# 5.形态变换

## 目标：

* 学习不同的形态学操作，如侵蚀，膨胀，开放，关闭等
* 学习不同的函数，如：cv.erode()，cv.dilate()，cv.morphologyEx()等

## 理论

形态学转换是基于图像形状的一些简单操作。它通常在二进制图像上执行。它需要两个输入参数，一个是我们的原始图像，第二个是称为结构元素或核，它决定了操作的性质。腐蚀和膨胀是两个基本的形态学运算符。然后它的变体形式如开运算，闭运算，梯度等也发挥作用。我们将在以下图片的帮助下逐一看到它们：

|  |
| --- |
|  |

## 1.腐蚀

腐蚀的基本思想就像土壤侵蚀一样，它会腐蚀前景物体的边界（总是试图保持前景为白色）。它是如何做到的呢？卷积核在图像中滑动（如在2D卷积中），只有当卷积核下的所有像素都是1时，原始图像中的像素（1或0）才会被认为是1，否则它会被腐蚀（变为零）。

所以腐蚀作用后，边界附近的所有像素都将被丢弃，具体取决于卷积核的大小。因此，前景对象的厚度或大小减小，或者图像中的白色区域减小。它有助于消除小的白噪声（正如我们在色彩空间章节中看到的那样），或者分离两个连接的对象等。

## 例子

我将使用一个5x5卷积核，其中包含完整的卷积核。

|  |
| --- |
|  |

### 效果

|  |
| --- |
|  |

### 注意：图片的选择很重要，如果图片的内容和腐蚀的原理要求的图片内容相反，腐蚀后图片还会变大！！！

## 2.膨胀

它恰好与腐蚀相反。这里，如果卷积核下的像素至少一个像素为“1”，则像素元素为“1”。因此它增加了图像中的白色区域或前景对象的大小。通常，在去除噪音的情况下，侵蚀之后是扩张。因为，侵蚀会消除白噪声，但它也会缩小我们的物体,所以我们扩大它。由于噪音消失了，它们不会再回来，但我们的物体区域会增加。它也可用于连接对象的破碎部分。

## 膨胀例子

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 3.开运算

开运算是**先腐蚀后膨胀**的合成步骤。如上所述，它有助于消除噪音。这里我们使用函数cv.morphologyEx()

## 实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 4.闭运算

闭运算与开运算相反，他是先膨胀后腐蚀的操作。它可用于过滤前景对象内的小孔或对象上的小黑点。

## 实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 5. 形态学梯度

它的处理结果是显示膨胀和腐蚀之间的差异。

结果看起来像对象的轮廓。

## 实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 6.礼帽也叫做顶帽，tophat

**它的处理结果是输入图像和开运算之间的区别。下面的示例是针对9x9卷积核完成的**

**实例**

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 7. 黑帽

它是输入图像闭运算和输入图像之间的差异。

## 实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 8.结构元素：cv.getStructuringElement()

**我们在Numpy的帮助下手动创建了前面示例中的结构元素。它是正方形的，但在某些情况下可能需要椭圆或圆形卷积核。所以为此，OpenCV有一个函数cv.getStructuringElement()。只需传递卷积核的形状和大小，即可获得所需的卷积核。**

**# Rectangular Kernel**

**>>> cv.getStructuringElement(cv.MORPH\_RECT,(5,5))**

**array([[1, 1, 1, 1, 1],**

**[1, 1, 1, 1, 1],**

**[1, 1, 1, 1, 1],**

**[1, 1, 1, 1, 1],**

**[1, 1, 1, 1, 1]], dtype=uint8)**

**# Elliptical Kernel**

**>>> cv.getStructuringElement(cv.MORPH\_ELLIPSE,(5,5))**

**array([[0, 0, 1, 0, 0],**

**[1, 1, 1, 1, 1],**

**[1, 1, 1, 1, 1],**

**[1, 1, 1, 1, 1],**

**[0, 0, 1, 0, 0]], dtype=uint8)**

**# Cross-shaped Kernel**

**>>> cv.getStructuringElement(cv.MORPH\_CROSS,(5,5))**

**array([[0, 0, 1, 0, 0],**

**[0, 0, 1, 0, 0],**

**[1, 1, 1, 1, 1],**

**[0, 0, 1, 0, 0],**

**[0, 0, 1, 0, 0]], dtype=uint8)**

# 6. 图形梯度

## 目标：

* **查找图像梯度，边缘等**
* **学习函数：cv.Sobel()，cv.Scharr()，cv.Laplacian()**

## 理论

OpenCV提供三种类型的梯度滤波器或高通滤波器，Sobel，Scharr和Laplacian。

## 1. Sobel和Scharr

Sobel算子是高斯联合平滑加微分运算，因此它更能抵抗噪声。你可以指定要采用的导数的方向，垂直或水平（yorder和xorder），你还可以通过参数ksize指定卷积核的大小。如果ksize = -1，则使用3x3的Scharr滤波器，其结果优于3x3的Sobel滤波器。

## Scharr实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

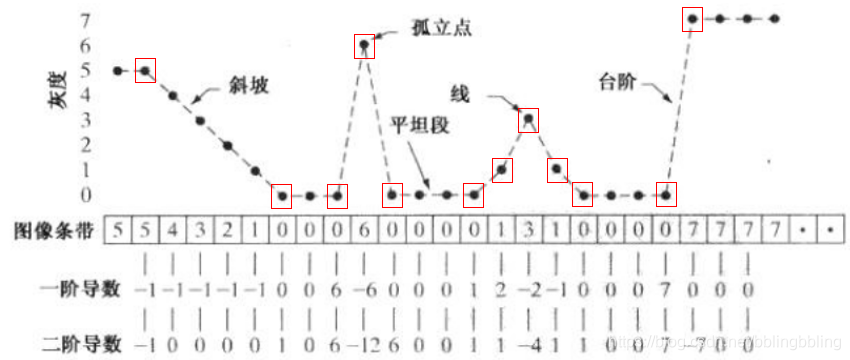
## 2. Laplacian算子

拉普拉斯算子是n维欧几里德空间中的一个二阶微分算子，是用二阶差分来计算图像的边缘。  
拉普拉斯算子是一种各向同性微分算子，它具有旋转不变性。

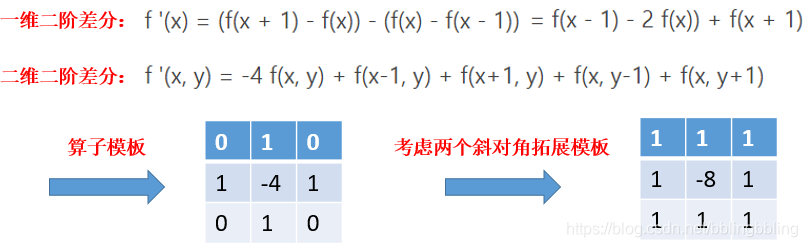
**laplace算子是一种高通滤波器**，用来保留图像的高频分量（变化剧烈的部分），抑制图像的低频分量（变化缓慢的部分），所以可以用来检测边缘。

拉普拉斯算子 (Laplace) 作用：  
1.对图像进行边缘检测；  
2.锐化图像；  
3.判断模糊。

### 二阶差分

一阶差分可以检测边缘存在的可能性，是把灰度变化的区域找出来，检测边缘是否存在。  
二阶差分能确定边缘的存在性，是把灰度值变化的拐点找出来，确定边缘的位置。  


### 算子模板



 如果ksize = 1，则使用以下卷积核进行过滤：

|  |
| --- |
|  |

### 应用原理

利用函数模板可以把图像中的奇异点如亮点变得更亮，对于图像中灰度变化剧烈的区域，该算子能实现其边缘检测。

该算子利用二阶微分特性和峰值间的过零点来判断边缘位置，对边界点更为敏感，常用于锐化图像。

锐化：产生的灰度突变的图像和原始图像叠加，既能产生锐化突变的边缘，又能保留背景信息。

### python代码实现

def laplace(img):

h, w = img.shape

new\_image = np.zeros((h, w))

# operator = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])

operator = np.array([[1,1,1],[1,-8,1],[1,1,1]])

for i in range(1, h-1):

for j in range(1, w-1):

new\_image[i, j] = abs(np.sum(img[i-1:i+2, j-1:j+2] \* operator))

return np.uint8(new\_image)

## 实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 一个重要的事情

在我们的上一个示例中，输出数据类型为cv.CV\_8U或np.uint8，但是这有一个小问题，将黑到白转换视为正斜率（它具有正值），而将白到黑转换视为负斜率（它具有负值）。因此，当你将数据转换为np.uint8时，所有负斜率都为零。简单来说，你丢掉了所有的边界。

如果要检测两个边，更好的选择是将输出数据类型保持为某些更高的形式，如cv.CV\_16S，cv.CV\_64F等，取其绝对值，然后转换回cv.CV\_8U。下面的代码演示了水平Sobel滤波器的这个过程以及结果的差异。

**import numpy as np**

**import cv2 as cv**

**from matplotlib import pyplot as plt**

**img = cv.imread('box.png',0)**

**# Output dtype = cv.CV\_8U**

**sobelx8u = cv.Sobel(img,cv.CV\_8U,1,0,ksize=5)**

**# Output dtype = cv.CV\_64F. Then take its absolute and convert to cv.CV\_8U**

**sobelx64f = cv.Sobel(img,cv.CV\_64F,1,0,ksize=5)**

**abs\_sobel64f = np.absolute(sobelx64f)**

**sobel\_8u = np.uint8(abs\_sobel64f)**

**plt.subplot(1,3,1),plt.imshow(img,cmap = 'gray')**

**plt.title('Original'), plt.xticks([]), plt.yticks([])**

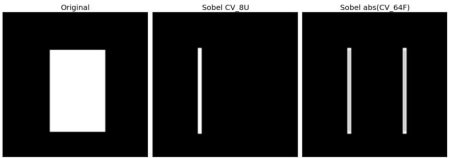
**plt.subplot(1,3,2),plt.imshow(sobelx8u,cmap = 'gray')**

**plt.title('Sobel CV\_8U'), plt.xticks([]), plt.yticks([])**

**plt.subplot(1,3,3),plt.imshow(sobel\_8u,cmap = 'gray')**

**plt.title('Sobel abs(CV\_64F)'), plt.xticks([]), plt.yticks([])**

**plt.show()**

窗口将如下图显示： [](https://camo.githubusercontent.com/f54e18395db6b88781167ddcbee513d7899ec54c322d3ec5a7a7a1a9c6992de8/68747470733a2f2f646f63732e6f70656e63762e6f72672f342e302e302f646f75626c655f656467652e6a7067)

# 7.Canny边缘检测

## 目标：

* Canny边缘检测的概念
* OpenCV的功能：cv.Canny()

## 理论

Canny边缘检测是一种流行的边缘检测算法，它是由John F. Canny开发的。

#### 1. 这是一个多阶段算法，我们将了解其中的每个阶段。

|  |
| --- |
|  |
|  |

#### 2. 降噪

由于边缘检测易受图像中的噪声影响，因此第一步是使用5x5高斯滤波器去除图像中的噪声。

#### 3. 计算图像的强度梯度

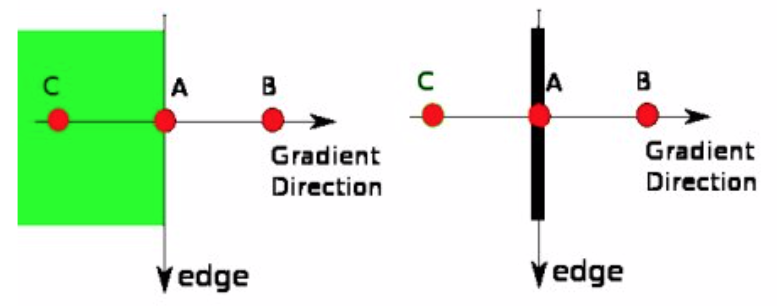
然后在水平和垂直方向上用Sobel核对平滑后的图像进行滤波，以获得水平方向($$ G\_{x}和垂直方向)和垂直方向(G\_{y} $$)的一阶导数。从这两个图像中，我们可以找到每个像素的边缘梯度和方向，如下所示：

|  |
| --- |
|  |

渐变方向始终垂直于边缘。梯度方向被归为四类：垂直，水平，和两个对角线。

#### 4. 非极大值抑制

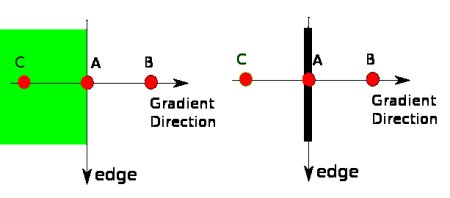
在获得梯度的大小和方向之后，完成图像的全扫描以去除可能不构成边缘的任何不需要的像素。为此，在每个像素处，检查像素是否是其在梯度方向上的邻域中的局部最大值。检查下图：

[](https://raw.githubusercontent.com/TonyStark1997/OpenCV-Python/master/4.Image%20Processing%20in%20OpenCV/Image/image24.png)

A点位于边缘（垂直方向）。渐变方向与边缘垂直。B点和C点处于梯度方向。因此，用点B和C检查点A，看它是否形成局部最大值。如果是这样，则考虑下一阶段，否则，它被抑制（置零）。

简而言之，你得到的结果是具有“细边”的二进制图像。

#### 5. 滞后阈值

这个阶段决定哪些边缘都是边缘，哪些边缘不是边缘。为此，我们需要两个阈值，minVal和maxVal。强度梯度大于maxVal的任何边缘肯定是边缘，而minVal以下的边缘肯定是非边缘的，因此被丢弃。位于这两个阈值之间的人是基于其连通性的分类边缘或非边缘。如果它们连接到“可靠边缘”像素，则它们被视为边缘的一部分。否则，他们也被丢弃。见下图： [](https://camo.githubusercontent.com/aa25a2662a4b917473f5aa2e07193753618cbcf6ab1c53e9c7585bc45978334a/68747470733a2f2f646f63732e6f70656e63762e6f72672f342e302e302f6e6d732e6a7067) 边缘A高于maxVal，因此被视为“确定边缘”。虽然边C低于maxVal，但它连接到边A，因此也被视为有效边，我们得到完整的曲线。但边缘B虽然高于minVal并且与边缘C的区域相同，但它没有连接到任何“可靠边缘”，因此被丢弃。所以我们必须相应地选择minVal和maxVal才能获得正确的结果。 假设边是长线，这个阶段也会消除小像素噪声。 所以我们最终得到的是图像中的强边缘。

penCV将以上所有步骤放在单个函数cv.Canny()中。我们将看到如何使用它。第一个参数是我们的输入图像。第二个和第三个参数分别是我们的minVal和maxVal。第三个参数是aperture\_size,它是用于查找图像渐变的Sobel卷积核的大小。默认情况下它是3。最后一个参数是L2gradient，它指定用于查找梯度幅度的等式。如果它是True，它使用上面提到的更准确的等式

|  |
| --- |
|  |

否则它使用这个函数

|  |
| --- |
|  |

# Canny实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

# 8.图像金字塔

## 目标：

* 了解Image Pyramids
* 使用Image金字塔创建一个新的水果，'Orapple'
* 了解函数：cv.pyrUp()，cv.pyrDown()

## 理论

通常，我们曾经使用恒定大小的图像。但在某些情况下，我们需要使用不同分辨率的（相同）图像。例如，在搜索图像中的某些内容时，如脸部，我们不确定该对象在所述图像中的大小。在这种情况下，我们需要创建一组具有不同分辨率的相同图像，并在所有图像中搜索对象。这些具有不同分辨率的图像被称为图像金字塔（因为当它们保持在堆叠中，底部具有最高分辨率图像而顶部具有最低分辨率图像时，它看起来像金字塔）。 图像金字塔有两种：高斯金字塔和拉普拉斯金字塔 通过去除较低级别（较高分辨率）图像中的连续行和列来形成高斯金字塔中的较高级别（低分辨率）。然后，较高级别中的每个像素由来自基础级别中的5个像素的贡献形成，具有高斯权重。通过这样做，M×N图像变为M/2 × N/2图像。因此面积减少到原始面积的四分之一。它被称为Octave。当我们在金字塔中上升时（即分辨率降低）将以相同的模式继续。同样，在扩展时，每个级别的区域变为4次。我们可以使用cv.pyrDown()和cv.pyrUp()函数找到高斯金字塔。

### pyrDown函数

函数cv2.pyrDown()，用于实现图像高斯金字塔操作中的向下采样，其语法形式为：

dst = cv2.pyrDown( src[, dstsize[, borderType]] )

* dst为目标图像
* src为原始图像
* dstsize为目标图像的大小
* borderType为边界类型，默认值为BORDER\_DEFAULT，且这里仅支持BORDER\_DEFAULT

默认情况下，输出图像的大小为Size((src.cols+1)/2, (src.rows+1)/2)。

在任何情况下，图像尺寸必须满足如下条件：

* |dst. width∗2-src. cols|≤2
* |dst. height∗2-src. rows|≤2

cv2.pyrDown()函数首先对原始图像进行高斯滤波变换，以获取原始图像的近似图像。在获取近似图像后，该函数通过抛弃偶数行和偶数列来实现向下采样。

### pyrDown实例

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

### pyrUp函数

在OpenCV中，使用函数cv2.pyrUp()实现图像金字塔操作中的向上采样，其语法形式如下：

dst = cv2.pyrUp( src[, dstsize[, borderType]] )

* dst为目标图像
* src为原始图像
* dstsize为目标图像的大小
* borderType为边界类型，默认值为BORDER\_DEFAULT，且这里仅支持BORDER\_DEFAULT。

目标图像的大小为Size(src.cols\*2, src.rows\*2)。

在任何情况下，图像尺寸需要满足下列条件:

* |dst. width-src. cols∗2|≤mod（dst. widh,2）
* |dst. height-src. rows∗2|≤mod（dst. height,2）

对图像向上采样时，在每个像素的右侧、下方分别插入零值列和零值行，得到一个偶数行、偶数列（即新增的行、列）都是零值的新图像New。接下来，用向下采样时所使用的高斯滤波器对新图像New进行滤波，得到向上采样的结果图像。

为了确保像素值区间在向上采样后与原始图像保持一致，**需要将高斯滤波器的系数乘以4。**

### pyrUp实例

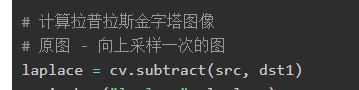
|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

## 拉普拉斯金字塔

拉普拉斯金字塔由高斯金字塔形成，没有特别的功能。拉普拉斯金字塔图像仅与边缘图像相似。它的大部分元素都是零。它们用于图像压缩。拉普拉斯金字塔中的一个层次由高斯金字塔中的该层次与高斯金字塔中的上层的扩展版本之间的差异形成



## 拉普拉斯金字塔实例

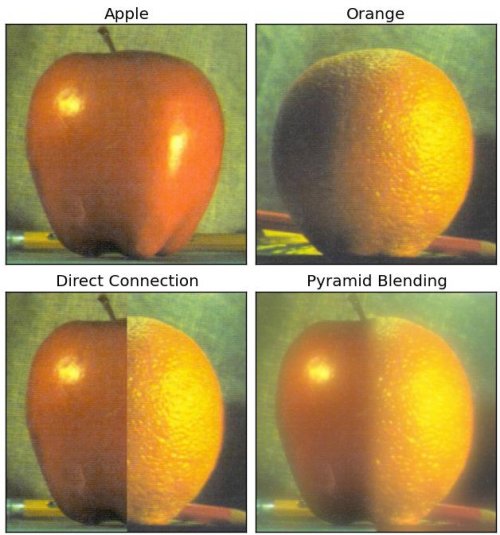
|  |
| --- |
|  |

### 注意：拉普拉斯金字塔有一个坑，图像的宽高/2如果是奇数，制作拉普拉斯金字塔就会失败

### 效果：

|  |
| --- |
|  |
|  |

## 使用金字塔的图像混合

金字塔的一个应用是图像混合。例如，在图像拼接中，你需要将两个图像堆叠在一起，但由于图像之间的不连续性，它可能看起来不太好。在这种情况下，与金字塔混合的图像可以让你无缝混合，而不会在图像中留下太多数据。其中一个典型的例子是混合了两种水果，橙子和苹果。 现在查看结果以了解我在说什么： [](https://camo.githubusercontent.com/8ae098d75fb8a2200b2cc8c9840d4d45e9e1d305f519c8ac976bc8032d446794/68747470733a2f2f646f63732e6f70656e63762e6f72672f342e302e302f6f726170706c652e6a7067)

请在附加资源中查看第一个参考资料，它有关于图像混合，拉普拉斯金字塔等的完整图表细节。简单地完成如下：

1. 加载苹果和橙色的两个图像
2. 找到苹果和橙色的高斯金字塔（在这个特殊的例子中，级别数是6）
3. 从高斯金字塔，找到他们的拉普拉斯金字塔
4. 现在加入左半部分的苹果和右半部分的拉普拉斯金字塔
5. 最后，从这个联合图像金字塔，重建原始图像。

以下是完整的代码。（为简单起见，每个步骤都是单独完成的，可能会占用更多内存。如果需要，可以对其进行优化）。

|  |
| --- |
| **import cv2 as cv import numpy as np, sys  A = cv.imread('../mydata/apple.jpg') B = cv.imread('../mydata/orange.jpg')  *# generate Gaussian pyramid for A* G = A.copy() gpA = [G] for i in range(6):  G = cv.pyrDown(G)  gpA.append(G)  *# generate Gaussian pyramid for B* G = B.copy() gpB = [G] for i in range(6):  G = cv.pyrDown(G)  gpB.append(G)  *# generate Laplacian Pyramid for A* lpA = [gpA[5]] for i in range(5, 0, -1):  GE = cv.pyrUp(gpA[i])  L = cv.subtract(gpA[i - 1], GE)  lpA.append(L)  *# generate Laplacian Pyramid for B* lpB = [gpB[5]] for i in range(5, 0, -1):  GE = cv.pyrUp(gpB[i])  L = cv.subtract(gpB[i - 1], GE)  lpB.append(L)  *# Now add left and right halves of images in each level* LS = [] for la, lb in zip(lpA, lpB):  rows, cols, dpt = la.shape  ls = np.hstack((la[:, 0:int(cols/2)], lb[:, int(cols/2):]))  LS.append(ls)  *# now reconstruct* ls\_ = LS[0] for i in range(1, 6):  ls\_ = cv.pyrUp(ls\_)  ls\_ = cv.add(ls\_, LS[i])  *# image with direct connecting each half* real = np.hstack((A[:, :int(cols / 2)], B[:, int(cols / 2):]))  cv.imwrite('Pyramid\_blending2.jpg', ls\_) cv.imwrite('Direct\_blending.jpg', real)** |

### 这个程序还是有点难度，需要慢慢理解