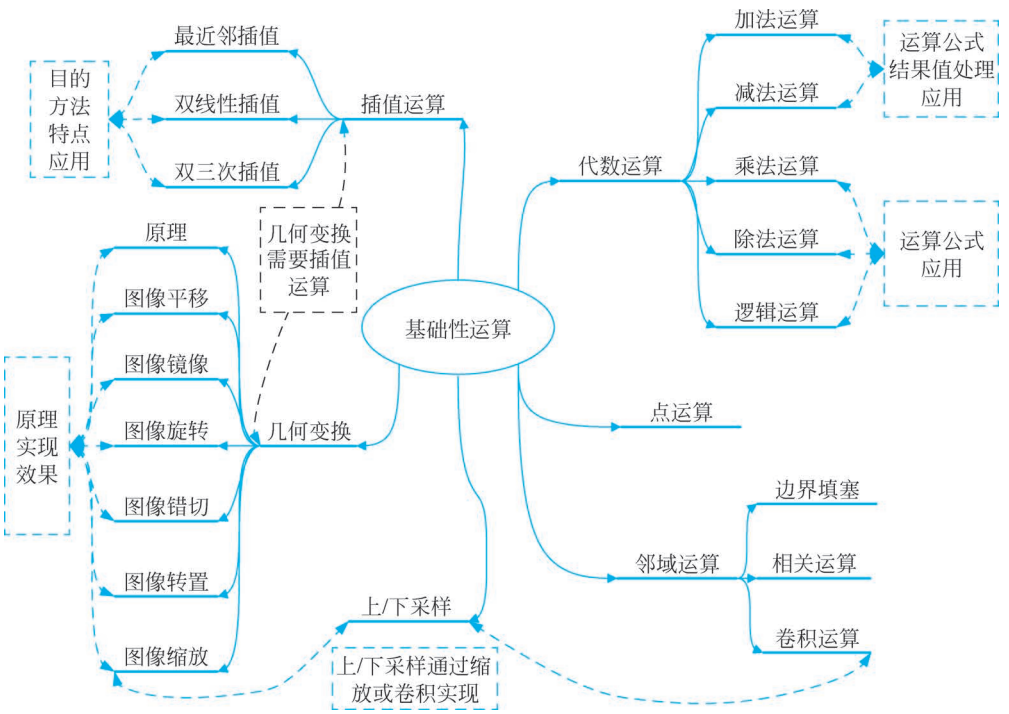
第 3 章

CHAPTER 3

图像基础性运算

本章思维导图



图像处理的过程是对图像像素值进行运算的过程，处理效果不同，运算对应的名称也不同。本章学习在不同的图像处理算法中经常用到的运算，即基础性运算，包括点运算、邻域运算、插值运算、几何变换、代数运算、上采样和下采样。

3.1 点运算

对图像进行点运算时，每个像素的输出值只取决于其自身的输入和相关参数，与其他像素无关，是一种常用的处理方法，如灰度级变换、代数运算。

【例 3.1】 设计程序，增大 office.jpg 图像的亮度。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('office.jpg', cv.IMREAD_GRAYSCALE)
Image = Image / 255
result = 4 * Image
cv.imshow("Original image", Image)
cv.imshow("Result image", result)
cv.waitKey()
```

运行程序，读取的灰度图像如图 3-1(a)所示，整体很暗；对图像进行点运算，将每个像素的像素值都增大为原来的 4 倍，处理后的像素值由自身原来的值和增大的倍数决定，跟周围像素的值无关。增大亮度的图像如图 3-1(b)所示，超出像素值表达范围的会被限幅为最大值。

(a) 原图　　(b) 增大亮度图

图 3-1 点运算示例

3.2 邻域运算

邻域运算是每个像素和其周围邻点共同参与的运算，通常通过模板操作进行，每个像素的输出值由其周围邻域内像素的值和模板中的数值共同决定。模板也叫滤波器、核、掩模或窗口，用一个小的二维阵列表示(如 3×3)。通常把对应的模板上的值称为加权系数。

1. 相关运算

相关运算如图 3-2 所示。将模板在图像上移动，每移动到一个位置，即模板的中心对准 1 个像素，模板所覆盖范围内的像素值分别与模板内对应系数相乘，乘积求和即为该像素的输出值，如式(3-1)所示。模板顺次移动，每个位置处计算出一个值，最终得到一幅新图像，如式(3-2)所示。

$$g(x,y)=\sum_{m,n} f(x+m,y+n)h(m,n) \tag{3-1}$$

$$\boldsymbol{g}=\boldsymbol{f}\otimes\boldsymbol{h} \tag{3-2}$$

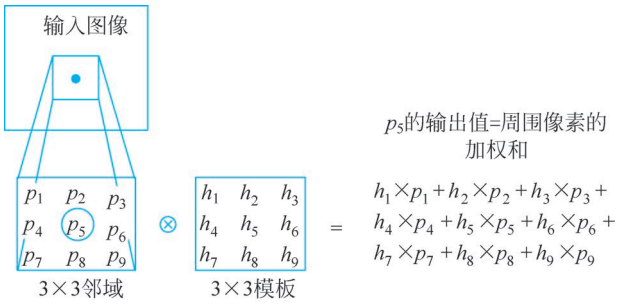


图 3-2　相关运算示意图

其中，$\boldsymbol{h}$ 是模板。

2. 卷积运算

对式(3-1)进行变形，将 f 中的偏移量符号反向，得

$$\begin{aligned} g(x,y)&=\sum_{m,n} f(x-m,y-n)h(m,n)\\ &=\sum_{m,n} f(m,n)h(x-m,y-n) \end{aligned} \tag{3-3}$$

称为卷积运算。图像的卷积运算表达为

$$\boldsymbol{g}=\boldsymbol{f}*\boldsymbol{h} \tag{3-4}$$

由式(3-1)和式(3-3)可以看出，卷积运算是将模板进行翻转(上下换位、左右换位)后，再进行相关运算，当模板中心对称时，两者运算结果一致。

邻域运算的效果由模板决定，模板中数据取值不同，处理效果也不同。

3. 边界填塞

当模板中心与图像外围像素重合时，模板的部分行和列可能会处于图像之外，没有相应的像素值与模板数据进行运算。对于这种问题，需要采用一定的措施来解决。

假设模板是大小为 $n\times n$ 的方形模板，对于图像中行和列方向上距离边缘小于 $(n-1)/2$ 像素的区域，根据不同要求，可以采用不同的处理方法。

(1) 不处理该区域中原来的像素。例如，对图像平滑去噪，外围像素即使存在一些噪声，一般不会影响对图像内容的理解，可以保留这些像素值。

(2) 在图像边缘以外再补上 $(n-1)/2$ 行和 $(n-1)/2$ 列像素。不同的情况下，有不同的填塞方法，例如，可以将对应的像素值置为零，设为固定的值，复制或者镜像反射外围像素值等方式。

在 OpenCV 的枚举 borderTypes 中给出多种填塞方法，取值如表 3-1 所示。

表 3-1　borderTypes 取值及含义

参　数	含　义
cv. BORDER_CONSTANT	指定为固定值 i：iiiiii|abcdefgh|iiiiii(|表示图像边界)
cv. BORDER_REPLICATE	复制最外围像素值：aaaaaa|abcdefgh|hhhhhhh
cv. BORDER_REFLECT	镜像反射外围像素：fedcba|abcdefgh|hgfedcb
cv. BORDER_WRAP	cdefgh|abcdefgh|abcdefg
cv. BORDER_REFLECT_101	gfedcb|abcdefgh|gfedcba
cv. BORDER_TRANSPARENT	填塞透明像素：uvwxyz|abcdefgh|ijklmno
cv. BORDER_DEFAULT	同 cv. BORDER_REFLECT_101

【例 3.2】 一幅图像为 $\begin{bmatrix}3&3&3&3&3\\3&7&7&7&3\\3&7&15&7&3\\3&7&7&7&3\\3&3&3&3&3\end{bmatrix}$，模板为 $\frac{1}{5}\begin{bmatrix}0&1&0\\1&1&1\\0&1&0\end{bmatrix}$，进行卷积运算，外围像素保持原像素值不变。

解：由于模板中心对称，卷积运算和相关运算一致。以模板中心点对准像素 15 为例，模板覆盖范围内像素值为 $\begin{bmatrix}7&7&7\\7&\underline{15}&7\\7&7&7\end{bmatrix}$，9 个数据分别和模板中的 9 个值对应相乘，乘积相加得 8.6，取整，输出图像中该点的值变为 9。对图像中每一点进行相同的处理，即卷积运算。

当模板中心点对准外围像素，如左下角像素 3，模板覆盖范围内像素值为 $\begin{bmatrix}\_&3&7\\\_&3&3\\\_&\_&\_\end{bmatrix}$，有 5 个数据为空，不进行相乘相加运算，保留像素值 3 不变，其他外围像素同样，输出结果如图 3-3(a)所示。或者将 5 个数据补充为 0，即 $\begin{bmatrix}\underline{0}&3&7\\\underline{0}&3&3\\\underline{0}&\underline{0}&\underline{0}\end{bmatrix}$ 和模板元素对应相乘再相加，得 1.8，取整，输出为 2，其他外围像素同样，输出结果如图 3-3(b)所示。两种处理方法仅影响外围像素值。

第 2 集
微课视频

$\begin{bmatrix}3&3&3&3&3\\3&5&8&5&3\\3&8&9&8&3\\3&5&8&5&3\\3&3&3&3&3\end{bmatrix}$　　$\begin{bmatrix}2&3&3&3&2\\3&5&8&5&3\\3&8&9&8&3\\3&5&8&5&3\\2&3&3&3&2\end{bmatrix}$

(a) 保留外围像素不变的计算结果　　(b) 外围像素外补 0 的计算结果

图 3-3　卷积运算 1

例题中对中心像素和上、下、左、右 4 个邻点取平均，能够实现抑制噪声的功能。

关于模板运算的含义及示例运算过程，请扫描二维码，查看讲解。

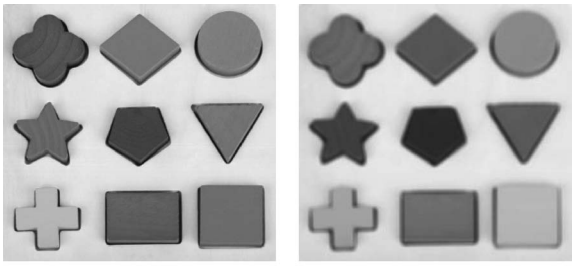
【例 3.3】 对 shape.png 图像采用取值全为 1/25 的 5×5 模板进行卷积运算，外围像素保持原像素值不变。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('shape.png', cv.IMREAD_GRAYSCALE)
height, width = np.shape(Image)
H = np.ones([5, 5]) / 25                    #定义 5×5 的模板
H = np.flip(H, (0, 1))                      #模板上下、左右翻转
h, w = np.shape(H)
r1, r2 = h // 2, w // 2                     #模板半径
result = np.array(Image)                    #将原图赋予 result,便于保留边缘像素值
for y in range(r1, height - r1):            #模板遍历图像时避开外围像素
    for x in range(r2, width - r2):
        neighbors = Image[y - r1 : y + r1 + 1, x - r2 : x + r2 + 1]
                                            #获取模板覆盖范围内像素值矩阵
```

```
        Parray = neighbors * H          #模板系数和像素值对应相乘
        result[y, x] = np.sum(Parray)   #乘积求和作为当前点输出
cv.imshow("Original image", Image)
cv.imshow("Result image", result)
cv.waitKey()
```

程序运行结果如图 3-4 所示。



(a) 原灰度图　　(b) 卷积结果图

图 3-4　卷积运算 2

OpenCV 的 filter2D 函数实现相关运算，其调用格式如下：

```
cv.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]]) -> dst
```

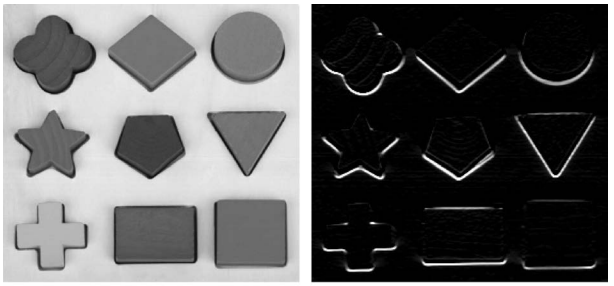
参数 src 是输入图像，可以是二维或三维数组；ddepth 指定目标图像的深度；kernel 是相关运算的模板；anchor 指定模板的参考原点，应在模板内部，默认(−1,−1)表示为几何中心；borderType 指定边界填塞方法，如表 3-1 所示。

【例 3.4】　采用 filter2D 函数实现对 shape.png 图像的卷积运算，卷积核为 $\boldsymbol{h}=\begin{bmatrix}-1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1\end{bmatrix}$。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('shape.png', cv.IMREAD_GRAYSCALE)
H = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
result = cv.filter2D(Image, cv.CV_8U, H, cv.BORDER_REPLICATE)
cv.imshow("original image", Image)
cv.imshow("result image", result)
cv.waitKey()
```

程序运行结果如图 3-5 所示。



(a) 原灰度图　　(b) 卷积结果图

图 3-5　卷积运算 3

3.3 插值运算

在图像处理过程中，可能会产生一些原图中没有的新的像素，即像素坐标非整数，给这些像素赋值需要应用插值运算，即利用已知邻近像素的灰度值产生未知像素的灰度值。插值效果的好坏将直接影响图像显示的视觉效果。常用的插值方法有最近邻插值(Nearest Neighbor Interpolation)、双线性插值(Bilinear Interpolation)、双三次插值(Bicubic Interpolation)等。

1. 最近邻插值

最近邻插值是最简单的插值方法，将新像素的像素值设为距离它最近的输入像素的像素值。当图像中邻近像素之间灰度级有较大的变化时，该算法产生的新图像的细节比较粗糙。

2. 双线性插值

双线性插值原理图如图 3-6 所示，对于一个插值点$(x+a,y+b)$(其中 x、y 均为非负整数，$0\leqslant a,b\leqslant 1$)，则该点的值 $f(x+a,y+b)$ 可由原图像中坐标为(x,y)、$(x+1,y)$、$(x,y+1)$、$(x+1,y+1)$所对应的 4 个像素的值决定：

$$\begin{aligned}
&f(x,y+b)=f(x,y)+b\left[f(x,y+1)-f(x,y)\right]\\
&f(x+1,y+b)=f(x+1,y)+b\left[f(x+1,y+1)-f(x+1,y)\right]\\
&f(x+a,y+b)=f(x,y+b)+a\left[f(x+1,y+b)-f(x,y+b)\right]
\end{aligned}\tag{3-5}$$

可以看出，双线性插值是根据非整数像素距周围 4 个像素的距离比，由 4 个邻点像素值进行线性插值，这种方法具有防锯齿效果，新图像拥有较平滑的边缘。

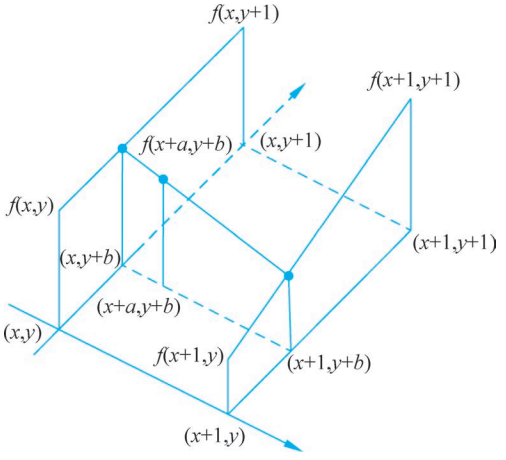


图 3-6 双线性插值原理图

3. 双三次插值

双三次插值是一种较复杂的插值方式，在计算新像素的值时，要将周围的 16 个点全部考虑进去。双三次插值图像边缘比双线性插值图像更平滑，同时也需要更大的计算量。

点$(x+a,y+b)$处的像素值$f(x+a,y+b)$可由式(3-6)计算。

$$f(x+a,y+b)=[\boldsymbol{A}]\ [\boldsymbol{B}]\ [\boldsymbol{C}]\tag{3-6}$$

其中，

$$[\boldsymbol{A}]=[s(a+1)\quad s(a)\quad s(a-1)\quad s(a-2)]$$

$$[\boldsymbol{B}]=\begin{bmatrix} f(x-1,y-1) & f(x-1,y) & f(x-1,y+1) & f(x-1,y+2) \\ f(x+0,y-1) & f(x+0,y) & f(x+0,y+1) & f(x+0,y+2) \\ f(x+1,y-1) & f(x+1,y) & f(x+1,y+1) & f(x+1,y+2) \\ f(x+2,y-1) & f(x+2,y) & f(x+2,y+1) & f(x+2,y+2) \end{bmatrix}$$

$$[\boldsymbol{C}]=\begin{bmatrix} s(b+1) \\ s(b+0) \\ s(b-1) \\ s(b-2) \end{bmatrix} \quad s(k)=\begin{cases} 1-2\times|k|^2+|k|^3, & 0\leqslant|k|<1 \\ 4-8\times|k|+5\times|k|^2-|k|^3, & 1\leqslant|k|<2 \\ 0, & |k|\geqslant 2 \end{cases}$$

在 OpenCV 的枚举 InterpolationFlags 中给出多种插值方法，其中，最近邻插值、双线性插值和双三次插值的表示分别为 cv. INTER_NEAREST、cv. INTER_LINEAR 和 cv. INTER_CUBIC。

3.4　几何变换

几何变换是指对图像进行平移、旋转、镜像、缩放、错切、转置等变换，改变图像的大小、形状和位置，常用于图像变形、几何失真图像的校正、图像配准、影像特技处理等。

3.4.1　图像几何变换原理

图像几何变换将图像中任一像素映射到一个新位置，是一种空间变换，关键在于确定变换前后图像中点与点之间的映射关系，明确原图像任意像素变换后的坐标，或者变换后的图像像素在原图像中的坐标位置，对新图像像素赋值而产生新图像。

1. 几何变换的齐次坐标表示

用 $n+1$ 维向量表示 n 维向量的方法称为齐次坐标表示法。图像空间一个点 (x,y) 用齐次坐标表示为 $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$，和某个变换矩阵 $\boldsymbol{T}=\begin{bmatrix} a & b & k \\ c & d & m \\ p & q & s \end{bmatrix}$ 相乘变为新的点 $\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$，如式(3-7)所示，这种变换称为几何变换。

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}=\begin{bmatrix} a & b & k \\ c & d & m \\ p & q & s \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3-7}$$

若 $\boldsymbol{T}=\begin{bmatrix} a & b & k \\ c & d & m \\ 0 & 0 & 1 \end{bmatrix}$，则称这种有 6 个参数的变换为仿射变换；若 $\boldsymbol{T}=\begin{bmatrix} a & b & k \\ c & d & m \\ p & q & 1 \end{bmatrix}$，称这种有 8 个参数的变换为投影变换。

二维图像可以表示为 $3\times MN$ 的点集矩阵 $\begin{bmatrix} x_1 & x_2 & \cdots & x_{MN} \\ y_1 & y_2 & \cdots & y_{MN} \\ 1 & 1 & \cdots & 1 \end{bmatrix}$，实现二维图像几何变换的一般过程如下。

变换后的点集矩阵＝变换矩阵 $\boldsymbol{T}$×变换前的点集矩阵。

2. 图像几何变换过程

图像几何变换可以采用前向映射法和后向映射法实现。前向映射法计算原图像中像素(x,y)在新图像中的对应点(x',y')，并给(x',y')赋值$f(x,y)$。但是，如果(x',y')为非整数像素，需要将(x',y')取整，然后再复制$f(x,y)$，或者使用加权的方法将$f(x,y)$分配给周围4个近邻，这些操作会丢失细节；同时，前向映射会产生裂缝或空洞，即新图像中某些像素没有赋值，需要再用邻近的像素填补，进一步导致图像模糊，在图像放大时更为明显。

实际几何变换中经常采用后向映射法，后向映射法计算新图像中的像素在原图像中的对应点，并反向赋值，具体步骤如下。

(1) 根据不同的几何变换公式计算新图像的尺寸。

(2) 根据几何变换的逆变换，确定新图像中的每一点在原图像中的对应点。

(3) 按对应关系给新图像中各像素赋值。

① 若原图像中的对应点存在，直接将其值赋给新图像中的点。

② 若原图像中的对应点坐标超出图像宽高范围，直接赋背景色。

③ 若原图像中的对应点坐标在图像宽高范围内，但坐标非整数，采用插值的方法计算该点的值，并赋给新图像。

3.4.2 图像平移

图像平移是将一幅图像上的所有点都按照给定的偏移量沿x轴、y轴移动，平移后的图像与原图像相同，内容不发生变化，只是改变了原有景物在画面上的位置。

将点(x,y)进行平移后，移到点(x',y')处，其中x轴方向的平移量为Δx，y轴方向的平移量为Δy，则平移变换为

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3-8}$$

平移变换求逆，得

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\Delta x \\ 0 & 1 & -\Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{3-9}$$

这样，平移后图像上每一点(x',y')都可在原图像中找到对应点(x,y)。

如果图像经过平移处理后，不想丢失被移出的部分图像，可将可视区域的宽度扩大$|\Delta x|$，高度扩大$|\Delta y|$。

【例3.5】 根据式(3-8)和式(3-9)，采用后向映射法实现图像平移，分别沿x轴、y轴平移20像素。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('lotus.jpg')
Image = Image / 255                                  # 像素值归一化
h, w, c = np.shape(Image)                            # 获取表示图像的三维数组形状
result = np.ones([h, w, c])                          # 新图像初始化
deltax, deltay = 20, 20                              # 指定平移量
for y in range(h):
```

```
    for x in range(w):                                    #循环扫描新图像中的点
        oldx, oldy = x - deltax, y - deltay               #确定新图像中点在原图中的对应点
        if (oldx >= 0) & (oldx < w) & (oldy >= 0) & (oldy < h):
            result[y, x, :] = Image[oldy, oldx, :]        #若对应点在图像内则赋值
cv.imshow("Original image", Image)
cv.imshow("Result image", result)
cv.waitKey()
```

程序运行结果如图 3-7 所示。其中，图 3-7(a)为原始图像，图 3-7(b)为分别沿 x、y 方向平移 20 像素后的结果，图 3-7(c)在图 3-7(b)的处理结果基础之上把可视区域进行了扩大。

(a) 原图　(b) 平移后的图像　(c) 可视区域扩大后的平移图像

图 3-7　图像平移变换实例

由于平移前后的图像相同，而且图像上的像素顺序放置，所以图像的平移也可以通过直接逐行地复制图像实现。

OpenCV 中 warpAffine 函数实现几何变换，理解原理后，可以采用函数实现图像的几何变换，调用格式如下：

```
cv.warpAffine(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) -> dst
```

参数 src 是输入图像，可以是二维或三维数组；M 是 2×3 的变换矩阵；dsize 是输出图像尺寸；flags 选定插值方法；borderMode 选择边界填塞方法，取 cv.BORDER_CONSTANT 时，borderValue 指定填充的颜色值。

【例 3.6】　采用函数 warpAffine 实现图像的平移变换。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('lotus.jpg')
Image = Image / 255
h, w, c = np.shape(Image)
deltax, deltay = 20, 20
newh, neww = h + np.abs(deltax), w + np.abs(deltay)      #新图像扩大尺寸
T = np.array([[1.0, 0, deltax], [0, 1, deltay]])
result = cv.warpAffine(Image, T, dsize = (neww, newh), flags = cv.INTER_LINEAR,
           borderValue = [1, 1, 1])
             #进行平移变换,双线性插值,边界外填充为白色,[1,1,1]为 BGR 颜色值
cv.imshow("Original image", Image)
cv.imshow("Result image", result)
cv.waitKey()
```

程序运行结果如图 3-7(a)和图 3-7(c)所示。

3.4.3　图像镜像

设图像的分辨率为 $M \times N$，采用像素坐标系，图像镜像变换如式(3-10)所示，可以看出，镜

像就是左右、上下或对角对换。

$$\text{水平镜像：}\begin{bmatrix}x'\\y'\\1\end{bmatrix}=\begin{bmatrix}-1&0&M-1\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

$$\text{垂直镜像：}\begin{bmatrix}x'\\y'\\1\end{bmatrix}=\begin{bmatrix}1&0&0\\0&-1&N-1\\0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}\tag{3-10}$$

$$\text{对角镜像：}\begin{bmatrix}x'\\y'\\1\end{bmatrix}=\begin{bmatrix}-1&0&M-1\\0&-1&N-1\\0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

【例 3.7】 采用 NumPy 中的矩阵翻转函数实现图像镜像变换，并将镜像图像拼接成大图。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('dog.jpg')
Image = Image / 255
HImage = np.flip(Image, 1)                  # 左右翻转，即水平镜像
VImage = np.flip(Image, 0)                  # 上下翻转，即垂直镜像
CImage = np.flip(HImage, 0)                 # 左右翻转后再上下翻转，即对角镜像
result1 = cv.hconcat((Image, HImage))       # 原图和水平镜像图水平拼接
result2 = cv.hconcat((VImage, CImage))      # 垂直镜像和对角镜像图水平拼接
result = cv.vconcat((result1, result2))     # 上下拼接为一幅大图
cv.imshow("Original image", Image)
cv.imshow("Result image", result)
cv.waitKey()
```

程序运行，镜像拼接效果如图 3-8 所示。

图 3-8 镜像图像拼接

3.4.4 图像旋转

图像旋转是指以图像中的某一点为原点，以逆时针或顺时针方向将图像上的所有像素都旋转一个相同的角度。经过旋转变换后，图像的大小一般会改变，并且图像中的部分像素可能会旋转出可视区域范围，因此需要扩大可视区域范围以显示所有的图像。

1. 图像旋转的原理

设原图像中点(x,y)绕原点逆时针旋转θ角后的对应点为(x',y')，如图3-9所示。

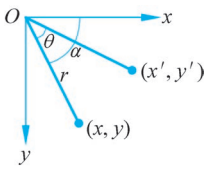


图3-9　图像旋转示意图

在旋转变换前，原图像中点(x,y)的坐标表达式为

$$\begin{cases} x = r \cdot \cos\alpha \\ y = r \cdot \sin\alpha \end{cases} \tag{3-11}$$

逆时针旋转θ角后为

$$\begin{cases} x' = r \cdot \cos(\alpha-\theta) = x \cdot \cos\theta + y \cdot \sin\theta \\ y' = r \cdot \sin(\alpha-\theta) = -x \cdot \sin\theta + y \cdot \cos\theta \end{cases} \tag{3-12}$$

则图像旋转变换的矩阵表达为

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3-13}$$

若顺时针旋转，则角度θ取负值。

绕原点旋转的逆变换为

$$\begin{cases} x = x'\cos\theta - y'\sin\theta \\ y = x'\sin\theta + y'\cos\theta \end{cases} \tag{3-14}$$

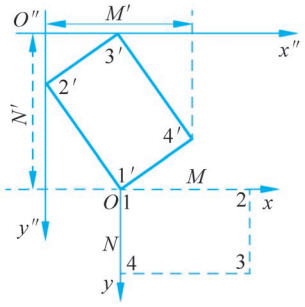


图3-10　绕原点逆时针旋转示意图

2. 图像旋转变换过程

1）确定旋转后新图像尺寸

绕原点逆时针旋转示意图如图3-10所示：xOy为原始图像坐标系，图像4个角标注为1、2、3、4，旋转后为1′、2′、3′、4′，新图像坐标系表示为$x''O''y''$。

设原始图像大小为$M\times N$，以图像起始点作为坐标原点，则原始图像4个角坐标分别为

$$(x_1,y_1)=(0,0),\quad (x_2,y_2)=(M-1,0),$$
$$(x_3,y_3)=(M-1,N-1),\quad (x_4,y_4)=(0,N-1)$$

按照逆时针旋转公式，即式(3-12)，旋转后，4个点在原坐标系中的坐标为

$$\begin{cases} (x'_1,y'_1)=(0,0) \\ (x'_2,y'_2)=((M-1)\cos\theta,-(M-1)\sin\theta) \\ (x'_3,y'_3)=((M-1)\cos\theta+(N-1)\sin\theta,-(M-1)\sin\theta+(N-1)\cos\theta) \\ (x'_4,y'_4)=((N-1)\sin\theta,(N-1)\cos\theta) \end{cases} \tag{3-15}$$

令$\max x'$和$\min x'$分别为坐标值x'_1,x'_2,x'_3,x'_4的最大值和最小值，$\max y'$和$\min y'$分别为坐标值y'_1,y'_2,y'_3,y'_4的最大值和最小值，则新图像的宽度M'和高度N'为

$$\begin{cases} M' = \max x' - \min x' + 1 \\ N' = \max y' - \min y' + 1 \end{cases} \tag{3-16}$$

2）坐标变换

对于新图像中的像素(x'',y'')，$x''\in[0,M'-1]$，$y''\in[0,N'-1]$，先进行平移变换，变换

到原像素坐标系

$$\begin{cases} x' = x'' + \min x' \\ y' = y'' + \min y' \end{cases} \tag{3-17}$$

3）旋转逆变换

对于每一个点(x', y')，利用旋转变换的逆变换式(3-14)，在原图像中找对应点。

4）给新图像赋值

按对应关系直接给新图像中各像素赋值，或采用插值方法给新图像中各像素赋值。

【例 3.8】 有一幅图像 $\boldsymbol{f}(x,y)=\begin{bmatrix} 59 & 60 & 58 \\ 61 & 59 & 57 \\ 62 & 56 & 55 \end{bmatrix}$，以图像原点为坐标原点，将其逆时针旋转 30°。

解：（1）确定旋转后新图像的分辨率。按照式(3-15)计算图像 4 角点旋转后的坐标为

$$\begin{cases} (x'_1, y'_1) = (0,0) \\ (x'_2, y'_2) = (2\cos 30°, -2\sin 30°) = (1.732, -1) \\ (x'_3, y'_3) = (2\cos 30° + 2\sin 30°, -2\sin 30° + 2\cos 30°) = (2.732, 0.732) \\ (x'_4, y'_4) = (2\sin 30°, 2\cos 30°) = (1, 1.732) \end{cases}$$

$$\max x' = 2.732 \quad \min x' = 0 \quad \max y' = 1.732 \quad \min y' = -1$$

计算新图像分辨率为

$$\begin{cases} M' = \max x' - \min x' + 1 = 3.732 \approx 4 \\ N' = \max y' - \min y' + 1 = 3.732 \approx 4 \end{cases}$$

所以，新图像中每一点(x'', y'')满足：$x'' \in [0,3]$，$y'' \in [0,3]$。

（2）对于新图像中的点(x'', y'')，先进行平移变换，变换到原像素坐标系(x', y')，再利用旋转变换的逆变换，在原图像中找对应点(x, y)，并赋值。对应关系如表 3-2 所示。

表 3-2 绕原点旋转像素对应关系

(x'', y'')	(x', y')	(x, y)	最近邻点	(x'', y'')	(x', y')	(x, y)	最近邻点
(0,0)	(0,−1)	(0.5,−0.866)	(1,−1)	(1,0)	(1,−1)	(1.366,−0.366)	(1,0)
(0,1)	(0,0)	(0,0)	(0,0)	(1,1)	(1,0)	(0.866,0.5)	(1,1)
(0,2)	(0,1)	(−0.5,0.866)	(−1,1)	(1,2)	(1,1)	(0.366,1.366)	(0,1)
(0,3)	(0,2)	(−1,1.732)	(−1,2)	(1,3)	(1,2)	(−0.134,2.232)	(0,2)
(2,0)	(2,−1)	(2.232,0.134)	(2,0)	(3,0)	(3,−1)	(3.098,0.634)	(3,1)
(2,1)	(2,0)	(1.732,1)	(2,1)	(3,1)	(3,0)	(2.598,1.5)	(3,2)
(2,2)	(2,1)	(1.232,1.866)	(1,2)	(3,2)	(3,1)	(2.098,2.366)	(2,2)
(2,3)	(2,2)	(0.732,2.732)	(1,3)	(3,3)	(3,2)	(1.598,3.232)	(2,3)

产生的新图像 $\boldsymbol{g}(x,y)$ 为

$$\boldsymbol{g}(x,y)=\begin{bmatrix} 255 & 60 & 58 & 255 \\ 59 & 59 & 57 & 255 \\ 255 & 61 & 56 & 55 \\ 255 & 62 & 255 & 255 \end{bmatrix}$$

在上述运算过程中，原图中的对应点超出图像范围，或新图像中的点在原图像中没有对应点，直接赋背景色 255；未超出图像范围但不是整数像素的对应点按最近邻插值。

为提高图像效果，可以采用双线性插值，如新图像中(1,2)点，对应原图中(0.366,1.366)点，该点位于(0,1)、(1,1)、(0,2)、(1,2)四点之间，可以按照式(3-5)计算(0.366,1.366)点的值，并赋给新图像中的(1,2)点。

$$f(0,1.366)=f(0,1)+0.366[f(0,2)-f(0,1)]=61+0.366\times(62-61)=61.366$$

$$f(1,1.366)=f(1,1)+0.366[f(1,2)-f(1,1)]=59+0.366\times(56-59)=57.902$$

$$f(0.366,1.366)=f(0,1.366)+0.366[f(1,1.366)-f(0,1.366)]\approx 60$$

绕中心点旋转先要将坐标系平移到中心点，再绕原点旋转进行变换，然后平移回原坐标原点。绕任意点旋转与此相同，仅仅是平移量的不同。

关于绕中心点的旋转变换，请扫描二维码，查看讲解。

3. 图像旋转的实现

图像旋转按照例 3.8 中所述步骤实现。OpenCV 中 rotate 函数可以将二维矩阵顺时针旋转 90°、180°和 270°，调用格式如下：

```
cv.rotate(src, rotateCode[, dst]) -> dst
```

参数 rotateCode 选择旋转的角度，可取：cv.ROTATE_90_CLOCKWISE、cv.ROTATE_180 和 cv.ROTATE_90_COUNTERCLOCKWISE。

函数 getRotationMatrix2D 计算绕中心点旋转的变换矩阵，调用格式如下：

```
cv.getRotationMatrix2D(center, angle, scale) -> retval
```

参数 center 指明图像旋转中心；angle 是旋转角度，正值表示逆时针旋转(相对于左上角)；scale 是比例变换因子。设置旋转变换矩阵后可以用 warpAffine 函数实现旋转变换。


第 3 集
微课视频

【例 3.9】　编写程序，实现图像旋转。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('dog.jpg')
Image = Image / 255
result1 = cv.rotate(Image, cv.ROTATE_90_CLOCKWISE)        # 顺时针旋转 90°
height, width, color = np.shape(Image)
angle, center = 15, (width // 2, height // 2)
T = cv.getRotationMatrix2D(center, angle, 1.0)            # 创建旋转变换矩阵
result2 = cv.warpAffine(Image, T, dsize=(width, height), flags=cv.INTER_LINEAR,
                borderValue=[1, 1, 1])     # 旋转变换，和原图尺寸一样，双线性插值，填充白色
cv.imshow("Original image", Image)
cv.imshow("Rotating by90 degrees clockwise", result1)
cv.imshow("Rotating any angle", result2)
cv.waitKey()
```

程序运行结果如图 3-11 所示。

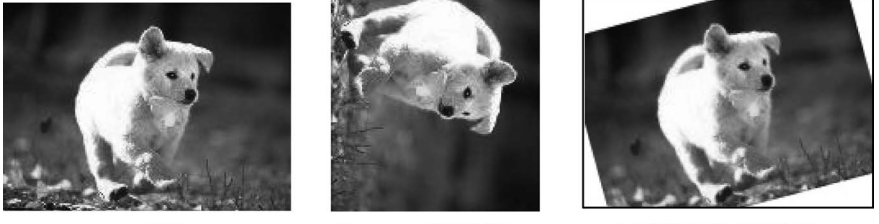

(a) 原图　(b) rotate函数旋转90°　(c) 旋转任意指定角度

图 3-11　图像旋转变换

3.4.5 图像缩放

图像缩放是指将给定图像的尺寸在 x、y 方向分别缩放 k_x、k_y 倍，获得一幅新的图像。其中，若 $k_x=k_y$，即在 x 轴、y 轴方向缩放的比率相同，则称为图像的按比例缩放。若 $k_x\neq k_y$，缩放会改变原始图像像素间的相对位置，产生几何畸变，称为图像的不按比例缩放。进行缩放变换后，新图像的分辨率为 $k_xM\times k_yN$。

图像的缩放处理分为图像的缩小和图像的放大处理：

(1) 当 $0<k_x,k_y<1$，实现图像的缩小处理；

(2) 当 $k_x,k_y>1$，则实现图像的放大处理。

设原图像中点(x,y)进行缩放处理后，变换到点(x',y')，则缩放处理的矩阵形式可表示为

$$\begin{bmatrix}x'\\y'\\1\end{bmatrix}=\begin{bmatrix}k_x&0&0\\0&k_y&0\\0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix} \tag{3-18}$$

可以根据后向映射法，按照式(3-18)实现图像缩放。可以通过定义变换矩阵，利用 warpAffine 函数实现图像缩放变换。此外，OpenCV 也提供了图像缩放变换函数 resize，调用格式如下：

```
cv.resize(src, dsize[, dst[, fx[, fy[, interpolation]]]]) -> dst
```

参数 src 可以是二维或三维数组；dsize 指定输出图像尺寸；fx 和 fy 是水平和垂直缩放因子，设为 0 时，根据 dsize 和原图像尺寸计算；interpolation 选择插值计算方法。

【例 3.10】 编写程序，实现图像缩放变换。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('dog.jpg')
result1 = cv.resize(Image, dsize = None, fx = 0.8, fy = 1.9,
                    interpolation = cv.INTER_LINEAR)     # 不按比例缩放
result2 = cv.resize(Image, dsize = None, fx = 0.8, fy = 0.8,
                    interpolation = cv.INTER_LINEAR)     # 按比例缩小
cv.imshow("Original image", Image)
cv.imshow("Result image 1", result1)
cv.imshow("Result image 2", result2)
cv.waitKey()
```

程序运行结果如图 3-12 所示。

(a) 原图　(b) 不按比例　(c) 按比例

图 3-12　图像缩放变换

3.4.6　图像错切

图像的错切变换是平面景物在投影平面上的非垂直投影。错切变换使图像中的图形产生扭变。这种扭变只在水平或垂直方向上产生时，分别称为水平方向错切和垂直方向错切。

设原图像中点(x,y)进行错切变换后，变换到点(x',y')，则错切变换的矩阵表达式为

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & d_x & 0 \\ d_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3-19}$$

【例 3.11】　采用函数 warpAffine 实现图像错切变换。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('dog.jpg')
Image = Image / 255
height, width, color = np.shape(Image)
dx, dy = 0.5, 0.2
T1 = np.array([[1, dx, 0], [0, 1, 0]])                    # 水平错切变换矩阵
T2 = np.array([[1, 0, 0], [dy, 1, 0]])                    # 垂直错切变换矩阵
T3 = np.array([[1, dx, 0], [dy, 1, 0]])                   # 水平、垂直同时错切变换矩阵
h1, w1 = height, int(width - 1 + (height - 1) * dx)  # 新图像尺寸
h2, w2 = int(height - 1 + (width - 1) * dy), width
h3, w3 = int(height - 1 + (width - 1) * dy), int(width - 1 + (height - 1) * dx)
hImage = cv.warpAffine(Image, T1, dsize = (w1, h1), flags = cv.INTER_LINEAR,
          borderValue = [1, 1, 1])                        # 水平错切
vImage = cv.warpAffine(Image, T2, dsize = (w2, h2), flags = cv.INTER_LINEAR,
          borderValue = [1, 1, 1])                        # 垂直错切
hvImage = cv.warpAffine(Image, T3, dsize = (w3, h3), flags = cv.INTER_LINEAR,
          borderValue = [1, 1, 1])                        # 水平、垂直同时错切
cv.imshow("Original image", Image)
cv.imshow("Horizontal shear transform", hImage)
cv.imshow("Vertical shear transform", vImage)
cv.imshow("Shear transform", hvImage)
cv.waitKey()
```

程序运行结果如图 3-13 所示。

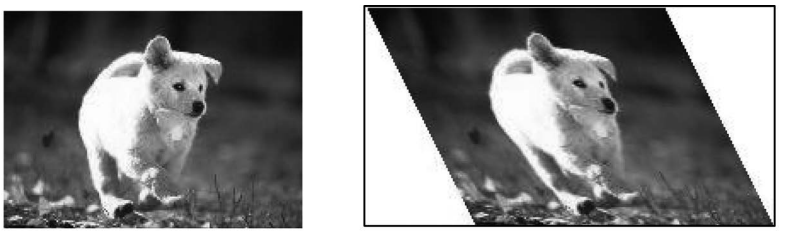
3.4.7　图像转置

图像转置变换是指将图像的行、列坐标互换。设原图像中的点(x,y)进行转置后，变换到点(x',y')，则转置变换的矩阵表达式为

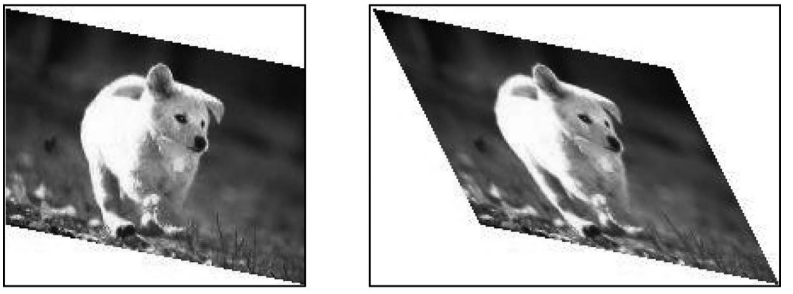
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3-20}$$

图像用矩阵表达，可以直接通过矩阵转置实现图像转置，也可以用 OpenCV 中的 transpose 函数实现，其调用格式如下：

```
cv.transpose(src[, dst]) -> dst
```



(a) 原图　(b) 水平错切



(c) 垂直错切　(d) 水平、垂直同时错切

图 3-13　图像错切变换

【例 3.12】 设计程序实现图像转置变换。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('dog.jpg')
height, width, color = np.shape(Image)
result1 = cv.transpose(Image)                          # 直接使用转置函数实现变换
result2 = np.zeros([width, height, color])
for i in range(color):
    result2[:, :, i] = Image[:, :, i].T                # 使用 NumPy 的".T" 对各通道转置
cv.imshow("Original image", Image)
cv.imshow("Transposing 1", result1)
cv.imshow("Transposing 2", result2 / 255)
cv.waitKey()
```

程序运行结果如图 3-14 所示。

(a) 原图　(b) 图像转置

图 3-14　图像转置变换

3.5　代数运算

图像代数运算是指对两幅或多幅输入图像进行点对点的加、减、乘、除，与、或、非等运算，有时涉及将简单的代数运算进行组合而得到更复杂的代数运算结果。从原理上来讲，代数运算简单易懂，但在实际应用中很常见。

3.5.1　加法运算

1. 原理

加法运算将两幅或多幅图像对应点像素值相加，如式(3-21)所示。

$$g(x,y)=f_1(x,y)+f_2(x,y) \tag{3-21}$$

式中，f_1、f_2 是同等大小的两幅图像。

进行相加运算，对应像素值的和可能会超出灰度值表达的范围，对于这种情况，可以采用下列方法进行处理。

(1) 截断处理。如果 $g(x,y)$ 大于 255，仍取 255；但新图像 $g(x,y)$ 像素值会偏大，图像整体较亮，后续需要灰度级调整。

(2) 加权求和，即

$$g(x,y)=\alpha f_1(x,y)+(1-\alpha)f_2(x,y) \tag{3-22}$$

式中，$\alpha\in[0,1]$。这种方法需要选择合适的 α。

编程时，可以用“+”运算符或者 OpenCV 中的函数 add 实现图像相加，采用函数 addWeighted 实现两幅图像线性组合，要求图像具有相同大小和通道数。调用格式如下：

```
cv.add(src1, src2[, dst[, mask[, dtype]]]) -> dst
cv.addWeighted(src1, alpha, src2, beta, gamma[, dst[, dtype]]) -> dst
```

参数 mask 是和图像同等大小的 8 位二维数组，其中取值不为 0 的像素将参与运算；dtype 设定输出数组的深度，默认值为－1，表示输入和输出深度相同。alpha 和 beta 分别是 src1 和 src2 的权系数，gamma 是叠加的常数项。

【例 3.13】 采用函数 add、addWeighted 实现图像相加。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Back = cv.imread('back.jpg')                    #背景图
Foreground = cv.imread('fore.jpg')              #前景图
cv.imshow("Back image", Back)
cv.imshow("Foreground image", Foreground)
result1 = cv.add(Back, Foreground)              #前景和背景直接相加
alpha, beta, gamma = 0.6, 0.4, 0                #alpha×Back + beta×Foreground + gamma
result2 = cv.addWeighted(Back, alpha, Foreground, beta, gamma)
thresh = 25
b, g, r = cv.split(Foreground)
Back[~((b < thresh) & (g < thresh) & (r < thresh))] = 0
                                                #前景图中目标部分在背景图中的对应部分被设为0
result3 = Back + Foreground
show = cv.hconcat((result1, result2, result3))  #三个相加结果拼接为一幅大图便于显示比较
cv.imshow("Image addition", show)
cv.waitKey()
```

程序运行结果如图 3-15 所示。采用不同的方式将图 3-15(a)和图 3-15(b)相加，图 3-15(c)是将和值限幅截断，图 3-15(d)是加权求和，图 3-15(e)是前景目标覆盖背景。

(a) 背景图　(b) 目标图　(c) 截断处理　(d) 加权求和　(e) 前景覆盖

图 3-15　加法运算效果

2. 主要应用

1) 多图像平均去除叠加性噪声

假设有一幅混有噪声的图像 $g(x,y)$ 由原始标准图像 $f(x,y)$ 和噪声 $n(x,y)$ 叠加而成，即

$$g(x,y)=f(x,y)+n(x,y) \tag{3-23}$$

若 $n(x,y)$ 为互不相关的加性噪声，且均值为 0。令 $E[g(x,y)]$ 为 $g(x,y)$ 的期望值，则有

$$E[g(x,y)]=E[f(x,y)+n(x,y)]=E[f(x,y)]=f(x,y) \tag{3-24}$$

对 L 幅重复采集的有噪声图像进行平均后的输出图像为

$$\bar{g}(x,y)=\frac{1}{L}\sum_{i=1}^{L}g_i(x,y)\approx E[g(x,y)]=f(x,y) \tag{3-25}$$

消除了图像中的噪声。

该方法常用于摄像机的视频图像中，用于减少电视摄像机光电摄像管或 CCD 器件所引起的噪声。

2) 图像合成和图像拼接

将一幅图像的内容经配准后叠加到另一幅图像上，以改善图像的视觉效果，不同视角的图像拼接到一起生成全景图像等，需要加法运算。

3) 在多光谱图像中的应用

在多光谱图像中，通过加法运算加宽波段，如绿色和红色波段图像相加可以得到近似全色图像。

3.5.2　减法运算

1. 原理

减法运算将两幅或多幅图像对应点像素值相减，如式(3-26)所示。

$$g(x,y)=f_1(x,y)-f_2(x,y) \tag{3-26}$$

式中，f_1、f_2 是同等大小的两幅图像。

进行相减运算，对应像素值的差可能为负数，对于这种情况，可以采用下列方法进行处理。

(1) 截断处理。如果 $g(x,y)$ 小于 0，仍取 0；但新图像 $g(x,y)$ 像素值会偏小，图像整体较暗，后续需要灰度级调整。

(2) 取绝对值，即

$$g(x,y)=|f_1(x,y)-f_2(x,y)| \tag{3-27}$$

2. 主要应用

1）显示两幅图像的差异

将两幅图像相减，灰度或颜色相同部分相减为 0，呈现黑色；相似部分差值很小；而相异部分差值较大，差值图像中相异部分突显，可用于检测同一场景两幅图像之间的变化，如运动目标检测中的背景减法、视频中镜头边界的检测。

2）去除不需要的叠加性图案

叠加性图案可能是缓慢变化的背景阴影或周期性的噪声，或在图像上每一像素处均已知的附加污染等，如电视制作的蓝屏技术。

3）图像分割

如分割运动的车辆，减法去掉静止部分，剩余的是运动元素和噪声。

4）生成合成图像

编程时，可以用“－”运算符或者 OpenCV 中的函数 subtract 实现图像相减，采用函数 absdiff 求绝对值差图像，调用格式如下：

```
cv.absdiff(src1, src2[, dst]) -> dst
cv.subtract(src1, src2[, dst[, mask[, dtype]]]) -> dst
```

【例 3.14】 编程实现两幅图像相减。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Back = cv.imread('hallback.bmp')
Foreground = cv.imread('hallforeground.bmp')
Back, Foreground = Back / 255, Foreground / 255
result1 = cv.absdiff(Back, Foreground)
result2 = Foreground - Back
show = cv.hconcat((Back, Foreground, result1, result2))
cv.imshow("Image subtraction", show)
cv.waitKey()
```

程序运行结果如图 3-16 所示。

(a) 背景图　(b) 前景图　(c) 差的绝对值图像　(d) 差值图像

图 3-16　减法运算结果图

3.5.3　乘法运算

乘法运算将两幅或多幅图像对应点像素值相乘，如式(3-28)所示。

$$g(x,y)=f_1(x,y)\times f_2(x,y) \tag{3-28}$$

式中，f_1、f_2 是同等大小的两幅图像。

乘法运算主要用于图像的局部显示和提取，通常采用二值模板图像与原图像做乘法来实现；也可以用来生成合成图像。

可以用矩阵点乘实现两幅图像相乘，也可以采用 OpenCV 的 multiply 函数，其调用格式如下：

```
cv.multiply(src1, src2[, dst[, scale[, dtype]]]) -> dst
```

参数 src1 和 src2 具有相同大小和数据类型，scale 是像素乘积的比例因子。

【例 3.15】 编程将两幅图像相乘，实现目标提取。

解：程序如下。

```
import cv2 as cv
import numpy as np
Back = cv.imread('bird.jpg')
Foreground = cv.imread('birdtemplet.bmp')
Back, Foreground = Back / 255, Foreground / 255    #8 位数据转换为 0～1 的浮点数,避免溢出
result1 = cv.multiply(Back, Foreground)
result2 = Foreground * Back                        #矩阵点乘
show = cv.hconcat((Back, Foreground, result1, result2))
cv.imshow("Image multiplication", show)
cv.waitKey()
```

程序运行结果如图 3-17 所示。

(a) 背景图　(b) 模板　(c) 图像相乘　(d) 矩阵点乘

图 3-17　乘法运算结果图

3.5.4　除法运算

除法运算将两幅或多幅图像对应点像素值相除，如式(3-29)所示。

$$g(x,y)=f_1(x,y)\div f_2(x,y) \tag{3-29}$$

式中，f_1、f_2 是同等大小的两幅图像。

除法运算可以用于消除空间可变的量化敏感函数、归一化、产生比率图像等。

可以用矩阵点除实现图像除法，也可以采用 OpenCV 的 divide 函数，其调用格式如下：

```
cv.divide(src1, src2[, dst[, scale[, dtype]]]) -> dst
cv.divide(scale, src2[, dst[, dtype]]) -> dst
```

缺少参数 src1 时，执行的是参数 scale 除以 src2。

【例 3.16】 设计程序，将两幅图像相除，实现目标提取。

解：程序如下。

```
import cv2 as cv
import numpy as np
Back = cv.imread('bird.jpg')
Templet = cv.imread('birdtemplet.bmp')
Back, Templet, Templet1 = Back / 255, Templet / 255, Templet / 255
Templet1[Templet1 != 1] = 1000              #模板图像中非模板区值设为较大值 1000
result1 = cv.divide(Back, Templet1)         #图像相除
```

```
result2 = Back / Templet1                    #矩阵点除
show = cv.hconcat((Back, Templet, result1, result2))
cv.imshow("Image division", show)
cv.waitKey()
```

程序运行结果如图 3-18 所示，可以看出，通过合理设置模板图像，除法也实现了模板运算。

(a) 背景　　(b) 模板　　(c) 图像相除

图 3-18　除法运算结果图

3.5.5　逻辑运算

在两幅图像对应像素间进行与、或、非等运算，称为逻辑运算。

非运算：$g(x,y)=255-f(x,y)$，用于获得原图像的补图像，或称反色。

与运算：$g(x,y)=f_1(x,y)\&f_2(x,y)$，求两幅图像的相交子图，可用于图像的局部显示和提取。

或运算：$g(x,y)=f_1(x,y)\mid f_2(x,y)$，合并两幅图像的子图像，可用于图像的局部显示和提取。

第 4 集
微课视频

OpenCV 中有按位求补 bitwise_not 函数，按位求与 bitwise_and 函数，按位求或 bitwise_or 函数，按位异或 bitwise_xor 函数，调用格式如下：

```
cv.bitwise_not(src[, dst[, mask]]) -> dst
cv.bitwise_and(src1, src2[, dst[, mask]]) -> dst
cv.bitwise_or(src1, src2[, dst[, mask]]) -> dst
cv.bitwise_xor(src1, src2[, dst[, mask]]) -> dst
```

【例 3.17】　对两幅图像按位进行逻辑运算。

解：程序如下。

```
import cv2 as cv
import numpy as np
Back = cv.imread('bird.jpg')
Templet = cv.imread('birdtemplet.bmp')
result1 = cv.bitwise_not(Back)               #求反
result2 = cv.bitwise_and(Back, Templet)      #求与
result3 = cv.bitwise_or(Back, Templet)       #求或
result4 = cv.bitwise_xor(Back, Templet)      #求异或
show = cv.hconcat((result1, result2, result3, result4))
cv.imshow("Logical operation", show)
cv.waitKey()
```

程序运行结果如图 3-19 所示。

关于图像的局部显示与提取，请扫描二维码，查看讲解。

(a) 背景图求反　(b) 与模板相与　(c) 与模板相或　(d) 与模板异或

图 3-19　按位逻辑运算结果图

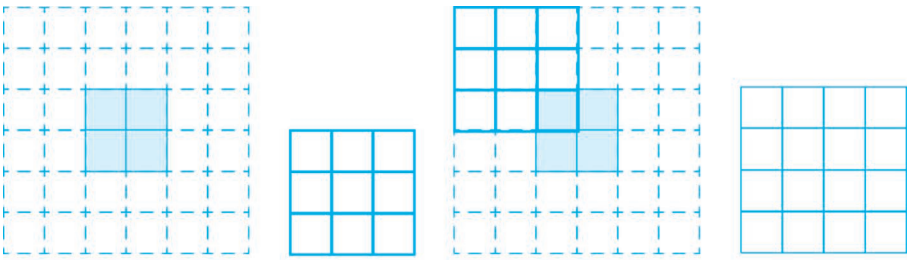
3.6　上采样和下采样

上采样指增大图像的分辨率，下采样指降低图像的分辨率，常用于生成图像的多分辨率表达，即同一幅图像采用不同大小的分辨率表示，以便实现不同尺度的特征提取，或者用于搜索算法。

上、下采样有不同的实现方法，可以采用图像缩放变换实现，即将新分辨率图像像素映射到原分辨率图像中，通过插值运算，获取像素值，生成新图像。

上采样也可以通过反卷积的方式实现。通过合理的方式将低分辨率图像补零，再进行卷积，提高分辨率。不同的补零方式，对应不同的反卷积。

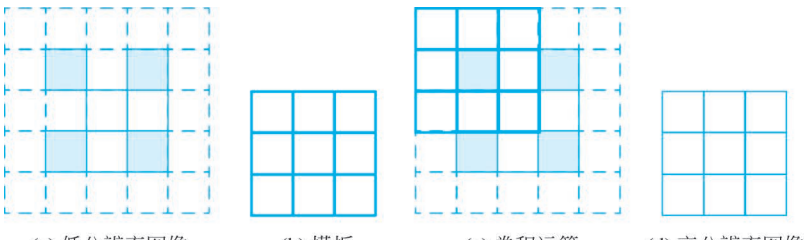
图 3-20(a)为 2×2 的低分辨率图像，在周围补两行两列的零，和图 3-20(b)所示模板进行卷积运算，如图 3-20(c)所示，得 4×4 的高分辨率图像，如图 3-20(d)所示，这种方法称为转置卷积。



(a) 低分辨率图像　(b) 模板　(c) 卷积运算　(d) 高分辨率图像

图 3-20　转置卷积

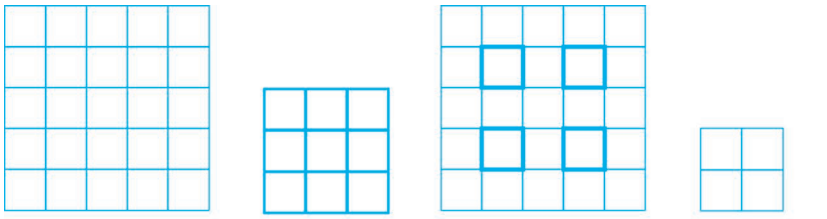
图 3-21(a)为 2×2 的低分辨率图像，在原图像素周围补零，并和模板进行卷积运算，如图 3-21(c)所示，得 3×3 的高分辨率图像，如图 3-21(d)所示，这种方法称为微步卷积。



(a) 低分辨率图像　(b) 模板　(c) 卷积运算　(d) 高分辨率图像

图 3-21　微步卷积

下采样也可以仅在某些像素处计算卷积，降低分辨率，如图 3-22 所示。



(a) 高分辨率图像　(b) 模板　(c) 计算卷积像素　(d) 低分辨率图像

图 3-22　下采样

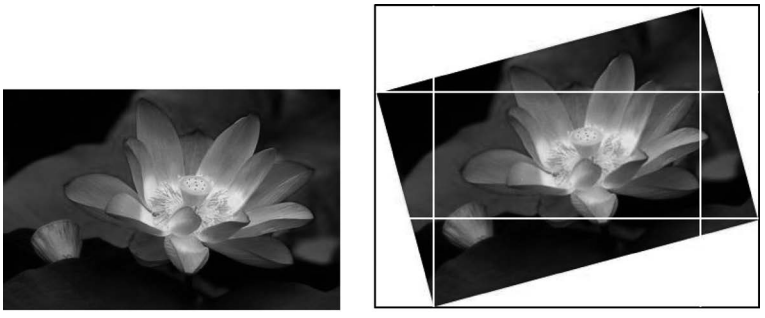
上、下采样方法很多，操作灵活，需要根据具体设计目的选择合适的方法。

3.7　综合实例

【例 3.18】　编写程序，将一幅图像连续旋转、显示，图像宽高不变且不出现背景色，并将旋转过程保存为 AVI 格式的视频。

解：设计思路如下。

设置旋转角度在一个范围内逐渐变化，即可实现图像的连续旋转。但是，如图 3-23(b)所示，图像旋转后，4 个角落会有填充的背景色。为去除背景像素，将图像放大，把背景色部分外推出图像宽高范围，关键在于放大比例的设置。找出图 3-23(b)中的白线，将中间部分放大到和原图像宽高一致即可去除背景色部分，所以，可以根据中间部分和图像宽高的相对比值设置放大比例。



(a) 原图　(b) 旋转后

图 3-23　图像旋转变换

利用 OpenCV 的 VideoWriter 类实现视频的生成，其构造函数如下：

```
cv.VideoWriter(filename, fourcc, fps, frameSize[, isColor]) -> <VideoWriter object>
```

参数 fourcc 用 4 个字符表示视频编码方式；fps 是帧率；frameSize 指定视频帧尺寸；isColor 如果非零，对彩色帧进行压缩编码。

设计框图如图 3-24 所示。

程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('flower.jpg')
```

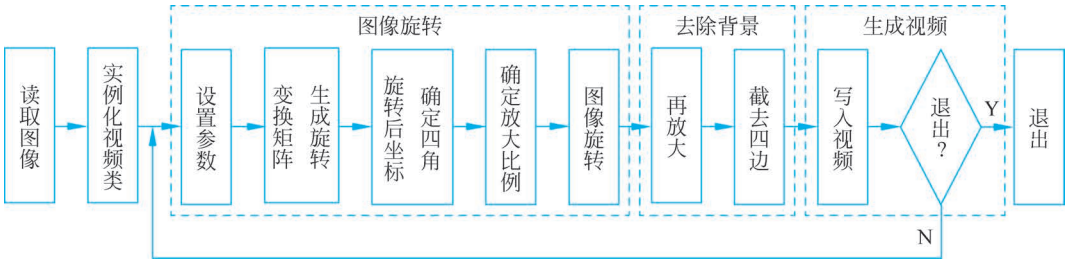


图 3-24　设计框图

```
Image = Image / 255
height, width, color = np.shape(Image)
times, angle_original = 0, 20                                    #times 控制旋转角度变化方向
angle, center = -angle_original, (width // 2, height // 2)
corner = np.array([[0, 0, 1], [width - 1, 0, 1], [width - 1, height - 1, 1],
                   [0, height - 1, 1]])                          #原图四个角的坐标
fourcc = cv.VideoWriter.fourcc( * 'XVID')                        #设置视频参数
out = cv.VideoWriter('rotating.avi', fourcc, 10, (width, height))     #视频类实例化
while 1:
    T = cv.getRotationMatrix2D(center, angle, 1.0)               #生成当前角度对应的旋转变换矩阵
    corner_new = np.sort(T @ corner.T)                           #旋转后四个角的坐标
    (w, h) = corner_new[:, 2] - corner_new[:, 1]                 #确定白线中间部分宽和高
    scale = width / w if width / w > height / h else height / h  #确定放大比例
    rotatedI = cv.warpAffine(Image, T, dsize = (width, height), flags = cv.INTER_LINEAR)
    resizedI = cv.resize(rotatedI, dsize = None, fx = scale, fy = scale,
                         interpolation = cv.INTER_LINEAR)        #先旋转再放大
    height_new, width_new, color = np.shape(resizedI)            #新图像尺寸
    rh, rw = height_new // 2, width_new // 2                     #新图像中心
    frame = (resizedI[rh - center[1]:rh + center[1],
             rw - center[0]:rw + center[0], :] * 255).astype(np.uint8)     #截去四边
    if not out.isOpened():
        print('Can not open video!')
        break
    out.write(frame)                                             #作为帧写入视频
    cv.imshow("Rotating", frame)                                 #显示
    key = cv.waitKey(60)
    if key == ord('q') or key == 27:                             #按下"Q"或"Esc"键退出
        break
    if times == 0:                                               #旋转角度逐渐增大
        angle += 1
        if angle == angle_original:                              #角度达到最大正值,反向变化
            times += 1
    else:                                                        #旋转角度逐渐减小
        angle -= 1
        if angle == -angle_original:                             #角度达到最大负值,反向变化
            times -= 1
```

运行程序，显示一幅图像不断旋转的画面，并在当前目录下生成对应的 AVI 格式的视频。

启发：

从图 3-24 的设计框图可以看出，无论多复杂的工程问题，都是由一个个的小问题构成的，将任务分解为若干模块，逐个击破，也就解决了复杂问题。同理，再宏大的目标，都可以一点一点完成。从一点一滴做起，每天进步一点点。

习题

3.1　编写程序，设计不同的模板，对图像进行卷积运算，尝试采用不同的边界填塞方法。

3.2　一幅图像为 $\boldsymbol{f}=\begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{bmatrix}$，设 $k_x=2.3$，$k_y=1.6$，根据几何变换过程编写程序，采用双线性插值法对其进行放大。

3.3　编写程序，对习题 3.2 中的图像逆时针旋转 60°，采用双线性插值法。

3.4　编写程序，实现手机图像编辑中的裁剪功能。

3.5　编写程序，在例 3.14 中的减法运算程序的基础上，从原图中抠取人物。

3.6　编写程序，尝试实现图像的上采样和下采样。