(f(x, y)) \approx |[f(x, y) - f(x+1, y+1)| + |f(x+1, y) - f(x, y+1)| \tag{4-38}$$

由上述可见，图像中样点 (x, y) 处的梯度值与邻近样点间的灰度差成正比，如图 4-31 所示。在图像中的边缘棱角等灰度突变处梯度值较大，在图像中的平滑区域梯度值较小，在图像中灰度为常数的区域梯度值为零。图 4-32 中示出了一个实际应用的例子，在图 4-32 (a) 中的图像只有两个灰度值，在全白或全黑的区域内梯度值为零，在黑白分界的区域梯度值不为零，其梯度图如图 4-32 (b) 所示。

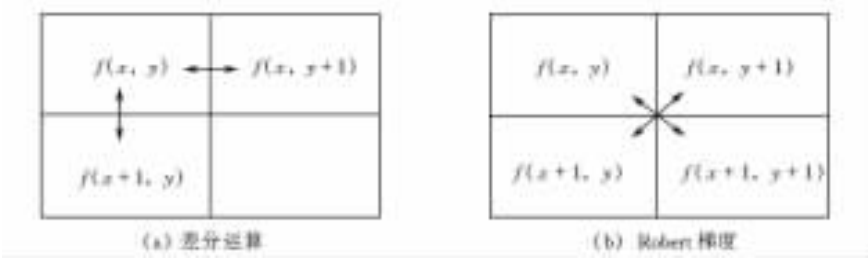


图 4-31　梯度法示意图

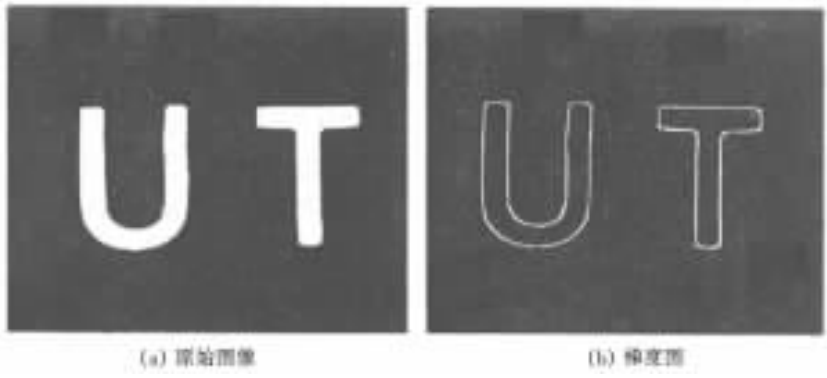


图 4-32 梯度图

由图 4-32 可见，上述方法有一个缺点，即在灰度图像 $f(x, y)$ 中比较平滑的区域在梯度图 $g(x, y)$ 中变得比较黑暗。对此常常采用一些改进的方法。一种方法是选取一个阈值 T，当梯度值超过某个阈值后，该点灰度值用梯度值来表示，否则仍保持原值，即

$$g(x, y) = \begin{cases} G(f(x, y)) & \text{若 } G(f(x, y)) \geqslant T \\ f(x, y) & \text{其他} \end{cases} \tag{4-39}$$

这样通过合理地选择 T 值，就可能既不破坏平滑区域的灰度值，又强调了图像的边缘。

另一种方法是给边缘处的灰度值规定一个特定的值 L_c，即

$$g(x, y) = \begin{cases} L_c & \text{若 } G(f(x, y)) \geqslant T \\ f(x, y) & \text{其他} \end{cases} \tag{4-40}$$

这种处理会使图像边缘的增强效果更明显。

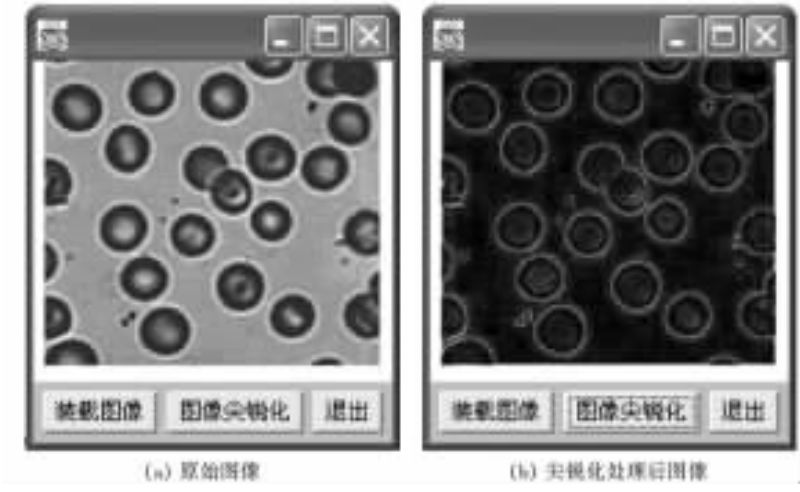


图 4-33 图像的尖锐化处理

若只研究图像边缘灰度级的变化，要求不受背景的影响，可采用下面的方法：

$$g(x, y) = \begin{cases} G(f(x, y)) & \text{若 } G(f(x, y)) \geqslant T \\ L_B & \text{其他} \end{cases} \tag{4-41}$$

式中 L_B——背景的灰度值。

若只对边缘位置感兴趣，也可采用下面的形式：

$$g(x, y) = \begin{cases} L_c & \text{若 } G(f(x, y)) \geqslant T \\ L_B & \text{其他} \end{cases} \tag{4-42}$$

这些梯度定义在数学上也许没什么道理，但运算简单，实用效果好，所以被广泛应用。程序清单 4-4 给出了式 (4-36) 的 Java Application 的程序内容，在图 4-33 中示出了该程序运行的结果。

程序清单 4-4　　SharpImage.java **源代码**

```
//SharpImage.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
//import javax.swing.*;

public class SharpImage extends Frame {
Image im, tmp;
int iw, ih;
int[] pixels;
boolean flag = false;

//构造方法
public SharpImage(){
    Panel pdown;
    Button load, run, quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    pdown = new Panel();
    pdown.setBackground(Color.lightGray);

    load = new Button("装载图像");
    run = new Button("图像尖锐化");
    quit = new Button("退出");

    this.add(pdown, BorderLayout.SOUTH);
```

```
        pdown.add(load);
        pdown.add(run);
        pdown.add(quit);

        load.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jLoad_ActionPerformed(e);
            }
        });

    run.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jRun_ActionPerformed(e);
            }
        });

        quit.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });
    }

        public void jLoad_ActionPerformed(ActionEvent e){
            //利用 MediaTracker 跟踪图像的加载
            MediaTracker tracker = new MediaTracker(this);
            im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
            tracker.addImage(im,0);

            //等待图像的完全加载
            try{
            tracker.waitForID(0);
            }catch(InterruptedException e2){ e2.printStackTrace();}

            //获取图像的宽度 iw 和高度 ih
            iw=im.getWidth(this);
            ih=im.getHeight(this);
            pixels=new int[iw*ih];

            try{
```

```
        PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        flag = true;
        repaint();
    }

    public void jRun_ActionPerformed(ActionEvent e){

        try{
        PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //像素的中间变量
        int tempPixels[ ] = new int[iw * ih];

        for(int i = 0; i < iw * ih; i++)
        {
            tempPixels[i] = pixels[i];
        }

        //对图像进行尖锐化处理, Alpha 值保持不变
        ColorModel cm = ColorModel.getRGBdefault();
        for(int i = 1; i < ih - 1; i++)
        {
            for(int j = 1; j < iw - 1; j++)
            {
                int alpha = cm.getAlpha(pixels[i * iw + j]);

                //对图像进行尖锐化
                int red6 = cm.getRed(pixels[i * iw + j + 1]);
```

```
                    int red5 = cm.getRed(pixels[i * iw + j]);
                    int red8 = cm.getRed(pixels[(i + 1) * iw + j]);
                    int sharpRed = Math.abs(red6 - red5) + Math.abs(red8 - red5);

                    int green5 = cm.getGreen(pixels[i * iw + j]);
                    int green6 = cm.getGreen(pixels[i * iw + j + 1]);
                    int green8 = cm.getGreen(pixels[(i + 1) * iw + j]);
                      int sharpGreen = Math.abs(green6 - green5) + Math.abs(green8 -
green5);

                    int blue5 = cm.getBlue(pixels[i * iw + j]);
                    int blue6 = cm.getBlue(pixels[i * iw + j + 1]);
                    int blue8 = cm.getBlue(pixels[(i + 1) * iw + j]);
                    int sharpBlue = Math.abs(blue6 - blue5) + Math.abs(blue8 - blue5);

                    if(sharpRed > 255) {sharpRed = 255;}
                    if(sharpGreen > 255) {sharpGreen = 255;}
                    if(sharpBlue > 255) {sharpBlue = 255;}

                    tempPixels[i * iw + j] = alpha << 24 | sharpRed << 16 | sharpGreen << 8
| sharpBlue;
                }
            }

            //将数组中的像素产生一个图像
            ImageProducer ip = new MemoryImageSource(iw, ih, tempPixels, 0, iw);
            tmp = createImage(ip);
            flag = true;
            repaint();
        }
        public void jQuit_ActionPerformed(ActionEvent e){
            System.exit(0);
        }

    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flag){
            g.drawImage(tmp, 10, 20, this);
        }else { }
    }
```

```
//定义 main 方法,设置窗口的大小,显示窗口
public static void main(String[ ] args) {
    SharpImage si = new SharpImage();
    si.setSize(400,350);
    si.show();
}
}
```

4.4.2　高通滤波法

高通滤波法是在频域内对图像进行尖锐化处理。从 4.3.2 中的叙述可见，图像的模糊实质是图像中高频分量被衰减了。因为图像的边缘信息主要集中在高频部分，所以为了增强图像的细节，可以采用高通滤波的方法。与低通滤波相类似，常用的高通滤波有理想高通滤波、Butterworth 高通滤波、指数高通滤波、梯形高通滤波等，下面分别加以介绍。

(1) 理想高通滤波。理想高通滤波的传递函数的形式如下

$$H(u,v)=\begin{cases}0 & \text{当 } D(u,v)\leqslant D_0\\ 1 & \text{当 } D(u,v)>D_0\end{cases} \tag{4-43}$$

这里 D_0 为截止频率，$D(u,v)$ 是频率平面上的点 (u,v) 到原点的距离，即

$$D(u,v)=\sqrt{u^2+v^2} \tag{4-44}$$

在图 4-34 中示出了理想高通滤波传递函数 $H(u,v)$ 的三维图形和二维截面图形，可见在以 D_0 为半径的圆外的所有频率的传递函数 $H(u,v)=1$，而在截止频率之内的传递函数 $H(u,v)=0$，即图像的低频分量全部被衰减掉，图像的高频分量全部通过，这就使得图像的边缘细节清晰，但牺牲了图像的背景信息。

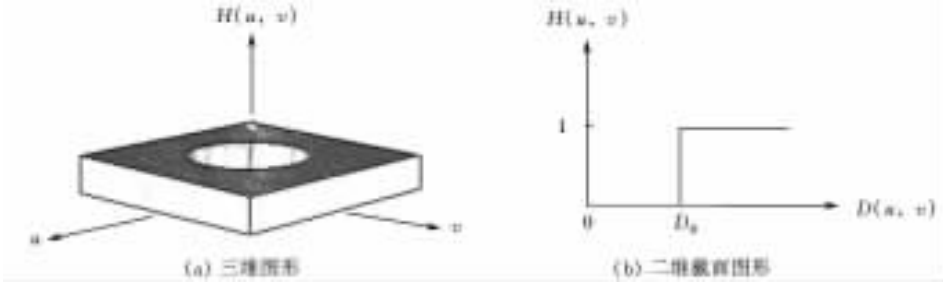


图 4-34　理想高通滤波的传递函数

(2) Butterworth 高通滤波。n 阶 Butterworth 高通滤波的传递函数满足下面的关系

$$H(u,v)=\frac{1}{1+\left[\dfrac{D_0}{D(u,v)}\right]^{2n}} \tag{4-45}$$

式中　D_0——截止频率，$D(u,v)$ 满足式 (4-44)，当 $n=1$，$H(u,v)=\frac{1}{2}$ 时，$D(u,v)=D_0$。

在图 4-35 中示出了 $n=1$ 时的传递函数 $H(u,v)$ 的三维图形和二维截面图形。与理想高通滤波相比，Butterworth 高通滤波的传递函数有一个平滑的过渡区，它可以保留部分图像背景。

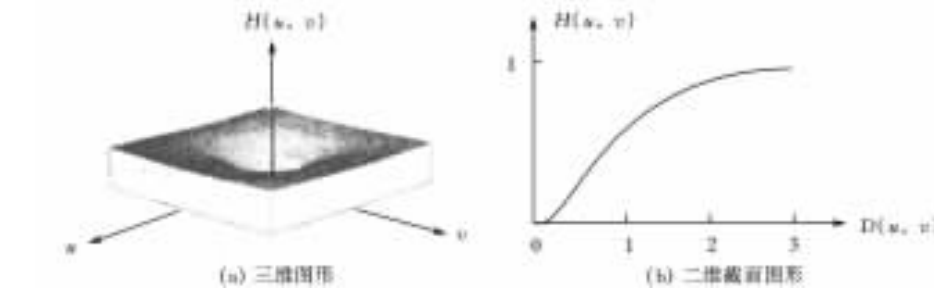


图 4-35 Butterworth 高通滤波的传递函数

⑶ 指数高通滤波。指数高通滤波的传递函数满足下面的关系

$$H(u,v) = \exp\left\{-\left[\frac{D_0}{D(u,v)}\right]^n\right\} \tag{4-46}$$

式中 D_0——截止频率，$D(u,v)$ 满足式 (4-44)，当 $n=1$，$H(u,v)=\frac{1}{e}$ 时，$D(u,v)=D_0$。

n 控制着指数函数上升的速度。传递函数 $H(u,v)$ 的三维图形和二维截面图形如图 4-36 所示，这种滤波的效果介于理想高通滤波和 Butterworth 高通滤波之间。

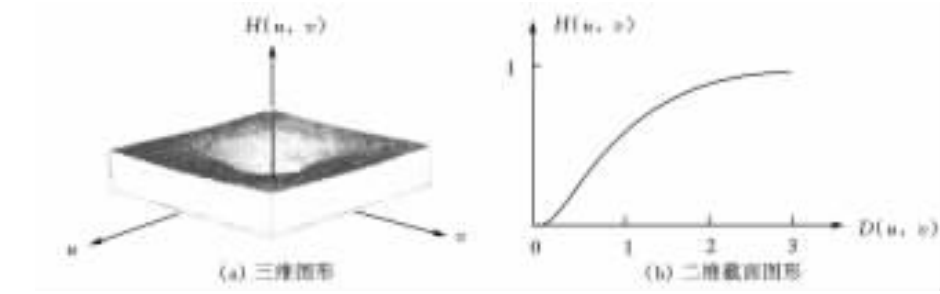


图 4-36 指数高通滤波的传递函数

⑷ 梯形高通滤波。梯形高通滤波的传递函数满足下面的关系

$$H(u,v) = \begin{cases} 0 & \text{当 } D(u,v) < D_1 \\ \dfrac{1}{D_0 - D_1}[D(u,v) - D_1] & \text{当 } D_1 \leqslant D(u,v) \leqslant D_0 \\ 1 & \text{当 } D(u,v) > D_0 \end{cases} \tag{4-47}$$

式中 D_0——截止频率，$D(u,v)$ 满足式 (4-44)，要求 $D_0 > D_1$。

传递函数 $H(u,v)$ 的三维图形和二维截面图形如图 4-37 所示。

高通滤波虽然加强了图像的边缘细节，但使噪声也增强了。在实际应用中，可以采用几种方法综合处理以得到较满意的图像。

4.5 中值滤波

从前面介绍的内容中可知，低通滤波可以减弱图像中的噪声，但同时也使图像细节模糊了；而高通滤波增强了图像中的边缘细节，但同时也增强了噪声信号。下面要介绍的中值滤波是介于两者之间，在一定条件下，中值滤波方法可以做到既减弱了噪声又保护了图像的细

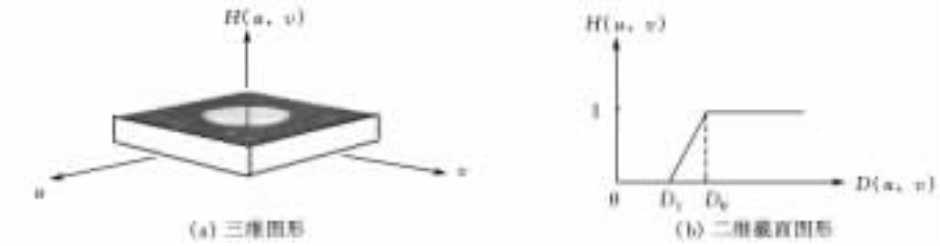


图 4-37　梯形高通滤波的传递函数

节，得到较好的处理结果。中值滤波也称为列滤波 (Rang filer)，它的基本思想是把数字序列中一点 x_i 的值用该点的一个邻域中各点的中值来代替。那么，什么是中值呢？假设有一组数值 x_{i1}，x_{i2}，…，x_{in}，把这 n 个数值按大小顺序排列成 $x_{i1} \leqslant x_{i2} \leqslant$，…，$x_{im}$，则这个数组的中值为

$$y = \mathrm{Med}(x_{i1}, x_{i2}, \cdots, x_{in}) = \begin{cases} x_{i[(n+1)/2]} & n \text{ 为奇数} \\ \dfrac{1}{2}\{x_{i(n/2)} + x_{i[(n/2)+1]}\} & n \text{ 为偶数} \end{cases} \tag{4-48}$$

4.5.1　一维中值滤波

中值滤波通常采用一个含有奇数个样点的滑动窗口，将窗口内正中间那个样点的值用窗口内各个样点的中值来代替。假设输入序列为 $\{x_i, i \in I\}$，I 为自然数集合或子集，窗口长度为 n，则滤波器的输出为

$$y = \mathrm{Med}\{x_i\} = \mathrm{Med}(x_{i-u}, \cdots, x_i, \cdots, x_{i+u}) \tag{4-49}$$

式中　$u = \dfrac{n-1}{2}$。

【例 4-3】　有一脉冲干扰信号为 $\{x_i\} = \{1\ \ 1\ \ 5\ \ 5\ \ 5\ \ 8\ \ 5\ \ 5\ \ 1\ \ 1\}$，如图 4-38 (a) 所示，采用长度为 3 的窗口对它进行中值滤波。根据式 (4-48)，序列 $\{x_i\}$ 的中值满足

$$y_i = x_{i[(n+1)/2]}$$

这里 $n = 3$。中值滤波后输出的结果为 $\{1\ \ 1\ \ 5\ \ 5\ \ 5\ \ 5\ \ 5\ \ 5\ \ 1\ \ 1\}$，如图 4-38 (b) 所示，可见处理后滤掉了脉冲干扰。

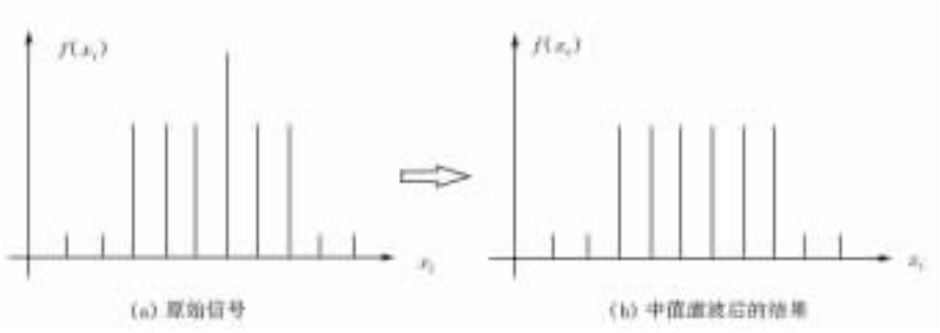


图 4-38　一维中值滤波

4.5.2 二维中值滤波

二维中值滤波是将图像中某一个样点邻域中的各个元素的灰度值形成一个行序列，然后用与一维中值滤波相同的方法求出该序列的中值，用这个中值代替这一个样点的灰度值，从而形成新的图像。假设 $\{x_{ij}, (i, j) \in I^2\}$ 表示数字图像各样点的灰度值，I 为自然数集合或子集，滤波窗口 A 的二维输出中值滤波为

$$y_{ij} = \text{Med}\{x_{(i+r),(j+s)}, (r, s) \in A, (i, j) \in I^2\} \tag{4-50}$$

二维窗口 A 可以取方形、圆形和十字形。有时为了强调某些特征，又引入了加权中值滤波，它是改变窗口中变量的个数，然后对扩张后的数字集求中值。下面以 3×3 的窗口为例对此进一步说明。

$x_{i-1,j-1}$	$x_{i-1,j}$	$x_{i-1,j+1}$
$x_{i,j-1}$	$x_{i,j}$	$x_{i,j+1}$
$x_{i+1,j-1}$	$x_{i+1,j}$	$x_{i+1,j+1}$

图 4-39 x_{ij}的邻域

【例 4-4】 假设某一窗口如图 4-39 所示。

它是样点 x_{ij}的一个邻域，可写成一个行序列

$$\{x_{i-1,j-1}, x_{i-1,j}, x_{i-1,j+1}, x_{i,j-1}, x_{i,j}, x_{i,j+1}, x_{i+1,j-1}, x_{i+1,j}, x_{i+1,j+1}\},$$

该序列的中值为

$$y_{ij} = \text{Med}\{x_{i-1,j-1}, x_{i-1,j}, x_{i-1,j+1}, x_{i,j-1}, x_{i,j}, x_{i,j+1}, x_{i+1,j-1}, x_{i+1,j}, x_{i+1,j+1}\}$$

对上述窗口也可以求加权中值，即对原窗口中间的点取三个值，上、下、左和右点各取两个值，对角线上的点各取一个值，形成序列

$$\{x_{i-1,j-1}, x_{i-1,j}, x_{i-1,j}, x_{i-1,j+1}, x_{i,j-1}, x_{i,j-1}, x_{i,j}, x_{i,j}, x_{i,j}, x_{i,j+1}, x_{i,j+1}, x_{i+1,j-1}, x_{i+1,j}, x_{i+1,j}, x_{i+1,j+1}\},$$

加权后的中值为

$$\begin{aligned} y_{ij} &= \text{Weighted} - \text{Med}\{x_{i-1,j-1}, x_{i-1,j}, x_{i-1,j+1}, x_{i,j-1}, x_{i,j}, x_{i,j+1}, x_{i+1,j-1}, x_{i+1,j}, x_{i+1,j+1}\} \\ &= \text{Med}\{x_{i-1,j-1}, x_{i-1,j}, x_{i-1,j}, x_{i-1,j+1}, x_{i,j-1}, x_{i,j-1}, x_{i,j}, x_{i,j}, x_{i,j}, x_{i,j+1}, x_{i,j+1}, x_{i+1,j-1}, x_{i+1,j}, x_{i+1,j}, x_{i+1,j+1}\} \end{aligned}$$

假设图 4-40 是某一图像中的一部分，在图像中阴影区域中央样点的灰度值应为 64，但由于受到噪声的影响变成了 255，现在使用一个 3×3 的窗口分别进行平滑化 (邻域平均法) 和中值滤波法加以处理。

根据图 4-40 中所示，以该噪声点为中心取一个 3×3 的邻域，由邻域平均法得出邻域中心样点处理后的灰度值为

$$\frac{64+64+64+64+255+64+64+255}{8} = 111.75$$

邻域平均法处理后中央样点的灰度值为 111.75，可见这种方法没有完全消除噪声。

下面使用中值滤波方法，中央样点的中值为

$$y = \text{Med}\{64, 64, 64, 64, 255, 255, 64, 64, 255\} = 64$$

中值滤波后这一点灰度值为 64，表明噪声被消除了。通常，如果噪声的样点数小于邻域中样点数的一半，使用中值滤波法可以较好地去除噪声。

64	64	64	64	64	64	64	64
64	64	64	64	64	64	64	64
64	64	64	64	64	255	255	64
64	64	64	255	255	64	64	64
64	64	64	64	255	64	64	255
64	64	64	64	64	255	64	128
64	64	64	64	64	64	128	128
64	64	64	64	255	128	128	128

图 4-40　某一图像

在图 4-41 示出了中值滤波、低通滤波和高通滤波处理后图像的效果，这里图 4-41 (b) 是中值滤波后的图像，图 4-41 (c) 是低通滤波后的图像，图 4-41 (d) 是高通滤波后的图像，从中可以看到它们间的差异。在程序清单 4-5 给出了 Java Application 进行中值滤波的程序。

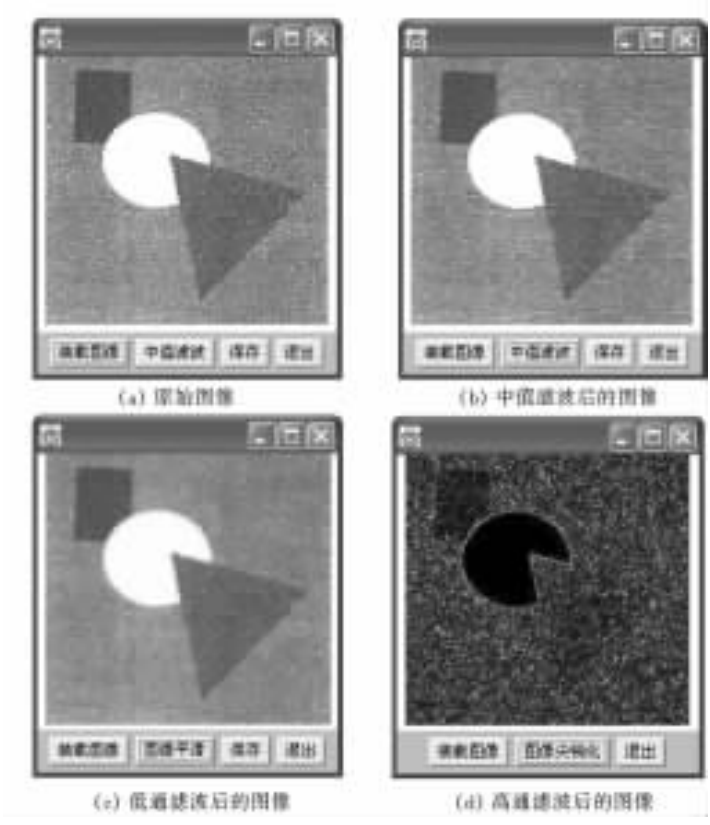


图 4-41　中值滤波、低通滤波和高通滤波

程序清单 4-5　　MedianImage.java **源代码**

```
//MedianImage.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class MedianImage extends Frame {
Image im, tmp;
int i, iw, ih;
int[] pixels;
boolean flag = false;

//构造方法
public MedianImage(){
    Panel pdown;
    Button load, run, save, quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    pdown = new Panel();
    pdown.setBackground(Color.lightGray);

    load = new Button("装载图像");
    run = new Button("中值滤波");
    save = new Button("保存");
    quit = new Button("退出");

    this.add(pdown, BorderLayout.SOUTH);

    pdown.add(load);
    pdown.add(run);
    pdown.add(save);
    pdown.add(quit);

    load.addActionListener(new ActionListener(){
```

```
        public void actionPerformed(ActionEvent e){
            jLoad_ActionPerformed(e);
        }
    });

    run.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jRun_ActionPerformed(e);
        }
    });

    quit.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jQuit_ActionPerformed(e);
        }
    });
    save.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            jSave_ActionPerformed(e);
        }
    });
}

public void jLoad_ActionPerformed(ActionEvent e){
    //利用 MediaTracker 跟踪图像的加载
    MediaTracker tracker = new MediaTracker(this);
    im=Toolkit.getDefaultToolkit().getImage("Color.jpg");
    tracker.addImage(im,0);

    //等待图像的完全加载
    try{
    tracker.waitForID(0);
        }catch(InterruptedException e2){ e2.printStackTrace();}

    //获取图像的宽度 iw 和高度 ih
    iw=im.getWidth(this);
    ih=im.getHeight(this);
    pixels=new int[iw*ih];

    try{
```

```
CPixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        flag = true;
        repaint();
}

public void jRun_ActionPerformed(ActionEvent e){

        try{
        PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //对图像进行中值滤波, Alpha 值保持不变
        ColorModel cm = ColorModel.getRGBdefault();
        for(i = 1;i < ih - 1;i++)
        {
            for(int j = 1;j < iw - 1;j++)
            {
                int red, green, blue;
                int alpha = cm.getAlpha(pixels[i * iw + j]);

                int red2 = cm.getRed(pixels[(i - 1) * iw + j]);
                int red4 = cm.getRed(pixels[i * iw + j - 1]);
                int red5 = cm.getRed(pixels[i * iw + j]);
                int red6 = cm.getRed(pixels[i * iw + j + 1]);
                int red8 = cm.getRed(pixels[(i + 1) * iw + j]);

                //水平方向进行中值滤波
                if(red4 >= red5){
                    if(red5 >= red6) {red = red5;}
```

```
        else{
        if(red4>=red6){red=red6;}
        else{red=red4;}
}}
else{
if(red4>red6){red=red4;}
        else{
        if(red5>red6){red=red6;}
        else{red=red5;}
}}

int green2=cm.getGreen(pixels[(i-1)*iw+j]);
int green4=cm.getGreen(pixels[i*iw+j-1]);
int green5=cm.getGreen(pixels[i*iw+j]);
int green6=cm.getGreen(pixels[i*iw+j+1]);
int green8=cm.getGreen(pixels[(i+1)*iw+j]);

//水平方向进行中值滤波
if(green4>=green5){
        if(green5>=green6){green=green5;}
        else{
        if(green4>=green6){green=green6;}
        else{green=green4;}
}}
else{
if(green4>green6){green=green4;}
        else{
        if(green5>green6){green=green6;}
        else{green=green5;}
}}


int blue2=cm.getBlue(pixels[(i-1)*iw+j]);
int blue4=cm.getBlue(pixels[i*iw+j-1]);
int blue5=cm.getBlue(pixels[i*iw+j]);
int blue6=cm.getBlue(pixels[i*iw+j+1]);
int blue8=cm.getBlue(pixels[(i+1)*iw+j]);

//水平方向进行中值滤波
if(blue4>=blue5){
```

```
                    if(blue5>=blue6) {blue=blue5;}
                    else{
                    if(blue4>=blue6) {blue=blue6;}
                    else{blue=blue4;}
                }}
                else{
                if(blue4>blue6) {blue=blue4;}
                    else{
                    if(blue5>blue6) {blue=blue6;}
                    else{blue=blue5;}
                }}
                pixels[i*iw+j]=alpha<<24|red<<16|green<<8|blue;
            }
        }

        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        flag=true;
        repaint();
    }
    public void jSave_ActionPerformed(ActionEvent e){
        //保存该图像,还不能实现.
        ImageIcon ii=new ImageIcon("copyColor.jpg");
        ii.setImage(tmp);
    }

    public void jQuit_ActionPerformed(ActionEvent e){
        System.exit(0);
    }
    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flag){
            g.drawImage(tmp,10,20,this);
        }else { }
    }

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[] args) {
        MedianImage mi = new MedianImage();
```

```
        mi.setSize(300,300);
        mi.show();
    }
}
```

练 习 题

1　某一图像如下所示

1	1	1	1	1	1	1	1
1	3	3	3	3	3	3	1
1	3	2	2	2	2	3	1
1	3	7	4	4	0	3	1
1	3	0	4	4	2	3	1
1	3	2	7	2	2	3	1
1	3	3	3	3	3	3	1
1	1	1	1	1	1	1	1

试画出它的直方图，并进行直方图均匀化处理。

2　将第一章练习 2 中存入计算机内的图像用 Java 语言进行平缓化处理。

3　将第一章练习 2 中存入计算机内的图像用 Java 语言进行尖锐化处理。

4　将第一章练习 2 中存入计算机内的图像用 Java 语言进行中值滤波。

第 5 章 图像复原

图像复原是图像处理的另一项主要内容，它的目的是改善图像的质量。在成像过程中，由于成像系统等各种因素的影响，可能使图像的质量降低，这种图像质量的降低称为“退化”或“降质”。图像复原是在研究图像退化的基础上，以被退化的图像为依据，根据某些先验知识建立一种数学模型，然后用逆推的方法估算出原始图像。

5.1 退化的数学模型

5.1.1 退化模型及复原的基本过程

图像复原的关键问题是建立退化模型，在建模过程中认为图像退化是由于存在一个退化系统，原始的输入图像经过退化系统和噪声的作用后变成了退化的输出图像。假设已知的一幅黑白静止的退化平面图像为 $g(x, y)$，并且退化系统为 H，噪声为 $n(x, y)$，这样退化过程可以表示为图 5-1。

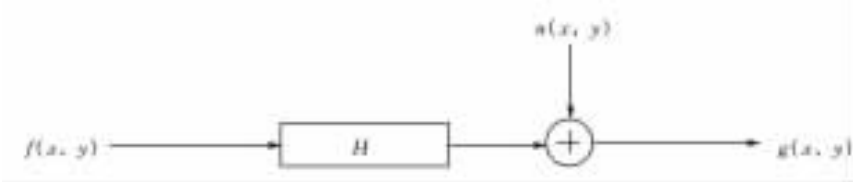


图 5-1 图像退化过程的模型

可见，一幅纯净的图像 $f(x, y)$ 是由于通过了一个系统 H 及加入噪声而产生了退化，从而输出一幅退化图像 $g(x, y)$。这样输入与输出之间的关系可以表示为

$$g(x, y) = Hf(x, y) + n(x, y) \tag{5-1}$$

为了讨论方便，这里暂时令 $n(x, y) = 0$，这样式 (5-1) 简化成

$$g(x, y) = Hf(x, y) \tag{5-2}$$

由式 (5-2) 可见，图像复原可以看成是一个估计过程，如果已知函数 $g(x, y)$ 和退化系统 H，根据式 (5-2) 可求出 $f(x, y)$。通常 $g(x, y)$ 是已知的，这样复原处理的关键是对系统 $H(x, y)$ 的了解，因此对退化系统作以下假设：

(1) 系统 H 是线性的，根据线性系统的特性，存在如下关系

$$\begin{aligned} H(k_1 f_1(x, y) + k_2 f_2(x, y)) &= Hk_1 f_1(x, y) + Hk_2 f_2(x, y) \\ &= k_1 Hf_1(x, y) + k_2 Hf_2(x, y) \\ &= k_1 g_1(x, y) + k_2 g_2(x, y) \end{aligned} \tag{5-3}$$

式中 $f_1(x, y)$ 和 $f_2(x, y)$ 是系统的两个输入图像；

$g_1(x, y)$ 和 $g_2(x, y)$ 是对应的输出图像；

k_1 和 k_2 为常数。

⑵ 系统 H 是空间位置不变系统，即对任意输入图像 $f(x, y)$ 有

$$Hf(x-\alpha, y-\beta) = g(x-\alpha, y-\beta) \tag{5-4}$$

式中　α 和 β 为空间位置的位移量。

可见在图像中任一点的响应仅取决于那一点的值，而于位移量无关。

5.1.2　连续函数的退化模型

在 2.1.2 中介绍了二维冲激函数的表达式

$$\delta(x-\alpha, y-\beta) = \begin{cases} \infty & x=\alpha, y=\beta \\ 0 & x\neq\alpha, y\neq\beta \end{cases} \tag{5-5}$$

$$\iint \delta(x-\alpha, y-\beta)\,\mathrm{d}x\mathrm{d}y = 1 \tag{5-6}$$

根据冲激函数的性质，可以将 $f(x, y)$ 写成

$$f(x, y) = \iint_{-\infty}^{\infty} f(\alpha, \beta)\,\delta(x-\alpha, y-\beta)\,\mathrm{d}\alpha\mathrm{d}\beta \tag{5-7}$$

根据式 (5-2) 有

$$g(x, y) = Hf(x, y) = H\iint_{-\infty}^{\infty} f(\alpha, \beta)\,\delta(x-\alpha, y-\beta)\,\mathrm{d}\alpha\mathrm{d}\beta \tag{5-8}$$

由于 H 是线性系统，所以式 (5-8) 可以写为

$$g(x, y) = Hf(x, y) = \iint_{-\infty}^{\infty} [f(\alpha, \beta)\,\delta(x-\alpha, y-\beta)]\,\mathrm{d}\alpha\mathrm{d}\beta \tag{5-9}$$

其中 $f(\alpha, \beta)$ 与 x 和 y 无关，式 (5-9) 可以继续写成

$$g(x, y) = Hf(x, y) = \iint_{-\infty}^{\infty} f(\alpha, \beta)\,H\delta(x-\alpha, y-\beta)\,\mathrm{d}\alpha\mathrm{d}\beta \tag{5-10}$$

令

$$h(x, \alpha, y, \beta) = H\delta(x-\alpha, y-\beta) \tag{5-11}$$

称 $h(x, \alpha, y, \beta)$ 为系统 H 的冲激响应，也就是说 $h(x, \alpha, y, \beta)$ 是系统 H 对坐标为 (α, β) 处的冲激函数 $\delta(x-\alpha, y-\beta)$ 的响应。在光学中，冲激函数可看作为一点光源，这时 $h(x, \alpha, y, \beta)$ 被称为点扩展函数。这样退化图像函数可以表示成

$$g(x, y) = \iint_{-\infty}^{\infty} f(\alpha, \beta)\,h(x, \alpha, y, \beta)\,\mathrm{d}\alpha\mathrm{d}\beta \tag{5-12}$$

式 (5-12) 在线性系统中是很重要的，如果系统 H 的冲激响应是已知的，则对于任一输入函数 $f(x, y)$ 的响应可以根据式 (5-12) 算出。

如果 H 是空间位置不变的系统，则

$$H\delta(x-\alpha, y-\beta) = h(x, \alpha, y, \beta) = h(x-\alpha, y-\beta) \tag{5-13}$$

这样式 (5-12) 可以写成

$$g(x, y) = \iint_{-\infty}^{\infty} f(\alpha, \beta)\,h(x-\alpha, y-\beta)\,\mathrm{d}\alpha\mathrm{d}\beta \tag{5-14}$$

根据卷积的定义，上式又可以表示成

$$g(x, y) = f(x, y) * h(x, y) \tag{5-15}$$

可见，若把退化过程看作是一个线性空间位置不变的系统，那么系统输出的退化图像

$g(x, y)$ 是输入图像 $f(x, y)$ 与系统冲激响应的卷积。对式 (5-15) 两边取傅立叶变换，得

$$G(u, v) = H(u, v)F(u, v) \tag{5-16}$$

这里 $G(u, v)$，$H(u, v)$ 和 $F(u, v)$ 分别为 $g(x, y)$，$f(x, y)$ 和 $h(x, y)$ 的傅立叶变换。

在考虑噪声 $n(x, y)$ 的影响，连续函数的线性退化模型为

$$g(x, y) = \iint_{-\infty}^{\infty} f(\alpha, \beta)h(x-\alpha, y-\beta)\mathrm{d}\alpha\mathrm{d}\beta + n(x, y) \tag{5-17}$$

或

$$g(x, y) = f(x, y) * h(x, y) + n(x, y) \tag{5-18}$$

或

$$G(u, v) = H(u, v)F(u, v) + N(u, v) \tag{5-19}$$

这里 $N(u, v)$ 是 $n(x, y)$ 的傅立叶变换。

在很多情况下，都可以将退化系统近似成线性的空间位置不变系统，这样就可以利用线性系统的理论来处理图像的复原问题。

5.1.3 离散函数的退化模型

1. 一维离散的退化模型

为了更容易理解一维离散的、空间位置不变的退化模型，首先从一个例子着手。假设 $f(x)$ 是输入函数，x 的取样点为 0，1，2，…，$A-1$；$h(x)$ 是退化系统的冲激响应，x 的取样点为 0，1，2，…，$B-1$。根据式 (5-17)，在不考虑噪声的情况下系统的输出函数为

$$g(x) = \sum_m f(m)h(x-m) \tag{5-20}$$

【例 5-1】 如果 $f(x)$ 的取样点数为 $A=4$，即 $f(0)$，$f(1)$，$f(2)$，$f(3)$。$h(x)$ 的取样点数为 $B=3$，即 $h(0)$，$h(1)$，$h(2)$。那么根据式 (5-20) 可以得到下列方程：

$$\begin{aligned}
g(0) &= f(0)h(0)\\
g(1) &= f(0)h(1) + f(1)h(0)\\
g(2) &= f(0)h(2) + f(1)h(1) + f(2)h(0)\\
g(3) &= f(1)h(2) + f(2)h(1) + f(3)h(0)\\
g(4) &= f(2)h(2) + f(3)h(1)\\
g(5) &= f(3)h(2)
\end{aligned}$$

上述用矩阵表示为

$$\begin{bmatrix} g(0)\\ g(1)\\ g(2)\\ g(3)\\ g(4)\\ g(5) \end{bmatrix} = \begin{bmatrix} h(0) & 0 & 0 & 0 & 0 & 0\\ h(1) & h(0) & 0 & 0 & 0 & 0\\ h(2) & h(1) & h(0) & 0 & 0 & 0\\ 0 & h(2) & h(1) & h(0) & 0 & 0\\ 0 & 0 & h(2) & h(1) & h(0) & 0\\ 0 & 0 & 0 & h(2) & h(1) & h(0) \end{bmatrix} \begin{bmatrix} f(0)\\ f(1)\\ f(2)\\ f(3)\\ 0\\ 0 \end{bmatrix}$$

经过上述运算后，$f(x)$ 和 $g(x)$ 均成为 6 阶列矩阵，$h(x)$ 成为 6×6 的矩阵。这里需要注意的是，$f(x)$ 和 $h(x)$ 原来的周期不同 (即 $A=4$，$B=3$)，但代入退化模型后两者具有相同的周期 (即 $4+3-1=6$)，可见共同的周期可以表示成 $A+B-1$。在上面的例子中，为了使两者具有共同的周期，在原有的周期外用 0 来填补，这种方法称为加零延拓。将上述方法加以推广，若 $f(x)$ 和 $h(x)$ 不具备周期性或周期不同，可以用加零延拓的方法，使之具有共同

的周期

$$M = A + B - 1 \tag{5-21}$$

令

$f_e(x)$ 和 $h_e(x)$ 分别表示加零延拓后的函数，则一维离散的、空间位置不变的退化模型为

$$g_e(x) = \sum_{m=0}^{M-1} f_e(m)\, h_e(x - m) \tag{5-22}$$

这里　$x = 0,1,2,\cdots,M-1$，$M \geqslant A + B - 1$

$$f_e(x) = \begin{cases} f(x) & 0 \leqslant x \leqslant A-1 \\ 0 & A-1 < x < M-1 \end{cases} \tag{5-23}$$

$$h_e(x) = \begin{cases} h(x) & 0 \leqslant x \leqslant B-1 \\ 0 & B-1 < x < M-1 \end{cases} \tag{5-24}$$

上述离散模型的矩阵表达式为

$$\boldsymbol{g}_e(x) = \boldsymbol{f}_e(x)\, \boldsymbol{h}_e(x) \tag{5-25}$$

其中

$$\boldsymbol{f}_e(x) = \begin{bmatrix} f_e(0) \\ f_e(1) \\ \cdots \\ f_e(M-1) \end{bmatrix} \tag{5-26}$$

$$\boldsymbol{g}_e(x) = \begin{bmatrix} g_e(0) \\ g_e(1) \\ \cdots \\ g_e(M-1) \end{bmatrix} \tag{5-27}$$

$$\boldsymbol{h}_e(x) = \begin{bmatrix} h_e(0) & h_e(-1) & h_e(-2) & \cdots & h_e(-M+1) \\ h_e(1) & h_e(0) & h_e(-1) & \cdots & h_e(-M+2) \\ h_e(2) & h_e(1) & h_e(0) & \cdots & h_e(-M+3) \\ \cdots & \cdots & \cdots & & \cdots \\ h_e(M-1) & h_e(M-2) & h_e(M-3) & \cdots & h_e(0) \end{bmatrix} \tag{5-28}$$

因为周期为 M，所以 $h_e(x) = h_e(x + M)$，这样式 (5-28) 可以表示成

$$\boldsymbol{h}_e(x) = \begin{bmatrix} h_e(0) & h_e(M-1) & h_e(M-2) & \cdots & h_e(1) \\ h_e(1) & h_e(0) & h_e(M-1) & \cdots & h_e(2) \\ h_e(2) & h_e(1) & h_e(0) & \cdots & h_e(3) \\ \cdots & \cdots & \cdots & & \cdots \\ h_e(M-1) & h_e(M-2) & h_e(M-3) & \cdots & h_e(0) \end{bmatrix} \tag{5-29}$$

可见这是一个 $M \times M$ 的循环矩阵。

2. 二维离散的退化模型

将一维离散的退化模型推广到二维的情况，假设某一数字图像函数 $f(x, y)$ 的大小为 $A \times B$，点扩展函数 $h(x, y)$ 的大小为 $C \times D$，通过加零延拓后的大小为 $M \times N$，即

$$f_e(x,y)=\begin{cases}f(x,y) & 0\leqslant x\leqslant A-1,\quad 0\leqslant y\leqslant B-1\\ 0 & A-1<x\leqslant M-1,\quad B-1<x\leqslant N-1\end{cases} \tag{5-30}$$

$$h_e(x,y)=\begin{cases}h(x,y) & 0\leqslant x\leqslant C-1,\quad 0\leqslant y\leqslant D-1\\ 0 & C-1<x\leqslant M-1,\quad D-1<y\leqslant N-1\end{cases} \tag{5-31}$$

这样 $f_e(x,y)$ 和 $h_e(x,y)$ 分别成为二维周期函数，它们在 x 和 y 上的周期分别为 M 和 N。根据式 (5-22) 就可以得到二维、线性、空间不变的离散退化模型，

$$g_e(x,y)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}f_e(m,n)\,h_e(x-m,y-n) \tag{5-32}$$

其中

$$x=0,1,2,\cdots,M-1$$
$$y=0,1,2,\cdots,N-1$$

$g_e(x,y)$ 也是周期函数，其周期与 $f_e(x,y)$ 和 $h_e(x,y)$ 相同，为 $M\geqslant A+C-1$，$N\geqslant B+D-1$。如果考虑噪声的影响，式 (5-32) 表示为

$$g_e(x,y)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}f_e(m,n)\,h_e(x-m,y-n)+n_e(x,y) \tag{5-33}$$

这里 $x=0,1,2,\cdots,M-1$；$y=0,1,2,\cdots,N-1$。$n_e(x,y)$ 是 $M\times N$ 的离散拓延噪声项。

式 (5-33) 可以用矩阵表示为

$$\boldsymbol{g}=\boldsymbol{h}\boldsymbol{f}+\boldsymbol{n} \tag{5-34}$$

$\boldsymbol{g}$，$\boldsymbol{h}$，$\boldsymbol{f}$ 和 $\boldsymbol{n}$ 分别代表 $g_e(x,y)$，$h_e(x,y)$，$f_e(x,y)$ 和 $n_e(x,y)$ 的矩阵，$\boldsymbol{g}$，$\boldsymbol{f}$ 和 $\boldsymbol{n}$ 为阶 MN 列矩阵，即

$$\boldsymbol{f}=\begin{bmatrix}f(0,0)\\ f(0,1)\\ \cdots\\ f(0,N-1)\\ f(1,0)\\ f(1,1)\\ \cdots\\ f(1,N-1)\\ \cdots\\ f(M-1,0)\\ f(M-1,1)\\ \cdots\\ f(M-1,N-1)\end{bmatrix} \tag{5-35}$$

$$\boldsymbol{g}=\begin{bmatrix} g(0,0) \\ g(0,1) \\ \cdots \\ g(0,N-1) \\ g(1,0) \\ g(1,1) \\ \cdots \\ g(1,N-1) \\ \cdots \\ g(M-1,0) \\ g(M-1,1) \\ \cdots \\ g(M-1,N-1) \end{bmatrix} \tag{5-36}$$

$\boldsymbol{n}$ 的形式与上述相同，这里就不具体表达了。$\boldsymbol{h}$ 为 $MN\times MN$ 阶矩阵，它可以分成 M^2 个部分，每一部分为 $N\times N$ 阶分块矩阵，即

$$\boldsymbol{H}=\begin{bmatrix} \boldsymbol{H}_0 & \boldsymbol{H}_{M-1} & \boldsymbol{H}_{M-1} & \cdots & \boldsymbol{H}_1 \\ \boldsymbol{H}_1 & \boldsymbol{H}_0 & \boldsymbol{H}_{M-2} & \cdots & \boldsymbol{H}_2 \\ \boldsymbol{H}_2 & \boldsymbol{H}_1 & \boldsymbol{H}_0 & \cdots & \boldsymbol{H}_3 \\ \cdots & \cdots & \cdots & & \cdots \\ \boldsymbol{H}_{M-1} & \boldsymbol{H}_{M-2} & \boldsymbol{H}_{M-3} & \cdots & \boldsymbol{H}_0 \end{bmatrix} \tag{5-37}$$

其中每一块 $\boldsymbol{H}_j$ 为

$$\boldsymbol{H}_j=\begin{bmatrix} h_e(j,0) & h_e(j,N-1) & h_e(j,N-2) & \cdots & h_e(j,1) \\ h_e(j,1) & h_e(j,0) & h_e(j,N-1) & \cdots & h_e(j,2) \\ h_e(j,2) & h_e(j,1) & h_e(j,0) & \cdots & h_e(j,3) \\ \cdots & \cdots & \cdots & & \cdots \\ h_e(j,N-1) & h_e(j,N-2) & h_e(j,N-3) & \cdots & h_e(j,0) \end{bmatrix} \tag{5-38}$$

根据上述模型，在已知 $g(x,y)$，$h(x,y)$ 和 $n(x,y)$ 的情况下，可以估算出理想的原始图像 $f(x,y)$。

5.2　线性滤波图像复原

图像复原就是根据给定的退化图像 g 和对退化系统 h 及噪声 n 的先验知识，寻找一个对原始图像 f 的最优估计图像 $\hat{f}$。

5.2.1　无约束图像复原

根据式 (5-34)，在复原模型中的噪声项为

$$\boldsymbol{n}=\boldsymbol{g}-\boldsymbol{hf} \tag{5-39}$$

在对 n 一无所知的情况下，希望找到一个估计图像 $\hat{f}$，使得其符合最小平方准则，即

$$\| \boldsymbol{n} \|^2 = \| \boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}} \|^2 \tag{5-40}$$

为最小。其中

$$\| \boldsymbol{n} \|^2 = \boldsymbol{n}\boldsymbol{n}' \tag{5-41}$$

$$\| \boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}} \|^2 = (\boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}})(\boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}})' \tag{5-42}$$

这里 $\boldsymbol{n}'$、$(\boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}})'$分别为 $\boldsymbol{n}$、$(\boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}})$ 的转置矩阵。

假设一目标函数为

$$J(\hat{\boldsymbol{f}}) = \| \boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}} \|^2 \tag{5-43}$$

要使得 $J(\hat{\boldsymbol{f}})$ 为最小，应将 $J(\hat{\boldsymbol{f}})$ 对 $\hat{\boldsymbol{f}}$求导数，并令其求导结果为零，即

$$\frac{\partial}{\partial \hat{\boldsymbol{f}}} J(\hat{\boldsymbol{f}}) = 2\boldsymbol{h}'(\boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}}) = 0 \tag{5-44}$$

式 (5-44) 的解为

$$\hat{\boldsymbol{f}} = (\boldsymbol{h}'\boldsymbol{h})^{-1}\boldsymbol{h}'\boldsymbol{g} \tag{5-45}$$

如果 $\boldsymbol{h}$ 为方阵，令 $\boldsymbol{h}^{-1}$存在，则式 (5-45) 为

$$\hat{\boldsymbol{f}} = \boldsymbol{h}^{-1}(\boldsymbol{h}')^{-1}\boldsymbol{h}'\boldsymbol{g} = \boldsymbol{h}^{-1}\boldsymbol{g} \tag{5-46}$$

这里选择的$\hat{\boldsymbol{f}}$除了要求 $J(\hat{\boldsymbol{f}})$ 为最小外不受任何其他约束，所以这种复原方法称为无约束图像复原。

5.2.2 约束图像复原

在约束图像复原中，除了满足式 (5-43) 外，还需满足约束条件，即假设 Q 是一个作用于 f 的线性运算，希望找到一个估计图像$\hat{f}$，使得目标函数

$$J(\hat{\boldsymbol{f}}) = \| \boldsymbol{Q}\hat{\boldsymbol{f}} \|^2 + \alpha(\| \boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}} \|^2 - \| \boldsymbol{n} \|^2) \tag{5-47}$$

为最小。这里 α 为一待定常数。

要使得 $J(\hat{\boldsymbol{f}})$ 为最小，应将 $J(\hat{\boldsymbol{f}})$ 对 $\hat{\boldsymbol{f}}$求导数，并令其结果为零，即

$$\frac{\partial}{\partial \hat{\boldsymbol{f}}} J(\hat{\boldsymbol{f}}) = 2\boldsymbol{Q}\boldsymbol{Q}'\hat{\boldsymbol{f}} - 2\alpha\boldsymbol{h}'(\boldsymbol{g} - \boldsymbol{h}\hat{\boldsymbol{f}}) = 0 \tag{5-48}$$

式 (5-48) 的解为

$$\hat{\boldsymbol{f}} = (\boldsymbol{h}'\boldsymbol{h} + \gamma\boldsymbol{Q}'\boldsymbol{Q})^{-1}\boldsymbol{h}'\boldsymbol{g} \tag{5-49}$$

式中 $\gamma = \frac{1}{\alpha}$。

这里选择的$\hat{\boldsymbol{f}}$除了要求 $J(\hat{\boldsymbol{f}})$ 为最小外，还要求选择 Q 及 γ，选择不同的 Q 及 γ 就会有不同的解，这种复原方法称为约束图像复原。

5.3 反向滤波图像复原

5.3.1 基本原理

反向滤波图像复原是无约束图像复原的一个应用例子。根据无约束图像复原的表达式

(5-46)，假设 $g(x, y)$，$f(x, y)$ 和 $h(x, y)$ 的傅立叶变换分别为 $G(u, v)$，$F(u, v)$ 和 $H(u, v)$，$H(u, v)$ 称为系统的滤波传递函数，对式 (5-46) 两边取傅立叶变换，可以得到下面的关系

$$F(u, v) = \frac{G(u, v)}{H(u, v)} \tag{5-50}$$

式中　$u, v = 0, 1, 2, \cdots, N-1$。

这样最优估计图像为

$$f(x, y) = F^{-1}(F(u, v)) = F^{-1}\left(\frac{G(u, v)}{H(u, v)}\right) \tag{5-51}$$

式中　$x, y = 0, 1, 2, \cdots, N-1$。

可见，若已知退化图像的傅立叶变换 $G(u, v)$ 和滤波传递函数 $H(u, v)$，根据式 (5-51) 可以求出复原后图像的傅立叶变换，再经过傅立叶逆变换求出复原后的图像 $f(x, y)$，这里 $1/H(u, v)$ 起了反向滤波的作用，所以这样的处理过程称为反向滤波图像复原。

在考虑噪声的情况下，反向滤波图像复原可以写成如下的形式

$$F(u, v) = F(u, v) + \frac{N(u, v)}{H(u, v)} \tag{5-52}$$

这里 $N(u, v)$ 是噪声 $n(x, y)$ 的傅立叶变换。

5.3.2　零点问题

从式 (5-52) 中可以看出，如果 uv 在平面上有一些点或区域的 $H(u, v)$ 趋近于零，$N(u, v)/H(u, v)$ 将趋近于无限大，就会导致方程出现不定的解，这样就为实际用反向滤波进行图像复原带来了困难，这种问题称为零点问题。

为了解决零点问题，若 $H(u, v)$ 的零点只在 uv 平面上的少数几个已知点上，那么在计算 $F(u, v)$ 时一般可以将它们忽略不计，这样做不会影响图像复原的结果。在大多数实际问题中，$H(u, v)$ 离开原点衰减得很快；而噪声项多在高频区，并衰减速度慢。为了回避零点问题，复原可局限于离原点不太远的有限区域内进行。图 5-2 示出了一个反向滤波的例子。图 5-2 (a) 是原始图像，图 5-2 (b) 是它的退化图像，选择滤波传递函数 $H(u, v) \approx G(u, v)$ (退化图像的傅立叶变换)，根据式 (5-51) 在 uv 平面原点附近进行复原的效果如图 5-2 (c) 所示，图 5-2 (d) 是在距离 uv 平面原点较远区域进行复原的效果。

5.3.3　消除因匀速直线运动引起的图像模糊

在拍摄照片时，由于景物与摄像机之间的相对运动而造成图像的模糊，比如在行走的汽车上拍摄地面景物的照片。对于这样的退化图像如何进行复原处理呢，这就是本节要介绍的内容。

假设图像 $f(x, y)$ 有一平面运动，并且某一 t 时刻在 x 方向的移动分量为 $x_0(t)$，在 y 方向的移动分量为 $y_0(t)$，那么原始图像在 t 时刻变为 $f[(x - x_0(t), y - y_0(t)]$，可见图像函数发生了变化，所以图像退化了。若曝光时间为 T，在 T 时间内由于这种平面移动而产生的退化图像为

$$g(x, y) = \int_0^T f[x - x_0(t), y - y_0(t)]\mathrm{d}t \tag{5-53}$$

根据傅立叶变换的定义，对 $g(x, y)$ 取傅立叶变换得

图 5-2 反向滤波的例子（选自 C.Rafael et al.）

$$G(u,v) = \iint_{-\infty}^{\infty} g(x,y)\exp[-j2\pi(ux+vy)]\,dx\,dy$$
$$= \iint_{\infty}\left\{\int_0^T f[x-x_0(t),\ y-y_0(t)]\,dt\right\}\exp[-j2\pi(ux+vy)]\,dx\,dy \tag{5-54}$$

对式 (5-54) 进行积分顺序交换

$$G(u,v) = \int_0^T\left\{\iint_{\infty} f[x-x_0(t),\ y-y_0(t)]\exp[-j2\pi(ux+vy)]\,dx\,dy\right\}dt \tag{5-55}$$

令 $x'=x-x_0(t)$，$y'=y-y_0(t)$，式 (5-55) 变为

$$G(u,v) = \int_0^T\left\{\iint_{\infty} f[x',y']\exp[-j2\pi(ux'+vy')]\,dx\,dy\right\}\exp\{-j2\pi[ux_0(t)+vy_0(t)]\}\,dt \tag{5-56}$$

并且

$$G(u,v) = \int_0^T F(u,v)\exp\{-j2\pi[ux_0(t)+vy_0(t)]\}\,dt$$
$$= F(u,v)\int_0^T\exp\{-j2\pi[ux_0(t)+vy_0(t)]\}\,dt \tag{5-57}$$

因为

$$G(u,v) = H(u,v)F(u,v) \tag{5-58}$$

将式 (5-57) 与式 (5-58) 对比，得出该退化系统的滤波传递函数为

$$H(u,v) = \int_0^T\exp\{-j2\pi[ux_0(t)+vy_0(t)]\}\,dt \tag{5-59}$$

然后就可以用反向滤波方法进行图像复原了。

下面针对一个例子进一步讨论一下如何处理零点问题。假设图像只在 x 方向做匀速直线运动，在曝光时间 T 内图像移动的总距离为 a。这样在 t 时刻图像的移动为

$$x_0(t) = \frac{at}{T} \tag{5-60}$$

$$y_0(t) = 0 \tag{5-61}$$

将式 (5-60) 和式 (5-61) 代入式 (5-59)，得到这个退化系统的滤波传递函数为

$$\begin{aligned} H(u, v) &= \int_0^T \exp[-\mathrm{j}2\pi u x_0(t)]\,\mathrm{d}t \\ &= \int_0^T \exp\left(-\mathrm{j}2\frac{\pi u a t}{T}\right)\mathrm{d}t \\ &= \frac{T}{\pi u a}\sin(\pi u a)\exp(-\mathrm{j}\pi u a) \end{aligned} \tag{5-62}$$

可见，当 $u = n/a$，n 为整数时，$H(u, v) = 0$，所以在进行复原时应取 $u \neq n/a$。

假设 $f(x, y)$ 在 $0 \leqslant x \leqslant L$ 内无零点，为了回避零点问题可以在此区间内做复原处理。这里是根据对退化模型的了解，用递推的方法复原出原始图像。由于 $y_0(t) = 0$，所以式 (5-53) 可以表示成

$$g(x) = \int_0^T f[x - x_0(t)]\,\mathrm{d}t = \int_0^T f\left(x - \frac{at}{T}\right)\mathrm{d}t \qquad (0 \leqslant x \leqslant L) \tag{5-63}$$

令 $\tau = x - \frac{at}{T}$，式 (5-63) 变为

$$g(x) = -\frac{T}{a}\int_{x-a}^{x} f(\tau)\,\mathrm{d}\tau \qquad (0 \leqslant x \leqslant L) \tag{5-64}$$

为了计算简便，忽略 $-\frac{T}{a}$ 项，式 (5-64) 变为

$$g(x) = \int_{x-a}^{x} f(\tau)\,\mathrm{d}\tau \qquad (0 \leqslant x \leqslant L) \tag{5-65}$$

对式 (5-65) 两侧进行微分，得

$$g'(x) = f(x) - f(x - a) \qquad (0 \leqslant x \leqslant L) \tag{5-66}$$

及

$$f(x) = g'(x) + f(x - a) \qquad (0 \leqslant x \leqslant L) \tag{5-67}$$

为了推导方便，设变量 x 为

$$x = z + ma \tag{5-68}$$

这里 z 的取值范围是 $[0, a]$，m 为 $\frac{x}{a}$ 的整数部分。如 $a = 2$，$x = 3.5$，则 $m = 1$，若取 $z = 1.5$，可得 $z + ma = 3.5$。

取 $L = Ka$，K 为整数，因为 $0 \leqslant x \leqslant L$，所以 m 可以取 0，1，2，…，$K - 1$，比如 $x = L$，$z = a$ 时，$m = K - 1$。将 $x = z + ma$ 代入式 (5-67) 中，得

$$f(z + ma) = g'(z + ma) + f(z + (m - 1)a) \tag{5-69}$$

令

$$\Phi(z) = f(z - a) \qquad (0 \leqslant z < a) \tag{5-70}$$

用递推的方法，根据式 (5-69) 可以得出下列式子，

当 $m = 0$ 时，

$$f(z) = g'(z) + f(z - a) = g'(z) + \Phi(z) \tag{5-71}$$

当 $m=1$ 时，

$$f(z+a)=g'(z+a)+f(z) \tag{5-72}$$

将式(5-71)代入式(5-72)得

$$f(z+a)=g'(z+a)+g'(z)+\Phi(z) \tag{5-73}$$

当 $m=2$ 时，

$$f(z+2a)=g'(z+2a)+f(z+a) \tag{5-74}$$

将式(5-73)代入(5-74)得

$$f(z+2a)=g'(z+2a)+g'(z+a)+g'(z)+\Phi(z) \tag{5-75}$$

依此类推，可以得出

$$f(z+ma)=\sum_{k=0}^{m}g'(z+ka)+\Phi(z) \tag{5-76}$$

将 $x=z+ma$ 代回式(5-76)中，

$$f(x)=\sum_{k=0}^{m}g'(x-(m-k)a)+\Phi(x-ma)\qquad(0\leqslant x\leqslant L) \tag{5-77}$$

在式(5-77)中，$g'[x-(m-k)a]$ 可以由退化图像求出，而 $\Phi(x-ma)$ 是未知的。下面介绍一种求 $\Phi(x-ma)$ 的近似方法，根据式(5-76)得

$$\Phi(z)=f(z+ma)-\sum_{k=0}^{m}g'(z+ka) \tag{5-78}$$

这里 $m=0$，1，2，…，$K-1$，可见根据式(5-78)可以得到 K 个方程式，将这 K 个方程式相加后得，

$$K\Phi(z)=\sum_{m=0}^{K-1}f(z+ma)-\sum_{m=0}^{K-1}\sum_{k=0}^{m}g'(z+ka) \tag{5-79}$$

式(5-79)两边除以 K 得

$$\Phi(z)=\frac{1}{K}\sum_{m=0}^{K-1}f(z+ma)-\frac{1}{K}\sum_{m=0}^{K-1}\sum_{k=0}^{m}g'(z+ka) \tag{5-80}$$

当 K 很大时，上式右边第一项可以近似为复原处理后图像 f 的灰度的平均值 A，A 是一个未知数，但可以用试验的方法来确定，这样式(5-80)可以表示为

$$\Phi(z)\approx A-\frac{1}{K}\sum_{m=0}^{K-1}\sum_{k=0}^{m}g'(z+ka) \tag{5-81}$$

将式(5-81)代入式(5-76)中，得

$$f(z+ma)\approx A+\sum_{k=0}^{m}g'(z+ka)-\frac{1}{K}\sum_{m=0}^{K-1}\sum_{k=0}^{m}g'(z+ka) \tag{5-82}$$

将 $z=x-ma$ 代入式(5-82)，得

$$f(x)\approx A+\sum_{k=0}^{m}g'(x-(m-k)a)-\frac{1}{K}\sum_{m=0}^{K-1}\sum_{k=0}^{m}g'(x-(m-k)a) \tag{5-83}$$

因为 k 的取值范围是 $0\sim m$，$m-k$ 的取值范围是 $m\sim 0$，这样上式可以等效为

$$\begin{aligned}f(x)&\approx A+\sum_{k=0}^{m}g'(x-ka)-\frac{1}{K}\sum_{m=0}^{K-1}\sum_{k=0}^{m}g'(x-ka)\\&\approx A-mg'(x-ma)+\sum_{k=0}^{m}g'(x-ka)\qquad(0\leqslant x\leqslant L)\end{aligned} \tag{5-84}$$

若加入变量 y，则式(5-84)变为

$$f(x, y) \approx A - mg'(x - ma, y) + \sum_{k=0}^{m} g'(x - ka, y) \qquad (0 \leqslant x, y \leqslant L) \tag{5-85}$$

这样就得到了运动模糊图像做复原处理后的图像，图 5-3 给出了根据上述进行处理的结果。

图 5-3　图像复原的例子 (选自 C. Rafael et al.)

5.4　最小二乘方约束图像复原

最小二乘方约束图像复原是约束图像复原的一个应用例子。对于约束图像复原，除了满足式 (5-43) 外，还应满足一定的约束条件。这里引入这样一个约束条件，即使函数的二阶导数满足最小平方准则。首先以一维情况为例来表示这种约束条件。

某一离散函数 $f(x)$，$x=0$，1，2，…，$M-1$，$f(x)$ 对 x 的二阶导数可以表示为

$$\frac{\partial^2 f(x)}{\partial x^2} \approx f(x+1) - 2f(x) + f(x-1) \tag{5-86}$$

最小平方的约束条件为

$$\left\{\sum_{x} [f(x+1) - 2f(x) + f(x-1)]^2\right\} \to \min \tag{5-87}$$

上式可以用矩阵表示成

$$(\boldsymbol{f}'\boldsymbol{C}'\boldsymbol{C}\boldsymbol{f}) \to \min \tag{5-88}$$

这里

$$\boldsymbol{C} = \begin{bmatrix} 1 & & & & & & & & \\ -2 & 1 & & & & & & & \\ 1 & -2 & 1 & & & & & & \\ & 1 & -2 & 1 & & & & & \\ & & & \cdots & \cdots & & & & \\ & & & & & 1 & -2 & 1 & \\ & & & & & & 1 & -2 & \\ & & & & & & & 1 & \end{bmatrix} \tag{5-89}$$

是光滑矩阵。

推广到二维的情况，最小平方的约束条件为

$$\left[\frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2}\right]^2 \rightarrow \min \tag{5-90}$$

其中

$$\left[\frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2}\right] \approx f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y) \tag{5-91}$$

上式可以在计算机中进行运算。

清单 5-1 给出了图像复原的 Java 语言实现的主函数，关于辅助函数的代码请见 CD 盘。

清单 5-1 Restore.java **源代码**

```
//Restore.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class Restore extends Frame {
Image im, tmp;
int iw, ih;
int[] pixels;

double [] newPixels;
double [] newKernel;

Complex [] complex;
Complex [] comKernel;

boolean flagLoad = false;
boolean flagBlur = false;

FFT2 fft2;
IFFT2 ifft2;

//构造方法
public Restore(){
    this.setTitle("图形模糊和复原");
CPanel pdown;
    Button load, blur, run, quit;
    //添加窗口监听事件
```

```
addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});

pdown = new Panel();
pdown.setBackground(Color.lightGray);

load=new Button("装载图像");
blur=new Button("图像模糊");
run=new Button("图像复原");
quit=new Button("退出");

this.add(pdown,BorderLayout.SOUTH);

pdown.add(load);
pdown.add(blur);
pdown.add(run);
pdown.add(quit);

load.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        jLoad_ActionPerformed(e);
    }
});

blur.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        jBlur_ActionPerformed(e);
    }
});

run.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        jRun_ActionPerformed(e);
    }
});

quit.addActionListener(new ActionListener(){
```

```
            public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });
    }

    public void jLoad_ActionPerformed(ActionEvent e){
        //利用 MediaTracker 跟踪图像的加载
        MediaTracker tracker = new MediaTracker(this);
        im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
        tracker.addImage(im,0);

        //等待图像的完全加载
        try{
        tracker.waitForID(0);
            }catch(InterruptedException e2){ e2.printStackTrace();}

            //获取图像的宽度 iw 和高度 ih
        iw=im.getWidth(this);
        ih=im.getHeight(this);
        pixels=new int[iw*ih];

        try{
        PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        flagLoad=true;
        repaint();
    }

    public void jBlur_ActionPerformed(ActionEvent e)
    {
        if(flagLoad){
        //对图像进行模糊处理
```

```
newPixels = new double [iw * ih];
newKernel = new double [iw * ih];

for(int i = 0;i < ih;i ++ )
{
    for(int j = 0;j < iw;j ++ )
    {
        newPixels[i * iw + j] = pixels[i * iw + j] &0xff;
        if((i < 5) && (j < 5))
        {
            newKernel[i * iw + j] = 1.0/25;

        }else{ newKernel[i * iw + j] = 0; }
    }
}
```

//初始化

```
complex = new Complex[iw * ih];
comKernel = new Complex[iw * ih];

for(int i = 0;i < iw * ih;i ++ )
{
    complex[i] = new Complex(0,0);
    comKernel[i] = new Complex(0,0);
}
```

//对原图像进行 FFT

```
fft2 = new FFT2();
fft2.setData(iw, ih, newPixels);
complex = fft2.getComplex();
```

//对卷积核进行 FFT

```
fft2 = new FFT2();
fft2.setData(iw, ih, newKernel);
comKernel = fft2.getComplex();
```

//频域相乘

```
for(int i = 0;i < iw * ih;i ++ )
{
    double re = complex[i].re * comKernel[i].re - complex[i].im * comKernel[i].im;
```

```
        double im=complex[i].re*comKernel[i].im +complex[i].im*comKernel[i].re;
        complex[i].re=re;
        complex[i].im=im;
    }

    //进行 FFT 反变换
    ifft2=new IFFT2();
    ifft2.setData(iw,ih,complex);
    newPixels=ifft2.getPixels();

    //归一化
    double max=newPixels[0];
    for(int i=1;i<iw*ih;i++)
    {
        if(max<newPixels[i])
        {
            max=newPixels[i];
        }
    }

    //System.out.println("max: "+max);

    ColorModel cm=ColorModel.getRGBdefault();

    for(int i=0;i<ih;i++)
    {
        for(int j=0;j<iw;j++)
        {
            int alpha=cm.getAlpha(pixels[i*iw+j]);
            int x=(int)(newPixels[i*iw+j]*255/max);

            pixels[i*iw+j]=alpha<<24|x<<16|x<<8|x;
        }
    }

    //将数组中的像素产生一个图像
    ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
    tmp=createImage(ip);

    repaint();
```

```
        flagBlur = true;

        }else{
        JOptionPane.showMessageDialog(null,"请先打开一幅图片!",
                        "Alert",JOptionPane.WARNING_MESSAGE);
        }

}

public void jRun_ActionPerformed(ActionEvent e){

        if(flagLoad & flagBlur){

        //对图像进行复原
        newPixels = new double [iw * ih];
        newKernel = new double [iw * ih];

        //初始化
        for(int i = 0;i < ih;i ++ )
        {
            for(int j = 0;j < iw;j ++ )
            {
                newPixels[i * iw + j] = pixels[i * iw + j] &0xff;
                if((i < 5) && (j < 5))
                {
                    newKernel[i * iw + j] = 1.0/25;

                }else{ newKernel[i * iw + j] = 0; }
            }
        }

        //初始化
        complex = new Complex[iw * ih];
        comKernel = new Complex[iw * ih];
        for(int i = 0;i < iw * ih;i ++ )
        {
            complex[i] = new Complex(0,0);
            comKernel[i] = new Complex(0,0);
        }
```

```
//对原图像进行 FFT
fft2 = new FFT2();
fft2.setData(iw, ih, newPixels);
complex = fft2.getComplex();

//对卷积核进行 FFT
fft2 = new FFT2();
fft2.setData(iw, ih, newKernel);
comKernel = fft2.getComplex();

//逆滤波复原
for(int i = 0; i < ih; i ++ )
{
    for(int j = 0; j < iw; j ++ )
    {
        double re = complex[i * iw + j].re;
        double im = complex[i * iw + j].im;
        double reKernel = comKernel[i * iw + j].re;
        double imKernel = comKernel[i * iw + j].im;
        double x = reKernel * reKernel + imKernel * imKernel;

        if(x > 1e - 3)
        {
            double r = (re * reKernel + im * imKernel)/x;
            double m = (im * reKernel - re * imKernel)/x;
            complex[i * iw + j].re = r;
            complex[i * iw + j].im = m;
        }

    }
}

//进行 FFT 反变换
ifft2 = new IFFT2();
ifft2.setData(iw, ih, complex);
newPixels = ifft2.getPixels();

//归一化
double max = newPixels[0];
for(int i = 1; i < iw * ih; i ++ )
```

```
        }
            if(max<newPixels[i])
            {
                max=newPixels[i];
            }
        }

        System.out.println("max: "+max);

        ColorModel cm=ColorModel.getRGBdefault();

        for(int i=0;i<ih;i++)
        {
            for(int j=0;j<iw;j++)
            {
                int alpha=cm.getAlpha(pixels[i*iw+j]);
                int x=(int)(newPixels[i*iw+j]*255/max);
                //int x=(int)newPixels[i*iw+j];

                pixels[i*iw+j]=alpha<<24|x<<16|x<<8|x;
            }
        }

        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);

        repaint();
    }else{
        JOptionPane.showMessageDialog(null,"请先打开一幅图片,并进行模糊处理!",
                    "Alert",JOptionPane.WARNING_MESSAGE);
        }
    }

    public void jQuit_ActionPerformed(ActionEvent e){
        //System.exit(0);
        JOptionPane op =new JOptionPane();
        int exit=op.showConfirmDialog(this,"你要退出吗???","退出",JOptionPane.YES_NO_OP-
TION);
```

```
        if(exit = = JOptionPane.YES_OPTION)
        {
            System.exit(0);

        }else{ }
    }

    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flagLoad){
            g.drawImage(tmp,10,20,this);
        }else { }
    }

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[ ] args) {
        Restore restore = new Restore();
        restore.setLocation(50,50);
        restore.setSize(500,400);
        restore.show();
    }
}
```

练 习 题

1 图像复原与图像增强的异同点是什么？

2 将第 1 章练习 2 中存入计算机内的图像用 Java 语言进行图像复原处理。

第 6 章　图像编码与压缩

数字图像信号的一个显著缺点是数据量太大，无论是进入计算机还是保存其数据都是困难的，特别是传输图像时，数字图像的频带很宽，这就给图像传输和存储带来了相当大的困难。那么就提出了这样一个问题：在保证一定图像质量的条件下，能否减少存储图像所需的比特数？最少能减少到多少？用什么样的编码方法可以使比特数减少？这些就是本章要讨论的问题。

6.1　基本概念

6.1.1　图像编码

所谓数字图像就是对连续图像进行取样和量化后，将原来的连续变量变为数字变量的图像，数字图像中的数字变量在送入计算机前要将它们变成二进制的数字信号，这一过程就称为图像编码，如图 6-1 所示。

1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
2	2	2	2	2
1	1	1	1	1

(a) 量化后灰度图

01	01	01	01
10	10	10	10
11	11	11	11
10	10	10	10
01	01	01	01

(b) 编码

图 6-1　图像编码示意

6.1.2　冗余信息和不相关信息

数字图像信号的数据之所以能够在保证图像质量的前提下被压缩，原因是在数字图像数据中包含了大量的冗余信息和不相干信息。那么什么是冗余信息和不相关信息呢？对此通过一个例子加以说明。

【例 6-1】　假设某一旅行者正在旅途中，现在他正在亚洲某国家的旅馆内，还没有安排好回家的路线，这时他收到了下面的消息：

“你的妻子，Lina，将于明天晚上 7 点 10 分在巴黎的 roissy charles de gaulle 机场接你”。

这是一则消息，但它不是最简练的消息，也就是说这一消息中的数据是可以被压缩的。怎样才能使这条消息即简短又清楚，同时又不损失信息呢？让我们来分析一下它所包含的信息，首先它包含了旅行者已经知道的一些内容，比如他一定知道 roissy charles de gaulle 机场位于巴黎，以及他妻子的名字，删除掉这些内容，上面的消息就可以写成：

“你的妻子将于明天晚上 7 点 10 分在 roissy charles de gaulle 机场接你”。

这则消息被压缩处理后没有造成信息的损失，这时被删除的这些内容称为冗余信息。进一步还可以在不造成严重信息损失的前提下继续压缩该消息，比如上述进一步压缩为：

“Lina 将于明晚 7 点在 de gaulle 接你”。

通过上述处理后，信息的准确度受到了影响，但不是很严重。比如旅行者必须猜想 Lina 是否指他的妻子，de gaulle 是否巴黎的飞机场，以及他将焦急地等待 10 分钟，因为他到达的时间被量化成整数了，他可能会想 Lina 是否出意外了，如生病了、出车祸了……但经过这样的处理后没有造成严重的信息损失，这时被删除的这些内容称为不相关信息。

6.1.3 数据的压缩

由上述可见，因为在图像数据中存在冗余信息和不相关信息，所以图像数据是可以被压缩的。这里主要是因为图像中的样点之间、行列之间都存在较强的相关性。从统计观点出发，就是某一个样点的灰度值总是和周围其他样点的灰度值有某种依赖关系，比如：临近两点几乎有同样的灰度值。那么应用某种编码方法就可以减少这些相关性。另外，人的视觉的分辨能力是有限的，这样可以去掉一些不影响视觉质量的部分等。利用图像数据固有的冗余度和不相关性，将一个大的数据文件转换成较小的文件的处理称为数据压缩。

图像数据的压缩一般分为无损压缩和有损压缩。所谓无损压缩是删除了图像数据中的冗余度信息，在解除压缩后可以对图像进行精确地复原。有损压缩是把图像数据中的冗余信息和不相干的信息都删除掉了，因此只能对原图像进行近似的重构，而不能精确地复原。前已述及，图像数据压缩的目的是在满足一定图像质量的条件下，尽可能减少比特数，那么，怎样来衡量图像的质量呢？

6.2 图像质量的衡量准则

图像质量的衡量准则称为保真度准则，它是用来衡量图像编码方法或系统质量的优劣程度的方法。通常这种衡量准则可分为客观保真度准则和主观保真度准则。

6.2.1 客观保真度准则

1. 输入图像和输出图像的均方根误差

假设输入图像 $f(x, y)$ 是由 $N \times N$ 个像素点组成，即 $x, y = 0, 1, 2, \cdots, N-1$。$f(x, y)$ 经过压缩编码处理后，再经过解码并重建原图像，重建的图像为 $g(x, y)$，是由 $N \times N$ 个像素点组成的，即 $x, y = 0, 1, 2, \cdots, N-1$。对于任一对 x 和 y，输入像素的灰度 $f(x, y)$ 和输出像素的灰度 $g(x, y)$ 之间的误差为

$$e(x, y) = g(x, y) - f(x, y) \tag{6-1}$$

因为图像是由 $N \times N$ 个像素点组成的，考虑 $N \times N$ 个像素点，总的均方误差为

$$\begin{aligned} \overline{e^2} &= \frac{1}{N^2} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} e^2(x, y) \\ &= \frac{1}{N^2} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} [g(x, y) - f(x, y)]^2 \end{aligned} \tag{6-2}$$

根据式 (6-2)，得均方根误差为

$$e_{rms} = \sqrt{\overline{e^2}} \tag{6-3}$$

在讨论图像编码方法中，可根据 e_{rms} 的大小来表示编码方法 (或系统) 的优劣。

2. 输入图像和输出图像的均方根信噪比

如果把输入图像与输出图像间的误差看作噪声 $e(x, y)$，重建图像 $g(x, y)$ 可表示为

$$g(x, y) = f(x, y) + e(x, y) \tag{6-4}$$

这样重建图像的均方信噪比为

$$\left(\frac{S}{N}\right)_{ms} = \frac{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} g^2(x, y)}{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} e^2(x, y)} \tag{6-5}$$

根据式 (6-5)，均方根信噪比为

$$\left(\frac{S}{N}\right)_{rms} = \sqrt{\frac{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} g^2(x, y)}{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} e^2(x, y)}} \tag{6-6}$$

它也是一个表征编码方法 (或系统质量) 好坏的标准。

6.2.2　主观保真度准则

图像处理的结果绝大部分是给人看的，由研究人员来解释。因此图像质量的好坏与否除了与图像本身的客观质量有关外，也与人的视觉系统的特性有关。有时候客观保真度完全一样的两幅图像可能会有完全不同的视觉效果。因此，为了对图像质量有一个更全面的评价，还需要有一个主观评定标准——主观保真度准则。这种方法是把图像显示给观察者，然后将评价结果加以平均，以此来评价一幅图像的主观质量。主观保真度准则通常有三种方法：绝对尺度、妨害准则和品质准则。

1. 绝对尺度

在绝对尺度准则下的评价标准如下：

(a) 优秀的：具有较高质量的图像。

(b) 好的：可供观赏的高质量的图像，干扰并不令人讨厌。

(c) 可通过的：图像的质量可以接受，干扰并不讨厌。

(d) 边缘的：图像的质量比较低，希望能加以改善，干扰有些讨厌。

(e) 劣等的：图像质量很差，尚能观看，干扰显著令人讨厌。

(f) 不能用：图像质量非常差，无法观看。

2. 妨害准则

妨害准则的评价标准是：

(a) 没有妨害感觉。

(b) 有妨害，但不讨厌。

(c) 能感觉到妨害，但没有干扰。

(d) 妨害严重，并有明显干扰。

(e) 不能接收信息。

3. 品质准则

在品质准则中的评价标准是：

(a) 非常好。
(b) 好。
(c) 稍好。
(d) 普通。
(e) 稍坏。
(f) 恶劣。
(g) 非常恶劣。

此外，还有一些其他的方法，也就是说主观保真度评价方法的准则可以不同，但基本原理都是一样的。

6.3 图像编码过程

图像的编码过程大致可以分为三部分：图像数据的转换，图像数据的量化和图像数据的编码，如图 6-2 所示。

图 6-2 图像编码过程示意图

6.3.1 图像数据转换

图像数据的转换是根据图像像素间的相关性将输入的原始像素组合转换成另一种形式，以减少数据的比特数，比如作如下的线性变换

$$\begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \tag{6-7}$$

式 (6-7) 也可以表示为

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} \tag{6-8}$$

这里 x 为输入的图像数据，A 为转换矩阵，y 为转换后的数据。在下面的例子中可以看出，y 比 x 具有较少的比特数。

【例 6-2】 A 取如下的形式

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix}$$

这里第一个 y 值为 $y_1 = x_1$，其他为 $y_i = x_{i-1} - x_1$。如果邻近样点间的灰度差很小，那么用 $y_i = x_{i-1} - x_i$ 表示图像数据所需的比特数将统计意义地小于由原像素所表示的图像数据。

当然上述只是一个典型的例子，在实际应用中可以采用其他不同的方法。

6.3.2　图像数据量化

经过 6.3.1 的图像数据转换后，图像数据由 $\{y_i\}$ 表示，将 $\{y_i\}$ 作为输入值，根据量化理论，把 $\{y_i\}$ 的取值范围量化成一些离散的区间，每个区间有一个量化输出值 $\{w_i\}$，如图 6-3 所示。

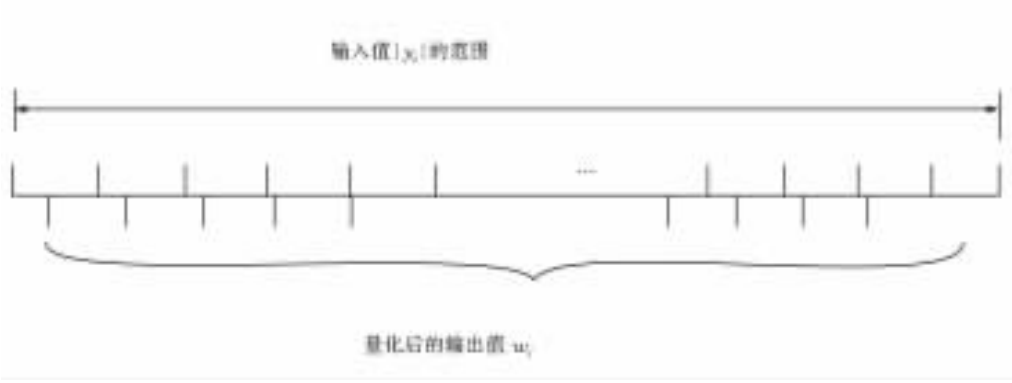


图 6-3　数据量化

6.3.3　图像数据编码

在 6.3.1 和 6.3.2 中的图像数据转换中，为了减少图像的数据量，将图像数据 $\{x_i\}$ 换成 $\{y_i\}$ 的形式，然后对 $\{y_i\}$ 进行了量化处理，输出的量化值为 $\{w_i\}$，那么下一步就应该为量化输出值 $\{w_i\}$ 编码了，输出值的代码为 $\{c_i\}$，图 6-4 示出了这个过程。

$$\begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} \rightarrow \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_n \end{bmatrix} \rightarrow \begin{bmatrix} c_1 \\ c_2 \\ \cdots \\ c_n \end{bmatrix}$$

图 6-4　编码过程示意图

如果量化器输出了 M 个值，那么对应于 M 个值中的任何一个值编码器将给出一个二进制码字，编码器的输入和输出是一一对应的。码字有两种形式，即等长代码和非等长代码。所谓等长代码，是指每个码字都具有相同的比特数，如自然二进制码。而非等长代码中的每个码字不具有相同的比特数。假如有八个输入代码（w_1，w_2，w_3，w_4，w_5，w_6，w_7，w_8），现在用自然二进制等长代码对其进行编码，结果为 $c_1=000$，$c_2=001$，$c_3=010$，$c_4=011$，$c_5=100$，$c_6=101$，$c_7=110$，$c_8=111$；对此也可以给出非等长代码，如 $c_1=00$，$c_2=01$，$c_3=10$，$c_4=1100$，$c_5=1101$，$c_6=1110$，$c_7=111100$，$c_8=111101$，见表 6-1。

表 6-1 编 码

输 入	等长代码	非等长代码
w_1	000	00
w_2	001	01
w_3	010	10
w_4	011	1100
w_5	100	1101
w_6	101	1110
w_7	110	111100
w_8	111	111101

6.4 统计编码

统计编码是一种非等长代码，它是根据像素灰度出现概率的分布特性而进行的压缩编码。这里将介绍两种常用的统计编码，即香农-费诺码和霍夫曼编码。

6.4.1 基本概念

1. 信息量与信息商

信息是指对消息接受者来说预先不知道的报道。从表面上看信息似乎是一种不可度量的抽象量，但如果从概率统计的角度分析，假设由某一消息源发出的一系列消息 M 为

$$M = \{m_1, m_2, m_3, \cdots, m_N\} \tag{6-9}$$

其中某一消息 m_k 出现的可能性——概率——为 p_k，因为概率 p_k 是可以度量的，这样信息就成为可以度量的量了。假设某一事件 m_i 出现的概率为 1，记 $p_i = 1$，那么这个事件就称为必然事件。因为必然事件是预先可以知道的，所以从必然事件中消息接受者没有得到任何信息，或者说信息量为零。

在信息论中对信息量的定义为

$$I(m_k) = -\log_2 p(m_k) \quad \text{(比特)} \tag{6-10}$$

这里，$I(m_k)$ 为消息接收者从消息 m_k 中得到的信息量，$p(m_k)$ 为 m_k 出现的概率，式中的负号是保证 $I(m_k)$ 的数值不为负。

若有一系列消息 $M = \{m_1, m_2, m_3, \cdots, m_N\}$，它们出现的概率分别为 $P = \{p_1, p_2, p_3, \cdots, p_N\}$，则由这些消息提供的平均信息量为

$$\begin{aligned} H(m) &= \sum_{k=1}^{N} p(m_k) I(m_k) \\ &= \sum_{k=1}^{N} p(m_k) \log_2 p(m_k) \quad \text{(比特/消息)} \end{aligned} \tag{6-11}$$

$H(m)$ 表示平均每个消息提供的信息量，称为信息熵。

2. 图像商

将信息熵的概念推广到图像中，把信息源看成是数字图像的灰度源，即每个像素的灰度 w_k 作为一个消息，其灰度出现的概率为 p_k，这样就得到了图像熵的定义。

设图像像素的灰度集合为 $\{w_1, w_2, w_3, \cdots, w_N\}$，灰度出现的概率为 $\{p_1, p_2, p_3, \cdots, p_N\}$，则图像熵 $H(w)$ 为

$$H(w) = -\sum_{k=1}^{N} p(w_k)\log_2 p(w_k) \quad \text{(比特/像素)} \tag{6-12}$$

表示平均每个像素提供的信息量。图像商提供了一幅图像中包含的平均信息量，这就意味着在图像数据压缩过程中，为了不造成图像信息的损失，压缩后图像中的信息量不应该小于图像商。图像数据编码的平均码字位数与图像中的信息量是相对应的，若一幅图像的灰度级为 $\{w_1, w_2, w_3, \cdots, w_N\}$，灰度出现的概率为 $\{p_1, p_2, p_3, \cdots, p_N\}$，在进行压缩编码时所需的平均码字的位数不得小于图像商，可见图像商提供了进行编码时所需的码字平均位数的下限。

【例 6-3】　假设某一数字图像的灰度级和概率分布如下：

$$x = \begin{Bmatrix} w_1 & w_2 & w_3 & w_4 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{8} \end{Bmatrix}$$

根据式 (6-12)，图像商为

$$\begin{aligned} H(w) &= -\sum_{k=1}^{4} p(w_k)\log_2 p(w_k) \\ &= -\frac{1}{2}\log_2\left(\frac{1}{2}\right) - \frac{1}{4}\log_2\left(\frac{1}{4}\right) - \frac{1}{8}\log_2\left(\frac{1}{8}\right) - \frac{1}{8}\log_2\left(\frac{1}{8}\right) \\ &= \frac{7}{4} \quad \text{(比特/像素)} \end{aligned}$$

上述计算结果表明，在编码中为了不造成图像信息的损失，平均每个灰度所需要的最小比特为$\frac{7}{4}$，也可以说进行编码时所需的码字平均位数值的下限为$\frac{7}{4}$。

3. 平均码字长度

假设一幅数字图像编码的码字为 $\{c_1, c_2, c_3, \cdots, c_N\}$，对应的码字长分别为 $\{\beta_1, \beta_2, \beta_3, \cdots, \beta_N\}$，则编码所需的平均位数为

$$\overline{N} = \sum_{k=1}^{N} \beta_k p_k \quad \text{(比特/像素)} \tag{6-13}$$

4. 编码效率

通常编码效率为

$$\eta = \frac{H}{\overline{N}} \times 100\% \tag{6-14}$$

可见，$\overline{N}$越接近图像商 H 的数值，编码效率越高。

5. 冗余度

冗余度定义为

$$R_d = 1 - \eta \tag{6-15}$$

在编码中应尽量使编码效率趋近于 1，从而使冗余度趋近于 0。

6. 变长最佳编码定理

前面已经介绍了有两种图像编码：等长编码和变长编码。在变长编码中应根据变长最佳编码定理来编码，其最佳编码定理是对出现概率大的消息符号赋予短码字，而对于出现概率小的消息符号赋予长码字。

【例 6-4】 假设某一数字图像的灰度级和概率分布如下，

$$x=\begin{Bmatrix} w_1 & w_2 & w_3 & w_4 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{8} \end{Bmatrix}$$

分别采用等长码和变长码进行编码，编码结果如表 6-2 所示。

表 6-2 编 码 结 果

灰度级	概率变长码	变长码	等长码
w_1	$\frac{1}{2}$	00	0
w_2	$\frac{1}{4}$	01	10
w_3	$\frac{1}{8}$	10	110
w_4	$\frac{1}{8}$	11	111

下面分别计算表 6-2 中所示的等长码与变长码的编码效率。

对于等长码：

$$\begin{aligned} H(w) &= -\sum_{k=1}^{4} p(w_k)\log_2 p(w_k) \\ &= -\frac{1}{2}\log_2\left(\frac{1}{2}\right)-\frac{1}{4}\log_2\left(\frac{1}{4}\right)-\frac{1}{8}\log_2\left(\frac{1}{8}\right)-\frac{1}{8}\log_2\left(\frac{1}{8}\right) \\ &= \frac{7}{4} \quad \text{(比特/像素)} \end{aligned}$$

$$\overline{N}=\sum_{k=1}^{4}\beta_k p_k = 2\times\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{8}\right)=2$$

$$\eta=\frac{H}{\overline{N}}\times 100\%=\frac{7}{4}\Big/2=\frac{7}{8}$$

$$R_d=1-\eta=1-\frac{7}{8}=\frac{1}{8}$$

可见等长编码没有达到最佳效果，如何使编码的效率更高呢？下面计算下变长编码的效率：

$$\begin{aligned} H(w) &= -\sum_{k=1}^{4} p(w_k)\log_2 p(w_k) \\ &= -\frac{1}{2}\log_2\left(\frac{1}{2}\right)-\frac{1}{4}\log_2\left(\frac{1}{4}\right)-\frac{1}{8}\log_2\left(\frac{1}{8}\right)-\frac{1}{8}\log_2\left(\frac{1}{8}\right) \\ &= \frac{7}{4} \quad \text{(比特/像素)} \end{aligned}$$

$$\overline{N}=\sum_{k=1}^{4}\beta_k p_k = \frac{1}{2}+2\times\frac{1}{4}+3\times\frac{1}{8}+3\times\frac{1}{8}=\frac{7}{4}$$

$$\eta=\frac{H}{\overline{N}}\times 100\%=\left(\frac{7}{4}\Big/\frac{7}{4}\right)\times 100\%=100\%$$

$$R_d=1-\eta=1-1=0$$

可见这里变长编码的效率更好。

6.4.2 香农-费诺码

香农-费诺码是一种变长压缩编码，它的编码规则如下：

⑴ 设信源有非递增的概率分布，即

$$X=\begin{Bmatrix} w_1 & w_2 & w_3 & \cdots & w_N \\ p_1 & p_2 & p_3 & \cdots & p_N \end{Bmatrix} \tag{6-16}$$

式中　$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_N$。

把 X 分成两个子集合，得

$$X_1=\begin{Bmatrix} w_1 & w_2 & w_3 & \cdots & w_k \\ p_1 & p_2 & p_3 & \cdots & p_k \end{Bmatrix} \tag{6-17}$$

$$X_2=\begin{Bmatrix} w_{k+1} & w_{k+2} & w_{k+3} & \cdots & w_N \\ p_{k+1} & p_{k+2} & p_{k+3} & \cdots & p_N \end{Bmatrix} \tag{6-18}$$

并保证

$$\sum_{i=1}^{k} p_i \approx \sum_{i=k+1}^{N} p_i \tag{6-19}$$

成立或近似成立。

⑵ 给两个子集中的消息赋值，X_1 赋 1、X_2 赋 0，或反之。

⑶ 重复第一步，将两个子集 X_1 和 X_2 再分别细分为两个小子集，使每对的两个小子集的概率之和相等或近似相等；然后重复第二步，给各个小子集中的消息赋值，以这样的步骤重复下去，直至每个子集内只包含一个消息为止，对每个消息所赋过的值依次排列出来就可以构成香农-费诺码。

【例 6-5】　假设某一图像的灰度分布及对应的概率如下所示，试对其进行香农-费诺码编码。

$$X=\begin{Bmatrix} w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{8} & \frac{1}{8} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \end{Bmatrix}$$

图 6-5 是编码流程图。

码　字	灰　度	概　率	第一步	第二步	第三步	第四步
00	w_1	$\frac{1}{4}$	0	0		
01	w_2	$\frac{1}{4}$		1		
100	w_3	$\frac{1}{8}$	1	0	0	
101	w_4	$\frac{1}{8}$			1	
1100	w_5	$\frac{1}{16}$		1	0	0
1101	w_6	$\frac{1}{16}$				1
1110	w_7	$\frac{1}{16}$			1	0
1111	w_8	$\frac{1}{16}$				1

图 6-5　香农-费诺码编码流程图

6.4.3 霍夫曼编码

霍夫曼编码也是一种变长压缩编码，它的编码规则如下：

假设原始信源有 N 个消息，即

$$X=\begin{Bmatrix} w_1 & w_2 & w_3 & \cdots & w_N \\ p_1 & p_2 & p_3 & \cdots & p_N \end{Bmatrix}$$

式中 p_i 为 w_i 出现的概率。

可用下述步骤编出霍夫曼码：

⑴ 把信源 X 中的消息按出现的概率从大到小的顺序排列，即

$$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_N。$$

⑵ 把最后两个出现概率最小的消息合并成一个消息，从而使信源的消息数减少一个，并同时再将信源中的消息再按概率从大到小排列，得

$$X'=\begin{Bmatrix} w_1' & w_2' & w_3' & \cdots & w'_{N-1} \\ p_1' & p_2' & p_3' & \cdots & p'_{N-1} \end{Bmatrix}$$

式中 $p'_1 \geqslant p'_2 \geqslant \cdots \geqslant p'_{N-1}$。

⑶ 重复上述步骤，直到信源最后为 X^0 的形式为止。

$$X^0=\begin{Bmatrix} w_1^0 & w_2^0 \\ p_1^0 & p_2^0 \end{Bmatrix}$$

式中 $p_1^0 \geqslant p_2^0$。

⑷ 分配码字。码字分配从最后一步开始反向进行，对最后的两个概率一个赋予 0，另一个赋予 1。如此反向进行到开始的概率排列为止。在此过程中，若概率不变仍用原码字。若概率分裂为两个，其码字前几位仍用原来的，码字的最后一位码字一个赋予 0、另一个赋予 1。

【例 6-6】 假设某一图像的灰度分布及对应的概率如下所示，试对其进行霍夫曼编码。

$$X=\begin{Bmatrix} w_1 & w_2 & w_3 & w_4 & w_5 & w_6 \\ 0.25 & 0.25 & 0.20 & 0.15 & 0.10 & 0.05 \end{Bmatrix}$$

霍夫曼编码流程如图 6-6 所示。首先将此排列中最小的两个概率相加，形成一个新的概率分布（如图 6-6 中的第二步所示），然后重复上述过程直到只有两个概率值为止（如图 6-6 的第二步到第五步所示）。下面的过程是对它们进行编码，这时从第五步开始反向进行，为 0.55 赋 0（或 1），为 0.45 赋 1（或 0），反向进入第四步，0.45 的概率不变仍用原码字为 1，而 0.55 分解为 0.30 和 0.25，0.30 在原来代码的基础上加 0 成为 00，0.25 在原来代码的基础上加 1 成为 01。再反向进入第三步，0.30 的概率不变仍用原码字为 00，0.25 的概率不变仍用原码字为 01，而 0.45 分解为 0.25 和 0.20，0.25 在原来代码的基础上加 0 成为 10，0.20 在原来代码的基础上加 1 成为 11。然后再反向进入第二步，上面的 0.25 的概率不变仍用原码字 01，其次的 0.25 概率不变仍用原码字 10，0.20 的概率不变仍用原码字 11，而 0.30 分解为 0.15 和 0.15，上面的 0.15 在原来代码的基础上加 0 成为 000，下面的 0.15 在原来代码的基础上加 1 成为 001。反向进入第一步，上面的 0.25，0.25，0.20，0.15 的代码保持不变，只有 0.10 和 0.05 是由 0.15 分解得到的，0.10 的代码为 0010，0.05 的代码为 0011。

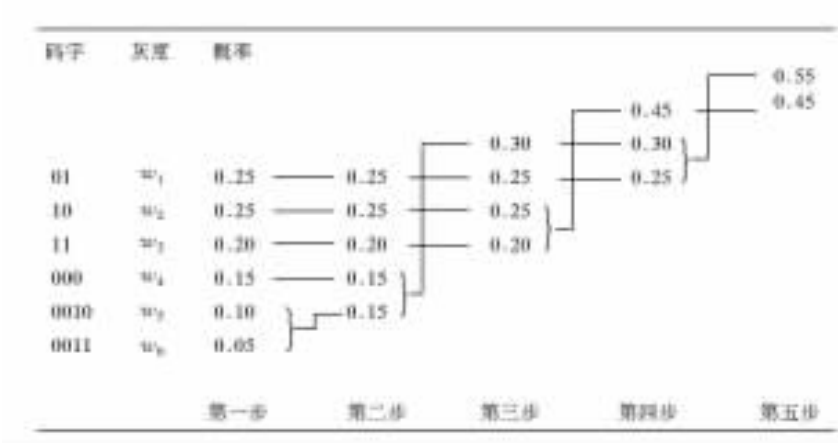


图 6-6　霍夫曼编码示意图

程序清单 6-1 给出了 Java 语言实现的霍夫曼编码程序清单。

程序清单 6-1　　Huffman.java 源代码

```
//Huffman.java

public class Huffman {
int ii;
//灰度等级数
int ColorNum=256;
//图像的灰度
int [] imageData;
//图像的宽度和高度
int iw,ih;
//各个灰度的概率
float []freq=new float[ColorNum];
//中间数组
float []freqTemp;
//映射关系的数组
int []map;
//图像熵
float entropy=0;
//huffman 编码表
String [] sCode=new String[ColorNum];;
//平均码长
float avgCode=0;
//编码效率
float efficiency=0;
```

```
public Huffman(int data[],int iw,int ih){

    imageData=new int[iw*ih];
    for(int i=0;i<iw*ih;i++)
    {
        imageData[i]=data[i];
    }

    this.iw=iw;
    this.ih=ih;

    //初始化
    for(int i=0;i<ColorNum;i++)
    {
        sCode[i]="";
    }

    //初始化
    for(int i=0;i<ColorNum;i++)
    {
        freq[i]=0;
    }
    //各个灰度的计数
    for(int i=0;i<iw*ih;i++)
    {
        int temp=imageData[i];
        freq[temp]=freq[temp]+1;
    }
    //各个灰度出现的频率
    for(int i=0;i<ColorNum;i++)
    {
        freq[i]=freq[i]/(iw*ih);
    }

    //计算图像熵
    for(int i=0;i<ColorNum;i++)
    {
        if(freq[i]>0)
        {
```

```
                entropy -= freq[i] * Math.log(freq[i])/Math.log(2.0);
            }
        }
    }

    //huffman 编码
    public void huff(){

        freqTemp=new float[ColorNum];
        map=new int[ColorNum];

        for(int i=0;i<ColorNum;i++)
        {
            freqTemp[i]=freq[i];
            map[i]=i;
        }
        //冒泡排序
        for(int j=0;j<ColorNum-1;j++)
        {
            for(int i=0;i<ColorNum-j-1;i++)
            {
                if(freqTemp[i]>freqTemp[i+1])
                {
                    float temp=freqTemp[i];
                    freqTemp[i]=freqTemp[i+1];
                    freqTemp[i+1]=temp;

                    //更新映射关系
                    for(int k=0;k<ColorNum;k++)
                    {
                        if(map[k]==i)
                        {
                            map[k]=i+1;
                        }else if(map[k]==i+1)
                        {
                            map[k]=i;
                        }
                    }
                }
            }
```

```
}
//开始编码
for(ii=0;ii<ColorNum-1;ii++)
{
    if(freqTemp[ii]>0)
    {
        break;
    }
}

//
for(;ii<ColorNum-1;ii++)
{
    for(int k=0;k<ColorNum;k++)
    {
        if(map[k]==ii)
        {
            sCode[k]="1"+sCode[k];
        }
        else if(map[k]==ii+1)
        {
            sCode[k]="0"+sCode[k];
        }
    }
    //最小的两个相加
    freqTemp[ii+1]+=freqTemp[ii];
    //改变映射关系
    for(int k=0;k<ColorNum;k++)
    {
        if(map[k]==ii)
        {
            map[k]=ii+1;
        }
    }
    //重新排序
    for(int j=ii+1;j<ColorNum-1;j++)
    {
        if(freqTemp[j]>freqTemp[j+1])
        {
            float temp=freqTemp[j];
```

```
                    freqTemp[j] = freqTemp[j + 1];
                    freqTemp[j + 1] = temp;
                    //更新映射关系
                    for(int k = 0; k < ColorNum; k ++ )
                    {
                        if(map[k] = = j)
                        {
                            map[k] = j + 1;
                        }else if(map[k] = = j + 1)
                        {
                            map[k] = j;
                        }
                    }
                }
            else {
                break;
                }
            }

        }
        //计算编码的平均长度
        avgCode = 0;
        for(int i = 0; i < ColorNum; i ++ )
        {
            avgCode = avgCode + freq[i] * (sCode[i].length());
        }
        //计算编码效率
        efficiency = entropy/avgCode;

    }

    public float getEntropy()
    {
        return entropy;
    }

    public float getAvgCode()
    {
        return avgCode;
    }
```

```
public float getEfficiency()
{
    return efficiency;
}

public float[] getFreq()
{
    return freq;
}

public String [] getCode()
{
    return sCode;
}

public void test(){

    System.out.println("the iw:" + iw + "the ih" + ih);
}

}
```

练 习 题

1 **将第一章练习 2 中存入计算机内的图像用 Java 语言进行霍夫曼编码，并计算图像商和编码效率。**

2 **一图像如下所示，**

1	1	1	1	1	1	1	1
1	3	3	3	3	3	3	1
1	3	2	2	2	2	3	1
1	3	7	4	4	0	3	1
1	3	0	4	4	7	3	1
1	3	2	2	2	2	3	1
1	3	3	3	3	3	3	1
1	1	1	1	1	1	1	1

试对其进行霍夫曼编码，并计算图像商和编码效率。

第 7 章　图像的分割与描述

前面所讨论的方法是从一幅图像的整体来进行数字图像处理。目的是使输出图像成为人们所需要的输入图像的一种改进方式，使经过处理后的图像和原始图像尽可能相似。

图像处理还有另一个重要的分支称为图像分割，它的输出不一定是一幅完整的图像，可能是图像的某些特征描述，其目的是从一幅图像中找出所需要的目标，或将图像划分成性质不同的若干区域。例如，从侦察卫星所拍摄的照片中找出军事目标，从卫星所拍摄的照片中区分出森林和农田区等，这些都是图像分割的任务。

7.1　图像的分割

所谓图像分割，就是把图像空间按照一定的要求分成一些“有意义”区域的技术。下面介绍几种常用的方法。

7.1.1　灰度阈值分割法

1. 基本原理

从一幅图像中找出目标最常用的方法是在该图像中确定区分目标与背景的分界点——阈值，例如一幅图像 $f(x, y)$ 的灰度分布范围是 $[z_1, z_K]$，令 T 为 z_1 与 z_K 之间的任意值，选取 T 为阈值可将图像变为

$$g_T(x, y) = \begin{cases} 1 & f(x, y) \geqslant T \\ 0 & f(x, y) < T \end{cases} \tag{7-1}$$

在这个新图像 $g_T(x, y)$ 中，原图像 $f(x, y)$ 中灰度值小于阈值 T 的像素点的灰度值变为 0，而其他像素点的灰度值变为 1，形成了一个二值图像。

阈值的选取方法还有许多，例如

$$g_T(x, y) = \begin{cases} 1 & f(x, y) \leqslant T \\ 0 & f(x, y) > T \end{cases} \tag{7-2}$$

也可以将阈值设置为一个灰度范围 $[T_1, T_2]$，凡是灰度在此范围内的像素点的新灰度值为 1，其他像素点的新灰度值均为 0，即

$$g_{T_1, T_2}(x, y) = \begin{cases} 1 & T_1 \leqslant f(x, y) \leqslant T_2 \\ 0 & \text{其　他} \end{cases} \tag{7-3}$$

可见阈值的选取直接影响图像分割的效果。图 7-1 示出了选取不同的阈值形成的二值图像。

假设某一图像中，凡是属于“目标”部分的像素点的灰度级都比较高，而属于“背景”部分像素点的灰度级都比较低，这样在“目标”与“背景”两部分间可能存在一个灰度的边界，这个边界就可以作为灰度的阈值，根据式 (7-1) 或式 (7-2) 就形成了一个目标与背景分界清晰的二值图像。由此可见，根据图像像素点的灰度级将图像空间划分成一些区域，在这些区域内部其特性是相同的，或者说是均匀的，两个相邻区域彼此特性是不相同的，其间存在

着边界，这个边界称为灰度阈值，这种方法称为灰度阈值分割法。如果能够通过某种科学的方法求出一个阈值 T，并认为灰度值大于 T 的点属于目标点，而灰度值小于 T 的点是背景点，那么就可以从一整幅图像中找出目标部分了。

图 7-1　二值图像

2. 阈值的选取

(1) 简单寻谷法。这种方法通常在没有任何先验知识的前提下使用。在简单寻谷法中需要借用直方图来进行，这里用 r 表示图像样点的灰度，p_r 表示灰度 r 出现的概率，将每个 p_r 与 p_{r-1} 和 p_{r+1} 相比较，并规定若

$$p_r < p_{r-1} \text{且} \ p_r < p_{r+1} \tag{7-4}$$

则 r 是一个谷值点，如果在整个图像中只有一个谷值点，这个谷值点就可以作为目标与背景的分界点——阈值 T。在图 7-2 中给出了简单寻谷法的示意图。图 7-3 示出了用简单寻谷法寻找目标的实际例子，这里认为图像中呈圆球状的物体为目标物 (图 7-3 (a))。为了将目标物输出，采用了简单寻谷法，即根据原始图像的直方图选取阈值 $T=100$ (图 7-3 (b))。大于阈值 T 的样点作为目标，小于阈值 T 的样点作为背景，最后将目标物图像输出，如图 7-3 (c) 所示。

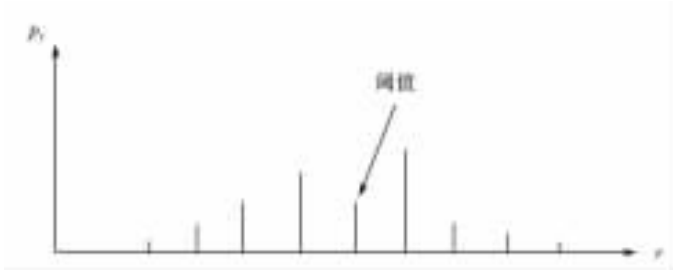


图 7-2　简单寻谷法示意图

在实际应用中，由于噪声等干扰的存在，情况可能会复杂一些，即在图像对应的直方图中可能有多个谷值点，如图 7-4 所示，那么应该选哪一个谷值点作为阈值呢？为了求一个最合理的阈值点，首先引入一个新的概念——“价值数”，若某个灰度值 r 所对应的点是一个谷值点，p_r 是该点的概率值，在直方图中过点 (r，p_r) 做一条直线平行于水平轴，这条直线和 (r，p_r) 点两侧的灰度直方图曲线相交得出两块区域，分别将这两块区域的面积记做 S_r (左) 和 S_r (右)，并规定这两个数中较小的一个称为 r 点的“价值数 ($\mathrm{d}r$)”，即

$$\mathrm{d}r = \min\{S_r \text{(左)}, S_r \text{(右)}\} \tag{7-5}$$

“价值数 ($\mathrm{d}r$)”最大的谷值点作为目标与背景分界的阈值。根据上述可以对图 7-4 中的情况加以处理，1 点可以作为目标与背景分界的阈值点。

在上述算法中，若同时求得几个价值数较大的谷值点，则图像应被分割成多个部分，而不是两个部分。

(2) 试探法。在实际应用中，可以根据对图像的某些先验知识来选取阈值点。已知图像中某一目标占全图的百分比，借助直方图，试取一系列阈值，当取其中的某一个阈值时，其分割以后图像中目标所占的比率等于或接近于原值，那么就选定其为阈值，这种方法称为试探法。如，已知某图像中文字占的比率为 $P\%$，取一系列阈值 t，当选中其中的一个 t_0 后，分割图像中文字占的比率等于或接近 $P\%$，那么就定 t_0 为阈值。又如，已知一图像中某目标物的宽度为 D，试取一系列阈值 t，当选中其中的一个 t_1 后，分割图像中目标物的宽度等于或接近于 D，那么就定 t_1 为阈值。

3. 最佳阈值的讨论

所谓最佳阈值，是指使图像中目标物与背景分割误差为最小的阈值。假设某一图像是由

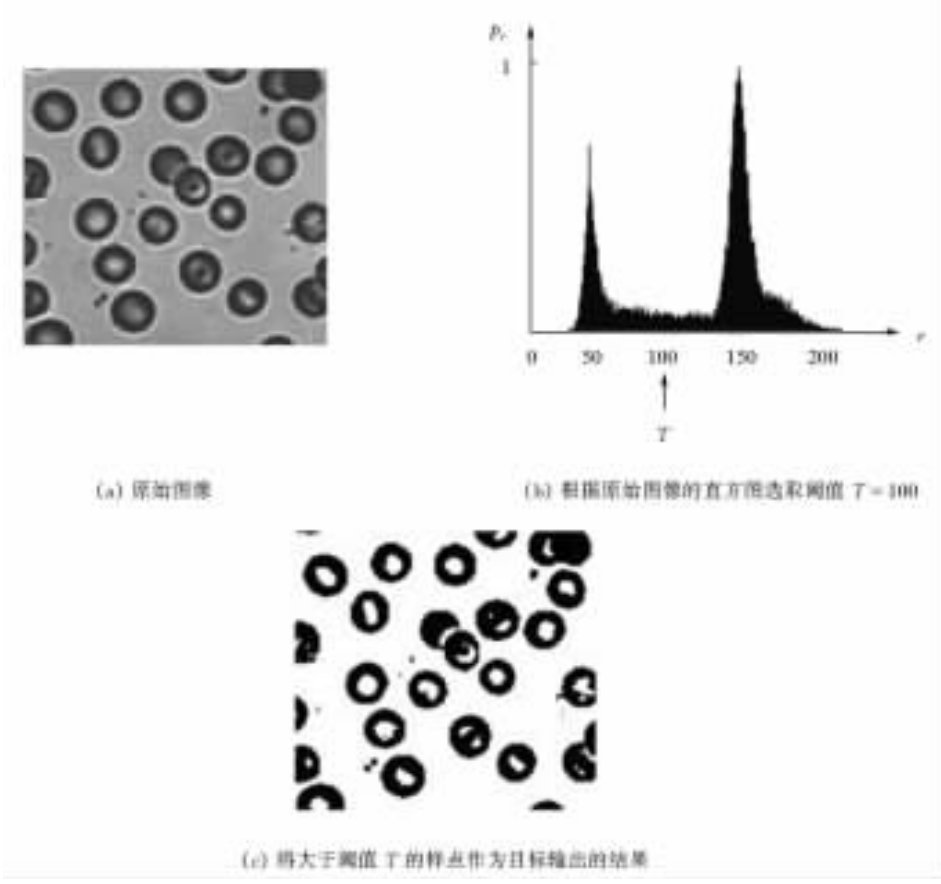


图 7-3 只有一个谷值点的简单寻谷法

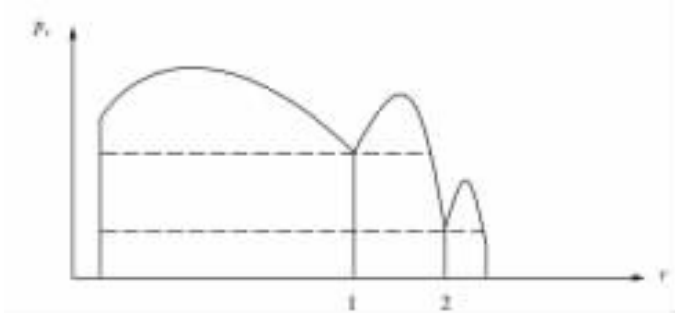


图 7-4 “价值数”法选取阈值点

亮背景上的暗物体所组成的，已知目标点灰度分布的概率密度为 $P_1(r)$，背景点灰度分布的概率密度为 $P_2(r)$，如图 7-5 所示。且已知目标物像素点占全图像素点的比率为 θ，那么背景像素点占全图像素点的比率为 $1-\theta$，这样该图像总的灰度概率密度为

$$P(r) = \theta P_1(r) + (1-\theta) P_2(r) \tag{7-6}$$

如果选取灰度阈值为 T，灰度值小于阈值 T 的样点皆认为是目标点，灰度值大于阈值 T 的样点皆认为是背景点。这样将背景样点错认为是目标点的概率为

$$E_1(T) = \int_{-\infty}^{T} P_2(r)\,\mathrm{d}r \tag{7-7}$$

将目标点错认为是背景样点的概率为

$$E_2(T) = \int_{T}^{\infty} P_1(r)\,\mathrm{d}r \tag{7-8}$$

这样总的错误出现的概率为

$$E(T) = (1-\theta)E_1(T) + \theta E_2(T) \tag{7-9}$$

最佳阈值应该是使错误 $E(T)$ 为最小时对应的阈值，为了求最佳阈值，应将 $E(T)$ 对 T 求导数，并令其导数为零时的 T。

为了具体计算最佳阈值，假设 $P_1(r)$ 和 $P_2(r)$ 均为正态分布函数，即

$$P_1(r) = \frac{1}{\sqrt{2\pi}\sigma_1}\exp\left[\frac{(r-\mu_1)^2}{2\sigma_1^2}\right] \tag{7-10}$$

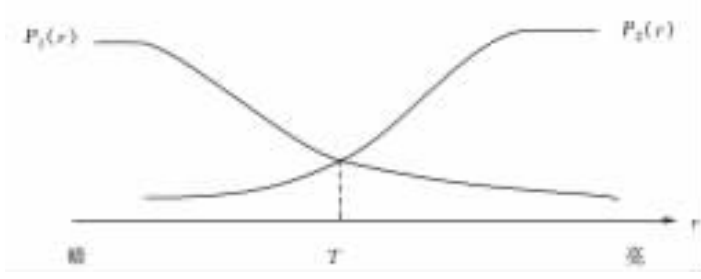


图 7-5　灰度分布

$$P_2(r) = \frac{1}{\sqrt{2\pi}\sigma_2}\exp\left[-\frac{(r-\mu_2)^2}{2\sigma_2^2}\right] \tag{7-11}$$

其中，μ_1 和 μ_2 为灰度均值，σ_1 和 σ_2 为对灰度均值的标准偏差。

当 $\sigma_1=\sigma_2=\sigma$，$\theta=\frac{1}{2}$时，

$$T=\frac{\mu_1+\mu_2}{2} \tag{7-12}$$

可见，假如图像的目标物和背景像素灰度级概率呈正态分布，且偏差相等、背景和目标样点总数也相等，这个图像的最佳分割阈值就是目标物和背景像素灰度级的平均值。程序清单 7-1 给出了 Java 语言实现阈值分割的程序代码。

程序清单 7-1　　ChangeGrey.java 源代码

```
//ChangeGrey.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class ChangeGrey extends Frame {
```

```
Image im, tmp;
int iw, ih;
int[ ] pixels;
boolean flag = false;

//构造方法
public ChangeGrey( ){
    this.setTitle("二值化处理");
    Panel pdown;
    Button load, run, quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter( ){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    pdown = new Panel( );
    pdown.setBackground(Color.lightGray);
    load = new Button("装载图像");
    run = new Button("二值化处理");
    quit = new Button("退出");
    this.add(pdown, BorderLayout.SOUTH);
    pdown.add(load);
    pdown.add(run);
    pdown.add(quit);

    load.addActionListener(new ActionListener( ){
        public void actionPerformed(ActionEvent e){
            jLoad_ActionPerformed(e);
        }
    });

    run.addActionListener(new ActionListener( ){
        public void actionPerformed(ActionEvent e){
            jRun_ActionPerformed(e);
        }
    });

    quit.addActionListener(new ActionListener( ){
```

```
            public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });

}

public void jLoad_ActionPerformed(ActionEvent e){
    //利用 MediaTracker 跟踪图像的加载
    MediaTracker tracker = new MediaTracker(this);
    im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
    tracker.addImage(im,0);

    //等待图像的完全加载
    try{
    tracker.waitForID(0);
    }catch(InterruptedException e2){ e2.printStackTrace();}

    //获取图像的宽度 iw 和高度 ih
    iw=im.getWidth(this);
    ih=im.getHeight(this);
    pixels=new int[iw*ih];

    try{
    PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
    pg.grabPixels();
    }catch (InterruptedException e3) {
        e3.printStackTrace();
    }

    //将数组中的像素产生一个图像
    ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
    tmp=createImage(ip);
    flag=true;
    repaint();
}

public void jRun_ActionPerformed(ActionEvent e){
    if(flag) {
    try{
```

```
        PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //设定二值化的域值,默认值为 100
        int grey = 100;
        Object tmpGrey = "100";
            String s = JOptionPane.showInputDialog(null, "输入二值化的域值(0 - 255):",
tmpGrey);
            //还会有异常抛出!
            if(s! = null){
                grey = Integer.parseInt(s);
            }
        if(grey>255)
        {
            grey = 255;
        }else if(grey<0)
        {
            grey = 0;
        }
        //对图像进行二值化处理,Alpha 值保持不变
        ColorModel cm = ColorModel.getRGBdefault();
        for(int i = 0;i<iw * ih;i++)
        {
            int red,green,blue;
            int alpha = cm.getAlpha(pixels[i]);
            if(cm.getRed(pixels[i])>grey)
            {
                red = 255;
            }else{ red = 0;}

            if(cm.getGreen(pixels[i])>grey)
            {
                green = 255;
            }else{green = 0;}

            if(cm.getBlue(pixels[i])>grey)
            {
```

```
                blue=255;
            }else{blue=0;}

            pixels[i]=alpha<<24|red<<16|green<<8|blue;
        }

        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        repaint();
        }else{
        JOptionPane.showMessageDialog(null,"请先打开一幅图片!",
                                "Alert",JOptionPane.WARNING_MESSAGE);
        }
    }

    public void jQuit_ActionPerformed(ActionEvent e)
    {
        //System.exit(0);
        JOptionPane op =new JOptionPane();
        int exit=op.showConfirmDialog(this,"你要退出吗???","退出",JOptionPane.YES_
NO_OPTION);

        if(exit==JOptionPane.YES_OPTION)
        {
            System.exit(0);

        }else{ }
    }

    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flag){
            g.drawImage(tmp,20,20,this);
        }else { }
    }

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[] args) {
        ChangeGrey cg = new ChangeGrey();
```

```
        cg.setLocation(50,50);
        cg.setSize(500,400);
        cg.show();
    }
}
```

7.1.2 模板匹配法

1. 基本原理

为了在一幅图像中检测出已知形状的目标物，可以使用这个目标物的形状模板（或窗口）与图像匹配，在约定的某种准则下检测出目标物图像，通常称为模板匹配法。

它能检测出图像中的线条、曲线和图案等等。模板匹配法在许多方面已被广泛应用，如将字模与印刷页图像的匹配、星图模板与天空图像的匹配等。还可以从不同角度对同一景物摄取两幅图像，通过匹配能够识别两幅图像中表示同一目标物的那些部分。

模板是为了检测某些不变区域特性而设计的阵列。模板可以根据检测目的的不同而分为点模板、线模板、梯度模板和正交模板等。

2. 模板

在某些情况下，图像中背景的灰度级是恒定的，目标物的灰度级也基本相同，利用这一点来进行模板匹配。

−1	−1	−1
−1	8	−1
−1	−1	−1

图 7-6 点模板

(1) 点模板。图 7-6 示出了一点模板，用这一点模板来检测图像中的小块目标，其步骤如下所述。

将模板中心（标号数 8）在图像中移动，并计算各个位置上模板内各点的值和图像中对应像素灰度级的乘积，并将其结果相加求和。

然后判断结果。若其和为零，表明这个位置上模板所对应的图像局部其灰度级相同，即为背景或目标内部。若其和不为零，表明这个图像局部中一定即有目标、也有背景；当点模板中心正好位于目标和背景的交界处时，其和值最大；模板中心远离目标和背景的交界处时，其和值下降。

在实际应用中，根据不同的模板和图像内容可以设一个阈值，当“和值”大于阈值时，可以认为与其对应的位置为目标与背景的交界处，这样可以检测出目标物。其点模板尺寸的大小可以根据图像的内容和应用要求而定。

(2) 线模板。用来检测图像中的线条目标的模板称为线模板，主要有检测水平直线的模板如图 7-7 所示，检测 45°直线的模板如图 7-8 所示，检测垂直直线的模板如图 7-9 所示，检测 135°直线的模板如图 7-10 所示，用线模板进行检测的步骤与点模板相同。

−1	−1	−1
2	2	2
−1	−1	−1

图 7-7 水平直线模板

−1	−1	2
−1	2	−1
2	−1	−1

图 7-8 45°直线模板

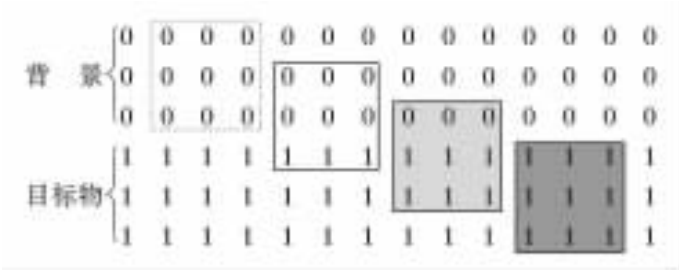
−1	2	−1
−1	2	−1
−1	2	−1

图 7-9　垂直直线模板

2	−1	−1
−1	2	−1
−1	−1	2

图 7-10　135°直线模板

【例 7-1】　采用水平直线模板来检测下图中的水平直线。



模板中心在图中虚线区域中心乘积的和值为

$$S_1=(-1)*0+(-1)*0+(-1)*0+2*0+2*0+2*0+(-1)*0+(-1)*0+(-1)*0=0$$

模板中心在图中实线区域中心乘积的和值为

$$S_2=(-1)*0+(-1)*0+(-1)*0+2*0+2*0+2*0+(-1)*1+(-1)*1+(-1)*1=-3$$

模板中心在图中浅灰色区域中心乘积的和值为

$$S_3=(-1)*0+(-1)*0+(-1)*0+2*1+2*1+2*1+(-1)*1+(-1)*1+(-1)*1=3$$

模板中心在图中深灰色区域中心乘积的和值为

$$S_4=(-1)*1+(-1)*1+(-1)*1+2*1+2*1+2*1+(-1)*1+(-1)*1+(-1)*1=0$$

可见当模板中心在目标物与背景交界处时，其乘积的和值为最大。所以和值为最大的模板中心所对应的图像像素构成了要检测的水平线。

为了进一步研究用比较复杂的模板匹配来检测图像的特征，下面引入矢量运算作为分析问题的工具。这里以 3×3 模板为例，如图 7-11 所示。

假设 M 代表模板，为

$$\boldsymbol{M}=\begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \\ m_9 \end{bmatrix} \tag{7-13}$$

m_1	m_2	m_3
m_4	m_5	m_6
m_7	m_8	m_9

图 7-11　3×3 模板

这里，m_1，m_2，m_3，m_4，m_5，m_6，m_7，m_8，m_9 称为 3×3 模板的权 (系数)。

$\boldsymbol{X}$ 代表图像，为

$$\boldsymbol{X}=\begin{bmatrix}x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9\end{bmatrix} \tag{7-14}$$

这里，x_1，x_2，x_3，x_4，x_5，x_6，x_7，x_8，x_9 代表与模板对应区域上各个像素的灰度级。

定义矢量 $\boldsymbol{M}$ 与 $\boldsymbol{X}$ 的内积为

$$\boldsymbol{M}^{\mathrm{T}}\boldsymbol{X}=m_1x_1+m_2x_2+m_3x_3+m_4x_4+m_5x_5+m_6x_6+m_7x_7+m_8x_8+m_9x_9 \tag{7-15}$$

这里 $\boldsymbol{M}^{\mathrm{T}}$ 为 $\boldsymbol{M}$ 的转置矩阵。选定一阈值 T，当 $\boldsymbol{M}^{\mathrm{T}}\boldsymbol{X}>T$ 时，则认为已经检测到小块目标物。

上述也可以推广，若

$\boldsymbol{M}_1$ 代表水平直线模板；

$\boldsymbol{M}_2$ 代表 45°直线模板；

$\boldsymbol{M}_3$ 代表垂直直线模板；

$\boldsymbol{M}_4$ 代表 −45°直线模板；

$\boldsymbol{X}$ 代表模板对应区域的图像矩阵。

假如在图像的某个区域上 $\boldsymbol{M}_i^{\mathrm{T}}\boldsymbol{X}$ 值最大，若 $i=1$ 表明图像此区域有水平性质；若 $i=3$ 表明图像此区域有垂直性质等。

7.1.3 边缘检测法

7.1.1 中介绍了利用一幅图像中目标和背景上样点特性 (如灰度) 的不同，来达到将目标从背景中分割出来的目的。除此而外，还有其他的分割方法。如将图像中各个不同区域的边缘检测出来，也可以达到图像分割的目的。

1. 边缘的特点

在一幅图像中，目标点的边缘大致有两类，其一称为阶跃状边缘，这种边缘两边样点的灰度值明显不同，如图 7-12 所示。在图 7-12 中，P 点处为一阶跃状边缘，其灰度分布曲线的一阶导数在该处为极大值，灰度分布曲线的二阶导数在该处为零。

其二称为屋顶状边缘，如图 7-13 所示。在图 7-13 中，Q 点处为一屋顶状边缘，其灰度分布曲线的一阶导数在该处为零，灰度分布曲线的二阶导数在该处为极小值。所以根据边缘处的灰度曲线的导数特性就可以分割图像中的不同区域。

2. 常用的边缘检测算法

(1) 梯度算法。最简单的边缘检测算法是求一阶导数$\frac{\partial f}{\partial x}$和$\frac{\partial f}{\partial y}$，即求出灰度在 x 和 y 方

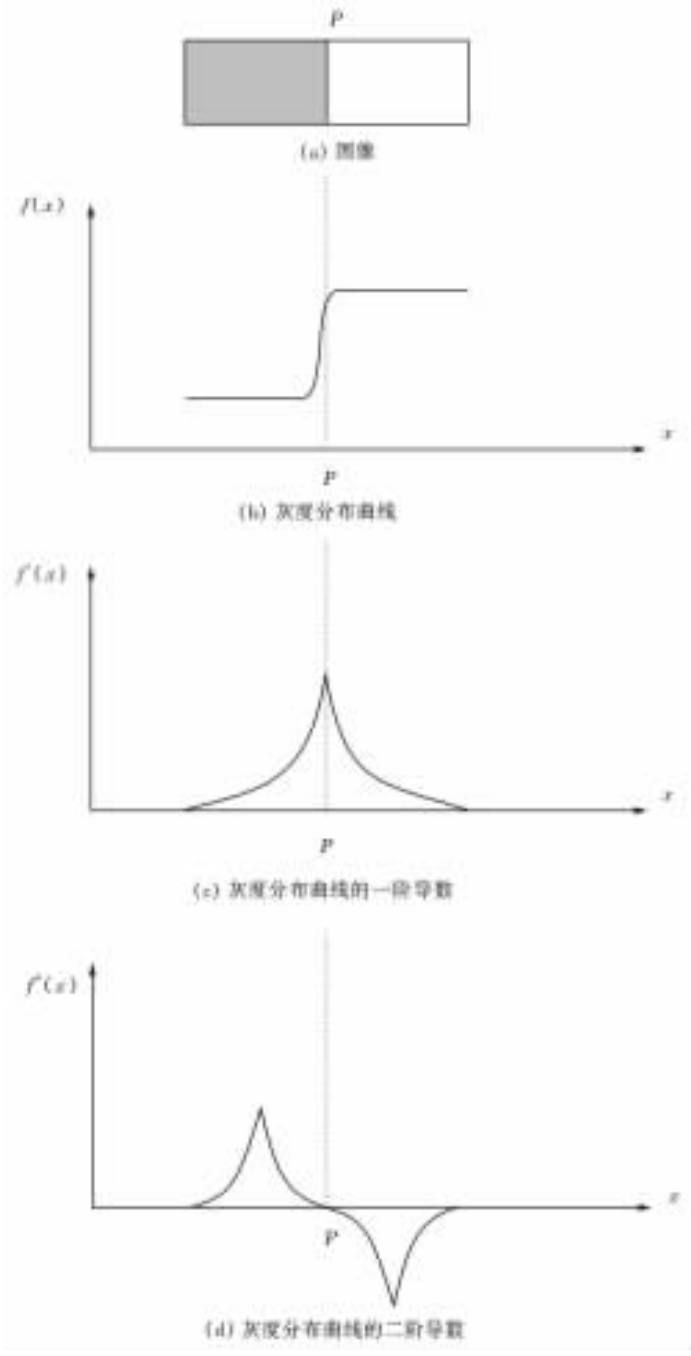


图 7-12　阶跃状边缘图示

向的变化率。假设图像函数为 $f(x, y)$，它的一阶导数 (梯度) 为

$$\boldsymbol{G}(f(x, y)) = \frac{\partial f}{\partial x}i + \frac{\partial f}{\partial y}j \tag{7-16}$$

梯度的模为

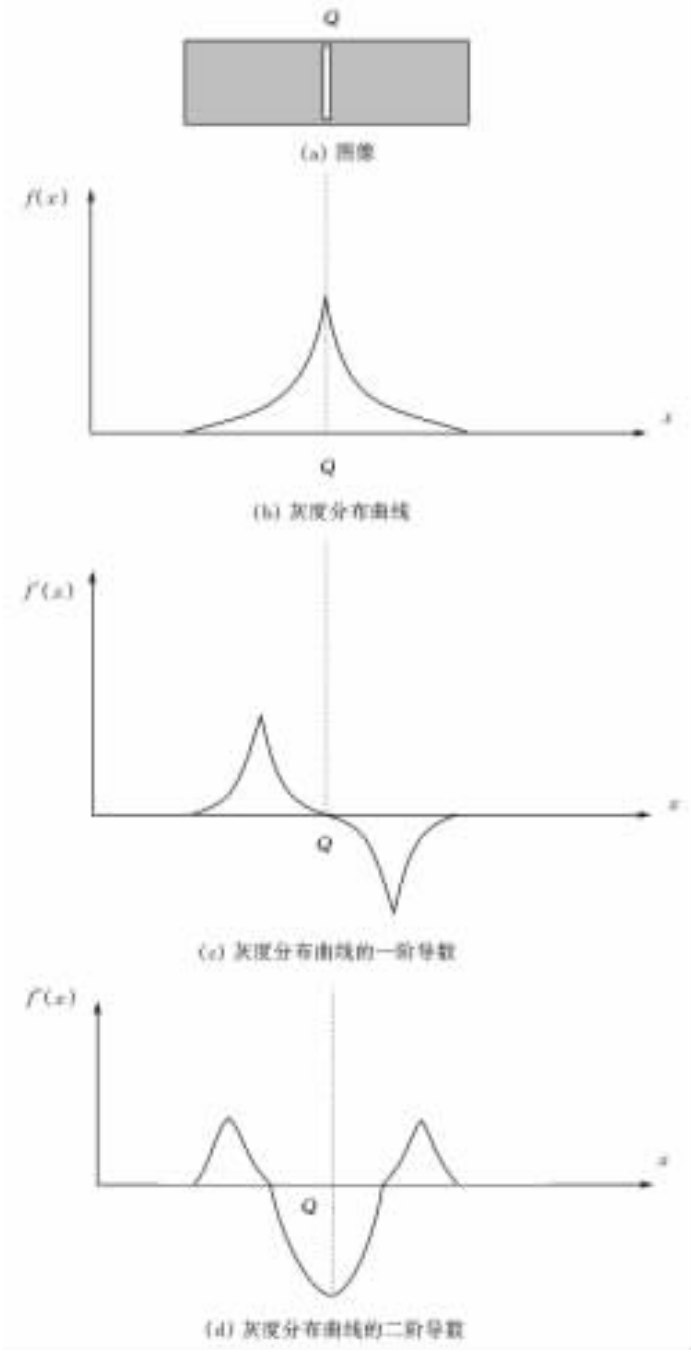


图 7-13　屋顶状边缘图示

$$G(f(x,y)) = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{7-17}$$

对于数字图像，式 (7-17) 要采用离散的形式，为此可用差分运算来代替微分运算。在 x 方向的一阶差分为

$$\Delta_x f(x, y) = f(x, y) - f(x+1, y) \tag{7-18}$$

在 y 方向的一阶差分为

$$\Delta_y f(x, y) = f(x, y) - f(x, y+1) \tag{7-19}$$

梯度的模可以简化为

$$G(f(x, y)) = \sqrt{[f(x, y) - f(x+1, y)]^2 + [f(x, y) - f(x, y+1)]^2} \tag{7-20}$$

为了便于计算机的计算，通常使用绝对值来代替式 (7-20)，即

$$G(f(x, y)) \approx |f(x, y) - f(x+1, y)| + |f(x, y) - f(x, y+1)| \tag{7-21}$$

关于梯度处理还有一种近似方法 (一种经验公式) 叫 Robert 梯度，即

$$G(f(x, y)) = \sqrt{[f(x, y) - f(x+1, y+1)]^2 + [f(x+1, y) - f(x, y+1)]^2} \tag{7-22}$$

式 (7-22) 用绝对值近似表示为

$$G(f(x, y)) \approx |[f(x, y) - f(x+1, y+1)]^2 + [f(x+1, y) - f(x, y+1)]^2| \tag{7-23}$$

利用上述运算可以突出梯度值大的边缘，如阶跃式边缘，从而达到分割图像的目的。图 7-14 示出了用 Java 语言实现的 Robert 梯度算法来检测的图像中各个区域边缘的效果。其运行程序如程序清单 7-2 所示。

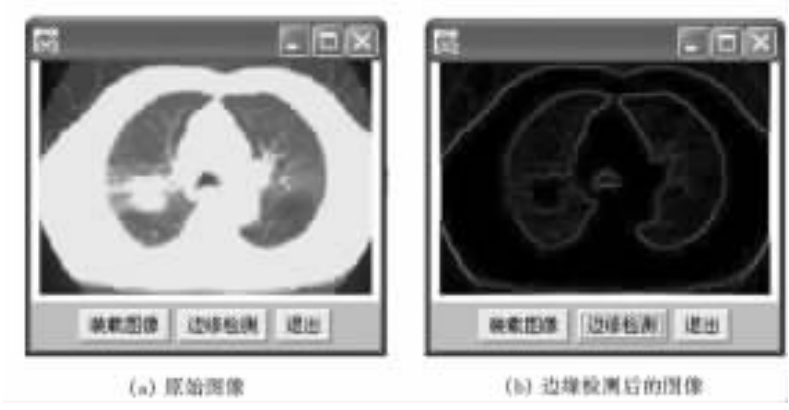


图 7-14　Robert 梯度进行边缘检测

程序清单 7-2　　Robert.java 源代码

```
//Robert.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class Robert extends Frame {
```

```
Image im, tmp;
int i, iw, ih;
int[ ] pixels;
boolean flag = false;

//ImagePixel 的构造方法
public Robert( ){
    this.setTitle("Robert 边缘检测");
    Panel pdown;
    Button load, run, quit;
    //添加窗口监听事件
    addWindowListener(new WindowAdapter( ){
        public void windowClosing(WindowEvent e){
            System.exit(0);
        }
    });

    pdown = new Panel( );
    pdown.setBackground(Color.lightGray);

    load = new Button("装载图像");
    run = new Button("边缘检测");
    quit = new Button("退出");

    this.add(pdown, BorderLayout.SOUTH);

    pdown.add(load);
    pdown.add(run);
    pdown.add(quit);

    load.addActionListener(new ActionListener( ){
        public void actionPerformed(ActionEvent e){
            jLoad_ActionPerformed(e);
        }
    });

    run.addActionListener(new ActionListener( ){
        public void actionPerformed(ActionEvent e){
            jRun_ActionPerformed(e);
        }
```

```
        });

        quit.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });
    }

    public void jLoad_ActionPerformed(ActionEvent e){
        //利用 MediaTracker 跟踪图像的加载
        MediaTracker tracker = new MediaTracker(this);
        im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
        tracker.addImage(im,0);

        //等待图像的完全加载
        try{
        tracker.waitForID(0);
        }catch(InterruptedException e2){ e2.printStackTrace();}

        //获取图像的宽度 iw 和高度 ih
        iw=im.getWidth(this);
        ih=im.getHeight(this);
        pixels=new int[iw*ih];

        try{
        PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }

        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        flag=true;
        repaint();
    }

    public void jRun_ActionPerformed(ActionEvent e){
```

```
if(flag) {
try{
PixelGrabber pg = new PixelGrabber(im, 0, 0, iw, ih, pixels, 0, iw);
pg.grabPixels();
}catch (InterruptedException e3) {
    e3.printStackTrace();
}

//对图像进行边缘提取, Alpha 值保持不变
ColorModel cm = ColorModel.getRGBdefault();
for(i = 1; i < ih - 1; i++ )
{
    for(int j = 1; j < iw - 1; j++ )
    {
    //对图像进行边缘提取
    int alpha = cm.getAlpha(pixels[i * iw + j]);
    int red5 = cm.getRed(pixels[i * iw + j]);
    int red6 = cm.getRed(pixels[i * iw + j + 1]);
    int red8 = cm.getRed(pixels[(i + 1) * iw + j]);
    int red9 = cm.getRed(pixels[(i + 1) * iw + j + 1]);

    int robertRed = Math.max(Math.abs(red5 - red9), Math.abs(red8 - red6));

    int green5 = cm.getGreen(pixels[i * iw + j]);
    int green6 = cm.getGreen(pixels[i * iw + j + 1]);
    int green8 = cm.getGreen(pixels[(i + 1) * iw + j]);
    int green9 = cm.getGreen(pixels[(i + 1) * iw + j + 1]);

     int robertGreen = Math.max(Math.abs(green5 - green9), Math.abs(green8 -
green6));

    int blue5 = cm.getBlue(pixels[i * iw + j]);
    int blue6 = cm.getBlue(pixels[i * iw + j + 1]);
    int blue8 = cm.getBlue(pixels[(i + 1) * iw + j]);
    int blue9 = cm.getBlue(pixels[(i + 1) * iw + j + 1]);

    int robertBlue = Math.max(Math.abs(blue5 - blue9), Math.abs(blue8 - blue6));

    pixels[i * iw + j] = alpha << 24 | robertRed << 16 | robertGreen << 8 | robertBlue;
    }
```

```
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        repaint();
        }else{
        JOptionPane.showMessageDialog(null, "请先打开一幅图片!",
            "Alert", JOptionPane.WARNING_MESSAGE);
        }
    }
        public void jQuit_ActionPerformed(ActionEvent e){
        //System.exit(0);
        JOptionPane op = new JOptionPane();
        int exit = op.showConfirmDialog(this, "你要退出吗? ? ?", "退出", JOptionPane.YES_
NO_OPTION);

        if(exit = = JOptionPane.YES_OPTION)
        {
            System.exit(0);

        }else{ }
    }

    //调用 paint()方法,显示图像信息。
    public void paint(Graphics g){
        if(flag){
            g.drawImage(tmp, 10, 20, this);
        }else { }
    }

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[ ] args) {
        Robert ro = new Robert();
        ro.setLocation(50, 50);
        ro.setSize(500, 400);
        ro.show();
    }
    }
```

(2) Laplace 算子。对于屋顶状边缘处，其灰度分布的二阶导数出现极小值，这样应该利用对图像的各个样点求二阶导数的方法找出边缘，常用的方法就是计算 Laplace 算子。假设图像函数为 $f(x, y)$，它的 Laplace 算子为

$$\nabla^2 f(x, y) = \frac{\partial f^2}{\partial x^2} + \frac{\partial^2 f}{\partial^2 y} \tag{7-24}$$

对于数字图像，式 (7-24) 要采用离散的形式，为此可用差分运算来代替微分运算。

$$\nabla^2 f(x, y) = |[f(x+1, y) - f(x, y)] - [f(x, y) - f(x-1, y)]| + |[f(x, y+1) - f(x, y)] - [f(x, y) - f(x, y-1)]| \tag{7-25}$$

这就是用 Laplace 算子来检测图像中目标物边缘的方法。图 7-15 示出了用 Laplace 算子来检测的图像中各个区域边缘的效果。

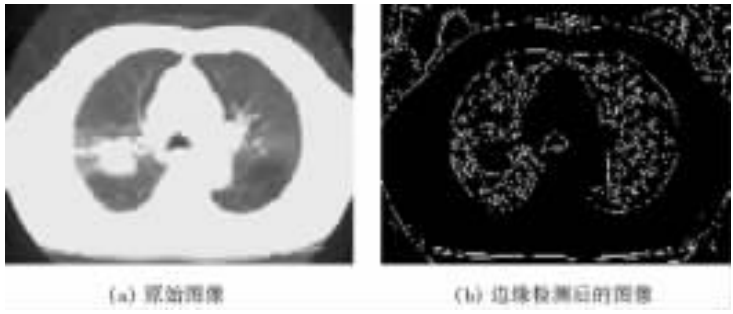


图 7-15 Laplace 算子进行边缘检测

7.2 图像的描述

图像经过分割处理后，具有不同特性的区域将被分开，下一步的任务是要用一系列数字或语句对已分割的图像进行描述，以便使用计算机进行处理。

7.2.1 区域描述

1. 搜索性描述

有些时候无法对图像中区域的形状进行定量的描述，所以常采用一种搜索性描述。它是针对目标区域的周长的平方与该区域面积的比值来描述目标区域的形状。因为这个比值可以较好地区分出长形区域和圆形区域，对于长形区域，其比值较大；对于圆形区域，其比值较小。

2. 傅立叶描述法

这种方法是沿着目标区域的边界点作傅立叶变换，根据傅立叶级数的系数来描述该区域。如果目标区域的边界是一条闭合的曲线，相对边界上的某一个固定的点，沿边界线上一个移动点的位置变化是一个周期函数，这个周期函数可以展开为傅立叶级数，然后用傅立叶级数的系数来描述区域的形状。

假设图 7-16 为一复平面，其中 x 轴表示实数轴，y 轴表示虚数轴，边界上一动点从 P_0 点沿弧 s 逆时针移动而形成闭合曲线，那么 P 点的位置为

$$P(s) = x(s) + jy(s) \tag{7-26}$$

P 点曲线的正切为

$$\tan(\Phi(s)) = \frac{\Delta y(s)}{\Delta x(s)} \tag{7-27}$$

式 (7-27) 的离散形式为

$$\tan(\Phi(s)) = \frac{y(s_i) - y(s_{i-1})}{x(s_i) - x(s_{i-1})} \tag{7-28}$$

P 点曲率为

$$q(s) = \frac{\Delta y(s)}{\Delta x(s)} \tag{7-29}$$

因为 $q(s)$ 是周期为 N 的周期函数，所以可展开成傅立叶级数为

$$q(s) = \sum_{n=-\infty}^{\infty} C_n \exp(j2\pi ns/N) \tag{7-30}$$

这里

$$C_n = \frac{1}{N}\int_0^N q(s)\exp(-j2\pi ns/N)\,ds \tag{7-31}$$

傅立叶描述就是用 C_n 来描述区域的边界特征。

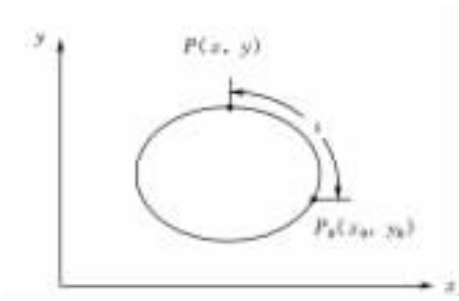


图 7-16　复平面上一闭合曲线

图 7-17　具有两个空洞的区域

3. 拓扑描述

拓扑描述是对一幅图像的整体性质的描述。假设某一图像如图 7-17 所示，有两个空洞 (空洞数目用 H 表示)，两个相关联的区域 (相关联区域的数目用 C 表示)，这样定义一个新概念——Euler 数 (E) 为

$$E = C - H \tag{7-32}$$

Euler 数描述了一个区域的拓扑性质，如图 7-18 中的两区域。对于区域 a：$C=1$，$H=1$，则

$$E = 1 - 1 = 0$$

对于区域 b：$C=1$，$H=2$，则

$$E = 1 - 2 = -1$$

7.2.2　关系描述

所谓关系描述，就是根据从图像中提取的目标的结构，利用句法模式语言来描述。如图 7-19 (a) 所示是从一幅图像中分割出的一个楼梯结构，为了对这个结构加以描述，现在定义

两个基本元素 a 和 b，这样图 7-19 (a) 就可以描述成图 7-19 (b) 的形式。这个简单的描述可以用下面的递归关系表达：

(1) $S \rightarrow aA$

(2) $A \rightarrow bS$

(3) $A \rightarrow b$

这里 S 和 A 是变量，a 和 b 是常数。

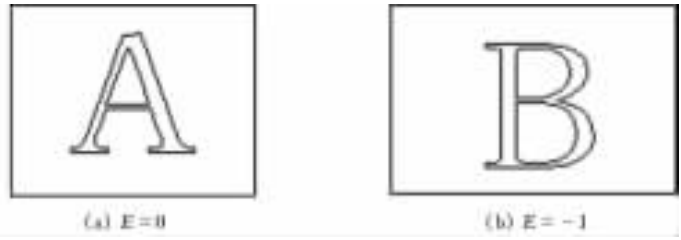


图 7-18 两个区域

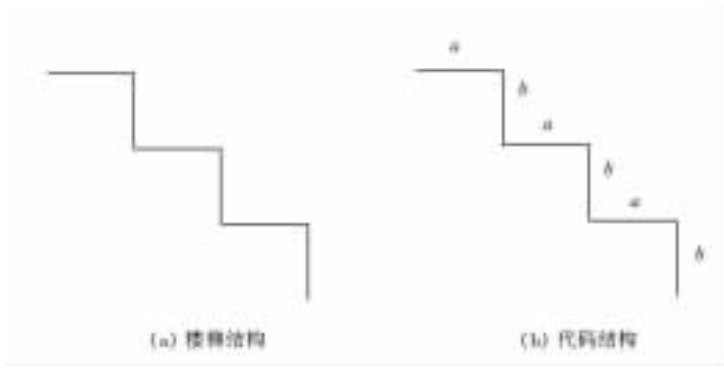


图 7-19 关系描述

第一条规则表明 S 可以由 a 和 A 代替，而 A 也可以由 b 和 S 代替或仅由 b 代替。如果用 bS 代替 A，那么又会出现 S 由 a 和 A 代替，即出现了循环过程；而如果 A 由 b 代替，则该过程结束。图 7-20 示出了上述规则的应用例子，图中 1、2 和 3 分别表示规则 (1)、(2) 和 (3)。

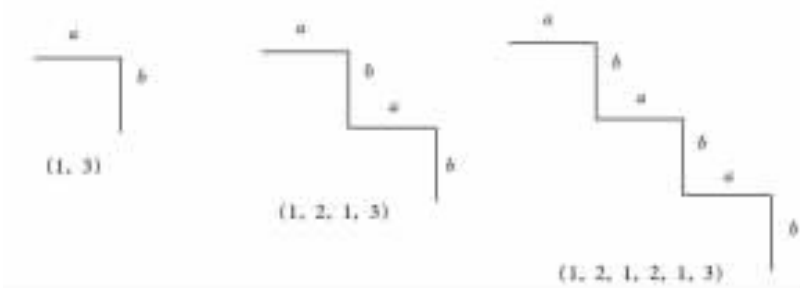


图 7-20 规则应用例子

上述可见，图像中的模式是可以用一些符号来表示的。通常用线段的长度和方向来描述目标区域的边界，如图 7-21 所示。

如果目标区域是很小的均匀区域，可以用有方向的抽象线段来表示该区域，如图 7-22 所示。

图 7-21　用线段的长度和方向描述目标区域的边界

图 7-22　区域抽象成有向线段

为了描述各个小区域间的关系，可以对抽象的线段进行一些运算，如图 7-23 所示。

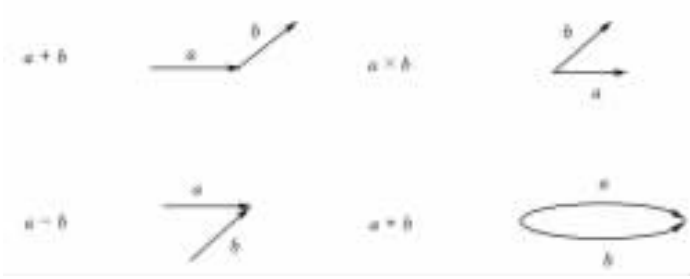


图 7-23　抽象线段的几种运算

此外，还有许多利用句法模式语言来描述图像的方法，这里就不一一列举了。

7.2.3　相似性描述

相似性描述是为了描述两个或几个区域的相似程度，常使用的描述方法如下：

1. 距离测量

为了比较两个区域的相似程度，首先将这两个区域的描述子分别用矢量 $\boldsymbol{X}_1$ 和 $\boldsymbol{X}_2$ 表示，然后计算 $\boldsymbol{X}_1$ 和 $\boldsymbol{X}_2$ 间的距离，即

$$D(\boldsymbol{X}_1, \boldsymbol{X}_2) = \| \boldsymbol{X}_1 - \boldsymbol{X}_2 \| = \sqrt{(\boldsymbol{X}_1 - \boldsymbol{X}_2)^{\mathrm{T}} (\boldsymbol{X}_1 - \boldsymbol{X}_2)} \tag{7-33}$$

用 $D(\boldsymbol{X}_1, \boldsymbol{X}_2)$ 来描述两个区域的相似性。对此可以推广应用，若已知 L 个描述子 $\boldsymbol{X}_1$，$\boldsymbol{X}_2$，…，$\boldsymbol{X}_L$ 和一个未知的描述子 $\boldsymbol{X}$，根据式 (7-33)，若未知的描述子 $\boldsymbol{X}$ 与第 i 个已知描述子 $\boldsymbol{X}_i$ 的距离最近，即

$$D(\boldsymbol{X}, \boldsymbol{X}_i) < D(\boldsymbol{X}, \boldsymbol{X}_j) \quad (j = 1, 2, \cdots, L, j \neq i) \tag{7-34}$$

这时 $\boldsymbol{X}$ 更接近与第 i 个描述子 $\boldsymbol{X}_i$。

2. 相似性

这里是通过求两个图像的相关来比较它们的相似性。假设一个 $M \times N$ 的数字图像 $f(x, y)$ 和一个 $J \times K$ 的数字图像 $w(x, y)$，这里 $J < M$ 及 $K < N$。现在要确定是否图像 $f(x, y)$ 中包含与图像 $w(x, y)$ 相似的区域。常用的方法是根据 $f(x, y)$ 和 $w(x, y)$ 的相关函数

$$R(m, n) = \sum_x \sum_y f(x, y)\, w(x - m, y - n) \tag{7-35}$$

来确定。这里 $m = 0, 1, 2, \cdots, M - 1$，$n = 0, 1, 2, 3, \cdots, N - 1$。运行过程是

$w(x, y)$ 在图像 $f(x, y)$ 内移动搜索，如图 7-24 所示，对于图像 $f(x, y)$ 内的任何 (m, n) 点均可根据式 (7-35) 求出 R 值，即一共可以求出 $(M-J+1)\times(N-K+1)$ 个 $R(m, n)$ 值，从中找出最大 R 值的位置就是图像 $f(x, y)$ 中与 $w(x, y)$ 最相似的区域。

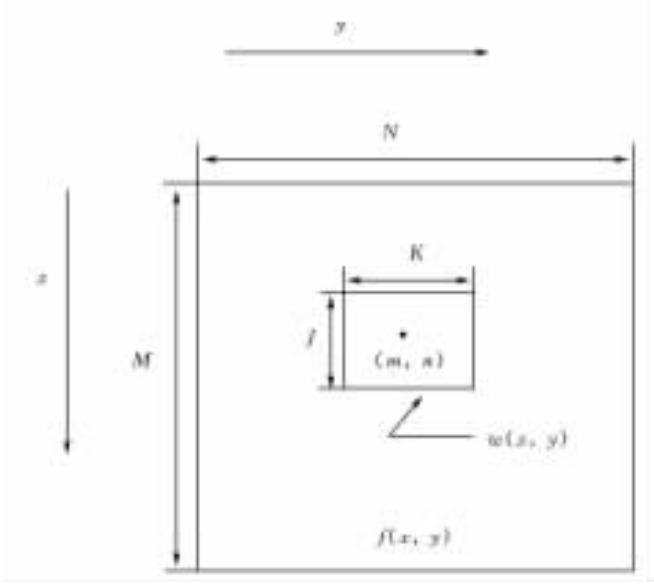


图 7-24 相似性

练 习 题

1 采用阈值分割法，试用 Java 语言实现对某一图像中的目标物进行提取。

2 试用 Java 语言实现对某一图像进行梯度边缘检测和 Laplace 算子边缘检测，并说明为什么梯度边缘检测和 Laplace 算子边缘检测的效果不同。

第 8 章　数字图像处理技术的应用——医学图像处理

8.1　医学成像系统的物理基础

首先介绍一下医学图像处理系统中所涉及的一些物理概念。

8.1.1　原子模型

1. 原子结构的基本性质

原子是由原子核和绕核运动的电子组成的，原子核又是由质子和中子组成的，质子带正电，中子不带电，如图 8-1 所示。原子核可用符号

$$^{A}_{Z}\mathrm{X}_{N}$$

表示，其中 N 代表原子核内的中子数；Z 代表原子核内的质子数；$A = N + Z$ 称为原子核的核子数，又称质量数；X 代表与 Z 相联系的元素符号。具有相同质子数和中子数的原子称为同一核素。不同核素的原子具有不同的能量状态，在天然存在的核素中，约有 280 种是稳定核素，约 60 种是天然放射性核素。此外，人们还可以制造出 2000 多种放射性核素。Z 相同 N 不同的核素称为同位素，N 相同 Z 不同的核素称为中子异荷素，A 相同 Z 不同的核素称为同量异位素。

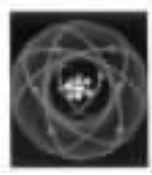


图 8-1　原子结构示意图

原子中的电子是沿着一定的“轨道”绕原子核转动的，每个轨道都有确定的能量，被称为电子能级。如果电子由高能级向低能级跃迁将向外辐射能量，反之则吸收能量。假设一个电子从 K 层 (70keV) 跃迁到 L 层 (11keV)，发出的电磁辐射为 70keV − 11keV = 59keV。

2. 原子核的放射性衰变

不稳定的放射性核素会自发地蜕变，而成为另一种核素，同时放射出各种射线，这种现象称为放射性衰变。目前主要的放射性衰变形式有：α 衰变，β 衰变和 γ 衰变。α 衰变是放出两个带正电荷的氦核。β 衰变包括 β^{+} 衰变、β^{-} 衰变和电子俘获 (EC)，其中 β^{-} 衰变是放出电子的同时又放出反中微子；β^{+} 衰变是放出正电子的同时又放出中微子；EC 是原子核俘获一个核外电子；γ 衰变是放出波长很短的电磁辐射。

原子核是一个量子体系，核蜕变是原子核自发产生的变化，是量子跃迁的过程，服从量子力学的统计规律。假设在 Δt 时间内发生核蜕变的数目为 $-\Delta N$，则有关系

$$-\Delta N = \lambda N \Delta t \tag{8-1}$$

式中　λ——衰变系数；

N——原子核数目。

若 $t = 0$ 时的原子核数目为 N_0，则对式 (8-1) 积分后为

$$N = N_0 \exp(-\lambda t) \tag{8-2}$$

式 (8-2) 为放射性衰变服从的指数定律。放射性核素衰变为原有核素一半所需要的时间为半衰期 (T)。当 $t = T$ 时，$N = N_0/2$，根据式 (8-2) 有

$$T = \frac{\ln 2}{\lambda} \tag{8-3}$$

可见 λ 越大 T 越小，半衰期和衰变系数是表示放射性核素的特征常数。

8.1.2 电磁波

波是一种能量的转换，通常波分为两类：其一是传播依赖于媒质的波，如声波是媒质中质点振动状态的传播，离开媒质就不存在声波了；另一类是传播不依赖于媒质的波，如电磁波，图 1-3 中示出了电磁波谱，其中的 X 射线谱在医学上被广泛应用。

频率为 ν 的电磁波的能量为

$$E = h\nu \tag{8-4}$$

式中 $h = 6.626 \times 10^{-34}$ J·s，称为普朗克常量，在真空中电磁波的速度传播为

$$c = \lambda\nu = 2.9979 \times 10^8 \text{m/s} \tag{8-5}$$

电磁波不但具有波动性，还具有粒子性 (光子)，即具有波粒二重性，光子的能量为

$$E = h\nu = mc^2 \tag{8-6}$$

式中 m——光子的等效质量。

8.1.3 伦琴射线

伦琴射线又称为 X 射线，它是 1895 年德国物理学家伦琴在研究阴极射线时发现的，X 射线也是一种电磁波，它直线传播，经过电场不发生偏转，具有很强的穿透能力。X 射线的第一个应用是在医学领域，当时伦琴做了他妻子手的 X 射线照相，看到了软组织、骨头结构和结婚戒指的形状 (如图 1-4 所示)，目前 X 射线已经成为医学界检查人体病理状态内部组织结构变化的主要手段。

医用 X 射线是由 X 射线管发射出的，X 射线管是由阴极和阳极组成的，在两极间加上高电压 (U)，使从阴极产生的电子具有很高的速度，当高速电子撞击阳极时，与阳极物质发生作用产生了 X 射线，如图 8-2 所示。这里加速电子的电量为 e，能量为 eU。X 射线管发出的典型 X 射线谱如图 8-3 所示，谱线由两部分组成：平滑连续的部分叠加了几个尖锐的峰，前者称为连续谱，后者称为标识谱。

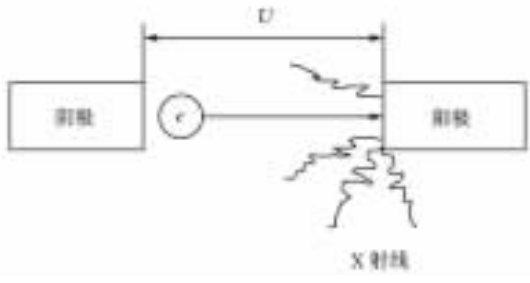


图 8-2 X 射线的产生过程

连续谱是高速电子突然撞击到阳极时产生的，若电子的能量全部转化为光子的能量，即

$$eU = \frac{hc}{\lambda_{\min}} \tag{8-7}$$

这里 $\lambda_{\min}$ 为连续谱的短波极限，可见 $\lambda_{\min}$ 随 U 的增加而减少。当电子撞击阳极时，使阳极物质的原子激发，引起内层电子跃迁，从而形成波长不连续的射线，称为标识谱。

当 X 射线进入物质时，X 射线的能量与物质发生相互作用，部分能量被散射，部分能量被吸收，使得透过物质后原射线强度被减弱。假设一束 X 射线射入物质中，如果入射强度为 I_0，在物质中传播的强度为

$$I = I_0 \exp(-\mu x) \tag{8-8}$$

式中　x——物质的厚度；

μ——全线性衰减因子。

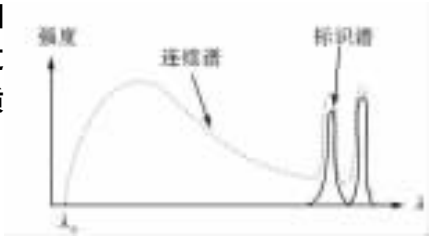


图 8-3　X 射线谱

式 (8-8) 称为指数定律。通常物质对 X 射线的吸收程度随射线波长的增加而增大，波长越小穿透能力越强，这种短波射线称为硬射线，而长波射线称为软射线。

8.1.4　磁共振效应

一个角动量为 l 的粒子，它的磁矩为

$$\boldsymbol{\mu} = \gamma \boldsymbol{l} \tag{8-9}$$

式中　γ——旋磁比，$\gamma>0$，$\boldsymbol{\mu}$ 与 $\boldsymbol{l}$ 同向；$\gamma<0$，$\boldsymbol{\mu}$ 与 $\boldsymbol{l}$ 反向。

这样的粒子处于外磁场 $\boldsymbol{B}_0$ 中，将发生塞曼效应，即一个能级会分裂为几个能级。

如果表示角动量 $\boldsymbol{l}$ 大小的角量子数为 L，则该角动量对应的能级在静磁场中将分裂为 $2L+1$ 个能级，在理想情况下，分裂能级的间距为

$$\Delta E = 2\mu_B B_0 \tag{8-10}$$

式中　μ_B——磁矩 $\boldsymbol{\mu}$ 在静磁场 $\boldsymbol{B}_0$ 方向的投影。

电子具有自旋角动量和自旋磁矩，电子自旋角动量的量子数为 $S=\frac{1}{2}$，那么在外静磁场中，电子自旋角动量可分裂成 $2S+1=2$ 个能级。可见在外磁场 $\boldsymbol{B}_0$ 中，电子自旋磁矩只能有两个取向：平行于 $\boldsymbol{B}_0$ (μ_B) 和反平行于 $\boldsymbol{B}_0$ ($-\mu_B$)，它们对应的能量分别为 $\mu_B B_0$ 和 $-\mu_B B_0$，如图 8-4 所示。

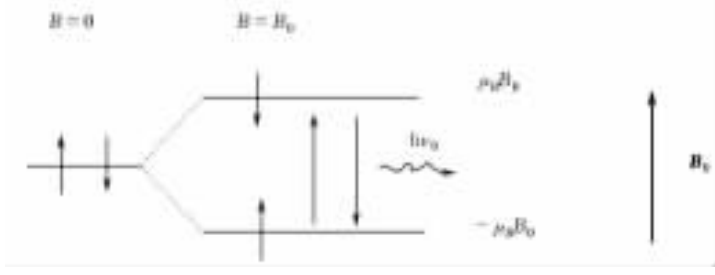


图 8-4　电子自旋共振

如果在外加静磁场 $\boldsymbol{B}_0$ 的基础上又加一个高频电磁场，这个高频电磁场的频率满足 $h\nu_0=2\mu_B B_0$，若电子自旋由平行 $\boldsymbol{B}_0$ 的状态跃迁到反平行 $\boldsymbol{B}_0$ 的状态，则称为共振吸收；若电子自旋由反平行 $\boldsymbol{B}_0$ 的状态跃迁到平行 $\boldsymbol{B}_0$ 的状态，则称为共振发射。发生共振吸收和共振发射的现象称为电子自旋共振。

原子核也有自旋和磁矩，当原子核中质子数和中子数均为偶数时，核自旋量子数 $I=0$，原子核无核磁矩，如 ${}_6C^{12}$，${}_8O^{16}$ 核等。当原子核中质子数和中子数均为奇数时，核自旋量子数

为偶数。当原子核中质子数和中子数之和为奇数时，核自旋量子数为半整数 $\left(\frac{1}{2}, \frac{3}{2}, \cdots\right)$，如 ${}_1H^1$，${}_9F^{19}$ 核等。以 ${}_1H^1$ 例，其 $I=\frac{1}{2}$，在外静磁场中分裂为 $2I+1=2$ 个能级，这时核磁矩在 $\boldsymbol{B}_0$ 方向的投影为 $\mu_B=\pm 2.79\mu_N$，这里 $\mu_N=5.05\times10^{-27}J/T$ 为核磁子，能级间距为

$$\Delta E=2\mu_B B_0=2\times 2.79\mu_N B_0 \tag{8-11}$$

如果在静磁场的基础上再加交变磁场 $h\nu_0=2\times2.79\mu_N B_0$，使核磁矩由平行 $\boldsymbol{B}_0$ 的状态跃迁到反平行 $\boldsymbol{B}_0$ 的状态，发生共振吸收；或由反平行 $\boldsymbol{B}_0$ 的状态跃迁到平行 $\boldsymbol{B}_0$ 的状态，发生共振发射，核磁矩的这种跃迁称为核磁共振。

可见，核磁共振是物质在静磁场和射频 (电) 磁场共同作用下的效应，静磁场使磁矩振动，射频 (电) 磁场使振动发生改变，出现共振跃迁。通过共振信号的测量可以得到核磁矩，了解它们所处的周围环境和相互作用，通过共振信号的强弱和空间分布可以了解核分布的密度，这些就是核磁共振成像技术的基础。在外加的静磁场中，表示物质的总磁矩由零达到饱和值过程的时间称为纵向弛豫时间 T_1，表示物质的总磁矩的横向分量达到平衡过程的时间称为横向弛豫时间 T_2，在核磁共振成像过程中 T_1、T_2 和核密度等参数都是很重要的量。

8.1.5 多普勒效应

当波源与观察者之间存在相对运动时，观察者所接受到的波的频率不等于波源振动的频率，这种现象称为多普勒效应。通常多普勒效应分为声多普勒效应和光多普勒效应。

1. 声多普勒效应

当高速行驶的列车鸣笛向我们驶来时，笛声的声调变高；而鸣笛离去时，笛声的声调变低，这种现象就是声多普勒效应。假设声源的振动频率为 ν，波长为 λ，声波在媒质中的传播速度为 c'。若声源不动，观察者以速度 v 向声源方向运动，这时观察者不是停在原地等待着一个个波的到来，而是迎上去接受更多的波，所以波相对于观察者的速度为 $c'+v$，这样观察者接受到声波的频率为

$$\nu'=\frac{c'+v}{\lambda}=\frac{c'+v}{\frac{c'}{\nu}}=\frac{c'+v}{c'}\nu \tag{8-12}$$

可见当观察者以速度 v 向静止的声源方向运动时，接收到的声波频率为声源振动频率的 $\left(1+\frac{v}{c'}\right)$ 倍，所以听到的声调变高。那么，当观察者以速度 v 向离开声源的方向运动时，观察者接受到声波的频率为

$$\nu'=\frac{c'-v}{\lambda}=\frac{c'-v}{\frac{c'}{\nu}}=\frac{c'-v}{c'}\nu \tag{8-13}$$

可见当观察者以速度 v 离开静止的声源时，接收到的声波频率为声源振动频率的 $\left(1-\frac{v}{c'}\right)$ 倍，所以听到的声调变低。

2. 光多普勒效应

当光源与接受器之间有相对运动时，接受器感受到的光的频率不同于光源的频率，这种现象称为光多普勒效应或电磁波多普勒效应。光多普勒效应与声多普勒效应有本质的区别，即声的传播依赖于媒质，而光的传播不依赖于媒质，在任何媒质中光的传播速度均与真空中

相同。如果光源与接受器之间的相对速度为 u，光源的频率为 ν_s，则接受器测得的频率为

$$\nu_t = \frac{\sqrt{1-\frac{u^2}{c^2}}}{1+\frac{u\cos\theta}{c}}\nu_s \tag{8-14}$$

这里 c 为光速，$u\cos\theta$ 是光源速度 u 在视线方向上的投影。如果相对运动发生在光源与接受器的连线上，式 (8-14) 变为

$$\nu_t = \sqrt{\frac{c-u}{c+u}}\nu_s \tag{8-15}$$

这种情况称为纵向多普勒效应，当光源和接受器相互离开时，u 为正值，这时接受器收到的光的频率小于光源的发光频率；当光源和接受器相互接近时，u 为负值，这时接受器收到的光的频率大于光源的发光频率。

8.1.6　超声波

超声波是机械波的一种，它是频率大于 20kHz 的超过人耳听阈上限的机械波。机械波根据频率的不同可以分为三大类：次声波、声波和超声波。频率低于 20Hz 的波动称为次声；频率在 20Hz～20kHz 之间的波动称为声；频率在 20kHz 以上的波动称为超声，其中次声和超声人耳是听不到的。超声波与可闻声在本质上是一致的，即都是由于机械力产生的机械振动在介质中的传播而产生的，并且它们仅能在介质中传播，通常是以纵波的方式在弹性介质内传播。但由于超声波的频率高、波长短，所以在一定距离内沿直线传播并具有良好的束射性和方向性。

超声可通过某一适当频率的交变电场作用在具有压电效应的晶体片中，当电场方向与晶体压电轴方向一致时，压电晶片沿一定方向发生压缩和拉伸而形成振动，这种振动在介质中传播就形成了超声波的传播。在医疗诊断中，利用超声波传播过程中遇到不同声阻抗的界面发生反射的现象，使超声反射回来的声 (回声) 到达压电晶片中，这样其回声的机械能会转变为电能，在处理机上再将其电信号经过处理、放大，并显示在荧光屏上。当电信号显示为振幅高低不同的波形时称为 A 型超声诊断法，显示为点状回声扫描时称为 M 型超声诊断法，显示为灰度不同的点状回声进而组成图像时称为 B 型超声诊断法，显示为超声多普勒效应所产生的差频时称为 D 型超声诊断法。

8.2　医学成像系统的技术基础

医学成像技术将人体在生理和病理状态下的内部结构展现出来，从而促进了对人体生理过程的了解和对疾病的诊断，主要的医学成像系统有：X 射线成像，核医学成像，磁共振成像和超声成像等。其中 X 射线成像，核医学成像和磁共振成像是利用电磁波作为信息载体或测量手段，根据辐射物的性质和波长及它与组织的相互作用特性建立了相应的成像技术。超声成像是以组织的声学特性为基础的成像技术。

8.2.1　计算机断层扫描 (CT) 成像

X 射线成像是日常生活中最常用的医学成像技术，它使用 X 射线来透视病人以获得人体内部结构方面的信息，X 射线在人体内直线传播，不受组织界面的影响，可以提供不失真

的图像。最早出现的 X 射线成像是 X 射线投影成像，这种成像技术很难检测到那些被身体中其他不感兴趣区域的重叠图像掩盖的结构细节，并且深度信息也有损失。为了解决这一问题，出现了计算机断层扫描成像（Computerized Tomography，简称 CT）技术，它是利用 X 线进行断层扫描，电光子检测器接收扫描信号，然后把信号转化为数字信号输入到电子计算机内，再由计算机进行图像重建，从而有选择地记录了身体某一切面的图像。

按照时间的顺序，扫描器的发展可分为四代：

第一代扫描器是采用单束 X 射线源和单个检测器，X 射线管-检测器每进行一次扫描后就转动一个角度。在扫描中仅 X 射线源的一部分输出可以被利用，这样为了获得适合进行统计计算的数据，需要较长的扫描时间，通常整个扫描时间约为 5min。

第二代扫描器是采用多个检测器，如 10 个检测器，每个间隔 1°，X 射线管-检测器每进行一次扫描后就转动与第一代扫描器相应角度的 10 倍，整个扫描时间不到 1min。

第三代扫描器是使 X 射线管产生大角度扇形射线束，由几百个元件组成的圆周检测器阵列同步地绕病人身体转动，整个扫描时间约为 5s。

第四代扫描器与第三代扫描器相似，不同的是圆周检测器阵列固定地绕病人身体排列，扇形的 X 射线源绕病人身体转动，整个扫描时间约为 1s，如图 8-5（a）所示。检测器将接受到的信号转化成为数字信号，通常形成一个 512×512 个像素组成的图像矩阵，图像的每一个像素对应于厚度为 1～13mm，面积为 1×1mm 的一个组织体积元素，像素的亮度正比于体积元素的平均衰减系数。CT 扫描器能够记录到 0.5% 的 X 射线衰减变化量，主要提供轴向横截面图像。通过收集许多平行的投影图像，用图像重建技术重建目标物图像。

在重建算法中，假设选择的体积元素很小，认为其密度和原子数目是恒定的，X 射线的穿透强度 I 服从指数衰减定律，即

$$I = I_0 \exp\left[-(\mu_1 + \mu_2 + \cdots + \mu_n)x\right] \tag{8-16}$$

式中 I——入射强度；

μ_i——不同体积元素组织的全线性衰减因子；

x——体积元素的宽度。

对式（8-16）取对数，得

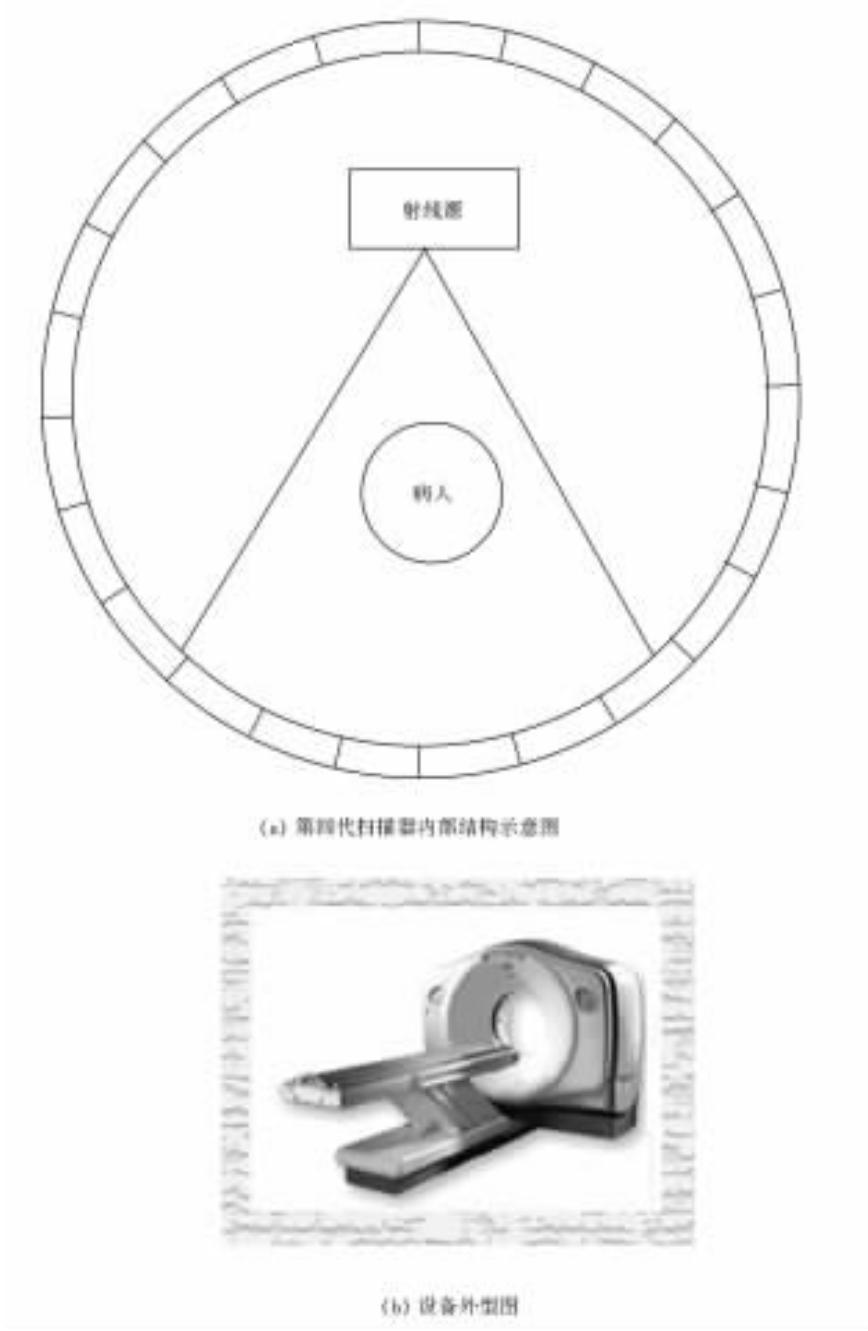
$$\ln \frac{I}{I_0} \propto \mu_i + \mu_2 + \cdots + \mu_n \tag{8-17}$$

可见穿透强度是由与射线束通过体积元素的全线性衰减因子之和（称为射线和 p）决定的，射线和 p 是图像重建的基本因素。目前有不同的投影图重建目标物的算法，如傅立叶重建算法和逆投影算法等。下面仅简单介绍一下逆投影算法，它将射线和逆投影到矩阵平面上，然后将像素不同投影点的值加在一起，即

$$f(x, y) = \sum_{j=1}^{m} p_j \Delta\Phi \tag{8-18}$$

式中 $f(x, y)$——间隔 $\Delta\Phi$ 的 m 次投影的射线和。

图 8-6 给出了一个使用简单逆投影算法的例子。假设有一个未知的矩阵（由 3，4，1，8 组成），如图 8-6（a）所示，第一次投影是在水平方向，其射线和为 $p_1=7$，$p_2=9$，将其逆投影到射线扫过的像素位置上；假设第二次投影与第一次投影相差 45°，如图 8-6（b）所示，其射线和为 $p_1=4$，$p_2=11$，$p_3=1$，将这些值叠加到相应的像素上；第三次投影是在垂直方向，如图 8-6（c）所示，其射线和为 $p_1=4$，$p_2=12$，将这些值叠加到相应的像素上；第四次



(a) 第四代扫描器内部结构示意图

(b) 设备外型图

图 8-5　第四代扫描器

投影相差 135°，**如图** 8-6 (d) **所示，其射线和为** $p_1=3$，$p_2=5$，$p_3=8$，**将这些值叠加到相应的像素上。现在进行最后一步，即从所有的值中减去背景值** 16，**并除以** 3，**答案是** 3，4，1，8，**如图** 8-6 (e) **所示，这样就得到了需要的值，完成了四次逆投影的重建过程。由上述可见，进行逆投影重建相对比较容易，但是每一次投影就形成一束扫过矩阵的高亮度区，结果** m **次投影就出现** $2m$ **个条幅的伪迹，当投影次数增加时，各个条幅综合出现一个高密度**

的中央斑，从而影响了图像的质量。为了解决这个问题，在逆投影重建过程中还要考虑使用相应的校正因子。

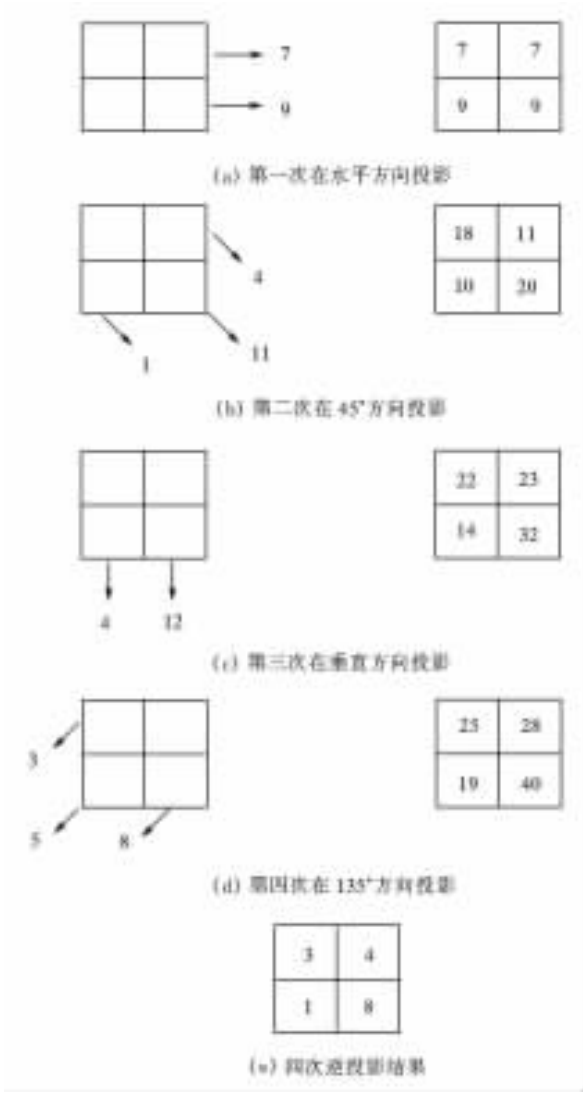


图 8-6 四次逆投影

8.2.2 正电子发射计算机断层扫描 (PET) 成像

正电子发射计算机断层扫描成像 (Positron Emission Computed Tomography，简称 PET) 是目前在分子水平上进行人体功能显像的最先进的医学影像技术。PET 的基本原理是利用加速器产生的超短半衰期同位素，如 18F、13N、15O、11C 等作为示踪剂注入人体，参与体内的生理生化代谢过程。这些超短半衰期同位素是组成人体的主要元素，利用它们发射的正电子与体内的负电子结合释放出一对 γ 光子，这对 γ 光子被探头的晶体所探测，经过计算机对原始数据重建处理，得到高分辨率、高清晰度的活体断层图像，以显示人脑、心、全身其他器官及肿瘤组织的生理和病理的功能及代谢情况。

图 8-7　GE 公司 PET 的外形

一台完整 PET 装置是由数据采集系统 (探头及其附属电路)、电子计算机系统 (具有大容量储存器的快速计算机、磁盘、磁带和打印机) 及图像显示系统 (包括图像分析设备) 等组成的，图 8-7 示出了 PET 的外形。

在 PET 中用共线对置的两个探测器采用快符合探测技术来探测正负电子湮没时间中辐射的两个相反方向的 γ 光子。在图 8-8 中，采用环形检测器，对所有那些到达环中任何两个共线对置的探测器的光子对进行记数

通过在一段时间间隔内不同角度上采集的 γ 光子对数，由计算机将收集到的数据进行处理，采用与 CT 中相同的逆投影重建技术来产生放射性同位素分布的横截面图像，即断层图像。PET 中检测器的排列不仅仅限于图 8-8 所示的环形，还可以采用平行板和六边形等。因为人体内重要有机物质可以用 C，N，O，F 的这些正电子发射的放射性同位素来标记，通过 PET 的检查可以分析该物质在人体内的分布随时间的变化，就可以得到这种生物分子在人体内的新陈代谢过程，这就在分子水平研究了人体内生物分子的作用。

可见 PET 是一种反映分子代谢的显像，当疾病早期处于分子水平变化阶段，病变区的形态结构尚未呈现异常，MRI，CT 检查还不能明确诊断时，PET 检查即可发现病灶所在，并可获得三维影像，还能进行定量分析，达到早期诊断，这是目前其他影像检查所无法匹敌的。MRI，CT 检查发现脏器有肿瘤时，是良性还是恶性很难作出判断，但 PET 检查可以根据肿瘤高代谢的特点而作出诊断。

8.2.3　磁共振 (MRI) 成像

磁共振成像 (Nuclear Magnetic Resonance，简称 MRI) 是当前最先进的医学成像设备之一。它在医疗诊断方面一个最突出的特点是对软组织的显像特别清晰，目前没有一种影像诊断设备能与磁共振成像 (MRI) 设备相匹敌，尤其是在提供脑、脊髓、骨骼肌肉的精细结构方面更没有比 MRI 成像更有效的设备，它在临床上被广泛地应用。

含单数质子的原子核，例如人体内广泛存在的氢原子核，其质子有自旋运动，带正电，产生磁矩，宛若一个小磁体 (图 8-9)。小磁体自旋轴的排列无一定规律。但如在均匀的强磁场中，小磁体的自旋轴则将按磁场磁力线的方向重新排列 (图 8-10)。

根据物理学原理，若某样品被放在均匀磁场 $\boldsymbol{B}_0$ 中，由拉莫尔公式有

$$\omega_0 = kB_0 \tag{8-19}$$

式中　ω_0——拉莫频率；

k——比例系数。

从式 (8-19) 可见，如果所用磁场是均匀的，那么对相同物质的所有质子来说，其拉莫频率都是相同的，则无法区分各点所产生的信号强度。如果在均匀的磁场 $\boldsymbol{B}_0$ 上叠加一个 z 方向的梯度磁场 G_z，则总的磁感应强度为

$$\boldsymbol{B} = \boldsymbol{B}_0 + z\boldsymbol{G}_z \tag{8-20}$$

那么在 Z 轴方向各点的拉莫频率亦随 z 线性增加，即

$$\omega = \omega_0 + kzG_z \tag{8-21}$$

拉莫频率成为位置 z 的函数，即通过 z 轴的同一 x-y 平面上所有的点具有相同的拉莫频率；

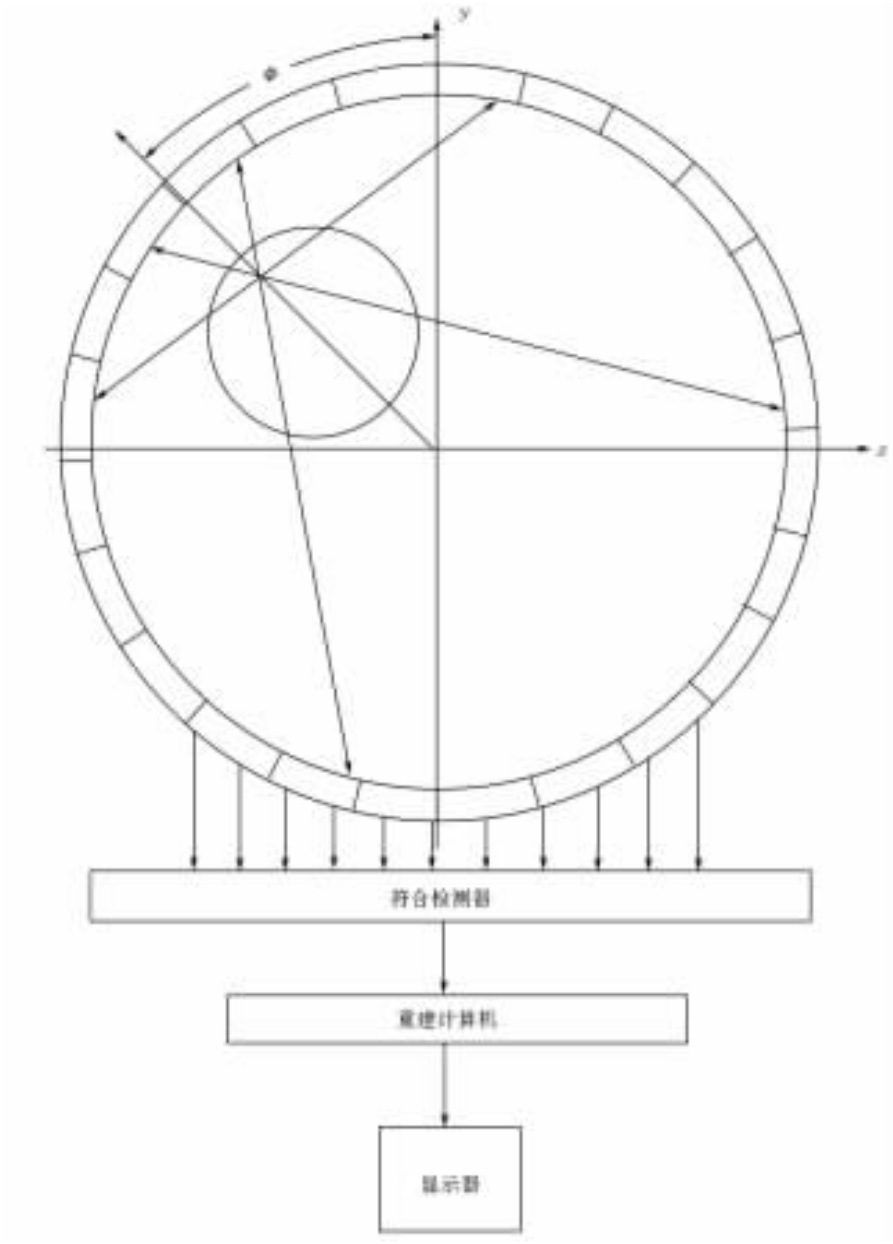


图 8-8 PET 中使用的环形检测器

同理若在 x 方向加一个梯度场，同一 y-z 平面上的点具有相同的拉莫频率，在 y 方向加一个梯度场，同一 x-z 平面上的点具有相同的拉莫频率。所以，在恒定磁场 $\boldsymbol{B}_0$ 上叠加三个互相垂直的梯度磁场，则可决定三个互相垂直的平面，而三个互相垂直平面的相交就决定了交点处的原子核的拉莫频率，这样就确定了拉莫频率的空间位置。所以为了得到人体的三维空间信息，需要对静磁场进行修改，即加上梯度磁场。

然后，用特定频率的射频脉冲（Radion-Frequency，简称 RF）产生的射频磁场照射处于磁场中的人体，人体中作为小磁体的氢原子核吸收一定的能量而发生了磁共振现象。当射频脉

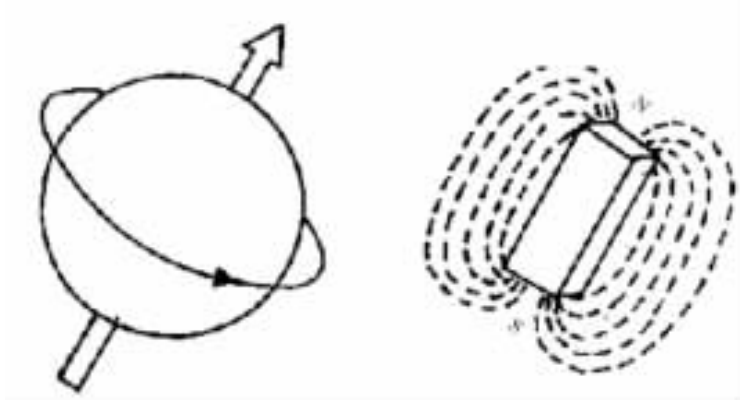


图 8-9　质子带正电荷，它们像地球一样在不停地绕轴旋转，并有自己的磁场

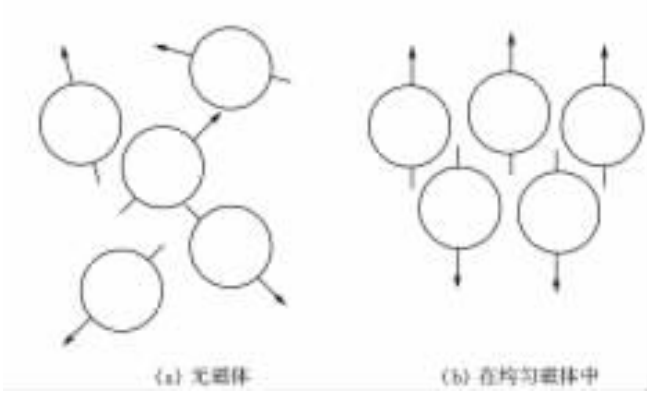


图 8-10　小磁体的自旋轴在磁场中方向重新排列

冲撤离后，被激发的氢原子核把所吸收的能量逐步释放出来，其相位和能级都恢复到激发前的状态。这一恢复过程称为弛豫过程，而恢复到原来平衡状态所需的时间则称之为弛豫时间。有两种弛豫时间：一种是自旋-晶格弛豫时间，又称纵向弛豫时间，它是反映自旋核把吸收的能量传给周围晶格所需要的时间，称为 T_1；另一种是自旋-自旋弛豫时间，又称横向弛豫时间，它是反映横向磁化衰减、丧失的过程，是横向磁化所维持的时间，称为 T_2。

人体不同器官的正常组织与病理组织的 T_1 是相对固定的，而且它们之间有一定的差别，T_2 也是如此。这种组织间弛豫时间上的差别就是 MRI 的成像基础。MRI 不像 CT 只有一个参数（吸收因子），而是有 T_1、T_2 和自旋核密度等几个参数，其中 T_1 与 T_2 尤为重要。

重建氢核密度和 T_1、T_2 加权图像的基本原理是，在图 8-11 所示的过程中，先发射一个 90°RF 脉冲，间隔数十毫秒后再发射一个 180°RF 脉冲，然后连续施加 180°RF 脉冲，并测量自旋回波信号的强度。这一系列回波信号的波峰之间的连线为一呈指数衰减的曲线，这个曲线代表了弛豫时间 T_2。T_1 取决于工作磁场，其形式如下：

$$T_1 \propto A\nu^B \tag{8-22}$$

式中　A，B——组织的特定常数；

ν——为工作磁场的频率。

90°脉冲至第一个回波的时间称回波时间，用 T_E 表示。90°脉冲至下一个 90°脉冲的时间

称重复时间，用 T_R 表示。当 $T_R \geqslant T_1$、$T_E \leqslant T_2$ 时，信号强度仅取决于质子密度，用这种信号重建的图像称为质子密度像。当 $T_R \leqslant T_1$、$T_E \leqslant T_2$ 时，信号强度取决于质子密度和 T_1，用这种方法重建的图像称为质子密度和 T_1 加权图像。当 $T_R \geqslant T_1$，$T_E \geqslant T_2$ 时，信号强度取决于质子密度和 T_2，用这种信号重建的图像称为质子密度和 T_2 加权图像。

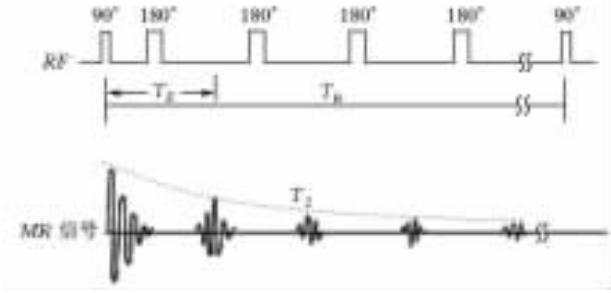


图 8-11 MRI 参数的确定

MRI 设备包括磁体、梯度线圈、射频发射器和 MR 信号接收器，计算机系统，如图 8-12 所示。磁体产生原子核极化所需的静磁场，直接关系到磁场强度、均匀度和稳定性，并影响 MRI 的图像质量。成像磁体分为四类：电流激励的空气芯磁体、电流激励的铁芯磁体、超导体和永磁体四种。通常用磁体类型来说明 MRI 设备的类型。电流激励的空气芯磁体和电流激励的铁芯磁体的线圈用铜、铝线绕成；超导体的线圈用铌-钛合金线绕成，用液氦

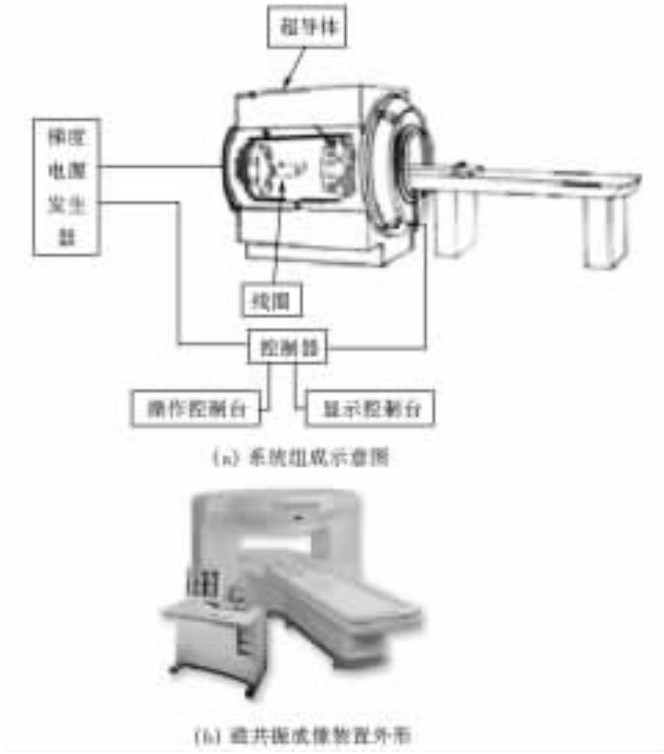


图 8-12 磁共振成像装置示意图

及液氮冷却；永磁型的磁体由用磁性物质制成的磁砖所组成，较重、磁场强度偏低。

梯度线圈产生梯度磁场。其磁场强度虽只有主磁场的几百分之一，但梯度磁场为人体 MR 信号提供了空间定位的三维编码的可能，梯度场由 x、y、z 三个梯度磁场线圈组成，并有驱动器以便在扫描过程中快速改变磁场的方向与强度，迅速完成三维编码。

射频发射器与 MR 信号接收器称为射频系统，射频发射器是为了产生临床检查目的不同的脉冲序列，以激发人体内氢原子核产生 MR 信号。射频发射器及射频线圈很像一个短波发射台及发射天线，向人体发射脉冲，人体内氢原子核相当一台收音机接收脉冲。脉冲停止发射后，人体氢原子核变成一个短波发射台，而 MR 信号接受器则成为一台收音机接收 MR 信号。脉冲序列发射完全在计算机控制下进行。

计算机系统承担整个系统的控制、指挥和信息处理等，它从磁共振信号中萃取出 T_1、T_2 和核密度参数，实现图像重建和图像显示。

8.2.4　超声多普勒成像

前面介绍的成像技术均是以电磁波作为信息载体，根据辐射物的性质和波长及与组织的相互作用，形成了基于不同辐射类型的特定成像原理。而超声成像则不同，它是以组织的机械波为基础的。超声多普勒成像是超声成像的一种，超声多普勒系统是根据多普勒效应，即由于发射器和检测器的相对运动或者波被运动目标反射时产生的波频的显著变化，来检测这一频率移动能够记录的运动和流动。其连续波多普勒发射一恒定超声速，它能够提供速度信息。而脉冲多普勒能够提供空间信息和速度信息。生物介质有着非常复杂和强烈的多点散射界面，当超声射到皮肤表面时，部分透过皮肤作用到微血管，经运动的血流散射回的声将根据多普勒效应发生频移 (波长发生改变)，而射到静止结构和血管床时，反射回来后声波不发生频移，波长变化的程度及频率分布与血细胞的数量和运动速度有关，而与运动方向无关。超声多普勒系统框图如图 8-13 所示。发射系统由振荡器、功率放大器和发射探头组成，用

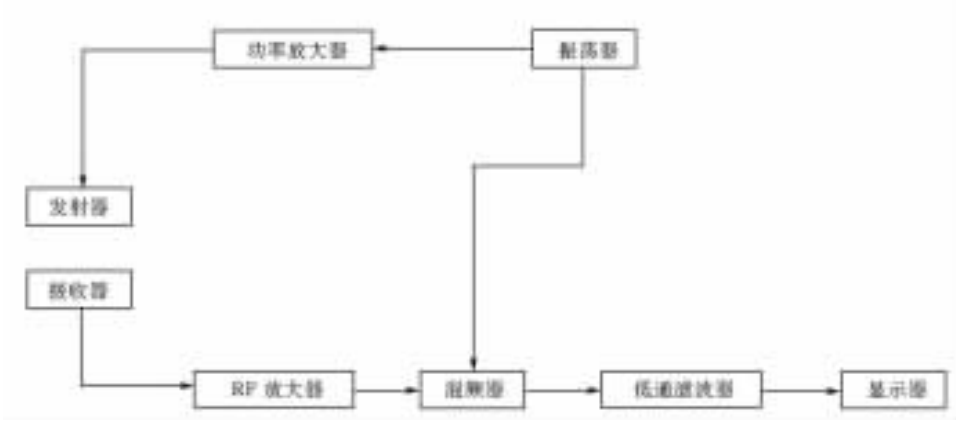


图 8-13　超声多普勒系统框图

来发射超声波。超声波射到被测组织中，撞击运动的血细胞的一部分超声波被反射回来，波长发生改变 (多普勒频移)，而射到静止组织的超声波，反射回来后波长不变。接收器是与发射探头对应的接收探头，来接收这些回波信号，这些信息经过放大后，在混频率器中与发射信号相乘，产生出和频与差频，由于生理信号处于低频范围，所以对信号进行低通滤波，再

由模-数转换器，输入到计算机内形成二维断面图像。图 8-14 示出了设备的外形图。

图 8-14 设备的外形图

8.3 常用医学图像处理方法

8.3.1 图像修正

1. 几何修正

在成像过程中，由于噪声等干扰的存在，经常出现图像几何形状歪曲的现象。图 8-15 示出了成像光学系统干扰造成的正、负图像歪曲现象。对于这样的情况可以通过几何修正的方法对图像进行处理。

假设歪曲图像的坐标系是 (x, y)，修正后图像的坐标系是 (x', y')，两者的变换关系为

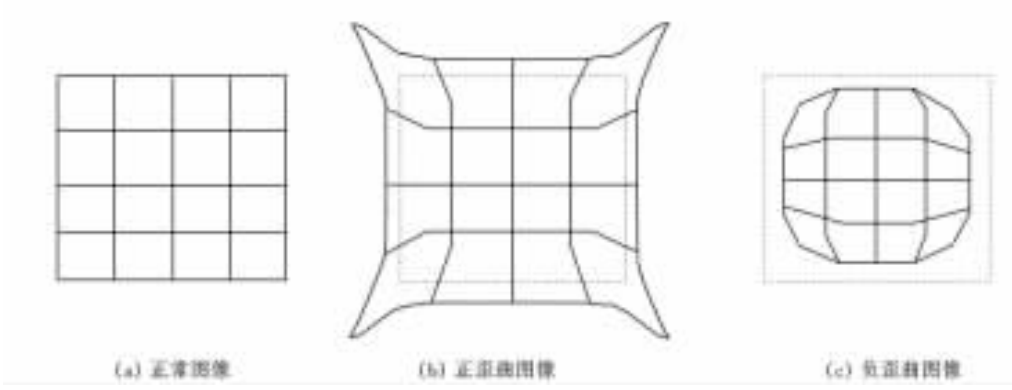


图 8-15 图像歪曲现象

$$s'(x', y') = s(x, y) \tag{8-23}$$

其中 $x' = f_1(x, y)$，$y' = f_2(x, y)$，f_1 和 f_2 可以表示成 n 阶多项式的形式，即

$$f_1(x, y) = a_0 + a_1x + a_2y + a_3x^2 + a_4xy + a_5y^2 + \cdots \tag{8-24}$$

$$f_2(x, y) = b_0 + b_1x + b_2y + b_3x^2 + b_4xy + b_5y^2 + \cdots \tag{8-25}$$

对于不同的图像，系数 a_i 和 b_i 不同，为了进行图像修正，需要确定系数 a_i 和 b_i，对此可以采取不同的方法。这里采用的方法是，对应图像中的某一点 (x_i, y_i) 取一个测量点 (u_i, v_i)，一共取 P 对图像中的样点与测量点，要求其测量的均方误差为最小，对此应满足式 (8-26) 和式 (8-27)

$$\sum_{i=0}^{P-1} [v_i - (a_0 + a_1x + a_2y + a_3x^2 + a_4xy + a_5y^2 + \cdots)]^2 \Rightarrow \min \tag{8-26}$$

$$\sum_{i=0}^{P-1} [u_i - (b_0 + b_1x + b_2y + b_3x^2 + b_4xy + b_5y^2 + \cdots)]^2 \Rightarrow \min \tag{8-27}$$

根据式 (8-26) 和式 (8-27) 可以求出使测量均方误差最小的系数 a_i 和 b_i，将其代入式 (8-23)、式 (8-24) 和式 (8-25) 中就求出了校正后的坐标系，实现了对歪曲图像的校正。

图 8-16 是口腔 X 射线成像的例子，放射线镜筒放置于患者的脸部，射线的聚焦点距牙齿的距离大约为 25cm，射线成锥形照射在牙齿上，如果对歪曲图像中某点 (x, y) 取测量点

(u, v)，该点校正后的位置为 (x', y')，根据前面叙述的原理，求得校正后图像与歪曲图像间的变换关系为

$$x' = f_1(x, y) = \frac{a_1 x + a_2 y + a_3}{a_7 x + a_8 y + 1} \tag{8-28}$$

$$y' = f_2(x, y) = \frac{a_4 x + a_5 y + a_6}{a_7 x + a_8 y + 1} \tag{8-29}$$

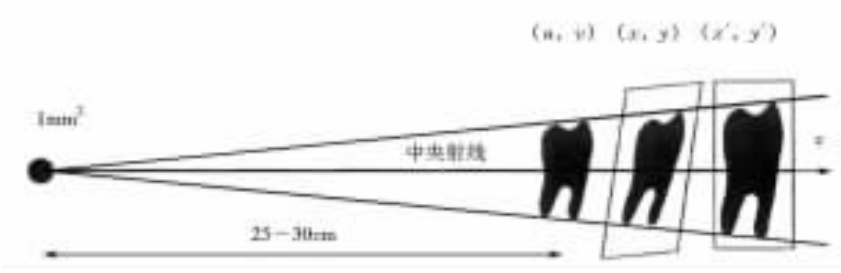


图 8-16　口腔 X 射线成像的例子

2. 用信号模型进行图像校正

这里将图像的歪曲过程视为一个系统，如图 8-17 所示，假设正确图像函数为 $f(x, y)$，歪曲图像函数 $g(x, y)$ 为

$$g(x, y) = [f(x, y) + n_1(x, y)]h(x, y) + n_2(x, y) \tag{8-30}$$

其中 $n_1(x, y)$ 和 $n_2(x, y)$ 为干扰噪声，$h(x, y)$ 为歪曲系统的系统函数。

图 8-17　图像歪曲过程示意图

为了对歪曲图像 $g(x, y)$ 进行校正，选取滤波函数 $q(x, y)$ 对歪曲图像 $g(x, y)$ 进行滤波，如图 8-18 所示。

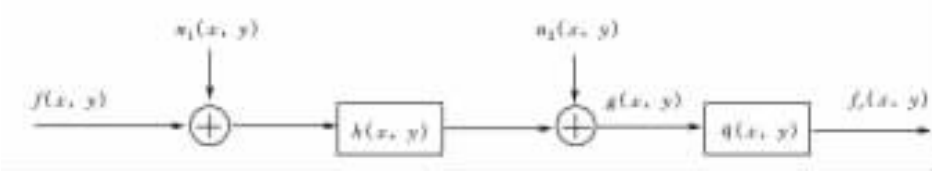


图 8-18　对歪曲图像进行校正的示意图

滤波后得到校正函数 $f_r(x, y)$ 为

$$f_r(x, y) = g(x, y) * q(x, y) \tag{8-31}$$

对式 (8-30) 两侧取傅立叶变换，得

$$G(u, v) = [F(u, v) + N_1(u, v)]H(u, v) + N_2(u, v) \tag{8-32}$$

式中　$G(u, v)$，$F(u, v)$ —— 测量函数 $g(x, y)$ 和原始函数 $f(x, y)$ 的傅立叶变换；

$N_1(u, v)$，$N_2(u, v)$ —— 噪声的傅立叶变换；

$H(u, v)$ ——系统的传递函数。

根据 Parsevalschen 理论 (T. Lehmann et al.)，校正函数应满足

$$\sum_u \sum_v \left| G(u, v) - F_r(u, v) H(u, v) \right|^2 \Rightarrow 0 \tag{8-33}$$

这样得出测量函数与校正函数傅立叶变换的关系

$$F_r(u, v) = G(u, v) H(u, v)^{-1} \tag{8-34}$$

将式 (8-32) 代入式 (8-34) 得

$$F_r(u, v) = [F(u, v) + N_1(u, v)] + N_2(u, v) H(u, v)^{-1} \tag{8-35}$$

式 (8-35) 给出原始函数与校正函数傅立叶变换的关系。

3. 图像修正的线性方法

用线性方法对图像进行修正最常用的是改变图像的对比度，如果一幅图像具有 M 行、N 列，该图像的总对比度可表示为

$$Kg = \frac{\max[f(m, n)] - \min[f(m, n)]}{G - 1} \tag{8-36}$$

式中 $f(m, n)$ ——图像的灰度；

$G - 1$——图像的灰度的量化范围。

如果图像灰度的平均值为 f'，则灰度值 $f(m, n)$ 相对平均值的均方误差为

$$K_v = \frac{4}{G^2 MN} \sum_{m=0}^{M-1} \sum_{n=0}^{M-1} [f(m, n) - f']^2 \tag{8-37}$$

在对比度修正中，根据式 (8-36) 和式 (8-37)，首先设定一个阈值，可以令大于阈值的灰度值为 1，小于阈值的灰度值为 0；或反之，来改变对比度。

另一种方法是借用直方图，将直方图 $p(f)$ 表示成为 δ 函数的形式，即

$$p(f) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \delta[f(m, n) - f'] \tag{8-38}$$

其中

$$\delta(\varepsilon) = \begin{cases} 1 & \text{当 } \varepsilon = 0 \\ 0 & \text{其他} \end{cases}$$

式 (8-38) 所示函数的熵 $H(f)$ (信息的包含量) 定义为

$$H(f) = \frac{1}{MN} p(f) \tag{8-39}$$

根据 $H(f)$ 定义图像的对比度如下

$$K_e = \frac{-1}{\log_2 G} \sum_{f=0}^{G-1} H(f) \log_2 [H(f)] \tag{8-40}$$

在医院临床中经常用上式比较两幅同一部位，不同时间拍摄的图片，来找出两者的差异，如图 8-19 所示。图 8-19 (a) 是在病人的骨头内扎入一个铁钉的 X 射线图片。三个月后在同一部位又拍摄了一张 X 射线图片如图 8-19 (b) 所示。将两幅图片对齐，并使之具有相同的直方图后，可见在铁钉的右上部有骨头坏损，如图 8-19 (c) 所示。图 8-19 (d) 是根据式 (8-40) 得到的对比度差，从中可见骨头坏损部位。

4. 图像修正的非线性方法

为了减弱图像中的局部干扰，通常不使用图像的线性修正方法，而是选择合适的模板，

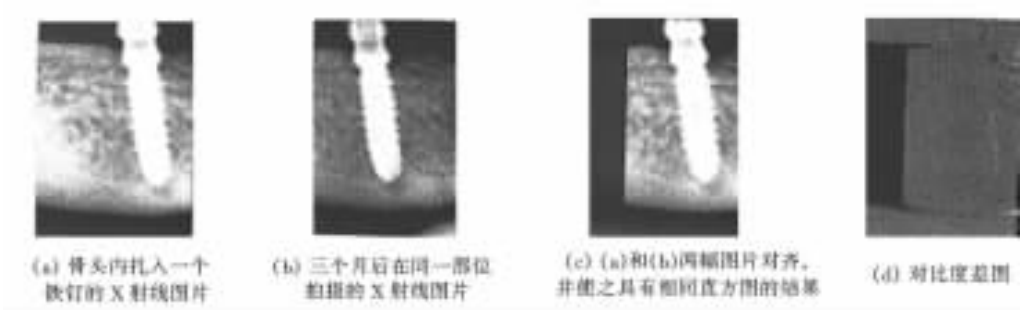


图 8-19　X 射线图片处理例子 (选自 T.Lehmann et al.)

用中值滤波的方法 (非线性修正方法) 进行处理。图 8-20 示出了三种中值滤波的模板，图 8-20 (a) 是用来消除点干扰的模板，图 8-20 (b) 是用来消除行干扰的模板，图 8-20 (c) 是用来消除列干扰的模板。图 8-21 示出了选择 7×7 的点模板进行中值滤波的结果。

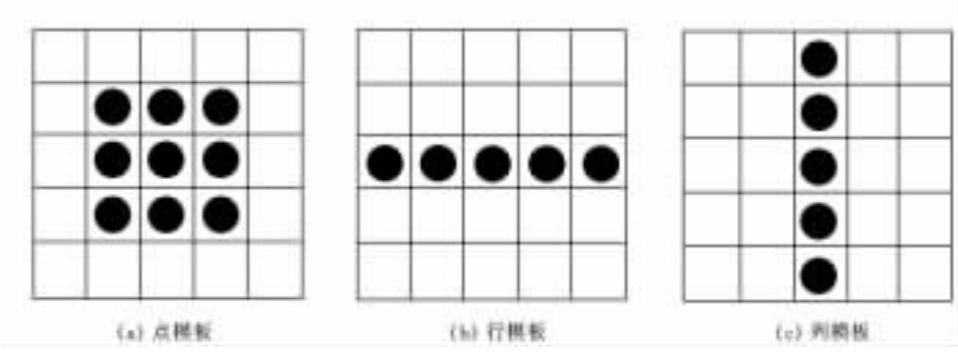


图 8-20　中值滤波模板 (选自 T.Lehmann et al.)

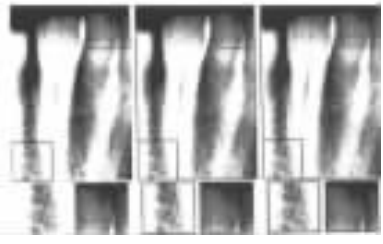


图 8-21　对各个图像标志区域 7x7 中值滤波的结果 (选自 T.Lehmann et al.)

此外还可以使用局部直方图均匀化处理的方法进行图像修正，图 8-22 (a) 为原始图像；图 8-22 (b) 是进行全图直方图均匀化处理后的效果；图 8-22 (c) 是进行 25×25 的局部直方图均匀化处理后的效果，可见经过局部直方图均匀化处理后，图像的细节清晰。

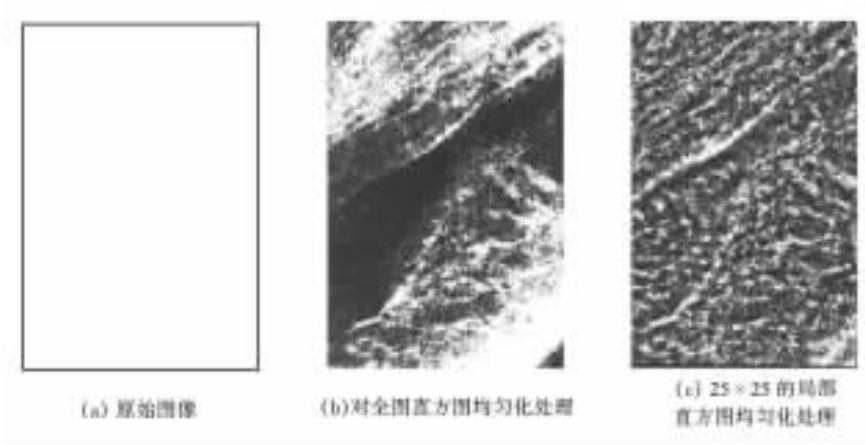


图 8-22 直方图均匀化处理后的结果 (选自 T. Lehmann et al.)

程序清单 8-1 示出了对图像进行旋转的 Java 代码，该程序运行结果如图 8-23 所示。

程序清单 8-1 Mirror.java **源代码**

```
//Mirror.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class Mirror extends Frame {
    Image im, tmp;
    int iw, ih;
    int[] pixels;
    boolean flagLoad = false;

    public Mirror(){
        this.setTitle("图像的水平和垂直镜像");
        Panel pdown;
        Button load, horizon, vertical, quit;

        //添加窗口监听事件
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
```

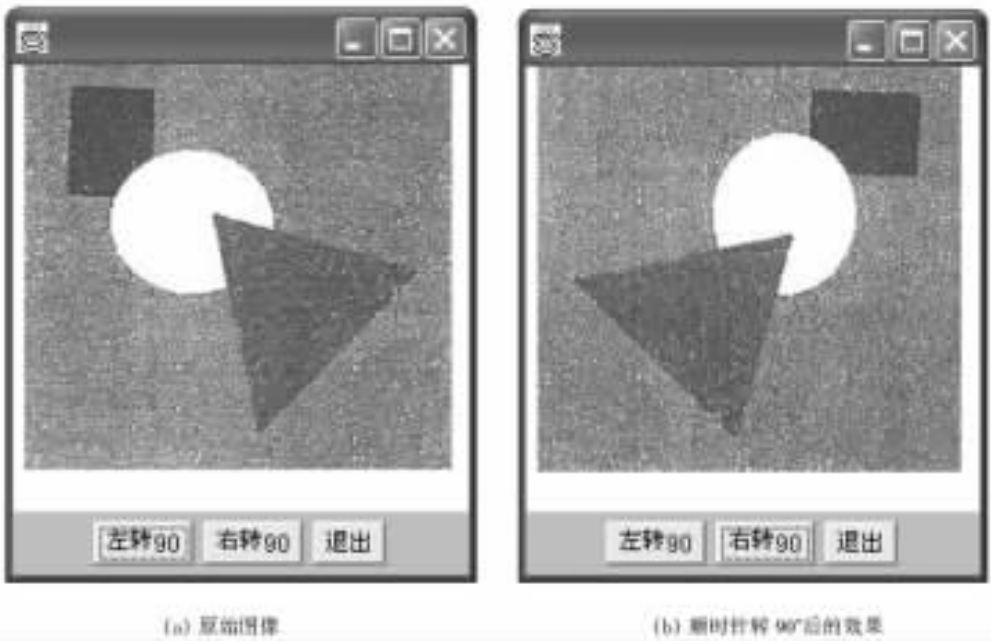
```
        pdown = new Panel();
        pdown.setBackground(Color.lightGray);

        load=new Button("加载图像");
        horizon=new Button("水平镜像");
        vertical= new Button("垂直镜像");
        quit=new Button("退出");

        this.add(pdown,BorderLayout.SOUTH);

        pdown.add(load);
        pdown.add(horizon);
        pdown.add(vertical);
        pdown.add(quit);

        load.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jLoad_ActionPerformed(e);
            }
        });

        horizon.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                jHorizon_ActionPerformed(e);
            }
        });

        vertical.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
                jVertical_ActionPerformed(e);
            }
        });

        quit.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
                jQuit_ActionPerformed(e);
            }
        });
    }
```

```
public void jLoad_ActionPerformed(ActionEvent e){
    //利用 MediaTracker 跟踪图像的加载
    MediaTracker tracker = new MediaTracker(this);
    im=Toolkit.getDefaultToolkit().getImage("Miss.jpg");
    tracker.addImage(im,0);

    //等待图像的完全加载
    try{
    tracker.waitForID(0);
    }catch(InterruptedException e2){ e2.printStackTrace();}

    //获取图像的宽度 iw 和高度 ih
    iw=im.getWidth(this);
    ih=im.getHeight(this);
    pixels=new int[iw*ih];

    try{
    PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
    pg.grabPixels();
    }catch (InterruptedException e3) {
        e3.printStackTrace();
    }

    //将数组中的像素产生一个图像
    ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
    tmp=createImage(ip);
    flagLoad=true;
    repaint();
}

public void jHorizon_ActionPerformed(ActionEvent e){
    if(flagLoad){
    //可以进行连续的镜像!
    /*
    try{
    PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
    pg.grabPixels();
    }catch (InterruptedException e3) {
        e3.printStackTrace();
```

```
            }
            */

            //对图像进行水平镜像,Alpha 值保持不变
            int [] tempPixels=new int[iw*ih];
            for(int i=0;i<iw*ih;i++)
            {
                tempPixels[i]=pixels[i];
            }

            for(int i=0;i<ih;i++)
            {
                for(int j=0;j<iw;j++)
            {
                //核心算法:第一列变为最后一列
                pixels[i*iw+j]=tempPixels[i*iw+(iw-j-1)];
            }
        }

        //将数组中的像素产生一个图像
        ImageProducer ip=new MemoryImageSource(iw,ih,pixels,0,iw);
        tmp=createImage(ip);
        repaint();
        }else{
        JOptionPane.showMessageDialog(null,"请先打开一幅图片!",
                    "Alert",JOptionPane.WARNING_MESSAGE);
        }
}

public void jVertical_ActionPerformed(ActionEvent e){
        if(flagLoad){
        //可以进行连续的镜像
        /*
        try{
        PixelGrabber pg=new PixelGrabber(im,0,0,iw,ih,pixels,0,iw);
        pg.grabPixels();
        }catch (InterruptedException e3) {
            e3.printStackTrace();
        }
        */
```

```
        //对图像进行垂直镜像,Alpha 值保持不变
        int [ ] tempPixels = new int[iw * ih];
        for(int i = 0;i < iw * ih;i ++ )
        {
            tempPixels[i] = pixels[i];
        }

        for(int i = 0;i < ih;i ++ )
        {
            for(int j = 0;j < iw;j ++ )
            {
            //核心算法:第一行变为最后一行
            pixels[i * iw + j] = tempPixels[(ih - i - 1) * iw + j];
            }
        }

        //将数组中的像素产生一个图像
        ImageProducer ip = new MemoryImageSource(iw, ih, pixels, 0, iw);
        tmp = createImage(ip);
        repaint();
        }else{
        JOptionPane.showMessageDialog(null, "请先打开一幅图片!",
                "Alert", JOptionPane.WARNING_MESSAGE);
        }
    }

    //程序退出
    public void jQuit_ActionPerformed(ActionEvent e){
        //System.exit(0);
        JOptionPane op  = new JOptionPane();
        int exit = op.showConfirmDialog(this, "你要退出吗? ? ?", "退出", JOptionPane.YES_
NO_OPTION);

        if(exit = = JOptionPane.YES_OPTION)
        {
            System.exit(0);

        }else{ }
    }
```

```
//调用 paint()方法,显示图像信息。
public void paint(Graphics g){
    if(flagLoad){
        g.drawImage(tmp,10,20,this);
    }
    else{ }
}

    //定义 main 方法,设置窗口的大小,显示窗口
    public static void main(String[ ] args) {
        Mirror mirror = new Mirror();
        mirror.setLocation(50,50);
        mirror.setSize(500,400);
        mirror.show();
    }
}
```



(a) 原始图像　　(b) 顺时针转 90°后的效果

图 8-23　图像进行旋转

8.3.2　图像分析

图像分析是图像处理的重要部分，它包括图像区域分割与图像描述。对同一图像区域采用不同的描述方法直接影响对图像的理解。图 8-24 就是一个例子，其中图 8-24 (a) 是原始图像，将这个图像分为三个区域，但对此可以有不同的描述，图 8-24 (b) 是一种描述，图 8-24 (c) 是另一种描述。

图 8-24 对图像区域不同的描述 (选自 T.Lehmann et al.)

下面以点处理、边缘处理和区域处理对此加以介绍。

1. 点处理

点处理是一种运算简单、快捷的方法。典型的点处理是阈值处理法，它是借助直方图求出阈值。图 8-25 示出了一个直方图，这时灰度阈值可由式 (8-41) 给出

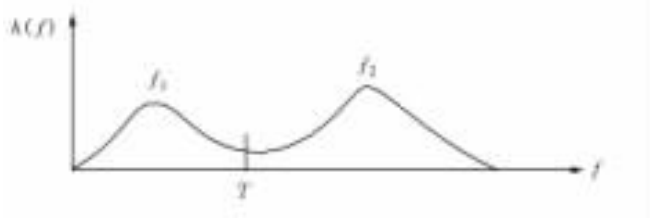


图 8-25 根据直方图确定阈值

$$T = \frac{| f_1 - f_2 |}{2} \tag{8-41}$$

这里

$$f(x, y) = \begin{cases} 0 & \text{当 } f \leqslant T \\ G-1 & \text{当 } f > T \end{cases} \tag{8-42}$$

或

$$f(x, y) = \begin{cases} 0 & \text{当 } f \geqslant T \\ G-1 & \text{当 } f < T \end{cases} \tag{8-43}$$

这样可以将目标物与背景分割开。

对于灰度阈值的选取还可以取其他的方法，例如，若直方图中样点出现的概率分布满足高斯分布，这样直方图可以表达为

$$p_q(f) = \sum_k a_k \exp\left[-\frac{(f-u_k)^2}{\sigma_k^2}\right] = \sum_k q_k \tag{8-44}$$

式中 μ_k——灰度平均值；

σ_k——偏差。

要求实际直方图 $p(f)$ 与理论近似直方图 $p_q(f)$ 满足条件

$$\sum_f | p(f) - p_q(f) | \Rightarrow \min \tag{8-45}$$

由两个相邻的高斯分布 q_k 和 q_{k+1} 可以求出灰度阈值

$$\frac{\mathrm{d}}{\mathrm{d}T}\left[\int_0^T q_{k+1}(f)\,\mathrm{d}f \int_T^{G-1} q_k(f)\,\mathrm{d}f\right] = 0 \tag{8-46}$$

在很多情况图像不是很简单的，不能用上述的方法定义灰度阈值。比如在图像中的某个感兴趣的区域内有一个很小的目标物，这时就应选择局域灰度阈值。

2. 边缘处理

边缘处理是医学图像处理的常用方法，其基本原理已经在第 7 章中介绍过，不再赘述。图 8-26 示出了几种边缘提取方法的效果，图 8-26 (a) 是原始图像；图 8-26 (b) 是采用梯度边缘提取法；图 8-26 (c) 是采用 Sobel 边缘提取法；图 8-26 (d) 是首先进行二值化，然后进行了中值滤波及采用 Sobel 边缘提取法；图 8-26 (e) 是采用 Khoros DRF 边缘滤波法；图 8-26 (f) 是采用 Khoros GEF 边缘滤波法。

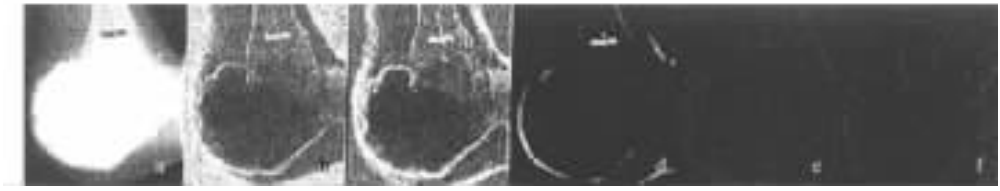


图 8-26　边缘处理 (选自 T.Lehmann et al.)

3. 区域处理

与点处理不同，区域处理要涉及各个区域间的相互关系，区域处理的基本过程是选择相邻区域内的一个或几个像素点，然后确定像素点间的灰度距离，以此为判据，来讨论相邻区域的性质。

像素点间距离测量的一个主要目的是确定两个区域间的相似性，最简单的情况是每个区域只有一个点，那么两个样点间的灰度距离为

$$D(X_1, X_2) = |f_1 - f_2| \tag{8-47}$$

式中　f_1 和 f_2 是像素点 X_1 和 X_2 的灰度值，可以根据灰度距离 D 来确定是否某一像素点属于所讨论的区域。

现在考虑两个相邻的区域A和B，如图 8-27 所示，要判断是否区域A和B是相似的，可以合并为一个区域，对此有不同的处理方法。这里假设有两个观测像素点 X_A 和 X_B，并且 $X_A \in A$，$X_B \in B$，T 为灰度距离阈值，如果

$$D(X_A, X_B) \leqslant T \tag{8-48}$$

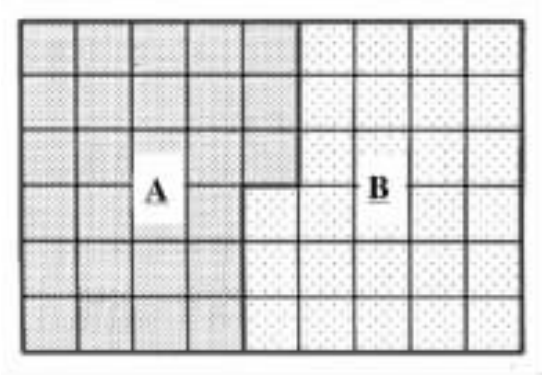


图 8-27　区域处理 (选自 T.Lehmann et al.)

并且区域A和B是均匀的，则可以认为区域A和B是可以合并为一个区域。或者选择多个像素点对，若每一对都满足式 (8-48)，可以认为区域A和B是可以合并为一个区域。

图 8-28 示出了又一个例子，区域A有三个相邻的区域B、C和D，这时不能用上述的方法处理，即只是讨论两个相邻区域或两个相邻区域对应的像素点，而是要对所有的区域进行整体的讨论。假设区域A和B间的平均灰度距离为 $D(\underline{A},\underline{B})=0.4$，区域A和C间的平均灰度距离为 $D(\underline{A},\underline{C})=0.65$，区域A和D间的平均灰度距离为 $D(\underline{A},\underline{D})=0.35$，这里令灰度距离阈值 $T=0.6$，要求当

$$D_{\min}=\min\{D(\underline{A},\underline{B}),D(\underline{A},\underline{C}),D(\underline{A},\underline{D})\}\leqslant T \tag{8-49}$$

满足的区域最先合并，可见区域A和D最先合并为一个区域，然后再继续判断，即

$$D'_{\min}=\min\{(\underline{A}+\underline{D},\underline{B}),D(\underline{A}+\underline{D},\underline{C})\} \tag{8-50}$$

若 $D'_{\min}\geqslant T$，则不能再进行区域合并了。

除上述方法外，还有其他处理方法，这里就不一一叙述了。

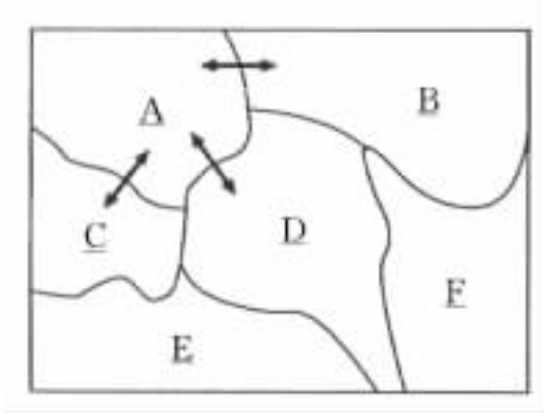


图 8-28 区域处理 (选自 T. Lehmann et al.)

练 习 题

1 试述 PET 成像、MRI 成像和超声多普勒成像的异同点。

2 试用 Java 语言编写对图 8-21 中各个图像进行 7×7 中值滤波的程序。

参 考 文 献

[1] Efford N. Digital Image Processing. USA：Addison-Wesley，2000

[2] Lehmann T，Oberschelp W，Pelikan E and Repges R. Bilderverarbeitung fuer die Medizin. Germany：Springer-Verlag，1997

[3] Breuer H. Taschenatlas Physik fuer Mediziner. Germany：Springer-Verlag，1989

[4] Gonzalez R C and Wintz P. Digital Image Processing. USA：Addison-Wesley，1977

[5] Rosenfeld A and Kak A C. Digital Picture Processing. London：Academic Press，1976

[6] Ehricke H H. Medical Imaging. Germany：Frieder Vieweg & Sohn Verlagsgesellschaft mbH，1997

[7] 徐飞，施晓红编著. MATLAB 应用图像处理. 西安：西安电子科技大学出版社，2002

[8] 刘榴娣，刘明奇，党长民编著. 实用数字图像处理. 北京：北京理工大学出版社，1998

[9] Kenneth. R Castleman. 数字图像处理. 朱志刚，林学阁，石定机等译. 北京：电子工业出版社，1998

[10] 杨福家，王炎林，陆福全著. 原子核物理. 上海：复旦大学出版社，1993

[11] 孙燕主编. Java2 入门与实例教程. 北京：中国铁道出版社，2003

[12] 虞颂庭，翁铭庆主编. 生物医学工程的基础与临床. 天津：天津科学技术出版社，1988

[13] 胡昌华，张军波，夏军，张伟编著. 基于 MATLAB 的系统分析与设计——小波分析. 西安：西安电子科技大学出版社，1999

[14] 吴思诚，王祖铨主编. 近代物理实验. 北京：北京大学出版社，1995

[15] 陈宜生，周佩瑶，冯艳全编. 物理效应及其应用. 天津：天津大学出版社，1995

附　录

附录 A　Java 语言简介

Java 语言是在 1990 年 12 月由 Sun Microsystems 作为 Green 项目的一部分而开发的。Java 语言最初被命名为 Oak 语言，后来改为 Java。Java 语言最初是为嵌入式电子设备而设计的，而不是为现在流行的 PC 而设计的，所以它必须小巧、高效、可靠，而且很容易移植到范围更广泛的硬件设备上。

Java 是一种编程语言，它适合设计与 Internet 有关的软件。Java 是一种跨平台的编程语言，Java 程序无需修改就可在 Windows、Linux、Macintosh、Solaris 和其他系统上运行。Java 是一种面向对象的编程语言。面向对象的编程方法是进行计算机程序设计的一种高效方法，而且编写的程序易于理解、修改和复用。

Sun 每发布一个新的 Java 版本，都通过 Web 发布一个免费使用的开发工具包。例如，本书中使用的是 SDK1.4，全称是 Java Development Kit 1.4。读者可以在 Sun 的 Java Web 站点http：//java.sun.com，免费下载最新的 SDK 开发工具包。

Java 语言易于学习，这也是本书的程序选用 Java 语言的一个原因。Java 语言比其他面向对象的编程语言易于编写、编译、调试和学习。它以 C++ 语言为原型，并借鉴了 C++ 语言的许多语法。如果学习过 C++ 语言，可以很容易的转到 Java 语言上来。Java 语言放弃了很容易出错的指针操作，使很多复杂的问题都得到了解决。内存管理是 Java 自动进行的，而不是要求编程人员分配和释放内存，而且不支持多重继承。

除了 Java 开发工具包 SDK 以外，程序员还可以使用很多免费的或者商业的开发工具。其中，比较流行的有：Borland 公司的 Jbuilder、IBM 公司的 Visual Age for Java、Sun 公司的 Forte for Java 等等。对于初学者，作者建议采用 JDK1.4 + Jcreator。Jcreator 是一个免费的 Java 语言编辑器。

附录 B　JDK1.4 的安装及配置

1) 在 Win 9X 下,单击 j2sdk - 1_4_2 - windows - i586.exe,按照提示安装到:c:\j2sdk1.4.2。在 c:\autoexec.bat 文件中加入以下各行:

SET PATH=c:\jdk1.4.2\bin;c:\jdk1.4.2\jre\bin //指定 java 路径
SET CLASSPATH=c:\jdk1.4.2\lib;c:\jdk1.4.2\jre\lib //指定类路径
SET JAVA_HOME=c:\jdk1.4.2

2) 在 Win2000 和 Windows XP 下,单击 j2sdk - 1_4_2 - windows - i586.exe,按照提示安装到:c:\j2sdk1.4.2。选择"我的电脑"右键 - >属性 - >高级 - >环境变量,在用户变量中加入 JAVA_HOME=c:\j2sdk1.4.2 ,在系统变量的 Path 变量的末尾加入 ;c:\j2sdk1.4.2\bin 即可。

3) 在 Linux 下,1.JDK1.4.1 安装:

下面的命令把 j2sdk1.4.1 拷贝到/usr/local 目录中

#cp j2sdk - 1_4_1 - linux - i586 - rpm.bin /usr/local

下面的命令进入该目录

#cd /usr/local

下面的命令自动产生 j2sdk - 1.4.0.i386.rpm 文件

#./j2sdk - 1_4_1 - linux - i586 - rpm.bin

解压此文件,并产生/usr/local/jdk1.4.1 目录

#rpm - ivh j2sdk - 1.4.1.i586.rpm

建立 jdk 快捷方式

#ln - s jdk1.4.1 jdk

建立 jre 快捷方式

#ln - s jdk/jre

安装和设置好了以后,可以在 Window 和 Linux 的命令行下面,输入 java 命令和 javac 命令,按回车键,看一下系统是否能够识别这些命令。如果系统能够识别,证明已经安装成功 如果不能识别,说明 jdk 的设置有问题,需要重新设置。在 Win9X 下,单击"开始",选择"运行",然后输入 command,按回车键,就可以进入 DOS 命令环境。在 Win2000 和 Windows XP 下,单击"开始",选择"运行",然后输入 cmd,按回车键,也可以进入 DOS 命令环境。

附录 C 光盘说明

本书所附 CD-ROM 光盘包含下列目录：

CODE

本书的源程序均在该目录中，每章的程序分别对应于该目录中的相应子目录。同时，也包含了本书中用到的一些图片。本书全部程序在支持 Java 2 平台的 JDK1.4.0 和 JDK1.4.1 上测试通过，并且相信也能在更高的 JDK 版本和企业版上运行通过。

SOFT

该目录中有一个用 JAVA 编写的一个小游戏——俄罗斯方块，并且给出了全部的源代码和详细的说明。

Sun 公司每发布一个新的 Java 版本，都通过 Web 发布免费使用的开发工具包，其中包括 Windows 平台和 Linux 平台的软件包和开发文档等。读者可以在 Sun 的 Java Web 站点 http://java.sun.com，免费下载最新的 SDK 开发工具包．具体安装过程参考：JDK 的安装.txt 文件。