Python下图片的高斯模糊化的优化

#算法 #python

前言

在上一篇博文中，我们介绍了高斯模糊的算法，在文末的时候点到了当时我们的方法模糊一张很小的图要2.82s的时间，这对于很多需要高斯模糊需求的场景而言有着很大的局限性，以及在上一篇介绍的方法中，对于边缘的问题直接不进行处理了，这导致模糊半径大的时候图片四周有很大一块原图区，所以在本文中就会对这两个问题的解决方法进行解决.并且将对在上一篇文章中提到的随着半径增加模糊时间是以什么增长方式进行的问题进行简单的阐释.

归纳一下本篇的所讲内容

- 解决边缘未模糊问题
- 高斯模糊速度优化
- 解释随模糊半径增加所需时间关于平方增长还是线性增长的问题

上一篇博文>> Python下尝试实现图片的高斯模糊化 (传送门)

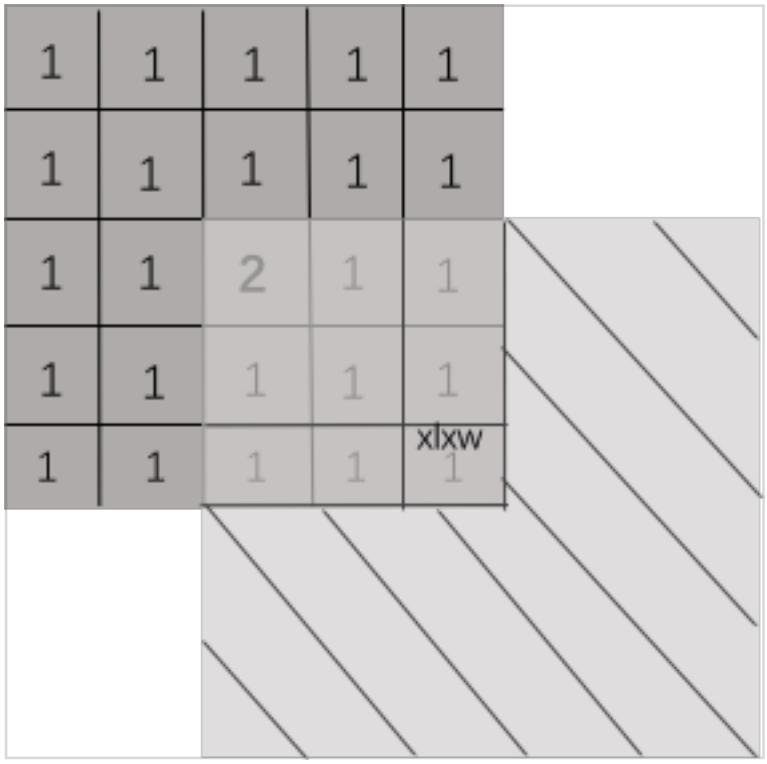
边缘问题的解决方案

上次文末用的例子中模糊半径R和标准差σ都取了3，我们可以通过图片可以看到很明显的边缘没有被处理，经过我们在上一篇博文的介绍，我们现在都应该知道每个像素点的灰度值或者RGB三通道值都是通过周围像素得到的，那么边缘处的像素点该怎么办呢？它们周围至少有一边缺少了参考点，这个时候我们怎么对它们(边缘)进行模糊化呢?

四周未模糊

忽略超出部分影响的方法

首先我们先来观察一下对于边缘像素而言，下图深灰色的为图像的边角一部分，而模糊半径为2，需要模糊所需的像素矩阵范围我们这里用浅灰色(边缘外部加上了斜线)来表示:



我们可以看到，以最右下角的像素为例，这个例子中模糊半径为2，它所需要的矩阵范围内只有9个格子可以给它参考，而剩下的16个格子都分布在了图像的外面.那么我们在边上的格子不可取的时候，我们能做到的就是先用存在的9个格子的像素对它边缘的像素进行模糊，其他边缘处的处理方法也是一样，所以我们先来用代码来看一下模糊的效果（PS：由于完整代码过长，所以本篇博文除了最终代码集合外，只展示对应部分的代码）

```
def newrgb(ma,nr,ng,nb,r):#生成新的像素rgb矩阵
    timer[2] = clock()
    newr = np.empty((sizepic[0],sizepic[1]))
    newg = np.empty((sizepic[0],sizepic[1]))
    newb = np.empty((sizepic[0],sizepic[1]))
    for i in range(0,sizepic[0]):
        for j in range(0,sizepic[1]):
            o = 0
            for x in range(i-r,i+r+1):
                p = 0
                for y in range(j-r,j+r+1):
                    if x<0 or x >sizepic[0]-1 or y<0 or y>sizepic[1]-1:   #此处是对
边缘的判断
                        pass
                    else:
                        newr[i][j] += nr[x][y]*ma[o][p]
                        newg[i][j] += ng[x][y]*ma[o][p]
                        newb[i][j] += nb[x][y]*ma[o][p]
```

```
                    p += 1
                o += 1
    print("已经计算出新的三通道矩阵，所花时间为{:.3f}s".format(timer[2]))
    return newr,newg,newb
```

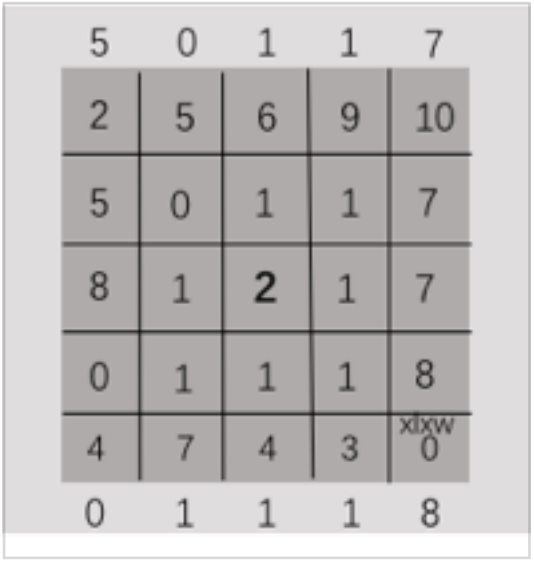
优化后的效果图：

经过边缘优化与优化前的对比图

通过上述方法对边缘进行优化后我们可以发现边框确实有了模糊的效果，但是毕竟只处理了一部分内部像素，把外部的像素都忽略了，所以整体边框偏暗，有类似黑边的存在.那么我们怎么才能够做到将黑边去掉呢？这里我们来看看镜像扩充的方法.

镜像扩充

（镜像扩充这个名字是因为我觉得挺像的所以这样子叫，如果有和其他专业术语重合请跳过这个名字）我之所以叫这种方法镜像扩充，是因为由于我们的边缘的参考矩阵超过了图片范围，并且忽略超出部分的上种方法也存在了局限性会留有小黑边（最边上的参数缺少过多）.在这里我们用的这种方法就是把缺少的点都补上来避免黑边的出现.具体的方法是通过边缘的点在边缘外界补上关于边缘点对称的点的RGB值/灰度值，就像是里面的数字和边缘镜像对称一样.而补多少由模糊半径决定，示意图如下图所示：



第一步填充顶底

第二步填充左右

- 左图中我们可以看到顶部(第一行)50117是第三行的数字关于第二行（图片上边缘）对称得到

的，同理下边缘也是.

- 右图中我们可以看到最左列0501171是第三列关于第二列（图片左边缘）对称得到的，最右列也是同理
- （由于四个角上对于中心值而言影响最小，故虽然不是很准确但影响不大）

通过这种方法我们可以有效的避免 上一种方法因为无视边缘超出部分的影响而造成的边缘偏暗有黑边的现象.

我们用代码来验证一下我们的这种方法（为了清晰，四周的添加方法单独列出）

```
def getrgb(path,r):#得到图像中各个点像素的RGB三通道值
    timer[0]=clock()
    pd = p.open(path)
    sizepic[0] = pd.size[0]
    sizepic[1] = pd.size[1]
    nr = np.zeros((sizepic[0],sizepic[1]))
    ng = np.zeros((sizepic[0],sizepic[1]))
    nb = np.zeros((sizepic[0],sizepic[1]))
    for i in range(0,sizepic[0]):
        for j in range(0,sizepic[1]):
            nr[i][j] = pd.getpixel((i,j))[0]
            ng[i][j] = pd.getpixel((i,j))[1]
            nb[i][j] = pd.getpixel((i,j))[2]
    #镜像扩充
    for i in range(1,r+1):#顶部
        nxr = nr[i*2-1]
        nxg = ng[i*2-1]
        nxb = nb[i*2-1]
        nr = np.insert(nr,0,values = nxr ,axis = 0)
        ng = np.insert(ng,0,values = nxg ,axis = 0)
        nb = np.insert(nb,0,values = nxb ,axis = 0)
    for i in range(sizepic[0]+r-1,sizepic[0]-1,-1):#底部
        nxr = nr[i]
        nxg = ng[i]
        nxb = nb[i]
        nr = np.insert(nr,(sizepic[0]+r-1)*2-i,values = nxr ,axis = 0)
        ng = np.insert(ng,(sizepic[0]+r-1)*2-i,values = nxg ,axis = 0)
        nb = np.insert(nb,(sizepic[0]+r-1)*2-i,values = nxb ,axis = 0)
    for i in range(1,r+1):#左侧
```

```
        nxr = nr[:,i*2-1]
        nxg = ng[:,i*2-1]
        nxb = nb[:,i*2-1]
        nr = np.insert(nr,0,values = nxr ,axis = 1)
        ng = np.insert(ng,0,values = nxg ,axis = 1)
        nb = np.insert(nb,0,values = nxb ,axis = 1)
    for i in range(sizepic[1]+r-1,sizepic[1]-1,-1):#右侧
        nxr = nr[:,i]
        nxg = ng[:,i]
        nxb = nb[:,i]
        nr = np.insert(nr,(sizepic[1]+r-1)*2-i,values = nxr ,axis = 1)
        ng = np.insert(ng,(sizepic[1]+r-1)*2-i,values = nxg ,axis = 1)
        nb = np.insert(nb,(sizepic[1]+r-1)*2-i,values = nxb ,axis = 1)
    print("已经得到所有像素的R,G,B的值，所花时间为{:.3f}s".format(clock()-timer[0]))
    return nr,ng,nb
```

通过镜像扩充优化边缘后的效果图：

边缘的模糊总结

在上文中，我们提到了两种处理边缘的方法，分别是：

- 忽略超出部分影响
- 镜像扩充

这两种方法，第二种比较适合使用，因为这样边缘不会存在黑边（暗区），两种法都是有意义的.第一种虽然有黑边，但是黑边部分并非是纯黑色，所以作为图片之间的分界也具有一定美观性并且减少了边缘部分的大量计算也可以提升模糊速度.这两种方法具体使用视实际情况而定.

高斯模糊速度优化

我们在上一节的尝试中，我们最终可以看到最终，我们对实验室LOGO（一个120x120大小的图片）进行高斯模糊时，在模糊半径为3时我们所用的时间2.82s.这个时间太久了，我们用过PS的都知道它的高斯模糊几乎是瞬时完成的.相比而言，我们上一节使用的方法过于低效，所以在下文中我们对于如何优化模糊速度展开尝试.

图片压缩->模糊->放大

我们可以通过实验发现发现当图片压缩->放大时图片会丢失细节；而我们的高斯模糊也是一样，当图片经过高斯模糊后，细节会丢失，所以对于这两者而言，都会使图片失去细节，那么我们来看看实际上这两者产生效果有没有大的区别吧.

直接模糊　　缩小模糊后放大

这里我们的缩小模糊后放大的方法长宽都先缩小为原来的一半，相同的模糊半径我们可以发现右边的模糊感更强，即模糊半径相对更大.所以我么可以知道缩小的比例和模糊半径的选定有关.(但是这种方法对于时间的缩减有巨大贡献)直接模糊所花的时间为4.805s，而先缩小(甚至得到的是一个更大模糊的半径)的图片，我们只用了2.2s就模糊完毕，缩短了一半多的时间.只是我们需要探索模糊半径和缩小比例之间的关系.

高斯函数的可分离性

可分离性的意义

由于高斯函数具有可分离性所以可以将高斯函数写作可分离的形式，可以采用可分离滤波的方式来实现模糊加速.分离滤波-指将多维的卷积化成多个一维卷积.所以这里我们要使用的是二维的高斯函数，所以可以分离为：

- 先沿水平方向对图像的灰度 / RGB卷积一次
- 对卷积完的矩阵再对铅垂方向卷积

我们可以发现，一维的高斯函数如下：

$$G = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x||y)^2}{2\sigma^2}}$$

这里x||y指的是x或者y（要分别对x，y进行卷积）

上述一维函数可以对每一个（x，y）坐标进行计算（需要镜像扩充），所以可以排除边缘的干扰.（这里由于使用了两次卷积来代替二维卷积，也大大提高了卷集的速度(在最后会提到速度的变化)）

分离性应用于高斯模糊

改进代码如下：

```
def matcombine(r,g,b,rd):
    #模糊矩阵
    summat = 0
```

```
    timer[1] = clock()
    ma = np.zeros(2*rd+1)
    for i in range(0,2*rd+1):
        ma[i] = (1/(((2*PI)**0.5)*rd))*math.e**(-((i-rd)**2)/(2*(rd**2)))
        summat += ma[i]
    ma[0::1] /= summat
    print("已经计算出高斯函数矩阵，所花时间为{:.3f}s".format(clock()-timer[1]))
    timer[1] = clock()-timer[1]
    #blur
    ner,neg,neb = np.zeros_like(r),np.zeros_like(g),np.zeros_like(b)
    u,p,q = 0,0,0
    #y向模糊
    timer[2] = clock()

    for i in range(rd+1,sizepic[0]+rd-1):
        for j in range(rd+1,sizepic[1]+rd-1):
            u += r[j-rd:j+rd+1:1,i]*ma[0::1]
            p += g[j-rd:j+rd+1:1,i]*ma[0::1]
            q += b[j-rd:j+rd+1:1,i]*ma[0::1]
            ner[j][i],neg[j][i],neb[j][i] = u.sum(0),p.sum(0),q.sum(0)
            u,p,q = 0,0,0
    #x向模糊
    for i in range(rd+1,sizepic[0]+rd-1):
        for j in range(rd+1,sizepic[1]+rd-1):
            u += ner[i,j-rd:j+rd+1:1]*ma[0::1]
            p += neg[i,j-rd:j+rd+1:1]*ma[0::1]
            q += neb[i,j-rd:j+rd+1:1]*ma[0::1]
            ner[i][j] = u.sum(0)
            neg[i][j] = p.sum(0)
            neb[i][j] = q.sum(0)
            u,p,q = 0,0,0
    print("已经完成生成，所花时间为{:.3f}s".format(clock()-timer[2]))
    timer[2] = clock()-timer[2]
    return ner,neg,neb
```

优化截图：

```
>>>
======== RESTART: /Users/xulvxiaowei/Desktop/Gossac blur/1D Gossa.py =
请输入模糊的半径: 5
请输入图片的地址(包括后缀): ./bt.png
已经得到所有像素的R,G,B的值，所花时间为0.086s
已经计算出高斯函数矩阵，所花时间为0.000s
已经完成模糊，所花时间为0.343s
已经完成生成，所花时间为0.033s
正在导出图片..
bt.png - >> blurred.jpg
总计耗时:0.467s,感谢您的使用.
>>>
======== RESTART: /Users/xulvxiaowei/Desktop/Gossac blur/1D Gossa.py =
请输入模糊的半径: 10
请输入图片的地址(包括后缀): ./bt.png
已经得到所有像素的R,G,B的值，所花时间为0.086s
已经计算出高斯函数矩阵，所花时间为0.000s
已经完成模糊，所花时间为0.350s
已经完成生成，所花时间为0.029s
正在导出图片..
bt.png - >> blurred.jpg
总计耗时:0.468s,感谢您的使用.
```

优化效果：(半径为10)

半径为10的优化效果

分离性定理对高斯模糊的优化总结

我们可以看到，用了高斯函数的分离性（再加上numpy对于矩阵处理的优化），无论是半径5还是半径10，都仅仅只需要500ms左右（由于计算机的差别具体时间可能略有区别，但是500ms去处读取等其他的操作，对于大部分计算机而言在运算上时间上几乎一致）这个算法对比起来快了很多，如果不考虑读取等因素，大约仅需要400ms，已经几乎可以实现我们前面说的实时高斯模糊了.如果将高斯函数的分离性和图片压缩的方法结合在一起，甚至时间应该能压缩到200ms.(但本文中不对这两种方法结合进行介绍)

全文方法总结集合

全文代码展示(这里不集成先缩小->模糊->放大的方式)

```
#一维高斯模糊 -xlxw
from PIL import Image as p
import numpy as np
```

```
from time import clock
import math

#define
sizepic = [0,0]
timer = [0,0,0,0]
PI = math.pi

def getrgb(path,r):#得到图像中各个点像素的RGB三通道值
    timer[0] = clock()
    pd = p.open(path)
    sizepic[0],sizepic[1] = pd.size[0],pd.size[1]
    nr = np.zeros((sizepic[0],sizepic[1]))
    ng = np.zeros((sizepic[0],sizepic[1]))
    nb = np.zeros((sizepic[0],sizepic[1]))
    for i in range(0,sizepic[0]):
        for j in range(0,sizepic[1]):
            nr[i][j] = pd.getpixel((i,j))[0]
            ng[i][j] = pd.getpixel((i,j))[1]
            nb[i][j] = pd.getpixel((i,j))[2]
    #镜像扩充
    for i in range(1,r+1):#顶部
        nxr = nr[i*2-1]
        nxg = ng[i*2-1]
        nxb = nb[i*2-1]
        nr = np.insert(nr,0,values = nxr ,axis = 0)
        ng = np.insert(ng,0,values = nxg ,axis = 0)
        nb = np.insert(nb,0,values = nxb ,axis = 0)
    for i in range(sizepic[0]+r-1,sizepic[0]-1,-1):#底部
        nxr = nr[i]
        nxg = ng[i]
        nxb = nb[i]
        nr = np.insert(nr,(sizepic[0]+r-1)*2-i,values = nxr ,axis = 0)
        ng = np.insert(ng,(sizepic[0]+r-1)*2-i,values = nxg ,axis = 0)
        nb = np.insert(nb,(sizepic[0]+r-1)*2-i,values = nxb ,axis = 0)
    for i in range(1,r+1):#左侧
        nxr = nr[:,i*2-1]
        nxg = ng[:,i*2-1]
        nxb = nb[:,i*2-1]
```

```
        nr = np.insert(nr,0,values = nxr ,axis = 1)
        ng = np.insert(ng,0,values = nxg ,axis = 1)
        nb = np.insert(nb,0,values = nxb ,axis = 1)
    for i in range(sizepic[1]+r-1,sizepic[1]-1,-1):#右侧
        nxr = nr[:,i]
        nxg = ng[:,i]
        nxb = nb[:,i]
        nr = np.insert(nr,(sizepic[1]+r-1)*2-i,values = nxr ,axis = 1)
        ng = np.insert(ng,(sizepic[1]+r-1)*2-i,values = nxg ,axis = 1)
        nb = np.insert(nb,(sizepic[1]+r-1)*2-i,values = nxb ,axis = 1)
    print("已经得到所有像素的R,G,B的值，所花时间为{:.3f}s".format(clock()-timer[0]))
    timer[0] = clock()-timer[0]
    return nr,ng,nb

def matcombine(r,g,b,rd):
    #模糊矩阵
    summat = 0
    timer[1] = clock()
    ma = np.zeros(2*rd+1)
    for i in range(0,2*rd+1):
        ma[i] = (1/(((2*PI)**0.5)*rd))*math.e**(-((i-rd)**2)/(2*(rd**2)))
        summat += ma[i]
    ma[0::1] /= summat
    print("已经计算出高斯函数矩阵，所花时间为{:.3f}s".format(clock()-timer[1]))
    timer[1] = clock()-timer[1]
    #blur
    ner,neg,neb = np.zeros_like(r),np.zeros_like(g),np.zeros_like(b)
    u,p,q = 0,0,0
    #y向模糊
    timer[2] = clock()

    for i in range(rd+1,sizepic[0]+rd-1):
        for j in range(rd+1,sizepic[1]+rd-1):
            u += r[j-rd:j+rd+1:1,i]*ma[0::1]
            p += g[j-rd:j+rd+1:1,i]*ma[0::1]
            q += b[j-rd:j+rd+1:1,i]*ma[0::1]
            ner[j][i],neg[j][i],neb[j][i] = u.sum(0),p.sum(0),q.sum(0)
            u,p,q = 0,0,0
    #x向模糊
```

```
    for i in range(rd+1,sizepic[0]+rd-1):
        for j in range(rd+1,sizepic[1]+rd-1):
            u += ner[i,j-rd:j+rd+1:1]*ma[0::1]
            p += neg[i,j-rd:j+rd+1:1]*ma[0::1]
            q += neb[i,j-rd:j+rd+1:1]*ma[0::1]
            ner[i][j] = u.sum(0)
            neg[i][j] = p.sum(0)
            neb[i][j] = q.sum(0)
            u,p,q = 0,0,0
    print("已经完成模糊，所花时间为{:.3f}s".format(clock()-timer[2]))
    timer[2] = clock()-timer[2]
    return ner,neg,neb

def cpic(r,g,b,path,rd):#图片输出
    timer[3] = clock()
    pd = p.new("RGB",(sizepic[0]-rd-1,sizepic[1]-rd-1))
    for i in range(rd+1,sizepic[0]):
        for j in range(rd+1,sizepic[1]):
            pd.putpixel((i-rd-1,j-rd-1),(int(r[i][j]),int(g[i][j]),int(b[i][j])))
    print("已经完成生成，所花时间为{:.3f}s".format(clock() - timer[3]))
    timer[3] = clock()-timer[3]
    print("正在导出图片..")
    pd.save("blurred.jpg")

def main():
    rd = eval(input("请输入模糊的半径："))
    path = input("请输入图片的地址(包括后缀)：")
    nr,ng,nb = getrgb(path,rd)
    nr,ng,nb = matcombine(nr,ng,nb,rd)
    cpic(nr,ng,nb,path,rd)
    print("{} - >> {}".format(path.split('/')[-1],"blurred.jpg"))
    print("总计耗时:{:.3f}s,感谢您的使
用.".format(timer[0]+timer[1]+timer[2]+timer[3]))
main()
```

阐释高斯模糊运算时间的增长方式

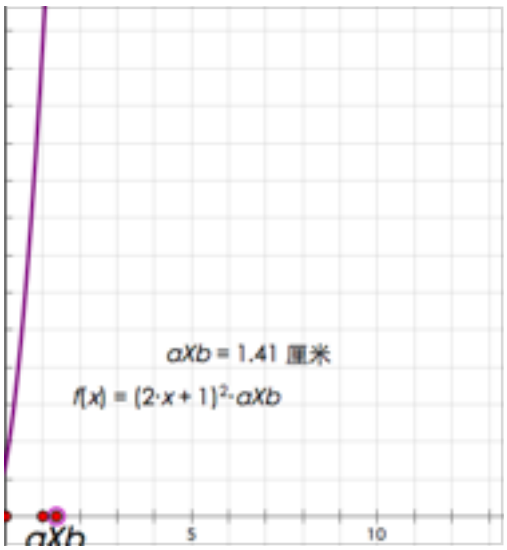
会出现两种增长方式就是因为前文中用到的用到的两种情况（二维高斯函数和一维高斯函数）（这里不考虑逼近得到的图像）

首先我们来分析二维高斯函数：

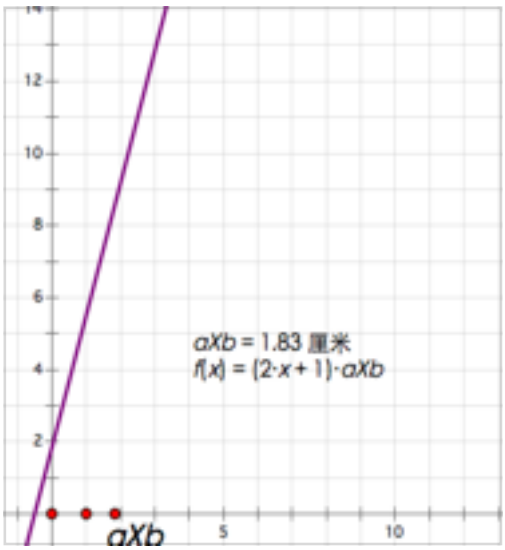
- 模糊的半径为x，那么每个像素点需要运算的次数为(2Xx+1)^2次，那么设图像大小为aXb，则aXbX(2Xx+1)^2次运算

那么一维函数会是怎么样的结果，让我们看下面：

- 模糊的半径为x，那么每个像素点需要运算的次数为(2Xx+1)次，那么设图像大小为aXb，则进行了aXbX(2Xx+1)次运算



二维高斯函数运算次数图像



一维高斯函数运算次数图像

结论：一维高斯函数的计算时间随线性增长，而二维函数则是平房增长，所以意为高斯函数在这方便更有优势.

其他的快速高斯模糊的办法

- 当模糊半径为一很大的数值时，我们可以发现高斯二维函数对于离中心点越远的权重越小，所以当到达了某种数量级后可以忽略后方范围的点对于中心像素点的影响

- 通过傅立叶变换与高斯模糊函数结合能大量减少卷积的时间

- 通过多次均值模糊来达到高斯模糊的效果

可参考文献：

Fastest Gaussian Blur (in linear time)