Poisson image editing-代码实现-python 版

Dezeming Family

2023 年 3 月 8 日

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列电子书，可以从我们的网站 [https://dezeming.top/] 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

目录

一　引文

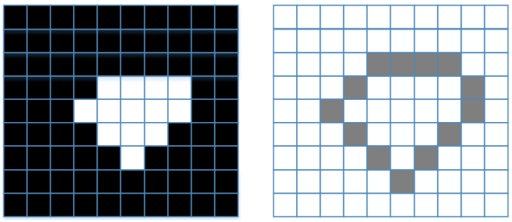
在介绍代码复现之前，还是先详细解释一下论文 [1] 中的几个比较重要的公式：

$$\text{for all } p\in\Omega,\quad |N_p|f_p-\sum_{q\in N_p\cap\Omega} f_q=\sum_{q\in N_p\cap\partial\Omega} f^*_q+\sum_{q\in N_p} v_{pq}. \quad (7) \qquad |N_p|f_p-\sum_{q\in N_p} f_q=\sum_{q\in N_p} v_{pq}. \quad (8)$$

$$\text{for all } \langle p,q\rangle,\ v_{pq}=g_p-g_q, \quad (11) \qquad v_{pq}=\begin{cases} f^*_p-f^*_q & \text{if } |f^*_p-f^*_q|>|g_p-g_q|,\\ g_p-g_q & \text{otherwise,}\end{cases} \quad (13)$$

我们现在假设源图像 g 和目的图像（背景图像）以及 mask 都是同样宽高的（如果不同，可以通过裁剪部分区域来得到同样宽高的子图像）。

下图中，左边是 mask，白色区域对应于 Ω 区。根据论文里的描述，边界点就是那些不属于 Ω 区但是四邻域与 Ω 有交集的区域，也就是左图中的灰色区域：

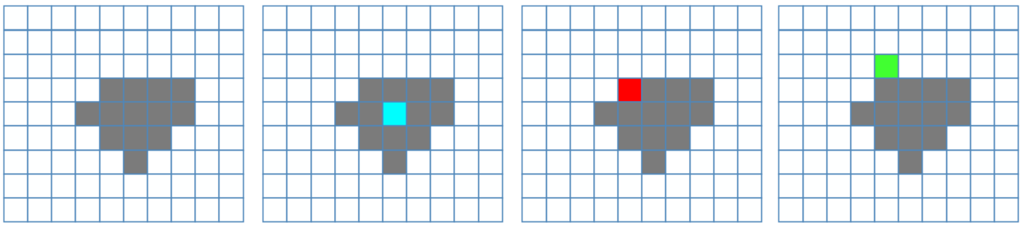


关键在于怎么把公式 (7) 转换为线性系统 $A\mathbf{x}=\mathbf{b}$ 这种形式，这里我会用较多篇幅来详解。

首先注意公式 (11)，代入到公式 (7) 中的 $\sum_{q\in N_p} v_{pq}$ 中，就发现其实就是中心样本的四倍然后减去周围样本的值，因此其实 $\sum_{q\in N_p} v_{pq}$ 就是对源图 g 做一个拉普拉斯，取其负数。我们记录为 $-gL_p$，即源图的 Laplacian 在 p 位置的值。

既然要保证边界值相等，这就意味着，生成的图像中，Ω 以外的区域（包括边界上的像素）的值要等于背景图的值。

我们把像素分三种情况，分别对应于下图的第 2、3、4 张图像，第一张图的灰色区域表示 Ω 区；第二张图是第一种情况，亮蓝色点是四邻域完全在 Ω 内的点；第三张图是第二种情况，红色点表示四邻域与边界相交，但是点本身在 Ω 内的点；第四张图是第三种情况，绿点表示在边界上的点。



第一种情况：此时，公式 (7) 中的 $\sum_{q\in N_p\cap\partial\Omega} f^*_q$ 值为 0。我们设 $p=(i,j)$（第 i 排第 j 列），那么四邻域就是 $[(i-1,j)\ (i+1,j)\ (i,j-1)\ (i,j+1)]$。

对于我们要得到的目标 $A\mathbf{x}=\mathbf{b}$，矩阵 A 是一个 $n*n$ 的矩阵，其中，n 是图像的像素数，也就是宽 w 乘以高 h。则对于矩阵 A，先将其初始化为一个单位矩阵，除了对角线其余值都是 0。$\mathbf{x}$ 是目标待求图像，被拉伸为 $(w*h,1)$。$\mathbf{b}$ 也是被拉伸为 $(w*h,1)$。

对于矩阵 A，其第 m 行与 $\mathbf{x}$ 各个元素相乘再相加，就得到了 $\mathbf{b}$ 中的第 m 个元素（这个过程其实就是公式 (7)）。矩阵 A 的第 m 行用于对像素 (i,j)（$m=i*w+j$）进行约束。对应于 (i,j) 位置，也就是 A 中的 $(i*w+j,i*w+j)$ 位置，值就设为 4，而四邻域对应于 A 中的位置的值都是 -1（比如，四邻域之一的 $(i-1,j)$ 在 A 中的位置是 $(i*w+j,(i-1)*w+j)$，注意它也在 A 中的第 $i*w+j$ 行）。

$\mathbf{b}$ 中第 $i*w+j$ 个元素是 $-gL_p=-gL_{(i,j)}$。

在实际 python 实现时，我们让 A 中的 $(i*w+j,i*w+j)$ 位置为 -4，四邻域对应于 A 中的位置的值都是 1，$\mathbf{b}$ 的第 $i*w+j$ 个元素是 $gL_p=gL_{(i,j)}$。其实只是正负号的区别罢了（公式 (7) 等式两边都取负号）。

第二种情况：在这张 9×10 的图像上，红点表示的像素是 $(3,4)$。对于该点的公式 (7) 中的 $\sum_{q\in N_p\cap\partial\Omega} f^*_q$ 值不再是 0。$N_p\cap\partial\Omega$ 区间有两个像素，$N_p\cap\Omega$ 区间也有两个像素。所以 A 中，$(3*10+4,3*10+4)$ 的位置是 4，$(3,5)$ 和 $(4,4)$ 这两个像素是属于 Ω 的邻域，所以置为 1（其在 A 中对应的位置是 $(3*10+4,3*10+5)$ 和 $(3*10+4,4*10+4)$）。

b 中第 $3*10+4$ 个元素的值是 $-gL_{(3,4)}+f^*_{(2,4)}+f^*_{(3,3)}$（实际实现时因为公式 (7) 等式两边都取了负号，所以代码中是 $gL_{(3,4)}-f^*_{(2,4)}-f^*_{(3,3)}$）。

第三种情况：对于第三种情况，$p=(2,4)$ 不在 Ω 内，因此不被公式 (7) 约束。此时 A 中的第 $2*10+4$ 行元素除了第 $(2*10+4)$ 列元素为 1 以外（该元素在矩阵 A 的对角线上），其他元素值都是 0。**b** 中第 $2*10+4$ 个元素值为背景图像值 $f^*_{(2,4)}$。

当 Ω 包含有 S 的边界，则有些区域没有 $N_p\cap\partial\Omega$，公式 (7) 会简化为公式 (8)。很多人在实现时，即使像素不在 S 边界，也会直接使用公式 (8)，这样相当于一种近似的实现，实现起来比较简单（比如 [2]）。我参考过 [2] 的代码，对该代码进行了优化和完善。

二　源码实现

前面说过，假设源图像 g 和目的图像（背景图像）以及 mask 都是同样宽高的。但实际可能它们都不相同，解决方案就是从原图中抠出等同大小的 g 和目的图像（背景图像）以及 mask。

源码实现在代码总目录 [4] 下。

源码有两个 python 文件，source1.py 和 source2.py 里面的程序基本都是一样的（泊松求解程序），只是操作的图像不同。images1 目录下的图像需要先裁剪，然后求解高斯融合，而 images2 目录下不需要先裁剪。images2 目录下有两个 mask，都是可以用的。

关于 mask 的制作，我是在 ps 上对源图做的 mask。

由于前面一节已经进行了详细讲解，这里我们就不再花过多精力介绍了，源码中也进行了非常详细的注释。

参考文献

[1] Pérez, Patrick, Michel Gangnet, and Andrew Blake. "Poisson image editing." ACM SIGGRAPH 2003 Papers. 2003. 313-318.

[2] https://zhuanlan.zhihu.com/p/355055346

[3] https://cloud.tencent.com/developer/article/2066941

[4] https://github.com/feimos32/ComputerVision-Code-Implementation-and-Collection