On the Effectiveness of Visible Watermarks

前景蒙版（alpha matte）

也称前景透明度或透明度蒙版，是前背景分离的结果，是一个灰度图，每一个像素点的灰度值表示原始图像每个像素属于前景物体的程度，白色代表某一个像素确定属于前景，黑色代表某一个像素确定属于背景。

$I = \alpha \times F + (1 - \alpha) \times B$

I: 图像 F: 前景图像 B: 背景图像　alpha matte α

如果再具体一点，针对每个像素，背景颜色为B = [RB,GB,BB]，前景对象颜色为F = [αRF, αGF, αBF]，于是matting方程为$I = F + (1 - \alpha) \times B$。通常在应用中都是求这个方程的解，以获得图像的最后分割结果。

$R = \alpha \times RF + (1 - \alpha) \times RB$
$G = \alpha \times GF + (1 - \alpha) \times GB$

$B = \alpha \times BF + (1 - \alpha) \times BB$

解法：灰度图。令RF = GF = BF。

高斯滤波

【转自：https://blog.csdn.net/lvquanye9483/article/details/81592574 / https://blog.csdn.net/Ibelievesunshine/article/details/104881204】

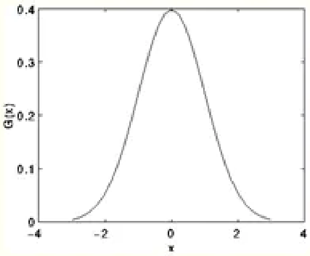
含义

高斯滤波就是对整幅图像进行加权平均的过程，每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到。

高斯函数

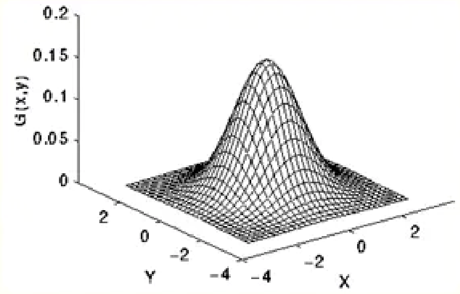
一维高斯分布：

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



二维高斯分布：

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



σ描述正态分布资料数据分布的离散程度，σ越大，数据分布越分散，σ越小，数据分布越集中。也称为是正态分布的形状参数，σ越大，曲线越扁平，反之，σ越小，曲线越瘦高。

高斯核

理论上，高斯分布在所有定义域上都有非负值，这就需要一个无限大的卷积核。实际上，仅需要取均值周围3倍标准差内的值，以外部份直接去掉即可。
高斯滤波的重要两步就是先找到高斯模板然后再进行卷积。

对二维高斯分布，不必纠结于系数$\frac{1}{(\sqrt{2*\pi*\sigma})^2}$，因为它只是一个常数！并不会影响互相之间的比例关系，而且最终都要进行归一化，所以在实际计算时我们忽略它而只计算后半部分 $e^{-((x-ux)^2+(y-uy)^2)/2\sigma^2}$

其中(x,y)为掩膜内任一点的坐标，(ux,uy)为掩膜内中心点的坐标，在图像处理中可认为是整数；σ是标准差。

例如：要产生一个3×3的高斯滤波器模板，以模板的中心位置为坐标原点进行取样。（x轴水平向右，y轴竖直向下）

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

（模板在各个位置的坐标，如上图所示↑）

这样，将各个位置的坐标代入到高斯函数中，得到的值就是滤波器的系数。

如果窗口模板的大小为 (2k+1)×(2k+1)，则：

$$H_{i,j}=\frac{1}{2\pi\sigma^2}e^{-\frac{(i-k-1)^2+(j-k-1)^2}{2\sigma^2}}$$

（窗口模板中各个元素的计算公式↑）

这样计算出来的模板有两种形式：小数和整数

小数形式的模板，就是直接计算得到的值，没有经过任何的处理；

整数形式的模板，需要进行归一化处理，将模板左上角的值归一化为1。使用整数的模板时，需要在模板的前面加一个系数，系数为模板中元素和的倒数。

例如，标准差σ =1.3 的 3*3 的整数形式的高斯滤波器如下：

$$K = \frac{1}{16} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

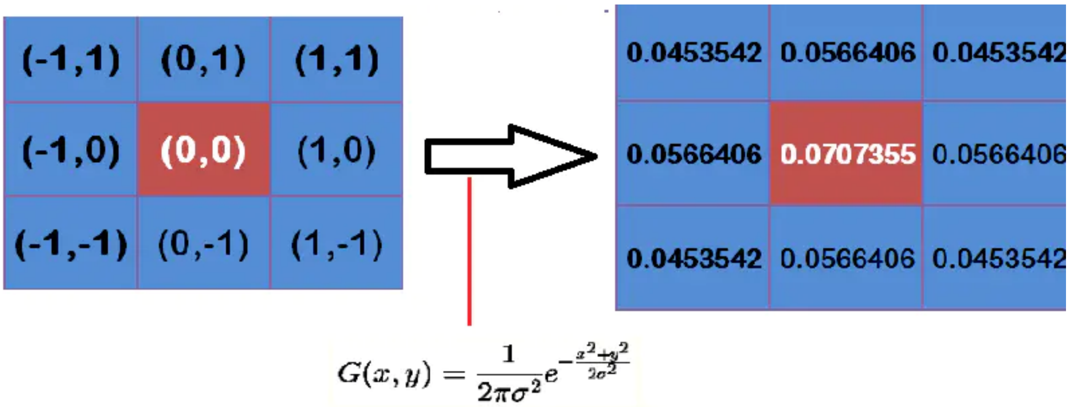
（标准差 σ=1.3 的8近邻高斯滤波器如图 ↑）

σ的意义及选取

通过上述的实现过程，不难发现，高斯滤波器模板的生成最重要的参数就是高斯分布的标准差σ。标准差代表着数据的离散程度，如果σ较小，那么生成的模板的中心系数较大，而周围的系数较小，这样对图像的平滑效果就不是很明显；反之，σ较大，则生成的模板的各个系数相差就不是很大，比较类似均值模板，对图像的平滑效果比较明显。

举个例子

假定中心点的坐标是（0,0），那么取距离它最近的8个点坐标，为了计算，需要设定σ的值。假定σ=1.5，则模糊半径为1的高斯模板就算如下



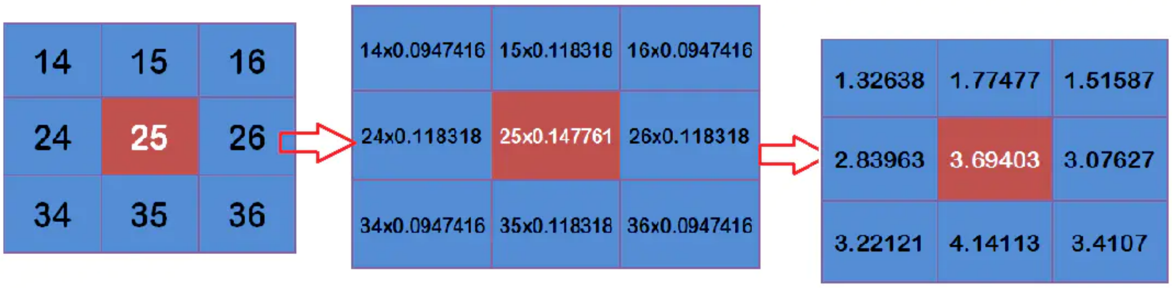
这个时候我们我们还要确保这九个点加起来为1（这个是高斯模板的特性），这9个点的权重总和等于0.4787147，因此上面9个值还要分别除以0.4787147，得到最终的高斯模板。

0.0947416	0.118318	0.0947416
0.118318	0.147761	0.118318
0.0947416	0.118318	0.0947416

高斯滤波计算

举个例子：

假设现有9个像素点，灰度值（0-255）的高斯滤波计算如下：



参考来源：（https://blog.csdn.net/nima1994/article/details/79776802）

将这9个值加起来，就是中心点的高斯滤波的值。
对所有点重复这个过程，就得到了高斯模糊后的图像。

由于图像的长宽可能不是滤波器大小的整数倍，因此我们需要在图像的边缘补0，这种方法叫做 zero padding 。

opencv函数实现高斯滤波

```
import cv2

img=cv2.imread('../paojie.jpg')

#(3, 3)表示高斯滤波器的长和宽都为3，1.3表示滤波器的标准差

out=cv2.GaussianBlur(img,(3,3),1.3)

cv2.imwrite('out.jpg',out)

cv2.imshow('result',out)

cv2.waitKey(0)

cv2.destroyAllWindows()
```

Canny边缘检测

【转自：https://www.cnblogs.com/mmmmc/p/10524640.html】

评价标准

Canny提出了一个对于边缘检测算法的评价标准，包括：

（1）以低的错误率检测边缘，也即意味着需要尽可能准确的捕获图像中尽可能多的边缘。

（2）检测到的边缘应精确定位在真实边缘的中心。

（3）图像中给定的边缘应只被标记一次，并且在可能的情况下，图像的噪声不应产生假的边缘。

简单来说就是，检测算法要做到：**边缘要全，位置要准，抵抗噪声的能力要强。**

实现步骤

step1：高斯平滑滤波

滤波是为了去除噪声，选用高斯滤波也是因为在众多噪声滤波器中，高斯表现最好。一个大小为(2k+1)x(2k+1)的高斯滤波器核（核一般都是奇数尺寸的）的生成方程式由下式给出：

$$H_{ij} = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{\left(i-(k+1)\right)^2+\left(j-(k+1)\right)^2}{2\sigma^2}\right);1\le i,j\le(2k+1) \quad (3\text{-}1)$$

下面是一个sigma = 1.4，尺寸为3x3的高斯卷积核的例子，注意矩阵求和值为1（归一化）：

$$H = \begin{bmatrix} 0.0924 & 0.1192 & 0.0924 \\ 0.1192 & 0.1538 & 0.1192 \\ 0.0924 & 0.1192 & 0.0924 \end{bmatrix}$$

高斯卷积核的大小将影响Canny检测器的性能。尺寸越大，去噪能力越强，因此噪声越少，但图片越模糊，canny检测算法抗噪声能力越强，但模糊的副作用也会导致定位精度不高，一般情况下，推荐尺寸5x5，3x3也行。

step2： 计算梯度强度和方向

边缘的最重要的特征是灰度值剧烈变化，如果把灰度值看成二元函数值，那么灰度值的变化可以用二元函数的”导数“（或者称为梯度）来描述。由于图像是离散数据，导数可以用差分值来表示，差分在实际工程中就是灰度差，说人话就是两个像素的差值。一个像素点有8邻域，那么分上下左右斜对角，因此Canny算法使用四个算子来检测图像中的水平、垂直和对角边缘。算子是以图像卷积的形式来计算梯度，比如Roberts，Prewitt，Sobel等，这里选用Sobel算子来计算二维图像在x轴和y轴的差分值（这些数字的由来？），将下面两个模板与原图进行卷积，得出**x和y轴的差分值图**，最后计算该点的梯度G和方向θ

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = arc\tan\left(G_y / G_x\right)$$

首先了解opencv的二维滤波函数：dst=cv.filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]])

dst： 输出图片

src： 输入图片

ddepth： 输出图片的深度, 详见 combinations,如果填-1，那么就跟与输入图片的格式相同。

kernel： 单通道、浮点精度的卷积核。

以下是默认参数：

anchor：内核的基准点(anchor)，其默认值为(-1,-1)表示位于kernel的中心位置。基准点即kernel中与进行处理的像素点重合的点。举个例子就是在上面的step1中，e=HA*得到的e是放在原像素的*33的哪一个位置，一般来说都是放在中间位置，设置成默认值就好。

delta： 在储存目标图像前可选的添加到像素的值，默认值为0。

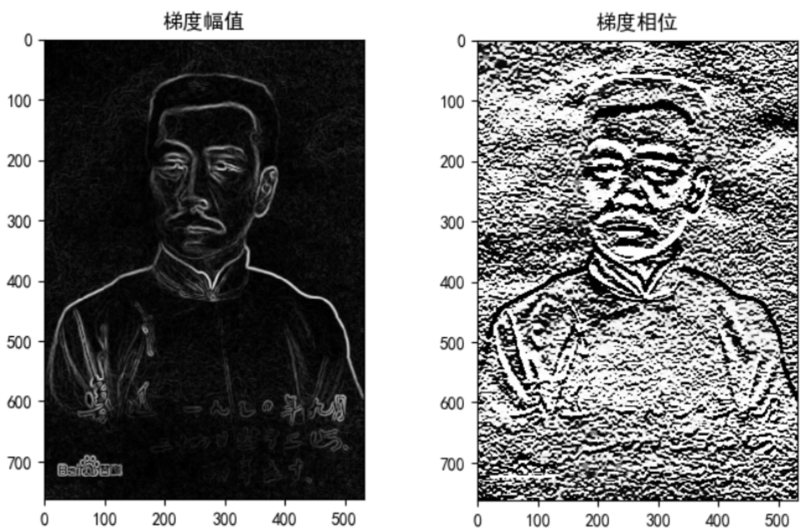
borderType：像素向外逼近的方法，默认值是BORDER_DEFAULT,即对全部边界进行计算。

```
import cv2
import numpy as np
import matplotlib.pyplot as plt


img=cv2.imread("images/luxun.png",cv2.IMREAD_GRAYSCALE)  # 读入图片
sobel_x = np.array([[-1, 0, 1],[-2,0,+2],[-1, 0, 1]])   # sobel的x方向算子
sobel_y = np.array([[1, 2, 1],[0,0,0],[-1, -2, -1]])    # sobel的x方向算子
sobel_x=cv2.flip(sobel_x,-1)           # cv2.filter2D()计算的是相关，真正的卷积需要翻转，再进行相关计算。
sobel_x=cv2.flip(sobel_y,-1)
# cv2.flip()第二个参数：等于0：沿着x轴反转。大于0：沿着y轴反转。小于零：沿着x轴，y轴同时反转

# 卷积 opencv是用滤波器函数实现的
img_x=cv2.filter2D(img,-1, sobel_x)
img_y=cv2.filter2D(img,-1, sobel_y)
# 画图 plt不支持中文，但是可以通过以下方法设置修复
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus'] = False

plt.subplot(221), plt.imshow(img_x, 'gray'),plt.title('sobel_x')
plt.subplot(222), plt.imshow(img_y, 'gray'),plt.title('sobel_y')
plt.subplot(223), plt.imshow(img_y+img_x, 'gray'),plt.title('sobel')
plt.subplot(224), plt.imshow(img, 'gray'),plt.title('原图')
plt.show()
```

需要注意一点：在图像处理领域，卷积运算的定义是先将核关于x轴和y轴反转，然在做相关运算。然而工程实践中往往跳过反转，用相关运算代替卷积（比如opencv）。如果你需要严格的卷积运算，应该注意原函数的具体实现方式。sobel算子天生关于中心对称，所以反转与否并不影响结果。

作者自己写的卷积函数：

```
def conv2d(src,kernel):  # 输入必须为方形卷积核
    # 本函数仍然是相关运算，没有反转。如果非要严格的卷积运算，把下面一行代码的注释取消。
    #kernel=cv2.flip(kernel,-1)
    [rows,cols] = kernel.shape
    border=rows//2                                # 向下取整 获得卷积核边长
    [rows,cols]=src.shape
    dst = np.zeros(src.shape)                     # 采用零填充再卷积，卷积结果不会
变小。
    # print("图像长：",rows,"宽：",cols,"核边界",border)
    # print(border,rows-border,border,cols-border)
    temp=[]
    for i in range(border,rows-border):
        for j in range(border,cols-border):
            temp=src[i-border:i+border+1,j-border:j+border+1]  # 从图像获取与核匹配
的图像
            # 切片语法：索引位置包括开头但不包括结尾 [start: end: step]
            dst[i][j]=(kernel*temp).sum()  # 计算卷积
    return dst
```

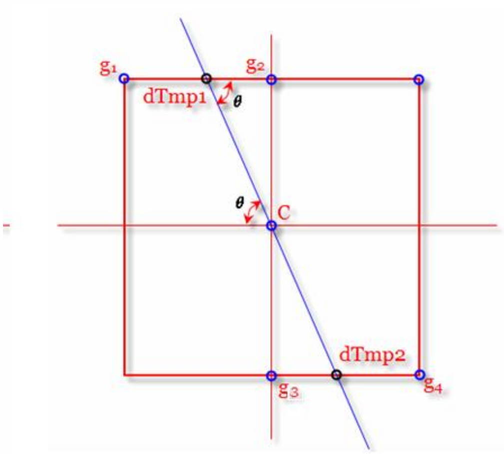


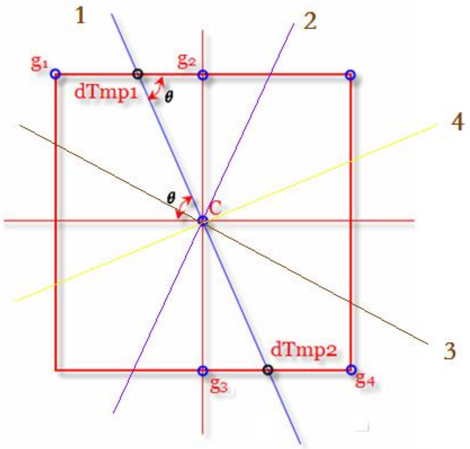
sobel算子计算的边缘很粗很亮，比较明显，但是不够精确，我们的目标是精确到一个像素宽，至于梯度相位就很难看出什么特征，并且梯度相位实际上是为了下一步打基础的。下面附上代码：

step3:非极大值抑制

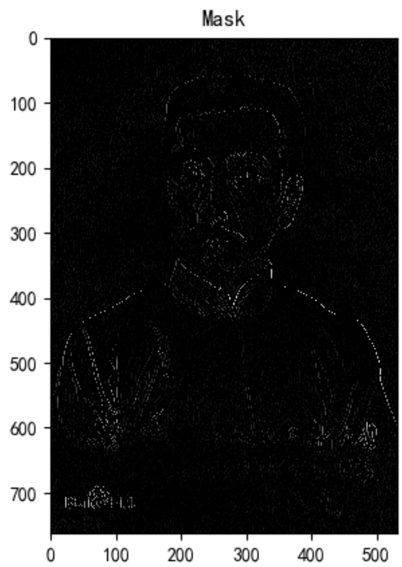
sobel算子检测出来的边缘太粗了，我们需要**抑制那些梯度不够大的像素点，只保留最大的梯度，从而达到瘦边的目的**。这些梯度不够大的像素点很可能是某一条边缘的过渡点。按照高数上二位函数的极大值的定义，即对点（x0，y0）的某个邻域内所有（x，y）都有f（x，y）≤(f（x0，y0），则称f在（x0，y0）具有一个极大值，极大值为f（x0，y0）。简单方案是判断一个像素点的8邻域与中心像素谁更大，但这很容易筛选出噪声，因此我们需要用梯度和梯度方向来辅助确定。

如下图所示，中心像素C的梯度方向是蓝色直线，那么只需比较中心点C与dTmp1和dTmp2的大小即可。由于这两个点的像素不知道，假设像素变化是连续的，就可以用g1、g2和g3、g4进行线性插值估计。设g1的幅值M(g1)，g2的幅值M(g2),则M(dtmp1)=w*M(g2)+(1-w)*M(g1) ，其w=distance(dtmp1,g2)/distance(g1,g2) 。也就是利用g1和g2到dTmp1的距离作为权重，来估计dTmp1的值。w在程序中可以表示为tan（θ）来表示，具体又分为四种情况（下面右图）讨论。





如下图，经过非极大值抑制可以很明显的看出去除了很多点，边缘也变得很细。在程序实现中，要注意opencv的默认坐标系是从左到右为x轴，从上到下是y轴，原点位于左上方，计算g1、g2、g3、g4的位置的时候，一定要小心。**经过非极大值抑制可以看出来图片的边缘明显变细**，很多看起来黑色的部分其实有值的，只是因为值太小了看不清楚，而这些黑色的部分可能是噪声或者其他原因造成的局部极大值，下一步我们就要用双阈值来限定出强边缘和弱边缘，尽可能的减少噪声的检出。

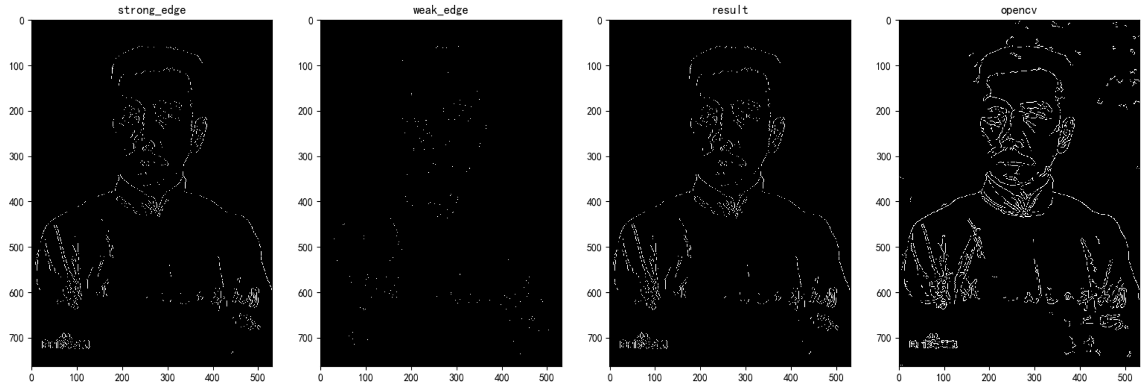



```
# step3:非极大值抑制
anchor=np.where(G!=0)  # 获取非零梯度的位置
Mask=np.zeros(img.shape)

for i in range(len(anchor[0])):
    x=anchor[0][i]
    y=anchor[1][i]
    center_point=G[x,y]
    current_theta=theta[x,y]
    dTmp1=0
    dTmp2=0
    W=0
    if current_theta>=0 and current_theta<45:
```

```
            #         g1          第一种情况
            # g4  C  g2
            # g3
            g1 = G[x + 1, y - 1]
            g2 = G[x + 1, y]
            g3 = G[x - 1, y + 1]
            g4 = G[x - 1, y]
            W=abs(np.tan(current_theta*np.pi/180))  # tan0-45范围为0-1
            dTmp1= W*g1+(1-W)*g2
            dTmp2= W*g3+(1-W)*g4

        elif current_theta>=45 and current_theta<90:
            #      g2 g1          第二种情况
            #      C
            # g3  g4

            g1 = G[x + 1, y - 1]
            g2 = G[x, y - 1]
            g3 = G[x - 1, y + 1]
            g4 = G[x, y + 1]
            W = abs(np.tan((current_theta-90) * np.pi / 180))
            dTmp1= W*g1+(1-W)*g2
            dTmp2= W*g3+(1-W)*g4

        elif current_theta>=-90 and current_theta<-45:
            # g1  g2              第三种情况
            #      C
            #      g4 g3
            g1 = G[x - 1, y - 1]
            g2 = G[x, y - 1]
            g3 = G[x + 1, y + 1]
            g4 = G[x, y + 1]
            W = abs(np.tan((current_theta-90) * np.pi / 180))
            dTmp1= W*g1+(1-W)*g2
            dTmp2= W*g3+(1-W)*g4

        elif current_theta>=-45 and current_theta<0:
            # g3                  第四种情况
            # g4  C  g2
            #         g1
            g1 = G[x + 1, y + 1]
            g2 = G[x + 1, y]
            g3 = G[x - 1, y - 1]
            g4 = G[x - 1, y]
            W = abs(np.tan(current_theta * np.pi / 180))
            dTmp1= W*g1+(1-W)*g2
            dTmp2= W*g3+(1-W)*g4

        if dTmp1<center_point and dTmp2<center_point:   # 记录极大值结果
                Mask[x,y]=center_point
#Mask=(Mask-Mask.min())/(Mask.max()-Mask.min())*256 #归一化
plt.imshow(Mask,'gray'),plt.title('Mask')
plt.show()
```

step4：用双阈值算法检测和连接边缘

双阈值法非常简单，我们假设两类边缘：经过非极大值抑制之后的边缘点中，**梯度值超过T1的称为强边缘，梯度值小于T1大于T2的称为弱边缘，梯度小于T2的不是边缘**。可以肯定的是，强边缘必然是边缘点，因此必须将T1设置的足够高，以要求像素点的梯度值足够大（变化足够剧烈），而弱边缘可能是边缘，也可能是噪声，如何判断呢？**当弱边缘的周围8邻域有强边缘点存在时，就将该弱边缘点变成强边缘点**，以此来实现对强边缘的补充。实际中人们发现T1:T2=2:1的比例效果比较好，其中T1可以人为指定，也可以设计算法来自适应的指定，比如定义梯度直方图的前30%的分界线为T1，我实现的是人为指定阈值。检查8邻域的方法叫边缘滞后跟踪，连接边缘的办法还有区域生长法等等。强边缘、弱边缘、综合效果、和opencv的canny函数对比如下：



作者总结：

实现结果还是很打击的，我检测到的边缘过于断续，没有opencv实现的效果好。查了一下opencv的源码，这里猜测两个可能的原因：源码里梯度的方向被近似到四个角度之一（0，45，90，135），但我用线性插值的的结果是梯度方向更精确，而过于精确-->过于严格-->容易受到噪声干扰，所以在非极大值抑制这之后，我比opencv少了更多的点，最终导致了边缘不够连续；第二个原因可能是边缘连接算法效果不够好，把图象放大来看，我产生的边缘倾向于对角线上连接，而opencv的边缘倾向于折线连接，因此opencv的边缘更完整连续，而我的边缘更细，更容易断续。

论文中的公式

$$J(p) = \alpha(p)W(p) + (1 - \alpha(p))\, I(p), \qquad (1)$$

$$I(p) = \frac{J(p) - \alpha(p)W(p)}{1 - \alpha(p)}. \qquad (2)$$

J: watermarked image W: watermark   I: natural image

$$J_k = \alpha W + (1 - \alpha) I_k, \;\; k = 1, \cdots, K \qquad (3)$$

$\{I_k\}$: 图像集

Our goal is to recover W, α and $\{I_k\}_{k=1}^{K}$ given $\{J_k\}_{k=1}^{K}$.

This *multi-image matting* problem is still under-determined as there are $3K$ equations and $3(K+1)+1$ unknowns per pixel, for K color images. However, the coherency of W and α over the image collection, together with natural image priors, allow solving it to high accuracy, fully automatically.

3.1. Initial Watermark Estimation & Detection

I. Estimating the Matted Watermark

$$\nabla \widehat{W}_m(p) = \text{median}_k(\nabla J_k(p)). \qquad (4)$$

As the number of images K increases, Eq. 4 converges to the gradients of the true matted watermark, $W_m = \alpha W$, up to a shift (see Fig. 3). To demonstrate why that is the case, we treat I_k and J_k as random variables, and compute the exception $E[\nabla J_k]$. Using Eq. 3 we have,

$$\begin{aligned} E[\nabla J_k] &= E[\nabla W_m] + E[\nabla I_k] - E[\nabla(\alpha I_k)] \\ &= \nabla W_m + E[\nabla I_k] - \nabla \alpha E[I_k] - \alpha E[\nabla I_k] \\ &= \nabla W_m - \nabla \alpha E[I_k], \qquad (5) \end{aligned}$$

use Canny with 0.4 threshold

use Poisson reconstruction to obtain the initial matted watermark $\widehat{W}_m \approx W_m$.

II. Watermark Detection

use Chamfer Distance to detect the watermark in each of the images

3.2. Multi-Image Matting and Reconstruction

arg min F(x,y)： 当F(x,y)取得最小值时，变量x,y的取值

$$\begin{aligned} \arg\min_{W,\alpha,\{I_k\}} \sum_k \Big(& E_{\text{data}}(W,\alpha,I_k) + \lambda_I E_{\text{reg}}(\nabla I_k) \Big) \\ & + \lambda_w E_{\text{reg}}(\nabla W) + \lambda_\alpha E_{\text{reg}}(\nabla \alpha) + \beta E_f(\nabla(\alpha W)). \quad (6) \end{aligned}$$

$$E_{\text{data}}(I_k, W, \alpha) = \sum_p \Psi\left(|\alpha W + (1-\alpha) I_k - J_k|^2\right), \quad (7)$$

where $\Psi(s^2) = \sqrt{s^2 + \epsilon^2}$, $\epsilon = 0.001$ is a robust function that approximates L_1 distance (p is omitted for brevity).

惩罚项

$$E_{\text{reg}}(\nabla I) = \sum_p \Psi(|\alpha_x| I_x^2 + |\alpha_y| I_y^2), \quad (8)$$

使

得水印图像平滑

$$E_f(\nabla W_m) = \sum_p \Psi(\|\nabla W_m - \nabla \widehat{W}_m\|^2). \quad (9)$$

保持真实性

Optimization

Optimization The resulting optimization problem (Eq. 6) is non-linear and the number of unknowns may be very large when dealing with a large collection ($O(KN)$ unknowns, where N and K are the number of pixels per image, and number of images, respectively). To deal with these challenges, we introduce auxiliary variables $\{W_k\}$, where W_k is the watermark of the k^{th} image. Each per-image watermark W_k is required to be close to W. Formally, we rewrite the objective as follows

$$\arg\min \sum_k (E_{\text{data}}(I_k, W_k, \alpha) + \lambda_I E_{\text{reg}}(\nabla I_k) + \lambda_w E_{\text{reg}}(\nabla W_k) + \lambda_\alpha E_{\text{reg}}(\nabla \alpha) + \beta E_f(\nabla(\alpha W_k)) + \gamma \sum_k E_{\text{aux}}(W, W_k), \quad (10)$$

where $E_{\text{aux}}(W, W_k) = \sum_p |W - W_k|$.

Using these auxiliary variables, we solve smaller and simpler optimization problems (using alternating minimization). The resulting iterative algorithm consists of the following steps.

I. Image–Watermark Decomposition

I. Image–Watermark Decomposition At this step, we minimize the objective w.r.t. W_k, and I_k, while keeping α and W fixed. Thus, the optimization in Eq. 10 reduces to:

$$\arg\min_{W_k, I_k} E_{\text{data}}(I_k, W_k) + \lambda_I E_{\text{reg}}(\nabla I_k) + \lambda_w E_{\text{reg}}(\nabla W_k) + \beta E_f(\nabla(\alpha W_k)) + \gamma E_{\text{aux}}(W, W_k). \quad (11)$$

We solve this minimization problem using Iteratively-Reweighed-Least-Square (IRLS), where the resulting linear system is derived in Supplementary Materials (SM).

【！ 迭代加权最小二乘法】

II. Watermark Update

估计全局W -> 使得Eaux最小 -> 求{Wk}平均值

$$W = \text{median}_k W_k.$$

III. Matte Update

III. Matte Update. Here, we solve for α, while keeping the rest of the unknowns fixed. In this case, we minimize the following objective over α:

$$\sum_k E_{\text{data}}(\alpha, I_k, W) + \lambda_\alpha E_{\text{reg}}(\nabla\alpha) + \beta E_f(\nabla(\alpha W)). \quad (12)$$

Here too, the solution is obtained using IRLS (the final linear system is derived in the SM).

These steps are iterated several times until convergence.

Matte and Blend Factor Initialization

$$\alpha = c \cdot \alpha n$$

【！ 自适应阈值】初始化matte

【最小二乘法】初始化c

去水印：利用W和a