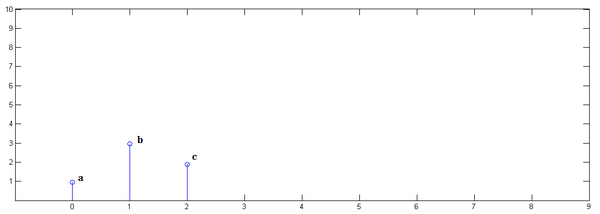
# 什么是卷积

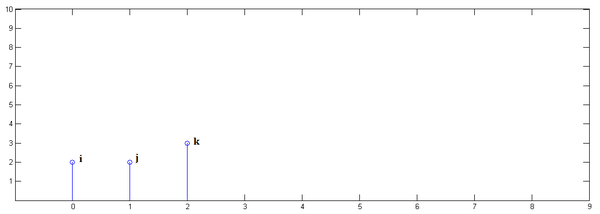
## 1.如何通俗易懂的理解卷积

以离散信号为例，连续信号同理。

已知x[0] = a, x[1] = b, x[2]=c

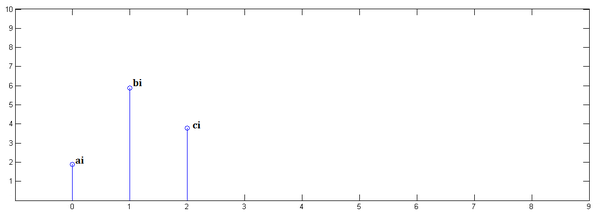


已知y[0] = i, y[1] = j, y[2]=k

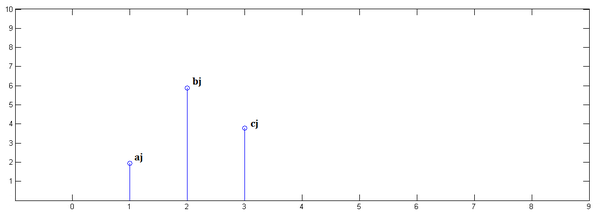


下面通过演示求x[n] \* y[n]的过程，揭示卷积的物理意义。

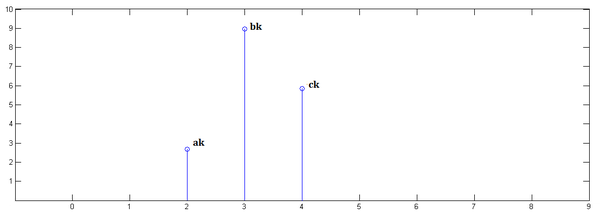
第一步，x[n]乘以y[0]并平移到位置0：



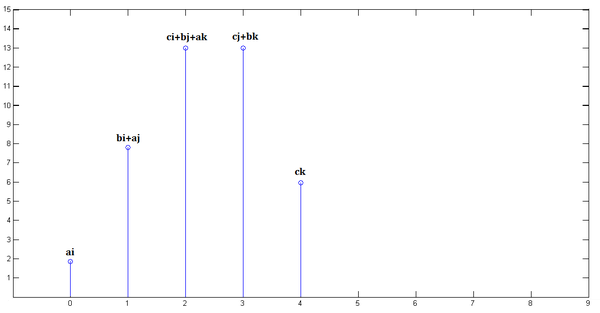
第二步，x[n]乘以y[1]并平移到位置1：



第三步，x[n]乘以y[2]并平移到位置2：



最后，把上面三个图叠加，就得到了x[n] \* y[n]：



从这里，可以看到卷积的重要的物理意义是：一个函数（如：单位响应）在另一个函数（如：输入信号）上的加权叠加。

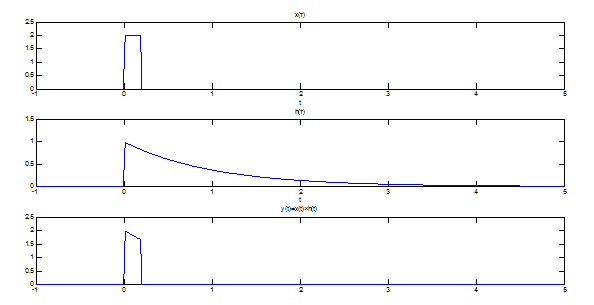
## 2.从信号与系统的角度来理解卷积



在这个模型中，每当给系统一个输入脉冲信号时，都会在不同的时刻产生一个对应的输出。

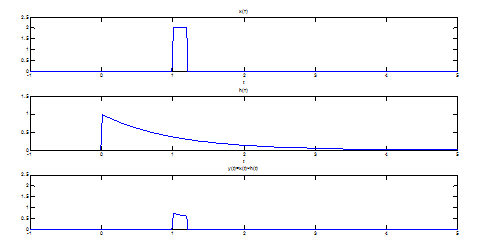
当t=t1时：

第一个函数x（t）为输入脉冲，第二个函数h（t）为系统函数，第三个函数为在t1时刻产生的响应输出。



当t=t2时：

第一个函数x（t）为输入脉冲，第二个函数h（t）为系统函数，第三个函数为在t2时刻产生的响应输出。



下面的两个动图可以演示卷积的过程(请双击打开)



## 3.卷积的数学公式





上述为离散和连续信号的卷积公式。简单的解释起来就是：在每个时间点t，将x(τ)翻转为x(-τ)，再平移为x(t-τ)，与h(τ)乘积的结果，求面积，就得到卷积的结果

卷积最早来自于信号系统理论，后来被数学家们发扬光大了，而且其威力已经远远超出了发明者的初衷。

先来看信号处理中如何出现卷积的。假设B是一个系统，其t时刻的输入为x(t)，输出为y(t)，系统的响应函数为h(t)、，按理说，输出与输入的关系应该为Y(t)=h(t)x(t)，然而，实际的情况是，系统的输出不仅与系统在t时刻的响应有关，还与它在t时刻之前的响应有关，不过系统有个衰减过程，所以t1（<t）时刻的输入对输出的影响通常可以表示为x(t)h(t-t1)，这个过程可能是离散的，也可能是连续的，所以t时刻的输出应该为t时刻之前系统响应函数在各个时刻响应的叠加，这就是卷积。

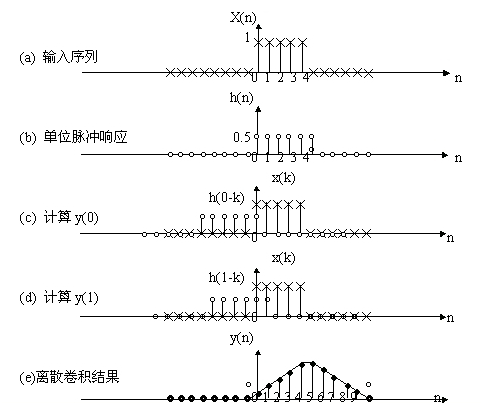
## 4.离散卷积的计算方法

### 4.1 直接计算

(1)对h（n）绕纵轴折叠，得h（-n）；

(2)对h（-m）移位得h（n-m）；

(3)将x（m）和h（n-m）所有对应项相乘之后相加得离散卷积结果y（n）。



### 4.2 快速傅立叶变换

卷积定理指出，函数卷积的傅里叶变换是函数傅里叶变换的乘积。即，一个域中的卷积相当于另一个域中的乘积，例如时域中的卷积就对应于频域中的乘积。



其中表示f的傅里叶变换。

利用卷积定理可以简化卷积的运算量。对于长度为n的序列，按照卷积的定义进行计算，需要做2n-1组对位乘法，其计算复杂度为。而利用傅里叶变换将序列变换到频域上后，只需要一组对位乘法，利用傅里叶变换的快速算法之后，总的计算复杂度为。

### 4.3 分段卷积

將 f [ n ] 切成好幾段，每一段分別和 g [ n ] 做卷積後，再將結果相加。

## 5.python傅立叶变换和卷积计算

python中有numpy和scipy用于数学计算。

### 5.1 观察信号的频谱

import numpy as np

import pylab as pl

sampling\_rate **=** 8000

fft\_size **=** 512

t **=** np**.**arange**(**0**,** 1.0**,** 1.0**/**sampling\_rate**)**

x **=** np**.**sin**(**2**\***np**.**pi**\***156.25**\***t**)** **+** 2**\***np**.**sin**(**2**\***np**.**pi**\***234.375**\***t**)**

xs **=** x**[:**fft\_size**]**

xf **=** np**.**fft**.**rfft**(**xs**)/**fft\_size

freqs **=** np**.**linspace**(**0**,** sampling\_rate**/**2**,** fft\_size**/**2**+**1**)**

xfp **=** 20**\***np**.**log10**(**np**.**clip**(**np**.**abs**(**xf**),** 1e-20**,** 1e100**))**

pl**.**figure**(**figsize**=(**8**,**4**))**

pl**.**subplot**(**211**)**

pl**.**plot**(**t**[:**fft\_size**],** xs**)**

pl**.**xlabel**(**"Taime"**)**

pl**.**title**(**"156.25Hz and 234.375HZ"**)**

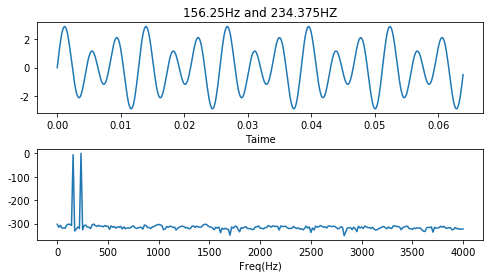
pl**.**subplot**(**212**)**

pl**.**plot**(**freqs**,** xfp**)**

pl**.**xlabel**(**"Freq(Hz)"**)**

pl**.**subplots\_adjust**(**hspace**=**0.4**)**

pl**.**show**()**



下面逐行对这个程序进行解释：

首先定义了两个常数：sampling\_rate, fft\_size，分别表示数字信号的取样频率和FFT的长度。

然后调用np.arange产生1秒钟的取样时间，t中的每个数值直接表示取样点的时间，因此其间隔为取样周期1/sampline\_rate：

t **=** np**.**arange**(**0**,** 1.0**,** 1.0**/**sampling\_rate**)**

用取样时间数组t可以很方便地调用函数计算出波形数据，这里计算的是两个正弦波的叠加，一个频率是156.25Hz，一个是234.375Hz：

x **=** np**.**sin**(**2**\***np**.**pi**\***156.25**\***t**)** **+** 2**\***np**.**sin**(**2**\***np**.**pi**\***234.375**\***t**)**

为什么选择这两个奇怪的频率呢？因为这两个频率的正弦波在512个取样点中正好有整数个周期。满足这个条件波形的FFT结果能够精确地反映其频谱。

N点FFT能精确计算的频率

假设取样频率为fs**,** 取波形中的N个数据进行FFT变换。那么这N点数据包含整数个周期的波形时，FFT所计算的结果是精确的。于是能精确计算的波形的周期是**:** n**\***fs**/**N。对于8kHz取样，512点FFT来说，8000**/**512.0 **=** 15.625Hz，前面的156.25Hz和234.375Hz正好是其10倍和15倍。

下面从波形数据x中截取fft\_size个点进行fft计算。np.fft库中提供了一个rfft函数，它方便我们对实数信号进行FFT计算。根据FFT计算公式，为了正确显示波形能量，还需要将rfft函数的结果除以fft\_size：

xs **=** x**[:**fft\_size**]**

xf **=** np**.**fft**.**rfft**(**xs**)/**fft\_size

rfft函数的返回值是N/2+1个复数，分别表示从0(Hz)到sampling\_rate/2(Hz)的N/2+1点频率的成分。于是可以通过下面的np.linspace计算出返回值中每个下标对应的真正的频率：

freqs **=** np**.**linspace**(**0**,** sampling\_rate**/**2**,** fft\_size**/**2**+**1**)**

最后我们计算每个频率分量的幅值，并通过 20\*np.log10() 将其转换为以db单位的值。为了防止0幅值的成分造成log10无法计算，我们调用np.clip对xf的幅值进行上下限处理：

xfp **=** 20**\***np**.**log10**(**np**.**clip**(**np**.**abs**(**xf**),** 1e-20**,** 1e100**))**

### 5.2 两个一维信号卷积

import numpy as np

x**=**np**.**array**([**1**,**2**,**3**])**

h**=**np**.**array**([**4**,**5**,**6**])**

import scipy**.**signal

result **=** scipy**.**signal**.**convolve**(**x**,**h**)** #卷积运算

print**(**result**)**

输出为：

**[** 4 13 28 27 18**]**

卷积运算大致可以分成3步，首先先翻转，让两个信号列反过来，如上面就是1,2,3和6,5,4。然后作平移，6,5,4最开始在1,2,3的左边，没有重叠，现在向右移动，4和1就重叠了。对于重叠的部分，作乘积求和。也就是1x4得到第一个结果1，然后再移动后5x1+4x2得到第二个结果13以此类推。

### 5.3 快速卷积

如果我们要计算数组a和b的卷积，a和b的长度都为128，那么它们的卷积结果的长度为 len(a) + len(b) - 1 = 255。为了用FFT能够计算其线性卷积，需要将a和b都扩展到256。下面的程序演示这个计算过程：

import numpy as np

def fft\_convolve**(**a**,**b**):**

n **=** len**(**a**)+**len**(**b**)-**1

N **=** 2**\*\*(**int**(**np**.**log2**(**n**))+**1**)**

A **=** np**.**fft**.**fft**(**a**,** N**)**

B **=** np**.**fft**.**fft**(**b**,** N**)**

**return** np**.**fft**.**ifft**(**A**\***B**)[:**n**]**

**if** \_\_name\_\_ **==** "\_\_main\_\_"**:**

a **=** np**.**random**.**rand**(**128**)**

b **=** np**.**random**.**rand**(**128**)**

c **=** np**.**convolve**(**a**,**b**)**

print**(**np**.**sum**(**np**.**abs**(**c **-** fft\_convolve**(**a**,**b**))))**

此程序的输出为直接卷积和FFT快速卷积的结果之间的误差6.43183586249e-12。

在这段程序中，a,b的长度为128，其卷积结果c的长度为n=255，我们通过下面的算式找到大于n的最小的2的整数次幂：

N **=** 2**\*\*(**int**(**np**.**log2**(**n**))+**1**)**

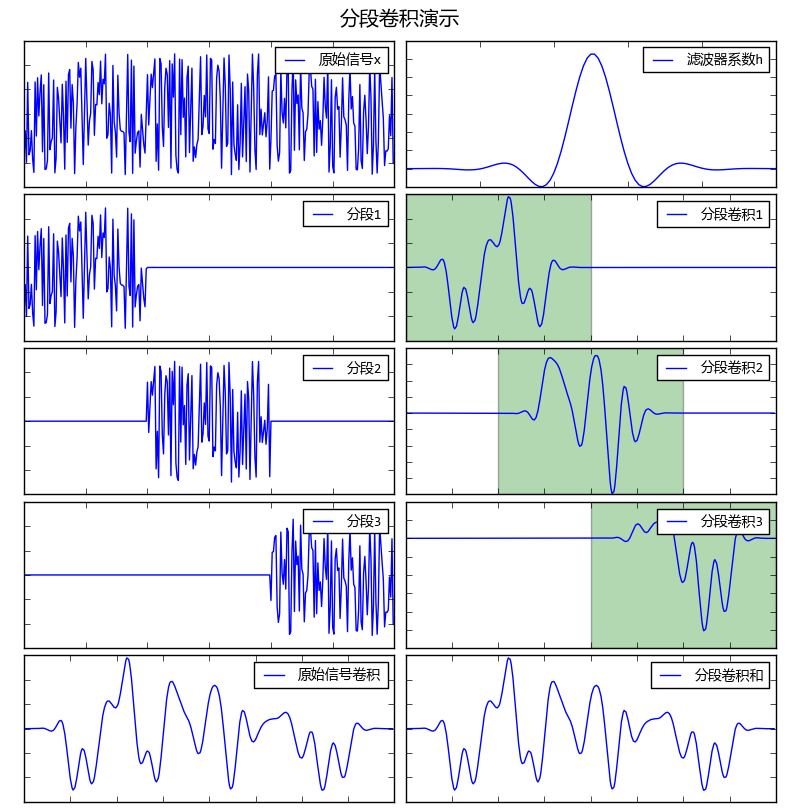
在调用fft函数对其进行变换时，传递第二个参数为N(FFT的长度)，这样fft函数将自动对a,b进行补零。最后通过ifft得到的卷积结果c2的长度为N，比实际的卷积结果c要多出一个数，这个多出来的元素应该接近于0。

### 5.4 分段卷积

现在考虑对于输入信号x和系统响应h的卷积运算，通常x是非常长的，例如要对某段录音进行滤波处理，假设取样频率为8kHz，录音长度为1分钟的话，那么x的长度为480000。而且x的长度也可能不是固定的，例如我们可能需要对麦克风的连续输入信号进行滤波处理。而h的长度通常都是固定的，例如它是某个房间的冲击响应，或者是某种FIR滤波器。

根据前面的介绍，为了有效地利用FFT计算卷积，我们希望它的两个输入长度相当，于是就需要对信号x进行分段处理。对卷积的分段运算被称作：overlap-add运算。

overlap-add的计算方法如下图所示：



原始信号x长度为300，将它分为三段，分别与滤波器系数h进行卷积计算，h的长度为101，因此每段输出200个数据，图中用绿色标出每段输出的200个数据。这3段数据按照时间顺序进行求和之后得到结果和原始信号的卷积是相同的。

因此将持续的输入信号x和滤波器h进行卷积的运算可以按照如下步骤进行，假设h的长度为M：

1.建立一个缓存，其大小为N+M-1，初始值为0

2.每次从x中读取N个数据，和h进行卷积，得到N+M-1个数据，和缓存中的数据进行求和，并放进缓存中，然后输出缓存前N个数据

3.将缓存中的数据向左移动N个元素，也就是让缓存中的第N个元素成为第0个元素，后面的N个元素全部设置为0

4.跳转到2重复运行

下面是实现这一算法的演示程序：

import numpy as np

x **=** np**.**random**.**rand**(**1000**)**

h **=** np**.**random**.**rand**(**101**)**

y **=** np**.**convolve**(**x**,** h**)**

N **=** 50 # 分段大小

M **=** len**(**h**)** # 滤波器长度

output **=** **[]**

#缓存初始化为0

buffer **=** np**.**zeros**(**M**+**N**-**1**,**dtype**=**np**.**float64**)**

**for** i in range**(**int**(**len**(**x**)/**N**)):**

#从输入信号中读取N个数据

xslice **=** x**[**i**\***N**:(**i**+**1**)\***N**]**

#计算卷积

yslice **=** np**.**convolve**(**xslice**,** h**)**

#将卷积的结果加入到缓冲中

buffer **+=** yslice

#输出缓存中的前N个数据，注意使用copy，否则输出的是buffer的一个视图

output**.**append**(** buffer**[:**N**].**copy**()** **)**

#缓存中的数据左移动N个元素

buffer**[**0**:**M**-**1**]** **=** buffer**[**N**:]**

#后面的补0

buffer**[**M**-**1**:]** **=** 0

#将输出的数据组合为数组

y2 **=** np**.**hstack**(**output**)**

#计算和直接卷积的结果之间的误差

print**(**np**.**sum**(**np**.**abs**(** y2 **-** y**[:**len**(**x**)]** **)** **))**

注意第23行需要输出缓存前N个数据的拷贝，否则输出的是数组的一个视图，当此后buffer更新时，视图中的数据会一起更新。

将FFT快速卷积和overlap-add相结合，可以制作出一些快速的实时数据滤波算法。但是由于FFT卷积对于两个长度相当的数组时最为有效，因此在分段时也会有所限制：例如如果滤波器的长度为2048，那么理想的分段长度也为2048，如果将分段长度设置得过低，反而会增加运算量。因此在实时性要求很强的系统中，只能采用直接卷积。

## 6.数字图像处理和卷积

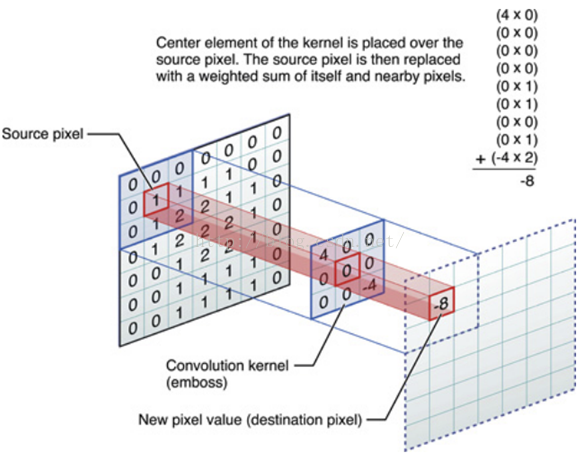
### 6.1数字图像处理中卷积

在数字图像处理中, 有一种基本的处理方法:线性滤波. 待处理的平面数字图像可被看做一个大矩阵, 图像的每个像素对应着矩阵的每个元素, 假设我们平面的分辨率是 1024\*768, 那么对应的大矩阵的行数= 1024, 列数=768。

用于滤波的是一个滤波器小矩阵(也叫卷积核), 滤波器小矩阵一般是个方阵, 也就是 行数 和 列数 相同, 比如常见的用于边缘检测的 Sobel 算子 就是两个 3\*3 的小矩阵。

进行滤波就是对于大矩阵中的每个像素, 计算它周围像素和滤波器矩阵对应位置元素的乘积, 然后把结果相加到一起, 最终得到的值就作为该像素的新值, 这样就完成了一次滤波。

上面的处理过程可以参考这个示意图：

对图像大矩阵和滤波小矩阵对应位置元素相乘再求和的操作就叫卷积(Convolution)或协相关(Correlation)。

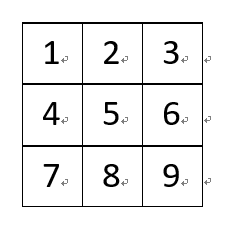
协相关(Correlation)和卷积(Convolution)很类似, 两者唯一的差别就是卷积在计算前需要翻转卷积核, 而协相关则不需要翻转。

### 6.2 边界补充问题

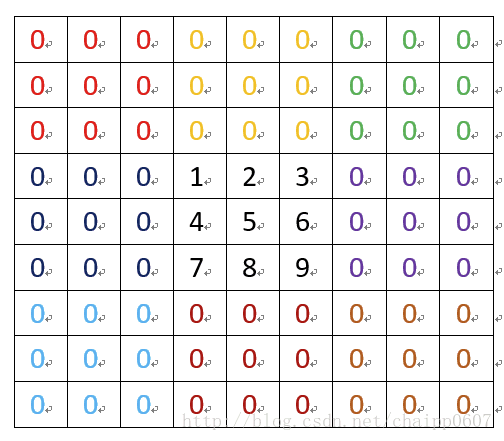
上面的图片说明了图像的卷积操作，但是他也反映出一个问题，如上图，原始图片尺寸为7\*7，卷积核的大小为3\*3，当卷积核沿着图片滑动后只能滑动出一个5\*5的图片出来，这就造成了卷积后的图片和卷积前的图片尺寸不一致，这显然不是我们想要的结果，所以为了避免这种情况，需要先对原始图片做边界填充处理。在上面的情况中，我们需要先把原始图像填充为9\*9的尺寸。

常用的区域填充方法包括：

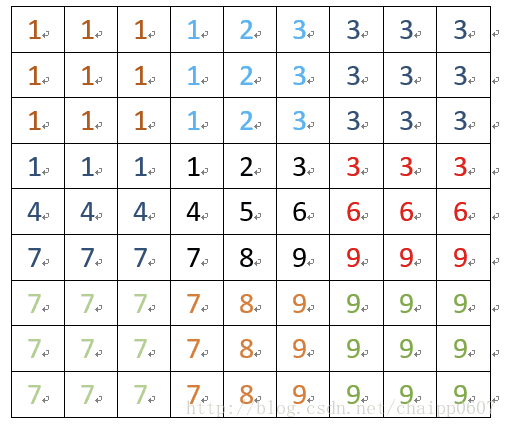
原始图像：



(1)补零



(2)边界复制



(3)镜像

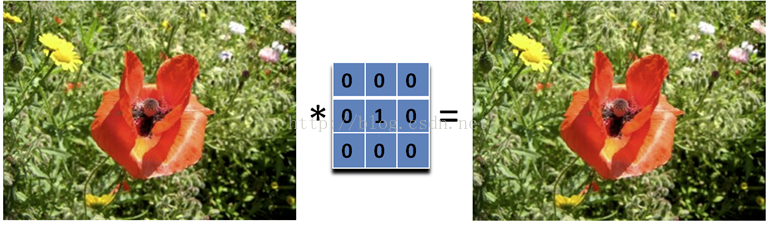


(4)块复制

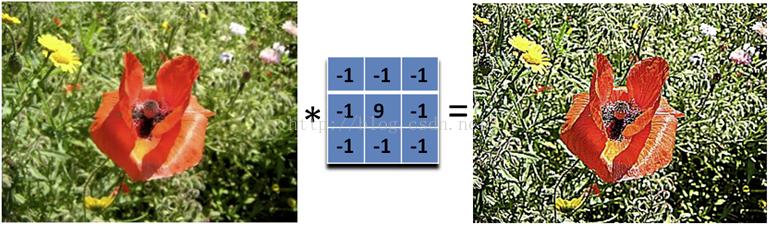


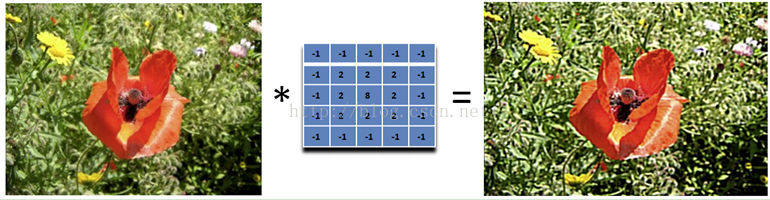
### 6.3 不同卷积核下的卷积意义

(1)什么也不做

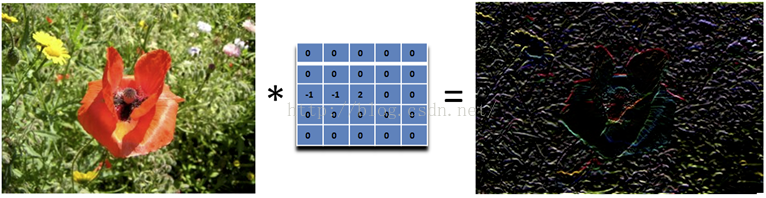


(2)锐化

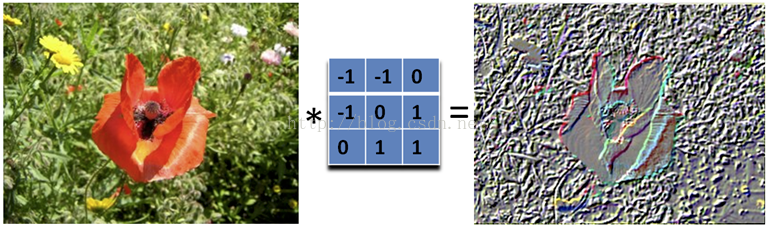




(3)边缘检测

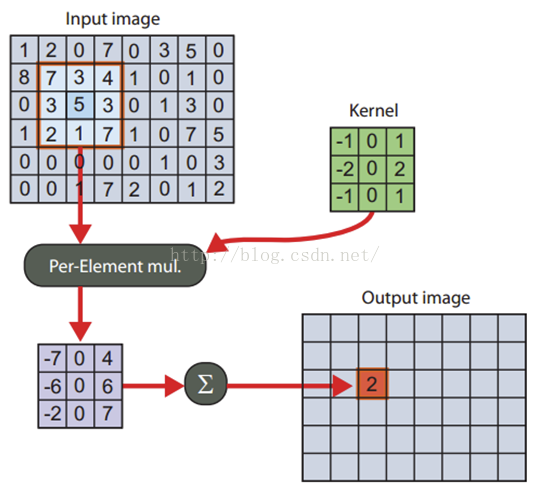


(4)浮雕



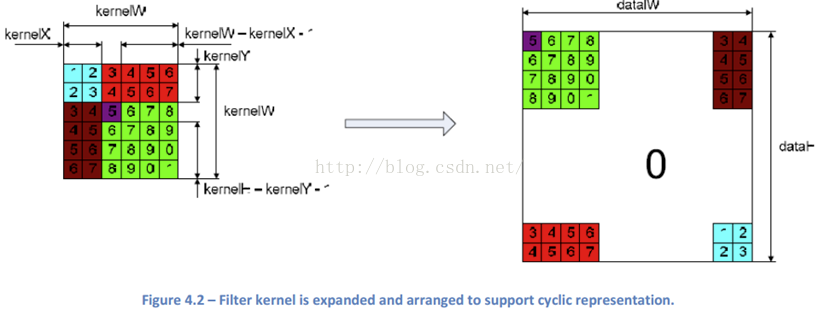
### 6.4 图像卷积的计算方法

(1)2D卷积



(2)快速傅立叶变换

要在频域中对一副图像进行滤波，滤波器的大小和图像的大小必须要匹配，这样两者的相乘才容易。因为一般滤波器的大小比图像要小，所以我们需要拓展我们的kernel，让它和图像的大小一致。



在拓展的过程中，滤波器进行了180度旋转，将协相关用卷积来计算。

### 6.5 在线模拟卷积和数字图像处理

<https://graphics.stanford.edu/courses/cs178/applets/convolution.html>

