申请上海交通大学学士学位论文

基于四元小波变换的图像彩色化方法

论文作者 李粤龙

学　　号 5090309241

指导教师 徐奕

专　　业 信息工程

答辩日期

Submitted in total fulfilment of the requirements for the degree of
Bachelor
in Information Engineering

IMAGE COLORIZATION TECHNIQUES BASED ON QUATERNION WAVELET TRANSFORM

Yuelong Li

Supervisor

Yi Xu

Department of Eletrical Engineering, School of Eletrical
Information and Electronic Engineering
Shanghai Jiao Tong University
Shanghai, P.R.China

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：_____

日　期：_____年 ____月 ____日

上海交通大学
学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保　密 □，在 _____ 年解密后适用本授权书。

本学位论文属于

不保密 □。

（请在以上方框内打“✓”）

学位论文作者签名：_____　　指导教师签名：_____

日　期：_____年 ___月 ___日　　日　期：_____年 ___月 ___日

基于四元小波变换的图像彩色化方法

摘　　要

黑白图像的彩色化源起源起黑白电影的彩色化，近年来引起了广泛的研究兴趣。在本课题组的前期工作中，已经实现了基于空间分布熵的自动 scribble 生成算法和基于四元 Gabor 变换和 Canny 边缘检测的图像彩色化方法。由于目前的算法实现仅在 Matlab 平台上，可移植性差且运算效率较低，且流程上存在一些不合理之处。因此，本论文尝试将这一算法移植到 C++ 平台上，并修正了该算法原有的一些问题。并对算法进行了性能评估。最后，我们结合实验结果，讨论了有关该算法存在的一些缺陷，以及未来的工作展望。

关键词： 图像处理　彩色化　四元 Gabor 变换

IMAGE COLORIZATION TECHNIQUES BASED ON QUATERNION WAVELET TRANSFORM

ABSTRACT

Colorization of grayscale image, originating from coloring black and white movies, has attracted much research interest in recent years. In our previous work, we have developed an automatic scribble generation algorithm based on spatial entropy distribution, and a colorization method based on quaternion Gabor transform and Canny edge detection. Because we only implement this algorithm on Matlab platform, it is quite inefficient, showing poor portability, and some minor flaws. Therefore, we managed to implement it via C++, and correct some minor issues. Additionally evaluation is carried upon this implementation. Finally, we discuss certain limitations of this algorithm and some future works according to our experimental results.

KEY WORDS: Image processing, Colorization, Quaternion Gaboor transform

目　　录

表格索引

插图索引

第一章　绪论

1.1　彩色化问题产生的背景

黑白图像彩色化是计算机视觉领域一个非常有意思的课题。这一问题可以追溯到非常久远的年代。在彩色摄像技术发明之前，人们拍摄了大量的黑白图像；在彩色摄像技术出现以后，如何将现有的大量黑白图像加上色彩，就自然成为了人们关心的问题。在电子计算机出现之前，艺术家常常需要手工给黑白图像进行上色。这一方法非常的乏味单调，且耗费时间，无论从劳动力还是时间上都是极大的浪费。另外，传统的手工上色方法需要操作人员具有极高的技术水平和艺术水准，这也构成了手工上色方法的一大局限性。

在电子计算机问世后，人们开始寻求自动上色的方案。Wilson Markle 应该是最早将计算机技术应用于彩色化的人；他于 1970 年尝试利用计算机技术将黑白影像加上色彩。此后，越来越多的人开始寻找借助于计算机技术的上色方法。由于黑白摄像技术存在的时间跨度非常长，由此积累了相当数量的黑白影像，因此如果能找到一个高效准确的自动上色方法，将使得对旧电影翻新的成本大幅降低。此外，黑白图像彩色化技术还可以被应用于其他的一些用途，比如：颜色的编辑和修改，压缩等等。因此，黑白图像彩色化是一个非常有实际意义和广阔前景的研究课题。

1.2　彩色化问题的研究现状

然而，目前对于这一课题，还没有非常令人满意的研究结果。之前的一些做法或者需要大量的人工干预，或者准确性不高，或者两者兼具；典型的方法是先将图像分割成各个区块，然后对每个区块分别涂上色彩。这种方法既费时间，又容易引起错误的结果。特别的，当边缘信息较为模糊时，这一做法常常带来明显的失真像素 [1]。对于电影的上色，这一做法则更不经济。

Levin [1] 的方法由用户预先指定一些（彩色化过的）scribble，然后以这些 scribble 为起始点，逐步蔓延到整幅图像来达到彩色化的目的。这一方法降低了人工干预的程度，也引起了许多人的关注。然而，这一方法的运算性能较低，

而且容易引起色彩的泄漏。Yatzi [2] 致力于提出一种运算效率高，而又不牺牲处理性能的改进算法。他从测地学最短路径的快速计算方法得到启发，采用一种调和最短路径的方法，在基本保证处理结果准确的前提下大幅改进了运算性能。Daniel [3] 则是采用一种概率松弛的方法，专门研究了卡通图像彩色化算法。

Levin 的算法还有一个问题就是在计算权值的时候，只考虑像素间的灰度差异，从而不可避免地忽视了一些结构信息。有鉴于此， [4] [5] [6] 等尝试加入图形的几何结构，以期改进处理效果。在本课题组先前的工作中，Ding [1] [7] 提出了采用四元小波变换来衡量图像的结构差异性。同时，针对原算法中需要人工选择 scribble 的不足，一种基于空间分布熵的自动产生 scribble 的算法也被提了出来。这两者以及一些细节上的改进，构成了较为完备的一套彩色化方法。实验结果显示，这一方法相对于原算法的处理效果有一定改进，也避免了有些情形下出现的颜色泄漏等问题。

然而，Ding 的算法目前只在 Matlab 平台上予以实现，C++ 实现尚处在初步阶段。此外，现有的 Matlab 代码效率比较低下，而且存在一些细节上的问题。比如，颜色泄漏问题还是没有彻底解决；此外，scribble 生成算法应用的插值方法也比较粗糙。因此，我们仍需要较多的改进与实现工作，包括将代码移植到 C++ 平台上，并对现有的问题给出改进方案。

1.3 本文主要内容和安排

第二章我们回顾当下比较主流的彩色化算法，对其进行分类，并重点讨论了 Levin 的算法以及基于他的算法衍生出来的一些方法。在这个基础上，对于我们课题组前期的工作作出了概括，并明确了本文的工作重点。

第三章从算法的层面上，探讨了前期工作存在的一些不足，提出了对应的改进方案，以及针对之前未能重视的细节问题进行了详尽的讨论。

第四章我们针对彩色化算法的 C++ 实现，将整个算法细分为各个模块，介绍了各个模块各自的功能及具体实现，并讨论了实现过程中需注意的一些问题以及算法的效率问题等。

结论部分总结了我们现有的工作成果，对现有算法依然存在的局限性进行了总结和概括，并对后续工作提出了一些建议和初步的规划。

第二章　黑白图像彩色化技术的综述

2.1　彩色化问题的研究现状

彩色化这一术语最早于 1970 年由 Wilson Markle 提出，当时是用于给黑白影像增加色彩。现今这一术语被广泛用来指代各种给黑白图像增加颜色的方法。

将彩色图像转换为黑白图像，主要应用的是公式 2–1（亮度公式）或 2–2（照度公式）。由于这两个公式均非可逆，因此彩色化实际上是一个病态问题。对于同一个灰度值，在采用单字节（8 比特）存储方式下，有可能存在 256×256 种不同的色彩组合。

$$Gray = (R + B + G)/3 \tag{2–1}$$

$$Gray = 0.299R + 0.587G + 0.114B \tag{2–2}$$

按照在彩色化过程中人为干预的程度，可以将彩色化方法分为三大类 [8]：手工上色，半自动上色和自动上色。

1. 手工上色：借助 Photoshop、BlackMagic 等软件人为地对图像涂抹色彩。这一方法常常需要人为对图像进行分割，需要较多的时间和劳动力。

2. 半自动上色：寻找一种映射关系，从照度直接映射到色度。通常是借助于一个查找表（LUT）来实现。这一方法的一个典型例子就是伪彩色图像。这些方法在医学图像处理上应用比较广泛 [9]。

3. 自动上色方法是被研究得最为广泛的一类上色方法，涵盖了非常广泛的范畴。这一方法又可以被粗略的划分为三种类型：变换型，图像匹配型和用户选择型。

   (a) 变换型：对各个灰度值 $I_g(x, y)$ 应用某个变换方程 T_k，得到各个色度 $I_{ck}(x, y)$ 的值。见公式 2–3。

   $$I_{ck}(x, y) = T_k[I_g(x, y)] \tag{2–3}$$

其中，k 表示第 k 个通道。对于每个通道，都对应一个单独的变换函数 T_k。有时候，变换函数的构建还会利用到更多信息，例如像素点的位置等。

(b) 图像匹配型：这一方法常常采用一幅已经上好色的图像作为参考。首先，在原灰度图像的各个像素 I_g 和参考图像的各个像素 $\mathbf{I_s}$ 间寻求匹配，使得两像素间（对于某个相似度衡量）其距离 E 最小。然后建立一个 $\mathbf{I_s}$ 和 I_g 间的映射关系，将每个灰度值映射成彩色。

显然，这一做法的关键便在于选择一幅合理的参考图像。选择参考图像的主要方法有三种：手工选择，自动挑选和基于聚类的方法。

(c) 用户选择型：这一方法首先需要用户对灰度图像进行标记。这一标记一般是将图像的某些区域指定为彩色。用户做的标记常常称为 scribble 或 seeds。然后，从这些 scribble 开始进行蔓延，将相似的区域用与之匹配的 scribble 的颜色进行涂抹，并最终达到给整幅图像进行上色的目的。这一方法也是我们所要重点讨论的类型。

对于用户选择型彩色方法，较为典型的例子有 [1] [2] [3] 等。

在 [1] 中，作者假定对于亮度相近的点具有相似的颜色；基于这一设想，可以依据像素值的差别构造出一个代价函数。同时，将 scribble 涉及的像素点作为已知条件，并通过对相近的像素赋予较大的权重，从而优化这个代价函数即可获得彩色化的效果。与此相类似，Sapiro [4] 考虑灰度图像的梯度，并将 scribble 作为边界条件求解；最终把彩色化问题归结为 Possion 方程的求解。

Levin 的方法引起了很多人的注意。后来的 [5]，[6]，[10] 等方法都是针对 Levin 的方法进行的改进。在本文中提出的算法也是以 Levin 的模型作为基础发展而来的。

Yatzi [2] 基于最小路径的快速求解算法提出了一种快速的彩色化方法。该方法首先计算各点到 scribble 上各点的最短路径，然后将这些路径利用特定的函数进行调和，从而得到需要的彩色值。这一方法在不产生严重性能损失的前提下具有较快的运算速度，而且对 scribble 的位置依赖性较低，即对于不同的 scribble 能输出较为接近的结果。

2.2 彩色化问题的数学表述

在下面的叙述中，假定图像在 YUV 颜色空间；$\mathbf{r},\mathbf{s}$ 等代表像素点的坐标 $(x_r,s_r),(x_s,y_s)$。Ω 和 Ω_c 代表像素坐标的集合。

承 2.1中所述，我们关心的一类彩色问题可以表述为:

给定集合 $\Omega=[0,R]\times[0,C]$，$T\subset R^+$；给定三个通道 $Y,U,V:\Omega\to T$。已知灰度通道 $Y(x,y)$ 以及 scribble 区域 $\Omega_c\subset\Omega$，$|\Omega_c|\ll|\Omega|$；已知 $\forall U(x,y):(x,y)\in\Omega_c$ 及 $\forall V(x,y):(x,y)\in\Omega_c$，欲求各 $U(x,y):(x,y)\in\Omega/\Omega_c$ 及 $V(x,y):(x,y)\in\Omega/\Omega_c$。

在 Levin [1] 的模型中，$U(x,y),V(x,y)$ 的求解是通过式 2–7得到。其中的 $N(\mathbf{r})$ 为 $\mathbf{r}$ 的 8-邻域；$w_{\mathbf{rs}}$ 为特定的代价函数，它满足下面的几个特性:

1. 对于 scribble 当中的点，保留其原有的颜色值。即有

$$\forall\mathbf{r}\in\Omega_c, w_{\mathbf{rs}}=\begin{cases}1,\mathbf{r}=\mathbf{s}\\0,\mathbf{r}\neq\mathbf{s}\end{cases}\tag{2–4}$$

2. 若某个点的像素值接近中心点，则它将具有较大的权值；反之，权值较小。用数学公式来表达就是:

   若对于 $\mathbf{s}$ 邻域内两个像素点 $\mathbf{r_1},\mathbf{r_2}\in N(\mathbf{s})$，$\mathbf{r_1}\neq\mathbf{r_2}$，若 $||Y(\mathbf{r_1})-Y(\mathbf{s})||^2<||Y(\mathbf{r_2})-Y(\mathbf{s})||^2$，则 $w_{\mathbf{sr_1}}>w_{\mathbf{sr_2}}$。

Levin 在文中推荐了两种类型的代价函数，分别见式 2–5和 2–6。

$$w_{\mathbf{rs}}\propto e^{-(Y(\mathbf{r})-Y(\mathbf{s}))^2/2\sigma_{\mathbf{r}}^2}\tag{2–5}$$

$$w_{\mathbf{rs}}\propto 1+\frac{1}{\sigma_{\mathbf{r}}^2}(Y(\mathbf{r})-\mu_{\mathbf{r}})(Y(\mathbf{s})-\mu_{\mathbf{r}})\tag{2–6}$$

$$U=arg\min_U\sum_{\mathbf{r}}(U(\mathbf{r})-\sum_{\mathbf{s}\in N(\mathbf{r})}w_{\mathbf{rs}}U(\mathbf{s}))^2\tag{2–7}$$

2.3 小组的前期工作

在先前的工作中，本课题小组已经完成了下面的一些工作:

1. 提出了完整的自动 scribble 生成算法;
2. 提出了采用四元小波变换及 canny 边缘检测产生代价函数的方法，结合最小代价优化，从而构成了整个图像彩色化算法体系;
3. 在 Matlab 平台上有该算法的完整实现;
4. 在 C++（OpenCV）平台上有该算法的试验性实现。

算法 1和 2是课题组在前期工作中提出的 scribble 自动生成算法的流程。其中的分割算法是采用基于图的分割方法 [11]。空间分布熵 [12]， [13]， [14] 的计算由公式 2–8 [15] 确定。式中 $p_{ij} = |R_{ij}|/|R_i|$ 表示颜色 B_i 在环 j 中分布的概率密度。

算法 3是彩色化部分的算法流程。

我们原来算法的主要框架还是建立在 Levin 的算法的基础上。在 Levin 的算法中，首先要求用户输入一些 scribble，即在图像上涂抹一些区域，并使得这些区域的颜色和原图接近；然后按照灰度图像（视为 Y 通道）建立一张权值图，即通过各个像素的 8-邻域来确定该像素的权值；然后利用公式 2–7求解两次优化问题，来分别得到 U，V 两个通道的值。最后，综合 YUV 通道，得到彩色化的结果。在我们的方法中，通过自动 scribble 的生成算法，确定 scribble 的合理位置；用户不需自己勾勒 scribble，只需对现有的 scribble 指定颜色。此外，我们对权值图进行了改进。与原图仅仅通过灰度值来确定权值不同，我们综合了四元 gabor 变换所得的结构信息，和多次 canny 边缘检测得到的边缘信息。

图 2–1是 scribble 搜索算法的流程图。我们采用的检索思想是借助于空间分布熵 [13] 来定位一些重要的支撑点，然后以这些点作为骨架，通过插值、腐蚀与膨胀等方法来描绘出连续的 scribble。算法的关键部分就在于 scribble 支撑点的定位。

我们的想法是：根据实验的结果，算法产生误判的地方常常是一些非边缘而灰度值变化又比较明显的区域。这样的一些区域，其伪边缘往往阻止颜色向

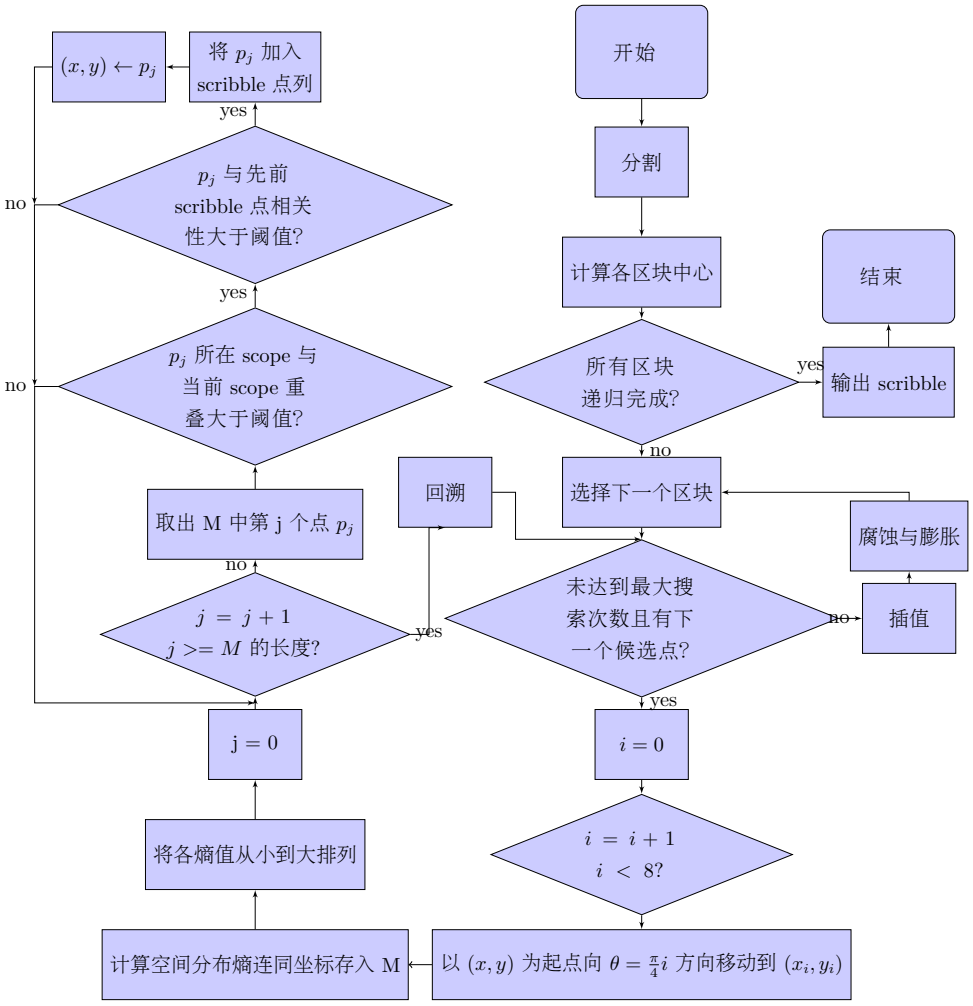


图 2–1: scribble 生成算法流程图

算法 1 scribble 生成算法

输入: 灰度图像: G

输出: scribble 标记图像: S

将 G 分割为 G_s
for all s **in** G_s **do** ▷ 当前搜索的区块
    找出 s 的中心点: (c_x, c_y)
    $r \leftarrow r_0$ ▷ 搜索半径
    $(x, y) \leftarrow (c_x, c_y)$ ▷ 记录当前搜索点的坐标
    $stack \leftarrow empty$ ▷ 堆栈结构，存储当前点的坐标
    $p[\,] \leftarrow empty$ ▷ 记录 scribble 的支撑点
    $mean\_relate \leftarrow 0$ ▷ 平均相关系数
    $Mscope \leftarrow empty$ ▷ 记录被覆盖的 scope
    while 未达到最大搜索次数且具有下一个候选点 **do**
        $M[\,] \leftarrow empty$ ▷ 记录 8 个方向中候选的下一个搜索点及其相关信息
        for $i = 1 \rightarrow 8$ **do**
            $\theta \leftarrow i * \frac{\pi}{4}$
            以 (x, y) 为起点，向 θ 方向移动半径 r 的距离得到点 (x_i, y_i)
            计算 (x_i, y_i) 的最大半径 r_i
            if $(x_i, y_i) \notin s$ **or** $r_i < MIN\_R$ **then** ▷ MIN_R 是所允许的最小半径
                continue
            else
                $entr_i \leftarrow entropy(x_i, y_i)$ ▷ 计算空间分布熵
                $M \leftarrow M + (x_i, y_i, entr_i)$
            end if
        end for
        将 M 按 $entr_i$ 依升序排序

算法 2 scribble 生成算法（延续）

for all $(x_i, y_i) \in M$ **do**
  if $|Mscope \cap scope(x_i, y_i)| > TH * |scope(x_i, y_i)|$ **then**
    continue
  end if
  $H_c \leftarrow hist(x, y)$ ▷ 计算直方图
  $H_n \leftarrow hist(x_i, y_i)$
  $relate \leftarrow compare(H_c, H_n)$ ▷ 相关性衡量
  更新 $mean\_relate$
  if $mean\_relate * 0.6 > relate$ **then**
    continue
  end if
  $Mscope \leftarrow Mscope + scope(x_i, y_i)$ ▷ 加入当前 scope
  $p \leftarrow p + (x_i, y_i)$ ▷ 当前点加入 scribble 队列
  $stack.push(x, y)$
  $(x, y) \leftarrow (x_i, y_i)$
end for
if not $stack.empty()$ **then**
  $(x, y) \leftarrow stack.pop()$
end if
end while
对 p 进行插值，得到曲线 l
对 l 进行腐蚀和膨胀，得到最终的 scribble，并将其标记到 S 上
end for

算法 3 彩色化算法

输入: Scribble 图像: s, 灰度图像: g

输出: 彩色化结果: c

转换 s 为 YUV 图像
$channel[0] \leftarrow g$
$channel[1] \leftarrow s$ 的 U 通道
$channel[2] \leftarrow s$ 的 V 通道
for $i = 1 \rightarrow 2$ **do**
    $W_g \leftarrow gabor(channel[0])$ ▷ Gabor 滤波，得到结构权值
    $W_e \leftarrow strength(channel[0])$ ▷ Canny 边缘检测，得到边缘权值
    $W \leftarrow W_e * W_g$
    $channel[i] \leftarrow arg\min_{\mathbf{x}} ||W\mathbf{x} - g||^2$
end for
$c \leftarrow merge(channel)$
转换 s 为 RGB 图像

其蔓延，然而这些区域又恰恰可能是同态区域。这样便极有可能导致上色的不充分。因此，能够产生较好上色结果的 scribble，应该是那些尽可能多的经过灰度值变化明显区域。用空间分布熵来表述，就是经过的区域空间分布熵尽可能小。由于全图搜索过于复杂，为了简化搜索过程，我们首先对整幅图像进行分割，然后对分割所得的各个区块分别进行搜索。

在这样的算法框架下，一个很自然的问题就是各个区块搜索起始点的选取。为了使得搜索得到的 scribble 尽可能的遍布各个区块，起始点应当尽量靠近各个区块的几何中心。然而简单的以几何中心为起始点是存在问题的；这在 3.1部分有较为详细的讨论。

另外一个问题则是搜索方法问题。即：采用什么样的顺序进行搜索，和当前搜索的点，需要满足什么样的条件才能作为 scribble 的支撑点。对于任何一个点，对应有一个圆形的邻域，称作 *scope*。空间分布熵的计算，就在这个 scope 内进行。根据前文所述，为使 scribble 经过的范围有尽量小的空间分布熵，对给定的点 p，我们以一定的半径（与点 p 到区域边界的距离正相关）向该点八个方向搜寻相邻的点，并采用熵值从小到大的顺序。对于当前点 p 的某

个邻接点 p_i，仅当其满足以下两个条件时，将其作为 scribble 的支撑点：

1. 同态性判定；计算 p 和 p_i 在各自 scope 的直方图，然后比较两个直方图得相关系数 r。若 r 高于先前所得各支撑点的相关系数的平均值乘上某个系数，则接受 pi。

2. 最大重叠判定：若 p_i 的 scope 与先前所有点的 scope 重叠区域面积大于 p_i 的 scope 面积的某个百分比，则舍弃 p_i。这样可以防止在某个区域内的 scope 支撑点过于密集。

图 2–2是彩色化算法的流程图。我们的权值图依然是指数形式的；这借鉴了 Levin 的表达式（式 2–5）。所不同的是，我们的权值是通过 gabor 滤波得到的特征权重和边缘检测得到的边缘权重（称作 *strength map*）综合（相乘）而得。这样，我们既考虑了原图的结构信息，又考虑了边缘信息，相比于单纯的灰度值，在大多数情况下能够取得更好的处理效果 [16]。

$$E_i = -\sum_{j=1}^{M} p_{ij} \log_2(p_{ij}) \tag{2–8}$$

虽然该算法已经基本成形，但是现有的 matlab 实现并不是十分成熟；一些细节地方仍需要进一步完善。此外现有的代码效率也比较低下，运算时间较长。

此外，算法的 C++ 实现也仍旧处于起步阶段；现有代码仅完成了 MFC 的 UI 框架。仍旧需要完成的工作有：

1. 实现 scribble 自动生成算法；

2. 实现 gabor 滤波器，以及最小代价函数的计算；

3. 实现优化最小代价函数的求解算法。

由于 Matlab 和 C++ 差异较大，将 Matlab 代码移植到 C++ 需要较多的工作，两者之间的差别也会产生许多的细节问题。在后面的部分中，我们将针对移植过程中遇到的问题及对细节的处理进行详尽的讨论。

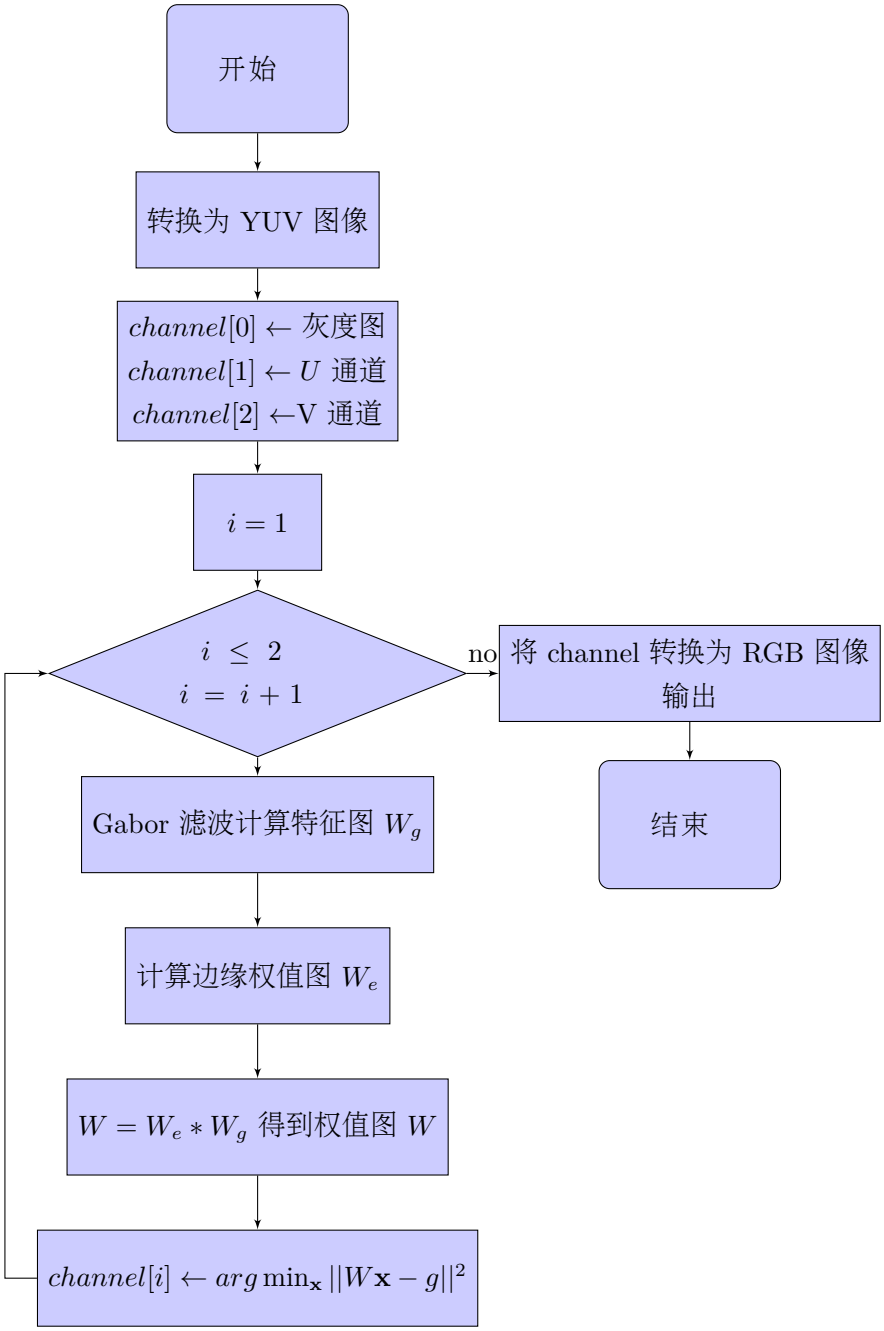


图 2–2: 彩色化算法流程图

第三章　基于四元相位的黑白图像彩色化算法及其优化

3.1　scribble 生成算法

现有 scribble 的生成算法存在下面的几个问题：

1. 在设计代码结构时，作者似乎有意避免采用递归算法。根据往届的论文，作者提到这么做的意图是担心递归带来的效率损失。但是事实上，scribble 生成算法非常适合采用深度优先搜索的递归结构；由于刻意避免递归，反而导致不得不人为保存后来用于回溯的数据。也即代替编译器进行了堆栈的管理。这样的做法不仅使代码书写和维护的难度大幅增加，而且从最后的效果来看，反而增加了运算量和极大的增加了内存的开销。

   实际上，由于搜索步长较大，因此分割后得到的每个区块在搜索 scribble 的支撑点时最多也就经过数十次的搜索。这样的一点函数调用开销对于整个运算过程而言完全是可以忽略的。基于这些考虑，如果将原算法用递归的方式实现，不仅能优化代码结构，而且可以改善运算性能。

2. 原代码的运算效率较低。一些地方可以有较多的改进和优化。例如，在记录用于回溯的数据时，存储了大量的冗余信息。事实上，原来的方法存储了当前搜索点和 8-领域内所有后继点的熵值、半径和 scope 信息。其中熵值仅需在计算相关性时用到，而计算相关性只要知道之前的相关性和目前为止经历的点数即可。

   相比较而言，浪费较严重的是 scope 的存储。如第 2.3节所述，后面真正需要用到 scope 的地方是在计算当前点所在 scope 与先前的重叠，以此来判断重叠区域不至于过大。由于各个区块相对独立，每个点的 scope 都被严格限制在区块内，因此不同区块间的 scope 不会发生重叠。所以，记录当前经历的 scope 实际上只需要一张图就已足够，完全没有必要对每个点都存储一张 scope 图。

当然，由于 Matlab 采用了 “copy on write” 技术，真正运算时空间的浪费可能会小于我们预先的估计。但是可以断定，先前的方法对内存的浪费是严重的，有较大的改进空间。

除空间浪费外，原程序在时间效率上也有待加强。比如大量的循环嵌套以及冗余计算等，都降低了原有实现的效率。

3. 在代码实现的有些细节上，发现有与原文不相符的地方。例如，原文中提到计算各区块的起始点时，采用的是**几何中心**，而在实际实现时，却是根据灰度值直方图进行计算得到。此外，原有代码还有一些逻辑性的错误。比如，在搜索到某点与先前的点相关度低于阈值时，应该是放弃该点继续往下搜索，而在代码中却停止了整个搜索流程。等等。

在我们的 C++ 实现中，对现有的问题进行了针对性的改善。如 3中提到的，代码中找寻几何中心的方法有不符原文之处。为使得 scribble 在空间上尽量伸展，起始点应该尽量靠近当前区块的中心。然而，简单的选择几何中心也是有问题的；因为区块的形状任意，在有些情形下（凹集）几何中心可能不一定落在区块内部。图 3–1是两个典型的例子。

因此，我们采用对 x 方向和 y 方向两次取中位数的方法，既保证所得点尽量靠近中心，又不至于落在区块外面。记当前区块的所有点构成点集 $\mathbf{s}:\{(x,y)\}$，并记点 $(x_i,y_i)\in\mathbf{s}, i=1,2,\cdots,|\mathbf{s}|$，则具体做法可表述如下：

1. 对 $\forall(x_i,y_i)\in\mathbf{s}$，计算 x 方向的中位数，记为 x_m，即

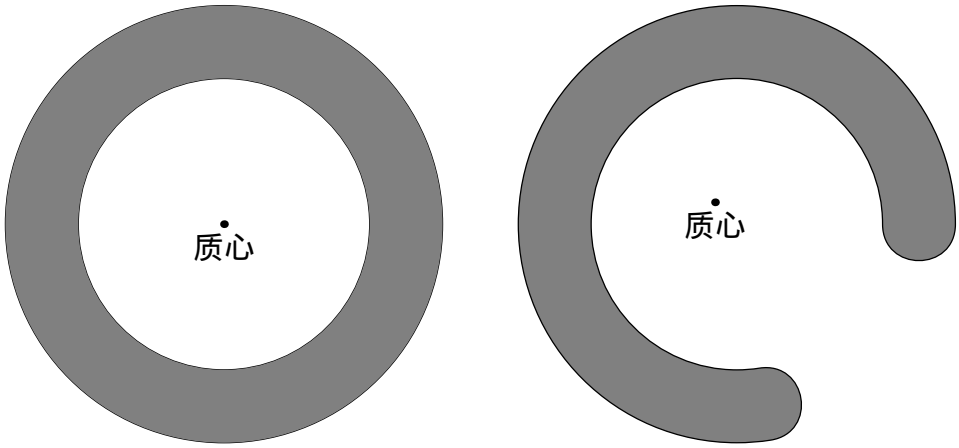
$$x_m = median\{x_1,x_2,\cdots,x_{|\mathbf{s}|}\}$$

对于连续的区域，这一操作得到的 x_m 将位于其 “中轴” 上。

2. 取出所有横坐标为 x_m 的点，对 y 方向再计算一次中位数得到 y_m，即 $y_m = median\{y_i|(x_m,y_i)\in\mathbf{s}\}$。直观上，我们可以理解为取该区块的中心轴线的中心。注意这条 “轴线” 有可能是断开的。

3. (x_m,y_m) 即可作为所求起始点。

针对问题 1，我们决定在 C++ 实现中采用递归的方式。在我们的实现中，需要维护的数据结构有：



(a) 环形　　(b) 凹形

图 3–1: 两种典型的质心不在内部的区域

1. 当前的平均相关系数 R_{mean};
2. scribble 列表 p，代表合格的 scribble 点；可以采用堆栈的结构，在我们的代码中为简便直接采用了向量（vector）;
3. 分割后得到的标记图 G_s;
4. 记录当前搜索遍及到的 scope 区域的二值图像 $Mscope$;
5. 记录当前点是否已经搜索过的二值图像 T，用于避免重复搜索。

利用上述结构，整理可得基于深度优先搜索（递归）的 scribble 生成算法，如算法 4所示。其中的递归主体函数 $track$ 的定义如算法 5所示。

采用递归实现能大幅度简化算法的流程；对于递归子函数，只需要依次检索其八个方向的后继节点，并递归搜索符合条件的结点即可。原实现中复杂的逻辑判断以及冗杂的数据结构都得到了简化，有助于在实现时减少逻辑错误，也有利于代码的维护。

在实际运行时，我们发现原有代码还存在一些尚未修复的 bug。比如，理论上每个区块的 scribble 搜索都是限制在当前区块内进行的，因此生成的 scribble 范围应当在当前区块内部。然而，实际运行得到的 scribble 却常常跨越了区块。

算法 4 基于递归的 scribble 生成算法

输入: 灰度图像: G

输出: scribble 标记图像: S

1: 将 G 分割为 G_s
2: **for all** s **in** G_s **do** ▷ 当前搜索的区块
3: 找出 s 的中心点: (c_x, c_y)
4: $r \leftarrow r_0$ ▷ 搜索半径
5: $p[\,] \leftarrow empty$ ▷ 记录 scribble 的支撑点
6: $R_{mean} \leftarrow 0$ ▷ 平均相关系数
7: $Mscope \leftarrow empty$ ▷ 记录被覆盖的 scope
8: $depth \leftarrow 0$
9: $track(p, c_x, c_y, r_0, Mscope, R_{mean}, T, G_s, depth)$
10: 对 p 进行插值，得到曲线 l
11: 对 l 进行腐蚀和膨胀，得到最终的 scribble，并将其标记到 S 上
12: **end for**

图 3–2展示了两个典型的例子。为了方便观测，将 scribble 标记图像和分割后所得的图像进行了叠加。

此外，原有算法采用的插值方法也比较粗糙；仅仅是分段的多项式插值。由图 3–2可见，这样插值所得的曲线较为不光滑，其中有些甚至呈现折线型。因此，为了能得到更为平滑的 scribble，我们改为采用 B 样条进行插值。

B 样条曲线通过式 3–3来进行定义，是贝塞尔曲线的一种推广。其中 $\mathbf{p_i}$ 称作 B 样条的**控制点**；曲线 $\mathbf{c}$ 的维度即等于控制点的维度。例如，一维控制点对应函数插值，二维控制点对应平面曲线插值。很明显，在我们的算法中，应该选用二维控制点。

从式 3–3中可以看出，B 样条曲线 $\mathbf{c}$ 是样条 $B_{i,k,\mathbf{t}}$ 的线性组合。而每个样条又可由节点向量 $\mathbf{t}$ 和**阶数**k 唯一决定。阶数 k 等于 B 样条的度数加上 1。例如，若 B 样条为二次曲线，则 $k = 3$。

参数 t 的范围在区间 $[t_k, t_{n+1}]$ 之间。若记 d 为控制点 $\mathbf{p_i}$ 的维度，则曲线 $\mathbf{c}$ 可以视作从 $[t_k, t_{n+1}]$ 到 R^d 的映射，即 $\mathbf{c} : [t_k, t_{n+1}] \to R^d$。

算法 5 递归主体函数定义

function $track(p, x, y, r, Mscope, R_{mean}, T, G_s, depth)$
  if $T(x, y)$ *is true* **or** $depth \geq MAX\_DEPTH$ **then** ▷ 达到递归次数上限
return
  end if
  $p.push(x, y)$ ▷ 加入 scribble 支撑点列表
  $Mscope \leftarrow Mscope + scope(x_i, y_i)$ ▷ 加入当前 scope
  for $i = 1 \rightarrow 8, \theta = i * \frac{\pi}{4}$ **do**
    以 (x, y) 为起点，向 θ 方向移动半径 r 的距离得到点 (x_i, y_i)
    计算 (x_i, y_i) 的最大半径 r_i
    if $(x_i, y_i) \notin$ **s or** $r_i < MIN\_R$ **then**
      continue
    else
      $entr_i \leftarrow entropy(x_i, y_i)$
      $M \leftarrow M + (x_i, y_i, entr_i)$ ▷ 计算空间分布熵
    end if
  end for
  将 M 按 $entr_i$ 依升序排序
  for all $(x_i, y_i) \in M$ **do**
    if $|Mscope \cap scope(x_i, y_i)| > TH * |scope(x_i, y_i)|$ **then**
      continue
    end if
    $H_c \leftarrow hist(x, y), H_n \leftarrow hist(x_i, y_i)$
    $relate \leftarrow compare(H_c, H_n)$ ▷ 相关性衡量
    更新 R_{mean}
    if $R_{mean} * 0.6 > relate$ **then**
      continue
    end if
    $track(p, x_i, y_i, r_i, Mscope, R_{mean}, T, G_s, depth + 1)$
  end for
end function

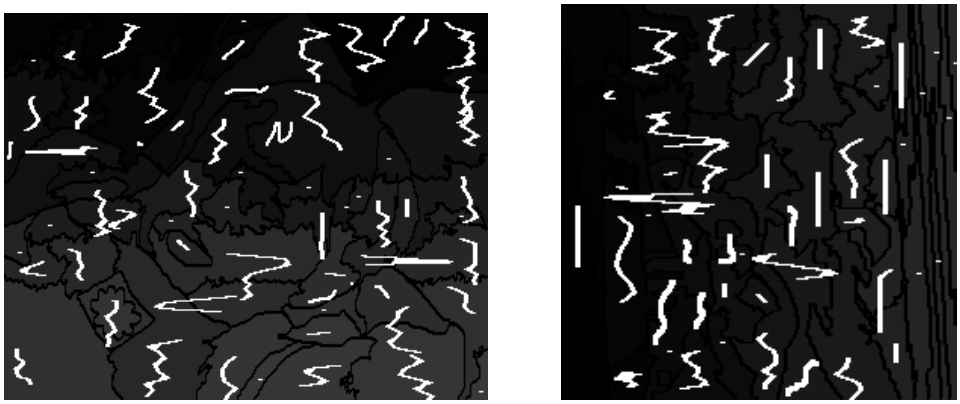


图 3–2: scribble 溢出当前块的例子，为方便观测将 scribble 图和分割后的图像进行了叠加

B 样条是通过递归定义的；即高阶的 B 样条可以由低阶的 B 样条通过相乘相加的方式得到。对于 1 阶的 B 样条，其形状便是一段水平线段；如式 3–1所示。由 $k-1$ 阶 B 样条得到 k 阶 B 样条的公式见式 3–2。

$$B_{i,1}(t) = \begin{cases} 1 & t \in [t_i, t_{i+1}) \\ 0 & o.w. \end{cases} \tag{3–1}$$

$$B_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - ti + 1} B_{i-1,k-1}(t) \tag{3–2}$$

B 样条满足一些非常重要的性质；其中包括 [17]:

1. 每个 B 样条均不为负；
2. 各个样条之和恰为 1，即 $\sum_{i=1}^{n} B_{i,k,\mathbf{t}}(t) = 1$；
3. 凸性：B 样条曲线一定落在其各个控制点组成的凸包之内；
4. 由于 $B_{i,k,\mathbf{t}}$ 局限于区间 $[t_i, t_{i+k}]$ 之内，因此改变 B 样条的某个控制点仅会改变曲线的局部特性。
5. 在 B 样条曲线的端点处，起始点的值等于第一个控制点的值，终结点的值等于最后一个控制点的值；也即:

$$\mathbf{c}(t_k) = \mathbf{p}_1, \mathbf{c}(t_{n+1}) = \mathbf{p}_n$$

$$\mathbf{c}(t) = \sum_{i=1}^{n} \mathbf{p_i} B_{i,k,\mathbf{t}}(t) \tag{3–3}$$

由于 B 样条具备上述的一些优良性质，使得它在数值分析、函数逼近等领域具备广泛的应用。

3.2 最优化算法

我们在构建优化模型时，依旧借鉴了 Levin[1] 的思路。首先是对代价函数的表示问题。我们采用式2–5所示的形式，其中的权值表示为：

$$W_{\mathbf{pq}} = W^e_{\mathbf{pq}} W^g_{\mathbf{pq}} \tag{3–4}$$

$$W^e_{\mathbf{pq}} = e^{(-||E(\mathbf{p})-E(\mathbf{q})||^2)} \tag{3–5}$$

$$W^g_{\mathbf{pq}} = e^{(-H(\mathbf{p},\mathbf{q}))} \tag{3–6}$$

其中 $W^e_{\mathbf{pq}}$，$W^g_{\mathbf{pq}}$ 分别是边缘的权重和结构的权重，分别由 strength map 和 gabor 变换产生；$E(\mathbf{p})$ 表示点 $\mathbf{p}$ 的 strength；$H(\mathbf{p},\mathbf{q})$ 是各个尺度分量和角度分量的 *Gabor* 滤波所得结果的叠加，其展开形式见公式 3–10 [18]。其中 $|\cdot|_\beta$ 为取模运算符。

对于 Gabor 和 strength map 的实现，我们基本上参照现有的 Matlab 代码；根据四元 Gabor 滤波器的表达式（式 3–7）[19]，展开可得四项; 从而可以分别将四个滤波器（不同方向和尺度）与原图像滤波，并最后综合得到最终结果。

$$G^q_{\sigma\alpha}(\mathbf{x},\mathbf{u}) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} e^{-i\frac{2\pi x'}{\lambda}} e^{-j\frac{2\pi y'}{\lambda}} \tag{3–7}$$

$$H(\mathbf{p},\mathbf{q}) = \sum_{\sigma=0}^{7}\sum_{\alpha=0}^{3} \frac{1}{4}|\rho(\mathbf{p})\psi_{\alpha\sigma}(\mathbf{p}) - \rho(\mathbf{q})\psi_{\alpha\sigma}(\mathbf{q})|_{2\pi} \tag{3–8}$$

$$+\frac{1}{2}|\rho(\mathbf{p})\theta_{\alpha\sigma}(\mathbf{p}) - \rho(\mathbf{q})\theta_{\sigma\alpha}(\mathbf{q})|_{\pi} \tag{3–9}$$

$$+|\rho(\mathbf{p})\varphi_{\sigma\alpha}(\mathbf{p}) - \rho(\mathbf{q})\varphi_{\sigma\alpha}(\mathbf{q})|_{\frac{\pi}{2}} \tag{3–10}$$

设原灰度图像 G 的尺寸为 $m \times n$；将原图按行进行拼接，则得到一个长为 mn 的向量 $\mathbf{b}$。在原图中坐标为 (i,j) 的点（i,j 从零开始标起，即 $i = 0,1,\cdots,m-1, j = 0,1,\cdots,n-1$），在 $\mathbf{b}$ 中将位于下标 $i*n+j$ 处。现定义一个 $mn \times mn$ 的矩阵 M，令该矩阵的第 i 行存储下标为 i 的像素对原图中各个像素 $j(j = 0,1,\cdots,mn)$ 的权重；即：若设点 $\mathbf{p},\mathbf{q}$ 在向量 $\mathbf{b}$ 中的下标分别为 x,y，则有 $W_{\mathbf{pq}} = M(x,y)$。

不难看出，$M(x,y)$ 为一个*稀疏矩阵*；且其任意一行最多有 9 个元素非零。回想我们需要求解的问题（公式 2–7），利用这一结构即可将其改写成公式 3–11的形式。其中 $\mathbf{b}$ 是由 U 或 V 通道逐行拼接得到的长度为 mn 的向量；计算得到 $\mathbf{x}$ 之后，将其还原成 $m \times n$ 的矩阵，即得到 U 通道的彩色化结果。

$$\mathbf{x} = arg\min_{\mathbf{x}} ||\mathbf{b} - M\mathbf{x}||^2 \tag{3–11}$$

这一稀疏矩阵的构造方法，在 Levin 的代码中也有所体现；然而对于公式 3–11的求解，在 Levin 的代码中是直接运用 Matlab 来求逆。而我们的目标是将算法移植到 C++ 平台上，所以必须自己寻找这一稀疏线性系统的 solver。

起初，考虑到 M 为稀疏矩阵，我们找到了著名的 *suitesparse* 库（http://www.cise.ufl.edu/research/sparse/SuiteSparse）。这个库性能优良，曾经被 Google，nVidia 以及 Mathworks 用来进行相关的计算。但是在实现了基于 suitesparse 的优化算法之后（包括 *umfpack* 和 *SuitesparseQR*，分别对应稀疏矩阵的 LU 分解和 QR 分解），经比较发现运算时间过长，且远远多于 Levin 提供的参考代码。

后来经仔细分析，我们发现，其实这一方法还可进一步简化。对于每个像素，只有其 8-领域内的点权值有可能不为零；也就是说，对于下标为 i 的像素，仅有下标为 $i-n-1$，$i-n$，$i-n+1$，$i-1$，$i+1$，以及 $i+n-1$，$i+n$，$i+n+1$ 这些点（如果它们存在的话）可能对其权值构成贡献，或者说，M 的第 i 行只有下标为上述数字的列可能非零。据此，我们可以断言 M 是一个*带状矩阵*，即在对角线的某条带状区域外所有元素均为零的一类稀疏矩阵。

需要说明的是，M *并非*对称矩阵。这是因为，在需要填充彩色的 U,V 通道中，有少部分的像素位于 scribble 上。依据公式 2–4，为保证 scribble 上的点保留其原有色彩，若下标为 i 的点 $\mathbf{p}$ 位于 scribble 上，则 M 的第 i 行除了 (i,i) 外均为零。而另一方面，若下标为 j 的某个点 $\mathbf{q} \in N(\mathbf{p})$，且 $\mathbf{q}$ 不在 scribble

上，那么很可能有 $W_{\mathbf{qp}} \neq 0$。这样便有 $M(j,i) \neq 0$。据前所述 $M(i,j) = 0$，从而 $M(i,j) \neq M(j,i)$。

因此，我们需要求解的是一个非对称而带宽受限的稀疏矩阵。对于这一问题，已经有大量的研究工作和成熟的快速计算方法。这使得我们算法需要的优化能够得到较为高效的解决，也保证了我们的算法整体上的效率。

3.3 本章小节

在这一章节中，我们针对在前期工作中实现的彩色化算法的 Matlab 代码作了详尽深入的讨论。通过分析，我们指出了原代码存在的一些不足之处：运算效率低，存在一些 bug 等。更进一步地，我们提出了改进这些问题的一些方法，并在我们的 C++ 实现中应用了这些方法。

由于时间所限，我们有些关于算法上的想法或改进方案暂时还未能得到验证。例如，对于有些非凸的区域，其搜索起始点往往离边界较为接近；而对于现有的搜索方案（直接舍弃半径较小的 scope），在此种情形下很容易造成算法的提前终止而导致遍历不充分。图 3–3即是一个典型的例子。图中虽然当前可以搜索的区域很大，但是由于起始点落在了半径很小的颈部，因而刚开始搜索就因为半径过小（低于阈值）而提前中止，从而导致整块区域的 scribble 只有一个点。这显然不是我们希望得到的结果，因此需要改进。目前考虑的改进方法是在搜索下一个 scribble 点时，考虑当前区块的凸包（convex hull）。此外，考虑改进中止的判断条件，即有条件放宽以防止过早中止。

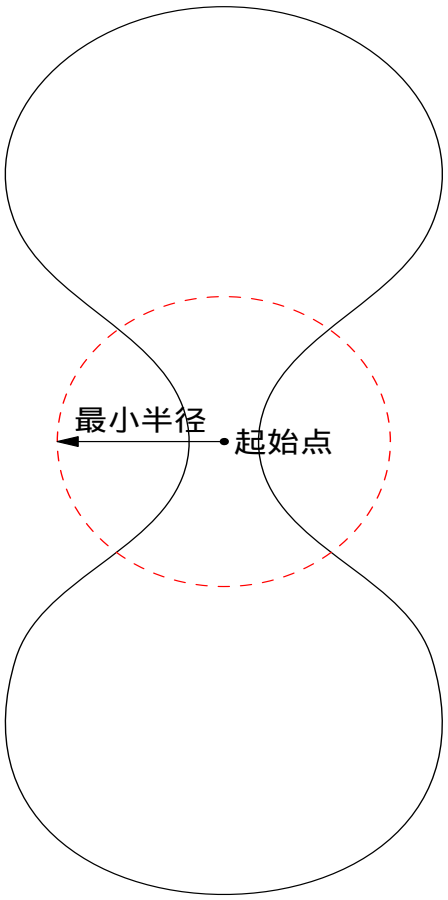


图 3–3: 某些不规则区域导致的遍历不充分

此外，原算法存在的一些不足依然未能得到解决。比如，我们目前应用于算法中的基于图的分割算法，其分割的结果在一定程度上依赖其参数的设定；而且，对于不同的图像，最优的参数是各不相同的。然而，由于我们的 scribble 自动生成算法的性能很大程度上依赖于分割的结果，因此也会不可避免的受其影响，造成表现上的不稳定。另外，为保证 scribble 能提供足够的信息，避免遗漏细节，目前往往对图像进行过分割；然而过分割会产生很多小区块，导致产生的 scribble 非常细碎。对于这些问题，目前想到的解决方案有两种：一是寻找更好的分割算法，一是修改目前的 scribble 生成算法，使之不依赖于图像分割。

在下面的一章中，我们将从实现细节的角度来深入讨论算法的底层实现及其注意事项；我们还将对算法的性能进行评估，并分析仍然存在的不足之处，以及可能的一些解决方案。

第四章　软件界面和功能模块设计

按照现有的算法流程，我们可以将算法的实现划分为下面的三个主要模块：

1. scribble 生成模块：用于实现 scribble 自动生成算法；
2. 最优化模块：计算 weight map，以及实现优化求解算法；
3. 界面模块：和用户交互，显示处理结果等。

下面分别具体叙述这三个模块的功能及其实现。

4.1　scribble 生成模块

这一模块接收原灰度图像，通过 3.1中所述算法产生带有 scribble 标记的二值图像（0 或 255），然后将其传递给主界面，以备进行后续运算。该模块和主界面的接口函数如下：

```
1 Mat scribble_gen(Mat_<unsigned char> img_gray);
```

在分割算法的后期，我们将分割的信息记录在一个指标图 S 中，S 中每个像素 $\mathbf{p}$ 的值代表其所在区块的索引。设总共有 n 个区块，则 $G(\mathbf{p}) \in \{0, 1, \cdots, n-1\}$。我们用两个二维数组 $buffer\_x[][], buffer\_y[][]$ 记录各点坐标，例如 $buffer\_x[i]$ 即储存第 i 块的所有 x 坐标。这样一来，仅需对 S 扫描一次，即可以获得 $buffer\_x, buffer\_y$；而且，由于采用顺序式扫描，$buffer\_x$ 是按照递增顺序生成的，这样在计算中位数时，又可以省去一次排序。

在实现 B 样条插值算法时，我们采用的是 *SINTEF* 的插值库 *SISL*（www.sintef.no）。这个库早在 1980 年开始开发，最早是为了满足产业界对于 NURBS 算法库的需求。到 1988 年完成了 NURBS 的插值算法库，后来开始不断加入新功能。该项目直到现在依旧在维护开发中，经过不断的修改已经非常成熟（可能是当前最为成熟的开源插值库了），代码的稳定性和运算效率均能

经受考验。SISL 除了提供 B 样条插值和 NURBS 插值算法外，还带有了曲线操作的一些辅助功能。目前其最新版本是 4.2，也便是我们所采用的版本。

作为一个纯 C 语言实现的运算库，其 API 较为难用；现有的范例和文档也并不十分丰富。这使得在实际运用它时，编写代码上有一定的难度。较为重要的函数有四个：$s1356, s1227, s1920, s1850$。它们各自的功能见表 4–1。

由于 SISL 插值后得到的是连续曲线，只能对参数 t 求值而不支持直接将曲线的整点坐标输出，为了获得插值后的曲线，我们只能采用间接的方法。注意到将连续的曲线离散化实际上相当于一个采样的过程；我们可以人为构造一个二维的 “采样网格”，和目标曲线进行相交，即可获得我们需要的取值。具体做法参见算法 6。其中所采用的各函数名与表 4–1相同。

算法 6 scribble 插值算法

输入: scribble 的支撑点 p_0

输出: 插值得到的 scribble 曲线 p

$p \leftarrow empty$

$c \leftarrow s1356(p_0)$ ▷ 获得插值曲线

利用 $s1920$ 计算 x 方向的极小值点 x_{min}

利用 $s1920$ 计算 x 方向的极大值点 x_{max}

利用 $s1920$ 计算 y 方向的极大值点 y_{max}

利用 $s1920$ 计算 y 方向的极小值点 y_{min}

for $i = x_{min} \rightarrow x_{max}$ **do** ▷ 在 x 方向构造虚拟网格对曲线采样

    利用 $s1850$ 计算 $c \cap \{(x, y)|x = i\}$ 得点集 p_x

    $p \leftarrow p \cup p_x$

end for

for $j = y_{min} \rightarrow y_{max}$ **do** ▷ 在 y 方向构造虚拟网格对曲线采样

    利用 $s1850$ 计算 $c \cap \{(x, y)|y = j\}$ 得点集 p_y

    $p \leftarrow p \cup p_y$

end for

表 4–1: SISL 相关函数介绍

函数名	功能
s1356	对给定的一系列点计算曲线插值
s1227	计算曲线在某点的值或者一阶导数
s1850	找出给定曲线和直线的交点
s1920	对给定曲线找寻极值点

表 4–2: Octave 常用稀疏线性系统求解函数

函数名	求解矩阵类型
bsolve	带状矩阵
fsolve	通用稀疏矩阵；LU 或 cholesky 分解
dsolve	对角矩阵
utsolve	上三角矩阵
ltsolve	下三角矩阵

4.2 最优化模块

这一模块实现了 3.2中描述的最优化算法，通过综合 scribble 标记和原灰度图像的信息，产生彩色化的结果。该模块和主界面的接口函数原型如下：

```
1 Mat getVolColor(Mat_<unsigned char> scribble_map, Mat_<Vec3b> data,
        Mat_<unsigned char> gray_channel)
```

我们求解带状矩阵采用的库是 *liboctave*。它是开源数学软件 GNU Octave 的一部分，是其各种数学运算的核心库。它自带的 API 非常简洁明了，使用也较为简便。缺点是文档不够丰富，很多自带的功能都没有相应的文档说明，有时候为了知晓某些函数的用法甚至需要阅读源代码。

Octave 的 SparseMatrix 类自带有求解各种稀疏线性系统的方法。其中比较典型的几个方法及其对应的问题类型记录在表 4–2中。这些方法都是用来求解形如 $A\mathbf{x} = \mathbf{b}$，其中 A 为稀疏矩阵之类的线性系统。

在使用 liboctave 的稀疏矩阵的 API 时，比较麻烦的问题是如何创建稀疏矩阵。目前稀疏矩阵的存储方法主要有两种 [20]：压缩列（或行）方法和非压

缩方法。在 liboctave 中采用的是压缩列方法。具体的，设稀疏矩阵 A 中（至多）有 n_z 个非零元素，其尺寸为 $m \times n$，则存储稀疏矩阵 A 需要三个数组：c_{idx}, r_{idx} 和 $data$。其中 r_{idx} 和 $data$ 的长度均为 n_z，而 c_{idx} 的长度为 $n+1$。

$data$ 用来存储 A 中的实际数据，而 r_{idx} 和 c_{idx} 则是用来索引数据。具体的，$c_{idx}[0] = 0$，$c_{idx}[i](i = 1, 2, \cdots, n)$ 用来表示第 i 列的数据在 $data$ 中的最大下标（但不包含）。c_{idx} 和 $data$ 配套，$c_{idx}[j]$ 存储 $data[j]$ 的行标。这样，如果我们要访问矩阵 A 中位于 (x, y) 处的数据，首先要读取 $c_{idx}[y]$ 和 $c_{idx}[y-1]$ 的数值；据定义，A 在第 y 列的数据便是 $data[c_{idx}[y-1]]$ 到 $data[c_{idx}[y]]$ 的值。然后，下标从 $c_{idx}[y-1]$ 开始，依次访问 $r_{idx}[c_{idx}[y-1]+1], r_{idx}[c_{idx}[y-1]+2], \cdots, r_{idx}[c_{idx}[y]-1]$，查找等于 x 的某个 $r_{idx}[k]$ 的值。若找到，则 (x, y) 的值即是 $data[k]$；否则 $A(x, y) = 0$。注意：$r_{idx}[c_{idx}[y]]$ 并不包含在内。

按照上述定义，我们在创建稀疏矩阵（权值图）时，需对原图逐个像素扫描，然后每个像素的权值占据一行。对每个像素，扫描其领域内的各点，并将他们与中心点计算所得的权值填入权值图对应的位置。实际上，出于方便起见，我们在构建时采用逐行填充的方法，只是在构建完成后再转置回去。

一旦权值图创建完成，我们便可以直接利用 liboctave 自带的各种 solve 方法，来解决我们的优化问题。当然由于 Octave 采取列主方式存储数据，所以构建以后不可避免的要将原矩阵转置。由于 OpenCV 采用行主方式存储，因此将 OpenCV 的 Mat 类型转换为逐行相连的向量是非常容易的。至此，整个优化过程的细节问题都得到了解决。

4.3 界面设计

界面的绘制借助的是开源界面库 *Qt*。Qt 最早于 1991 年由 Trolltech 公司开发，基于 C++ 编程语言。Qt 是一个跨平台的非常优秀的界面库，著名的 KDE 项目就是以 Qt 为基础开发的。2008 年 Nokia 收购 Qt 并将其开源。现在 Qt 则处于 Digia 公司的维护之下。

选用 Qt 主要是基于 Qt 的以下几项优点：

1. 开源免费，质量也很高；

2. 支持目前流行的各种平台（windows，linux 和 Mac OS X 等）；

3. 丰富的文档和样例代码，学习起来容易，上手简单。

4. Qt 和 OpenCV 的兼容性非常好，两者之间传递图像数据不仅方便，而且开销非常小。关于两者之间的数据交互后面也会提到。

我们的界面实现改编自 Qt 的官方样例 *ImageViewer*；该程序实现了一个简单的图片浏览器，能够打开图片，打印图片，放大、缩小图片大小等等。虽然功能比较简单，但已经能够基本满足我们的需求。实际上，我们需要做的事情主要有以下两项：

1. 增加选色功能：需要给 ImageViewer 类添加鼠标点击事件；另外由于 Qt 自带有 QColorDialog 类，所以选色对话框的实现并不困难；在用户选定颜色后，便可以从当前鼠标点击坐标出发，进行深度优先搜索涂色，直到颜色涂满整个 scribble 区域。

2. 实现 Qt 和 OpenCV 间的数据传递：Qt 在打开图像后会将其存储到 QImage 类型；而 OpenCV 则是 cv::Mat 类型（比较老的是 IplImage* 结构）。为了能在 Qt 中使用 OpenCV 的算法，势必要实现两种数据类型的转换。实际上，Qt 和 OpenCV 存储图像的方式极为接近：都是以行主方式存储数据，而且都是通过 *step* 成员记录每行有多少个字节。因此，从 Mat 转换到 QImage，可以直接调用 QImage 的构造函数；下面是一个例子：

```
1  QImage image(data.data,data.cols,data.rows,data.step,QImage::
     Format_RGB888);
2
```

   对于灰度图像，只需将 “Format_RGB888” 替换成 “Format_Indexed8” 即可。实际上，QImage 主要是用于显示图像；按照上面的构造方式，QImage 和 Mat 实际上共享了一块空间。这样，我们可以对同一块空间用 OpenCV 进行各种运算，得到结果后再借助 QImage 将图像显示出来。

至此，各个模块均已实现。通过 GUI，用户可以首先读入灰度图像，然后单击菜单上的 “show scribble” 按键生成 scribble。此后再点击 “Start Colorization” 会弹出对话框让用户选择色彩，选色完毕后便会开始彩色化进程，并最终显示彩色化的结果。

4.4 实验结果与相关讨论

图 4–2c是用 GUI 打开图像的截图；图 4–2a是选色时候的界面截图。图 4–2b是产生的 scribble。

图 4–4c和图 4–4d分别是上色良好和上色不充分的例子；作为对比原图（图 4–4a和图 4–4b）也被放在了一起。在测试时，我们将 scribble 的颜色用原彩色图像的颜色填充。

通过对比图 4–4c和图 4–4d不难发现，尽管两图在处理时采用了较为相似的分割参数，但是处理效果差别却很大。图 4–4c虽然颜色较之原图稍有淡化，但上色比较均匀，没有明显的颜色泄漏等视觉上比较明显的瑕疵。事实上，多数情况下彩色化的目的并不是在于重建原彩色图像，而是能够产生视觉上比较舒适的彩图；因此图 4–4c的处理结果是可以接受的。

然而，图 4–4d的处理结果则不令人满意；图像中很多地方颜色都未涂上，上色非常不均匀；通过研究该图的 scribble 图与原图叠加的图像（图 4–1），可以发现在涂色不充分的地方，例如山峦等处，常常是结构比较丰富的地方。在这些区域，分割算法常常难以准确判断。从 scribble 的形状来看，在有些局部区域，分割算法未能判断山峦的轮廓，从而将其归到了一块。这样在一些 scribble 未能涉及的一些局部区域，就没有足够的色彩信息，从而导致有些区域上色不充分。

此外，在我们的实现中有效避免原算法有些 scribble 会跨出当前区块的问题。图 4–3a是原实现中分割图像和 scribble 标记的叠加；图 4–3b是新实现中分割图像和 scribble 的叠加。从中可以看到，图 4–3a在某些细小的区域，有的 scribble 越出了当前区块，有的甚至越过了好几个区块；而在图 4–3b中，却不存在这种现象。不过，我们也可以看到，由于对 scribble 范围的严格限定（搜索区域局限于当前分割区块），因此在一定程度上造成了 scribble 的碎片化（许多 scribble 非常短，有的只有一两个点）。在现行算法下，如果采用过分割，这种情形的发生几乎是无法避免的。当然也有类似于图 3–3中的情形。这些问题，都是有待解决的。

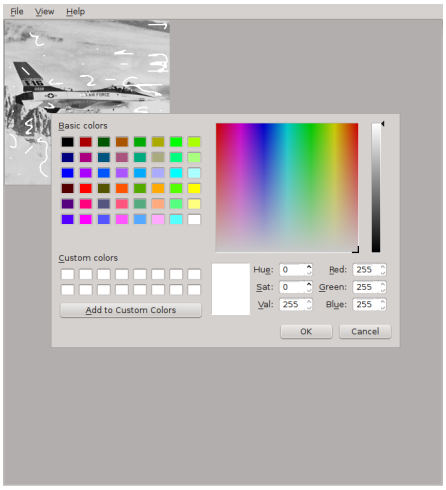
另外，通过比较图 4–3a和 4–3b也可发现，相对于旧的基于分段多项式插值的插值方法，新的 B 样条插值产生的 scribble 明显更为光滑。不过，值得商榷的一点是，我们在连接各 scribble 支撑点时，对于经过点的顺序并未特别讲究，而仅仅是按照行坐标从小到大来进行顺序连接。这样产生的有些 scribble

弯曲幅度比较大。

图 4–1: Airplane 图像与其 scribble 的叠加

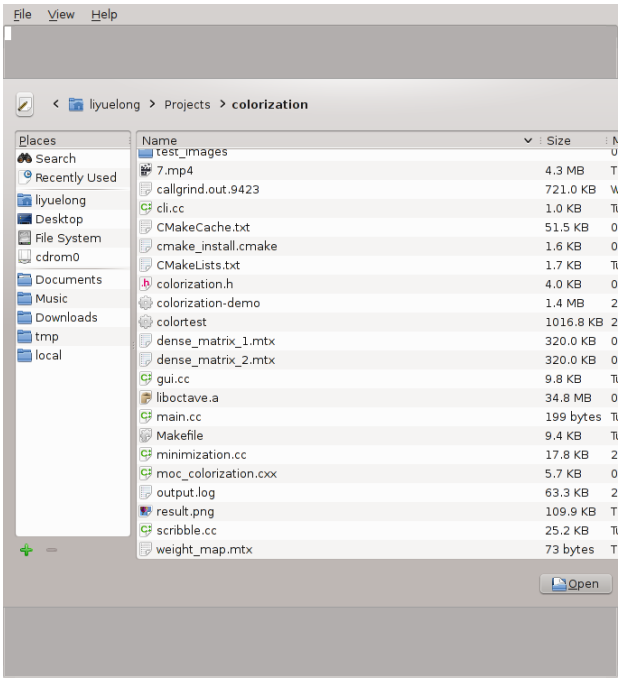
图 4–5将我们的实现和原实现的处理结果进行了对比。从图中可以看出，在对各图像的处理上，我们的算法颜色偏淡，而原算法则有些过于饱和。我们的算法有时候会丢失一些局部的颜色信息（注意小女孩的奶嘴，原来是蓝色的在我们的处理结果中变为了白色），而原算法则可能添加一些原图没有或者饱和度没那么高的颜色（例如：小女孩的头发比原来的要亮，而 Lena 的脸色也比原图略深）。单从视觉效果上来看，两种算法实现均能取得看上去比较让人满意的效果。

此外，由于我们将算法移植到了 C++ 平台，而且在优化运算方面做了很多努力，因此我们的实现比原 Matlab 实现快了许多。这一差异在 scribble 生成上体现的尤其明显。在笔者的 Dell Inspiron N4120 笔记本、Debian 7 操作系统上，C++ 程序和 matlab 程序在 Matlab 2012b UNIX 版下的运行时间见表 4–3和 4–4。其中表 4–3是 scribble 生成算法的对比结果，而表 4–4是彩色化算法的对比结果。我们测试的图像为 Airplane、Child、Lena、以及 Fruits。为避免偶然情形，对于每一副图像，我们重复进行了 5 次实验。



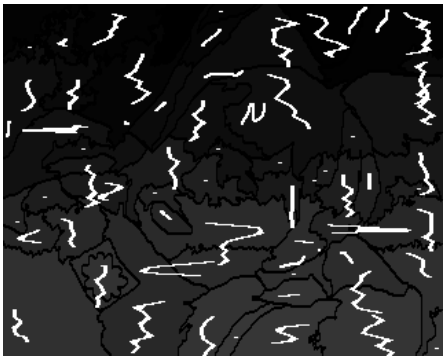
(a) 选色界面

(b) GUI 的显示 scribble 功能



(c) 打开图像截图

图 4–2: 图形界面



(a) 原实现有些 scribble 跨越了边界

(b) 我们的实现不再有 scribble 跨越边界的问题

图 4–3: scribble 越界问题

表 4–3: Matlab 和 C++ 程序的运算时间对比（scribble 生成部分）（单位：秒 (s)）

图像	尺寸	C++ 算法					Matlab 算法				
序号		1	2	3	4	5	1	2	3	4	5
Airplane	256 × 256	0.26	0.27	0.26	0.27	0.27	13.63	14.27	14.43	14.76	14.59
Child	256 × 320	0.33	0.34	0.35	0.32	0.34	30.46	29.08	28.22	31.77	27.95
Lena	256 × 256	0.30	0.28	0.29	0.28	0.28	19.61	19.60	19.95	20.85	19.79
Fruits	480 × 512	1.08	1.15	1.55	1.65	1.68	101.20	99.01	116.35	100.10	99.62

表 4–4: Matlab 和 C++ 程序的运算时间对比（优化部分）单位：秒 (s)

图像	尺寸	C++ 算法					Matlab 算法				
序号		1	2	3	4	5	1	2	3	4	5
Airplane	256 × 256	5.86	5.67	5.84	5.86	6.33	23.45	36.04	23.43	25.09	23.67
Child	256 × 320	9.95	10.80	9.65	9.43	10.30	31.21	32.13	31.06	30.80	32.13
Lena	256 × 256	4.81	5.08	5.42	5.16	5.23	24.64	24.40	24.32	25.29	24.52
Fruits	480 × 512	45.36	45.00	48.14	46.26	46.06	98.40	117.65	116.56	115.87	114.90

(a) 原图 1

(b) 原图 2

(c) 上色良好

(d) 上色不充分

图 4–4: 上色良好的图像与上色不充分的图像对比；参数均为 $k = 0.01, \sigma = 0.8, c = 0.05^2 * MN$

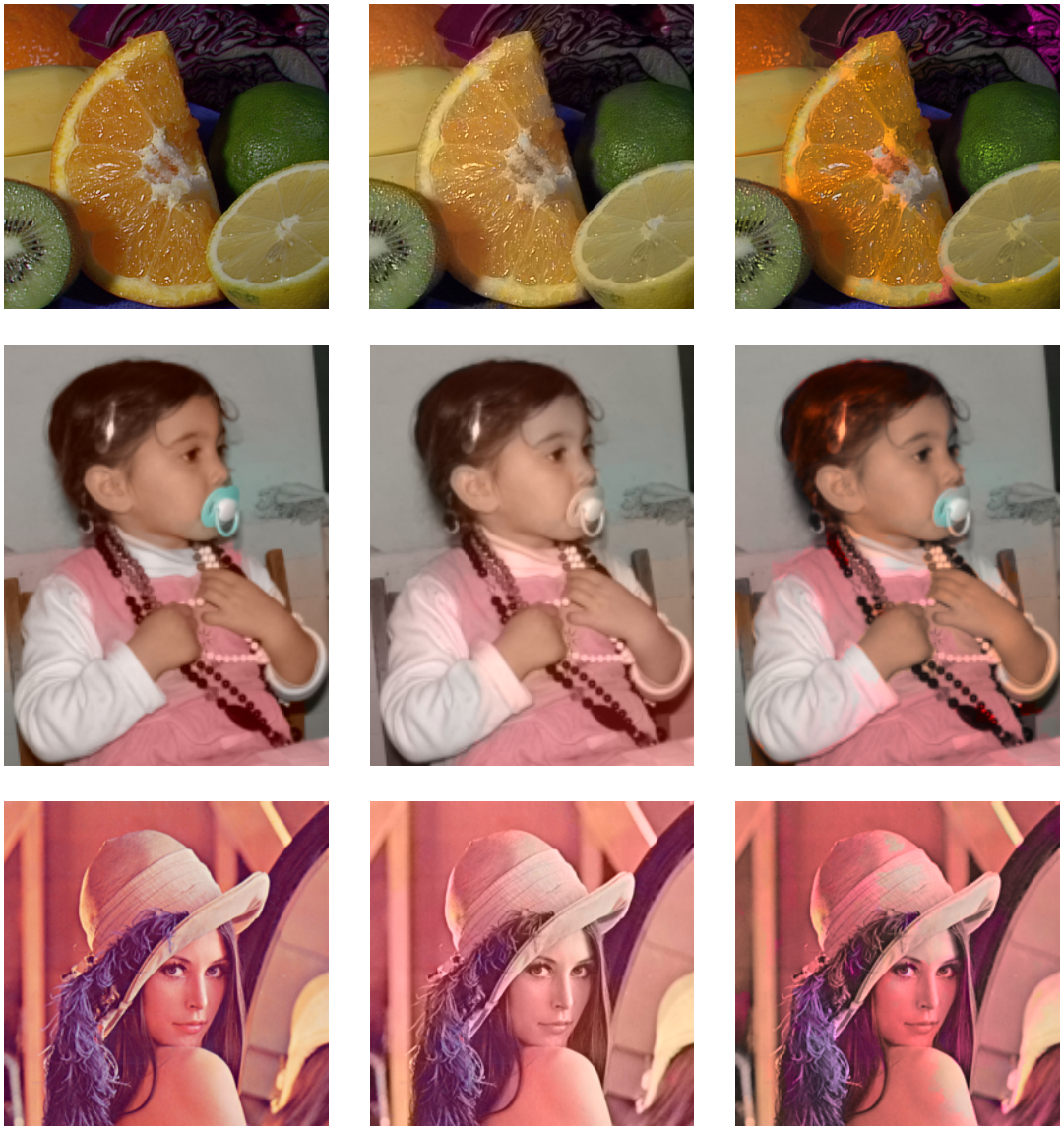


图 4–5: 我们的算法与原实现处理效果对比
左边是原图，中间是我们的处理结果，右边是原有实现处理的结果

从表中不难看出，在所有情形下 C++ 算法均比 Matlab 算法有大幅度的性能提升；尤其是对于 scribble 生成算法，这一性能的差距更为明显。特别的，对于 Child 和 Fruits，其差距相差接近 100 倍。这得益于我们先前针对算法所进行的大量优化，以及 C++ 相对于 Matlab 本身在性能上的诸多优势，例如大量循环的嵌套并不会造成性能低下等。至于优化部分的性能提升则没有这么明显。这是因为优化部分大部分的运算消耗在于求解稀疏线性方程组，而 Matlab 对于数值算法本身进行了深度优化，其运算性能已经相当优秀。即便如此，利用 C++ 自身的优势，我们依然获得了 2-4 倍的性能提升。

需说明的是，由于交互式界面不便于计算执行时间，因此我们在对比时 scribble 上的颜色直接采取的原图颜色，而不是通过用户人工上色。这当然是一种比较理想的情形，不过在对比运算时间上，由于数据量相差不大，因此不会造成太大偏差。

4.5 本章小节

本章我们主要讨论算法实现的一些细节，以及实验处理的结果和一些讨论。通过以上诸多图片的比较和分析，不难发现我们的实现相对于原有算法的一些改进，也同时可以发现我们的算法还存在一些不足。

从前面的讨论可知，后续处理（包括 scribble 的生成以及最终的上色）在很大程度上依赖于分割算法的性能。而现有的分割算法在处理效果上还存在一些局限性。这使得我们的算法也受到了影响。

我们产生的 scribble 由于受几何形状的约束，有时候其伸展受到约束。后续工作可以考虑对该约束进行适当的放宽。

全文总结

整个毕设的过程中，在课题组前期工作的基础上，我们继续完成了前期工作中未能完成的部分；我们成功的将原算法由 Matlab 平台移植到了 C++ 上，完成了完整的 scribble 生成算法和优化算法，以及基于 Qt 开发的图形界面。更进一步的，对于现有工作中的不足之处进行了修补和完善。

从实验结果来看，我们的 C++ 实现修正了 Matlab 实现存在的一些问题，处理的效果也比较令人满意。在运算效率上，我们相对于原来的算法有了较为明显的提高。

然而，我们目前达到的效果远远谈不上完美；还有很大的改善空间，有待在后续工作中继续加强。其中几个可以考虑的改进方向有：

- 分割方法的探讨；如文章中所提到的，目前所使用的基于图的分割算法对于参数比较敏感，这使得后续的 scribble 生成算法也受到影响；
- 估价函数的设计；现有估价函数的形式（式 3–6和式 3–5）形式还比较简单，可能还存在一定的改进空间。例如，式 3–6中采用量纲不同的量相乘，或许未必妥当。在后续工作中可以探讨是否需要归一化等等。
- scribble 生成中，可能存在提前中止等现象，需要进一步完善。另外，也可以考虑发展不依赖于图像分割的 scribble 产生算法，以避免现有的因为分割不当而导致的很多问题。
- 参考新的彩色化研究成果，从中吸取和借鉴。例如参考 [21] 的研究，尝试进一步改善算法的效率等。

此外，目前实现的算法未能经过充分的测试，可能仍有遗留的 bug；虽然我们在实验中未发生崩溃或错误，但在 Valgrind 等软件的监控下确实可以发现一些问题。比如有些变量被指示未能合理初始化等。这些也是有待将来继续解决的。

参考文献

[1] Levin A, Lischinski D, Weiss Y. Colorization using optimization[C]//ACM Transactions on Graphics (TOG). .[S.l.]: [s.n.] , 2004, 23:689–694.

[2] Yatziv L, Sapiro G. Fast image and video colorization using chrominance blending[J]. Image Processing, IEEE Transactions on, 2006, 15(5):1120–1129.

[3] Yu Z, Xu Y, Yang X, et al. Structure-Preserving Colorization Based on Quaternionic Phase Reconstruction[M]//Advances in Multimedia Information Processing-PCM 2009.[S.l.]: Springer, 2009:847–857.

[4] Sapiro G. Inpainting the colors[C]//Image Processing, 2005. ICIP 2005. IEEE International Conference on. .[S.l.]: [s.n.] , 2005, 2:II–698.

[5] Li Y, Lizhuang M, Di W. Fast colorization using edge and gradient constrains[J]. Proceedings of WSCG'07, 2007:309–315.

[6] Kim T H, Lee K M, Lee S U. Edge-preserving colorization using data-driven random walks with restart[C]//Image Processing (ICIP), 2009 16th IEEE International Conference on. .[S.l.]: [s.n.] , 2009:1661–1664.

[7] W_glarz J. Project scheduling: recent models, algorithms, and applications[M], Vol. 14.[S.l.]: Kluwer Academic Pub, 1999.

[8] Semary N. Image coloring Techniques and Applications[M].[S.l.]: GRIN Verlag, 2012.

[9] Ho C w, Ke C y, Wu T l, et al. COLOR ADJUSTMENT CIRCUIT, DIGITAL COLOR ADJUSTMENT DEVICE AND MULTIMEDIA APPARATUS USING THE SAME[缺文献类型标志代码]. US Patent 20,130,044,122.

[10] Luan Q, Wen F, Cohen-Or D, et al. Natural image colorization[C]//Proceedings of the 18th Eurographics conference on Rendering Techniques. .[S.l.]: [s.n.] , 2007:309–320.

[11] Felzenszwalb P F, Huttenlocher D P. Efficient graph-based image segmentation[J]. International Journal of Computer Vision, 2004, 59(2):167–181.

[12] Lehua W, Jing S. An improved multiscale maximum entropy image restoration algorithm[C]//Innovative Computing, Information and Control, 2006. ICICIC'06. First International Conference on. .[S.l.]: [s.n.] , 2006, 1:640–643.

[13] 孙君顶, 丁振国, 周利华. 基于图像信息熵与空间分布熵的彩色图像检索方法 [J]. 红外与毫米波学报, 2005, 24(2):135–139.

[14] Zachary Jr J M. An information theoretic approach to content based image retrieval[D].[S.l.]: Citeseer, 2000.

[15] Rao A, Srihari R K, Zhang Z. Spatial color histograms for content-based image retrieval[C]//Tools with Artificial Intelligence, 1999. Proceedings. 11th IEEE International Conference on. .[S.l.]: [s.n.] , 1999:183–186.

[16] 丁晓伟. 黑白图像彩色化技术 [M]. 上海: 上海交通大学电子信息与电器工程学院电子工程系, 2012.

[17] ICT S. The SINTEF Spline Library Reference Manual[J]. 2005.

[18] Ding X, Xu Y, Deng L, et al. Colorization using quaternion algebra with automatic scribble generation[M]//Advances in Multimedia Modeling.[S.l.]: Springer, 2012:103–114.

[19] Chan W L, Choi H, Baraniuk R G. Coherent multiscale image processing using dual-tree quaternion wavelets[J]. Image Processing, IEEE Transactions on, 2008, 17(7):1069–1082.

[20] John W Eaton S H, David Bateman. GNU Octave Mannual[M].[S.l.]: [s.n.] , 2011: 425 – 430.

[21] Sỳkora D, Buriánek J, Žára J. Unsupervised colorization of black-and-white cartoons[C]//Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering. .[S.l.]: [s.n.] , 2004:121–127.

.1 谢辞 *

在临此毕业之际，首先我非常感谢我的指导老师徐奕老师。徐老师为人诚恳，工作认真，关心学生。自从“数字图像课程”起，我一直得到你不计回报的帮助。多少次你耐心的为我们讲解问题，授业解惑；多少次你教会我们科研要勤恳认真，态度严谨。在毕设的过程中，每次有问题都能得到你及时的帮助；从任务书，开题报告，中期报告到现在的毕设论文，哪一次不是您不辞辛劳认真帮我们修改。能成为您的学生，真是一种幸运。

我还要感谢给我授过课的各位老师，你们不仅哺育了我知识，更以你们敬业的精神给予我鼓舞。你们高尚的师德，比课本上的知识更能给予我启发意义。

当然，我也得感谢我的母校上海交通大学。是你提供给我一片赖以成长的土壤，是你用浓郁的学术氛围滋养着我。现在我行将离开这所学校，但是对我而言，这将是走向另一个开始。我将永远铭记在闵行校区温馨快乐的四年时光。

我也要感谢我的父母。是你们给了我来到这个世界的机会。是你们教授我做人的道理，帮助我树立了做人最基本的伦理观念。是你们在我最失落的时候陪伴在我身边，鼓励我不断进取。大学四年，我最盼望的就是回家的那些日子。

最后，我要感谢那些默默奉献的开源软件的作者。你们的出色工作为我批荆斩棘。站在巨人的肩膀上，才能登高瞰远。正是你们的辛勤打下了坚实的地基，才让我书写毕设代码少了许多无谓的辛劳。

IMAGE COLORIZATION TECHNIQUES BASED ON QUATERNION WAVELET TRANSFORM

Image colorization is the task of coloring a grayscale image, or equivalently, assigning from the single dimension of luminance or intensity a quantity that varies in three dimensions, such as red, green, and blue channels. Since we may take numerous colors for a given intensity value, the colorization problem is regarded as 'ill-posed' and thus requires certain amount of human interaction or external information.

There exists various forms of human intervention during the colorization processes; among them a typical one is to require human to manually annotate the image with a few color scribbles. Then, based on the assumption that neighboring pixels with similar intensities should have similar colors, a quadratic cost function is optimized, which lead to color propagation throughout the whole image, and in effect obtain colorization results. Levin is the first person coming up with this form of methods, and many people follow up to seek for improvements, such as Yatziv, Sapiro, etc.

Although such method eliminate in part the time consuming manual work, people still need to perform the tedious task of drawing scribbles; in addition, as the colorization result depends largely on information provided by scribbles, the colorization results might be unstable. Finally, this method focuses on intensity variations only, and thus is ignorant of structural differences. Further improvements may be obtained if structural information is to be considered.

In our previous work, a novel method was proposed as an attempt to fix all the above mentioned limitations. First, an automatic scribble generation scheme was developed, on the basis of spatial distribution entropy calculations. Then an improved cost function was calculated, which took into account the quaternion Gabor filtering coefficients and the hierarchical Canny edge map. These two parts represent the structural information in various directions and the edge information separately. We have also implemented this sophisticated algorithm using Matlab, and experimental results showed that we did obtain better results under certain circumstances. For instance, our algorithm proved more capable

of preserving complex color textures, and suffered much less from color leakage.

However, our implementation was a bit premature and thus had certain bugs to fix. We also plan to implement it on C++ platform, and this work has just begun. Therefore, we still need much to do to migrate our algorithm from Matlab to C++.

Under close scrutiny we found that the original implementation of scribble generation algorithm suffers from the following problems:

1. It intentionally avoid the use of recursive algorithms, and consequently the code is poorly structured and hard to maintain; In fact, our algorithm is among such kinds that could be easier implemented via recursion, and this should bring about negligible extra cost;

2. It does so many unnecessary work that it is inefficient both in terms of time and memory; for instance, it stores huge amount of redundant data during traversing the scribble points, and this could not only waste memory but also lead to performance loss;

3. Somewhere the code isn't consistent with what the paper stated; for instance, the paper said that the starting point for each segment is the geometry center, wheres this does not apply in the code.

Therefore, we aim at fixing those problems while writing our C++ implementation. Specifically, we use depth-first-search(DFS) to search for scribble candidates; we propose a new algorithm to find the starting points, and manage to have them near to the centroids of each segment while keeping them inside the segments; we also do a lot of work to cut off the unnecessary resource usage, and we succeed in saving huge memory and improving the execution speed simultaneously.

In addition to some minor fixes, we also seek for some improvements. For instance, we use B spline interpolation instead of piecewise polynomial interpolation, which is the method used in the original Matlab implementation. Our goal is to obtain more smooth scribbles, and the results look promising.

In terms of colorization optimization algorithms, we get the inspiration from Levin's method, in which the cost function was represented by a huge sparse matrix, whose each column corresponds to the weights of a single pixel. Furthermore, this matrix is a banded matrix, and there exists efficient software packages aiming at dealing with this special kind of sparse linear systems.

Although we have tried hard to overcome existing limitations, some problems still remain to be solved. For instance, our search algorithm might stop prematurely for certain shapes of segments, and due to the inherent limitations brought about by the segmentation algorithm currently in use, our scribbles turn out very segmented occasionally. Chapter three in our paper is dedicated to discussion about algorithm issues and possible limitations.

In practice, we divide the whole software implementation into three parts:

1. The scribble generation module;
2. The optimization module;
3. The graphical user interface(GUI);

Specifically, we utilize the SISL(SINTEF spline library) library to do B spline interpolation; for solving banded sparse matrix we use the well-known open source math library liboctave; and we write our graphical user interface on Qt platform, which is a really stable and efficient open source GUI library, and could easily interact with OpenCV. When these parts are combined together, the user could easily use this program by first loading an image, then displaying scribbles, and finally getting the colorized results. For detailed implementation issues, please refer to chapter four.

Finally, we carry out a comparison between our newly implemented C++ program and the original Matlab code. The results look promising, in that our implementation successfully overcome the limitations the original Matlab code has, and have some improvements. Judging from the colorized images, it seems that the C++ program could produce results at least comparable to the Matlab one. Generally the C++ program could produce more smoother scribbles, and

print out colorized images which in some cases better preserve the original colors. However, in most cases, both the C++ and the Matlab program differ slightly from the original image; and it is easily observed that the output images of the C++ program tend to be 'shallow', wheres the Matlab program has the tendency to produce images with 'darker' colors.

Additionally, the C++ program is much faster than the Matlab program. And this is one of our primary goal when implementing the C++ program. Our success is due to reasons from two aspects: first, C++ is inherently much faster than Matlab, it does not require vectorizing to get the code run faster; second, we optimize the code so that it does few 'unnecessary tasks', which is the key to code optimization. And the net result is that the C++ program takes over ten times less than the Matlab one.

Despite achieving some improvements, our algorithm still has its limitations. And there is a huge amount of work that needs to be done to get really pleasant results. Several of the possible enhancements are proposed below:

1. As motioned above, performance of our algorithm depends largely upon the segmentation algorithm used in the first step. Our algorithm is so sensitive to the segmentation results that we need to find a better one, or merely eliminate (at least in part) the need of segmentation;

2. Our scribble searching algorithm could stop prematurely, and this is mainly due to irregularities of some segments. We may think of utilizing the convex hull in order to remove such irregularities;

3. Think about more sophisticated cost function forms; the cost function currently being used multiply the magnitude and phase of Gabor filtering coefficients, which may lead to potential problems, and we may need to regularize it first.