Python下通过算法使图片高斯模糊

#算法 #python

高斯模糊是什么?

(先来看一下维基百科对它的定义)

> 高斯模糊是模糊图像的结果.它是一种广泛使用的图形软件的影响，通常会减少图像噪声和减少细节。这个模糊技术的视觉效果是一个平滑的模糊相似，查看图片通过一个半透明的屏幕，从明显不同散景在通常的照明的聚焦透镜或物体的阴影产生的影响。高斯平滑也被用来作为一个预处理阶段计算机视觉算法以提高图像在不同尺度的结构见尺度空间表示和尺度空间的实现。
>
> 在数学上，应用高斯模糊图像是一样的卷积一个图像高斯函数。这也被称为一个二维维尔斯特拉斯变换。相比之下，通过循环卷积\能更准确地再现散景功效自傅里叶变换一个高斯是另一个高斯，应用高斯模糊具有降低图像的高频成分的影响；高斯模糊是一个低通滤波器.

高斯模糊的应用

1.在程序的背景设计中，我们总想用一些图片来对程序进行美化，但是直接用图片会发现和字，控件等不是很和谐；这个时候，在我们需要的地方或者说整一张背景进行适当的高斯模糊，就能提升我们程序的颜值了.
2.高斯模糊还被用于边缘检测，用来使图片平滑化，以此来移除细节.以及应用于图片降噪.

高斯模糊处理的结果


这里我就把我们实验室的图标拿来用PS高斯模糊了一下来举个例子.

高斯模糊名词解释

高斯模糊将正态分布作用于图像处理的方面."高斯模糊"的算法，是一种数据平滑技术，适用于很多的场合和情景，但是在这里，图像处理恰好是一个很直观的应用实例.

高斯模糊的核心思想和数据平滑技术

模糊的本质，实际上就是每一个图片中的像素取周边像素的均值.如下图所示：

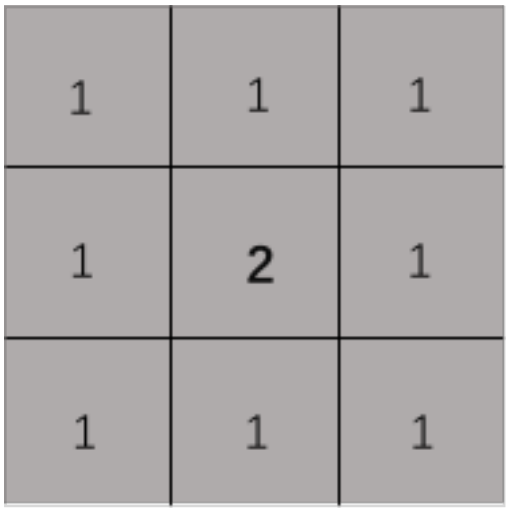


图1(原像素)

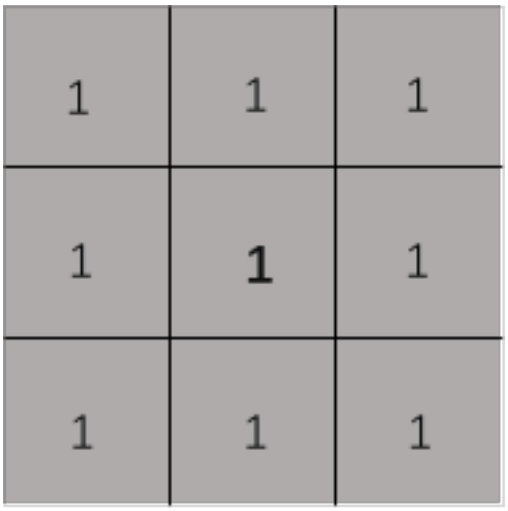


图2(平滑化)

中心处点的值取附近所有点值的平均值，就会变成了值1.这样在数值上，可以看作是一种平滑化.那么在二维图像上，也可以看作是有了"模糊"的效果，中心点失去了自己本身的像素值，相当于细节丢失.而图像的模糊程度就完全取决于图像的模糊半径了.从值德角度来说，数值更加的趋向于平滑.

总结：高斯模糊的核心就是取中心点周围所有像素点的均值作为自己的像素值，以此来达到平滑效果，在算法上，涉及到很多问题，从这些问题也是影响高斯模糊速度（模糊效率）的重要方面因素

模糊时间

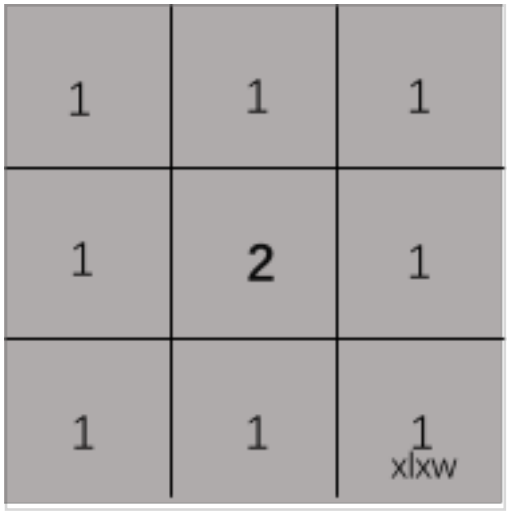
由于要对所有的点进行计算，所以高斯模糊是一个相对而言较为耗时的作用，而模糊所需的时间取决点也有很多，诸如算法，以及图片自身属性以及人的要求，机器性能都是影响到高斯模糊时间.

其中对高斯模糊的时间影响最大的是以下两个因素：

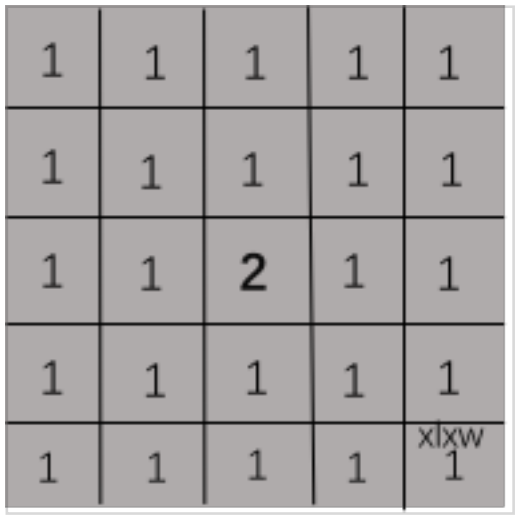
1.模糊半径

2.像素点(图片大小)

像素点(图像)这个因素很好理解，越大的图片意味着运算的点越多，而模糊半径可以大致上这样理解，如下图：

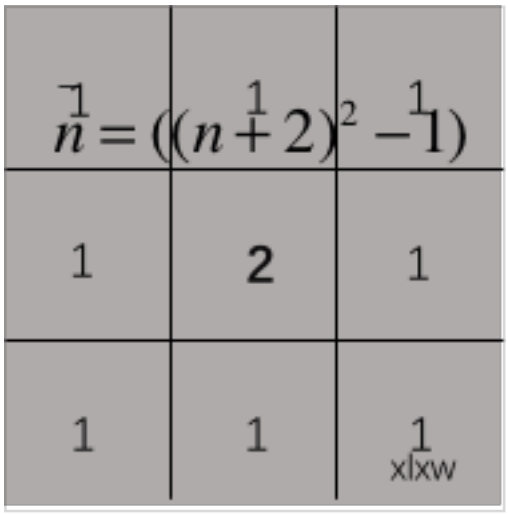


图一(r=1)

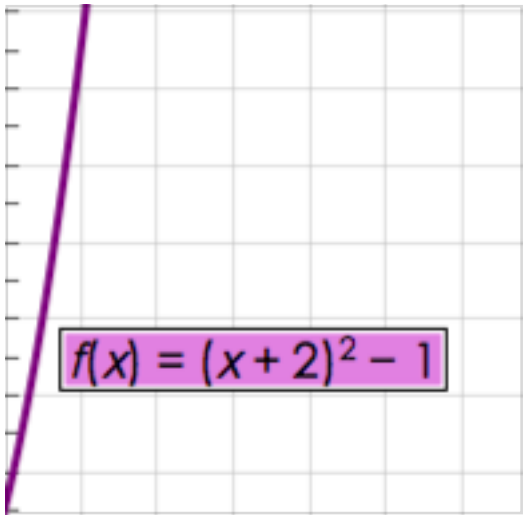


图二(r=2)

从上图我们就能看出图一中半径为1，中心点参考的只有周围8个格子中的像素值，而图二半径为2就要参考24个格子的像素值，以此类推可以得到：（其中n为半径）



像素点计算公式



二次函数图像

增速是递增的(网上看到有的说是处理的像素点的个数是指数上升，对于这个问题可能我理解的不是很到位，但是结论也是半径越大处理的点越多，好的时间也就越多.)

综上所述：模糊半径和图像的大小是影响高斯模糊速度的最大因素,而一般程序要求高斯模糊的速度在ms级(像PS几乎察觉不到模糊的时间，就是算法的重要性)，否则会影响用户体验.这里我们在后文中会做一些尝试.

高斯模糊的公式以及理解

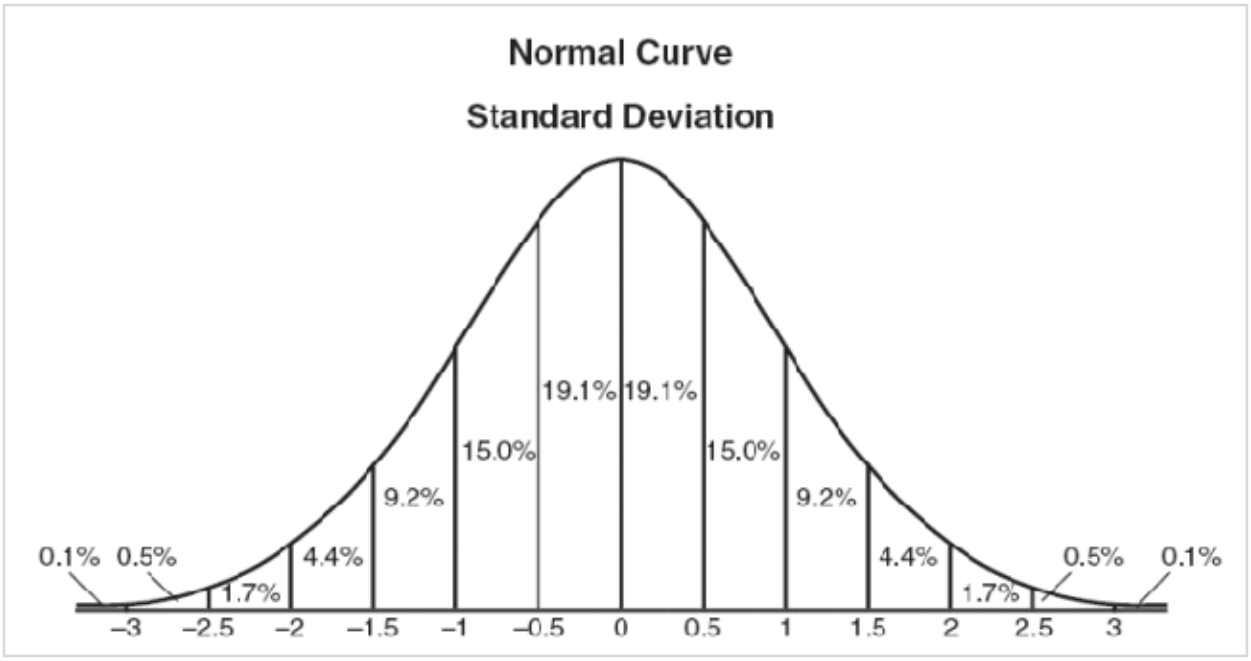
首先，高斯模糊需要数学概率中的正态分布公式，这也是高斯模糊名称的由来，因为正态分布的别名就是"高斯分布".

正态分布

公式（一维）图像以及参数含义：

$$f(x)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

正态分布的图像如下：



正态分布的两个参数：

即均数μ. &. 标准差σ，σ^2为方差

图像&正态分布的关系：

正态分布一种权重的分配模式.在图形的应用中，正态分布可以看作是一种钟形曲线，越接近中心，取值越大，越远离中心，取值越小.计算均值的时候，我们只将中间的点作为x=0的原点，其他附近的点按照其在正态曲线上的位置（离所处半径长度）,分配对应的权重，就能够得到一个加权的平均值了.之后再对像素点进行处理即可

正态分布密度函数-高斯函数

我们都知道，图像都是二维的，而上面的正态分布则是一维的，这个时候我们先引入一维的高斯函数：

$$f(x)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}},-\infty<x<\infty$$

推导过程(百度百科)：

证明 $f_X(x)=\int_{-\infty}^{\infty} f(x,y)\,dy$ 是一维正态分布

由于 $\frac{(y-\mu_2)^2}{\sigma_2}-2\rho\frac{(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2}=\left(\frac{y-\mu_2}{\sigma_2}-\rho\frac{x-\mu_1}{\sigma_1}\right)^2-\rho^2\frac{(x-\mu_1)^2}{\sigma_1^2}$

于是

$$xf_X(x)=\frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}}e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}}\int_{-\infty}^{\infty}e^{-\frac{\left(\frac{y-\mu_2}{\sigma_2}-\rho\frac{x-\mu_1}{\sigma_1}\right)^2}{2\left(1-\rho^2\right)}}dyx$$

令 $t=\frac{1}{\sqrt{1-\rho^2}}\left(\frac{y-\mu_2}{\sigma_2}-\rho\frac{x-\mu_1}{\sigma_1}\right)$

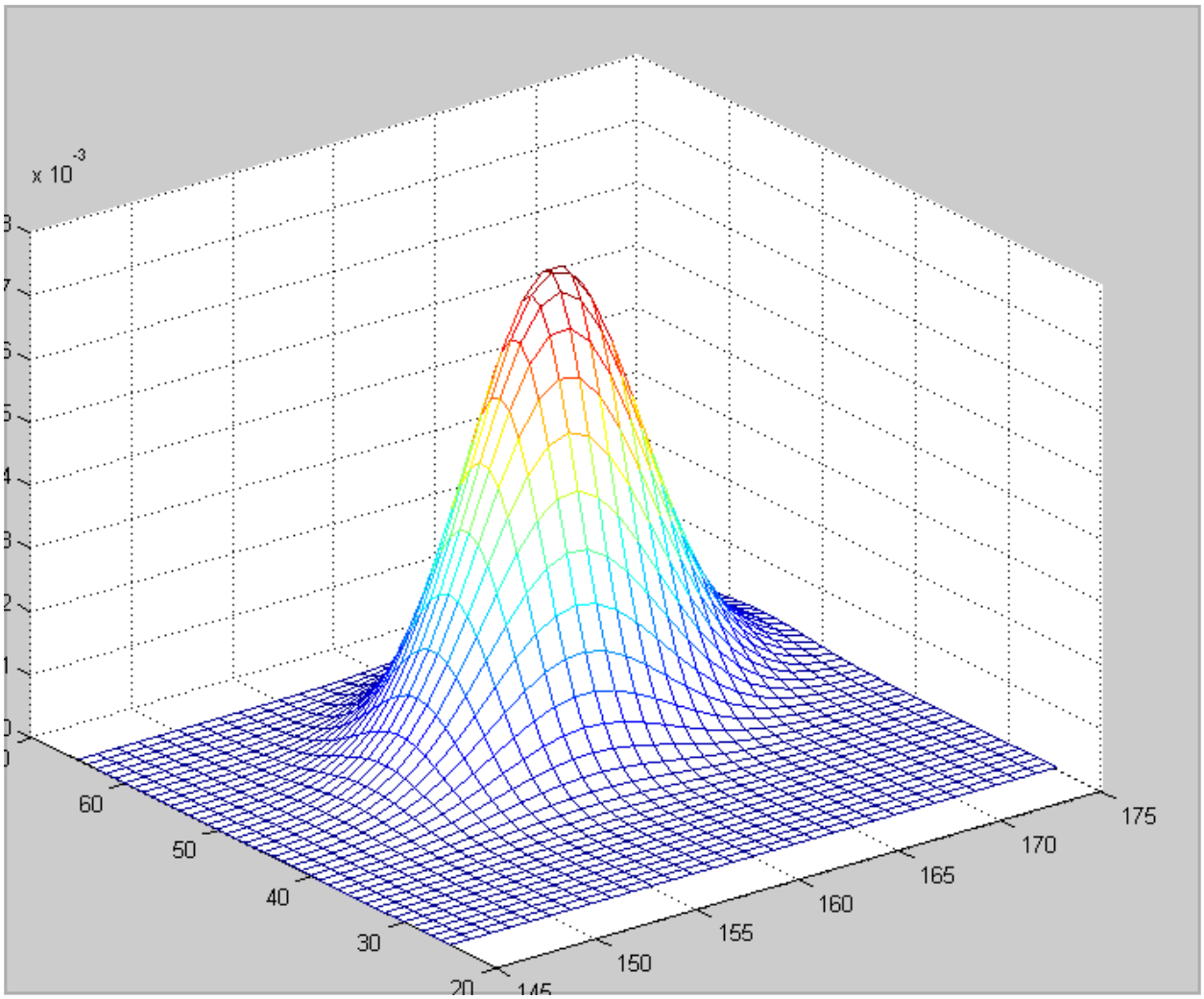
则有 $f_X(x)=\frac{1}{2\pi\sigma_1}e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}}\int_{-\infty}^{\infty}e^{-\frac{t^2}{2}}dt$

$$f_X(x)=\frac{1}{\sqrt{2\pi}\sigma_1}e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}},-\infty<x<\infty$$

同理

$$f_Y(y)=\frac{1}{\sqrt{2\pi}\sigma_2}e^{-\frac{(y-\mu_2)^2}{2\sigma_2^2}},-\infty<y<\infty$$

二维高斯分布图像：



二维高斯函数

N维高斯函数的通项公式：

$$G(r) = \frac{1}{\sqrt{2\pi\sigma^2}^N} e^{-r^2/(2\sigma^2)}$$

里面的参数中，r就是模糊半径，而在二维坐标系中，模糊半径就是x^2+y^2，σ 是正态分布的标准偏差，所以代入通项公式我们就可以得到二维的高斯函数公式，如下：

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

二维高斯函数公式生成的曲面的高线是从中心开始以正态分布辐向的同心圆.不为零的像素组成的矩阵（卷积）在原来的图像矩阵像素作做变换,每个像素的值都是相邻一圈的像素值的加权平均数.

高斯函数的权重矩阵和举例

一.我们要先计算出权重矩阵（所以要先得到矩阵的坐标,我们在这里以中间点的格子的坐标设置为（0，0）原点）

(-1,1)	(0,1)	(1,1)
(0,-1)	(0,0)	(1,0)
(-1,-1)	(0,-1)	xlxw (1,-1)

二.我们要通过高斯函数计算出图像的权重矩阵（我们定义半径为1）

0.058549	0.096532	0.058549
0.096532	0.159154	0.096532
0.058549	0.096532	xlxw 0.058549

三.由于我们要使得这个矩阵的所有数据的和为1，所以我们要计算出和的数据

$$\sum value = 0.167578$$

四.将矩阵中的九个数数据都除以所有数据的和，就可得到最终的权重矩阵，最终结果如下图：

0.075113	0.123841	0.075113
0.123841	0.204179	0.123841
0.075113	0.123841	xlxw 0.075113

五.我们已经得到了权重矩阵，就可以通过它来计算出各个像素的灰度值了.

灰度图像灰度矩阵的计算

假设我们随意创建一个任意灰度值（0-255）的矩阵

13	16	17
26	**28**	29
37	35	xlxw 38

计算方式很简单，就是每个点的像素乘以自己对应的权重值，如下图：

0.976476	1.981462	1.276931
3.219146	5.717038	3.591400
2.779203	4.334449	xlxw 2.854317

最后一步：将这九个点的数值加起来后就是中心点的高斯模糊后的值了，然后对于所有的点重复着这一个过程就可以得到高斯模糊化的图像.如下：

$$\sum center = 26.730422$$

引申为彩色图像

如果原图是彩色图片，则可以对彩色图像的RGB三个通道分别进行高斯模糊，这样得到的就会是彩色图像高斯模糊化后的图像.

代码举例

```
#对上文高斯模糊的方法阐释的相关代码  -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 = 3.1415926

def getrgb(path):#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.empty((sizepic[0],sizepic[1]))
    ng = np.empty((sizepic[0],sizepic[1]))
    nb = np.empty((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]
    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

def Matrixmaker(r):#通过半径和坐标计算高斯函数矩阵
    summat = 0
    timer[1] = clock()
    ma = np.empty((2*r+1,2*r+1))
    for i in range(0,2*r+1):
        for j in range(0,2*r+1):
            gaussp = (1/(2*PI*(r**2))) * math.e**(-((i-r)**2+(j-r)**2)/
(2*(r**2)))
            ma[i][j] = gaussp
            summat += gaussp
    for i in range(0,2*r+1):
        for j in range(0,2*r+1):
            ma[i][j] = ma[i][j]/summat
    print("已经计算出高斯函数矩阵，所花时间为{:.3f}s".format(clock()-timer[1]))
    print("矩阵如下:")
    return ma
```

```
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(r+1,sizepic[0]-r):
        for j in range(r+1,sizepic[1]-r):
            o = 0
            for x in range(i-r,i+r+1):
                p = 0
                for y in range(j-r,j+r+1):
                    #print("x{},y{},o{},p{}".format(x,y,o,p))
                    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

def cpic(r,g,b,path,rd):
    timer[3] = clock()
    pd = p.open(path)
    for i in range(rd+1,sizepic[0]-rd+1):
        for j in range(rd+1,sizepic[1]-rd+1):
            pd.putpixel((i,j),(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(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)
    matx = Matrixmaker(rd)
    print(matx)
    print("正在转换..")
    newr,newg,newb = newrgb(matx,nr,ng,nb,rd)
    print("正在准备输出..")
```

```
    cpic(newr,newg,newb,path,rd)
    print("{} - >> {}".format(path.split('/')[-1],"blurred.png"))
    print("总计耗时:{:.3f}s,感谢您的使
用.".format(timer[0]+timer[1]+timer[2]+timer[3]))
main()
```

程序运行的截图：

```
>>>
======= RESTART: /Users/xulvxiaowei/Desktop/Gossac blur/Gauss blur.py =======
请输入模糊的半径：3
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
矩阵如下：
[[ 0.01129725  0.01491455  0.01761946  0.01862602  0.01761946  0.01491455
   0.01129725]
 [ 0.01491455  0.01969008  0.02326108  0.02458993  0.02326108  0.01969008
   0.01491455]
 [ 0.01761946  0.02326108  0.02747972  0.02904957  0.02747972  0.02326108
   0.01761946]
 [ 0.01862602  0.02458993  0.02904957  0.03070911  0.02904957  0.02458993
   0.01862602]
 [ 0.01761946  0.02326108  0.02747972  0.02904957  0.02747972  0.02326108
   0.01761946]
 [ 0.01491455  0.01969008  0.02326108  0.02458993  0.02326108  0.01969008
   0.01491455]
 [ 0.01129725  0.01491455  0.01761946  0.01862602  0.01761946  0.01491455
   0.01129725]]
正在转换..
已经计算出新的三通道矩阵，所花时间为0.328s
正在准备输出..
已经完成生成，所花时间为1.933s
正在导出图片..
bt.png - >> blurred.png
总计耗时:2.820s,感谢您的使用.
```

高斯模糊的缺点&优化方案

用这种方法对图片进行高斯模糊，有几个缺点：

第一：由上一章程序的截图可知高斯模糊全过程总计花费2.82s，这个时间虽然是因为为了详细展示高斯模糊过程没有优化，但是和我们上文中提到的200ms相差甚远，这里就是因为有些地方需要改进.

第二：这种模糊办法无法有效解决边缘化问题，所以当模糊半径过大时会有明显的分界线的出现，这也是需要改进的地方

优化方案：

1.在速度上，主要用于改进算法，但是我们还可以从图像角度考虑，由于图像进行模糊后会丢失细节，这时候和我们直接对图片缩放有相似之处，所以我们如果将图片缩小一半，在再进行

模糊，之后再方法到原图，我们会发现图片模糊的效果和直接模糊没有太大区别.这个时候我们就能采用这样的方法优化.

2.另外在速度上，我们可以发现纯色模糊后是没有变化的，那么我们可以先找到大块的纯色区域模糊时跳过他们，这样子也能对时间进行优化.

3.优化算法.（利用对一个像素点权重的计算可以化简为两个一维高斯积分的乘积）的特性，这是国外的论坛上给出的一个优化方案.等等.