图像处理技术

图像处理技术是计算机视觉的核心,可以看作是有用的关键工具，可以使用它们来完成各种任务。换句话说，图像处理技术是在处理图像时在要记住的东西。因此，如果要进行计算机视觉，对图像处理要有基本的了解。

在本章学习常见图像处理技术。后面三章中补充其他图像处理技术，直方图、阈值技术、轮廓检测和滤波。

在本章中主要内容

- 分割和合并通道
- 图像几何变换，平移、旋转、缩放、仿射变换、透视变换和裁剪
- 带有图像逐位操作(AND, OR, XOR, AND NOT)和掩蔽的算术
- 平滑和锐化技术形态操作
- 颜色空间
- 色图

本章的技术要求如下

Python and OpenCV
A Python-specific IDE
The NumPy and Matplotlib packages A Git client

有关如何安装这些需求的详细信息，请参阅第1章，设置OpenCV。GitHub存储库，使用Python精通OpenCV 4，包含了贯穿本书从第一章到最后一章所需的所有支持项目文件，可以通过以下地址访问:https://github.com/packtpublishing/mastering-opencv-4with-Python。

OpenCV中通道的分割和合并

有时要在多通道图像上使用特定的通道。为此须将多通道图像分割为多个单通道图像。在完成处理之后，希望从不同的单通道图像创建一个多通道图像。为了分割和合并通道，可分别使用cv2.split()和cv2.merge()函数。cv2.split()函数将源多通道图像分割成几个单通道图像。cv2.merge()函数将几个单通道图像合并到一个多通道图像中。

在下一个例子中，splitting_and_merge.py学习如何使用这两个函数。使用cv2.split()函数，从加载的BGR映像获得三个通道，代码如下

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

如果再次从三个通道构建BGR映像,用cv2.merge()函数，代码如下

```
image_copy = cv2.merge((b, g, r))
```

cv2.split()是一个耗时的操作，应只非常必要时才用它。可用NumPy函数处理特定的通道。例如，如果希望获得图像的蓝色通道，可以执行以下操作

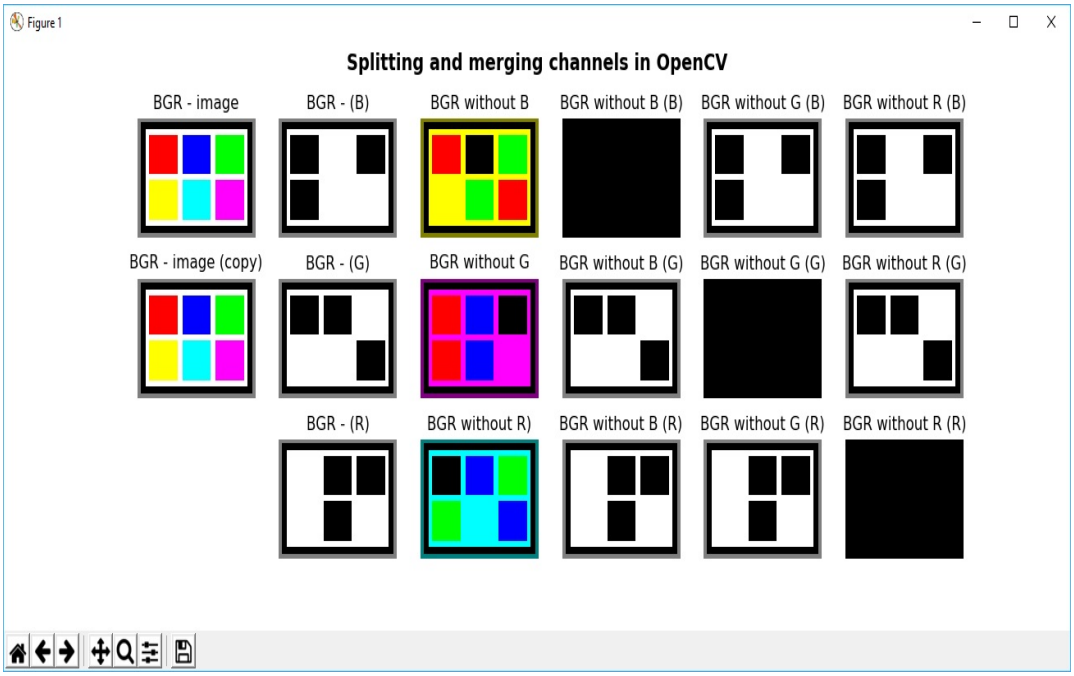
```
b = image[:, :, 0]
```

另外，可消除(设置为0)多通道图像的一些通道。生成的图像将具有相同数量的通道，但在相应的通道中值为0;例如，如果希望消除BGR映像的蓝色通道，可以使用以下代码

```
# We make a copy of the loaded image:
image_without_blue = image.copy()

# From the BGR image, we "eliminate" (set to 0) the blue component (channel 0):
image_without_blue[:, :, 0] = 0
```

如果执行splitting_and_merge.py脚本，您将看到以下屏幕截图



为理解屏幕截图，应记住RGB颜色空间的叠加属性。例如，在没有蓝色的BGR子图中，可看到大部分是黄色的，这是因为绿色和红色值产生黄色。可看到另一个关键特性是黑色子图与设置为0的通道相对应。

图像几何变换

这一节先介绍图像的主要几何变换，包括图像缩放，平移，旋转，仿射变换，透视变换和裁剪等例子。这些几何转换的两个关键函数是cv2.warpAffine()和cv2.warpPerspective()。

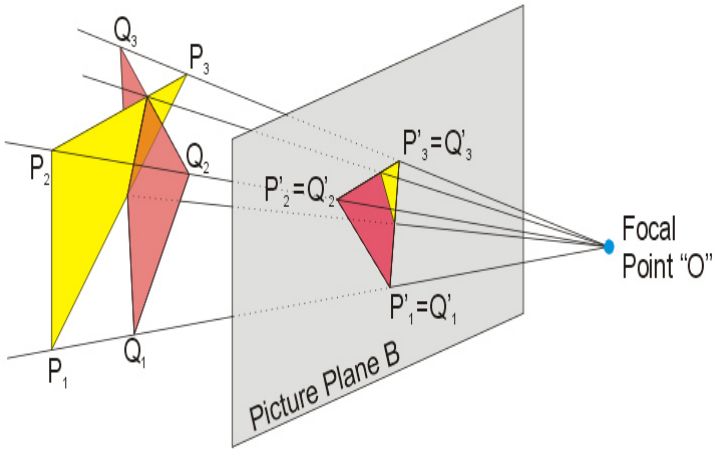
cv2.warpAffine()函数的作用是:用下面的$2 \times 3M$对源图像矩阵变换

$$dst(x,y) = src(M11x + M12y + M13, M21x + M22y + M23)$$

cv2.warpPerspective()函数的作用是:使用下面的3×3变换矩阵转换源图像

$$dst(x,y) = src((M11x + M12y + M13)/(M31x + M32y + M33), (M21x + M22y + M$$

> 变换矩阵
> 仿射变换是空间直角坐标系的变换，从一个二维坐标变换到另一个二维坐标，仿射变换是一个线性变换，他保持了图像的“平行性”和“平直性”，即图像中原来的直线和平行线，变换后仍然保持原来的直线和平行线，仿射变换比较常用的特殊变换有平移(Translation)、缩放（Scale）、翻转（Flip）、旋转（Rotation）和剪切(Shear)。
>
> $$\mathbf{q} = A\mathbf{p} + b$$
>
> $$\begin{bmatrix} \mathbf{q} \\ 1 \end{bmatrix} = \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$
>
> $$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$
>
> 透视变换(Perspective Transformation)是将图片投影到一个新的视平面(Viewing Plane)，也称作投影映射(Projective Mapping)。通用的变换公式为：
>
> $$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$
>
> u, v是原始图片坐标，变换后得到$x, y, x = x'/w', y = y'/w'$
> $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ 称为透视变换矩阵，
> $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$，表示线性变换，如scaling，shearing和ratotion等，$[a_{31}, a_{32}]$用于平移。仿射是透视变换的特殊形式，相对仿射变换来说，改变了直线之间的平行关系。
> 给定透视变换对应的四对像素点坐标，即可求得透视变换矩阵；反之，给定透视变换矩阵，即可对图像或像素点坐标完成透视变换，



在下一小节为常见几何转换技术，geometric_image_transform.py进一步了解这些技术。

缩放图像

缩放图像时可调用cv2.resize()获大小值，根据大小值计算缩放因子(fx和fy)，如下面代码所示

```
resized_image = cv2.resize(image, (width * 2, height * 2), interpolation=cv2.INTER_LINEAR)
```

另一方面，可以同时提供fx和fy值。例如，如果希望将图像缩小2倍，可以使用以下代码

```
dst_image  =  cv2.resize(image,  None,  fx=0.5,  fy=0.5,  interpolation=cv2.INTER_AREA)
```

如果想放大图像，最好的方法是使用cv2.INTER_CUBIC三次插值法(一种耗时的插值方法)或cv2.INTER_LINEAR。缩小图像一般方法是用cv2.INTER_LINEAR。

OpenCV提供的五种插值方法是cv2.INTER_NEAREST最近邻插值，cv2.INTER_LINEAR(双线性插值)，cv2.INTER_AREA(像素区域关系重新采样)，cv2.INTER_CUBIC(三次插值)，cv2.INTER_LANCZOS4(正弦插值)。

移动图像

转换对象需用NumPy数组创建$2x3$浮点值转换矩阵，以像素为单位提供x和y方向的转换，如下面的代码所示

```
M  =  np.float32([[1, 0, x], [0, 1, y]])
```

可得下面M变换矩阵

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

创建这个矩阵之后，调用cv2.warpAffine()函数，如下面代码所示

```
dst_image = cv2.warpAffine(image, M, (width, height))
```

cv2.warpAffine() 函数的作用是:用M矩阵转换源图像。第三个参数(width, height)设置输出图像的大小。

> 记住image.shape返回形状(width, height)

例如，如果想在x方向上转移图像200像素，在y方向上转移图像30像素，用以下方法

```
height, width = image.shape[:2]
M = np.float32([[1, 0, 200], [0, 1, 30]])
dst_image = cv2.warpAffine(image, M, (width, height))
```

注意，移动也可以是负的，如代码所示

```
M = np.float32([[1, 0, -200], [0, 1, -30]])
dst_image = cv2.warpAffine(image, M, (width, height))
```

旋转图像

旋转图像，用cv.getRotationMatrix2d()函数来构建2×3变换矩阵。此矩阵将图像旋转到所需的角度(以度为单位)，其中正值表示逆时针旋转。旋转中心和比例因子也可以调整。以下转换矩阵用这些参量进行计算

$$\begin{bmatrix} \alpha & \beta & (1-\beta)\cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1-\alpha)\cdot center.y \end{bmatrix}$$

该表达式的值为

$$\alpha = scale \cdot \cos\theta, \beta = scale.sin\theta$$

下面例子构建M变换矩阵，使其相对于图像中心旋转180度，缩放系数为1，没有缩放。M矩阵应用于图像旋转，如下所示

```
height, width = image.shape[:2]
M = cv2.getRotationMatrix2D((width / 2.0, height / 2.0), 180, 1)
dst_image = cv2.warpAffine(image, M, (width, height))
```

图象的仿射变换

在仿射变换中，用cv2.getAffineTransform()函数来构建$2x3$变换矩阵，该矩阵将从输入图像和图像变换后的相应坐标中获得。M矩阵传递给cv2.warpAffine()，如下所示

```
pts_1  =  np.float32([[135,  45],  [385,  45],  [135,  230]])
pts_2  =  np.float32([[135,  45],  [385,  45],  [150,  230]])
M    =   cv2.getAffineTransform(pts_1,   pts_2)
dst_image  =  cv2.warpAffine(image_points,  M,  (width,  height))
```

> 仿射变换是一种点、直线和平行不变的变换。变换之后，平行线将保持平行。仿射变换不能同时保持点与点之间的距离和角度不变。

图像的透视变换

为了透视变换(也称为透视图转换)，用cv2. getperspective tivetransform()函数创建$3x3$转换矩阵。这个函数需要四对点,即源图像和输出图像四边形坐标，并从这些点计算透视变换矩阵。M矩阵传递到cv2.warpPerspective()，源图像通过指定大小的矩阵转换，如下面的代码所示

```
pts_1  =  np.float32([[450,  65],  [517,  65],  [431,  164],  [552,  164]])
pts_2 = np.float32([[0, 0], [300, 0], [0, 300], [300, 300]])
M    =   cv2.getPerspectiveTransform(pts_1,   pts_2)
dst_image  =  cv2.warpPerspective(image,  M,  (300,  300))
```

图像裁剪

为裁剪图像，将使用NumPy切片，如下面的代码所示

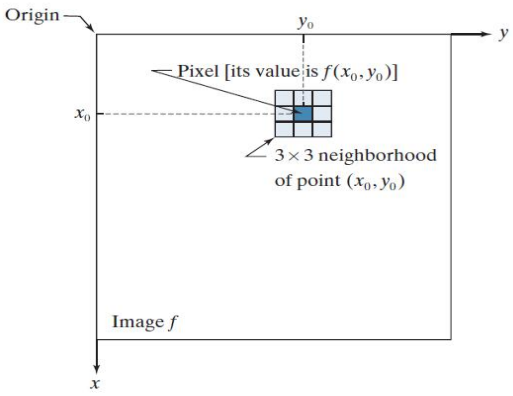
```
dst_image  =  image[80:200,  230:330]
```

前面所述几何转换代码在geometric_image_transform.py中。

附 图像处理

空间域处理

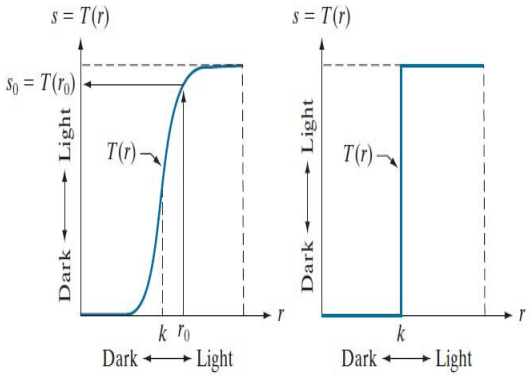
$$g(x,y) = T[f(x,y)]$$



邻域(neighborhood)从一个像素移动到另一个像素，生成输出图像

最小领域 1×1

灰度变换函数，取决于点，$s = T(r)$



例：

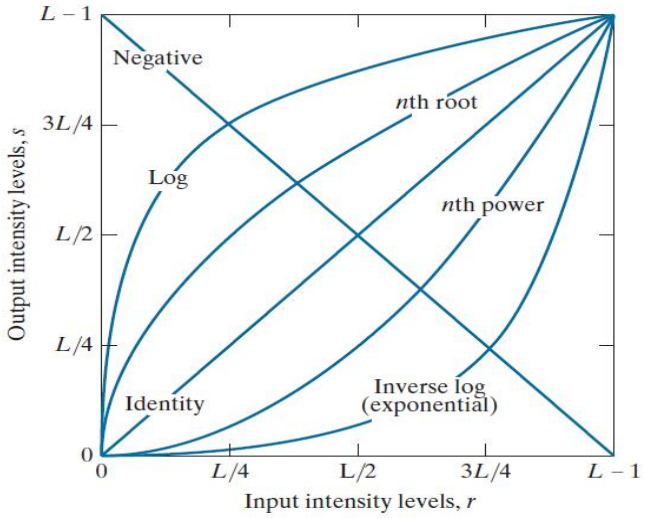
(1) 图像反转。在灰度范围$[0, L-1]$反转

$$s = L-1-r$$

(2) 对数变换

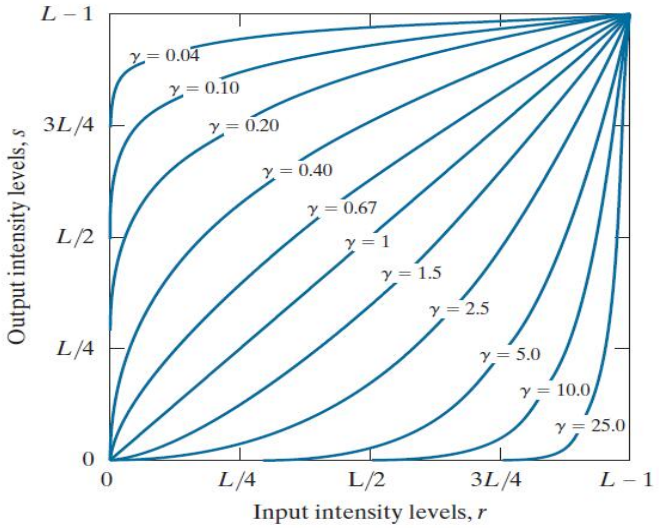
$$s = c \cdot log(1 + r)$$

扩展图像中低灰度像素值，压缩更高灰度像素值



(3)幂变换 （伽马）

$$s = c \cdot r^{\gamma}$$



c=1, γ=3. 0, 4. 0, 5. 0

（4）直方图处理

直方图是数据的统计,图形表示。 图像直方图表示数字图像中亮度分布的直方图。灰度直方图是一幅图像中个像素灰度值出现次数或频数的统计结果，反映该图像中灰度值出现的频率
，描述：概率密度函数(PDE)。

补充：空间滤波

图像滤波指在尽量保留图像细节特征的条件下对目标图像的噪声进行抑制。

（1）不能损坏图像的重要特征信息（如轮廓和边缘等）；
（2）图像经滤波处理后清晰度更高；
（3）抽出图像的特征作为图像识别的特征模式；

图像噪声的来源

图像噪声往往在图像上表现为掩盖图像有效信息的像素点或小块，更好的理解就是，噪声让图像变得不清晰了。那么，图像的噪声是怎么产生的呢？图像的噪声主要是来自两个过程：图像采集过程和图像传输过程。
(1)图像采集过程：在机器视觉系统获取图像信息时，我们需要用到图像传感器来采集图像，如CCD相机，但由于图像传感器受到制造材料、制造工艺、加工精度、结构设计和材料质量等影响，使得图像传感器在采集图像时引入图像噪声。
(2)图像传输过程：在机器视觉系统进行图像传输时，由于图像的传输介质和接收设备等的不理想，导致图像在其传输和接收过程中引入了噪声。其次，当实际输入图像与定义输入图像不统一时也会引入图像噪声。

- spatial filters,核，模

核其实是一组权重，决定了如何利用某一个点周围的像素点来计算新的像素点，核也称为卷积矩阵，对一个区域的像素做调和或者卷积运算，通常基于核的滤波器被称为卷积滤波器。

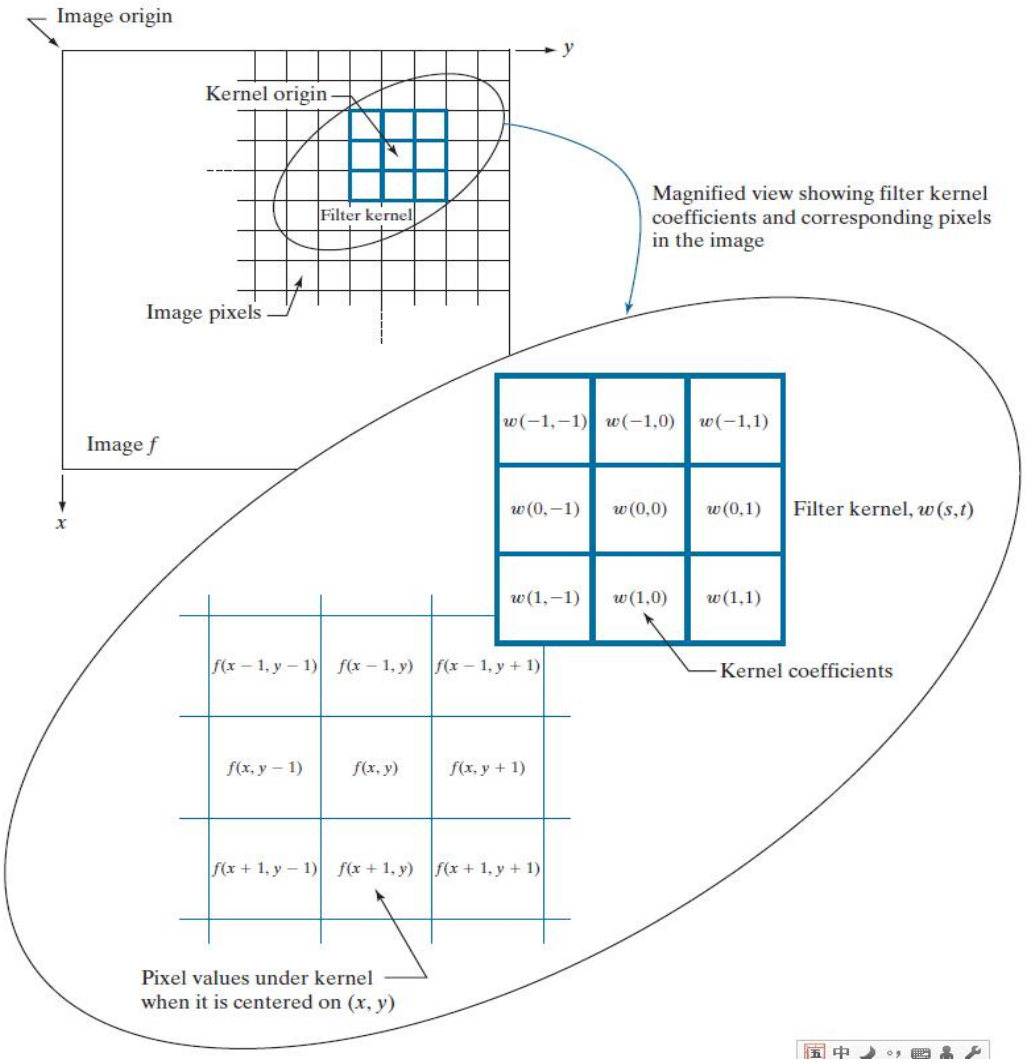
滤波操作就是图像对应像素与核的乘积之和。当核中心位置移动到图像(i,j)像素位置时，图像(i,j)位置像素称为锚点。要对原始图像的边缘进行某种方式的填充（一般为0填充）。

> 线性滤波

线性邻域滤波是一种常用的邻域算子；邻域算子利用给定像素周围的像素值的决定此像素的最终输出值的一种算子。

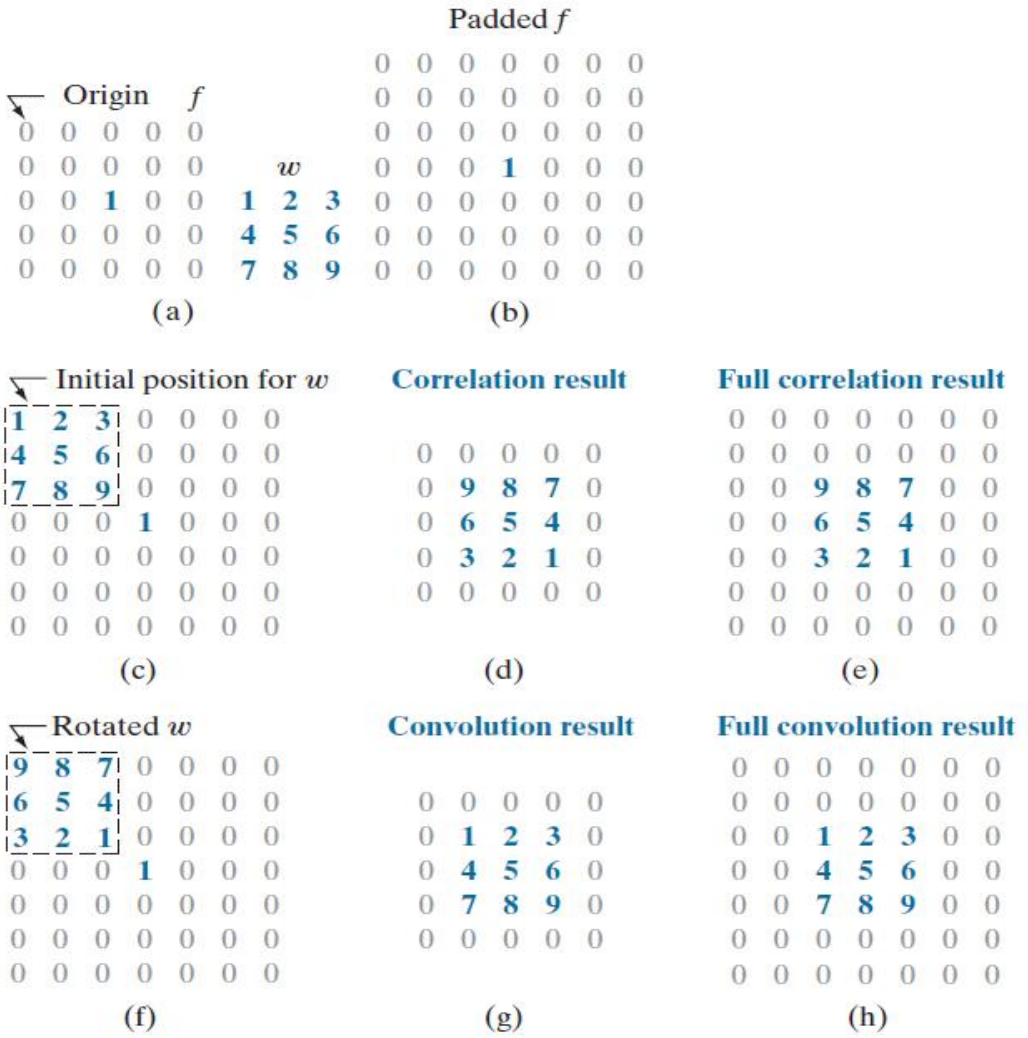
$$g(x,y)=\sum_{s=-a}^{a}\sum_{s=-b}^{b}w(s,t)f(x+s,y+t)$$

从左向右，从上向下移动



卷积与此相似，但要先旋转180度

$$g(x,y) = \sum_{s=-a}^{a} \sum_{s=-b}^{b} w(s,t) f(x-s, y-t)$$



OpenCV中的filter2D()函数，可以运用由用户指定的任意核来计算。通常核是一个二位数组，特征是奇数行、奇数列，中心位置对应着感兴趣的像素，数组每一个元素为整数或者浮点数，相对应值的大小对应其权重。

例：如果感兴趣区域像素值权重为9，相邻区域权重为-1，滤波操作则为将中心像素值乘以9，减去周围所有的像素，如果感兴趣的像素与周围像素存在差异，则该差异会被放大，达到了锐化的目的。

图像滤波

这一节中将用过滤器和定制的内核处理模糊和锐化图像。还将介绍用于执行其他图像处理功能的通用内核。

任意内核

OpenCV提供了cv2.filter2D()函数，将任意内核应用于图像，将图像与内核进行卷积运算。了解这个函数是如何工作的，应先构建内核，一个$5x5$的内核，如下面的代码所示

```
kernel_averaging_5_5 = np.array([[0.04, 0.04, 0.04, 0.04, 0.04], [0.04, 0.04, 0.04, 0.04, 0.04],
```

这相当于一个5x5平均的核函数。还可以这样创建核

```
# kernel_averaging_5_5 = np.ones((5, 5), np.float32)/25
kernel_averaging_5_5 = np.array([[0.04, 0.04, 0.04, 0.04, 0.04],
                                 [0.04, 0.04, 0.04, 0.04, 0.04],
                                 [0.04, 0.04, 0.04, 0.04, 0.04],
                                 [0.04, 0.04, 0.04, 0.04, 0.04],
                                 [0.04, 0.04, 0.04, 0.04, 0.04]])
```

通过调用上述函数将核应用到源映像，如下面的代码所示

```
smooth_image_f2D = cv2.filter2D(image, -1, kernel_averaging_5_5)
```

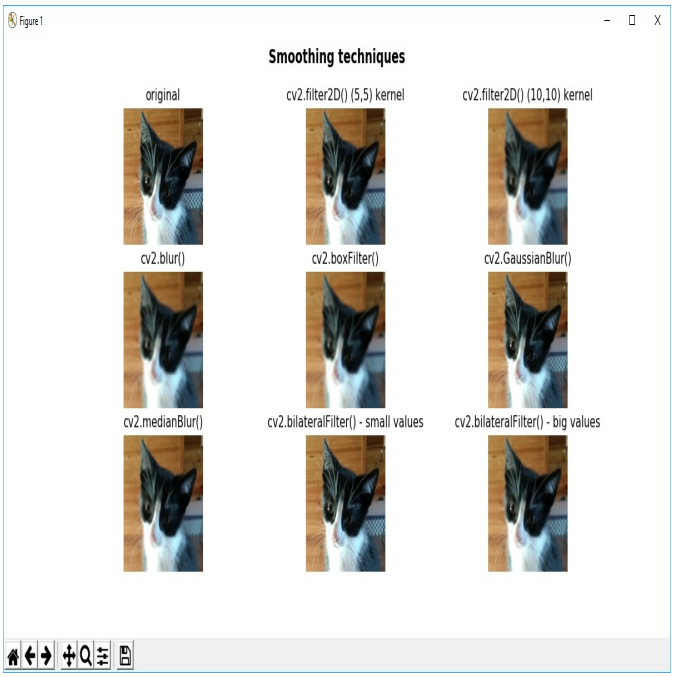
现在我们已经看到了将任意核应用于图像的方法。前面例子创建了一个平均核平滑图像。还有其他方法可以在不创建核的情况下执行图像平滑，也称为图像模糊。可为相应的OpenCV函数提供一些其他参数。在smoothing_technique.py脚本中，可看到前一个示例和下一节的完整代码。

平滑图像

> 补充
>
> 空间滤波增强目的可分为：平滑滤波和锐化滤波。
> 根据空间滤波的特点可分为：线性滤波和非线性滤波。
>
> 方框滤波 -> boxblur函数来实现 ->线性滤波
> 均值滤波（邻域平均滤波） -> blur函数 ->线性滤波
> 高斯滤波 ->GaussianBlur函数 ->线性滤波
> 中值滤波 ->medianBlur函数 ->非线性滤波
> 双边滤波 ->bilateralFilter函数 ->非线性滤波

如前所述，在smoothing_technique.py中，将看到常见的平滑滤波技术。平滑技术通常用来降低噪声，可用于降低低分辨率图像中的像素化效果。这些技术的注释如下。

屏幕截图中看到这个脚本的输出



在前面的屏幕截图中，您可以看到在图像处理中应用通用内核的效果。

平均滤波器

可用cv2.blur()和cv2.boxFilter()通过将图像与内核进行卷积来执行平均滤波，在cv2.boxFilter()的情况下，这可能是不规范化的。它们只是取内核区域下所有像素的平均值，然后用这个平均值替换中心元素。可以控制内核大小和锚内核(默认情况下(-1，-1)，这意味着锚位于内核中心)。当cv2.boxFilter()的normalize参数(默认为True)等于True时，两个函数执行相同的操作。通过这种方式，两个函数都使用内核平滑图像，如下面的表达式所示

$$\begin{bmatrix} 1 & 1 & 1\cdots & 1 & 1 \\ 1 & 1 & 1\cdots & 1 & 1 \\ \cdots & & & & \\ 1 & 1 & 1\cdots & 1 & 1 \end{bmatrix}$$

在cv2.boxFilter()函数的情况下

$$\alpha = ksize.width * heighwhennormalize = true; 1otherwise$$

在cv2.blur()函数的情况下

$$\alpha = ksize.width * heigh$$

换句话说，cv2.blur()用规范化的方框滤波器，如下面的代码所示

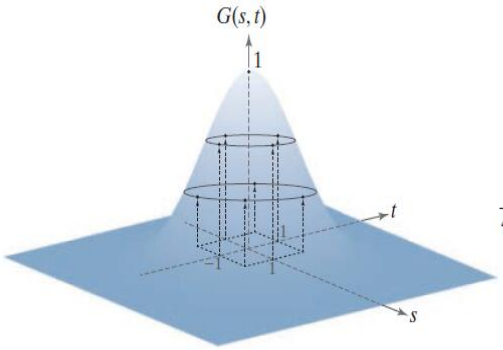
```
# The function cv2.blur() smooths an image using the normalized box filter
smooth_image_b = cv2.blur(image, (10, 10))

# When the parameter normalize (by default True) of cv2.boxFilter() is equals to True,
# cv2.filter2D() and cv2.boxFilter() perform the same operation:
smooth_image_bfi = cv2.boxFilter(image, -1, (10, 10), normalize=True)
```

在前面的代码中，两行代码是等价的。

高斯滤波

OpenCV提供了cv2.GaussianBlur()函数，该函数用高斯核使图像模糊。该内核可用以下参数进行控制:ksize(内核大小)、sigmaX(高斯核在x方向上的标准差)和sigmaY(高斯核在y方向上的标准差)。为了知道用了哪个内核，可调用cv2.getGaussianKernel()函数。

> 补充
>
> 高斯滤波是一种线性平滑滤波，对于抑制服从正态分布的噪声非常有效的。接近边缘处无效。
>
> 
>
>
> $\frac{1}{4.8976}\times$
>
0.3679	0.6065	0.3679
0.6065	1.0000	0.6065
0.3679	0.6065	0.3679
>
> 高斯核:
>
> $$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{s^2+t^2}{2\sigma^2}}$$
>
> 小数类型。整数类型：将得到的值进行归一化处理，加系数。。
>
> $$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$$

> 高斯噪音：给每一个像素点加一个随机的符合高斯分布的值。

```
from skimage import io
import random
import numpy as np

def gauss_noise(image):
    img = image.astype(np.int16)#此步是为了避免像素点小于0，大于255的情况
    mu =0
    sigma = 10
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            for k in range(img.shape[2]):
                img[i,j,k] = img[i,j,k] +random.gauss(mu=mu,sigma=sigma)
    img[img>255] = 255
    img[img<0] = 0
    img = img.astype(np.uint8)
    return img
```

例如，在下面的代码行中，cv2.GaussianBlur()用大小为(9,9)的高斯核使图像变得模糊

```
# The function cv2.GaussianBlur() convolves the source image with the specified Gaussian kernel
# This kernel can be controlled using the parameters ksize (kernel size),
# sigmaX(standard deviation in the x direction of the gaussian kernel) and
# sigmaY (standard deviation in the y direction of the gaussian kernel)
smooth_image_gb = cv2.GaussianBlur(image, (9, 9), 0)
```

中值滤波

> 补充
>
> 中值滤波法:将每一像素点的灰度值设置为该点某邻域窗口内的所有像素点灰度值的中值，不是平均值。中值滤波通过选择中间值避免图像孤立噪声点的影响，对脉冲噪声、斑点噪声、椒盐噪声有良好的滤除作用，特别是在滤除噪声的同时，能够保护信号的边缘，使之不被模糊。

OpenCV的cv2.medianBlur()函数，用中位数内核模糊图像，如下面的代码所示

```
# The function cv2.medianBlur(), which blurs the image with a median kernel:
smooth_image_mb = cv2.medianBlur(image, 9)
```

该滤波器可用于降低图像的椒盐噪声。

> 补充：椒盐噪音：简单点说就是在图像中随机加一些白点或者黑点

```
import random
import numpy as np
def salt_and_pepper_noise(img, proportion=0.05):
    noise_img =img
    height,width =noise_img.shape[0],noise_img.shape[1]
    num = int(height*width*proportion)#多少个像素点添加椒盐噪声
    for i in range(num):
        w = random.randint(0,width-1)
        h = random.randint(0,height-1)
        if random.randint(0,1) ==0:
            noise_img[h,w] =0
        else:
            noise_img[h,w] = 255
```

双边滤波

> 补充
>
> 双边滤波（Bilateral filter）是一种非线性的滤波方法，是结合图像的空间邻近度和像素值相似度的一种折衷处理，同时考虑空域信息和灰度相似性，达到保边去噪的目的。双边滤波器比高斯滤波多了一个高斯方差，它是基于空间分布的高斯滤波函数，所以在边缘附近，离的较远的像素不会太多影响到边缘上的像素值，这样就保证了边缘附近像素值的保存。

可以将cv2.bilateralFilter()函数应用于输入图像，以便应用双边滤波器。此函数降低噪声，同时保持边缘锐利，如下面的代码所示

```
smooth_image_bf = cv2.bilateralFilter(image, 5, 10, 10)
```

需要注意的是，之前所有的过滤器都倾向于平滑所有的图像，包括它们的边缘。

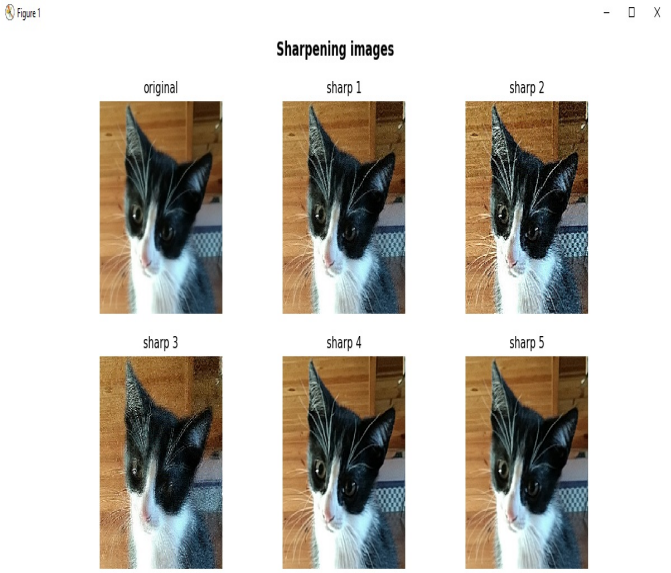
锐化图像

锐化滤波：减弱或消除图像中的低频分量，但不影响高频分量。低频分量对应图像中灰度值缓慢变化的区域，与图像的整体特性如整体对比度和平均灰度值等有关。锐化滤波将这些分量滤去可使图像反差增加，边缘明显。

与最后这个函数相关，可以尝试使用选项来锐化图像的边缘。一种简单的方法是执行非锐化掩蔽，即从原始图像中减去图像的非锐化或平滑版本。在下面的例子中，首先应用高斯平滑滤波器，然后从原始图像中减去得到的图像

```
smoothed = cv2.GaussianBlur(img, (9, 9), 10)
unsharped = cv2.addWeighted(img, 1.5, smoothed, -0.5, 0)
```

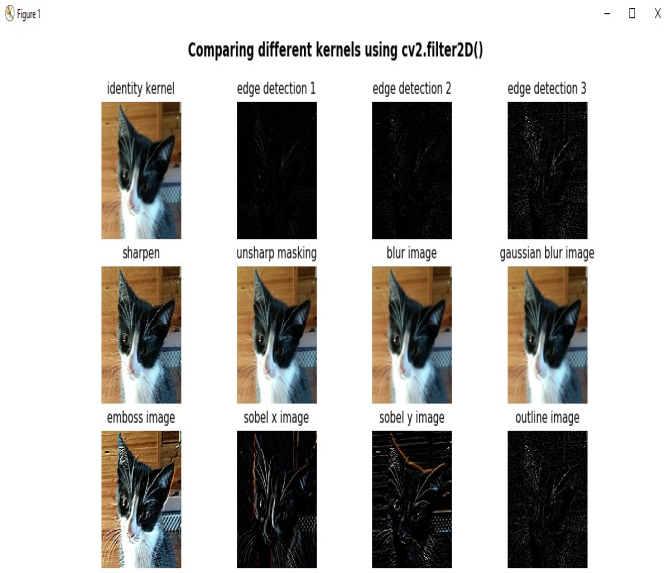
另一个选项是使用特定的内核来锐化边缘，再调用cv2.filter2D()函数。在sharpening_technique.py脚本中，有一些定义好的核可以应用于此目的。这个脚本的输出如下面的屏幕截图所示



在前面的屏幕截图中，可看到应用不同锐化内核的效果，这可以在sharpening_technique.py脚本中看到。

图像处理中的通用内核

我们已经看到内核对生成的图像有很大的影响。在filter_2d_kernel.py脚本定义一些用于不同目的的通用内核，其中包括边缘检测、平滑、锐化或浮雕。提醒一下，为了应用特定的内核，应该使用cv2.filter2D()函数。输出如下



使用cv2.filter2D()函数可以看到应用不同内核的效果，该函数可用于应用特定的内核。

创建cartoonized图像

如前所述，可用cv2.bilateralFilter()来降低噪声，同时保留锐边。然而，这种滤波器在滤波后的图像中可能产生强度稳定(阶梯效应)和假边缘(梯度反转)。图像过滤(有几个改进的双边过滤器处理这些工件)，可考虑创建非常酷的卡通化的图像。完整的代码可以在cartoon.py中看到，本节中一个简短的描述。

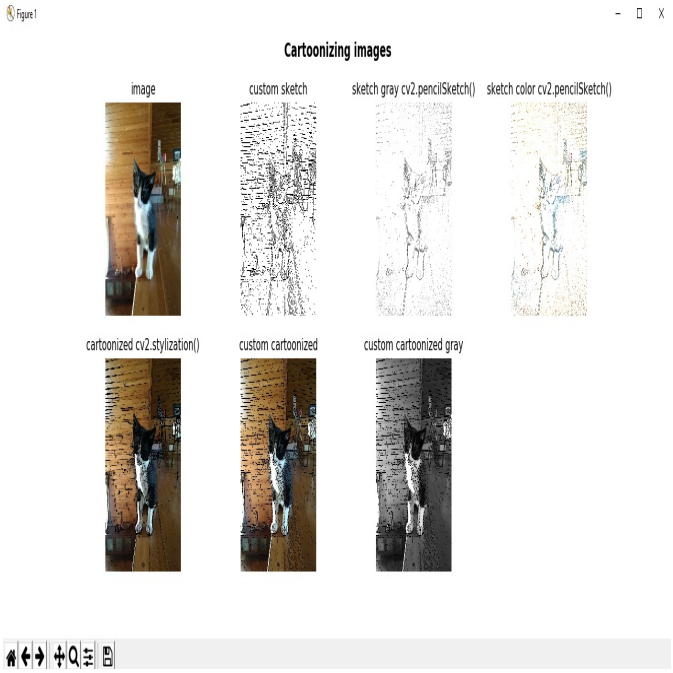
卡通化图像的过程非常简单，它是在cartonize_image()中执行的。首先，构造图像的草图(参见sketch_image()函数)，该函数基于图像的边缘。还可以使用其他边缘检测器，在本例中，用是拉普拉斯算子。在调用cv2.Laplacian()函数之前，通过使用cv2.medianBlur()中值滤波器平滑图像来降低噪声。得到边缘后，应用cv2.threshold()对得到的图像进行阈值处理。将在下一章讨论阈值化技术。在本例中，函数给出了给定灰度图像的二进制图像，该图像对应于sketch_image()函数的输出。可使用阈值(在本例中固定为70)来查看该值如何控制结果图像中出现的黑色像素(与检测到的边缘相对应)的数量。如果这个值很小(例如，10)，则会出现许多黑色边框像素。如果这个值很大(例如200)，输出的黑色边框像素就很少。为了获得卡通化效果，调用具有大值的cv2. bilateralfilter()函数,如cv2.bilateralFilter(img, 10,250, 250))。最后一步是使用cv2.bitwise_and()将草图图像和双边过滤器的输出放在一起，并使用草图图像作为掩码，以便将这些值设置为输出。如果需要，输出也可以转换成灰度。cv2.bitwise_and()函数按位操作。

> 补充
>
> 拉普拉斯算子 $g(x,y)=f(x+y)+c\nabla^2 f(x,y)$,变换:原来的灰度值加上c倍的二阶偏导数的和。图像模糊:图像受到平均运算或积分运算，逆运算微分运算能够突出图像细节，使图像变得更为清晰。

为了完整起见，OpenCV提供了类似的功能，并在这个脚本中进行了测试。它使用以下过滤器工作

cv2.pencilsketch():这个过滤器生成一个铅笔素描线条图(类似于我们的sketch_image()函数)。
cv2.stylization():这个过滤器可以用于产生各种各样的非真实感效果。在本例中，应用cv2.stylization()来获得卡通化效果(类似于cartonize_image()函数)。

cartoon.py脚本屏幕输出



可以看到，cartonize_image()函数还可以输出一个灰度图像，调用cv2.cvtColor()将图像从BGR转换为灰度。

算法与图像

在本节中，我们将学习一些可以对图像执行的常见算术操作，如按位操作、加、减等。与这些操作相关，需要考虑的一个关键点是饱和算法的概念，在下一小节中解释。

饱和运算

饱和算法是一种算术运算，通过限制运算的最大值和最小值，将运算限制在一个固定的范围内。例如，对图像的某些操作(例如，颜色空间转换、插值技术等)可以产生超出可用范围的值。饱和算法是用来解决这个问题。

例如，为了存储r，它可以是对一个8位图像(值范围从0到255)执行某个操作的结果，应用以下方程

$$I(x,y) = min(max(round(r),0),255)$$

这个概念可以在下面的saturation_algorithm.py脚本中看到

```
x   =   np.uint8([250])
y   =   np.uint8([50])
#  250+50  =  300  =>  255:
result_opencv = cv2.add(x, y)
print("cv2.add(x:'{}' , y:'{}') = '{}'".format(x, y, result_opencv))
#  250+50  =  300  %  256  =  44
result_numpy = x + y
print("x:'{}' + y:'{}' = '{}'".format(x, y, result_numpy))
```

在OpenCV中，超出[0,255]范围的值会切除，称饱和操作。在NumPy中，为模运算。

图像加减

图像的加法和减法可以分别使用cv2.add()和cv2.subtract()函数来执行。这些函数对两个数组的每个元素求和/相减。这些函数也可以用来对数组和标量求和或相减。例如，如果我们想要向图像的所有像素添加60，首先必须使用以下代码构建要添加到原始图像的图像

```
M = np.ones(image.shape, dtype="uint8") * 60
```

Then, we perform the addition, using the following code:
使用以下代码执行加法

```
added_image = cv2.add(image, M)
```

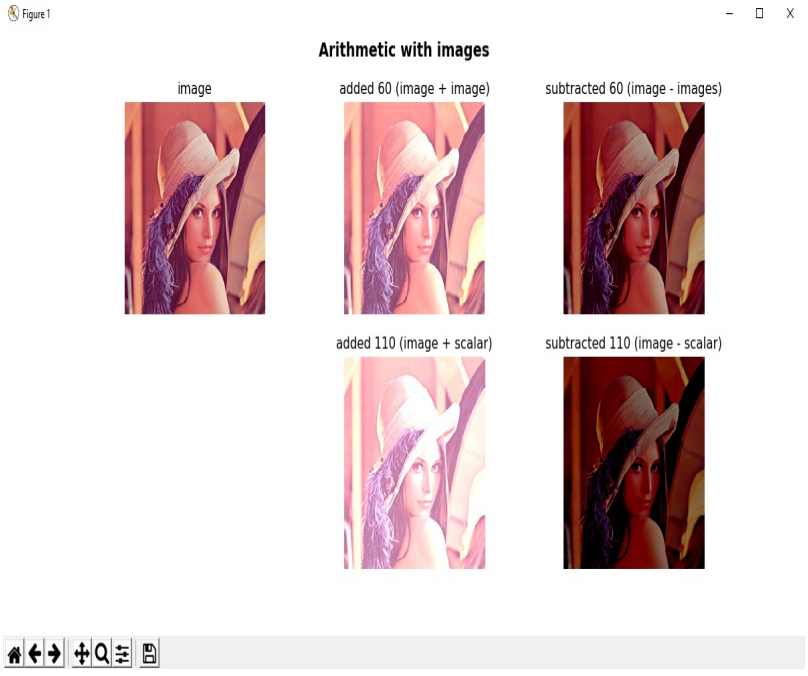
另一种可能性是创建一个标量并将其添加到原始图像中。例如，如果我们想要向图像的所有像素添加110，我们首先必须使用以下代码构建标量

```
scalar = np.ones((1, 3), dtype="float") * 110
```

使用以下代码执行加操作

```
added_image_2 = cv2.add(image, scalar)
```

在减法的情况下，过程是相同的，调用了cv2.subtract()函数。完整代码见arithmetic.py。这个脚本的输出可以在下面的屏幕截图中看到



在前面的屏幕截图中，可清楚地看到加和减预定义值的效果(以两种不同的方式计算，但显示相同的结果)。加一个值时，图像会变亮，减一个值时，图像会变暗。

图像混合

图像混合也是图像加，但给图像不同的权重，给人一种透明的印象。为此，将使用cv2.addWeighted()函数。这个函数通常用于从Sobel算子获取输出。

Sobel算子用于边缘检测，创建一个强调边缘的图像。Sobel运算符使用两个33个内核，它们与原始图像进行卷积，以计算导数的近似值，捕获水平和垂直变化，如下面的代码所示

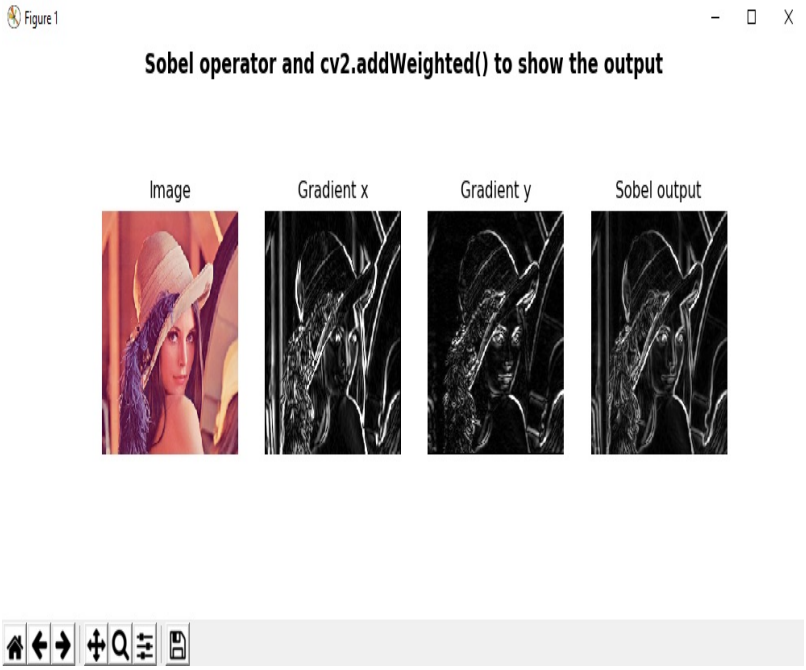
```
# Gradient x is calculated:
# the depth of the output is set to CV_16S to avoid overflow
# CV_16S = one channel of 2-byte signed integers (16-bit signed integers)
gradient_x = cv2.Sobel(gray_image, cv2.CV_16S, 1, 0, 3)
```

因此，在计算了水平和垂直变化之后，可以使用前面提到的函数将它们混合到图像中，如下所示

```
# Conversion to an unsigned 8-bit type:
abs_gradient_x = cv2.convertScaleAbs(gradient_x)
abs_gradient_y = cv2.convertScaleAbs(gradient_y)

# Combine the two images using the same weight:
sobel_image = cv2.addWeighted(abs_gradient_x, 0.5, abs_gradient_y, 0.5, 0)
```

这可以在arithfec_sobel.py脚本中看到。输出见截图



在屏幕截图中，显示了Sobel算子的输出，包括水平和垂直变化。

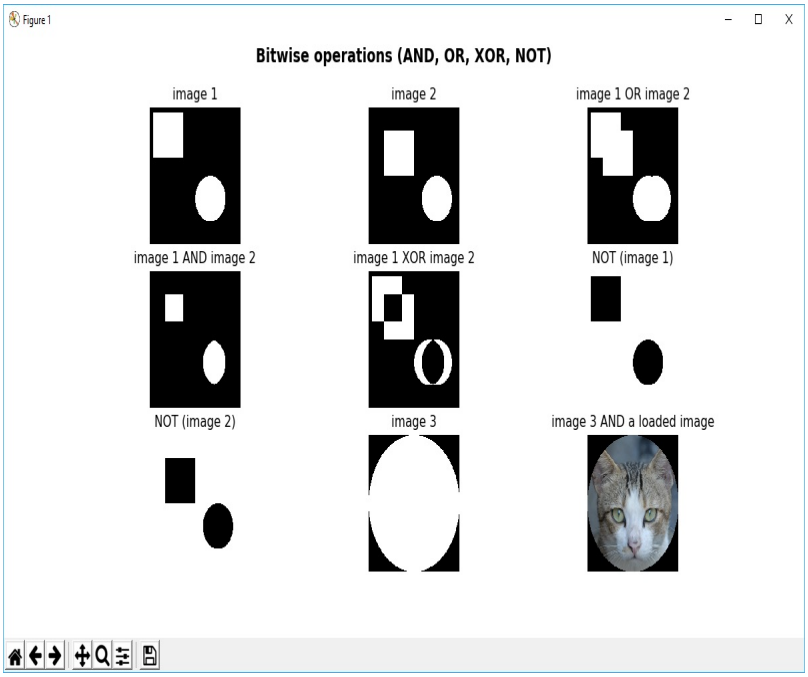
位运算

一些算子可以使用逐位操作在位级别上执行，可用于比较和计算。这些位操作简单，计算速度快，在处理图像时是有用的工具。

位操作包括AND、OR、NOT和XOR

- Bitwise AND: bitwise_and = cv2.bitwise_and(img_1, img_2)
- Bitwise OR: bitwise_xor = cv2.bitwise_xor(img_1, img_2)
- Bitwise XOR: bitwise_xor = cv2.bitwise_xor(img_1, img_2)
- Bitwise NOT: bitwise_not_1 = cv2.bitwise_not(img_1)

为解释这些操作是如何工作的，请看下面屏幕截图中bitwise_operations.py脚本的输出



进一步按位操作，可查看bitwise_operations_images.py，其中加载了两个图像并执行按位操作(and和OR)。应该注意的是，图像应该有相同的形状

```
# Load the original image (250x250):
image = cv2.imread('lenna_250.png')

# Load the binary image (but as a GBR color image - with 3 channels) (250x250):
binary_image = cv2.imread('opencv_binary_logo_250.png')

# Bitwise AND
bitwise_and = cv2.bitwise_and(image, binary_image)

# Bitwise OR
bitwise_or = cv2.bitwise_or(image, binary_image)
```

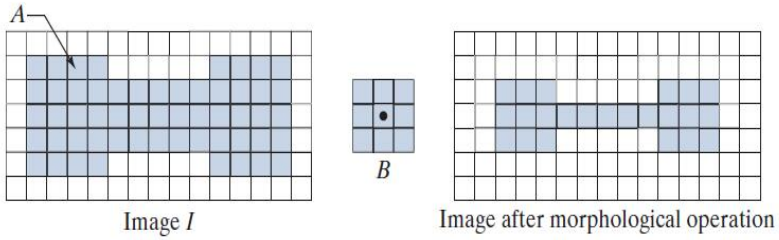
输出可以在下面的屏幕截图中看到

在前面的屏幕截图中，您可以看到执行按位操作(AND，OR)时的结果图像

形态转换

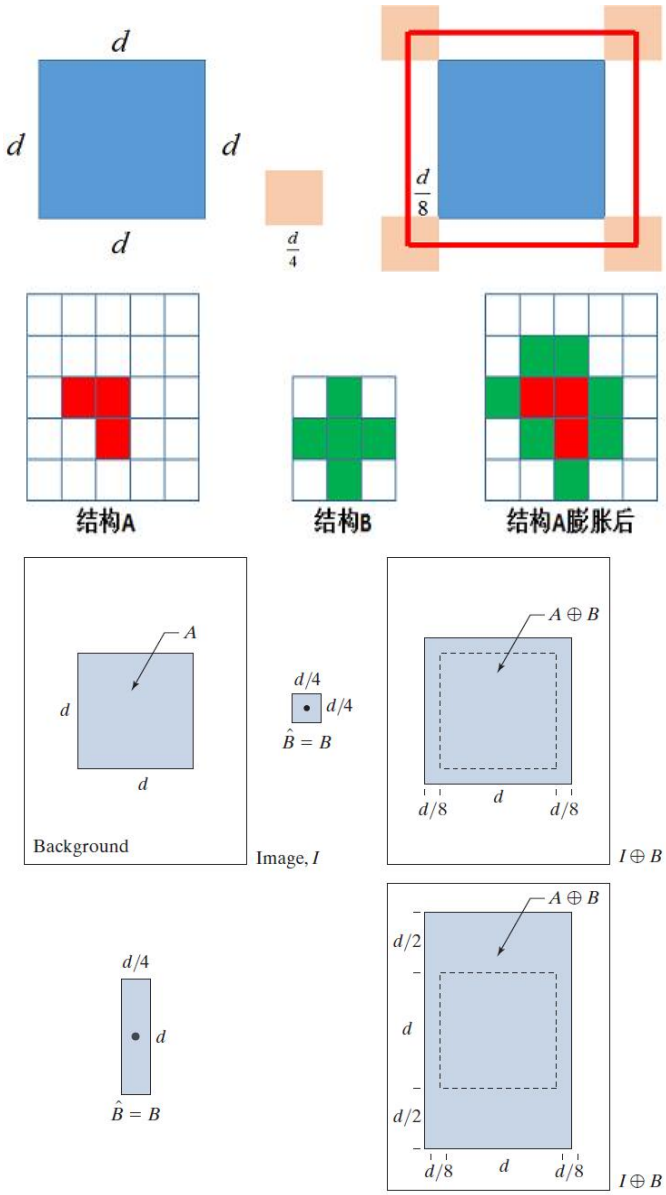
形态学变换是对二值图像进行的基于图像形状的变换。精确的操作是由一个决定操作性质的核结构元素决定的。扩张和侵蚀是形态转换领域的两个基本运算符。另外，开和闭是两个重要的操作，它们是由上述两个操作(扩张和侵蚀)派生出来的。最后，还有其他三种操作，它们基于前面的一些操作之间的差异。

这些形态转换在下面小节中描述，morphological_operations.py脚本将显示对一些测试图像应用这些转换时的输出。重点也将加以评论。



膨胀运算

如果移动结构B的过程中，与结构A存在重叠区域，则记录该位置，所有移动结构B与结构A存在交集的位置的集合为结构A在结构B作用下的膨胀结果。

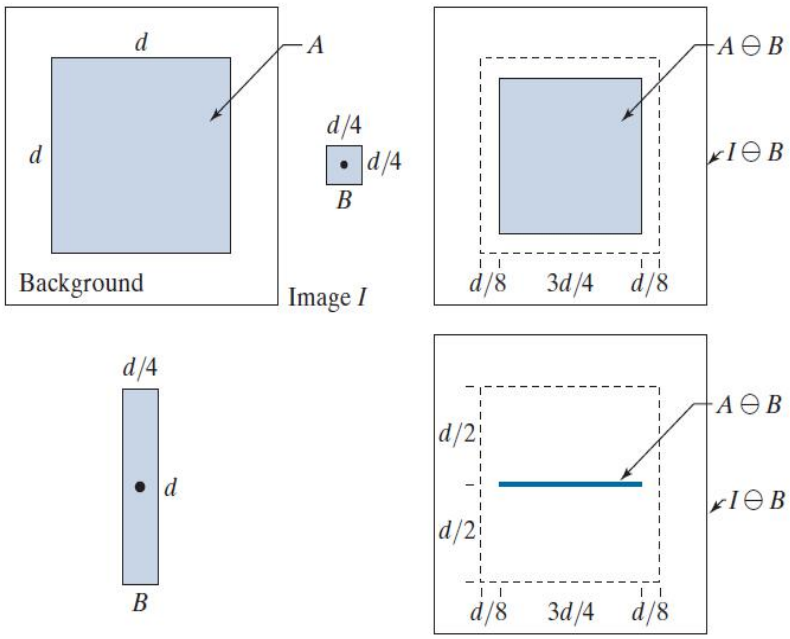


膨胀算子对二值图像的主要作用是逐步扩大前景对象的边界区域。这意味着前景对象的区域会变大，而这些区域中的洞会缩小。操作细节如下面的代码所示

```
dilation = cv2.dilate(image, kernel, iterations=1)
```

侵蚀运算

移动结构B，如果结构B与结构A的交集完全属于结构A的区域内，则保存该位置点，所有满足条件的点构成结构A被结构B腐蚀的结果。

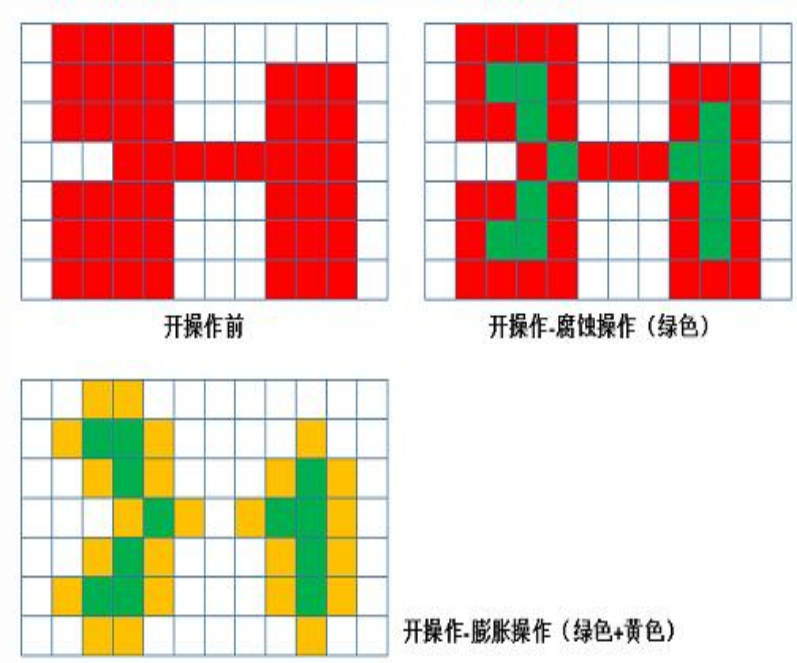


侵蚀运算对二值图像的主要作用是逐渐侵蚀前景物体的边界区域。这意味着前景对象的区域会变小，而这些区域内的洞会变大。您可以在以下代码中看到该操作的详细信息

```
erosion = cv2.erode(image, kernel, iterations=1)
```

开运算

开操作先腐蚀,再使用相同的结构元素(或内核)进后膨胀的操作。通过这种方式，可以应用侵蚀来消除不需要的像素(例如，盐和胡椒的噪声)。具有消除细小物体，在纤细处分离物体和平滑较大物体边界的作用。

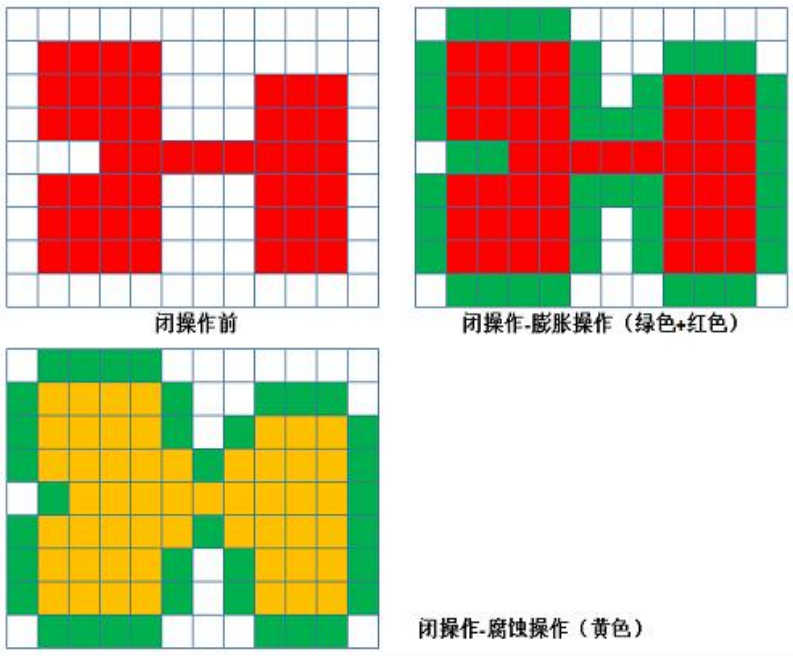


可以在以下代码中看到该操作的详细信息

```
opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
```

闭运算

与之相反，闭合算符可以由侵蚀和膨胀运算得到。在这种情况下，进行膨胀，后侵蚀。具有填充物体内细小空洞，连接邻近物体和平滑边界的作用。膨胀操作会使不需要的像素变大，通过对膨胀后的图像进行侵蚀操作，可以减少这种影响。




```
closing = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
```

形态梯度算子

形态学梯度算子定义为输入图像的扩张和侵蚀之间的差异

```
morph_gradient = cv2.morphologyEx(image, cv2.MORPH_GRADIENT, kernel)
```

顶帽操作

顶帽操作原始图像与进行开运算之后得到的图像的差值。可以在以下代码中看到该操作的详细信息

```
top_hat = cv2.morphologyEx(image, cv2.MORPH_TOPHAT, kernel)
```

黑帽操作

进行闭运算以后得到的图像与原图像的差。您可以在以下代码中看到该操作的详细信息

```
black_hat = cv2.morphologyEx(image, cv2.MORPH_BLACKHAT, kernel)
```

结构元素

关于结构化元素，OpenCV提供了cv2.getStructuringElement()函数。

这个函数输出所需的内核(uint8类型的NumPy数组)。应该向这个函数传递两个参数:内核的形状和大小。OpenCV提供了以下三种形状

Rectangular kernel: cv2.MORPH_RECT
Elliptical kernel: cv2.MORPH_ELLIPSE
Cross-shaped kernel: cv2.MORPH_CROSS

对图像应用形态学变换

在morphological_operations.py脚本中，我们使用不同的内核大小和形状、形态转换和图像。我们将在本节中描述这个脚本的一些关键点。

build_kernel()函数返回要在基于内核类型和大小的形态转换中使用的特定内核。其次，morphological_operations字典包含所有实现的形态学操作。如果我们打印字典，输出将如下

index: '0', key: 'erode', value: '<function erode at 0x0C1F8228>'
index: '1', key: 'dilate', value: '<function dilate at 0x0C1F8390>'
index: '2', key: 'closing', value: '<function closing at 0x0C1F83D8>' index: '3', key: 'opening', value: '<function opening at 0x0C1F8420>'
index: '4', key: 'gradient', value: '<function morphological_gradient at 0x0C1F8468>'
index: '5', key: 'closing|opening', value: '<function closing_and_opening at 0x0C1F8348>'
index: '6', key: 'opening|closing', value: '<function opening_and_closing at 0x0C1F84B0>'

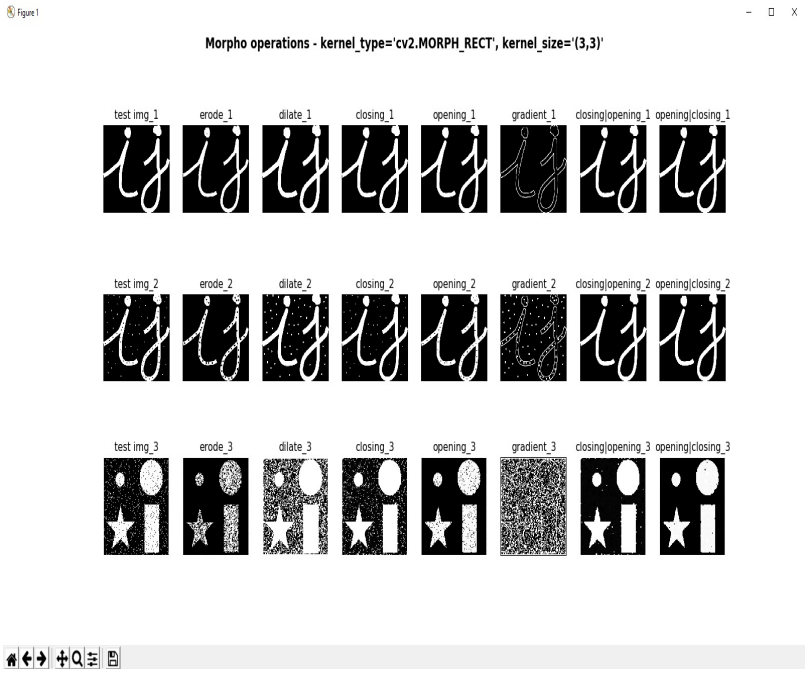
换句话说，字典的键标识要使用的形态学操作，值是使用相应键时要调用的函数。例如，如果我们想调用操作，我们必须执行以下操作

```
result = morphological_operations['erode'](image, kernel_type, kernel_size)
```

在前面的代码图像中，kernel_type和kernel_size是函数的参数(实际上，它们是字典中所有函数的参数)。

apply_morphological_operation()函数将字典中定义的所有形态学操作应用于图像数组。最后，调用show_images()函数，其中绘制了数组中包含的所有图像。具体的实现细节可以在morphological_operations.py脚本的源代码中看到，其中包含大量的注释。

脚本绘制了四个图，其中测试了不同的内核类型和大小。例如，在下面的屏幕截图中，您可以看到使用(3,3)大小的内核和矩形内核(cv2.MORPH_RECT)时的输出



在前面的截图中看到的，形态操作在预处理图像时是一种有用的技术，可去除一些干扰图像正常处理的噪声。此外，形态学操作也可以用来处理图像结构中的缺陷。

颜色空间

在本节中，将介绍流行颜色空间的基础知识。这些颜色空间是RGB、CIE L*ab*、HSL和HSV以及YCbCr。

OpenCV提供了超过150种颜色空间转换方法来执行用户所需的转换。在下面的示例中，将执行从RGB(OpenCV中的BGR)中加载的图像到其他颜色空间(例如，HSV、HLS或YCbCr)的转换。

显示颜色空间

RGB颜色空间是一个附加的颜色空间，其中特定的颜色由红色、绿色和蓝色值表示。人类视觉的工作方式也是类似的，所以这种颜色空间是一种合适的方式来显示计算机图形。

CIELAB颜色空间(也称为CIEL *ab或简称LAB)将特定颜色表示为三个数值，其中L*表示亮度，a*表示绿-红分量，*b*表示蓝-黄分量。这种颜色空间也用于一些图像处理算法。

色相、饱和度、明度(HSL)和色相、饱和度、值(HSV)是两个颜色空间，其中只有一个通道(H)用于描述颜色，这使得指定颜色非常直观。在这些颜色模型中，亮度分量的分离在应用图像处理技术时具有一定的优势

YCbCr是一组用于视频和数字摄影系统的颜色空间，它根据色度分量(Y)和两个色度分量/色度(Cb和Cr)来表示颜色。这种颜色空间在基于YCbCr图像颜色模型的图像分割中非常流行。

在color_spaces.py脚本中，将在BGR颜色空间中加载图像并将其转换为前面提到的颜色空间。在这个脚本中，关键函数是cv2.cvtColor()，它将一个颜色空间的输入图像转换成另一个颜色空间。

在转换到/从RGB颜色空间的情况下，应该显式地指定通道的顺序(BGR或RGB)。例如

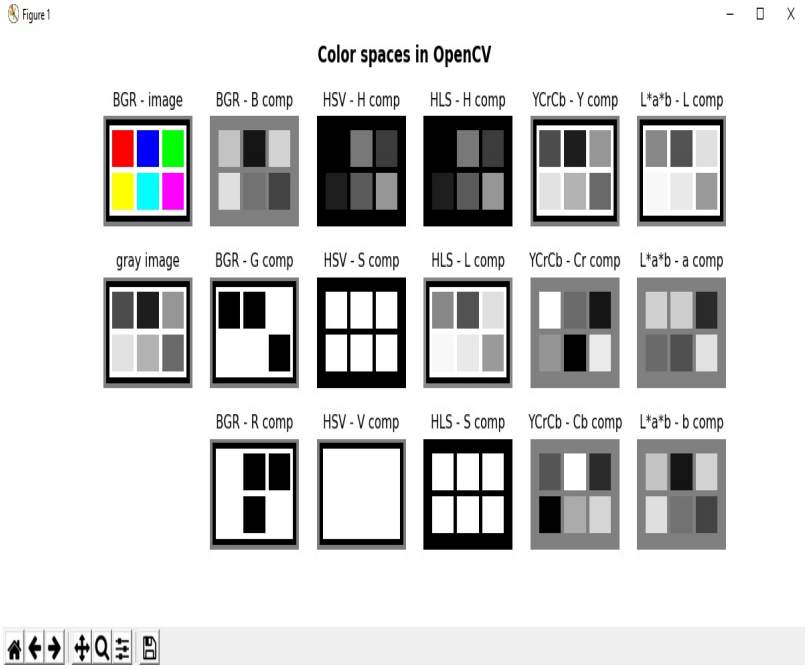
```
image = cv2.imread('color_spaces.png')
```

此图像加载在BGR颜色空间中。因此，如果要将其转换为HSV颜色空间，必须执行以下操作

```
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

注意，我们已经使用了cv2。COLOR_BGR2HSV而不是cv2.COLOR_RGB2HSV

这个脚本的完整代码可以在color_space.py中看到。输出可以在下面的屏幕截图中看到



如上图所示，BGR图像被转换成HSV、HLS、YCrCb和L*ab*颜色空间。每个颜色空间的所有组件(通道)也显示出来了。

皮肤分割在不同的颜色空间

上述颜色空间可以用于不同的图像处理任务和技术。例如，skin_segment.py脚本实现了在不同颜色空间(YCrCb、HSV和RGB)中执行皮肤分割的不同算法。脚本还加载了一些测试图像，以查看这些算法是如何工作的。

这个脚本中的函数是cv2.cvtColor()和cv2.inRange()，检查数组中包含的元素是否位于另外两个数组(下边界数组和上边界数组)的元素之间。

因此，使用cv2.inRange()函数来分割与皮肤对应的颜色。这两个数组(上下边界)定义的值对分割算法的性能起着重要的作用。

RGB-H-CbCr Skin Color Model for Human Face Detection by Nusirwan Anwar, Abdul Rahman, K. C. Wei, and John See Skin segmentation algorithm based on the YCrCb color space by Shruti D Patravali, Jyoti Waykule, and Apurva Katre Face Segmentation Using Skin-Color Map in Videophone Applications by D. Chai and K.N. Ngan

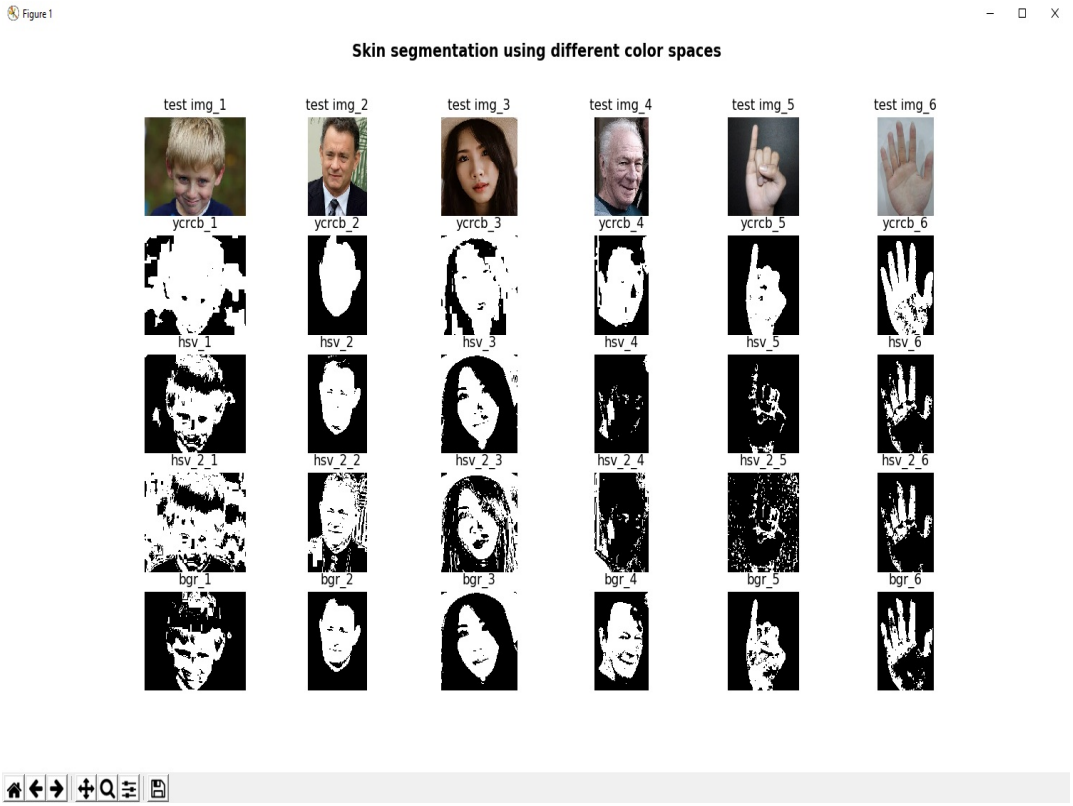
建立了skin_detector字典，将所有的皮肤分割算法应用于测试图像。如果我们打印它，输出将如下

```
index: '0', key: 'ycrcb', value: '<function skin_detector_ycrcb at 0x07B8C030>' index: '1', key:
```

可以看到定义了四个皮肤探测器。为了调用皮肤分割检测器(例如，skin_detector_ycrcb)，执行以下操作

```
detected_skin = skin_detectors['ycrcb'](image)
```

脚本的输出可以在下面的屏幕截图中看到



可看到应用不同的皮肤分割算法的效果，用测试图像，看看这些算法如何在不同的条件下工作。

色图

在许多计算机视觉应用中，算法的输出是灰度图像。人眼并不擅长观察灰度图像的变化，对彩色图像更加敏感，因此常用的方法是将灰度图像转换为伪彩色等值图像。

OpenCV中的色图

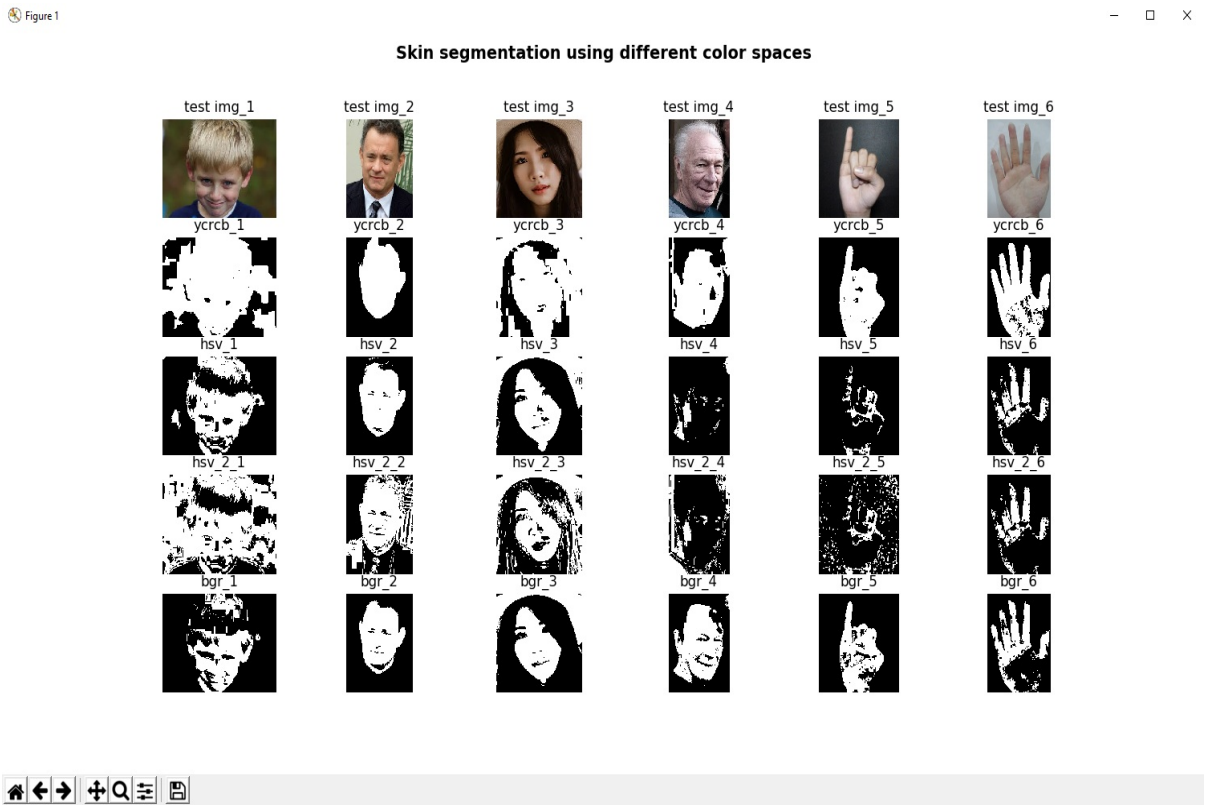
为了执行这个转换，OpenCV有几个颜色映射来增强可视化。applycolormap()函数对给定的图像应用颜色映射。py脚本加载一个灰度图像并应用cv2。COLORMAP_HSV颜色映射，如下面的代码所示

```
img_COLORMAP_HSV = cv2.applyColorMap(gray_img, cv2.COLORMAP_HSV)
```

应用所有的颜色映射到相同的灰度图像，并在相同的图形中绘制。color_map_all.py脚本中看到。OpenCV定义的颜色映射如下所示

```
COLORMAP_AUTUMN = 0
COLORMAP_BONE = 1
COLORMAP_JET = 2
COLORMAP_WINTER = 3
COLORMAP_RAINBOW = 4
COLORMAP_OCEAN = 5
COLORMAP_SUMMER = 6
COLORMAP_SPRING = 7
COLORMAP_COOL = 8
COLORMAP_HSV = 9
COLORMAP_HOT = 11
COLORMAP_PINK = 10
COLORMAP_PARULA = 12
```

screenshot:py脚本将所有这些颜色映射应用于灰度图像。这个脚本的输出可以在下面的屏幕截图中看到



在前面的屏幕截图中，您可以看到将所有预定义的颜色映射应用到灰度图像以增强可视化效果的效果

自定义颜色的地图

可以对图像应用自定义颜色映射。可以通过几种方式实现此功能。

第一种方法是定义一个颜色映射，将0到255的灰度值映射为256种颜色。这可以通过创建一个大小为256x1的8位彩色图像来完成，以便存储所有创建的颜色。然后，通过查找表将图像的灰度强度映射到定义的颜色。为了实现这一点，可执行以下操作之一

- 利用cv2.LUT()函数，
- 将图像的灰度强度映射到定义的颜色，这样就可以使用cv2.applyColorMap()

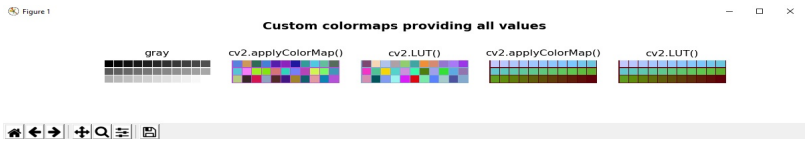
关键点是在创建大小为256x1的8位彩色图像时存储所创建的颜色。如果打算使用cv2.LUT()，则应该按如下方式创建映像

```
lut = np.zeros((256, 3), dtype=np.uint8)
```

用cv2.cv2.applyColorMap()，如下所示

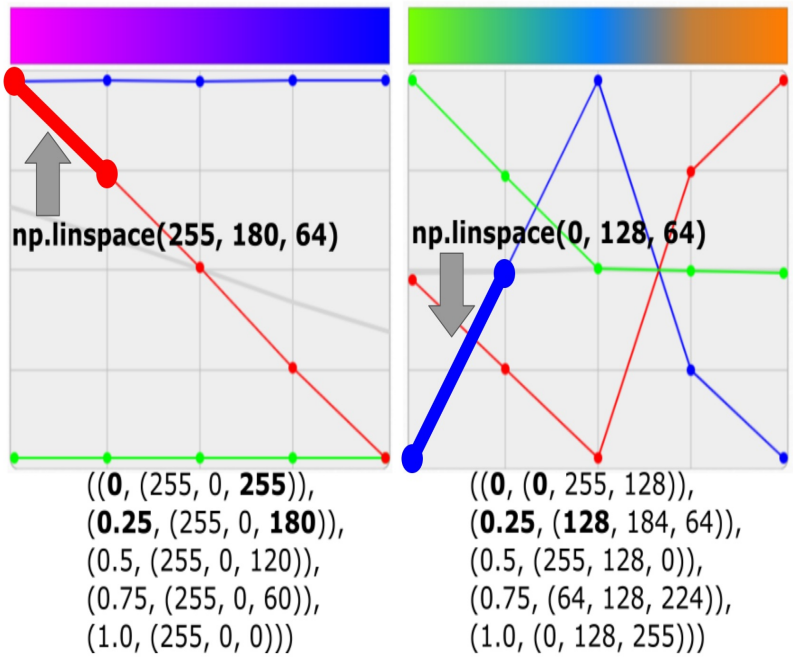
```
lut = np.zeros((256, 1, 3), dtype=np.uint8)
```

完整的代码见color_map_custom_values.py。脚本输出见屏幕截图



定义颜色映射的第二种方法是只提供一些关键颜色，然后插入这些值，以获得构建查找表所需的所有颜色。color_map_custom_key_colors.py脚本展示了如何实现这一点。

build_lut()函数的作用是:根据这些关键颜色构建查找表。基于5个颜色点，这个函数调用np.linespace()来获得在间隔内计算的所有64个均匀间隔的颜色，每个颜色点由两个颜色点定义。为了更好地理解这一点，请看下面的屏幕截图

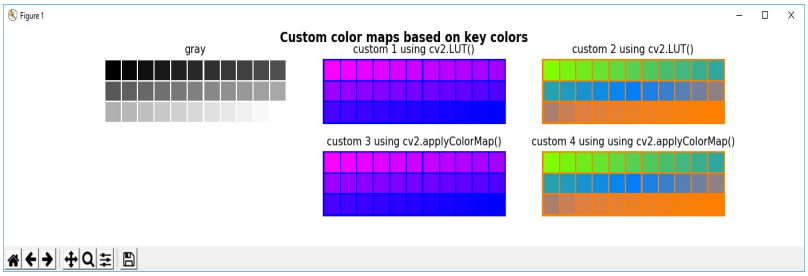


例如，在屏幕截图中，可看到如何计算两个线段的64种均匀间隔的颜色(参见绿色和蓝色高亮显示的线段)

为了构建以下5个关键颜色点((0，(0, 255, 128))、(0.25，(128, 184, 64))、(0.5，(255, 128, 0))、(0.75，(64, 128, 224))和(1.0，(0, 128, 255))查表，调用np.linespace()

```
blue : np.linspace('0', '128', '64' - '0' = '64')
green : np.linspace('255', '184', '64' - '0' = '64')
red : np.linspace('128', '64', '64' - '0' = '64')
blue : np.linspace('128', '255', '128' - '64' = '64')
green : np.linspace('184', '128', '128' - '64' = '64')
red : np.linspace('64', '0', '128' - '64' = '64')
blue : np.linspace('255', '64', '192' - '128' = '64')
green : np.linspace('128', '128', '192' - '128' = '64')
red : np.linspace('0', '224', '192' - '128' = '64')
blue : np.linspace('64', '0', '256' - '192' = '64')
green : np.linspace('128', '128', '256' - '192' = '64')
red : np.linspace('224', '255', '256' - '192' = '64')
```

截屏可以看到color_map_custom_key_colors.py脚本的输出



截图中，可看到将两个自定义颜色映射应用到灰度图像的效果。

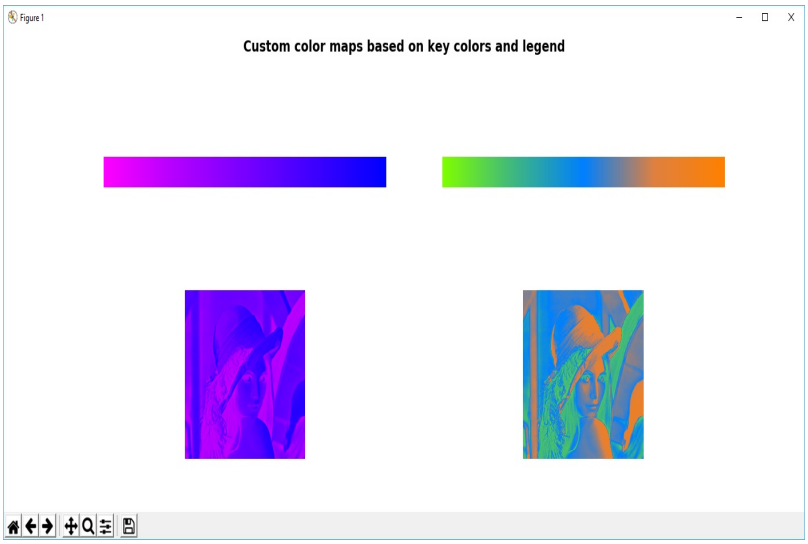
显示自定义颜色地图的图例

一个有趣的功能是在显示自定义颜色地图时提供图例。见color_map_custom_legend.py脚本

为了构建图例，调用build_lut_image()函数。先调用build_lut()函数查表，再调用np.repeat()多次复制查表，查表的形状是(256, 3)，希望输出图像的形状是高度(256, 3)，因此可用np.repeat()和np.newaxis()

```
image = np.repeat(lut[np.newaxis, ...], height, axis=0)
```

脚本输出见屏幕截图



屏幕截图中，可看到将两个自定义颜色映射应用到灰度图像并显示每个颜色映射的图例的效果。

总结

在本章中，我们回顾了计算机视觉项目中需要的大多数常见图像处理技术。在接下来的三章(第6章，构造和构建直方图，第7章，阈值技术，第8章，轮廓检测，滤波和绘图)中，将回顾最常见的图像处理技术。

在第6章，构建和构建直方图，你将学习如何创建和理解直方图，这是一个强大的技术，用来更好地理解图像内容。

问题

1. 哪个函数将一个多通道分割成多个单通道图像?
2. 哪个函数将多个单通道图像合并成一个多通道图像?
3. 在x方向上平移150像素，在y方向上平移300像素。
4. 将名为img的图像相对于图像中心旋转30度，比例系数为1。
5. 构建一个5 x 5的平均内核，并使用cv2.filter2D()将其应用于图像。
6. 将灰度图像中的所有像素加40。
7. 将COLORMAP_JET颜色映射应用于灰度图像。