第3章 数字图像处理基础

本章学习目标

- 理解数字图像特征和读取方法。
- 掌握图像的转换技术。
- 掌握图像直方图的绘制。
- 掌握图像的点运算、代数运算和几何运算。
- 了解图像插值。

了解数字图像的存储格式和特征是对图像进行处理的基础，本章在认识数字图像的前提下，介绍数字图像的读取和显示及不同格式之间的转换，最后介绍数字图像的点运算、代数运算和几何变换。

3.1 数字图像的基本概念

3.1.1 数字图像

1. 认识数字图像

数字图像是由模拟图像数字化得到的以像素为基本元素的，可以用数字计算机或数字电路存储和处理的图像。数字图像是把图像按行与列分割成 $m \times n$ 个网格，由一些尺寸很小的矩形小块组成，如图 3.1 所示。矩形小块就是像素，$m \times n$ 就是图像的分辨率。当把图像放大时，会出现马赛克的效果，其实就是一个个矩形的像素。

对于彩色数字图片，通常表示成一个 $H \times W \times C$ 的三维矩阵。其中，H 表示图片的高，W 表示图片的宽，C 表示图片的通道数。$H \times W$ 就是图片的分辨率，也就是像素点的个数。对于每个像素点，都会表示一个颜色，用一个 C 维的向量描述，即 C 个通道。

2. 像素

像素(Pixel)是数字图像中最小的基本元素，是离散化的。每个像素具有整数行(高)和列(宽)位置坐标，同时每个像素都具有整数灰度值或颜色值。像素并不像“g”和“cm”是绝对的度量单位，而是可大可小的。像素本身的大小对应实际物体空间大小，但像素对应的最小尺度是受到成像设备本身的分辨能力限制的，如某个 MR 扫描仪生成的图像的像素大小是 2×2mm。

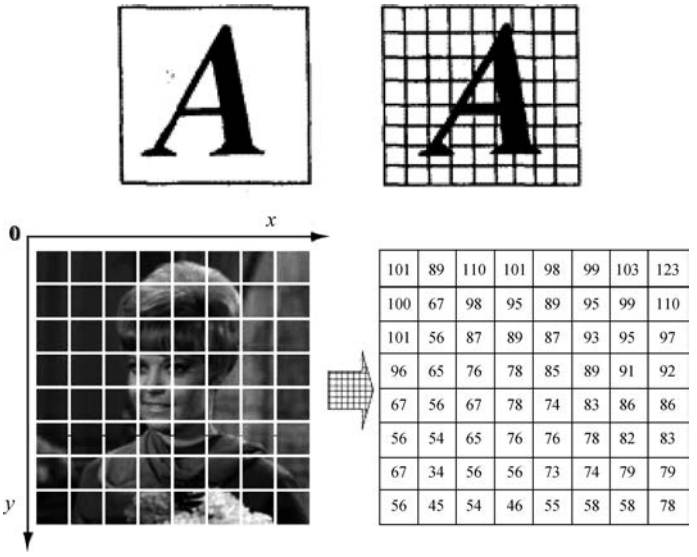


图 3.1　像素点阵表示的图像

3. 分辨率

图像分辨率表示图像垂直和水平方向的像素点的数量。例如，一张分辨率为 640×480 的图片，有 640 行、480 列像素点，即 640×480＝307200 个像素点，就是常说的 30 万像素的图片。而一张分辨率为 1600×1200 的图片，它的像素就是 200 万。图 3.2 给出不同分辨率的图像。显然，一幅图像的分辨率越大，它所能够表现的细节就越详细。

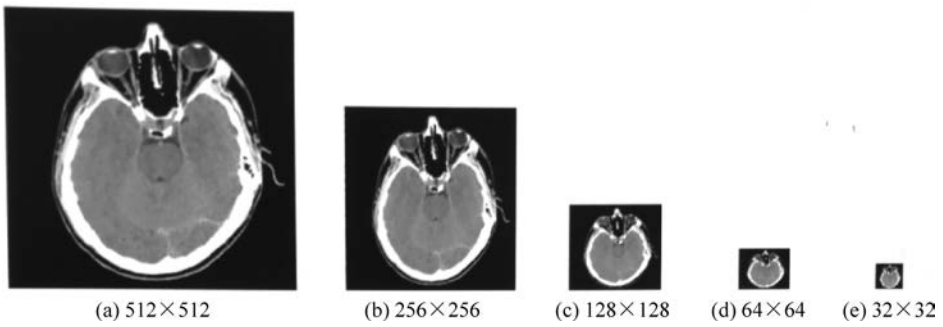


图 3.2　不同分辨率的脑 CT 截面图像

图像分辨率与显示器分辨率。显示器分辨率用于确定显示图像的区域大小，而图像分辨率用于确定组成一幅图像的像素数目。例如，在显示器分辨率为 1024×768 的显示屏上，一幅图像分辨率为 320×240 的图像约占显示屏的 1/12，而一幅图像分辨率为 2400×3000 的图像在这个显示屏上是不能完全显示的。对于具有相同图像分辨率的图像，屏幕分辨率越低(如 800×600)，则图像看起来越大，但屏幕上显示的细节少；屏幕分辨率越高(如 1024×768)，则图像看起来就越小。另外，在显示一幅图像时，可能会出现图像的高宽比与显示屏上显示图像高宽比不一致。这是由于显示设备定义的高宽比与图像的高宽比不一致。

数字图像质量由两个指标来衡量，即图像的空间分辨率和灰度分辨率。

1）空间分辨率

图像的空间分辨率是由单位面积内的像素数所决定，常以像素/英寸来表示，单位为dpi(pixels per inch)。例如，250dpi表示的就是该图像每英寸含有250个点或像素。在数字图像中，图像空间分辨率的大小直接影响到图像的质量。对于同样尺寸的一幅图，如果图像分辨率越高，单位长度包含的像素数目越多，像素点越小，图像越清晰、逼真。例如，72dpi分辨率的1×1英寸图像包含5184像素，而300dpi分辨率的1×1英寸图像包含90000像素。

空间分辨率越高，图像细节越清晰，但产生的文件尺寸大，同时处理的时间也就越长，对设备的要求也就越高。所以在采集图像时要根据需要选择分辨率。另外，图像的尺寸、图像的空间分辨率和图像文件的大小三者之间有着密切的联系。图像的尺寸越大，图像的空间分辨率越高，图像文件也就越大，所以调整图像的大小和空间分辨率，即可改变图像文件的大小。

2）灰度分辨率

图像的灰度分辨率又称色阶，指图像中灰度级的数目。图像的灰度级数量用2的整数次幂表示，如8bit有256个灰度级。

图像的灰度分辨率由图像的灰度级别决定，图像的灰度级越高，其灰度分辨率就越高；反之就越小。一幅图像的灰度分辨率越高，它能表现的细节就越细。图3.3给出不同灰度分辨率的图像。

(a) 灰度分辨率256　(b) 灰度分辨率128　(c) 灰度分辨率64

图3.3　不同灰度分辨率的人物图像

4. 图像的表示

数字图像是二维图像用有限数字数值像素的表示。那么，一幅数字图像就转化成矩阵。数字图像矩阵 $\boldsymbol{f}$ 的表示为

$$\boldsymbol{f}(y,x)=\begin{bmatrix} f(0,0) & \cdots & f(0,N-1) \\ \vdots & & \vdots \\ f(M-1,0) & \cdots & f(M-1,N-1) \end{bmatrix}$$

也可以用传统矩阵表示数字图像和像素，即

$$\boldsymbol{A}=\begin{bmatrix} a_{0,0} & \cdots & a_{0,N-1} \\ \vdots & & \vdots \\ a_{M-1,0} & \cdots & a_{M-1,N-1} \end{bmatrix}$$

M、N 为行列数，为正整数，像素的灰度级为2的 k 次幂，k 为整数，动态取值范围为[0，

255]，图像存储所需的比特数为 $b=M\times N\times k$。在矩阵 $\boldsymbol{f}(y,x)$ 的表示中，元素下标一般是行下标在前，列下标在后，因此这里先是纵坐标 y（对应行），然后才是横坐标 x（对应列）。

3.1.2　数字图像基本操作

在图像处理中，图像的读取、显示、保存是最基本的操作。使用 Python 进行图像处理时有多个库可以使用，如 OpenCV、PIL、Matplotlib. pyplot、skimage。OpenCV 读进来的是 NumPy 数组，PIL 有自己的数据结构，但可以转换成 NumPy 数组。OpenCV、PIL、Matplotlib、skimage 读取的图像数据类型均是 unit8，取值范围为 0～255。PIL、Matplotlib 和 skimage 读入的顺序是 RGB，而 OpenCV 读入顺序是 BGR。

1. 使用 OpenCV 库实现

1）读取图像

使用 OpenCV 中的 imread()函数读取图像，该函数支持各种静态图像文件格式，如 bmp、jpeg、jpg、png、TIFF 等。语法格式为：

```
retval = cv2.imread(filename[, flags])
```

（1）retval 为函数返回值，返回读取到的图像。若未读取到图像，则返回 None。

（2）filename 读取图像的路径。图像应该在工作目录下；否则给出图像完整路径。

（3）flags 指定读取图像文件的类型。常用参数设置如下。

① cv2.IMREAD_COLOR：加载彩色图像，将图像转换为三通道 BGR 彩色图像。图像的任何透明度都将被忽略（默认）。

② cv2.IMREAD_GRAYSCALE：以灰度模式加载图像。

③ cv2.IMREAD_UNCHANGED：保持原格式不变。

也可以使用 1、0、−1 代替表示上述 3 种图像读取方式。

例如：

```
import cv2
img = cv2.imread('lena.jpg',0)      #以灰度图像读取工作目录下的 lena 图片
```

【注意】 该函数通过内容而不是文件扩展名来决定图像类型；图像数据以 B、G、R 顺序存储。

2）显示图片

OpenCV 中提供了多个显示图像函数。

（1）imshow()函数。

imshow()函数用来显示图像。语法格式为：

```
cv2.imshow(winname, mat)
```

① winname：显示窗口的名字。

② mat：要显示的图像。

直接使用 imshow()函数显示图像，同时完成创建指定名称的窗口和在窗口显示图像两个操作。

(2) namedWindow()函数。

namedWindow()函数用来创建指定名称的窗口。语法格式为：

```
cv2.namedWindow(winname[, flags])
```

① winname：窗口的名称。

② flags：显示窗口的标志，有以下两个。

- cv2.WINDOW_AUTOSIZE：根据显示图像自动调整窗口，不能手动更改窗口大小(默认)。
- cv2.WINDOW_NORMAL：可以调整窗口大小。

例如：

```
cv2.namedWindow('showing', cv2.WINDOW_NORMAL)
```

(3) waitKey()函数。

waitKey()函数实现键盘绑定功能。用来等待按键，当用户按下任意键后，该语句会被执行，并获取返回值。语句格式为：

```
retval = cv2.waitKey([delay])
```

① retval：函数返回值。如果有按键被按下，则返回该按键的 ASCII 码；否则返回－1。

② delay：表示等待键盘触发的时间，单位为 ms。0 是指“永远”的特殊值。

(4) destroyAllWindows 函数。

cv2.destroyAllWindows 函数用来释放所有窗口。语法格式为：

```
cv2.destroyAllWindows()
```

【例 3.1】 在一个窗口显示读取的图像，并通过按键关闭所有打开的窗口。

```
import cv2
img = cv2.imread('E:/pyproject/lena.jpg',1)                    #读取彩色图像
cv2.namedWindow('showing', cv2.WINDOW_NORMAL)                  #创建一个名称为 showing 的窗口
cv2.imshow("showing",img)                                      #显示名称为"showing"窗口
cv2.waitKey()
cv2.destroyAllWindows()
```

图 3.4　例 3.1 运行结果

运行程序后，名为“showing”窗口显示图像，当按下键盘上的按键时，窗口“showing”会被释放；否则程序没有任何反应。

运行结果如图 3.4 所示。

3) 保存图像

OpenCV 中使用 imwrite()函数保存图像。语法格式为：

```
cv2.imwrite(filename,img[, params])
```

① filename：要保存图像文件的完整路径名。

② img：被保存图像的名称(可以理解为指向图片的指针)。

③ params：对于 JPEG，其表示的是图像的质量，用 0～100 的整数表示，默认为 95；对

于 png，参数 params 表示的是压缩级别，默认为 3。

【例 3.2】 将读取的图像保存到指定位置。

```
import cv2
img = cv2.imread('lena.jpg',1)
cv2.imwrite('E:/pyproject/1.jpg',img)  #将图片保存到指定位置,并进行命名
```

4）显示图片信息

打开图像文件后，可以通过一些属性来查看图片信息。

① shape：图像的行数和列数。

② size：返回图像的像素数目。

③ dtype：返回图像的数据类型。

【例 3.3】 读取图像，查看图片信息。

```
import cv2
img = cv2.imread('lena.jpg',1)
print (img.shape)          # shape[0] = 图像高,shape[1] = 图像宽,shape[2] = 图像通道数量
print(img.size)            #img.size 返回图像的像素数目
print(img.dtype)           # img.dtype 返回图像的数据类型, uint8 是 0～255 的整数
```

运行结果为：

```
(200, 200, 3)
120000
uint8
```

2. 使用 Pillow 库实现

通过 Pillow 库也可以实现图像的打开、保存和显示。

Image 类是 PIL 中的核心类。PIL 的 Image 类中常用的方法如下。

(1) Open()函数。

使用 Open()函数实现打开一张图像。语法格式如下：

```
dst = Image.open(filename[,mode])
```

① 该函数返回一个 Image 对象。

② filename：打开图像的完整路径。

③ mode：打开图像的模式，如“r”为只读模式；“r+”为可读可写模式；“w+”为可读可写，同时打开一个新的文件。

(2) Save()函数。

Save()函数实现保存指定格式的图像。语法格式为：

```
Save(filename,format)
```

① Filename：保存图像文件的完整路径。

② Format：保存图像的格式，如“jpg”“bmp”等。

例如：

```
im.save("1.png",'jpg')   #将 PNG 类型图片保存成 JPG 类型
```

(3) show()函数。

使用 show()函数显示图片。将图像保存到临时文件，并调用实用程序来显示图像。

(4) 查看图片信息。

Open()函数返回 Image 对象,该对象有 size、format、mode 等属性。size 表示图像的宽度和高度; format 表示图像的格式,如 JPEG、PNG 等; mode 表示图像的模式,定义的像素类型还有图像深度等,常见的有 RGB、HSV 等。"L"表示灰度图像,"RGB"表示彩色图像;"CMYK"表示预先压缩的图像。

【例 3.4】 读取一幅图像并显示图片及信息:

```
from PIL import Image
img = Image.open('lena.jpg')
img.show()
print(img.size)              #图片的尺寸
print(img.format)            #图片的格式
print(img.mode)              #图片的模式
```

程序运行如图 3.5 所示。

图 3.5 例 3.4 运行结果

运行结果:

```
(200, 200)
JPEG
RGB
```

3. 使用 Matplotlib 库实现

通过 Matplotlib.pyplot 实现图像的读取、显示和保存,见表 3.1。

表 3.1 Matplotlib 库实现图像基本操作的函数

函 数	作 用
plt.imread(filename)	读取图像
plt.imshow(image)	对图像处理,并显示格式
plt.show(image)	将 plt.Imshow()处理后的图像显示出来
plt.savefig()	保存图像

【例 3.5】 读取一幅图像并以灰度图像显示:

```
import Matplotlib.pyplot as plt
img = plt.imread('lena.jpg',plt.cm.gray)          #以灰度图像读取
plt.figure('image')                               #图像窗口名称
```

```
plt.imshow(img,cmap = 'gray')                    #灰度图像模式
plt.axis('on')                                   #关掉坐标轴为 off
plt.title('image')                               #图像标题
plt.show()
```

运行结果如图 3.6 所示。

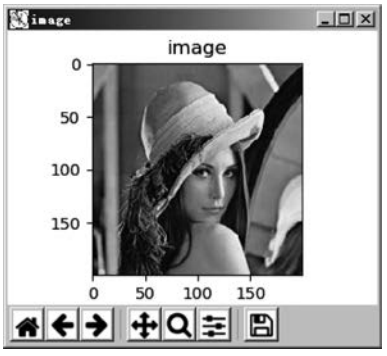


图 3.6　例 3.5 运行结果

4. 使用 Skimage 库实现

Skimage 提供了 io 模块，这个模块是用来实现图像输入输出操作的，见表 3.2。

表 3.2　Skimage 提供的图像输入输出函数

函　　数	作　　用	参　　数
imread(filename)	读取图片	filename 表示需要读取的文件路径
imshow(arr)	显示图片	表示需要显示的 arr 数组
imsave(filename,arr)	保存图片	filename 表示保存的路径和名称，arr 表示需要保存的数组变量

【例 3.6】 读取单张灰度图片：

```
from skimage import io
img = io.imread('lena.jpg',as_grey = True)       #以灰度图像打开图片
io.imshow(img)
io.show()
```

运行结果如图 3.7 所示。

图 3.7　例 3.6 的运行结果

为了方便练习，也提供一个 data 模块，里面嵌套了一些示例图片，可以直接使用。如果不想从外部读取图片，就可以直接使用这些示例图片，见表 3.3。

表 3.3 Skimage 自带的图片

函 数 名	图 片	函 数 名	图 片
astronaut	宇航员图片	coffee	一杯咖啡图片
lena	lena 美女图片	camera	拿相机的人图片
coins	硬币图片	moon	月亮图片
checkerboard	棋盘图片	horse	马图片
page	书页图片	chelsea	小猫图片
hubble_deep_field	星空图片	text	文字图片
clock	时钟图片	immunohistochemistry	结肠图片

保存图片的同时也起到了转换格式的作用。如果读取时图片格式为 JPG，保存为 PNG 格式，则将图片从 JPG 格式转换为 PNG 格式并保存。例如：

```
from skimage import io,data
img = data.chelsea()                    #读取自带的小猫图片
io.imsave('d:/cat.jpg',img)             #将图片保存到指定位置,格式为 jpg 图片
```

3.2 数字图像的类型与存储格式

3.2.1 数字图像类型

数字图像一般采用两种方式存储静态图像，即位图(Bitmap)存储模式和向量(Vector)存储模式。

向量图只存储图像内容的轮廓部分，而不是图像数据的每一点。这种方法的本质是用数学公式描述一幅图像，准确地说是几何学。图像中每个形状都是一个完整的公式，称为一个对象。公式化表示图像使得向量图具有两个优点：文件数据量小；图像质量与分辨率无关。无论将图像放大还是缩小，图像总是以显示设备允许的最大清晰度显示。向量图色彩不够丰富，绘制出来的图像不够逼真。

位图也称为栅格图像，是通过许多像素点表示一幅图像，每个像素具有颜色属性和位置属性。位图分为以下 4 种，即二值图像、灰度图像、真彩色图像和索引图像。

1. 二值图像

二值图像只有黑、白两种颜色，也叫黑白图像。二值图像中每个像素的取值仅有 0、1 两个值，“0”代表黑色，“1”代表白色，没有中间的过渡。图 3.8 所示为二值图像字母 A 在计算机内的存储形式。所以，在计算机中二值图像的每个像素值仅用一位二进制数表示。

二值图像通常用于文字、工程线条图的扫描识别和掩模图像的存储表示，如医学心电图中的线条图形就是典型的二值图像。

在 Python 中，最小的数据类型是无符号的 8 位数。因此，在 Python 中没有二值图像这种数据类型，二值图像通常是通过处理后得到的，然后使用 0 表示黑色，使用 255 表示白色。

2. 灰度图像

灰度图像也称为灰阶图像，图像中每个像素取值为[0,255]，0 表示纯黑，255 表示纯白，

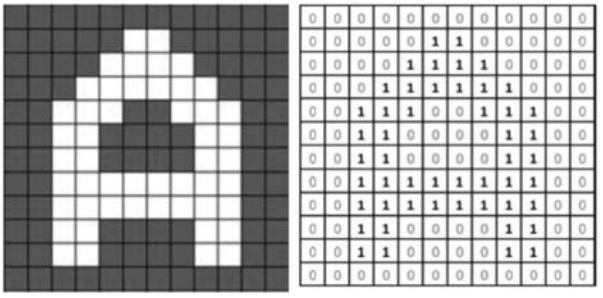


图 3.8　二值图像字母 A 在计算机内的存储形式

如图 3.9 所示。每个像素用 8 位二进制数表示，共有 $2^8=256$ 种灰度级。通常所说的黑白照片，就包含了黑白之间的所有灰度色调。在图像处理中，灰度图像通常存储为二维数组。

3. RGB 真彩色图像

$$I=\begin{pmatrix}0 & 150 & 200\\ 120 & 50 & 180\\ 250 & 220 & 100\end{pmatrix}$$

图 3.9　灰度图像的表示

RGB 图像分别用红(R)、绿(G)、蓝(B)三原色的组合来表示每个像素的颜色。每一个像素的颜色由 R、G、B 这 3 个分量来表示，直接存放在图像矩阵中，用 M、N 分别表示图像的行列数，3 个 $M\times N$ 的二维矩阵分别表示各个像素的 R、G、B 这 3 个颜色分量，如图 3.10 所示。RGB 图像为 24 位图像，R、G、B 分量分别占用 8 位，可以包含 2^{24} 种不同的颜色。通常用一个三维数组来表示一幅 RGB 彩色图像，表示为[$M\times N\times 3$]。

$$R=\begin{pmatrix}255 & 240 & 240\\ 255 & 0 & 80\\ 255 & 0 & 0\end{pmatrix}\quad G=\begin{pmatrix}0 & 160 & 80\\ 255 & 160 & 160\\ 0 & 255 & 0\end{pmatrix}\quad B=\begin{pmatrix}0 & 80 & 160\\ 0 & 0 & 240\\ 255 & 255 & 255\end{pmatrix}$$

图 3.10　RGB 真彩色图像的表示

4. 索引图像

计算机在还未能实现 24 位真彩色图像出现之前，就创造了索引颜色。索引颜色也称为映射颜色，在这种模式下，颜色都是预先定义的。索引图像把像素直接作为索引颜色的序号，根据索引颜色的序号就可以找到该像素的实际颜色。当把索引图像读入计算机时，索引颜色将被存储到调色板中。调色板是包含不同颜色的颜色表，每种颜色以红、绿、蓝 3 种颜色的组合来表示。调色板的单元个数是与图像的颜色数一致的。256 色图像有 256 个索引颜色，相应的调色板就有 256 个单元。

索引图像包括调色板和图像数据两个部分。调色板是把颜色进行排列、编号，图像数据对应为该点像素的颜色序号而非颜色本身。调色板为 $m\times 3$ 矩阵，每一行代表一种颜色，各元素的值介于[0,1]，乘以 255 来表示实际值。

3.2.2　图像类型的转换

在图像处理中，根据需要将图像类型进行转换，最常用的就是将 RGB 图像转换为灰度图像或二值图像。

1. RGB 图像转换为灰度图像

(1) 使用 PIL 的 convert()函数。

PIL 中使用 convert()函数实现转换。语法格式为：

```
convert(mode,matrix,dither,palette,colors)
```

其中，mode 表示转换模式，一般为 RGB(真彩色)、L(灰度图像)、CMYK(压缩图)例如：

```
from PIL import Image
img = Image.open('lena.jpg').convert('L')
```

(2) 使用 skimage 的 imread()函数。

imread()函数在读取图像时可以直接以灰度图像读取。例如：

```
from skimage import io
img = io.imread('lena.jpg', as_grey = True)
```

(3) 使用 OpenCV 中的 imread()函数。

imread()函数将图像文件以灰度图像读取。例如：

```
import cv2
img = cv2.imread('lena.jpg', 0)
```

2. RGB 图像转化为二值图像

彩色图像二值化是图像处理中非常常用的方法，处理方法也多种多样。二值化后做进一步处理。图像二值化要先将彩色图像转换为灰度图像，再将灰度图像转换为二值图像。彩色图像转换为灰度图像，上面已经介绍了，下面来看灰度图像转换为二值图像。在 OpenCV 中，图像的二值化提供了阈值 threshold 函数。语法格式为：

```
cv2.threshold(src, x, y, Methods)
```

① src：指原始图像，该原始图像为灰度图。

② x：指用来对像素值进行分类的阈值。

③ y：指当像素值高于(有时小于)阈值时应该被赋予的新的像素值。

④ Methods：指不同的阈值方法，这些方法有 cv2. THRESH_BINARY、cv2. THRESH_BINARY_INV、cv2. THRESH_TRUNC、cv2. THRESH_TOZERO、cv2. THRESH_TOZERO_INV。第 5 章还会详细介绍 threshold 函数在图像分割中的应用。

【例 3.7】 读取一幅彩色图像，将其二值化：

```
import cv2
img = cv2.imread('lena.jpg',0)
ret, thresh = cv2.threshold(img, 12, 255,cv2.THRESH_BINARY |cv2.THRESH_TRIANGLE)
cv2.imshow('grey', img)
cv2.imshow('binary', thresh)
cv2.waitkey()
cv2.destroyAllWindows()
```

运行结果如图 3.11 所示。

图 3.11　彩色图片的二值化

3.2.3　图像数据类型及转换

在 Skimage 库中，一张图片就是一个简单的 NumPy 数组，数组的数据类型有很多种，相互之间也可以转换。这些数据类型及取值范围如表 3.4 所示。

表 3.4　图像的数据类型及取值范围

数 据 类 型	取 值 范 围
uint8	0 或 255
uint16	0 或 65535
uint32	0 或 232
float	−1～1 或 0～1
int8	−128～127
int16	−32768～32767
int32	$-2^{31}\sim 2^{31}-1$

一张彩色图像的像素值范围是[0,255]，因此默认类型是 unit8，可用以下代码查看数据类型：

```
from skimage import io,data
img = data.chelsea()          #打开 skimage 自带图片"小猫图片"
print(img.dtype.name)
```

Skimage 提供的图像数据类型转换如表 3.5 所示。

表 3.5　Skimage 提供的图像数据类型转换

名　　称	作　　用
img_as_float	转换为 64bit
img_as_ubyte	转换为 8bit uint
img_as_uint	转换为 16bit uint
img_as_int	转换为 16bit int

一张彩色图像转换为灰度图像后，它的类型就由 unit8 变成了 float，float 类型的取值范围是[−1,1]或[0,1]。

【例 3.8】 图像数据类型 unit8 转换为 float：

```
from skimage import data,img_as_float
img = data.chelsea()
print(img.dtype.name)
dst = img_as_float(img)
```

```
print(dst.dtype.name)
```

运行结果：

```
uint8
float64.
```

3.2.4 图像像素操作

1. 使用 Skimage 库实现

在 Skimage 库中图片读入程序后，是以 NumPy 数组存在的。对数组元素的访问，实际上就是对图片像素点的访问。

灰度图片的访问方式：

```
gray[i,j]
```

彩色图片的访问方式：

```
img[i,j,c]
```

其中：i 表示图片的行数；j 表示图片的列数；c 表示图片的通道数(RGB 三通道分别对应 0、1、2)。坐标是从左上角开始。

【例 3.9】 输出小猫图片的 G 通道中的第 20 行 30 列的像素值：

```
from skimage import io,data
img = data.chelsea()
pixel = img[20,30,1]
print(pixel)
```

输出结果：

```
129
```

【例 3.10】 显示红色单通道图片：

```
from skimage import io,data
img = data.chelsea()
R = img[:,:,0]
io.imshow(R)
```

运行结果如图 3.12 所示。

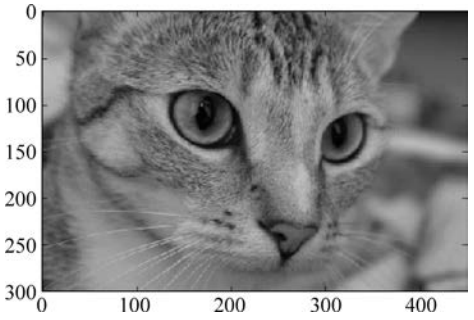


图 3.12 例 3.10 运行结果

除了对像素进行读取外，也可以修改像素值。

【例 3.11】　对小猫图片随机添加椒盐噪声：

```
from skimage import io,data
import numpy as np
img = data.chelsea()
#随机生成 5000 个椒盐
rows,cols,dims = img.shape
for i in range(5000):
    x = np.random.randint(0,rows)
    y = np.random.randint(0,cols)
    img[x,y,:] = 255
io.imshow(img)
```

运行结果如图 3.13 所示。

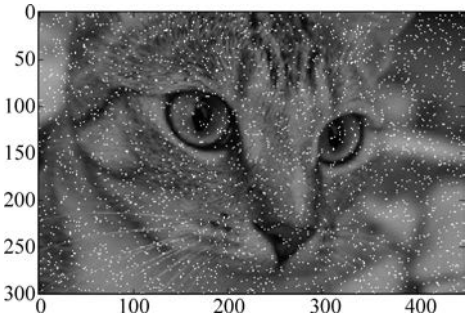


图 3.13　例 3.11 运行结果

这里用到了 NumPy 包里的 random 来生成随机数。randint(0,cols)表示随机生成一个整数，范围在 0～cols。用“img[x,y,:]＝255”语句来对像素值进行修改，将原来的三通道像素值变为 255。

通过对图像区域的访问，可以实现对图片的裁剪。

【例 3.12】　对小猫图片进行裁剪：

```
from skimage import io,data
img = data.chelsea()
roi = img[80:180,100:200,:]
io.imshow(roi)
```

运行结果如图 3.14 所示。

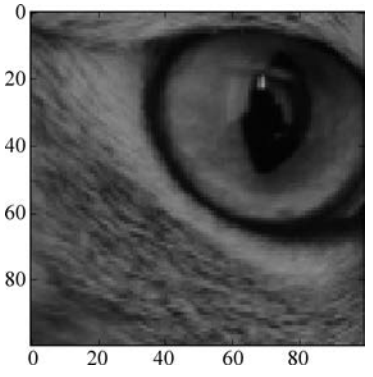


图 3.14　例 3.12 的运行结果

【例 3.13】　将 lena 图片进行二值化，像素值大于 128 的变为 1，否则变为 0。

```
from skimage import io,data,color
img = data.lena()
img_gray = color.rgb2gray(img)
rows,cols = img_gray.shape
for i in range(rows):
    for j in range(cols):
        if (img_gray[i,j]<=0.5):
            img_gray[i,j] = 0
        else:
            img_gray[i,j] = 1
io.imshow(img_gray)
```

运行结果如图 3.15 所示。

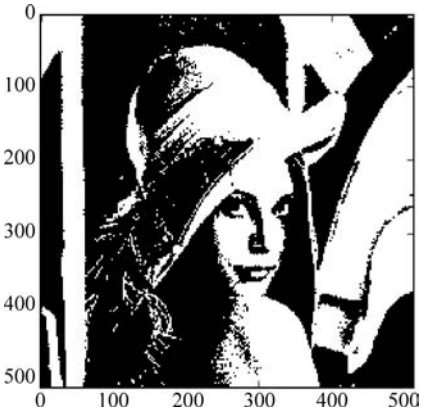


图 3.15　例 3.13 的运行结果

2. 使用 OpenCV 库实现

(1) 通过索引访问像素。

在 OpenCV 中,通过索引对图像某一像素点进行操作。

【例 3.14】 读取图像的像素并修改:

```
import cv2
img = cv2.imread('lena1.jpg',1)
pix = img[100,100,0]                    #获取(100,200)处 B 通道的像素值
pix1 = img[100,100]                     #获取(100,200)处的像素值
print(pix1,pix)
img[100:200,100:200] = 0                #将这一区域像素设置为 0
cv2.imshow('result',img)                #显示图像
cv2.waitKey(0)                          #保持图像
```

程序运行如图 3.16 所示。

图 3.16　例 3.14 运行结果

运行结果:

```
[26  55 106] 26
```

(2) 通过 item()和 itemset()函数访问像素。

访问和修改像素值还可以使用 numpy.array 提供的 item()和 itemset()函数。这两个函数都是经过优化处理的，在对像素点进行操作时，使用这两个函数处理速度要快得多。

item()函数能够更加高效地访问图像的像素点，语法格式为：

```
item(行,列)
```

itemset()函数可以用来修改像素值，语法格式：

```
itemset(索引值,新值)
```

【例 3.15】 读取一幅灰度图像，访问并修改该图像像素点的值：

```
import cv2
img = cv2.imread('lena.jpg',0)
print('读取像素点(2,3)的值: ',img.item(2,3))
img.itemset((2,3),255)
print('修改后像素点(2,3)的值: ',img.item(2,3))
```

输出结果：

```
读取像素点(2,3)的值: 167
修改后像素点(2,3)的值: 255
```

3.2.5　数字图像的基本文件格式

计算机中图像数据是以图像文件的形式存储。数字图像有多种存储格式，每种格式由不同的开发商支持。随着信息技术的发展和图像应用领域的不断拓展，还会出现新的图像格式。

每种图像文件均有一个文件头，在文件头之后才是图像数据。文件头的内容由制作该图像文件的公司决定，一般包括文件类型、文件制作者、制作时间、版本号、文件大小等内容。各种图像文件的制作还涉及图像文件的压缩方式和存储效率等。目前常用的图像格式有 BMP、JPG、TIFF、GIF 等，此外医学图像专用的格式还有 DICOM、IMG 等。

1. BMP 图像

BMP 文件(Bitmap)也称为位图文件，是 Microsoft 公司开发的最普通的栅格图像格式。这种图像文件格式中，位图的每个数据位置对应地确定了图像中像素的空间位置，位图数据值和相应像素的亮度值一一对应，存储开销相对较大。BMP 图像文件格式可以存储单色、16 色、256 色以及真彩色四种图像数据。

BMP 图像文件的结构分为四个部分，即文件头、位图信息头、颜色表和位图数据。

第一部分为位图文件头 BITMAPFILEHEADER，是一个结构体类型，该结构的长度是固定的，为 14 字节。

第二部分为位图信息头 BITMAPINFOHEADER，也是一个结构体类型的数据结构，该结构的长度也是固定的，为 40 字节。

第三部分为颜色表。颜色表是一个 RGBQUAD 结构的数组，数组的长度由 biClrUsed 指定。RGBQUAD 结构是一个结构体类型，占 4 字节。

第四部分是位图数据，即图像数据，其紧跟在位图文件头、位图信息头和颜色表之后，记录了图像的每一个像素值。对于真彩色图，位图数据就是实际的 R、G、B 值。

一般来说，BMP 文件的数据是从图像的左下角开始逐行扫描的，即从下到上、从左到右，将图像的像素值一一记录下来，因此图像坐标零点在图像左下角。

2. TIFF 图像

TIFF 是最复杂的一种位图文件格式。它是基于标记的文件格式，并广泛应用于对图像质量要求较高的图像存储与转换。由于其结构灵活和包容性大，已成为图像文件格式的一种标准，绝大多数图像系统都支持这种格式，并且是交换图像信息的最佳可选图像文件格式之一。

3. GIF 图像

GIF 格式图像文件是由 Compuserver 公司创建。存储色彩最高只能达到 256 种，仅支持 8 位图像文件。该格式文件是经过压缩的图像文件格式，所以大多用在网络传输上和 Internet 的 HTML 网页文档中，速度要比传输其他图像文件格式快得多。它的最大缺点是最多只能处理 256 种色彩，故不能用于存储真彩色图像文件。其可以将数张图存成一个文件，从而形成动画效果。

4. JPEG 图像格式

JPEG 图像格式是由国际标准化组织和国际电报电话咨询委员会两大标准化组织共同推出的。其特点是具有高效的压缩效率和标准化要求，由于 JPEG 的高压缩比和良好的图像质量，成为多媒体和网络中应用较广泛的图像格式。

JPEG 格式使用 24 位色彩深度使图像保持真彩。通过有选择地删除图像数据，从而节省存储空间和传输流量，但这些被删除的图像数据无法在解压缩时还原，因此 JPEG 压缩为有损压缩。在医学图像处理中，出于对安全性、合法性及成本等因素的考虑，对于图像的压缩需要十分谨慎。

5. DICOM 医学图像

DICOM(Digital Imaging and Communications in Medicine)，数字医学成像与通信标准，是美国放射学会(ACR)和美国电气制造商协会(NEMA)组织制定的专门用于医学图像的存储和传输的标准。制定目的是解决医学设备互联、统一图像格式和传输等问题。

自 1985 年 DICOM 标准第一版发布，发展到现在的 DICOM 3.0 版本，已被医疗设备生产商和医疗界广泛接受，成为医学影像信息学领域的国际通用标准。带有 DICOM 接口的医疗设备广泛应用于放射医疗，心血管成像以及放射诊疗诊断设备(X 射线、CT、核磁共振、超声等)，以及眼科和牙科等其他医学领域医疗设备。由于 DICOM 的开放性与互联性，它可以与其他医学应用系统(HIS、RIS 等)进行集成。

DICOM 文件的扩展名为“.dcm”，目前大多数的图像处理软件都不支持该文件，阅读该文件图像需要采用专用的软件，如 DICOM 图像浏览软件，实现打开 DICOM 图像文件、保存成常用图片格式等操作。DICOM 图像采用位图方式，逐点表示其位置上的灰度和颜色信息。DICOM 一般采用的是 RGB 三基色表示，即一个点由红、绿、蓝 3 个基色分量的值组成。DICOM 可以用 3 个矩阵分别表示三基色分量值，也可以用一个矩阵表示整个图像，即矩阵的每个点都是由 3 个值组成。

DICOM 文件格式提供了一种封装文件中数据集的方法，将信息对象定义(DICOM IOD)为一个服务对象对(SOP)实例，以数据集的形式封装在一个文件中。DICOM 标准文

件由 DICOM 文件头信息和 DICOM 数据集两部分组成，DICOM 文件结构如图 3.17 所示。每个文件包含一个单一的 SOP 实例，其中包含一帧或多帧图像。DICOM 文件数据集除了包含图像外，还包含许多与图像相关的信息，如患者姓名、性别、年龄、检查设备、传输语法等。

图 3.17　DICOM 文件结构

(1) DICOM 文件头。

DICOM 文件头信息位于文件的起始，用于描述该文件的版本信息、存储媒体、传输语法标识等信息。文件头的最开始是 128 字节的文件前导符；4 字节的 DICOM 前缀“D”“I”“C”“M”，标识该文件是 DICOM 文件；接下来是文件头元素。

(2) DICOM 数据集。

DICOM 的数据集是由一系列 DICOM 的数据元素组成，分为四类，即 Patient、Study、Series 和 Image。每个数据元素由唯一数据元素标记 tag 来表示。多个数据元素在数据集中以标记从小到大递增的顺序排列。每个数据元素由四部分组成，即标签、数据描述(VR)、数据长度和数据域，如图 3.18 所示。

标签	数据描述	数据长度	数据域

图 3.18　数据元素的组成

Python 利用 Pydicom 库对 DICOM 图像文件进行处理。Pydicom 库可以用来提取各种 DICOM 里的信息，如 PatientName 等，也可以把 DICOM 文件里的信息匿名化。

Pydicom 的 API 参考如下。

① data_element(个人，姓名)：返回与元素关键字 name 对应的 DataElement。

② dir(自身，\过滤器)：返回数据集中的 DataElement 关键字的字符顺序列表。

③ value(值)：返回 DICOM 标签值以模拟 dict。

【例 3.16】　读取一幅 DICOM 图像信息。

```
impor tpydicom
import Matplotlib.pyplot as plt
dcm = pydicom.read_file("E:\dicom\image\img02.dcm")          # 读取 dcm 文件
print(dcm.dir())                                             # 查看全部属性
print(dcm.PatientName)                                       # 查看病人信息
# 打印完整数据元素
data_element = dcm.data_element('PatientID')
print(data_element.tag,data_element.VR,data_element.value)
pix = dcm.pixel_array                                        # 像素值矩阵
pintr (pix.shape)
# 显示读取图像
```

```
plt.imshow(pix,"gray")                                    #以灰度图像显示
plt.show()
```

【注意】 读取 dcm 文件还可以写成“dcm＝pydicom.dcmread("E:\dicom\image\img02.dcm")”。运行结果可以结合第 2 章实例 2.4 的运行结果，虽然显示方法不同，但是目标结果图一样。

3.3 数字图像的灰度直方图

在数字图像处理中，灰度直方图是非常重要的，是最简单且最有用的工具，可以说，对图像的分析与观察，直到形成一个有效的处理方法，都离不开直方图。直方图在图像处理中有着十分广泛的应用。

3.3.1 图像灰度直方图概念

灰度直方图是关于灰度级分布的函数，描述的是图像中各个灰度级的像素个数或出现的频率，是对图像中全部像素灰度的统计。横坐标表示灰度级，纵坐标表示图像中各灰度级出现的个数或频率，这个关系图就是灰度直方图(Histogram)。

直方图是图像的一个重要特征，直观地反映了图像各个灰度级分布情况，是数字图像处理的基础。图 3.19 给出了一幅灰度分布均匀的 X 光平片影像的直方图。通过图像可以看到，横坐标越往左侧显示图像中越暗像素分布情况，越往右侧显示图像中越亮像素分布情况。

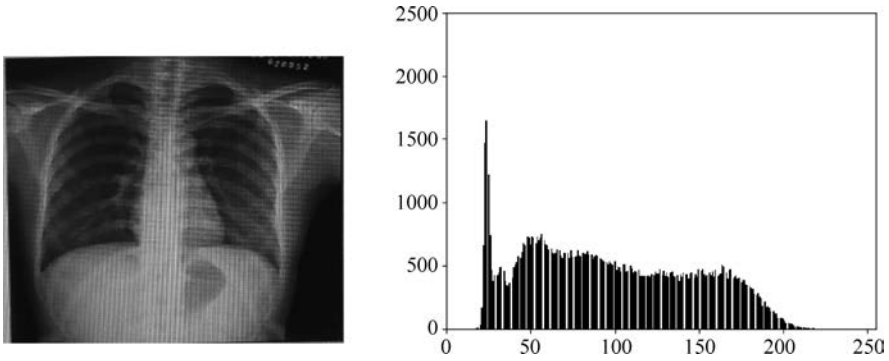


图 3.19 一幅灰度分布均匀的 X 光平片直方图

直方图的纵坐标通过像素出现的频率来设置。设一幅数字图像的像素总数为 n，灰度级为 k，具有第 i 灰度级的等级灰度 r_i 的像素共有 n_i 个。那么，灰度级为 r_i 的像素出现的频率为

$$p(r_i)=\frac{n_i}{n} \quad i=0,1,\cdots,k-1$$

$$\sum_{i=0}^{k-1} p(r_i)=1$$

直方图也称为归一化直方图。对于较暗图像，低灰度的背景区域较大，往往出现在靠近纵轴处高计数，而在其他灰度处幅度显示过低的情况，在绘制直方图时可采用归一化方法。

3.3.2　绘制直方图

对于直方图的计算，需要统计各个灰度级上像素的数目，然后根据此关系勾画出直方图。

1. 使用 hist()函数绘制直方图

通过调用 Matplotlib.pyplot 库中的 hist()函数直接绘制直方图。语法格式为：

```
plt.hist(x,bins)
```

hist()函数的参数非常多，这里只介绍最常用的前两个。

(1) x：指定要绘制直方图的数据。一个数组或一个序列，必须为一维。

(2) bins：指定 bin 的个数，即灰度级的分组情况。若为整数值，则为频数分布直方图柱子根数，若为数值序列，则该序列给出每根柱子的范围值，除最后一根柱子外，其他柱子的取值范围均为左闭右开，若数值序列的最大值小于原始数据的最大值，存在数据丢失。

例如：

```
import Matplotlib.pyplot as plt
x = [2500,3100,2750,4500,5100]                    #绘制直方图的数据
y = [1000,2000,3000,4000,5000,6000]               #直方图灰度级的分组
plt.hist(x,y)
plt.show()
```

运行结果如图 3.20 所示。

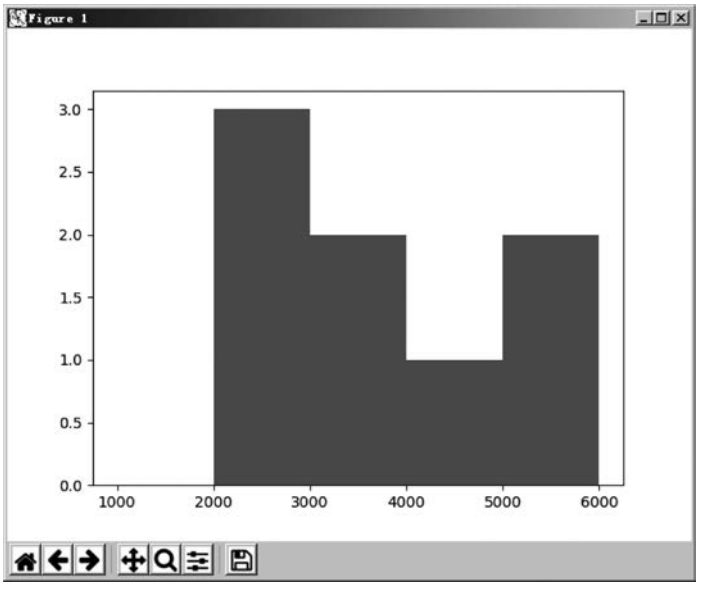


图 3.20　直方图绘制效果

说明：图像通常是二维的，需要使用 ravel()函数将图像处理为一维数据源以后，再作为参数使用。

【例 3.17】 绘制一幅灰度图像的直方图。

```
import Matplotlib.pyplot as plt
```

```
img = plt.imread('lena.jpg',plt.cm.gray)                    #以灰度图像读取
bins = 256                                                  #直方图灰度级分组 256 个,灰度图像
                                                            #的像素值为 0～255 共 256 个
plt.figure("绘制直方图")                                     #绘制图像窗口
n = img.flatten()                                           #将二维图像像素值转化为一维
#绘制直方图
plt.hist(n,bins,color = "black")
#添加 x 轴、y 轴标签
plt.xlabel("gray label")
plt.ylabel("number of pixels")
plt.show()                                                  #显示图形
```

运行结果如图 3.21 所示。

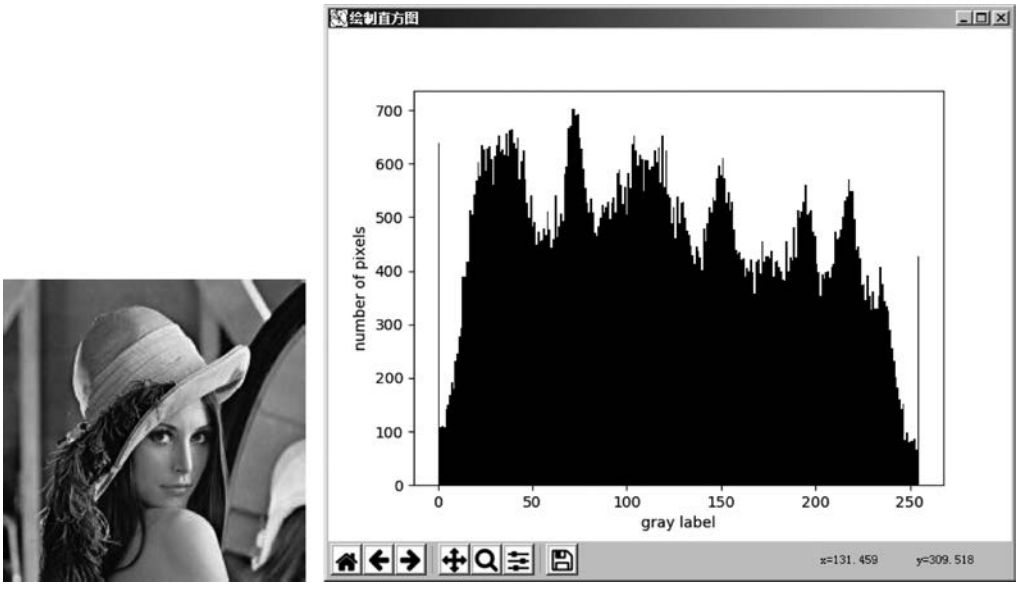


图 3.21　实例直方图绘制效果

图中未经归一化的灰度直方图的纵轴表示图像中所有像素取到某一特定灰度值的次数,横轴为 0～255 所有灰度值,覆盖了 unit8 存储格式的灰度图像中的所有可能取值。由于相邻的灰度值具有的含义是相似的,所以没有必要在每个灰度级上都进行统计。例如,将 0～255 总共 256 个灰度级平均划分为 32 个长度为 8 的灰度区间,此时纵轴分别统计每个灰度区间中的像素在图像中的出现次数,直方图绘制代码如下:

```
plt.hist(n,32,color = "black")      #共 32 个灰度区间,绘制直方图
```

运行结果如图 3.22 所示。

【例 3.18】 绘制彩色图像直方图。

直方图是灰度图像直方图,要绘制 RGB 图像的三通道直方图,将 R、G、B 通道的 3 个直方图进行叠加即可。

```
import cv2
import Matplotlib.pyplot as plt
img = cv2.imread('lena.jpg')                    #原图读取
#OpenCV 读取图像的通道顺序为 B、G、R
b = img[:,:,0].flatten()                        #蓝色通道的一维化
```

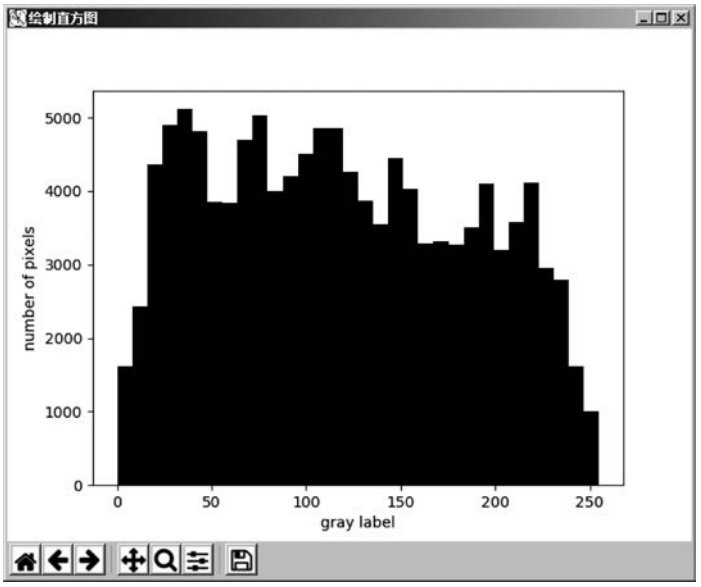


图 3.22　bins＝32 的直方图绘制

```
g = img[:,:,1].flatten()                    #绿色通道的一维化
r = img[:,:,2].flatten()                    #红色通道的一维化
plt.figure("image")
plt.hist(r, bins = 256, color = "red")
plt.hist(g, bins = 256, color = "green")
plt.hist(b, bins = 256, color = "blue")
plt.show()
```

运行结果如图 3.23 所示。

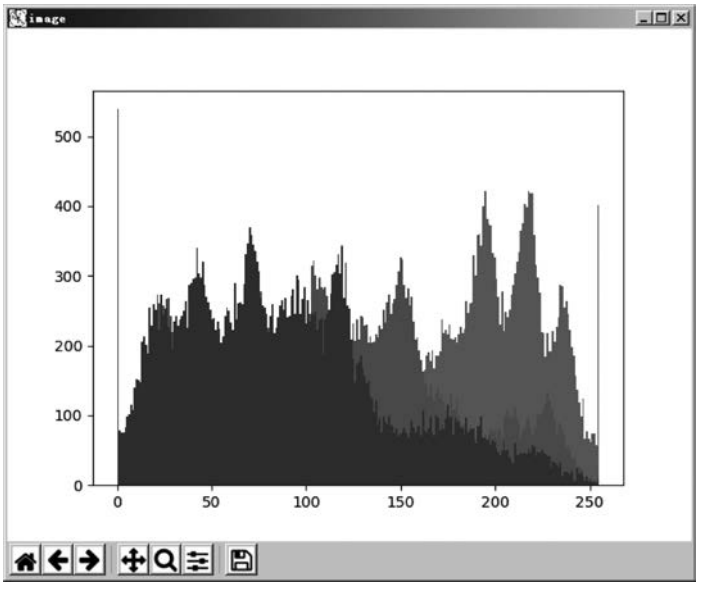


图 3.23　例 3.18 彩色图像直方图的绘制

2. 使用 plot()函数绘制直方图

利用 Matplotlib.pyplot 中的 Plot()函数绘制直方图，要和 cv2.calcHist()函数一起使用。OpenCV 提供了 cv2.calcHist()函数用来统计图像直方图各个灰度级的像素点个数。利用 Matplotlib.pyplot 中的 Plot()函数，将 cv2.calcHist()函数的统计结果绘制成直方图。

(1) cv2.calcHist()函数的使用。

cv2.calcHist()函数统计图像直方图信息，语法格式：

```
hist = cv2.calcHist(images,channels,mask,histSize,ranges,accumulate)
```

- hist：返回统计直方图的各个灰度级的像素个数，是一个一维数组。
- images：原始图像，要用“[]”括起来。
- channels：指定通道编号，要用“[]”括起来。灰度图像参数值为[0]；彩色图像可以是[0]、[1]、[2]，分别对应 B、G、R。
- mask：掩模图像。统计整个图像直方图时，该参数为 None。
- histSize：Bins 的值，要用“[]”括起来。
- ranges：像素值范围。
- accumulate：累计标识，默认值为 False。

例如：

```
img = cv2.imread("test.jpg")                              #以彩图读取
hist = cv2.calcHist([img],[0],None,[256],[0,255])         #统计图像 img 第 0 通道像素个数
```

(2) plot()函数的使用。

plot 是绘制一维曲线的基本函数，使用 plot()函数将 cv2.calcHist()函数的返回值绘制成直方图。语法格式：

```
plot(y) 或者 plot(x,y)
```

例如：

```
import Matplotlib.pyplot as plt
y = [0.3,0.4,2.5,3,4.5,4]                 #要绘制数的取值
plt.plot(y)
#以 y 的分量为纵坐标,以元素序号为横坐标,用直线依次连接数据点,绘制曲线
plt.show()
```

运行结果如图 3.24 所示。仅指定一个参数，x 轴默认为一个自然数序列。

利用 plot()函数实现实例 3.18，代码改为：

```
import cv2
import Matplotlib.pyplot as plt
img = cv2.imread('1.jpg')
histb = cv2.calcHist([img],[0],None,[256],[0,255])          #统计 b 通道直方图信息
histg = cv2.calcHist([img],[1],None,[256],[0,255])          #统计 g 通道直方图信息
histr = cv2.calcHist([img],[2],None,[256],[0,255])          #统计 r 通道直方图信息
plt.plot(histb,color = 'b')                                 #绘制 b 通道直方图
plt.plot(histg,color = 'g')                                 #绘制 g 通道直方图
plt.plot(histr,color = 'r')                                 #绘制 r 通道直方图
plt.show()
```

运行结果如图 3.25 所示。

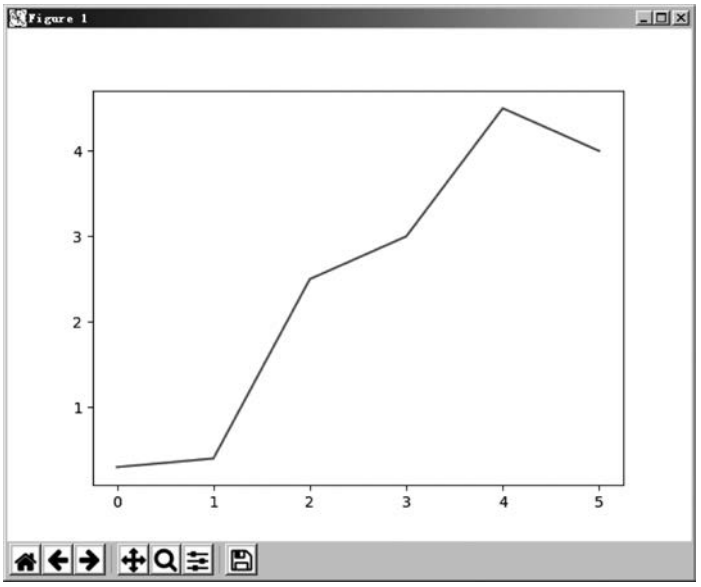


图 3.24　直方图绘制效果

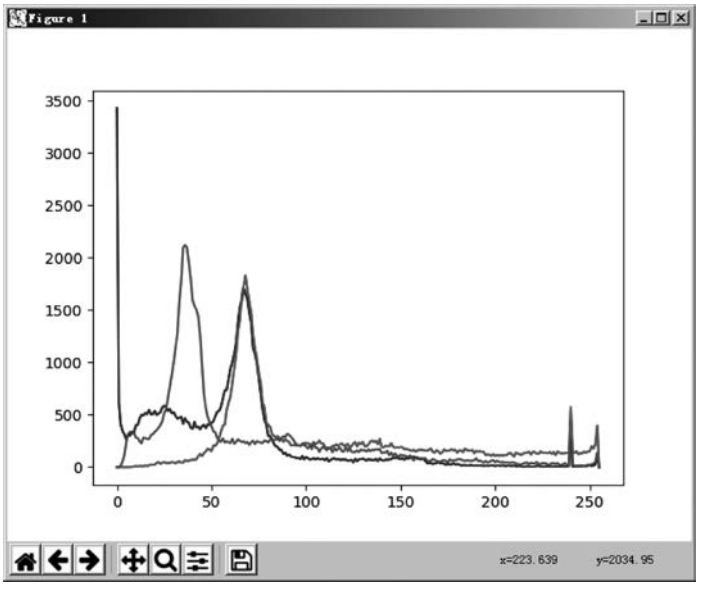


图 3.25　plot 绘制直方图

3.3.3　图像灰度直方图的性质

(1) 直方图只反映不同灰度像素值的分布信息，不包括图像像素的空间位置信息。

(2) 不同图像可能具有相同的直方图。

一幅图像对应唯一的灰度直方图；反之不成立。不同的图像可对应相同的直方图，如图 3.26 所示。

(3) 直方图反映图像的总体性质。

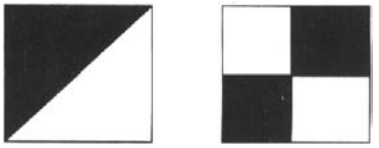


图 3.26　两幅图像内容不同但具有相同的直方图

一幅较好的图像应该明暗细节都有，在直方图上从左到右都有分布，同时直方图的两侧不会有像素溢出。而直方图的竖轴就表示相应部分所占画面的面积，峰值越高说明该明暗值的像素数量越多。

通过直方图可以观察出图像的整体特征，如图像的明暗程度、细节是否清晰、动态范围大小等。如图 3.27 所示，四幅图分别为较暗图像的直方图、较亮图像的直方图、对比度较弱图像的直方图和对比度较强图像的直方图。

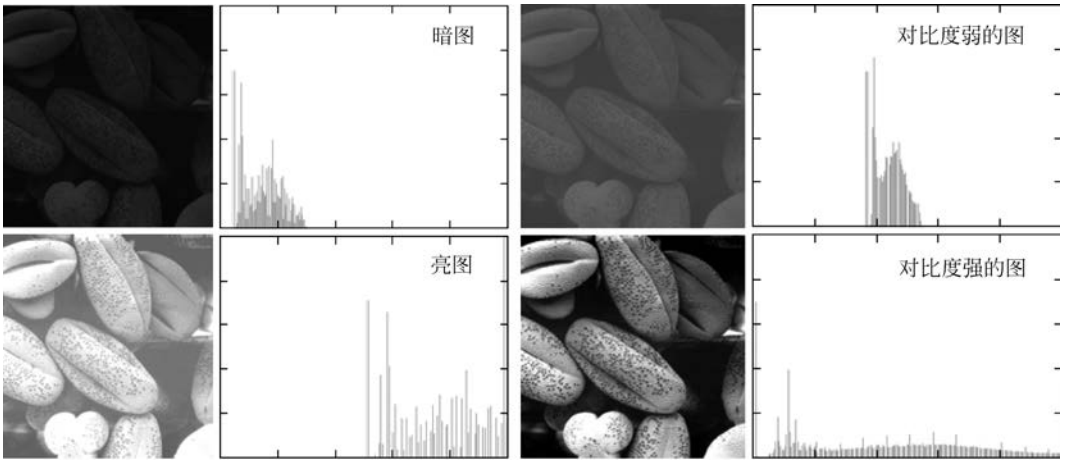


图 3.27　不同特征图像的直方图对比

图像的总体特征分别用直方图表示，如图 3.28 所示。

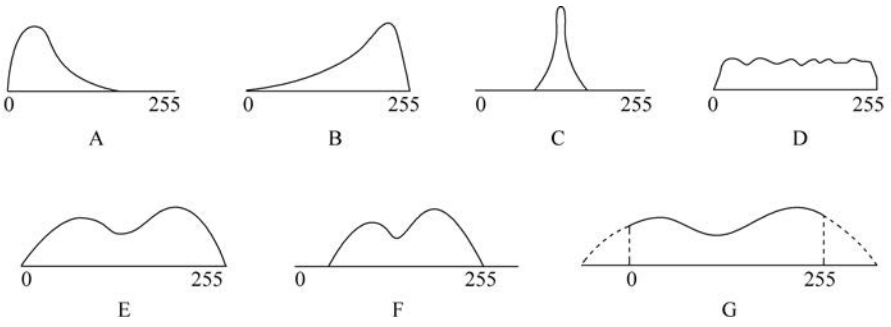


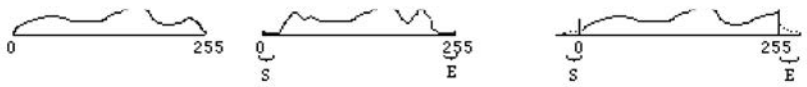
图 3.28　直方图反映出的图像特征

A—图像总体偏暗；B—图像总体偏亮；C—图像动态范围小，细节不够清楚；D—图像灰度分布均匀，清晰明快；E—图像动态范围适中；F—图像动态范围偏小；G—图像动态范围偏大

3.3.4　直方图的用途

(1) 直方图可用来判断一幅图像是否合理地利用了全部被允许的灰度级范围。

直方图给出了一个简单可见的指示，用来判断一幅图像是否合理地利用了全部被允许的灰度级范围。一般一幅图应该利用全部或几乎全部可能的灰度级；否则等于增加了量化间隔，如图 3.29 所示。



(a) 恰当量化　(b) 未能有效利用动态范围　(c) 超过了动态范围

图 3.29　直方图特征效果对比图

(2) 直方图可用来进行边界阈值选择。

图像的轮廓线提供了一个确立图像中简单物体边界的有效方法，使用轮廓线作为边界的技术称为阈值化。直方图因为提供了一幅图像的全部灰度信息，所以可以利用直方图来进行边界阈值选择。

在数字图像处理技术中，直方图的用途体现在以下几方面。

(1) 评价成像条件。

根据图像灰度直方图，分析图像在成像过程或数字化过程中是否合理地使用了灰度动态范围。例如，曝光不足或曝光过度都是没有合理地利用亮度范围，造成大部分的像素集中在较少的亮度范围内，从而影响了图像的清晰度。

(2) 进行图像增强处理。

根据图像的亮度直方图，设计一种亮度映射函数，实现处理后图像的像素尽可能充分地使用亮度动态范围，或将亮度映射到色彩空间，以不同的颜色强化图像的亮度变换。

(3) 进行图像分割。

根据图像的灰度直方图，将像素分割成不同的类别，实现不同景物的提取。同一景物的像素具有相近的灰度分布，不同景物间存在不同的灰度分布。如果将直方图拓展至亮度以外，表达一种参数的统计，则这种参数的直方图对于图像分割具有更一般性的应用价值。直方图对物体与背景有较强对比的景物的分割特别有用，可以确定图像二值化的阈值。

(4) 进行图像压缩。

利用灰度直方图的统计信息，设计一种编码方案，让具有最多像素的亮度以最少的字长表示，从而用最少的数据量表达整幅图像，如 Huffman 编码算法。

3.4　数字图像的色彩空间

3.4.1　常见的色彩空间

颜色是图像的重要属性之一，图像的色彩在图像处理中起着重要作用。基于不同领域和应用，对图像的颜色有不同的编码模型，称之为色彩空间。色彩空间是表示颜色的一种数学方法，用来指定和产生颜色，使颜色形象化。

常用的色彩空间主要有 GRAY 色彩空间、RGB 色彩空间、CMYK 色彩空间、HSV 色彩空间、HLS 色彩空间、YUV 色彩空间等。

1. GRAY 色彩空间

GRAY 灰度图像是指 8 位灰度图，具有 256 个灰度级，像素值的范围为[0，255]。当图像由 RGB 色彩空间转换为 GRAY 色彩空间时，转换公式为

$$I=\omega_R R+\omega_G G+\omega_B B$$

式中，ω_R、ω_G、ω_B 是 3 种颜色的权重。通常 3 种权重的取值为

$$\omega_R=0.299,\quad \omega_G=0.587,\quad \omega_B=0.114$$

2. RGB 色彩空间

RGB 色彩空间的颜色分别由 R(red)、G(green)、B(blue)三原色混合而成。每个值取值 0～255。RGB 值越大，颜色越亮。RGB 值都是 255 为白色，RGB 值都是 0 为黑色。

3. CMYK 色彩空间

CMYK 是一种彩色印刷使用的一种色彩模式。它由青(Cyan)、紫红(Magenta)、黄(Yellow)和黑(Black)4 种颜色组成。其中黑色用 K 来表示，区别于 RGB 三基色中的蓝色 B。这种色彩空间的创建和 RGB 不同，它不是靠增加光线，而是靠减去光线，因为打印纸不能创建光源，不会发射光线，只能吸收和反射光线。因此，通过该 4 种颜色组合，便可产生可见光谱中的绝大部分颜色。

4. HSV 色彩空间

RGB 是从硬件的角度提出的颜色模型，在与人眼匹配的过程中存在一定的差异，HSV 色彩空间是一种面向视觉感知的颜色模型。HSV 色彩空间指出色彩主要包含 3 个要素，即色调(H)、饱和度(S)和亮度(V)。HSV 色彩空间将亮度与反映色彩本质特性的两个参数—色调和饱和度分开处理。光照明暗给物体颜色带来的直接影响就是亮度分量，所以若能将亮度分量从色彩中提取出去，而只用反映色彩本质特性的色度、饱和度来进行聚类分析，会获得比较好的效果。这也正是 HSV 色彩空间在彩色图像处理和计算机视觉的研究中经常被使用的原因。

从 RGB 色彩空间转换到 HSV 色彩空间之前，需要先将 RGB 色彩空间的值转换到[0，1]，然后再进行处理。具体处理方法为

$$S=\begin{cases}\dfrac{V-\min(R,G,B)}{V}, & V\neq 0\\ 0, & \text{其他情况}\end{cases}\qquad V=\max(R,G,B)$$

$$H=\begin{cases}\dfrac{60(G-B)}{V-\min(R,G,B)}, & V=R\\ 120+\dfrac{60(B-R)}{V-\min(R,G,B)}, & V=G\\ 240+\dfrac{60(R-G)}{V-\min(R,G,B)}, & V=B\end{cases}$$

5. HLS 色彩空间

HLS 色彩空间包含色调 H、明度 L 和饱和度 S 三要素。与 HSV 色彩空间类似，只是用明度 L 替换了亮度 V。

6. YUV 色彩空间

YUV YCrCb 色彩空间是一种传输色彩模型，是电视系统常用的色彩空间。该色彩空间包含 Y 亮度和两个色差分量 U、V。YUV 和 RGB 之间的转换关系为

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \frac{1}{256} \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112 \\ 112 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}$$

3.4.2　色彩空间的转换

每个色彩空间都有处理问题的优势，所以为了方便地处理某个问题，就要用到色彩空间的转换。色彩空间的转换就是将图像从一个色彩空间转换到另一个色彩空间。

在大多数情况下看见的彩色图片都是 RGB 类型，但在图像处理时，需要用到灰度图、二值图、HSV、HLS 等颜色模式。例如，在 Python 中使用 OpenCV 处理图像时，可能会在 RGB 色彩空间和 HSV 色彩空间之间进行转换。在进行图像的特征提取、距离计算时，需要先将图像从 RGB 色彩空间处理为灰度色彩空间。在一些应用中，可能需要将色彩空间的图像转换为二值图像。

1. 使用 OpenCV 库实现

OpenCV 对于色彩空间的转换提供了很好的支持，cvtColor()函数可实现色彩空间的转换。语法格式：

```
dst = cvtColor(src, code,dst = None,dstCn = None)
```

其中：

① dst：输出图像，与原始图像的数据类型和深度一致。

② src：原始图像。

③ code：指定颜色空间转换类型。常见转换类型有以下几种。

cv2.COLOR_BGR2RGB：转换成 RGB。

cv2.COLOR_BGR2GRAY：转换成灰度图。

cv2.COLOR_BGR2HSV：转换成 HSV 模式。

cv2.COLOR_BGR2HLS：转换成 HLS 模式。

cv2.COLOR_BGR2Lab：转换成 Lab 模式。

cv2.COLOR_BGR2YCrCb：转换成 YCrCb 模式。

④ dstCn：指定目标图像通道数；默认 None，会根据 src、code 自动计算。

函数的作用是将一个图像从一个颜色空间转换到另一个颜色空间。从 BGR 向其他类型转换时，必须明确指出图像的颜色通道。在 OpenCV 中默认的颜色模式排列是 BGR，而不是 RGB。

常用的颜色空间转换主要是 RGB－灰度和 RGB－HSV。

【例 3.19】 将 BGR 图像转换为灰度图像。

```
import cv2
import Matplotlib.pyplot as plt
bgr = cv2.imread('fruit.png')
```

```
gray = cv2.cvtColor(bgr,cv2.COLOR_BGR2GRAY)                    #将原始图像转换为灰度模式
print ("bgr.shape = ", bgr.shape)
print ("gray.shape = ",gray.shape)
cv2.imshow("BGR",bgr)
cv2.imshow("GRAY",gray)
cv2.waitKey()
cv2.destroyAllWindows()
```

程序运行如图 3.30 所示。

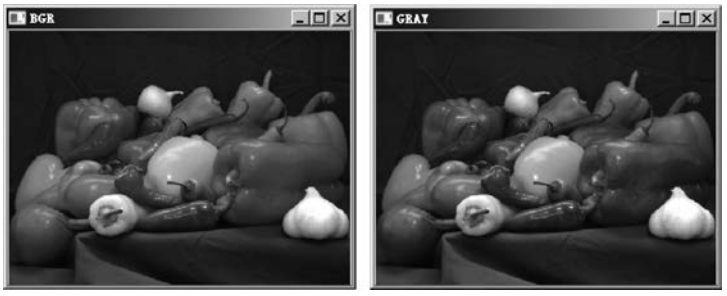


图 3.30　RGB 图像转换为灰度图像

运行结果为：

```
bgr.shape = (218, 293, 3)
gray.shape = (218, 293)
```

程序运行同时显示原始图像、灰度图像和 RGB 图像，并显示了各个图像的 shape 属性，可以看到图像在转换前后的色彩空间变化情况。

2. 使用 Skimage 库实现

Skimage 的 color 模块实现了所有的颜色空间转换函数。颜色空间转换后，图片类型都变成了 float 型。常用色彩空间转换函数如表 3.6 所示。

表 3.6　Skimage 色彩空间转换函数

转 换 函 数	作　　用
skimage.color.rgb2grey(rgb)	将 RGB 模式转换成灰度模式
skimage.color.rgb2hsv(rgb)	将 RGB 模式转换成 HSV 模式
skimage.color.rgb2lab(rgb)	将 RGB 模式转换成 Lab 模式
skimage.color.gray2rgb(image)	将灰度模式转换成 RGB 模式
skimage.color.hsv2rgb(hsv)	将 HSV 模式转换成 RGB 模式
skimage.color.lab2rgb(lab)	将 Lab 模式转换成 RGB 模式

【例 3.20】 将 RGB 图像转换成灰度图像。

```
from skimage import io,data,color
img = data.astronaut()                  #读取宇航员图片
gray = color.rgb2gray(img)              #RGB模式转换为灰度模式
io.imshow(gray)
io.show()
```

运行结果如图 3.31 所示。

上面的所有转换函数，还可以用 convert_colorspace()函数来代替。语法格式如下：

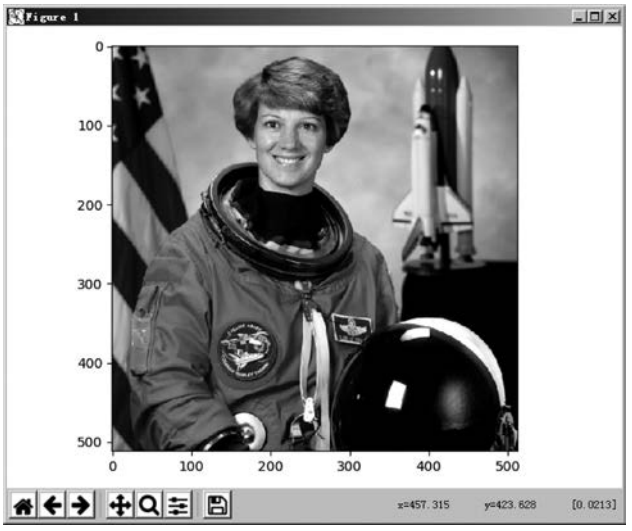


图 3.31　RGB 图像转换成灰度图像

```
skimage.color.convert_colorspace(arr,fromspace,tospace)
```

① arr：表示原始图像。

② fromspace：表示图像原来的色彩空间。

③ tospace：表示转换后的色彩空间。

【例 3.21】 将 RGB 图像转换成 HSV 图像。

```
from skimage import io,data,color
img = data.astronaut()              # 自带的宇航员图片
hsv = color.convert_colorspace(img,'RGB','HSV')
io.imshow(hsv)
io.show()
```

运行结果如图 3.32 所示。

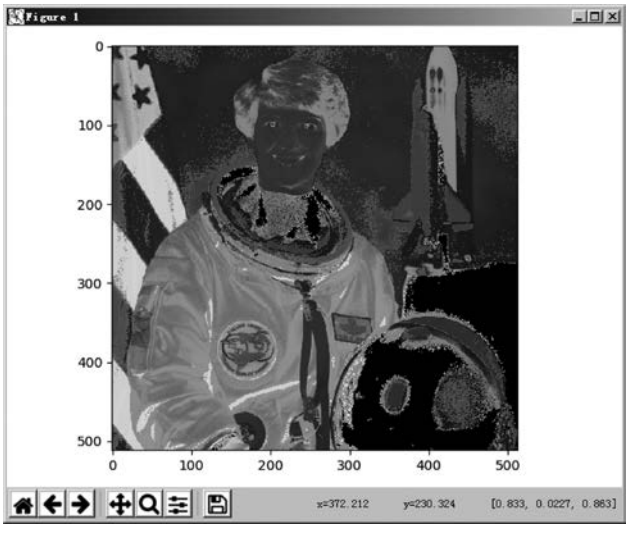


图 3.32　RGB 图像转换成 HSV 图像

Skimage 程序自带了一些示例图片，见表 3.3，如果不想从外部读取图片，就可以直接使用这些示例图片。图片名对应的就是函数名，如 camera 图片对应的函数名为 camera()。这些示例图片存放在 Skimage 的安装目录下，路径名称为 data_dir。显示这些图片可直接使用函数名，如 img＝data.astronaut()。

3.4.3 通道的拆分和合并

RGB 图像可以拆分出 R 通道、G 通道、B 通道，而在 OpenCV 中，通道是按照 B 通道、G 通道、R 通道顺序存储的。

1. 通道拆分

(1) 通过索引拆分。

通过索引方式，可以直接将各个通道从图像中提取出来。例如，使用 OpenCV 库读取 RGB 图像 img，分别提取 B 通道、G 通道、R 通道图像信息，代码如下：

```
import cv2
img = cv2.imread('lena.jpg')
b = img[:,:,0]
g = img[:,:,1]
r = img[:,:,2]
```

(2) 通过 split()函数拆分。

OpenCV 中提供了函数 cv2.split()将原始图像的各个通道进行分离。比如：RGB 图像，可以将其 R、G、B 这 3 个颜色通道分离。语法格式如下：

```
b,g,r = cv2.split(img)
```

b、g、r 分别返回 B 通道、G 通道、R 通道图像信息。

【例 3.22】 读取一个图像，使用函数 cv2.split()拆分图像通道。

```
import cv2
img = cv2.imread('fruit.png')
b,g,r = cv2.split(img)
cv2.imshow('B',b)
cv2.imshow('G',g)
cv2.imshow('R',r)
cv2.waitKey()
cv2.destroyAllWindows()
```

运行程序如图 3.33 所示，得到读取图像的 3 个通道图像。

图 3.33 RGB 图像通道拆分

【注意】 RGB 在 OpenCV 中存储为 BGR 的顺序。而且，cv2. split 的速度比直接索引要慢，但 cv2. split 返回的是副本，直接索引返回的是引用（改变 B 就会改变 BGR）。

2. 通道合并

函数 cv2. merge()可以实现图像通道的合并。Merge()函数与 Split()函数是相对的，其将多个通道的序列合并起来，组成一个多通道的图像。语法格式如下：

```
bgr = cv2.merge(mode,channels)
```

① mode：合并之后的图像模式，如 RGB。

② channels：多个单一通道组成的序列。

【例 3.23】 对 RGB 图像进行拆分再合并。

得到 B 通道图像、G 通道图像、R 通道图像，使用 cv2. merge()函数将 3 个通道合并为一幅三通道彩色图像。

```
import cv2
lena = cv2.imread("lenatest.jpg")
b,g,r = cv2.split(lena)
bgr = cv2.merge([b,g,r])
rgb = cv2.merge([r,g,b])
cv2.imshow("BGR",bgr)
cv2.imshow("RGB",rgb)
cv2.waitKey()
cv2.destroyAllWindows()
```

运行结果如图 3.34 所示。

图 3.34　RGB 图像通道拆分再合并

3.5　数字图像的基本运算

3.5.1　图像的点运算

图像的点处理运算（Point Operation）是一种通过图像中的每一个像素值（像素点的灰度值）进行运算的图像处理方式。点运算变换函数将图像的像素一一转化，最终构成一幅新的图像。由于操作对象是图像的单个像素值，故得名“点运算”。其特点就是输出图像每个像素点的灰度值仅由对应的输入像素点的灰度值决定，运算结果不会改变图像内像素点之

间的空间位置关系。点运算用于改变图像的灰度范围及分布，实现图像的对比度增强、对比度拉伸或灰度变换等，是数字图像处理中最基础的技术。

设 $g(x,y)$ 表示输入图像各点的像素值，$f(x,y)$ 表示输出图像各点的像素值，T 表示点运算的关系函数。点运算的处理过程可以用如下公式表示，即

$$g(x,y)=T[f(x,y)]$$

图像的点运算分为线性点运算和非线性点运算。

1. 线性点运算

线性点运算是输出灰度级与输入灰度级呈线性关系的点运算，即 $T[\cdot]$ 为线性函数，即

$$g(x,y)=T[f(x,y)]=af(x,y)+b$$

显然：

当 $a>1$ 时，输出图像的对比度将增大；

当 $0<a<1$ 时，输出图像的对比度将减小；

当 $a=1$ 且 $b\neq 0$ 时，所有像素灰度的上移或下移，整个图像更暗或更亮；

当 $a=1,b=0$ 时，原始图像不发生变化；

当 $a<0$ 时，则暗区域将变亮，亮区域将变暗，图像求补运算。

更为简便通俗的理解如下。

$a=1,b=0$：恒等。

$a<0$：黑白翻转。

$|a|>1$：增加对比度。

$|a|<1$：减小对比度。

$b>0$：增加亮度。

$b<0$：减小亮度。

【例 3.24】 利用线性点运算，调整图像对比度。

```
import cv2
img = cv2.imread('flower.jpg',1)
cv2.imshow('orignal',img)                    #显示原始图像
img1 = img * 1.05                            #增强对比度
cv2.imshow('up',img1)
img2 = img * 0.85                            #减小对比度
cv2.imshow('down',img2)
#增加灰度值
img3 = img + 50                              #灰度值增加 50
cv2.imshow('add',img3)
#图像反色,求补运算
img4 = 255 - img
cv2.imshow('reverse',img4)
cv2.waitkey()                                #无限等待键盘输入
cv2.destroyallwindows                        #删除窗口
```

运行结果如图 3.35 所示。

线性点运算还可以分段灰度处理，突出感兴趣的目标或灰度区间，相对抑制那些不感兴趣的灰度区域，用于数字图像的局部处理。

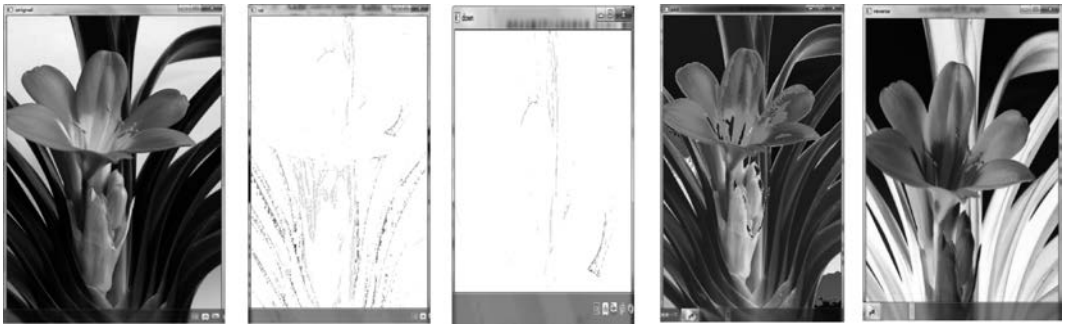


图 3.35　线性点运算改变图像的灰度值

2. 非线性点运算

非线性点运算是指输出灰度级与输入灰度级成非线性关系，即 $T[\cdot]$为非线性函数。常用非线性函数有对数函数、幂次函数和分段线性函数。引入非线性点运算主要是考虑到在成像时，可能由于成像设备本身的非线性失衡，需要对其进行校正，或者强化部分灰度区域的信息。

【例 3.25】 实现图像灰度的对数变换。结果如图 3.36 所示。

```
import numpy as np
import Matplotlib.pyplot as plt
import cv2
deflog_plot(c):
  x = np.arange(0, 256, 0.01)
  y = c * np.log(1 + x)
  plt.plot(x, y, 'r', linewidth = 1)        #绘制曲线
  plt.rcParams['font.sans - serif'] = ['SimHei']
  plt.title(u'对数变换函数')                  #正常显示中文标签
  plt.xlim(0, 255), plt.ylim(0, 255)
  plt.show()
#对数变换
def log(c,img):
  output = c * np.log(1.0 + img)
  output = np.uint8(output + 0.5)
  return output
img = cv2.imread('girl.png')                #读取原始图像
log_plot(42)                                #绘制对数变换曲线
output = log(42,img)                        #图像灰度对数变换
cv2.imshow('Input',img)
cv2.imshow('Output', output)                #显示图像
cv2.waitKey(0)                              #键盘绑定函数,等待(n)ms,设置 0 无限等待键盘输入
cv2.destroyAllWindows()                     #删除窗口
```

3. 点运算应用

(1) 光度学标定。

希望数字图像的灰度能够真实地反映图像的物理特性，如去掉非线性、变换灰度的单位。

(2) 对比度增强和对比度扩展。

将感兴趣特征的对比度扩展，使之占据可显示灰度级的更大部分。

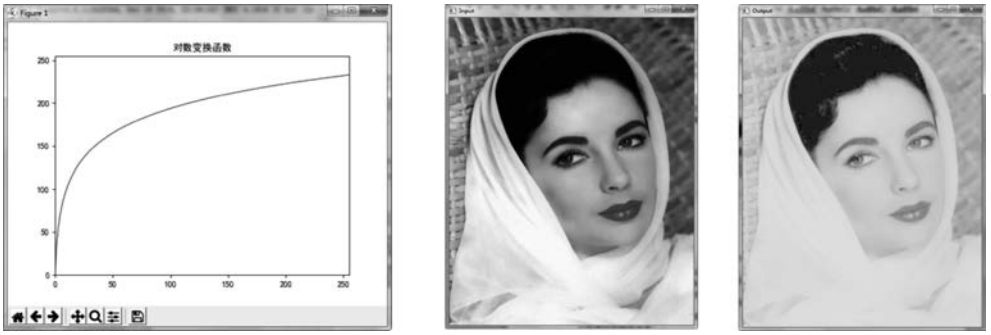


图 3.36　图像灰度的对数变换效果(左侧对数图形,中间原图,右侧变换图)

(3) 显示标定。

显示设备不能线性地将灰度值转换为光强度,因此点运算和显示非线性组合,以保持显示图像时的线性关系。

(4) 轮廓线确定。

用点运算进行阈值化。

(5) 裁剪。

每次点运算的最后一步,都将负值置为 0;而将正值约束在灰度级最大值。

3.5.2　图像的代数运算

在数字图像处理中,代数运算具有非常广泛的应用。数字图像的运算方式在实际的医学数字图像处理中可用于医学数字图像的比较、裁剪、拼接、特征提取等融合技术中。

1. 基本的代数运算

数字图像的代数运算是指图像像素位置不变,两幅或多幅图像通过对应像素之间的加、减、乘、除运算得到输出图像的方法。图像的代数运算,一种是图像和一个常数进行运算,一种是两幅或多幅图像的运算。

设图像为 $f(x,y)$,常数为 c,即 c 级灰度,那么图像和常数的代数运算的数学表达式为

$$g(x,y)=f(x,y)+c$$
$$g(x,y)=f(x,y)-c$$
$$g(x,y)=f(x,y)\times c$$
$$g(x,y)=f(x,y)\div c$$

设两幅图像为 $f(x,y)$、$h(x,y)$,输出图像为 $g(x,y)$,两幅图像的代数运算的数学表达式为

$$g(x,y)=f(x,y)+h(x,y)$$
$$g(x,y)=f(x,y)-h(x,y)$$
$$g(x,y)=f(x,y)\times h(x,y)$$
$$g(x,y)=f(x,y)\div h(x,y)$$

代数运算的用途如下。

(1) 加法运算。

图像与一个常数进行加法运算,可以给整幅图像增加灰度级,使图像亮度得到提高,整

体偏亮；还可以给个别像素加灰度值，可以使目标景物突出；通过对同一场景多幅图像求平均，可以降低叠加性随机噪声；两幅图像叠加达到二次曝光的效果等。

（2）减法运算。

图像的减法运算就是把两幅图像的差异显示出来，减法运算多用于去除图像的附加噪声；去除图像中不需要的叠加性图案；检测同一场景两幅图像之间的变化，如运动目标的跟踪及故障检测、计算物体边界的梯度等。

（3）乘、除法运算。

在数字图像处理中，乘、除运算应用相对较少，但也具有很重要的作用。乘法运算在获取图像的局部图案时发挥作用，将一幅图像与掩模图像（二值图像）相乘，可遮住该图像中的某些部分，使其仅保留图像中感兴趣的部分。在获取数字化图像中，图像数字化设备对一幅图像各点的敏感程度不可能完全相同，乘、除运算可用于纠正这方面的不利影响。除法运算还可以产生对颜色和多光谱图像分析十分重要的比率图像。

2. 像素操作实现代数运算

通过像素操作，实现常数和图像像素、两幅或多幅图像像素的代数运算。需要注意的是，进行代数运算的图像必须形状一致。

【例 3.26】 数字减影血管造影成像。

数字图像的减法运算可应用于 DSA（数字减影血管造影）的图像处理中。将受检部位没有注入造影剂和注入造影剂后的两幅图像的数字信息相减，获得了去除骨骼、肌肉和其他软组织，只留下单纯血管影像的减影图像。

实例通过对注射放射线液体前后的脑部拍摄 CT 图，再通过图像减法，获取血液流动情况。第一幅为注射前，第二幅为注射后，第三幅为通过减法得到。

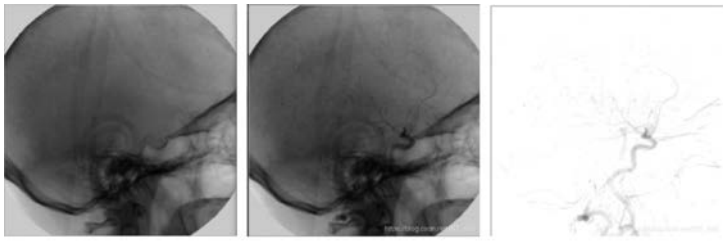
```
#图像减法 - 血液流动
import cv2
ori1 = cv2.imread('3.png')
#ori1 = cv2.cvtColor(ori1,cv2.COLOR_RGB2GRAY)
ori2 = cv2.imread('4.png')
#ori2 = cv2.cvtColor(ori2,cv2.COLOR_RGB2GRAY)
cv2.imshow('minus1',ori1)
cv2.imshow('minus2',ori2)
cv2.waitKey()
city3 = ori2 - ori1
city3[city3 <= 55] = 255
cv2.imshow('city',city3)
cv2.waitKey()
cv2.destroyAllWindows()
```

运行结果如图 3.37 所示。

3. 利用 OpenCV 实现代数运算

通过 OpenCV 中的 add()、subtract()、multiply()、divide()函数实现图像的代数运算，进行运算的图像要大小一致。语法格式如下：

```
dst = cv2.add(scr1,scr2)
dst = cv2.subtract (scr1,scr2)
dst = cv2.multiply (scr1,scr2)
```



(a) 注射前　(b) 注射后　(c) 减影图

图 3.37　数字减影血管造影成像

```
dst = cv2.divide (scr1,scr2)
```

其中，dst 表示目标图像，scr1、scr2 表示原始图像。

【例 3.27】 使用加号运算符和 cv2.add()函数计算两幅图像的像素值和。

```
import cv2
a = cv2.imread("lena.bmp",0)
b = a
result1 = a + b
result2 = cv2.add(a,b)
cv2.imshow("original",a)
cv2.imshow("result1",result1)
cv2.imshow("result2",result2)
cv2.waitKey()
cv2.destroyAllWindows()
```

运行结果如图 3.38 所示。

(a) 原始图像　(b) 使用加号运算得到的图像　(c) 使用add()函数得到图像

图 3.38　add()函数图像加法运算

使用加号运算符计算图像像素值的和时，将和大于 255 的值进行了取模处理，取模后大于 255 的这部分值变得更小了，导致本来应该更亮的像素点变得更暗了，相加所得的图像看起来并不自然。使用函数 cv2.add()计算图像像素值的和时，将和大于 255 的值处理为饱和值 255。图像像素值相加后让图像的像素值增大了，图像整体变亮。

OpenCV 中提供了 cv2.addWeighted()函数，用来实现图像的加权和(混合、融合)。语法格式如下：

```
dst = cv2.addWeighted(src1,alpha,src2,beta,gamma)
```

其中，参数 alpha 和 beta 是 src1 和 src2 所对应的系数，它们的和可以等于 1，也可以不等于 1。该函数实现的功能是 dst＝src1×alpha＋src2×beta＋gamma。需要注意，src1 和

src2 大小一致，式中参数 gamma 的值可以是 0，但是该参数是必选参数，不能省略。可以将上式理解为“结果图像＝图像 1×系数 1＋图像 2×系数 2＋亮度调节量”。

【例 3.28】 使用 cv2.addWeighted()函数将两幅图像混合。

```
import cv2
a = cv2.imread("Chrysanthemum.jpg",0)
b = cv2.imread("lena.jpg",0)
cv2.imshow("boat",a)
cv2.imshow("lena",b)
face = a[50:250,30:230]                    #选取混合区域
result = cv2.addWeighted(b,0.5,face,0.5,0)
a[50:250,30:230] = result
cv2.imshow("result",a)
cv2.waitKey()
cv2.destroyAllWindows()
```

运行结果如图 3.39 所示。

说明：利用 addWeighted()函数可以实现两个背景图片的融合，第 5 章和第 6 章都会用到该函数。

(a) 原始图像　　(b) 混合后的图像

图 3.39　addWeighted()函数图像加权混合

3.5.3　图像的几何变换

图像的几何变换是将一幅图像中的坐标映射到另一幅图像中的新坐标位置，它不改变图像的像素值，只是改变像素所在的几何位置，使原始图像按照需要产生位置、形状和大小的变化。

1. 基本几何变换

数字图像的基本几何变换包括图像的平移、旋转、放缩、镜像、转置等。

1) 图像的平移

图像的平移是几何变换中最简单、最常见的变换之一，它是将一幅图像上的所有点都按照给定的偏移量在水平方向、垂直方向上沿轴移动，平移后的图像与原始图像大小相同，如图 3.40 所示。设(x_0, y_0)为原始图像上的一点，图像水平平移量为 Δx，垂直平移量为 Δy，则平移后点坐标将变为(x', y')，它们之间的数学关系式为

$$x' = x_0 + \Delta x$$

$$y' = y_0 + \Delta y$$

矩阵的形式表示为

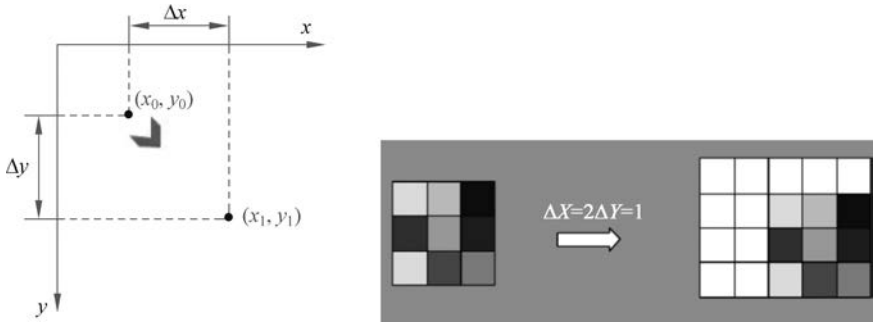


图 3.40　图像的平移

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

图像被平移后，有些点可能被移出显示区域，造成部分信息丢失。为避免被平移后丢失部分图像，可将新生成的图像宽度扩大$|\Delta x|$，高度扩大$|\Delta y|$。

当对图像做平移变化时，一定要根据平移距离估算图像平移后所需画布的大小(图像旋转和图像缩放都存在这一问题)，以免平移后信息的丢失。除了图像缩小变换外，对图像做其他几何变换还有可能出现坐标为小数的情况，因此，为尽可能保持几何运算后图像的质量，还要考虑图像的插值问题。

2) 图像的旋转

图像的旋转变换属于图像的位置变换，通常是以图像的中心为原点，将图像上的所有像素按顺时针方向或逆时针方向旋转一个相同的角度，如医学图像的旋转都按逆时针方向。

图像的旋转变换后，图像的大小一般会发生改变。和图像的平移一样，在图像旋转变换中既可以把转出显示区域的图像截去，也可以扩大图像范围以显示所有的图像。

图像的旋转变换也可以用矩阵变换表示。设点 $P_0(x_0, y_0)$ 旋转 θ 角后的对应点为 $P(x, y)$，如图 3.41 所示。那么，旋转前后点 $P(x, y)$ 的坐标分别为

$$\begin{cases} x = x_0\cos\theta + y_0\sin\theta \\ y = x_0\sin\theta + y_0\cos\theta \end{cases}$$

写成矩阵表达式为

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix}$$

进行图像旋转时需要注意以下 3 点。

(1) 图像旋转后像素的排列不是完全按照原有的相邻关系。这是因为相邻像素之间只能有 8 个方向，如图 3.42 所示。

(2) 图像旋转之前，为了避免信息的丢失，一定要有坐标平移。

(3) 图像旋转后，因像素值的填充是不连续的，会出现很多空洞点。对这些空洞点通常是通过插值的方法进行填充处理；否则画面效果不好。

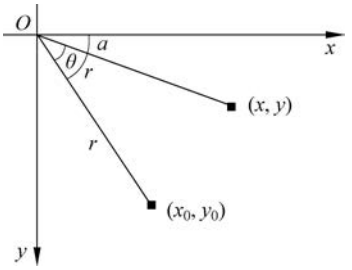


图 3.41　图像旋转 θ 角

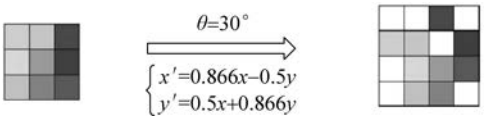


图 3.42　图像旋转后像素的排列

3）图像的镜像

图像的镜像分为两种，即垂直镜像和水平镜像，其中水平镜像是指图像的左半部分和右半部分以图像竖直中轴线为中心轴进行对换；垂直镜像是将图像上半部分和下半部分以图像水平轴线为中心轴进行对换，如图 3.43 所示。

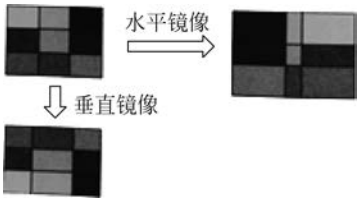


图 3.43　图像的镜像

设原始图像的值为

$$\boldsymbol{f}(x,y)=\begin{bmatrix} f(1,1),f(1,2),\cdots,f(1,m-1),f(1,m) \\ f(2,1),f(2,2),\cdots,f(2,m-1),f(2,m) \\ \vdots \\ f(n,1),f(n,2),\cdots,f(n,m-1),f(n,m) \end{bmatrix}$$

则水平变换后的图像值为

$$\boldsymbol{g}(x,y)=\begin{bmatrix} f(1,m),f(1,m-1),\cdots,f(1,2),f(1,1) \\ f(2,m),f(2,m-1),\cdots,f(2,2),f(2,1) \\ \vdots \\ f(n,m),f(n,m-1),\cdots,f(n,2),f(n,1) \end{bmatrix}$$

垂直变换后的图像值为

$$\boldsymbol{g}(x,y)=\begin{bmatrix} f(n,1),f(n,2),\cdots,f(n,m-1),f(n,m) \\ \vdots \\ f(2,1),f(2,2),\cdots,f(2,m-1),f(2,m) \\ f(1,1),f(1,2),\cdots,f(1,m-1),f(1,m) \end{bmatrix}$$

4）图像的转置

图像的转置即将图像的行、列像素值对调。

$$\boldsymbol{g}(x,y)=\begin{bmatrix} f(1,1),f(2,1),\cdots,f(n-1,1),f(n,1) \\ f(1,2),f(2,2),\cdots,f(n-1,2),f(n,1) \\ \vdots \\ f(1,m),f(2,m),\cdots,f(n-1,m),f(n,m) \end{bmatrix}$$

需要注意的是，进行图像转置后，图像的大小会发生改变。

5）图像的缩放

图像缩放是指将给定的图像在 x 轴方向按比例缩放 f_x 倍，在 y 轴方向按比例缩放 f_y

倍，从而获得一幅新的图像。如果在 x 轴方向和 y 轴方向缩放的比率相同，称这样的比例缩放为图像的全比例缩放。如果改变图像的比例缩放 $f_x \neq f_y$，会改变原始图像的像素间的相对位置，产生几何畸变。

(1) 图像的缩小。分为按比例缩小和不按比例缩小两种。图像缩小实际上是对原有的多个数据进行挑选或处理，获得期望缩小尺寸的数据，并且尽量保持原有的特征不丢失。最简单的方法就是等间隔地选取数据。

设水平、垂直方向均缩小 1/2，图像被缩到原始图像的 1/4。图像缩小方法效果如图 3.44 所示。

(2) 图像的放大。从字面上看，图像的放大就是图像缩小的逆操作，但从信息处理的角度看，则难易程度完全不一样。图像缩小是从多个信息中选出所需要的信息，而图像放大则是需要对多出的空位填入适当的值，是信息的估计。最简单的思想，如果将原始图像放大 k 倍，则将原始图像中的每个像素值，填在新图像中对应的大小的 $k \times k$ 子块中，如图 3.45 所示。

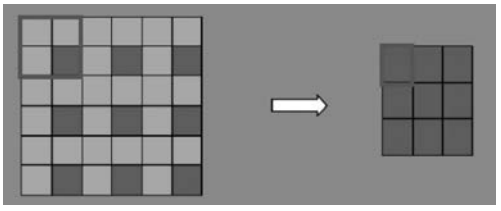


图 3.44　图像的缩小

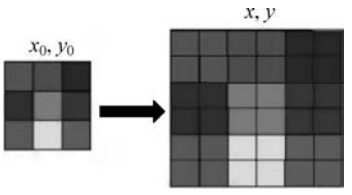


图 3.45　图像的放大

放大后的图像像素会出现“空洞”，即在几个像素中间的位置，这就需要利用灰度级插值算法来确定几何变换后的像素的灰度值。

设缩放前后两点 $A_0(x_0, y_0)$ 和 $A_1(x_1, y_1)$ 之间的关系为

$$\begin{cases} x_1 = \alpha x_0 \\ x_2 = \beta x_0 \end{cases}$$

用矩阵形式可以表示为

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix}$$

2. 利用 OpenCV 库实现图像几何变换

1) 图像的缩放

通过 resize()函数实现图像的缩放。语法格式如下：

```
dst = cv2.resize(src,dsize[,fx[,fy[,interpolation]]])
```

① scr：表示原始图像。

② dsize：表示缩放大小。

③ fx 和 fy：表示水平、垂直方向缩放比例。

④ interpolation：插值方法。常用 5 种方法，分别如下。

- INTER_NEAREST：最近邻插值法。

- INTER_LINEAR：双线性插值法(默认)。
- INTER_AREA：基于局部像素的重采样。
- INTER_CUBIC：基于 4×4 像素邻域的三次插值法。
- INTER_LANCZOS4：基于 8×8 像素邻域的 Lanczos 插值。

需注意以下几点。

① 在 resize()函数中,图像缩放的大小可以通过参数“dsize”和“fx、fy”两者之一来指定。

② 当缩小图像时,使用区域插值方式(INTER_AREA)能够得到最好的效果；当放大图像时,使用三次样条插值(INTER_CUBIC)方式和双线性插值(INTER_LINEAR)方式都能够取得较好的效果。三次样条插值方式速度较慢,双线性插值方式速度相对较快且效果并不逊色。

【例 3.29】 将图像进行缩放,图像大小为 200×100 像素。

```
import cv2
import numpy as np
#读取图片
original = cv2.imread('fruit.png')
print(original.shape)
#图像缩放
result = cv2.resize(original,(200,100))
print(result.shape)
#显示图像
cv2.imshow("original", original)
cv2.imshow("result", result)
cv2.waitKey()
cv2.destroyAllWindows()
```

运行结果：

```
(218, 293, 3)
(100, 200, 3)
```

结果如图 3.46 所示。

图 3.46　resize()函数图像缩放

乘以缩放系数进行图像缩放,代码如下：

```
result = cv2.resize(src, (int(cols * 0.6),int(rows * 1.2)))
```

通过参数(fx,fy)缩放倍数实现图像缩放,代码如下：

```
result = cv2.resize(src, None, fx = 0.3, fy = 0.3) #图像缩放为原来的 0.3 倍
```

2）图像的翻转

调用 flip()函数实现图像的翻转。语法格式如下：

```
dst = cv2.flip(src, flipCode)
```

① dst：表示翻转后的目标图像。

② src：表示原始图像。

③ flipCode：表示翻转类型。如果 flipCode 为 0，则以 x 轴为对称轴翻转，如果 flipCode＞0 则以 y 轴为对称轴翻转，如果 flipCode＜0，则在 x 轴、y 轴方向同时翻转。

【例 3.30】 使用 flip()函数实现图像的翻转。

```
import cv2
import Matplotlib.pyplot as plt
#读取图片
img = cv2.imread('fruit.png')
#图像翻转
#  =0 以 X 轴为对称轴翻转 >0 以 Y 轴为对称轴翻转 <0 X 轴 Y 轴翻转
img1 = cv2.flip(img, 0)
img2 = cv2.flip(img, 1)
img3 = cv2.flip(img, -1)
#显示图形
cv2.imshow('img',img)
cv2.imshow('img1',img1)
cv2.imshow('img2',img2)
cv2.imshow('img3',img3)
plt.show()
cv2.waitKey()
cv2.destroyAllWindows()
```

程序运行结果如图 3.47 所示。

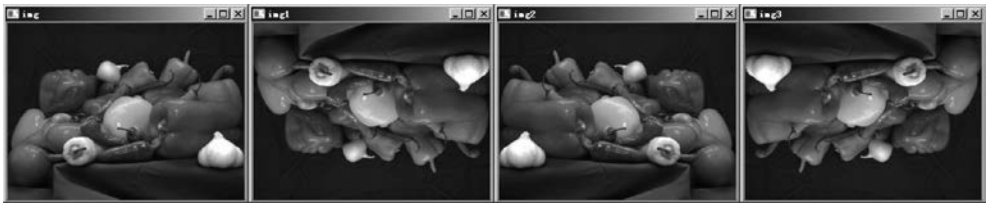


图 3.47　flip()函数图像翻转

3）图像的平移

利用 OpenCV 实现图像平移，首先定义平移矩阵 $\boldsymbol{M}$，再调用 warpAffine()函数实现平移。语法格式如下：

```
M = np.float32([[1, 0, x], [0, 1, y]])
dst = cv2.warpAffine(scr, M,dsize)
```

① dst：返回变换后的输出图像。

② scr：表示原始图像。

③ $\boldsymbol{M}$：表示一个 2×3 变换矩阵。使用不同的变换矩阵可实现不同的图像变换。

④ dsize：表示输出图像的大小。

【例 3.31】 将图像右移 50 像素、下移 50 像素。

```
import cv2
import numpy as np
import Matplotlib.pyplot as plt
#读取图片
img = cv2.imread('fruit.png')
#图像右移50、下移50
M = np.float32([[1, 0, 50], [0, 1, 50]])
img1 = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]))
#显示图形
cv2.imshow('original',img)
cv2.imshow('move',img1)
plt.show()
cv2.waitKey()
cv2.destroyAllWindows()
```

运行结果如图3.48所示。

图3.48　图像平移

4）图像的旋转

warpAffine()函数还可以实现图像旋转，可以通过getRotationMatrix2D()函数获取转换矩阵。getRotationMatrix2D()函数语法格式如下：

```
M = cv2.getRotationMatrix2D(center, angle,scale)
```

① center：表示旋转的中心点。

② angle：表示旋转的角度。正数表示逆时针方向旋转；负数表示顺时针方向旋转。

③ scale：表示变换尺度(缩放大小)。

【例3.32】 将图像绕中心旋转，逆时针方向旋转45°，并将目标图像缩小为原始图像的0.6倍。

```
import cv2
import numpy as np
#读取图片
src = cv2.imread('fruit.png')
#原图的高、宽以及通道数
rows, cols, channel = src.shape
#绕图像的中心旋转，逆时针方向旋转30°，并缩小为原来的0.6倍
M = cv2.getRotationMatrix2D((cols/2, rows/2), 30, 0.6)
rotated = cv2.warpAffine(src, M, (cols, rows))
#显示图像
cv2.imshow("src", src)
cv2.imshow("rotated", rotated)
```

```
#等待显示
cv2.waitKey(0)
cv2.destroyAllWindows()
```

运行结果如图 3.49 所示。

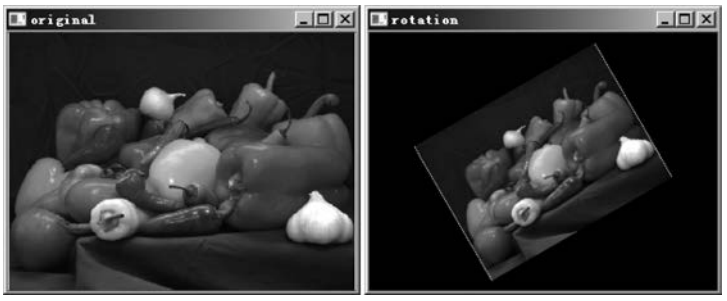


图 3.49　warpAffine()函数图像旋转

3. 使用 PIL 库实现图像几何变换

1) 图像的缩放

使用 PIL 中的 resize()函数实现图像的缩放，直接通过输入参数指定缩放后的尺寸即可。

【例 3.33】 将图像缩放为 128×128 像素。

```
from PIL import image
#读取图像
im = image.open("fruit.png")
im.show()
#将图像缩放为 128x128
im_resized = im.resize((128, 128))
im_resized.show()
```

运行结果如图 3.50 所示。

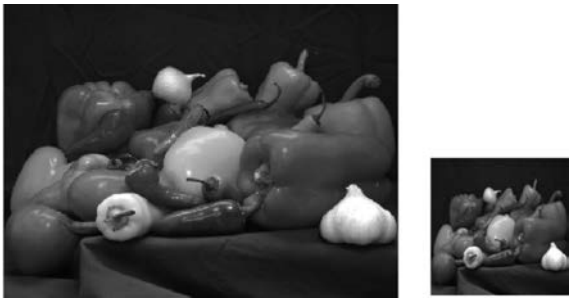


图 3.50　原始图像 293×218 和缩放后图像 128×128

2) 图像的旋转

使用 rotate()函数实现图像的旋转，通过输入参数直接指定按逆时针方向旋转的角度即可。

【例 3.34】 将图像逆时针方向旋转 45°。

```
from PIL import Image
#读取图像
```

```
im = Image.open("lenna.jpg")
im.show()
#将图像逆时针方向旋转 45°
im_rotate = im.rotate(45)
im_rotate.show()
```

运行结果如图 3.51 所示。

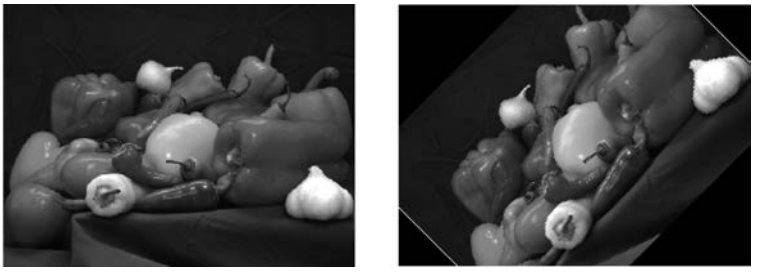


图 3.51　rotate()函数图像旋转

3) 图像的翻转

使用 transpose()函数实现图像的翻转，不仅支持上下、左右翻转；也支持逆时针方向 90°、180°、270°等角度的旋转，效果与 rotate()相同。语法格式如下：

```
dst = transpose(src,method)
```

① src：表示原始图像。

② method：表示旋转方式。有如下值：

- Image.FLIP_LEFT_RIGHT，表示将图像左右翻转；
- Image.FLIP_TOP_BOTTOM，表示将图像上下翻转；
- Image.ROTATE_90，表示将图像逆时针方向旋转 90°；
- mage.ROTATE_180，表示将图像逆时针方向旋转 180°；
- mage.ROTATE_270，表示将图像逆时针方向旋转 270°；
- mage TRANSPOSE，表示将图像进行转置(相当于顺时针方向旋转 90°)；
- Image TRANSVERSE，表示将图像进行转置，再水平翻转。

例如：

```
out = im.transpose(Image.FLIP_LEFT_RIGHT)
out = im.transpose(Image.FLIP_TOP_BOTTOM)
out = im.transpose(Image.ROTATE_90)
out = im.transpose(Image.ROTATE_180)
out = im.transpose(Image.ROTATE_270)
```

4. 使用 skimage 库实现图像代数运算

在 skimage 的 transform 模块中提供了很多函数，用于实现图像的形变与缩放。

(1) 改变图像尺寸 resize()函数。

语法格式如下：

```
transform.resize(image,output_shape)
```

① image：原始图像。

② output_shape：新的图片尺寸。

【例 3.35】 将图像大小调整为 80×60 像素。

```
from skimage import transform,data
import Matplotlib.pyplot as plt
img = data.camera()
dst = transform.resize(img,(80,60))                    #将图片大小变为 80×60 像素
plt.figure('resize')
plt.subplot(121)
plt.title('before resize')
plt.imshow(img,plt.cm.gray)
plt.subplot(122)
plt.title('after resize')
plt.imshow(dst,plt.cm.gray)
plt.show()
```

运行结果如图 3.52 所示。

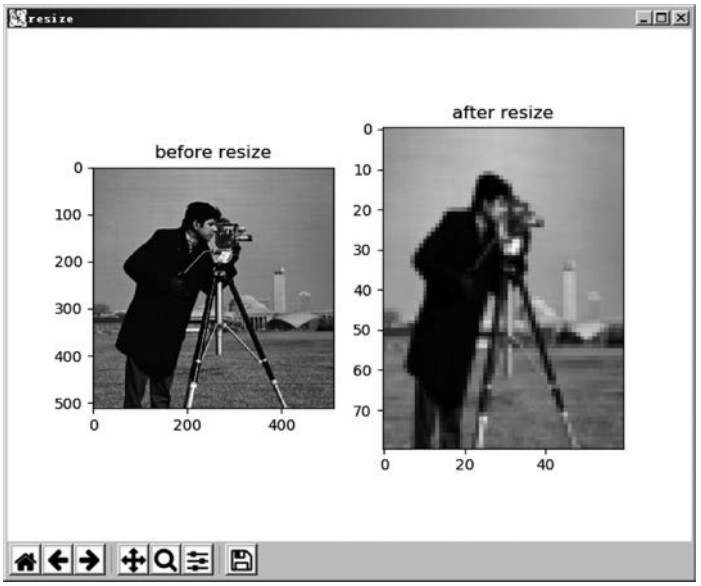


图 3.52 skimage 库更改图像尺寸

图像的大小由原来的 512×512 像素变成了 80×60 像素。

(2) 按比例缩放 rescale()函数。

语法格式如下：

```
transform.rescale(src,scale)
```

① src：表示原始图像。

② scale：表示缩放的倍数。可以是单个 float 数，也可以是一个 float 型的 tuple，如[0.2,0.5]表示将行列数分开进行缩放。

【例 3.36】 利用 rescale()函数实现图像的缩放。

```
from skimage import transform,data
img = data.camera()
print(img.shape)                                       #图像原始大小
print(transform.rescale(img,0.1).shape)                #缩小为原来图像大小的 0.1 倍
print(transform.rescale(img,[0.5,0.25]).shape)         #缩小为原来图像行数的 0.5,列数的 0.25
```

```
print(transform.rescale(img,2).shape)                    #放大为原来图像大小的2倍
```

运行结果：

```
(512,512)
(51,51)
(256,256)
(1024,1024)
```

(3) 旋转 rotate()函数。

语法格式如下：

```
transform.rotate(src,angle[,…],resize=false)
```

① src：表示原始图像。

② angle：参数是个 float 类型数，表示旋转的度数。

③ resize：用于控制在旋转时是否改变大小，默认为 False。

【例 3.37】 将图像进行旋转。

```
from skimage import transform,data
import Matplotlib.pyplot as plt
img = data.camera()
print(img.shape)                                         #图片原始大小
img1 = transform.rotate(img, 60)                         #旋转90°,不改变大小
print(img1.shape)
img2 = transform.rotate(img, 30,resize=True)             #旋转30°,同时改变大小
print(img2.shape)
plt.figure('resize')                                     #窗口名称
plt.subplot(121)                                         #在第一行第一列子图区绘制
plt.title('rotate 60')
plt.imshow(img1,plt.cm.gray)
plt.subplot(122)
plt.title('rotate 30')
plt.imshow(img2,plt.cm.gray)
plt.show()
```

运行结果如图 3.53 所示。

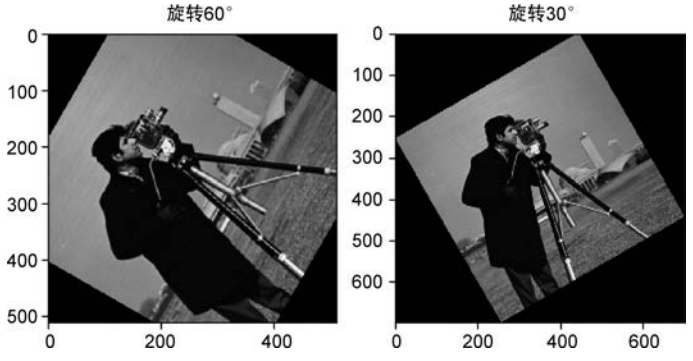


图 3.53　rotate()函数图像旋转

需注意以下几点。

① 在程序中，plt.subplot(121)语句相当于 plt.subplot(1,2,1)。

② plt.imshow()可以打印出图像的数字形式，但是无法可视化地显示出来。

3.6 数字图像的插值

3.6.1 插值的概念

数字图像都是以像素为单位的离散点来存储的，在很多时候对所获取的数字图像需要作进一步的处理。比如：为了做广告宣传，需要将拍摄的艺术照片做成巨幅海报；为了分析深层地质结构，需要对仪器采集的图像做局部细化；为了分析外星球的大气和地面状况，需要使遥感卫星图片模糊细节变得有意义；为了侦破缺少目击证人的案件，需要对监控录像做清晰化处理。输出图像的每一个像素点在输入图像中都有一个具体的像素点与之对应，就需要用到图像的像素点插值技术。图像插值是在基于模型框架下，从低分辨率图像生成高分辨率图像的过程，用以恢复图像中所丢失的信息。有时图像在获取、传输过程中不可避免地会产生噪声，这些噪声大大损坏了图像的质量，影响了图像的可用性，所以考虑要对图像进行去噪。而去噪的实质，是在去噪模型下用新的灰度估计值来取代原噪声点的灰度值，因此去噪也可以转化为插值问题来研究。

通过前面的介绍，我们知道数字图像在旋转时输出图像像素点坐标有可能对应于输入图像上几个像素点之间的位置；在图像放大时，有新的空像素填补灰度值，这些就需要通过灰度插值处理来计算出该输出点的灰度值。

插值分为图像内插值和图像间插值。图像内插值主要应用于对图像进行放大及旋转等操作，是根据一幅较低分辨率图像再生出另一幅均具有较高分辨率的图像。图像间的插值，也叫图像的超分辨率重建，是指在一图像序列之间再生出若干幅新的图像，可应用于医学图像序列切片和视频序列之间的插值图像。内插值实际上是对单帧图像的图像重建过程，这就意味着生成原始图像中没有的数据。常见的图像插值方法有最近邻插值、双线性插值、双平方插值、双立方插值及其他高阶方法。

图像插值技术广泛应用于军事雷达图像、卫星遥感图像、天文观测图像、地质勘探数据图像、生物医学切片及显微图像等特殊图像及日常人物景物图像的处理。按照应用目的，图像插值技术的应用场合可归为以下几种情况。

(1) 在图像采集、传输和现实过程中，不同的显示设备有着不同的分辨率，需要对视频序列和图像进行分辨率转换，如大屏幕显示图像和制作巨幅广告招贴画。

(2) 当用户需要专注于图像的某些细节时，对图像进行放缩变换，如图像浏览软件中的放大镜功能。

(3) 在视频传输中，为了有效利用有限的带宽，可以传输低分辨率的视频流，然后在接收端使用插值算法转换成高分辨率视频流。

(4) 为提高图像的存储和传输效率，而进行图像的压缩和重构，如计算机虚拟现实技术中的图像插值。

(5) 在图像恢复时，已经被损坏的图像或者有噪声污染的图像，可通过插值对图像进行重建和恢复，如警方在侦破案件时所发现的存在污损的身份证照片。

3.6.2　最邻近插值法

最近邻插值算法又称为零阶插值，它是一种比较容易实现且算法复杂度较低的插值算法。其原理是取待插值点周围 4 个相邻像素点中距离最短的一个邻点的灰度值作为该点的灰度值，如图 3.54 所示。

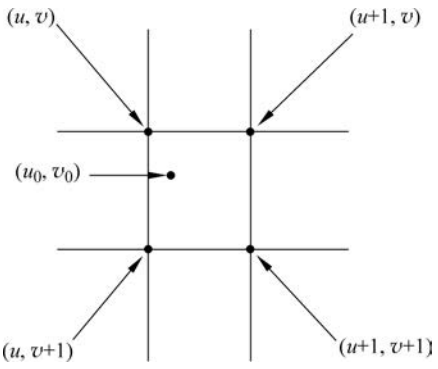


图 3.54　最邻近插值

假设，整数坐标(u,v)与点距离最近，则有

$$f(u_0,v_0)=f(u,v)$$

这种插值方法只用到距离及一个点的灰度值，简单、快速。但由于仅用对该插值点最近的像素的灰度值作为该点的值，没有考虑其他相邻像素的影响，因此插值后得到的图像会造成插值生成的图像灰度上的不连续，造成图像模糊，在灰度变化的地方可能出现明显的锯齿状和马赛克。

3.6.3　双线性内插法

双线性插值法是一阶插值，和最近邻插值算法（零阶插值）相比可产生更令人满意的效果，是对最近邻插值法的一种改进。

双线性插值原理是待插点像素值取原始图像中与其相邻的 4 个点像素值的水平、垂直两个方向上的线性内插，即根据待采样点与周围 4 个邻点的距离确定相应的权重，从而计算出待采样点的像素值。经过此算法处理后的图像，会产生许多新的像素值，它们主要由插值点周围像素的灰度值通过插值运算获得。

双线性内插法是利用待求像素 4 个邻像素的灰度在两个方向上作线性内插，要经过 3 次插值获得最终的结果，即先对两水平方向进行一阶线性插值，然后在垂直方向上进行一阶线性插值，如图 3.55 所示。

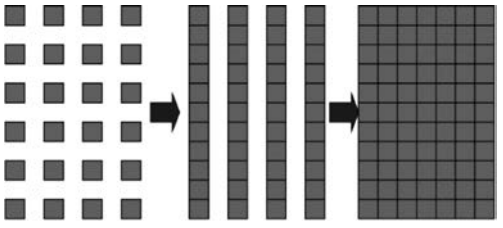


图 3.55　双线性插值

用$[S]$表示不超过 S 最大整数，则

$$u=[u_0]$$
$$v=[v_0]$$
$$\alpha=u_0-[u_0]$$
$$\beta=v_0-[v_0]$$

根据(u_0,v_0)4 个邻点灰度值，插值计算 $f(u_0,v_0)$，首先做水平方向插值。

第 1 步：从 $f(u,v)$及 $f(u+1,v)$求 $f(u_0,v)$。

即

$$f(u_0,v)=f(u,v)+\alpha[f(u+1,v)-f(u,v)]$$

第 2 步：从 $f(u,v+1)$及 $f(u+1,v+1)$求 $f(u_0,v+1)$

即

$$f(u_0,v+1)=f(u,v+1)+\alpha[f(u+1,v+1)-f(u,v+1)]$$

最后，做垂直方向插值，即

$$\begin{aligned}f(u_0,v_0)&=f(u_0,v)+\beta[f(u_0,v+1)-f(u_0,v)]\\&=f(u,v)(1-\alpha)(1-\beta)+f(u+1,v)\alpha(1-\beta)+f(u,v+1)(1-\alpha)\beta+\\&\quad f(u+1,v+1)\alpha\beta\end{aligned}$$

假设要得到点 $f(x,y)$的像素值[x、y 非整数，周围点的坐标为(0,0)(1,0)(0,1)(1,1)]，那么双线性插值的公式为

$$f(x,0)=f(0,0)+x[\,f(1,0)-f(0,0)\,]$$
$$f(x,1)=f(0,1)+x[\,f(1,1)-f(0,1)\,]$$
$$f(x,y)=f(x,0)+y[\,f(x,1)-f(x,0)\,]$$

双线性内插法的计算比最邻近点法复杂，计算量较大，但没有灰度不连续的缺点，结果基本令人满意。此方法仅考虑待测样点周围 4 个直接邻点灰度值的影响，而未考虑各邻点间灰度值变化率的影响，因此它具有低通滤波性质，使图像的高频分量受损，图像轮廓会在一定程度上变得比较模糊。该方法的输出图像与输入图像相比，仍然存在由于插值函数设计不周到而产生的图像质量受损与计算精度不高的问题。

3.6.4 三次多项式插值

对于图像灰度变化规律较复杂的图像，用两个邻点对间的数据点线性插值是不能得到较好结果的。可采用在同一直线方向上的更多采样点灰度对该数据点做非线性插值，常用的方法就是多项式插值。

已知数据表列

$$y_i \cong y(x_i)$$

试构造一多项式，使之在所有 x_i 处，满足 $y_i \cong y(x_i)$。插值多项式

$$y=c_0+c_1x+c_2x^2+\cdots+c_nx^n$$

n 阶多项式，须用 $n+1$ 个数据点来求出 $c_0,c_1,\cdots,c_n$。

通过线性方程组对系数求解，即

$$\begin{bmatrix}1 & x_0 & x_0^2 & \cdots & x_0^n\\1 & x_1 & x_1^2 & \cdots & x_1^n\\\vdots & \vdots & \vdots & & \vdots\\1 & x_n & x_n^2 & \cdots & x_n^n\end{bmatrix}\begin{bmatrix}c_0\\c_1\\\vdots\\c_n\end{bmatrix}=\begin{bmatrix}y_0\\y_1\\\vdots\\y_n\end{bmatrix}$$

n 阶多项式插值须用 $n+1$ 个数据点$(x_0,y_0),\cdots,(x_n,y_n)$。显然，线性插值是多项式插值的一个特例，即用 2 个数据点直线内插。

考虑到图像数据量较大，一般取三次多项式，精度基本可以保证。对每一维，三次多项

式插值需要用同一直线方向上的 4 个数据点做内插。

利用三次多项式 $S(x)$求逼近理论上最佳插值函数 $\sin(x)/x$，其数学表达式为

$$S(x)=\begin{cases}1-2|x|^2+|x|^3 & 0\leqslant|x|\leqslant 1\\ 4-8|x|+5|x|^2+|x|^3 & 1\leqslant|x|\leqslant 2\\ 0 & 2\leqslant|x|\end{cases}$$

在图像处理中，如二维医学图像插值须考虑 16 个邻点灰值影响，待求像素(x, y)的灰度值由其周围 16 个灰度值加权内插得到，如图 3.56 所示。

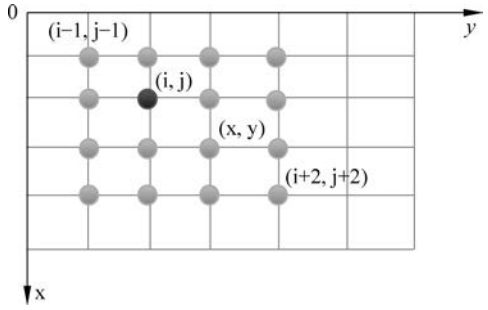


图 3.56　三次多项式插值

三次多项式插值方法计算量较大，但插值后的图像效果最好。

实训 1　数字图像的插值

利用 NumPy 和 OpenCV 实现图像插值，分别采用最邻近、双线性和双三次(Bell 分布)法。

实现过程如下。

(1) 以彩色图的方式加载图片。

(2) 根据想要生成的图像大小，映射获取某个像素点在原始图像中的浮点数坐标。

(3) 根据浮点数坐标确定插值算法中的系数、参数。

(4) 采用不同的算法实现图像插值。

参考代码如下：

```
#最邻近、双线性、双三次(Bell 分布)
import cv2
import numpy as np
import Matplotlib.pyplot as plt
def Nearest(img, bigger_height, bigger_width, channels):
    near_img = np.zeros(shape = ( bigger_height, bigger_width, channels ), dtype = np.uint8)
    for i in range( 0, bigger_height ):
        for j in range( 0,bigger_width ):
            row = (i / bigger_height ) * img.shape[0]
            col = ( j /bigger_width ) * img.shape[1]
            near_row =   round ( row )
            near_col = round( col )
            if near_row == img.shape[0] or near_col == img.shape[1]:
```

```
                    near_row -= 1
                    near_col -= 1
                near_img[i][j] = img[near_row][near_col]
    return near_img
def Bilinear(img, bigger_height, bigger_width, channels ):
    bilinear_img = np.zeros( shape = ( bigger_height, bigger_width, channels ), dtype = np.
uint8 )
        for i in range( 0, bigger_height ):
            for j in range( 0,bigger_width ):
            row = (i / bigger_height ) * img.shape[0]
            col = ( j /bigger_width ) * img.shape[1]
            row_int = int( row )
            col_int = int( col )
            u = row - row_int
            v = col - col_int
            if row_int == img.shape[0] - 1 or col_int == img.shape[1] - 1:
                row_int -= 1
                col_int -= 1
            bilinear_img[i][j] = (1 - u) * (1 - v) * img[row_int][col_int] + (1 - u) * v * img
[row_int][col_int + 1] + u * (1 - v) * img[row_int + 1][col_int] + u * v * img[row_int + 1][col_
int + 1]
    return bilinear_img
def Bicubic_Bell( num ):
    if  -1.5 <= num <= -0.5:
        return -0.5 * ( num + 1.5) ** 2
    if -0.5 < num <= 0.5:
        return 3/4 - num ** 2
    if 0.5 < num <= 1.5:
        return 0.5 * ( num - 1.5 ) ** 2
    else:
       return 0
def Bicubic (img, bigger_height, bigger_width, channels ):
    Bicubic_img = np.zeros( shape = ( bigger_height, bigger_width, channels ), dtype = np.
uint8 )
    for i in range( 0, bigger_height ):
        for j in range( 0,bigger_width ):
              row = (i / bigger_height ) * img.shape[0]
              col = ( j /bigger_width ) * img.shape[1]
              row_int = int( row )
              col_int = int( col )
              u = row - row_int
              v = col - col_int
              tmp = 0
              for m in range( -1, 3):
                for n in range( -1, 3 ):
                    if (row_int + m ) < 0 or (col_int + n) < 0 or ( row_int + m ) >= img.shape
[0] or (col_int + n) >= img.shape[1]:
                        row_int = img.shape[0] - 1 - m
                        col_int = img.shape[1] - 1 - n
                    numm = img[row_int + m][col_int + n] * Bicubic_Bell(m - u) * Bicubic_Bell
( n - v )
                    tmp += np.abs( np.trunc( numm ) )
            Bicubic_img[i][j] = tmp
    return Bicubic_img
```

```
if __name__ == '__main__':
    img = cv2.imread( 'lena.png',  cv2.IMREAD_COLOR)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    print(img[3][3] )
    height, width, channels = img.shape
    print( height, width )
    bigger_height = height + 200
    bigger_width = width + 200
    print(bigger_height, bigger_width)
    near_img = Nearest( img, bigger_height, bigger_width, channels )
    bilinear_img = Bilinear( img, bigger_height, bigger_width, channels )
    Bicubic_img = Bicubic( img, bigger_height, bigger_width, channels )
    plt.figure()
    plt.subplot( 2, 2, 1 )
    plt.title( 'Source_Image' )
    plt.imshow( img )
    plt.subplot( 2, 2, 2 )
    plt.title( 'Nearest_Image' )
    plt.imshow( near_img )
    plt.subplot( 2, 2, 3 )
    plt.title( 'Bilinear_Image' )
    plt.imshow( bilinear_img )
    plt.subplot( 2, 2, 4 )
    plt.title( 'Bicubic_Image' )
    plt.imshow( Bicubic_img )
    plt.show()
```

运行结果如图 3.57 所示。左上为原始图像，右上为最近邻插值，左下为双线性插值，右下为双立方插值(Bell 分布)。

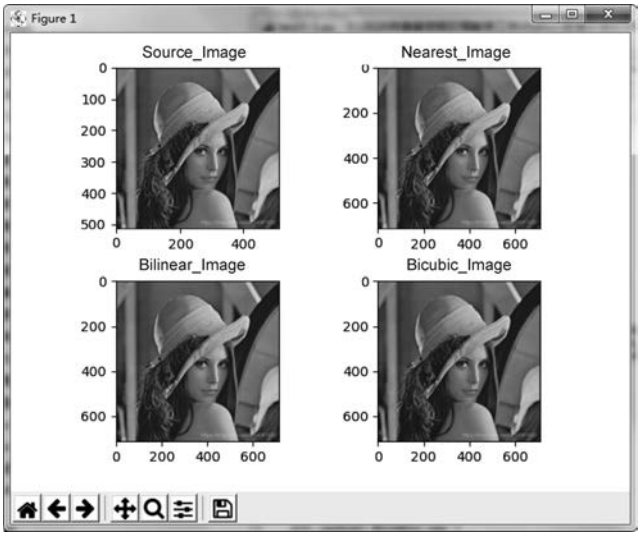


图 3.57　图像的插值效果