图像滤镜效果实现——晶格化实验报告

软件51 谢运帷 2015013185

一、实验算法

根据助教的作业说明PPT，此晶格化效果类似于马赛克，可以通过超像素方法结合块内的均值实现。由此我们可以知道，实现晶格化算法最重要的是实现超像素算法。而超像素算法中最良心，效果也较好的算法即是SLIC算法。此代码原文即给出了C++版本实现，虽然实现的效率感人（很慢，并不是夸它），代码风格也及其难读，但整体行文思路还是很好，提出了一种KMeans的简单有效的聚类方法来实现超像素化。

由于SLIC算法实在是太有名，效果也很好，在python库skimage有现成的实现，~~一行代码即可完成任务~~。根据作业要求，我使用C++从头到尾实现了此算法（仅使用了opencv中的io、图像空间变换功能）。下面先简单介绍一下此算法：

Algorithm 1 SLIC superpixel segmentation

/ Initialization */*
Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S.
Move cluster centers to the lowest gradient position in a 3×3 neighborhood.
Set label $l(i) = -1$ for each pixel i.
Set distance $d(i) = \infty$ for each pixel i.

repeat
  / Assignment */*
  for each cluster center C_k **do**
    for each pixel i in a $2S \times 2S$ region around C_k **do**
      Compute the distance D between C_k and i.
      if $D < d(i)$ **then**
        set $d(i) = D$
        set $l(i) = k$
      end if
    end for
  end for
  / Update */*
  Compute new cluster centers.
  Compute residual error E.
until $E \leq$ threshold

简单概括地说：

第一步，在图像空间均匀生成K个种子（聚类中心），间距$S = sqrt(width * height/K)$

第二步，把这些聚类中心移动到其3*3范围内梯度最小的位置，避免初始中心点落入图像的边缘或者噪音点

第三步，对于每一个聚类C_k，遍历其$2S*2S$范围内的邻居点，如果这个邻居点离此聚类C_k最近（包括几何距离和颜色空间距离），那么就把这个邻居点加到此聚类C_k中

第四步，更新每一个聚类的中心（成员点的几何平均），更新这个聚类的颜色信息（成员点的颜色空间平均）

在实际操作中，不需要确定这个残差E的阈值。大致循环第三步和第四步5次，即可保证超像素生成的质量，又不会使执行时间过长。

值得补充一点，上述距离由以下三个公式定义：

$$\begin{aligned} d_c &= \sqrt{(l_j - l_i)^2 + (a_j - a_i)^2 + (b_j - b_i)^2} \\ d_s &= \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \\ D' &= \sqrt{\left(\frac{d_c}{N_c}\right)^2 + \left(\frac{d_s}{N_s}\right)^2}. \end{aligned}$$

其中，N_c定义了颜色空间距离的权值，而N_s定义了几何空间距离的权值。通常情况下，我们会选取$N_s = S$，而N_c为一$[1, 40]$间的常数（实验中选择了30）。此种距离度量在超像素的生成中十分有效，它保证了一个超像素中的点既在几何空间中相近，在颜色空间上也类似。这种距离度量方法也十分易于扩展到三维空间中。

通过上述的算法，通过K个种子点生成除了它们分别的聚类。但这样的聚类结果有一个问题，它不能保证聚类的空间连续性，即有可能有两部分连通分量属于同一个聚类。原文给出了一个解决办法。它不断地选择图像中的连通分量，如果这个连通分量中包含较多的点，那么它就作为一个超像素保存下来；如果一个找到的连通分量中点数过少，我们就把它合并它周围的一个现有的连通分量中。

经过这最后一部连通分量的处理，我们就得到了一个较好的超像素聚类结果。我们再将同一聚类中的像素点均设置为此超像素的平均颜色，就得到了晶格化马赛克效果！

二、实验结果

我使用Lena图作为输入，在不同的种子点个数下进行了晶格化实验，下图展现了不同K下的晶格化结果。

可以发现，在1000的种子数下，图片的细节更多~晶格化效果看上去更加自然一些。

三、实验环境

我的电脑的实验环境win10 + Visual Studio 2015 Community + opencv3.3.0 + Qt5.7.1

如需编译我的整个工程，需要正确的配置opencv的路径。到opencv的网站上下载opencv3.3.0的Win Pack，解压路径选择C:\，然后在系统环境变量path中添加opencv3.3.0的vc14\bin路径。具体过程可以参照此篇 blog。或者你可以自己修改VS的编译链接属性，使用其他版本的opencv。

另外，还需要安装Qt5.7.1。具体过程可以参考此篇 blog。

运行环境应该已经配置好了，在exe文件夹下，可以直接双击运行。

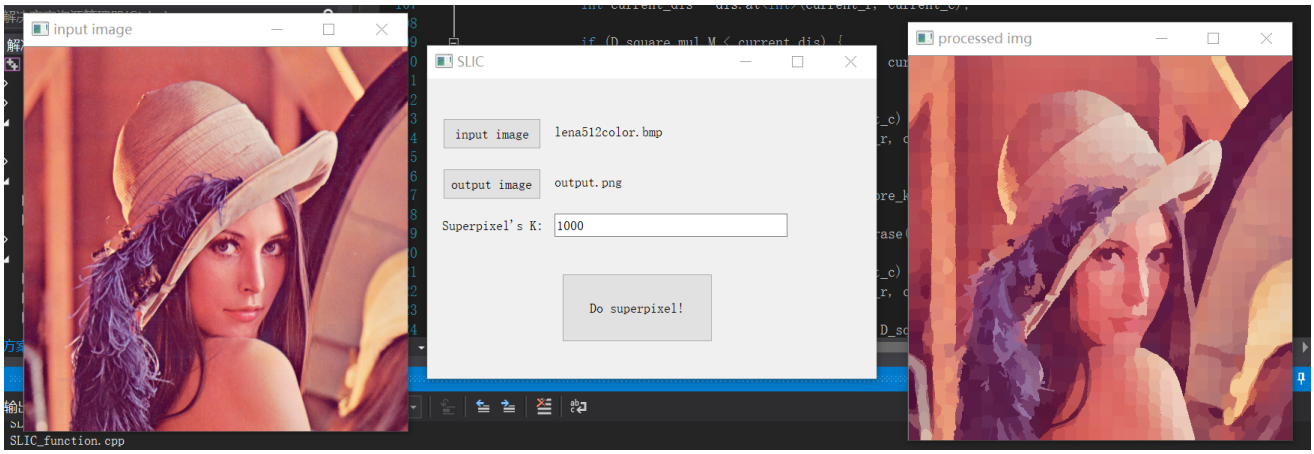
四、代码优化

我参考原文的代码实现，进行了一部分的优化，优化的过程也参考了此篇 **代码阅读&优化报告**。本文提出了很多的优化空间，我有针对性的实现了以下几个：

1. 计算距离时采用整形，避免浮点数
2. 存储时使用尽量小的空间
3. 简化梯度计算

当然，我的C++实现没有达到本篇博客的效果，不过在release模式下可以在1秒内给出晶格化结果，~~虽然我运行原版本代码时大概也是这个速度，可能是图片太小了~~。

五、运行



运行时会先弹出中间的窗口

1. 点击input image按钮输入图片，默认为本文件夹下的lena图
2. 点击output image按钮编辑输出的路径，文件名固定为output.png，默认为本文件夹下的output.png
3. 在输入框中输入超像素的个数K，默认为效果还不错的1000

最后点击Do superpixel!进行超像素的运算，随即弹出左右两个图片预览，一个是输入图片，一个是晶格化图片。与此同时，晶格化图片也会保存到输出路径中。

六、其他参考文献

论文中文翻译版

一个不完全的Python版本实现