第 3 章　直　方　图

直方图被用来以易于解释的可视格式描述图像的统计特性。利用直方图可以方便地确定关于图像的某些特定问题，例如，通过检查直方图可以判断一幅图像曝光是否合适。事实上，由于直方图如此有用，以至于现代数码相机通常会在取景器上提供一个实时的直方图窗口(图 3.1)，以防止拍摄到曝光不足的相片。在图像的拍摄过程中就能发现类似于曝光不足的错误是非常重要的，因为曝光不足会导致信息的丢失，而这些信息是无法通过后期的图像处理技术恢复的。除了在图像拍摄过程中发挥作用，直方图还可以用来帮助改善图像的视觉效果，或者用来确定图像之前经过了哪种类型的处理。

图 3.1　数码相机取景器中的直方图窗口

3.1　何谓直方图

直方图是一种频率分布图，它描述了不同强度值在图像中出现的频率。这个概念可以通过图 3.2 所示的灰度图像来解释，一幅图像 I 的灰度值范围为 $I(u,v)\in[0,K-1]$，它的直方图 h 中正好包含 K 个条目(对于一幅典型的 8 位灰度图像，$K=2^8=256$)，每一个单独的条目被定义为：

$h(i)=I$ 中具有灰度值 i 的像素点总数

其中 $0\leqslant i<K$。正式描述为

$$h(i)=\text{card}\{(u,v)\mid I(u,v)=i\} \tag{3.1}$$

这里，card{…}表示一个集合中元素的数目，即势（参见附录 A）。$h(0)$表示灰度值为 0 的像素点数目，$h(1)$表示灰度值为 1 的像素点数目，依此类推。最后，$h(255)$表示具有最大灰度值 $255=K-1$ 的白色像素点数目。灰度直方图运算的结果是一个长度为 K 的一维向量 h。图 3.3 给出了一个具有 $K=16$ 种可能灰度值的图像示例。

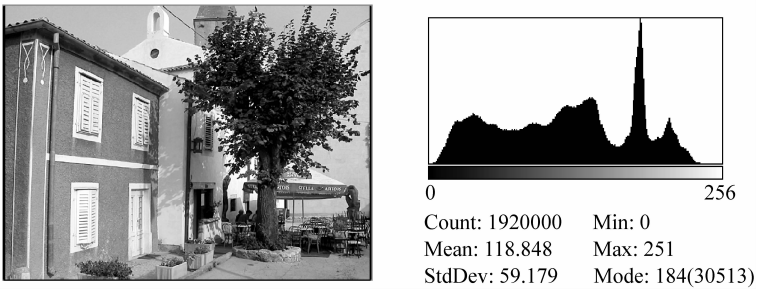


图 3.2　一幅 8 位灰度图像及其直方图，直方图描述了图中 256 种灰度值的频率分布

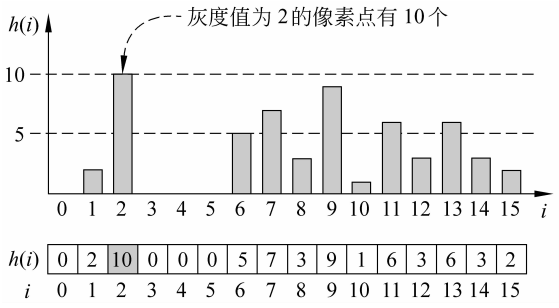


图 3.3　一幅具有 $K=16$ 种可能灰度值的图像的直方图向量。向量中元素的索引 $i=0,\cdots,15$ 代表灰度值，索引 2 的值 10 表示图像中灰度值为 2 的像素点有 10 个

由于直方图并未反映出其每个条目对应于图像中的位置，因此直方图不包含图像中像素点的空间排列信息。这是因为直方图的主要功能是以紧凑的形式反映图像的统计信息（例如强度值的分布）。是否可以仅利用直方图来重建一幅图像呢？也就是说，直方图是否可以以某种方式逆转？在非特殊情况下，答案是否定的。例如，用同样数目的具有特定灰度值的像素点可以建立多种多样的图像，尽管这些图像看上去差别很大，但它们却具有完全相同的直方图（图 3.4）。

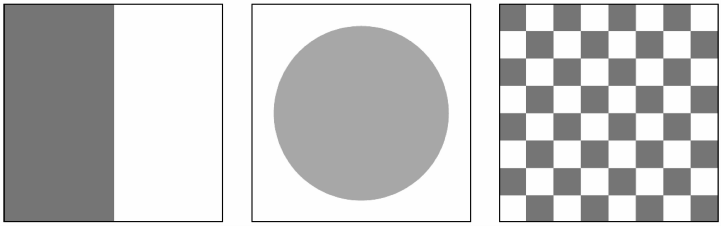


图 3.4　三幅具有相同直方图的不同图像

3.2 理解直方图

直方图能够描述图像获取过程中产生的问题，例如，那些涉及对比度以及动态范围的问题，以及由图像处理过程造成的瑕疵。通过检查直方图分布的范围和均匀程度，可以确定一幅图像是否有效地利用了它的强度范围(图 3.5)。

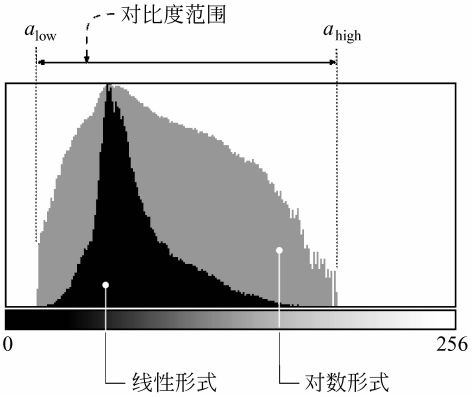


图 3.5 有效强度范围。图表用线性形式(黑色条目)和对数形式(灰色条目)描述了强度值在图像中出现的频率，对数形式使那些出现频率相对较低但可能很重要的部分变得显而易见

3.2.1 图像获取

曝光

直方图使典型的曝光问题变得显而易见。例如，一端较大强度取值范围未使用而另一端充满峰值的直方图(图 3.6)就代表图像曝光不合适。

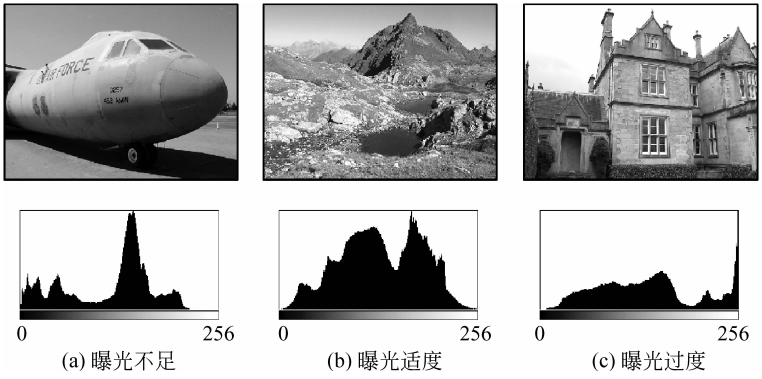
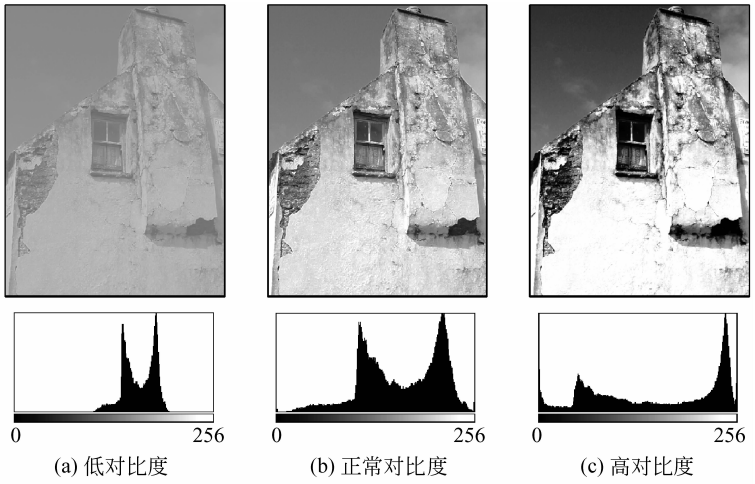


图 3.6 曝光错误在直方图中是显而易见的

对比度

对比度包含两方面的含义，一方面是指一幅给定图像中强度值的有效利用范围，另一方面是指图像中最大和最小像素强度值的差距。一幅全对比度图像有效地利用了全部的可用强度值范围：$a=a_{\min}\cdots a_{\max}=0\cdots K-1$（从黑色到白色）。利用这个定义，一幅图像的对比度可以轻易地从直方图中得到。图 3.7 说明了对比度的变化是怎样影响直方图的。



(a) 低对比度　(b) 正常对比度　(c) 高对比度

图 3.7　对比度变化对直方图的影响

动态范围

一幅图像的动态范围是指图像中不同像素灰度值的数目。理想情况下，即所有取值都被利用的情况下，动态范围是所有可能像素值 K。如果一幅图像的可用对比度范围是 $a=a_{\text{low}}\cdots a_{\text{high}}$，其中

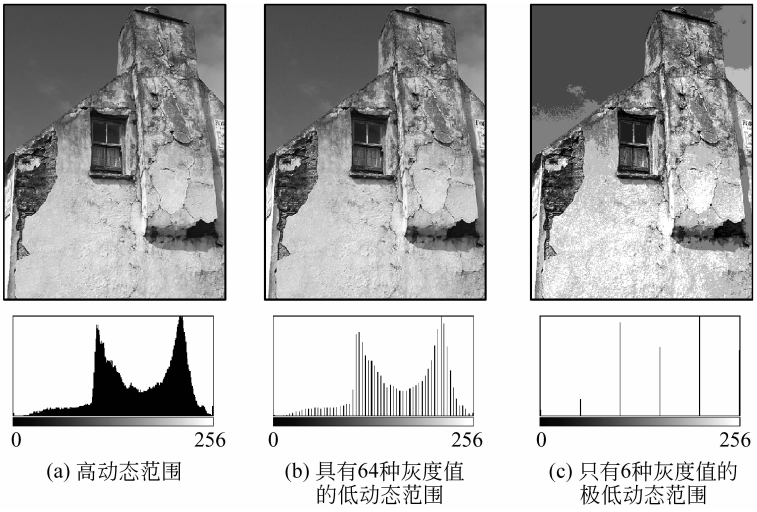
$$a_{\min}<a_{\text{low}}\quad 且\quad a_{\text{high}}<a_{\max}$$

那么动态范围的最大可能值将在这个可用对比度范围内所有强度值都被利用（即在图像中出现，见图 3.8）的情况下达到。

对比度的增强可以通过转换现有的像素值，使更多的可用取值范围被利用来实现；而动态范围的增强只能通过用诸如插值的方法引入人为的（即不是由成像传感器产生的）强度值来实现。高动态范围图像是人们所期望的，因为这些图像的质量在图像处理和压缩过程中损失会比较小。由于没有实际可行的方法能够在获取图像之后增强其动态范围，专业的摄像机和扫描仪一般工作在超过 8 位（通常是 12 位或 14 位）的深度下，以便在图像获取阶段增强其动态范围。这样做是为了方便后期的图像处理，因为大部分用于显示图像的输出设备（例如显示器和打印机）都无法产生多于 256 种色阶。

3.2.2　图像缺陷

直方图可用于检测图像获取或者后期处理过程中产生的各种缺陷。因为直方图依赖于图像中所捕获场景的视觉特征，所以不存在一种通用的“理想”直方图。一个特定场景的最优直



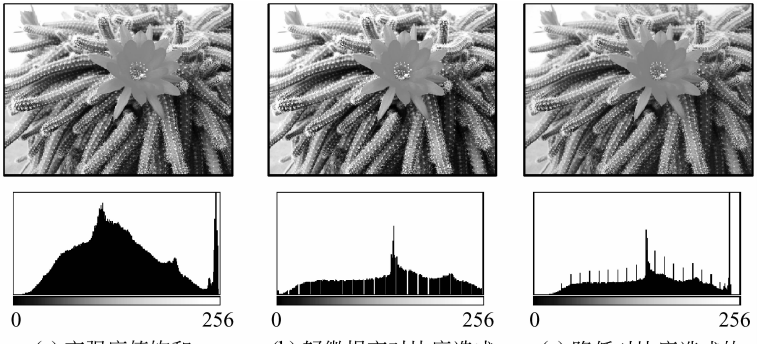
(a) 高动态范围　(b) 具有64种灰度值的低动态范围　(c) 只有6种灰度值的极低动态范围

图 3.8　动态范围变化对直方图的影响

方图，在另一个场景中可能会变得完全无法接受。例如，一幅天文图像的理想直方图可能会与一幅风景或肖像照片的理想直方图相去甚远。然而，一些一般性的准则还是存在的，例如，用一部数码相机拍摄一幅风景图时，可以期望直方图的分布均匀平坦且没有孤立峰值。

饱和度

理想状态下，诸如摄像机中传感器的对比度范围应该比它从场景中接收到的光线强度范围大。在这种情况下，所拍摄图像的直方图的两端会比较平坦，因为从场景中特别亮和特别暗的部分接收到的光线会比从其他部分接收到的光线少。不幸的是，实际的情况通常不是这种理想状态，光照往往超出了传感器的对比度范围，场景中特别亮和特别暗的部分无法被捕获，从而造成直方图的一端或两端饱和。那些超出传感器对比度范围的光照值将被映射到传感器的最小值或最大值，这种现象反映在直方图上就是端部出现显著的峰值。这种情况通常出现在曝光不足或者曝光过度的图像中，而且当场景的实际对比度范围超出系统传感器的对比度范围时，这种情况一般是无法避免的(参见图 3.9(a))。



(a) 高强度值饱和　(b) 轻微提高对比度造成的直方图中的间隙　(c) 降低对比度造成的直方图中的峰值

图 3.9　图像拍摄过程中的失误对直方图的影响

峰值和间隙

像上面讨论的那样，一幅未经处理图像的强度值分布一般是平坦的，也就是说，在它的直方图中不应该出现孤立的峰值(除了两端可能出现的饱和效果)和间隙。任何给定强度值出现的频率也不应该与它邻近亮度值出现的频率有过大差别(即局部平滑)。原始图像中出现这样的不自然现象是非常罕见的，而在一幅图像被操作处理后(例如改变对比度)，这些问题就会经常出现。增强对比度(参见第 4 章)会造成直方图中的条目彼此分散，由于取值的离散，直方图中将会产生间隙(图 3.9(b))。同样由于取值的离散，降低图像的对比度将会造成直方图中本来不同的条目合并，这将导致直方图中条目的值增大，并最终导致直方图中出现显著的峰值(图 3.9(c))[①]。

图像压缩的影响

图像压缩也会改变图像并在图像的直方图中反映出来。例如，在 GIF 压缩过程中，一幅图像的动态范围被削减到只有少量的强度或颜色值，从而造成直方图具有明显的线状结构，这种现象是无法在后续处理中被消除的(图 3.10)。一般来说，即使后来把图像转换成 TIFF 或 JPEG 那样的全彩色图像，利用直方图也能够快速发现图像是否经过了像 GIF 转换中那样的颜色量化过程。

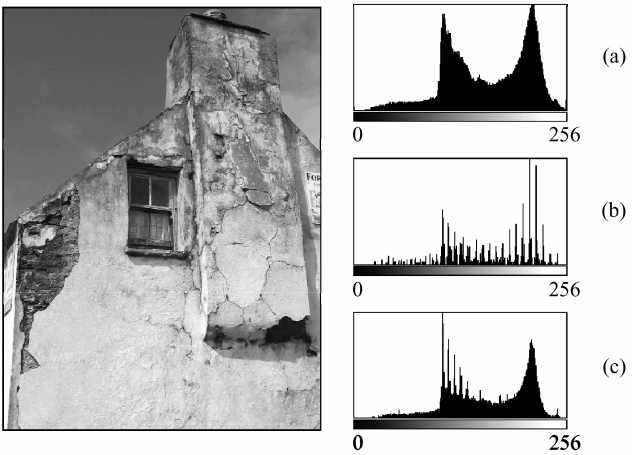


图 3.10　GIF 转换造成的颜色量化效应。原始图像被转换为一幅 256 色的 GIF 图像(左)，原始图像的直方图为(a)；经过 GIF 转换后的直方图为(b)；当 RGB 图像的尺寸改变 50%时，一些丢失的颜色因为插值重新产生，但是 GIF 转换造成的影响在直方图中仍然清晰可见(c)

图 3.11 说明，当一幅只有两个灰度值(128,255)的线框图形，受到不是为线框图形设计的而是为细节化的全彩色图像设计的压缩方法(例如 JPEG 方法)的影响时，会产生怎样的结果。所得图像的直方图明显显示，它包含大量没有在原始图像中出现的灰度值，导致所得

① 不幸的是，这些类型的错误也可能是由某些图像采集设备内部的对比度优化程序造成的，特别是低端扫描仪。

图像朦胧、模糊，质量变差[①]。

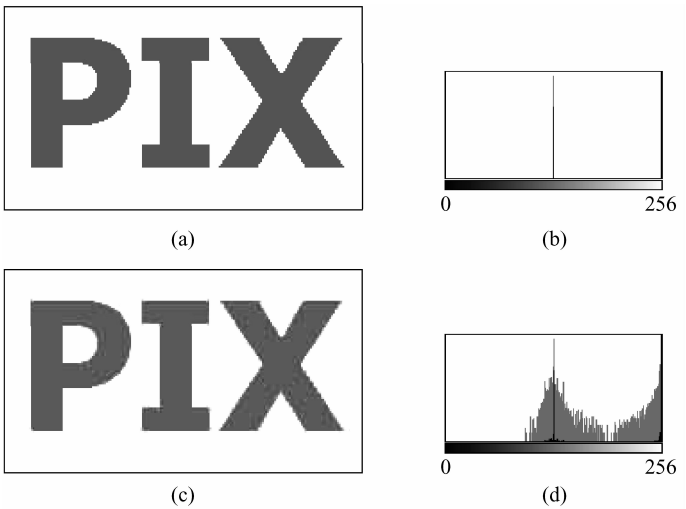


图 3.11 JPEG 压缩的影响。原始图像(a)只包含两个不同的灰度值，如同其直方图(b)中显示的那样。对于这种类型的图像，JPEG 压缩是一个错误的选择，其导致结果中出现大量其他的灰度值，在所得图像(c)以及其直方图(d)中均可以观察到。在两个直方图中，显示了线性频率(黑色条)以及对数频率(灰色条)

3.3 直方图计算

计算包含强度值从 0～255 的 8 位灰度图像的直方图是一件很容易的工作，所需要的是 256 个计数器，每一个可能的强度值对应一个计数器。首先，所有的计数器初始化为 0；然后，遍历图像 $I(u,v)$，确定每一个位置的像素值 p，对应的计数器增加 1；最后，在每一个计数器中将包含相应强度值在图像中的像素数目。

有 K 个可能强度值的图像需要 K 个计数器变量，例如，由于 8 位灰度图像至多包含 256 个不同的强度值，因此需要 256 个计数器。然而，独立的计数器只是在概念上讲得通，实际实现时不会用 K 个独立变量来表示计数器，而是利用具有 K 个元素的数组(在 Java 中为 int[256])。在此例中，实际实现为一数组是简单易懂的。由于强度值从 0 开始(类似 Java 中的数组)并且都是正数，所以其可以直接作为直方图数组中的下标 $i \in [0, N-1]$。程序 3.1 是利用 ImageJ 插件中的方法 run()计算直方图的完整 Java 代码。

程序 3.1 计算 8 位灰度图像直方图的 ImageJ 插件。方法 setup()返回 DOES_8G＋NO_CHANGES(第 4 行)，此插件要求 8 位灰度图像并且不会改变图像。在 Java 中，所有新建的实例化数组元素(第 8 行)都自动初始化，这里初始化为 0

① 由于不是专门设计，所以对这样的图像进行 JPEG 压缩是导致最糟糕的影像误差的原因之一。JPEG 是为颜色过渡平滑的自然场景照片而设计的，用它对具有大面积相同颜色的图标图像进行压缩会导致很严重的视觉瑕疵(例如图 2.9)。

```
 1  public class Compute_Histogram implements PlugInFilter {
 2
 3      public int setup(String arg,ImagePlus img) {
 4          return DOES_8G+NO_CHANGES;
 5      }
 6
 7      public void run(ImageProcessor ip) {
 8          int[] H=new int[256];            //直方图数组
 9          int w=ip.getWidth();
10          int h=ip.getHeight();
11
12          for (int v=0;v<h;v++) {
13              for (int u=0;u<w;u++) {
14                  int i=ip.getPixel(u,v);
15                  H[i]=H[i]+1;
16              }
17          }
18          ...                              //可以在这里使用直方图 H[]
19      }
20
21  } //Compute_Histogram 类的结尾
```

在程序 3.1 开始处，创建了类型为 int[]的数组 H，其元素自动初始化[①]为 0。此数组是沿着行的顺序还是列的顺序遍历是没有什么区别的，至少对于最终的结果来说，只要把图像中的所有像素刚好遍历一次即可。与程序 2.1 相反，此例中我们把数组按照标准的行优先顺序来回移动，因此外层 for 循环遍历垂直坐标 v，内层循环遍历水平坐标 u[②]。直方图计算出来之后，就可以进行诸如显示等进一步的处理。

在 ImageJ 中已经实现了直方图计算，可以通过 ImageProcessor 类的对象类方法 getHistogram()来使用它。如果使用 ImageJ 中提供的方法，程序 3.1 中的方法 run()可以简化为如下所示：

```
public void run(ImageProcessor ip) {
    int[] H=ip.getHistogram();  内置的 ImageJ 方法
    ... //可以在这里使用直方图 H[]
}
```

① 在 Java 中，基本数组例如 int、double 等整数类型在创建的时候初始化为 0，浮点类型初始化为 0.0，对象类数组初始化为 null。

② 在此方式中，图像元素的遍历方式与存储在内存中的方式是严格一致的，因此内存获取更加方便并且有可能提高性能，处理较大图像时尤其如此(参见附录 B)。

3.4 多于 8 位图像的直方图

通常，计算直方图是为了能在屏幕上显示图像的颜色分布。在处理有 $2^8=256$ 个灰度等级的图像时不存在什么问题，但是当一幅图像有更多数值，例如 16 和 32 位或者浮点图像（见表 1.1），对于如此大数目的灰度等级，如果不事先处理而直接在屏幕上显示是不现实的。

3.4.1 像素组合

因为不可能利用直方图的条目来表示每一个强度值，所以这里将利用直方图的一个单独条目来表示一系列强度值。此技术通常称为“装箱”，因为可以直观地把它看作是将一组像素值装进一个诸如箱柜或者桶之类的容器中。在一个大小为 B 的装箱直方图中，每一个箱 $h(j)$ 包含取值在区间 $a_j \leqslant a < a_{j+1}$ 内的像素的数目，因此（类比式(3.1)），

$$h(j) = \mathrm{card}\{(u,v) \mid a_j \leqslant I(u,v) < a_{j+1}\}, \quad \text{当 } 0 \leqslant j < B \tag{3.2}$$

一般地，可能的取值范围根据箱的个数 B 被分成相等大小为 $k_B = K/B$ 的箱，因此对应第 j 个箱的区间的起始值为

$$a_j = j \cdot \frac{K}{B} = j \cdot k_B$$

3.4.2 示例

为了对 14 位图像创建一个包含 $B=256$ 个条目的典型直方图，需要把 $j=0\cdots 2^{14}-1$ 分成 256 个相等的区间，每个区间的长度为 $k_B = 2^{14}/256 = 64$，因此 $a_0=0, a_1=64, a_2=128, \cdots, a_{255}=16320$ 以及 $a_{256}=a_B=2^{14}=16384=K$。下列结果为所得像素值到直方图箱 $\mathbf{h}(0)\cdots\mathbf{h}(255)$ 的映射：

$$\begin{array}{lcccccc}
\mathbf{h}(0) & \leftarrow & 0 & \leqslant & I(u,v) & < & 64 \\
\mathbf{h}(1) & \leftarrow & 64 & \leqslant & I(u,v) & < & 128 \\
\mathbf{h}(2) & \leftarrow & 128 & \leqslant & I(u,v) & < & 192 \\
\vdots & & \vdots & & \vdots & & \vdots \\
\mathbf{h}(j) & \leftarrow & a_j & \leqslant & I(u,v) & < & a_{j+1} \\
\vdots & & \vdots & & \vdots & & \vdots \\
\mathbf{h}(255) & \leftarrow & 16320 & \leqslant & I(u,v) & < & 16384
\end{array}$$

3.4.3 实现

如果如例中所述，值域范围 $0,\cdots,K-1$ 分成相等的长度间隔为 $k_B=K/B$ 的区域，那么就不需要利用转换表来寻找 a_j，因为给定一个像素值 $a=I(u,v)$ 很容易计算出正确的直方图元素 j。在这种情况下，把像素值 $I(u,v)$ 简单地分为长度为 k_B 的区域，即

$$\frac{I(u,v)}{k_B} = \frac{I(u,v)}{K/B} = \frac{I(u,v)\cdot B}{K} \tag{3.3}$$

这里需要一个整数值来作为直方图箱 $h(j)$ 的索引，即

$$j = \left\lfloor \frac{I(u,v) \cdot B}{K} \right\rfloor \tag{3.4}$$

其中 $\lfloor \cdot \rfloor$ 表示 floor 函数[①]。程序 3.2 给出了通过“线性装箱”计算直方图的 Java 方法。注意，式(3.4)中的所有计算都是整数运算而不包含任何浮点运算。另外，也不需要显式调用 floor 函数，因为在第 11 行的表达式

```
a * B/K
```

利用了整数除法以及 Java 对此类运算的浮点结果进行截断处理的特点(即浮点数结果简单切断)，这样得到的结果等同于 floor 函数[②]。按同样的方式，这种装箱方法也可用于浮点图像。

程序 3.2　利用“装箱”计算直方图(Java 方法)。示例中计算 $K=256$ 强度等级的 8 位灰度图像，当 $B=32$ 时得到的直方图。成员函数 binnerHistogram() 的返回值为图像对象 ip 所传递的大小为 B 的 int 数组直方图

```
 1  int[] binnedHistogram(ImageProcessor ip) {
 2      int K=256;                      //亮度值的数目
 3      int B=32;                       //需要定义的直方图尺寸
 4      int[] H=new int[B];             //直方图数组
 5      int w=ip.getWidth();
 6      int h=ip.getHeight();
 7
 8      for (int v=0;v<h;v++) {
 9          for (int u=0;u<w;u++) {
10              int a=ip.getPixel(u,v);
11              int i=a * B/K;          //只是整型运算!
12              H[i]=H[i]+1;
13          }
14      }
15      //返回装箱直方图
16      return H;
17  }
```

3.5　彩色图像直方图

彩色图像的直方图，通常是指图像强度(亮度)直方图或者单个颜色通道的直方图。这两个变体都依赖于实际的图像处理应用，可用于图像质量的主观评估，尤其是在图像获取之后立即的主观评估。

---

① 表示对 x 向下取最接近的整数值(见附录 A)。

② 更详细的讨论见附录 B 关于 Java 中整数除法部分的内容。

3.5.1 强度直方图

彩色图像的亮度直方图 $\mathbf{h}_{\text{Lum}}$ 就是相应灰度图像的直方图，因此先前关于直方图的所有方面的讨论也适用于这种类型的直方图。灰度图像是通过计算彩色图像各个通道的亮度来得到的。当计算亮度时，只是简单地求每个颜色通道的平均值是不够的，应该计算考虑了颜色感知理论的加权和。此处理过程将在第 8 章详细介绍。

3.5.2 单个颜色通道直方图

尽管亮度直方图考虑了所有颜色通道，但在单个通道中的图像错误仍然难以被发现。例如，当一种颜色通道过饱和时，亮度直方图可能依然是平坦均匀的。在 RGB 图像中，蓝色通道对总体亮度贡献较小，因此对此类问题尤其敏感。

分量直方图对每个颜色通道中的强度分布进行统计分析。计算分量直方图时，把每个颜色通道当作一个与其他通道相互独立的灰度图像，分别计算和显示其直方图。图 3.12 中显示了典型的 RGB 彩色图像的亮度直方图 $\mathbf{h}_{\text{Lum}}$ 及三个分量直方图 $\mathbf{h}_{\text{R}}$、$\mathbf{h}_{\text{G}}$ 和 $\mathbf{h}_{\text{B}}$。注意，所有三个通道中的饱和问题（红色分量分布在较高亮度区域，绿色和蓝色分量分布在较低亮度区域）在分量直方图中比较明显，而在亮度直方图中不明显。

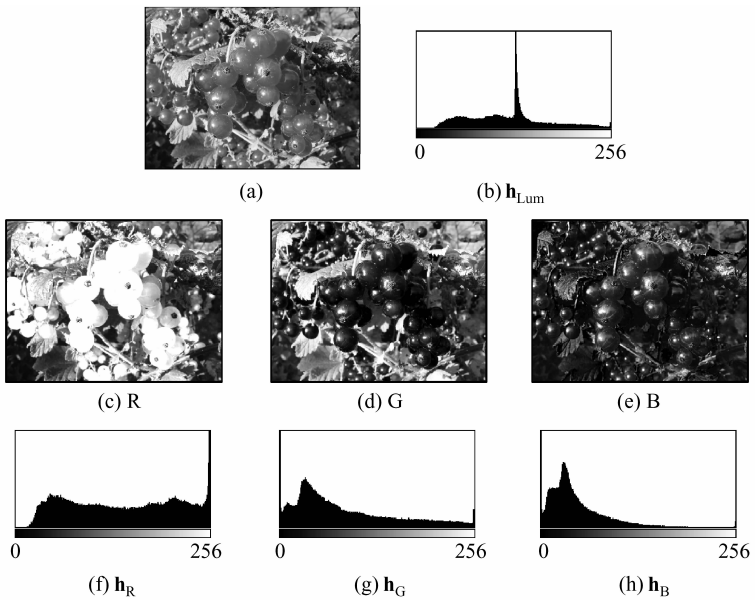


图 3.12 RGB 彩色图像的直方图：(a)原始图像；(b)亮度直方图 $\mathbf{h}_{\text{Lum}}$；(c～e)RGB 颜色分量作为强度图像及对应的分量直方图；(f～h)$\mathbf{h}_{\text{R}}$、$\mathbf{h}_{\text{G}}$ 和 $\mathbf{h}_{\text{B}}$。事实上，所有三个颜色通道都有饱和问题，只是在各个分量直方图中表现的比较明显。由此导致的亮度分布中的尖锐峰值可以在亮度直方图的中部观察到

此例中，三个分量直方图显得完全不同于相应的亮度直方图 $\mathbf{h}_{\text{Lum}}$，这种情况也是比较典型的（图 3.12(b)）。

3.5.3　组合颜色直方图

亮度直方图以及分量直方图都提供关于单独颜色分量光照、对比度、动态范围以及饱和效应的有用信息。切记：这两种直方图并没有提供关于图像中真实颜色分布的信息，因为它们是基于单个颜色通道的，而不是基于形成各个像素颜色的通道的组合。例如，红色通道的分量直方图 $\mathbf{h}_R$ 包含条目为

$$\mathbf{h}_R(200) = 24$$

那么只知道图像中有 24 个红色分量值为 200 的像素。这个条目并没有告诉关于那些绿色和蓝色像素的信息，因此可以为任意有效值，即

$$(r,g,b) = (200, *, *)$$

进一步假设三个分量直方图包含下列条目：

$$\mathbf{h}_R(50) = 100, \quad \mathbf{h}_G(50) = 100, \quad \mathbf{h}_B(50) = 100$$

是否可以从中得到结论：图像中有 100 个像素包含颜色组合

$$(r,g,b) = (50,50,50)$$

或者这个颜色完全存在？一般来说，答案是否定的。因为，没办法从这些数据里确定是否在图像中存在一个三个颜色成分都为 50 的像素点。这里能确定的唯一一点是，在此图像中，颜色值(50,50,50)至多出现 100 次。

因此，虽然彩色图像的常规直方图描述了图像的重要性质，但是它们并没有真正提供任何关于图像中实际颜色成分的信息。事实上，一组彩色图像可以有非常相似的分量直方图，但它们却有完全不同的颜色。这个问题引出了组合直方图这个非常有趣的题目，即利用组合颜色组件的统计信息，试图确定两幅图像的颜色组成是否大致相似。由这些类型的直方图计算得到的特征，通常形成了基于颜色的图像检索方法的基础。在第 8 章中将返回来讨论这个主题，从而更详细地研究彩色图像。

3.6　累积直方图

累积直方图源自于普通直方图，它在处理一些涉及直方图的操作时非常有用，如直方图均衡化(见 4.5 节)。累积直方图 $\mathbf{H}(i)$定义为：

$$\mathbf{H}(i) = \sum_{j=0}^{i} \mathbf{h}(j), \quad \text{其中 } 0 \leqslant i < K \tag{3.5}$$

因此，$\mathbf{H}(i)$的值是所有普通直方图中 $\mathbf{h}(j)$值的和，其中 $j \leqslant i$。又或者，可以将其定义为递归形式(如程序 4.2 所实现的)：

$$\mathbf{H}(i) = \begin{cases} \mathbf{h}(0) & i = 0 \\ \mathbf{H}(i-1) + \mathbf{h}(i) & 0 < i < K \end{cases} \tag{3.6}$$

累积直方图是单调递增函数，其最大值为：

$$\mathbf{H}(K-1) = \sum_{j=0}^{K-1} \mathbf{h}(j) = M \cdot N \tag{3.7}$$

即为在宽度 M、高度 N 的图像中的所有像素数。图 3.13 显示了累积直方图的一个具

体示例。

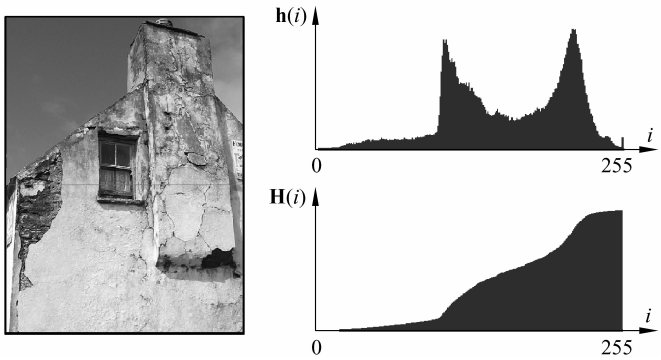


图 3.13 普通直方图 $\mathbf{h}(j)$及其对应的累积直方图 $\mathbf{H}(i)$

3.7 练习题

3.1 在程序 3.2 中,B 和 K 都是常数。考虑一下在循环外部计算 B/K 的值是否有优点,并解释理由。

3.2 开发一个计算 8 位灰度图像的累积直方图的 ImageJ 插件,并将此累积直方图当作一幅新的图像显示出来,类似于图 3.13 中的 $\mathbf{H}(i)$。

提示:利用 ImageProcessor 的方法 int[] getHistogram()来获取原始图像的直方图,并且根据式(3.6)计算对应位置的累积直方图。创建一个新的(黑色)合适尺寸的图像(例如 256×150),然后用黑色垂直条描绘出缩放后的直方图数据,使得最大的条目跨越图像的整个高度。程序 3.3 显示了此插件如何初始化以及如何创建和显示一副新图像。

3.3 利用区间极限为 a_j 的表(式(3.2)),开发一种非线性装箱技术。

3.4 利用 Java 方法 Math.random()或者 Random.nextInt(int n)开发一个 ImageJ 插件,用来创建一幅图像,其像素随机取值但在[0,255]间均匀分布。分析此图像的直方图以确定实际像素值均匀分布的程度。

3.5 开发一个 ImageJ 插件,用于创建一幅具有高斯分布的随机图像,其中高斯分布的均值为 $\mu=128$,标准方差为 $\sigma=50$。利用标准 Java 方法 double Random.nextGaussian()来生成正态分布随机数值(其中 $\mu=0,\sigma=1$),然后对其适度比例缩放后作为像素值。分析所得图像的直方图,看是否为高斯分布。

程序 3.3 创建和显示一幅新图像(ImageJ 插件)。首先创建 ByteProcessor 对象(histIp,第 15 行),随后初始化该对象。在此处,histIp 没有屏幕显示内容,因此不可见。然后,创建相关联的 ImagePlus 对象(第 25 行),并且利用 show()方法显示它(第 26 行)。注意标题(String)是如何从 setup()方法内部的原始图像中获得,然后用于组成新的图像标题(第 24 行和第 25 行)。如果在调用 show()之后,histIp 发生了变化,可以调用 updataAndDraw()方法以重新显示相应结果(第 27 行)

```
public class Create_New_Image implements PlugInFilter {
    String title=null;

    public int setup(String arg,ImagePlus im) {
        title=im.getTitle();
        return DOES_8G+NO_CHANGES;
    }

    public void run(ImageProcessor ip) {
        int w=256;
        int h=100;
        int[] hist=ip.getHistogram();

        //创建直方图图像
        ImageProcessor histIp=new ByteProcessor(w,h);
        histIp.setValue(255);    //white=255
        histIp.fill();           //清除图像

        //在 ip2 处绘制黑条为对应直方图值
        //例如使用 histIp.putpixel(u,v,0)
        //...

        //显示直方图
        String hTitle="Histogram of "+title;
        ImagePlus histIm=new ImagePlus(hTitle,histIp);
        histIm.show();
        //histIm.updateAndDraw();
    }

} //类 Create_New_Image 结束
```

第4章 点 运 算

点运算是在不改变图像大小、几何形状以及局部结构的情况下，对像素值进行修改。每一个新的像素值 $a'=I'(u,v)$ 完全依赖于相同位置的前一个值 $a=I(u,v)$，而与其他位置像素值无关，尤其与其相邻的任何像素无关[①]。原始像素值通过函数 $f(a)$ 映射到一个新的值，

$$a' \leftarrow f(a) \quad \text{或} \quad I'(u,v) \leftarrow f(I(u,v)) \tag{4.1}$$

其中 (u,v) 表示像素位置。如果函数 $f()$ 与图像坐标无关（也就是说在图像上各点取值相同），则此运算称为“均匀运算”，均匀点运算的典型示例包括如下。

- 改变图像的亮度和对比度。
- 进行任意的强度变换（“曲线”）。
- 量化（或“多色调分色”）图像。
- 全局阈值转换。
- Gamma 校正。
- 颜色转换。

后文将对上述一些方法进行更加详细的介绍。相比之下，非均匀点运算的映射函数 $g()$ 需要考虑当前图像坐标 (u,v)，例如

$$a' \leftarrow g(a,u,v) \quad \text{或} \quad I'(u,v) \leftarrow g(I(u,v),u,v) \tag{4.2}$$

一个典型的非均匀点运算是进行局部的对比度和亮度调整，用于补偿在图像获取时非均匀光照的影响。

4.1 图像强度修正

4.1.1 对比度与亮度

这里从一个简单的示例开始。要增加图像 50% 的对比度（即因子为 1.5）和提高 10 个单位的亮度，可以分别表示为映射函数

$$f_{\text{contr}}(a) = a \times 1.5 \quad \text{和} \quad f_{\text{bright}}(a) = a + 10 \tag{4.3}$$

第一个运算可以通过程序 4.1 中的代码作为 ImageJ 的插件来实现，对此代码做简单修改可以适用于任何其他形式的点运算。在第 7 行强制类型转换造成截断效应之前，加上

① 若结果依赖于多个像素值，则此运算称为“滤波”，如第 5 章所述。

0.5 可以非常简单地实现四舍五入到最近整数值(只针对正数)。此外要注意,在本例中使用了更高效的图像处理方法 get()和 set()(代替 getPixel()和 putPixel())。

程序 4.1　运用点运算增加 50%的对比度(ImageJ 插件)。注意在第 7 行整型像素值乘以常数 1.5(隐含为 double 型数据)后的结果是 double 类型。因此,需要一个显式的类型转换(int)将其值转换为整型赋给变量 a。在第 7 行所加的 0.5 是要把数据四舍五入到最近的整数值

```
 1  public void run(ImageProcessor ip) {
 2      int w=ip.getWidth();
 3      int h=ip.getHeight();
 4
 5      for (int v=0;v<h;v++) {
 6          for (int u=0;u<w;u++) {
 7              int a=(int) (ip.get(u,v)1.5+0.5);
 8              if (a>255)
 9              a=255        //限定到最大值
10              ip.set(u,v,a);
11          }
12      }
13  }
```

4.1.2　利用设定门限限制结果值

必须牢记,对像素值进行算术运算时,计算结果有可能会超出给定图像类型像素值的最大取值范围(8 位灰度图像的范围为[0…255])。为了避免这个问题,在程序 4.1 的第 9 行中,包含一个“设定门限”语句:

```
if (a>255) a=255;
```

此操作限制任意结果的最大值为 255。同样,可以限制最小值为 0,从而避免出现负的像素值(在 8 位图像中不存在负值),语句如下所示:

```
if (a<0) a=0;
```

但是,这条语句在程序 4.1 中是不需要的,因为在这个操作中,所得中间结果肯定不会是负值。

4.1.3　图像求反

对一幅灰度图像求反是一个简单的点运算过程,即反转图像的像素值顺序(通过乘以−1)并且加上一个常数值,使其映射到允许的取值范围内。因此,对于一个在取值范围$[0,a_{\max}]$内的像素值 $a=I(u,v)$,相应的点运算为:

$$f_{\text{invert}}(a) = -a + a_{\max} = a_{\max} - a \tag{4.4}$$

在 2.2.4 节(程序 2.1)的第一个插件示例中,列出了对 8 位灰度图像 $a_{\max}=255$ 进行求

反运算的操作。注意，在这种情况下，不需要任何门限限制，因为此函数总是映射到原始值域范围内。在 ImageJ 中，这个操作是通过 invert()方法来实现的(针对 ImageProcessor 类型的对象)，可以通过 Edit→Invert 菜单来使用它。从图 4.5(c)中可以看出，图像求反后的直方图与原图是镜像对称的。

4.1.4 阈值操作

图像的阈值操作是一种特殊的量化方法，它根据给定的阈值 a_{th} 将像素值分为两类。阈值函数 $f_{\text{threshold}}(a)$ 把所有像素值映射为两个固定强度值 a_0 或 a_1 中的一个，即

$$f_{\text{threshold}}(a)=\begin{cases}a_0 & a<a_{\text{th}}\\ a_1 & a\geqslant a_{\text{th}}\end{cases} \tag{4.5}$$

其中 $0<a_{\text{th}}<a_{\max}$。此方法广泛用于灰度图像的二值化，其值为 $a_0=0, a_1=1$。

ImageJ 为二值图像提供了一种特殊的数据类型(BinaryProcessor)，但实际上，这些二值图像实现为 8 位灰度图像(类似一般的灰度图像)，像素取值为 0 和 255。ImageJ 通过 ImapgProcessor 的方法 threshold(int level)来实现阈值操作，其中 level≡a_{th}；此操作可通过 Image→Adjust→Threshold 菜单调用(见图 4.1)。阈值操作通过将强度值分布分裂和合并到 a_0 和 a_1 处的两个条目来影响直方图的，如图 4.2 所示。

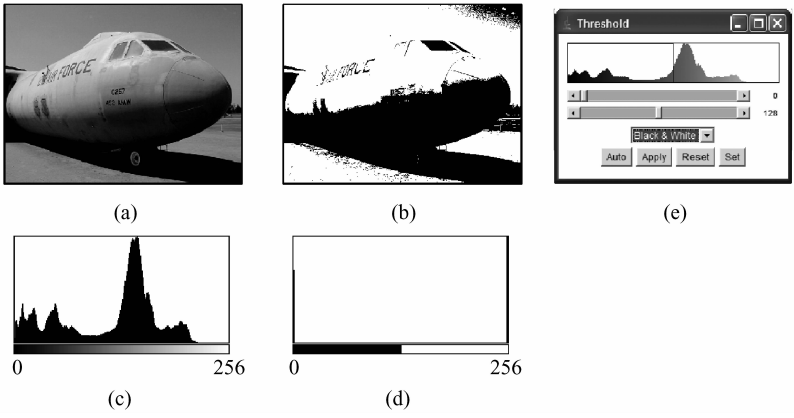


图 4.1 阈值操作：原始图像(a)及对应的直方图(c)；$a_{\text{th}}=128, a_0=0, a_1=255$ 时阈值操作后的结果(b)及对应直方图(d)；(e)为 ImageJ 的交互 Threshold 菜单

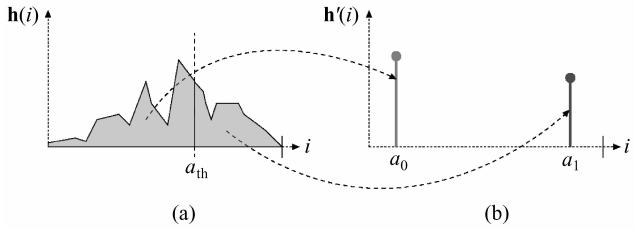


图 4.2 阈值操作对直方图的影响。阈值为 a_{th}。原始分布(a)，在位置 a_0 和 a_1 处被分裂和合并为有两个孤立条目后的直方图(b)

4.2　点运算与直方图

我们已经看到，在某种情况下，点运算对图像直方图的影响是非常容易预测的。例如，通过一个常量来增加图像的亮度将使直方图整体向右移动；提高对比度以加宽直方图；对图像求反使直方图反转。虽然这些操作非常简单，但是对于分析点运算与直方图之间的密切关系是非常有价值的。

如图 4.3 所示，直方图中位置 i 对应的条目（条状）映射到一个（大小为 $\mathbf{h}(i)$）包含所有像素值为 i 的像素集[①]。如果经过某个点运算，使得某个特定的直方图条目发生了位移，那么对应集合中的所有像素值都会发生相同的变化，反之亦然。如果一个点运算（如降低图像对比度）使得两个先前分开的直方图条目合并到位置 i 处，那么图像会发生什么变化？此问题的答案是相应像素集合被合并，新直方图的条目是两个（或更多）条目的总和（即两个合并集合的大小）。此时，合并后集合中的元素是不可区分（或分离）的，因此这个操作可能导致（或许是无意的）不可逆的动态范围减少，从而永久性地丢失图像中的信息。

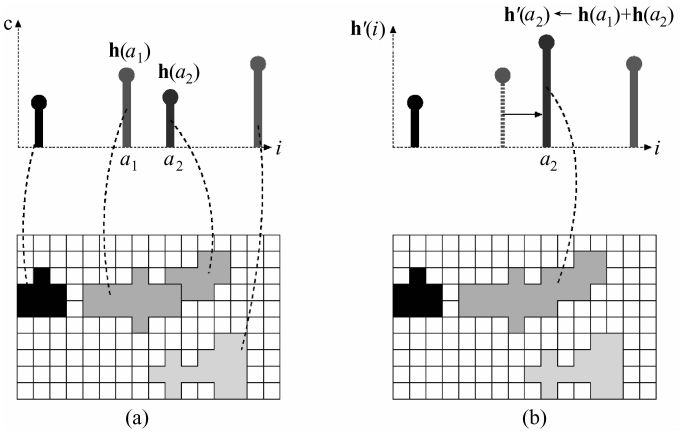


图 4.3　直方图条目映射到具有相同像素值的像素集合。(a)如果经过点运算，直方图的某一条目移动到新的位置，那么在相应集合中所有像素点的值都会发生相同的改变。(b)如果由于点运算使得两个直方图条目 $\mathbf{h}(a_1)$ 和 $\mathbf{h}(a_2)$ 碰巧有相同的索引值，那么对应的像素集合合并，并且对应的所有像素值不可分

4.3　自动对比度调整

自动对比度调整（auto-contrast）是一种点运算，其作用是改变像素值使得结果覆盖像

① 当然这只对每个强度值对应一个条目的普通直方图来说是正确的。如果使用了装箱技术，则每个直方图条目映射到一个特定范围的像素值。

素值的全部取值范围。此算法分别把当前最暗和最亮的像素值映射到可取灰度范围中最小和最大值，然后使中间值线性分布。

我们假设 a_{low} 和 a_{high} 是在当前图像中的最小和最大的像素值，且图像的整个强度取值范围为 $[a_{min}, a_{max}]$。要把此图像延伸到整个强度范围(图 4.4)，先把最小的像素值 a_{low} 映射为 0，然后根据因子 $(a_{max}-a_{min})/(a_{high}-a_{low})$ 增加对比度，最后加上 a_{min} 使得结果落到目标范围。自动对比度操作的映射函数可以定义为：

$$f_{ac}(a) = a_{min} + (a - a_{low}) \times \frac{a_{max} - a_{min}}{a_{high} - a_{low}} \tag{4.6}$$

其中 $a_{high} \neq a_{low}$，即图像至少包含两个不同的像素值。对于 $a_{min}=0$，$a_{max}=255$ 的 8 位图像，式(4.6)的函数可以简化为：

$$f_{ac}(a) = (a - a_{low}) \times \frac{255}{a_{high} - a_{low}} \tag{4.7}$$

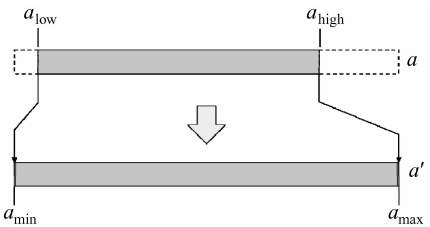


图 4.4 根据式(4.6)进行自动对比度调整。原始像素值 a 范围在 $[a_{low}, a_{high}]$，通过线性映射到目标范围 $[a_{min}, a_{max}]$

目标范围 $[a_{min}, a_{max}]$ 不一定是值域的最大取值范围，但应该是图像可以映射到的任何值域区间。当然，此方法也可应用于较小范围来减小图像对比度。图 4.5(b)中显示了自动对比度调整对相应直方图的影响，在灰度范围上做的线性拉伸，将导致在新的分布中出现了规则的分布间隙。

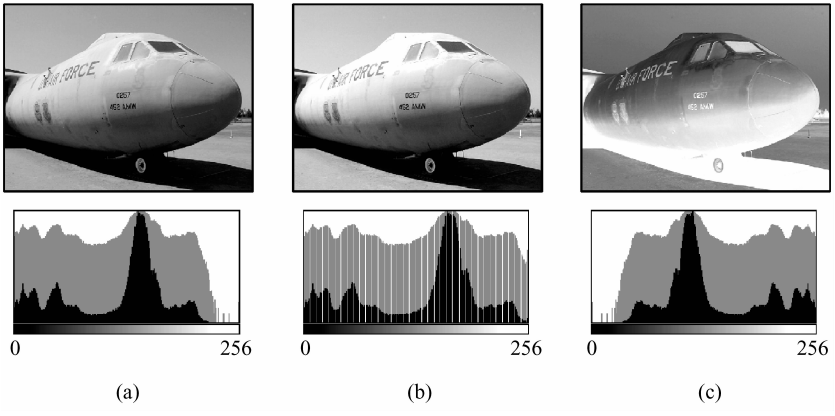


图 4.5 自动对比度调整的效果以及对结果直方图求反运算的效果。(a)原始图像；(b)自动对比度调整结果；(c)求反运算。显示的直方图条目是线性的(黑条)和对数化的(灰条)

4.4　修正的自动对比度调整

在实际中，式(4.6)中的映射函数只受到几个极值(小或大)的强烈影响，而这几个极限像素值未必能代表图像的主要内容。这个问题在很大程度上可以通过在目标灰度范围的上限和下限区域“充满”固定百分比(q_{low}，q_{high})的像素的方法得以避免。为了达到此目的，要确定两个限定值 a'_{low} 和 a'_{high}，使得图像 I 中所有像素值的预定义分位数 q'_{low} 小于 a'_{low}，同时这些值的预定义分位数 q_{high} 大于 a'_{high}(图 4.6)。值 a'_{low} 和 a'_{high} 依赖于图像的内容，可以通过图像的累计直方图 $\mathbf{H}(i)$ 求得(见 4.6 节)。

$$a'_{\text{low}} = \min\{i \mid \mathbf{H}(i) \geqslant M \times N \times q_{\text{low}}\} \tag{4.8}$$

$$a'_{\text{high}} = \max\{i \mid \mathbf{H}(i) \leqslant M \times N \times (1 - q_{\text{high}})\} \tag{4.9}$$

其中 $0 \leqslant q_{\text{low}}, q_{\text{high}} \leqslant 1$，$q_{\text{low}} + q_{\text{high}} \leqslant 1$，$M \times N$ 是图像中的像素数。所有在像素值 a'_{low} 和 a'_{high} 以外的值(包括此值)分别映射到极值 $a_{\min}$ 和 $a_{\max}$，中间值线性映射到区间$[a_{\min}, a_{\max}]$。这样修正后的自动对比度调整运算的映射函数 $f_{\text{mac}}()$ 定义如下：

$$f_{\text{mac}}(a) = \begin{cases} a_{\min} & a \leqslant a'_{\text{low}} \\ a_{\min} + (a - a'_{\text{low}}) \times \dfrac{a_{\max} - a_{\min}}{a'_{\text{high}} - a'_{\text{low}}} & a'_{\text{low}} < a < a'_{\text{high}} \\ a_{\max} & a \geqslant a'_{\text{high}} \end{cases} \tag{4.10}$$

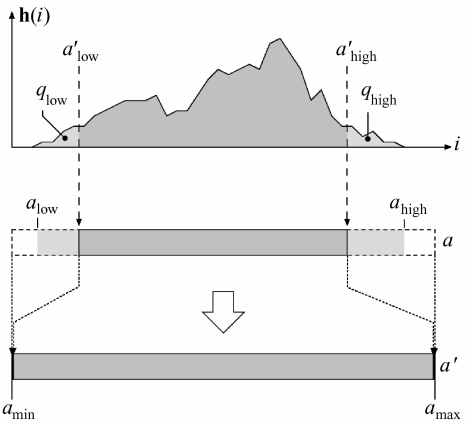


图 4.6　修正的自动对比度调整操作(式(4.10))。预先定义图像像素的分位数(q_{low}，q_{high})——直方图 $\mathbf{h}(i)$ 中显示的左边黑色区域以及右边末尾部分黑色区域是“渗透”域(即，映射到目标范围的极值)。中间值($a = a_{\text{low}} \cdots a_{\text{high}}$)线性映射到区间$[a_{\min}, a_{\max}]$

利用式(4.10)，映射到最小值和最大值就不只是依赖于单个极值，而是依赖于一组代表性像素集合。通常，上限和下限分位数的值是相同的(即 $q_{\text{low}} = q_{\text{high}} = q$)，常用值为 $q = 0.005, \cdots, 0.015$(即 0.5%，…，1.5%)。例如，在 Adobe Photoshop 的自动对比度调整中，强度取值范围的两端充满了所有像素数的 0.5%($q = 0.005$)。自动对比度调整是一个常用的点运算方法，因此几乎所有图像处理软件中都有此功能。ImageJ 将修正的自动对比度调