仪器科学与光电工程学院

SCHOOL OF INSTRUMENT SCIENCE & OPTO-ELECTRONICS ENGINEERING

《数字图像处理》（课程代码：E17D3430）

大作业设计报告

设计题目:

书画破损碎片图像拼接及修复

项目组成员	分工及主要完成任务	成绩
王云帆 13171104	整体算法设计，部分底层函数编写，预处理功能实现，匹配算法功能实现、碎片图像获取，报告撰写、校对、定稿	
陈霁阳 13171115	参与算法设计，灰度匹配实验，图像拼接部分功能实现，拼接实验部分，函数封装及代码整理，报告撰写	

注：分工较为均衡，希望得分一致

北京航空航天大学

2016 年 6 月

成 绩_____

北京航空航天大學
BEIHANG UNIVERSITY

数字图像处理大作业

书画破损碎片图像拼接及修复

院（系）名称 仪器科学与光电工程学院

专 业 名 称 测控技术与仪器

学 生 学 号 13171104

学 生 姓 名 王云帆

专 业 名 称 探测制导与控制

学 生 学 号 13171115

学 生 姓 名 陈霁阳

指 导 教 师 周富强

2016 年 6 月

书画破损碎片图像拼接及修复

王云帆、陈霁阳
（北京航空航天大学 仪器科学与光电工程，北京）

摘要：本文提出了一种二维数字图像拼接技术用于书画破损碎片的数字化修复。在数字修复过程中，首先对书画破损碎片的扫描图像进行预处理，之后依照碎片图像边缘灰度特征和轮廓子片段长度、数量等几何特征计算潜在的配对组合，根据量化评估结果得到最佳拼接方式拼接图像。多张碎片组合实验证明了程序的效果，实验结果展示了程序在拼接图像上的高效性和鲁棒性。

关键词：碎片、匹配、图像拼接

0 引言

古书画自身材料属性决定其脆弱、易损，其中碎片拼接是破损书画修复的主要任务之一。目前数字化图像处理技术广泛应用于文物文献修复中，通过扫描及计算机处理可实现对破损书画的数字化拼接复原。古书画碎片常有霉菌、墨点、折痕、刮痕等大面积污损，且局部存在颜色扩散而导致的模糊及灰度变化。传统上，拼接复原工作需由人工完成，准确率较高，但效率很低。特别是当碎片数量巨大，人工拼接很难在短时间内完成任务。数字图像碎片拼接技术通常用于重组一组碎片并且重构成一张完整的原始图像，随着计算机技术的发展，人们试图开发鲁邦性好的几何图形学算法用于碎片图像的自动拼接技术，以减少人工劳动，提高拼接复原效率。

匹配算法是碎片图像拼接算法的关键环节，现有的碎片图像拼接方法中匹配环节可以分为两类：1、基于颜色（灰度）匹配；2、基于几何特征（轮廓形状）匹配。基于颜色匹配以颜色信息为基础预测碎片图像之间的相邻关系，指导匹配和拼接，常用的匹配关系包括边缘颜色差、灰度矩等。该类算法效率较高，但是当轮廓不同但边缘灰度相似时无法匹配。几何特征匹配以碎片边缘轮廓为基础，常用特征包括轮廓角点、轮廓子片段斜率变化趋势等，该种匹配方法可以沿着碎片共有的边缘轮廓特征精准的配对碎片。该种匹配方法速度较慢且当碎片规则或多碎片轮廓相似时，常常难以得到正确的结果。本项目提出一种综合碎片图像灰度信息和轮廓信息的算法，用以书画碎片匹配。结合颜色和几何信息的匹配算法使综合两种算法的优点，一定程度上使匹配处理更加高效、精准、可信。

1 算法描述

书画碎片图像拼接算法分为 3 个主要步骤：图像预处理、碎片匹配和碎片拼接。具体流程图如下

图所示：

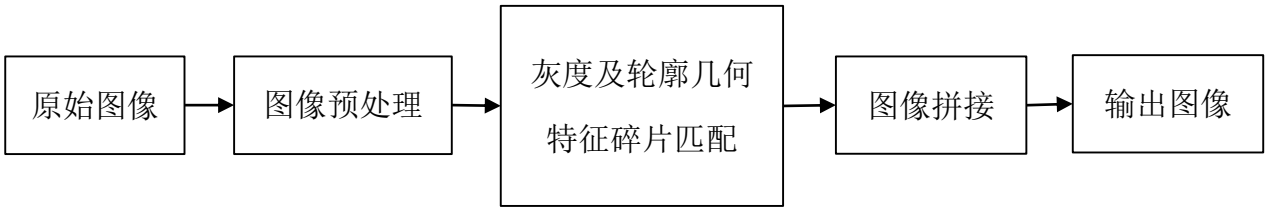

图 1

1.1 碎片图像预处理

预处理的目的是通过图像分割识别书画碎片目标。书画碎片图像常常由真实碎片经扫描或高分辨率相机拍照得到，其背景通常为纯色因而图像分割较为容易。对于一般白背景扫描的灰度图像，其直方图在近255区域有明显峰值，具有双峰性。一般碎片直方图如下图所示：

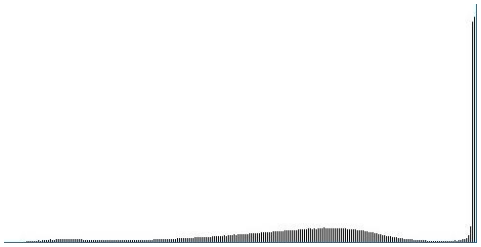

图 2 一般书画碎片直方图

考虑到书画碎片内部图案杂乱，直接运用Canny算子等边缘提取算法提取边缘会差生大量不需要的“碎边”，因此利用直方图特性首先设置较高的阈值将原始图像二值化，运用中值滤波滤波平滑图像，滤除轮廓内部噪声，最后利用Canny算子识别碎片并提取轮廓。此过程获得较好的轮廓图像用于后续操作。值得注意的是，中值滤波为非线性滤波，但由于匹配过程对边缘轮廓细节要求不大，因而影响可以忽略。

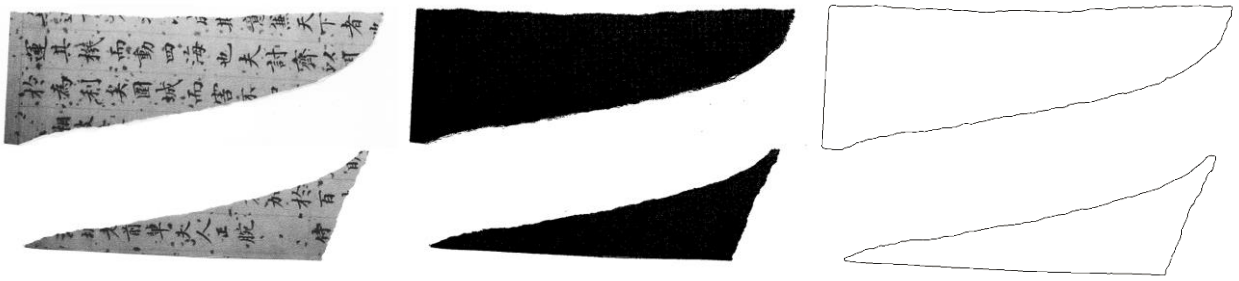

图 3 预处理图像。从左到右分别为碎片灰度图像，二值化图像，Canny 算子提取边缘后图像。

1.2 基于灰度及轮廓几何特征的碎片匹配

碎片匹配的目的是为“成对”识别相邻碎片并计算相邻碎片变换关系，对于两个以上的碎片，应

能首先识别出碎片中两个最有可能拼接的碎片。实现算法如下：（1）提取每个碎片的边缘轮廓并以曲线轮廓方式表示，将曲线打断成若干子片段（2）分析边界片段构成轮廓的形状和灰度信息，依照两碎片轮廓间子片段组合的几何和灰度相似关系生成潜在配对组合（3）评价潜在的配对组合，从中寻找最佳拼接方式。算法细节在2中阐述。

1.3 图像拼接

按照最佳匹配结果，进行图像拼接。首先从多张碎片构成的图像中分离碎片，依照先前生成的最佳子片段组合旋转平移碎片，搬移相应碎片至接合位置实现图像拼接，得到完整图像。算法细节将在3中阐述。

2 碎片图像匹配

碎片图像拼接的核心是寻找相邻碎片的变换关系。碎片轮廓信息是拼接可利用的重要线索之一，往往不同碎片具有不相似的自然不规则边缘。对于二维图像，碎片可以以轮廓信息建模，并且以二维曲线轮廓配对，因此匹配问题常常简化为曲线匹配问题。

对于相邻碎片，他们的拼接边缘必然具有高度相似的几何和灰度特征，这种匹配关系是拼接匹配的重要依据。我们首先将每个封闭轮廓打碎成若干子片段，由于两个碎片轮廓相邻（匹配），则必然拥有一个以上长度相近，灰度相似的子片段，这种子片段定义为共有子片段。根据上述原理，只要搜索出所有共有子片段的组合即可计算出相邻碎片的所有拼接可能。

匹配算法主要分三步实现：（1）将轮廓打碎成若干片段，以子片段集合形式表示（2）计算两个轮廓每个子片段间的特征匹配，生成轮廓拼接方式的组合（3）量化评价所有组合变换关系，根据得分选出最佳匹配组合。算法细节如下所示。

2.1 轮廓子片段化

首先，对于每个碎片F_i的曲线轮廓C_i，我们运用多边形近似分割轮廓为若干弧线子片段S_{ij}。定义轮廓C_i生成的子片段形成的向量为轮廓簇$\boldsymbol{S_i}$，即$\boldsymbol{S_i} = \{S_{ij}\}$。每个子片段起始点为$v_{ij1}$，终止点为$v_{ij2}$，即$S_{ij} = \overset{\frown}{v_{ij1}v_{ij2}}$。当多边形边数较多时，每个弧线子片段可以由$v_{ij1}$和$v_{ij2}$两点所表示的直线近似，即$S_{ij} = \overline{v_{ij1}v_{ij2}}$。

本实验中，我们使用findContours函数从预处理的图像中提取轮廓向量，运用DP算法的approxPolyDP函数进行多边形近似，生成向量。

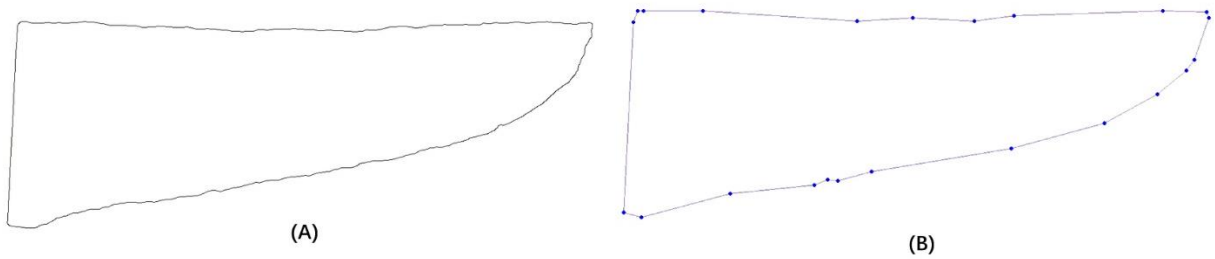

图 4 多边形近似结果

注意到，多边形近似的结果受起始点v_{ij1}，终止点v_{ij2}位置选择的影响，但考虑到是当多边形近似阈值设置足够小时，曲线轮廓的分割将足够精细，近似结果仍然主要由轮廓C_i的几何特征决定，因此近似结果对于v_{ij1}、v_{ij2}的位置选取不敏感。

2.2 特征匹配

若两碎片F_1、F_2相邻，则其轮廓簇$\boldsymbol{S_1}$、$\boldsymbol{S_2}$必然共享一个（或多个）子片段。设子片段$S_{1i} \subset \boldsymbol{S_1}$、$S_{2j} \subset \boldsymbol{S_2}$，则$S_{1i}$和$S_{2j}$长度相近，灰度相似。这种匹配关系即是两碎片拼接的依据，也是搜索匹配组合的依据。

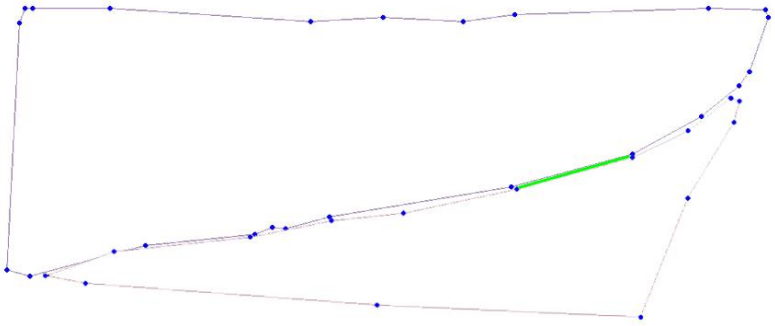

图 5 共享子片段

根据步骤1中生成的轮廓簇$\boldsymbol{S_i}$，计算其子片段的长度和平均灰度，生成描述子向量$\boldsymbol{CharaC_i}$。遍历两个轮廓$\boldsymbol{CharaC_1}$、$\boldsymbol{CharaC_2}$中的每种子片段组合，若两子片段S_{1i}和S_{2j}的平均长度和平均灰度均满足阈值，则记录这种标号组合关系，由此得到匹配组合。

2.3 量化评估

上步中生成的匹配组合存在大量的错误拼接方式，因此需要量化评估从中筛选出最恰当的拼接方式，得分越高说明两张图拼接越正确。若S_{1i}和S_{2j}为最佳拼接方式，则以它们的自子线段重心为旋转点，按照其变换关系（平移矢量、旋转矩阵）拼合轮廓后会有大量子片段重心相近，且直线近似方向相似，即拥有较多的共有子片段。图 5 展示了这一拼接效果，图中圆点代表子片段重心，可以看到，接缝处有大量重心、子线段方向符合要求。

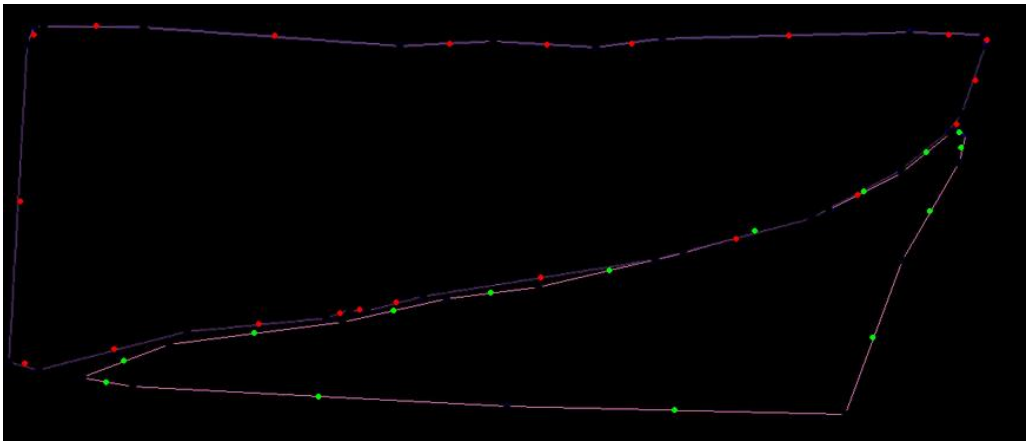

图 6 重心图

这样匹配的子片段数量越多，共有子片段的长度越长则该拼接方式最佳，因此该种情况的得分也应该越高。令 l 为长度，n 为数量，对于评价函数$y = f(l, n)$这种函数必须具备以下特点：

1. l 大且 n 大时 y 越大，仅 l 大或仅 n 大时 y 不大
2. l 和 n 量级不同，但影响权重相似
3. y 为连续单调函数，在第一卦限恒为正，且不存在奇点

我们选取$y = l^p \cdot n^q$作为量化评估函数，用以计算匹配关系的得分，其中 l 代表长度，n 代表数量，p，q 调节权重。考虑 l 的量级大于 n，经过实验发现，p 取 0.5，q 取 1 具有良好的效果，即$y = \sqrt{l} \cdot n$，其函数图像如下图所示，可以看到该函数满足上述三点要求。

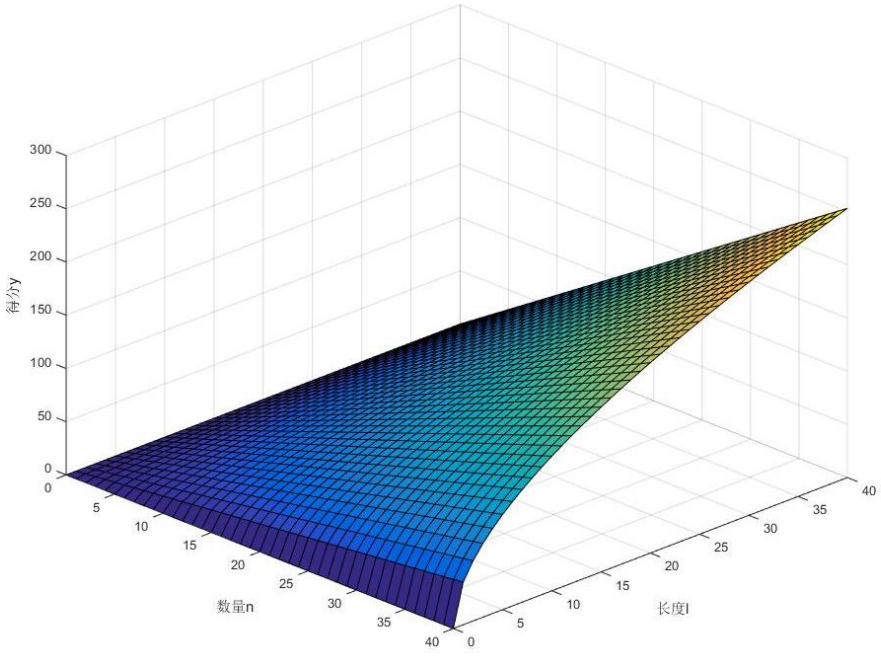

图 7 $y = \sqrt{l} \cdot n$函数图像

对于两轮廓C_1、C_2，S_{1i}和S_{2j}生成的轮廓间变换关系为T，令C_2变换后的轮廓为T(C_2)。考虑子片段$S_{1m} \subset C_1$和$S_{2n} \subset T(C_2)$，其长度分别为l(S_{1m})、l(S_{2n})。找到所有子片段重心欧氏距离小于阈值，且直线近似子片段的斜率小于阈值的组合(S_{1m}, S_{2n})，由以下公式计算

$$W_1 = \sum \frac{l(S_{1m}) + \mathrm{l}(S_{2n})}{2}$$

$W_2 = (S_{1m}, S_{2n})$的组合数

S_{1i}、S_{2j}拼接方式的最后得分

$$\mathrm{W} = \sqrt{W_1} \cdot W_2$$

首先依照 2.2 中生成的匹配组合，分别计算相应的变换关系（旋转矩阵和平移矢量），根据上述函数量化得到的分数。设定阈值 k，滤除明显不可能接合的碎片组合。对于同一对碎片的不同拼接方式，只选取其中 W 得分最高的(S_{1i}, S_{2j})作为最终拼接方式变换依据。实验得，阈值 k 的量级应至少为几十左右。

3 碎片图像拼接

匹配后得分最高的标号组合即为拼接匹配方式，根据前述步骤筛选出的(S_{1i}, S_{2j})即可实现坐标变换。程序流程图如下所示：

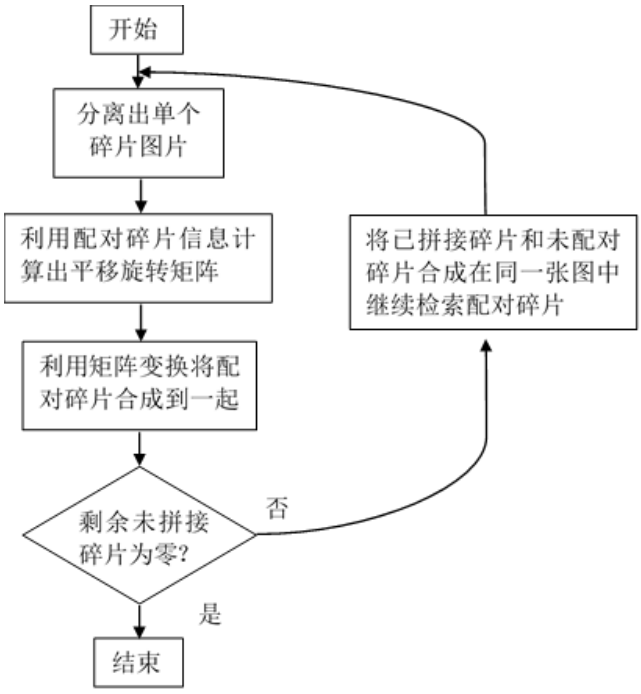

图 8 拼接流程图

3.1 分离单个碎片图片

为了分离多个碎片的图片，需要对其中每一个碎片生成相应的掩膜。首先，可以利用之前查找碎片轮廓的方法，生成每一个碎片的闭合轮廓线。然后利用闭合轮廓线作为掩膜边界，对每个闭合轮廓线利用漫水算法进行填充，将轮廓线内部填充为 255，外部填充为 0，以此作为掩膜图片。最后将所有的掩膜图像分别作用于原始图像即可得到分离的碎片图像。

例如，对于原始图片可以生成两张掩膜图像。

图 9 掩膜图像

利用掩膜分别生成碎片图片即可，结果如下

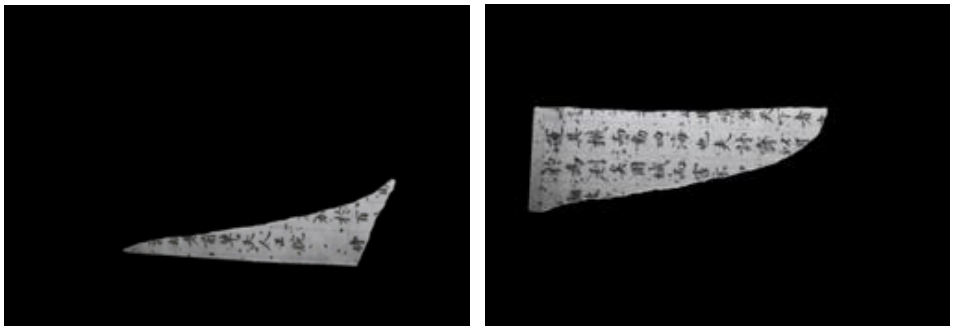

图 10 分离碎片图像

3.2 计算平移旋转矩阵

在碎片图像匹配这一步骤中已经计算出了最佳的匹配方式，其中包含的信息有最佳匹配的轮廓子片段对的坐标位置以及需要旋转的角度信息，即两张匹配碎片中的最匹配的边缘部分的坐标和交角。以此为信息可以计算出将一张碎片平移旋转到另一张碎片的吻合处所需要确定的移动方向、距离，以及转动角度。而方向、距离和转动角度可以用 2*3 的平移旋转矩阵来表示：

$$\mathrm{M} = \begin{bmatrix} cos\alpha & sin\alpha & x_1 \\ -sin\alpha & cos\alpha & y_1 \end{bmatrix}$$

上式中α为图像绕其原点旋转的角度，$\mathrm{x_1}$和$\mathrm{y_1}$为平移方向矢量。用 M 表示图像的变换关系记为

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathrm{M} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} cos\alpha & sin\alpha \\ -sin\alpha & cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

3.3 利用平移旋转矩阵进行碎片拼接

由于实际情况并不是绕图像原点旋转，而是绕匹配轮廓片段的重心旋转，所以将一张碎片拼接到

另一张碎片吻合的位置需要进行不止一次的平移。较为简单的方法是先将待移动的碎片的匹配轮廓重心平移置图像原点处进行旋转，再平移至目标碎片的吻合处即可完成拼接。

完成拼接后的结果图如图 10 所示

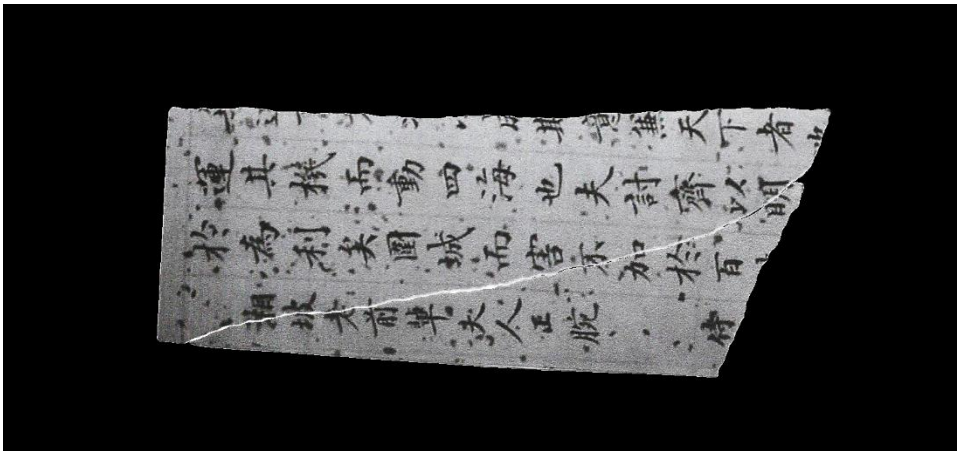

图 11 拼合图像

4 实验结果与分析

考虑文物获取的有限性，为测试程序的运行效果，本实验采用完整古书画图像仿真还原真实书画碎片。我们利用网上搜索的书画图片将其打印，撕碎后扫描生成碎片图像进行程序测试。解决方案与实际应用将有一定差距，但对于初期理论研究仍具有一定意义。

4.1 灰度匹配拼接理论实验

程序编写初期，为了探究边缘灰度对匹配效果的影响及其可行性，我们首先运用规则剪裁的文档进行试验，尝试仅用灰度匹配拼接图像。由于碎片边缘灰度为唯一匹配信息，所以这里假设碎片为规则矩形。现有一张完整图片人工切割出 19 张待匹配的碎片，其中两张碎片如下图：

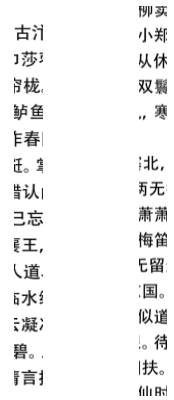

图 12 碎片图像

为了进行灰度匹配，需要提取出每张碎片的左右边缘灰度信息矩阵

$$G_i = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \vdots & \vdots \\ a_n & b_n \end{bmatrix}$$

其中 n 为图像长度，矩阵第一列为左边缘从上到下每一像素的灰度值，第二列为右边缘左右每一像素的灰度值。

评估两块碎片是否匹配时，可以比较G_i右边缘与G_j左边缘的相似程度，由下式进行量化计算：

$$V_{ij} = \frac{\sum_{k=1}^{k=n} \left| G_i(b_k) - G_j(a_k) \right|}{n}$$

当V_{ij}小于一定阈值时即判定为匹配，经实验测得当$V_{ij} < 20$时判定的正确率很高。

将 19 张碎片图进行两两比较，且G_i与G_j需交换位置比较，即一共比较 342 次。比较过程也可以以一张碎片图为起点，分别向左向右寻找匹配的图片，得到正确的图片排序，最后将排列好的图片输出即可。结果如图：

城上层楼叠巘。城下清淮古汴。举手揖吴云，人与暮天俱远。魂断。魂断。后夜松江月满。鲧鲧衣巾莎枣花。村里村北响缲车。牛衣古柳卖黄瓜。海棠珠缀一重重。清晓近帘栊。胭脂谁与匀淡，偏向脸边浓。小郑非常强记，二南依旧能诗。更有鲈鱼堪切脍，儿辈莫教知。自古相从休务日，何妨低唱微吟。天垂云重作春阴。坐中人半醉，帘外雪将深。双鬟绿坠。娇眼横波眉黛翠。妙舞蹁跹。掌上身轻意态妍。碧雾轻笼两凤，寒烟淡拂双鸦。为谁流睇不归家。错认门前过马。

我劝鬓张归去好，从来自己忘情。尘心消尽道心平。江南与塞北，何处不堪行。闲离阻。谁念素损襄王，何曾梦云雨。旧恨前欢，心事两无据。要知欲见无由，痴心犹自，倩人道、一声传语。风卷珠帘自上钩。萧萧乱叶报新秋。独携纤手上高楼。临水纵横回晚鞚。归来转觉情怀动。梅笛烟中闻几弄。秋阴重。西山雪淡云凝冻。凭高眺远，见长空万里，云无留迹。桂魄飞来光射处，冷浸一天秋碧。玉宇琼楼，乘鸾来去，人在清凉国。江山如画，望中烟树历历。省可清言挥玉尘，真须保器全真。风流何似道家纯。不应同蜀客，惟爱卓文君。自惜风流云雨散。关山有限情无限。待君重见寻芳伴。为说相思，目断西楼燕。莫恨黄花未吐。且教红粉相扶。酒阑不必看茱萸。俯仰人间今古。玉骨那愁瘴雾，冰姿自有仙风。海仙时遣探芳丛。倒挂绿毛么凤。

俎豆庚桑真过矣，凭君说与南荣。愿闻吴越报丰登。君王如有问，结袜赖王生。师唱谁家曲，宗风嗣阿谁。借君拍板与门槌。我也逢场作戏、莫相疑。晕腮嫌枕印。印枕嫌腮晕。闲照晚妆残。残妆晚照闲。可恨相逢能几日，不知重会是何年。茱萸仔细更重看。午夜风翻幔，三更月到床。簟纹如水玉肌凉。何物与侬归去、有残妆。金炉犹暖麝煤残。惜香更把宝钗翻。重闻处，余熏在，这一番、气味胜从前。荊暗荷枯一夜霜。新苞绿叶照林光。竹篱茅舍出青黄。霜降水痕收。浅碧鳞鳞露远洲。酒力渐消风力软，飐飐。破帽多情却恋头。烛影摇风，一枕伤春绪。归不去。凤楼何处。芳草迷归路。汤发云腴酽白，盏浮花乳轻圆。人间谁敢更争妍。斗取红窗粉面。炙手无人傍屋头。萧萧晚雨脱梧楸。谁怜季子敝貂裘。

图 13 拼接效果

可以看到，对于边缘灰度信息明显的图像，灰度匹配具有较高的精准性。

4.2 程序测试

我们从百度图片中搜索到一张古书画图像，将其用A4纸打印后随机地撕成若干碎片。碎片最小5cm × 1cm，最大为18cm × 8cm，每张碎片以1200dpi扫描。程序以C++语言编写，在双核4线程2.6GHz Intel(R) Core(TM) i5处理器、4.00GB RAM的笔记本电脑端运行。

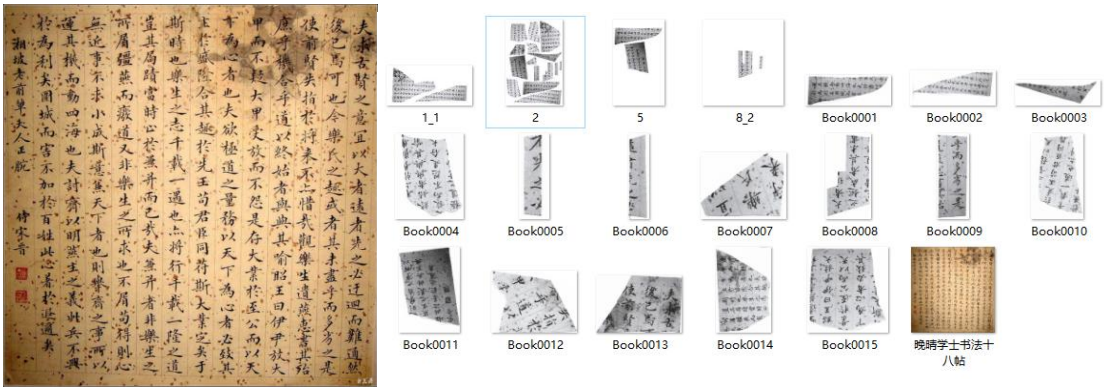

图 14 实验碎片

为测试程序的运行效果，在这里分多种情况测试程序对碎片拼接的准确性。考虑到，若程序可以成功识别并以最佳方式拼接两张碎片，则程序可以以递归的方式完成余下碎片的拼接，该部分程序的快速性主要受递归算法的影响，因此本实验主要以两碎片拼接进行测试，小范围验证程序的高效性和鲁棒性。

（1）选取原本不能拼接的两块碎片如下图所示

图 15

该种情况下，程序计算出的量化评估函数得分应小于设定的阈值。实验表明，程序计算匹配得分为 4.246 分，远低于预期的几十分量级，两块碎片无法拼接，与预期相符。

（2）选取边缘较规则的两块碎片如下图所示

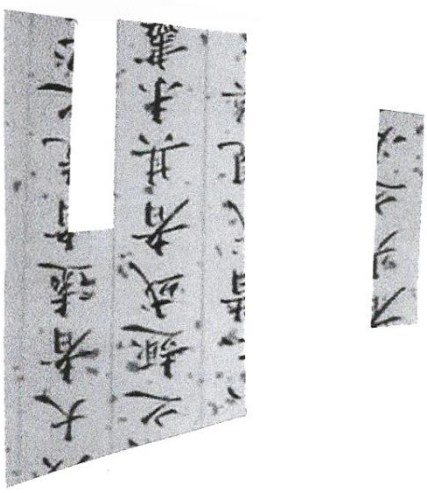

图 16

为检验在边缘形状极为相似的情况下程序拼接的准确性，采用上图所示的碎片组合进行试验。该种情况下，匹配需要更多的依赖于灰度信息。拼接结果如下图所示：

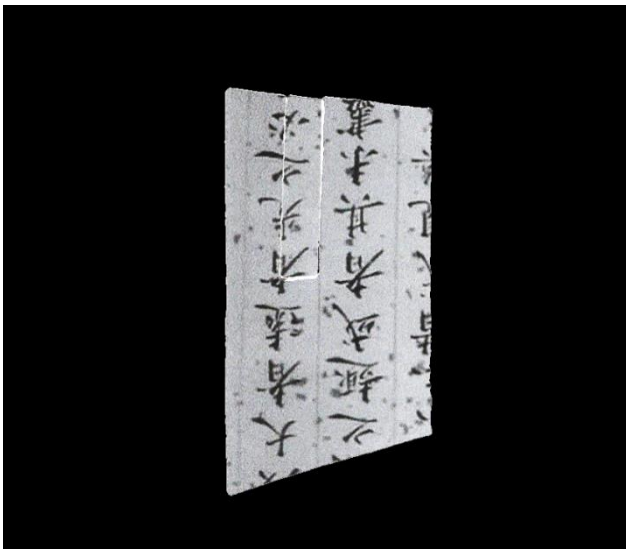

图 17

从拼接的结果可以看出，对于边缘特征不能明显的碎片组合，程序仍然能够正确拼接碎片。该种情况下，量化的分为 34.263 分。

（3）选取更加一般的两块碎片

为了测试程序的普适性，选取了外形较为普遍的两块碎片进行拼接实验，碎片如下图所示：

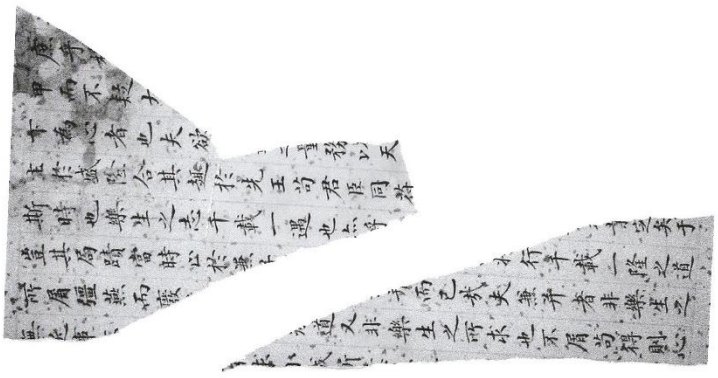

图 18

利用程序进行拼接的结果如下：

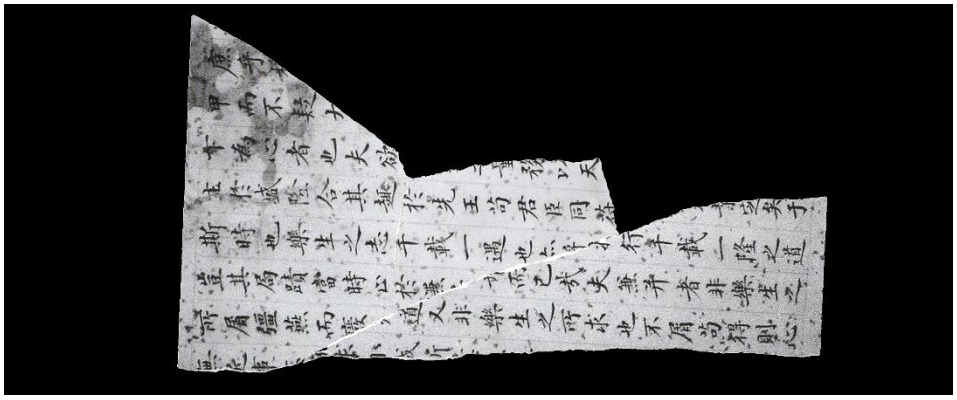

图 19

从结果上来看，碎片拼接的结果尚可，边缘吻合地很好，只是从边缘内容上来看，碎片边缘处部分文字存在错位。类似情况偶尔会出现在边缘形状匹配较好而边缘灰度匹配因素对评估函数得分响较小的时候，这可能是因为该对碎片组合由于边缘灰度信息较为相似，尽管接缝边缘共有子片段很长，程序仍然可能无法准确定位。该种情况量化得分为 62.609 分。

（4）多张碎片共存的图片

程序鲁棒性要求多张碎片同时存在时，程序依然可以识别其中最佳匹配碎片进行拼接。为了检验多张碎片同时进行匹配时程序的运行结果是否会被干扰，选取的如下碎片图进行拼接：

图 20

将源图片中评估函数得分最高的碎片进行拼接结果如下：

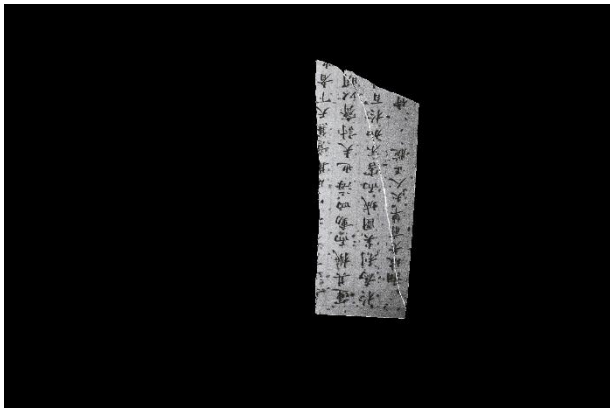

图 21

结果显示程序对于多碎片同时存在时的匹配识别准确，具有较好的鲁棒性。

上述四种情况的得分及运行时间总结如下表所示：

表 1 算法运行时间及量化得分

碎片图像	数量	量化评估得分	时间（s）
图 14	2	4.246	6.309
图 15	2	34.263	6.749
图 17	2	62.609	24.735
图 19	5	82.944	37.716

从上表可以看出，对于两碎片拼接时间从 6s-38s 不等，运行时间长短取决于碎片大小以及碎片数量。对于相同数量下，碎片越大，边缘子片段越多则匹配时间越长。

量化得分是实验的另一个关键点，多次试验发现阈值为 60 可以识别绝大多数正确的组合。此外实验发现，如图 15 所示情况，当碎片边缘整齐且过多依赖于灰度信息时，尽管拼接结果正确但量化得分低于预期，若阈值设置过高可能存在误判的可能，说明固定阈值下量化评估函数仍然有一定局限性。该种情况的准确性与阈值选取有关，动态阈值可能规避该种风险。

上述结果表明，程序可以较大概率的成功识别并以最佳方式匹配碎片，具有一定的准确性、高效性和鲁棒性。

5 结论及未来工作

我们提出了一种图像拼接算法用以实现书画破损碎片的数字化修复，实现了同一图像中两张碎片的最佳拼接。该算法主要分为三个步骤，图像预处理、特征匹配和图像拼接。在特征匹配中，我们运用多边形近似，结合灰度信息和集合信息进行曲线匹配，提出了一种新的量化评价函数用以评估拼接效果以及搜索最佳匹配方式。我们通过不同的碎片图像进行实验，验证了算法的效率和稳健。由于时间原因，未进行多张碎片的拼合探究，理论上依照现两两组合的方式拼接，多碎片拼接仍将具有较好的实验结果。

上述算法存在一定的局限性。程序依赖于 OpenCV 自带的 approxPolyDP 多边形近似函数，实验发现，同一碎片在不同位置或碎片数量增多时，多边形近似结果不同，且碎片多时容易出现较大偏差，导致拼接出错。分析发现该算法可能为全局算法，针对这一问题，未来我们会尝试修改该算法改善鲁棒性。此外，若多张碎片边缘的长度信息和几何信息均相似，或者边缘特征不明显时，得分最高的不一定为最佳匹配方式，可能出现拼错或错位的情况。更高水平的碎片图形纹理特征也许可以改善该问题或者加入精细匹配环节修正错位现象，具体方法有待探究。另外由于编程人员水平有限，主要搜索匹配由遍历实现，程序的运行速度还不够快，考虑将来引入优化的匹配算法用以提升程序的运行效率。

参考文献

[1] 冈萨雷斯，伍兹著 （阮秋琦译） 2011 数字图像处理（第三版）（北京：电子工业出版社）

[2] 毛星云，冷雪飞，王碧辉，吴松森 2015 OpenCV3编程入门（北京：电子工业出版社）

[3] Kang Zhang, Xin Li, A graph-based optimization algorithm for fragmented image reassembly, *Graphical Models* **76** (2014) 484–495

[4] 刘瞬强. 古书画损毁机理初探[J]. 文物保护与考古科学，2003,15(1)：39-42

[5] 王娟等. 图像拼接技术综述[J]. 计算机应用研究，2008,25(7)：1940-1947

[6] 张耀丹. 基于边缘灰度匹配的文档碎纸片拼接[J]. 软件, 2014,35(2): 65-66

分工情况：

王云帆：整体算法设计，部分底层函数编写（对应代码Functions_line.cpp、Function_l.cpp），预处理功能实现（代码PreProcess），匹配算法功能实现（对应代码vpairNum、matchingscore，vBarycenter等）、碎片图像获取，报告撰写、校对、定稿

陈霁阳：参与算法设计，灰度匹配实验，图像拼接部分功能实现（对应代码selectingAndStiching），拼接实验部分，函数封装及代码整理，报告撰写

分工较为均衡，希望得分一致

数字图像处理大作业——书画破损碎片图像拼接及修复

王云帆, 陈霁阳

程序代码

header.h

```
#include "opencv2/opencv.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "iostream"
#include "math.h"

using namespace std;
using namespace cv;

#define WINDOW_NAME1 "[Original Image]"
#define WINDOW_NAME2 "[Contour Image]"
```

function.h

```
#include "header.h"

using namespace std;
using namespace cv;

Mat g_srcImage;
Mat g_grayImage;
Mat g_binaryImage;
Mat g_dstImage1;
Mat g_dstImage2;
Mat tempImage;
int g_nThresh = 230;
int g_nThresh_Max = 245;
RNG g_rng(12345);
vector<vector<Point>> bCenter;
vector<Vec4i> pairNum;
vector<Vec3d> ms;
vector<vector<vector<Point>>>lcluster_2p;

int max(int *a);

//预处理函数
Mat PreProcess(Mat srcImage);

//“从”函数_将边缘轮廓打断成【弧】线段
vector<vector<vector<Point>>> cClusters(Mat srcImage);

//“从”函数_将边缘轮廓打断成【直】线段
vector<vector<Point>> lClusters(Mat srcImage);

//线段长度函数_欧几里得距离
int lineLength(Point a, Point b);

//直线平均灰度函数
int lineAveGscale(Mat srcImage, Point a, Point b);

//计算直线中点坐标
Point lineBaryCenter(Point a, Point b);

//小线段特征矢量表述
vector<vector<Vec2i>> lineCharacter(vector<vector<Point>> Clusters, Mat srcImage);
```

```
//计算matchin的w值
//vec1 轮廓i的小线段特征矢量
//vec2 轮廓j的小线段特征矢量，其中ij为旋转组合标号对
//PairNum 根据ij旋转组合标号得到的【旋转匹配直线标号】组合
double wCalculate(vector<Vec2i> vec1, vector<Vec2i> vec2, vector<Vec2i> PairNum, int a);

//匹配分数计算函数  matchingScore
//*********************************************************************************
//linecharacter 直线近似特征向量
//pairNum 初步匹配标号阵
//lcluster_2p 直线段“丛”近似轮廓
//bCenter 轮廓重心特征向量
//threshold_bc 重心距离阈值
//alpha w计算时权重系数，新版函数该参数已失效
//基本功能：根据初步筛选的标号阵
//*********************************************************************************
vector<Vec3d> matchingScore(vector<vector<Vec2i>> lineCharacter,
                            vector<Vec4i> pairNum,
                            vector<vector<Point>> bCenter,
                            vector<vector<vector<Point>>>lcluster_2p,
                            int threshold_bc,
                            int alpha);

//弧线重心
Point cBaryCenter(vector<Point>segment);

//旋转组合标号阵（初步筛选标号阵）
//标号阵—匹配轮廓1中segment_i和轮廓2中segment_j的标号，组合成初步筛选的标号阵
vector<Vec4i> vpairNum(vector<vector<Vec2i>> lineCharacter, int dlThreshold, int daThreshold);

//计算轮廓重心阵
//lcluster_2p 按两点组合存放的轮廓描绘子
//bCenter 生成重心阵存放的向量
vector<vector<Point>> vBarycenter(vector<vector<vector<Point>>> lcluster_2p,
     vector<vector<Point>> bCenter);

//旋转轮廓重心阵i
//Segment2p_1 线段1，两点向量表示
//Segment2p_2 线段2，两点向量表示
//bc_i 待旋转重心阵
//基本功能：根据线段1和线段2，将轮廓阵旋转至对应位置，得到旋转后轮廓重心阵坐标。注意，bCenter包含多个轮廓的重心阵，
//本函数传入参数仅为单一轮廓的重心阵。
vector<Point> rotateVecBcenter(vector<Point> Segment2p_1, vector<Point> Segment2p_2, vector<Point> bc_i);

//进一步封装多边形近似，返回以两点组成的线段为单元进行储存的向量
vector<vector<vector<Point>>> lcluster2p(vector<vector<Point>>contours_poly,
                                         vector<vector<vector<Point>>> lcluster);

//匹配分数计算函数  matchingScore
//*********************************************************************************
//linecharacter 直线近似特征向量
//pairNum 初步匹配标号阵
//lcluster_2p 直线段“丛”近似轮廓
//bCenter 轮廓重心特征向量
//threshold_bc 重心距离阈值
//alpha w计算时权重系数，新版函数该参数已失效
//基本功能：根据初步筛选的标号阵
//*********************************************************************************
vector<Vec3d> matchingScore(vector<vector<Vec2i>> lineCharacter,
                            vector<Vec4i> pairNum,
                            vector<vector<Point>> bCenter,
                            vector<vector<vector<Point>>> lcluster_2p,
                            int threshold_bc,
                            int alpha);

//旋转匹配直线标号函数    transMatchingNum
//c1Num 轮廓1的标号
//c2Num 轮廓2的标号
//c1_segmentNum 轮廓1的线段标号
//c2_segmentNum 轮廓2的线段标号
//Threshold 旋转后轮廓2重心和轮廓1重心之间的欧氏距离应小于的阈值
//基本功能：根据初步筛选结果，筛选出按当前直线段组合旋转后，满足
```

```
//1.重心距离小于一定阈值 2.夹角小于一定阈值 的线段组合，返回旋转后直线段匹配的组合标号
vector<Vec2i> transMatchingNum(vector<vector<Point>> bCenter,
                               vector<vector<vector<Point>>> lcluster_2p,
                               int c1Num,
                               int c1_segmentNum,
                               int c2Num,
                               int c2_segmentNum,
                               double Threshold);

//点旋转函数   rotatePoint
//src 待旋转点坐标
//angle 待旋转角度，旋转轴为原点
Point rotatePoint(const Point src, double angle);

Mat selectAndStiching(const Mat src, Mat PreProcessedImage);
```

main.cpp

```
#include "header.h"
#include "function.h"


int main()
{
    Mat original;
    g_srcImage = imread("../img/7_2.jpg", 1);
    const Mat src = g_srcImage;
    g_srcImage.copyTo(original);        //将原始图像保留到Original
    if (!g_srcImage.data)
    {
        printf("error,no image is found\n");
        return false;
    }
    g_srcImage = PreProcess(g_srcImage);
    Mat temp_g_srcImage;
    g_srcImage.copyTo(temp_g_srcImage);
    vector<vector<vector<Point>>>g_cluster = cClusters(g_srcImage);
    vector<vector<Point>>g_lcluster = lClusters(g_srcImage);
    lcluster_2p = lcluster2p(g_lcluster, lcluster_2p);
    vector<vector<Vec2i>>g_charalcluster = lineCharacter(g_lcluster, original);
    pairNum = vpairNum(g_charalcluster, 10, 10);
    bCenter = vBarycenter(lcluster_2p, bCenter);
    ms = matchingScore(g_charalcluster, pairNum, bCenter, lcluster_2p, 20, 5);
    //int maxnum = 0;
    //for (unsigned int i = 0; i < ms.size(); i++)
    //  if (ms[i][2] >= ms[maxnum][2])
    //      maxnum = i;
    //int maxw = ms[maxnum][2];
    //std::cout << maxnum << " " << maxw << std::endl;
    //rotateVecBcenter(lcluster_2p[0][ms[maxnum][0]], lcluster_2p[1][ms[maxnum][1]], bCenter[1]);
    //Mat output = selectAndStiching(src,
    //                               temp_g_srcImage,
    //                               lcluster_2p[ pairNum[maxnum][0] ][ms[maxnum][0]],
    //                               lcluster_2p[pairNum[maxnum][2] ][ms[maxnum][1]]);
    Mat output = selectAndStiching(src, temp_g_srcImage);
    namedWindow("output", 0);
    imshow("output", output);
    imwrite("../img/output.png", output);
    waitKey(0);
    return 0;
}
```

Functions_c.cpp

```
#include "header.h"

extern RNG g_rng;
```

```
//“从”函数_将边缘轮廓打断成【弧】线段
vector<vector<vector<Point>>> cClusters(Mat srcImage)
{
    //提取边缘轮廓
    vector<vector<Point>>contours;
    vector<Vec4i> hierarchy;
    vector<vector<Point>> contours_poly(contours.size());
    findContours(srcImage, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, Point(0, 0));
    //多边形近似提取分割点
    contours_poly.resize(contours.size());
    for (unsigned int i = 0; i < contours.size(); i++)
        approxPolyDP(Mat(contours[i]), contours_poly[i], 5, false);
    //按照多边形近似分割轮廓，将轮廓碎片封装成cluster
    vector<vector<vector<Point>>> vec_cluster;
    vector<Point> vecp;
    vector<vector<Point>> vecl;
    for (int i = 0; i < contours.size(); i++)
    {
        vecl.clear();
        unsigned int  k = 0;
        unsigned int j = 1;
        for (j; j < contours_poly[i].size(); j++)
        {
            vecp.clear();
            for (k; k < contours[i].size(); k++)
            {
                if (contours[i][k] != contours_poly[i][j])
                    vecp.push_back(contours[i][k]);
                else
                {
                    vecp.push_back(contours[i][k]);
                    break;
                }
            }
            vecl.push_back(vecp);
        }
        vec_cluster.push_back(vecl);
    }
    //作图观察
    Mat drawing = Mat::zeros(srcImage.size(), CV_8UC3);
    for (unsigned int i = 0; i < contours.size(); i++)
    {
        Scalar color = Scalar(g_rng.uniform(0, 255), g_rng.uniform(0, 255), g_rng.uniform(0, 255));
        drawContours(drawing, contours_poly, i, color, 1, 8, vector<Vec4i>(), 0, Point());
    }
    for (int i = 0; i < contours_poly.size(); i = i + 1)
    {
        for (int j = 0; j < contours_poly[i].size(); j++)
        {
            circle(drawing, contours_poly[i][j], 2, CV_RGB(0, 0, 255), 2);
        }
    }
    contours_poly.resize(contours.size());
    namedWindow("轮廓", 0);
    imshow("轮廓", drawing);
    return vec_cluster;
}

//弧线平均灰度
int cAvGscale(Mat srcImage, vector<Point> segment)
{
    int i = 0;
    int AveGscale = 0;
    for (i; i < segment.size(); i++)
    {
        AveGscale = AveGscale + srcImage.at<uchar>(segment[i]);
    }
    return AveGscale;
}

//弧线重心
Point cBaryCenter(vector<Point>segment)
```

```
{
    Moments mom = moments(Mat(segment));
    Point cBcenter = Point(mom.m10 / mom.m00, mom.m01 / mom.m00);
    return cBcenter;
}
```

Functions_l.cpp

```
#include "header.h"

extern RNG g_rng;

//“从”函数_将边缘轮廓打断成【直】线段
vector<vector<Point>> lClusters(Mat srcImage)
{
    //提取边缘轮廓
    vector<vector<Point>>contours;
    vector<Vec4i> hierarchy;
    vector<vector<Point>> contours_poly(contours.size());
    findContours(srcImage, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, Point(0, 0));
    //多边形近似提取分割点
    contours_poly.resize(contours.size());
    for (unsigned int i = 0; i < contours.size(); i++)
        approxPolyDP(Mat(contours[i]), contours_poly[i], 5, false);
    //作图观察
    Mat drawing = Mat::zeros(srcImage.size(), CV_8UC3);
    for (unsigned int i = 0; i < contours.size(); i++)
    {
        Scalar color = Scalar(g_rng.uniform(0, 255), g_rng.uniform(0, 255), g_rng.uniform(0, 255));
        drawContours(drawing, contours_poly, i, color, 1, 8, vector<Vec4i>(), 0, Point());
    }
    contours_poly.resize(contours.size());
    for (int i = 0; i < contours_poly.size(); i = i + 1)
    {
        for (int j = 0; j < contours_poly[i].size(); j++)
        {
            circle(drawing, contours_poly[i][j], 2, CV_RGB(0, 0, 255), 2);
        }
    }
    namedWindow("轮廓", WINDOW_AUTOSIZE);
    imshow("轮廓", drawing);
    return contours_poly;
}

//进一步封装多边形近似，返回以两点组成的线段为单元进行储存的向量
vector<vector<vector<Point>>> lcluster2p(vector<vector<Point>>contours_poly,
                                         vector<vector<vector<Point>>> lcluster)
{
    vector<vector<Point>> vec;
    vector<Point> vec0;
    for (unsigned int i = 0; i < contours_poly.size(); i++)
    {
        vec.clear();
        for (unsigned int j = 0; j < contours_poly[i].size() - 1; j++)
        {
            vec0.clear();
            vec0.push_back(contours_poly[i][j]);
            vec0.push_back(contours_poly[i][j + 1]);
            vec.push_back(vec0);
        }
        lcluster.push_back(vec);
    }
    return lcluster;
}
```

Functions_line.cpp

```
#include "header.h"
```

```
//线段长度函数_欧几里得距离
int lineLength(Point a, Point b)
{
    int l = sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
    return l;
}

//计算直线中点坐标
Point lineBaryCenter(Point a, Point b)
{
    Point bcenter;
    bcenter.x = (a.x + b.x) / 2;
    bcenter.y = (a.y + b.y) / 2;
    return bcenter;
}
//计算直线方向绝对theta角
double lineDirection(Point a, Point b)
{
    double del_y = abs(b.y - a.y);
    double del_x = abs(b.x - a.x);
    double theta = atan(del_y / del_x);
    return theta;
}

//小线段特征矢量表述
vector<vector<Vec2i>> lineCharacter(vector<vector<Point>> Clusters, Mat srcImage)
{
    int lineLength(Point a, Point b);
    int lineAveGscale(Mat srcImage, Point a, Point b);
    vector<Vec2i> vec;
    vector<vector<Vec2i>> lineCharacter;
    lineCharacter.clear();
    int l, a;
    for (unsigned int i = 0; i < Clusters.size(); i++)
    {
        vec.clear();
        unsigned int j = 0;
        //计算起始点和终止点【间】形成线段得长度,平均灰度
        for (j; j < Clusters[i].size() - 1; j++)
        {
            l = lineLength(Clusters[i][j], Clusters[i][j + 1]);//计算直线欧几里得长度
            a = lineAveGscale(srcImage, Clusters[i][j], Clusters[i][j + 1]);//计算直线的平均灰度
            //a = cAvGscale(srcImage, cClusters(srcImage)[i][j]);
            vec.push_back(Vec2i(l, a));//将长度和平均灰度存入向量
        }
        //  //计算【结束点和起始点构成】线段的长度，平均灰度
        //l = lineLength(Clusters[i][j], Clusters[i][0]);
        //a = lineAveGscale(srcImage, Clusters[i][j], Clusters[i][0]);
        //vec.push_back(Vec2i(l,a));
        lineCharacter.push_back(vec);
    }
    return lineCharacter;
}

//直线平均灰度函数
int lineAveGscale(Mat srcImage, Point a, Point b)
{
    //Mat dstImage;
    //dstImage.create(srcImage.size(), srcImage.type());//创建图像副本
    int nr = srcImage.rows;
    int nl = srcImage.cols*srcImage.channels();//合并三通道，row可能不连续因此不适用
    int count = 0, aveGScale = 0;
    //计算端点范围
    int max_y = (a.y > b.y) ? a.y : b.y;
    int min_y = (a.y < b.y) ? a.y : b.y;
    int max_x = (a.x > b.x) ? a.x : b.x;
    int min_x = (a.x< b.x) ? a.x : b.x;
    for (int k = 0; k < nr; k++)
    {
        const uchar* inData = srcImage.ptr<uchar>(k);
        //uchar*outData = dstImage.ptr<uchar>(k);//降阶，定义Data信息包含行数，故只需要标记列数即可
```

```
        for (int i = 0; i < nl; i++)
        {
            if (a.x == b.x)
            {
                if (k == a.x && (i <= max_y&&i >= min_y))
                {
                    aveGScale = aveGScale + inData[i];
                    count++;
                }
            }
            else if (a.y == b.y)
            {
                if (i == a.x && (k <= max_x&&k >= min_x))
                {
                    aveGScale = aveGScale + inData[i];
                    count++;
                }
            }
            //两点式描述直线方程，若点在直线段内则存入其灰度值
            else if ((i - a.y) / (b.y - a.y) == (k - a.x) / (b.x - a.x) && (i <= max_y&&i >= min_y))
            {
                aveGScale = aveGScale + inData[i];
                count++;
            }
        }
    }
    if (count == 0) count = 1;
    return aveGScale = aveGScale / count;
}
```

Functions_rotate.cpp

```
#include "header.h"

extern Mat g_srcImage;
extern vector<vector<Point>> bCenter;

//计算直线中点坐标
Point lineBaryCenter(Point a, Point b);

//点旋转函数   rotatePoint
//src 待旋转点坐标
//angle 待旋转角度，旋转轴为原点
Point rotatePoint(const Point src, double angle)
{
    Point dst;
    int x = (src.x)*cos(angle) - (src.y)*sin(angle);
    int y = (src.x)*sin(angle) + (src.y)*cos(angle);
    dst.x = x;
    dst.y = y;
    return dst;
}

//矢量旋转角计算
//a_2p 由两点组成的向量a
//b_2p 由两点组成的向量b
double vecRotateTheta(vector<Point> a_2p, vector<Point> b_2p)
{
    Point lineBaryCenter(Point a, Point b);
    //向量r1
    int r1_x = a_2p[1].x - a_2p[0].x;
    int r1_y = a_2p[1].y - a_2p[0].y;
    //向量r2
    int r2_x = b_2p[0].x - b_2p[1].x;
    int r2_y = b_2p[0].y - b_2p[1].y;
    int dot_r1r2 = r1_x*r2_x + r1_y*r2_y; //点乘
    double modr1 = sqrt(r1_x*r1_x + r1_y*r1_y);//取摸
    double modr2 = sqrt(r2_x*r2_x + r2_y*r2_y);
    double theta = acos(dot_r1r2 / (modr1*modr2));
    //叉乘
```

```
    double cross_r1r2 = r1_x*r2_y - r1_y*r2_x;
    if (cross_r1r2 >= 0) theta = -theta;
    return theta;
}
//旋转轮廓重心阵i
//Segment2p_1 线段1，两点向量表示
//Segment2p_2 线段2，两点向量表示
//bc_i 待旋转重心阵
//基本功能：根据线段1和线段2，将轮廓阵旋转至对应位置，得到旋转后轮廓重心阵坐标。注意，bCenter包含多个轮廓的重心阵，
//本函数传入参数仅为单一轮廓的重心阵。
vector<Point> rotateVecBcenter(vector<Point> Segment2p_1, vector<Point> Segment2p_2, vector<Point> bc_i)
{
    Mat drawing1 = Mat::zeros(g_srcImage.size(), CV_8UC3);
    vector<Point> bc_d;
    int endNum_a = Segment2p_1.size() - 1;
    int endNum_b = Segment2p_2.size() - 1;
    Point mid_a = lineBaryCenter(Segment2p_1[0], Segment2p_1[endNum_a]);
    Point mid_b = lineBaryCenter(Segment2p_2[0], Segment2p_2[endNum_b]);
    double dtheta = vecRotateTheta(Segment2p_1, Segment2p_2);
    for (unsigned int j = 0; j < bc_i.size(); j++)
    {
        Point temp = bc_i[j] - mid_b;
        Point bc = rotatePoint(temp, dtheta);
        bc = bc + mid_a;
        bc_d.push_back(bc);
    }
    //实验观察用，正式版可注释掉相应部分
    for (int i = 0; i < bc_d.size(); i++)
        circle(drawing1, bc_d[i], 2, CV_RGB(255, 0, 0), 2);
    for (unsigned int j = 0; j < bCenter.size(); j++)
        for (unsigned int i = 0; i < bCenter[j].size(); i++)
            circle(drawing1, bCenter[j][i], 2, CV_RGB(0, 255, 0), 2);
    namedWindow("zhongxin", 0);
    imshow("zhongxin", drawing1);
    return bc_d;
}
```

wCalculate.cpp

```
#include "header.h"

//计算matchin的w值
//vec1 轮廓i的小线段特征矢量
//vec2 轮廓j的小线段特征矢量，其中ij为旋转组合标号对
//PairNum 根据ij旋转组合标号得到的【旋转匹配直线标号】组合
double wCalculate(vector<Vec2i> vec1, vector<Vec2i> vec2, vector<Vec2i>PairNum, int a)
{
    double w1 = 0, w2 = 0, w3 = 0;
    for (unsigned int k = 0; k < PairNum.size(); k++)
    {
        for (unsigned int i = 0; i < vec1.size(); i++)
        {
            for (unsigned int j = 0; j < vec2.size(); j++)
            {
                if ((i == PairNum[k][0]) && (j == PairNum[k][1]))
                {
                    int da = abs(vec1[i][1] - vec2[j][1]);
                    if (da == 0) da = 1;
                    w1 = w1 + 0.5*(vec1[i][0] + vec2[j][0]);
                    //w2 = w2 + 0.5*(vec1[i][0] + vec2[j][0])*da;
                    w3++;
                }
            }
        }
    }
    //新版w值计算为匹配上线段组的平均长度和匹配数量的组合
    double w = sqrt(w1)*w3;
    return w;
}
```

max.cpp

```
int max(int *a)
{
    int length = sizeof(a);
    int max = a[0];
    for (int i = 0; i < length; i++)
        max = (a[i] > max) ? a[i] : max;
    return max;
}
```

matchingScore.cpp

```
#include "header.h"

//匹配分数计算函数  matchingScore
//*********************************************************************************
//linecharacter 直线近似特征向量
//pairNum 初步匹配标号阵
//lcluster_2p 直线段“丛”近似轮廓
//bCenter 轮廓重心特征向量
//threshold_bc 重心距离阈值
//alpha w计算时权重系数，新版函数该参数已失效
//基本功能：根据初步筛选的标号阵
//*********************************************************************************
vector<Vec3d> matchingScore(vector<vector<Vec2i>> lineCharacter,
                            vector<Vec4i> pairNum,
                            vector<vector<Point>> bCenter,
                            vector<vector<vector<Point>>>lcluster_2p,
                            int threshold_bc,
                            int alpha)
{
    double wCalculate(vector<Vec2i> vec1, vector<Vec2i> vec2, vector<Vec2i>PairNum, int a);
    vector<Vec2i> transMatchingNum(vector<vector<Point>> bCenter,
                                   vector<vector<vector<Point>>>lcluster_2p,
                                   int c1Num,
                                   int c1_segmentNum,
                                   int c2Num,
                                   int c2_segmentNum,
                                   double Threshold);
    vector<Vec3d> matchingScore;
    matchingScore.clear();
    for (unsigned int i = 0; i < pairNum.size(); i++)
    {
        //计算按当前组合旋转下，有多少线段匹配上了
        vector<Vec2i> Num = transMatchingNum(bCenter,
                                             lcluster_2p,
                                             pairNum[i][0],
                                             pairNum[i][1],
                                             pairNum[i][2],
                                             pairNum[i][3],
                                             threshold_bc);
        //根据匹配上的线段标号，计算该种旋转组合的w值，结果顺序存在w向量里
        double w = 0;
        if (Num.size() != 0)
            w = wCalculate(lineCharacter[pairNum[i][0]], lineCharacter[pairNum[i][2]], Num, alpha);
        matchingScore.push_back(Vec3d(pairNum[i][1], pairNum[i][3], w));
    }
    return matchingScore;
}
```

PreProcess.cpp

```
#include "header.h"

extern Mat g_srcImage;
extern int g_nThresh;
```

```
extern int g_nThresh_Max;

//预处理函数
Mat PreProcess(Mat srcImage)
{
    Mat grayImage;
    Mat binaryImage;
    Mat dstImage;
    cvtColor(g_srcImage, grayImage, COLOR_BGR2GRAY);
    threshold(grayImage, binaryImage, 235, 255, CV_THRESH_BINARY);
    medianBlur(binaryImage, binaryImage, 9);
    Canny(binaryImage, dstImage, g_nThresh, g_nThresh_Max);
    return dstImage;
}
```

selectAndStiching.cpp

```
#include "header.h"


extern vector<Vec4i> pairNum;
extern vector<Vec3d> ms;
extern vector<vector<vector<Point>>>lcluster_2p;

//重心
Point cBaryCenter(vector<Point>segment);
//计算直线中点坐标
Point lineBaryCenter(Point a, Point b);
//矢量旋转角计算
//a_2p 由两点组成的向量a
//b_2p 由两点组成的向量b
double vecRotateTheta(vector<Point> a_2p, vector<Point> b_2p);

Mat selectAndStiching(const Mat src, Mat PreProcessedImage)
{
    vector<vector<Point>>contours;
    vector<Vec4i> hierarchy;
    Rect ccomp;
    findContours(PreProcessedImage, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, Point(0, 0));

    Mat* pieceMask = new Mat[contours.size()];
    Mat* piece = new Mat[contours.size()];
    Point seedPoint;
    //提取分离的单独碎片准备旋转合成
    for (int i = 0; i < contours.size(); i++)
    {
        pieceMask[i].create(src.size(), src.type());
        drawContours(pieceMask[i], contours, i, Scalar(255, 255, 255), 1, 8, vector<Vec4i>(), 0, Point());
        seedPoint = cBaryCenter(contours[i]);
        floodFill(pieceMask[i], seedPoint, Scalar(255, 255, 255), &ccomp, Scalar(0, 0, 0), Scalar(50, 50, 50));
        floodFill(pieceMask[i], Point(0, 0), Scalar(0, 0, 0), &ccomp, Scalar(50, 50, 50), Scalar(0, 0, 0));
        src.copyTo(piece[i], pieceMask[i]);
    }

    int* maxnum = new int[(contours.size()*(contours.size()-1))/2];
    int index = 0;
    for (int k = 0;k<contours.size() - 1;k++)
    {
        for (int j = k+1 ; j < contours.size(); j++,index++)
        {
            maxnum[index] = 0;
            for (unsigned int i = 0; i < ms.size(); i++)
            {
                if (ms[i][2] > 30 && ms[i][2] >= ms[maxnum[index]][2] && pairNum[i][0] == k && pairNum[i][2] == j)
                    maxnum[index] = i;
            }
        }
    }
    int maxnummax = 0;
    for (unsigned int i = 0; i < (contours.size()*(contours.size() - 1))/2; i++)
```

```
        if (ms[maxnum[i]][2] >= ms[maxnum[maxnummax]][2])
            maxnummax = i;

    Mat* outputAll = new Mat[(contours.size()*(contours.size() - 1)) / 2];
    for (int i = 0; i < (contours.size()*(contours.size() - 1)) / 2; i++)
    {

        vector<Point> Segment2p_1 = lcluster_2p[pairNum[maxnum[i]][0]][ms[maxnum[i]][0]];
        vector<Point> Segment2p_2 = lcluster_2p[pairNum[maxnum[i]][2]][ms[maxnum[i]][1]];
        /*-----------------平移旋转部分-----------------*/
        Mat temppiece1, temppiece2;
        piece[pairNum[maxnum[i]][2]].copyTo(temppiece1);
        piece[pairNum[maxnum[i]][0]].copyTo(temppiece2);
        IplImage* piece1 = &IplImage(temppiece1);
        Point centerP0 = lineBaryCenter(Segment2p_1[0], Segment2p_1[1]),
            centerP1 = lineBaryCenter(Segment2p_2[0], Segment2p_2[1]);
        //计算二维旋转的仿射变换矩阵  P1->P0
        float m1[6] = { 1,0,(float)(centerP0.x - centerP1.x),0,1,(float)(centerP0.y - centerP1.y) };
        CvMat M1 = cvMat(2, 3, CV_32F, m1);
        //平移图像，并用黑色填充其余值
        cvWarpAffine(piece1, piece1, &M1, CV_INTER_LINEAR + CV_WARP_FILL_OUTLIERS, cvScalarAll(0));

        //旋转中心
        CvPoint2D32f center;
        center.x = float(centerP0.x);
        center.y = float(centerP0.y);
        //旋转角度
        double dtheta = vecRotateTheta(Segment2p_1, Segment2p_2);
        float m2[6];
        CvMat M2 = cvMat(2, 3, CV_32F, m2);
        cv2DRotationMatrix(center, dtheta, 1, &M2);
        //旋转图像，并用黑色填充其余值
        cvWarpAffine(piece1, piece1, &M2, CV_INTER_LINEAR + CV_WARP_FILL_OUTLIERS, cvScalarAll(0));
        /*----------------------------------------*/

        outputAll[i].create(src.size(), src.type());
        addWeighted(temppiece2, 1, temppiece1, 1, 0, outputAll[i]);
    }
    Mat output = outputAll[maxnummax];
    return output;
}
```

transMatchingNum.cpp

```
#include "header.h"

int lineLength(Point a, Point b);

//旋转匹配直线标号函数    transMatchingNum
//c1Num 轮廓1的标号
//c2Num 轮廓2的标号
//c1_segmentNum 轮廓1的线段标号
//c2_segmentNum 轮廓2的线段标号
//Threshold 旋转后轮廓2重心和轮廓1重心之间的欧氏距离应小于的阈值
//基本功能：根据初步筛选结果，筛选出按当前直线段组合旋转后，
//满足1.重心距离小于一定阈值 2.夹角小于一定阈值 的线段组合，返回旋转后直线段匹配的组合标号
vector<Vec2i> transMatchingNum(vector<vector<Point>> bCenter,
                               vector<vector<vector<Point>>>lcluster_2p,
                               int c1Num,
                               int c1_segmentNum,
                               int c2Num,
                               int c2_segmentNum,
                               double Threshold)
{
    vector<Point> rotateVecBcenter(vector<Point> Segment2p_1, vector<Point> Segment2p_2, vector<Point> bc_i);
    vector<Vec2i> Num;
    vector<Point> bCenter_c2Num = rotateVecBcenter(lcluster_2p[c1Num][c1_segmentNum],
                                                   lcluster_2p[c2Num][c2_segmentNum],
                                                   bCenter[c2Num]);
    vector<Point> rotateVecBcenter(vector<Point> Segment2p_1, vector<Point> Segment2p_2, vector<Point> bc_i);
```

```
    double lineDirection(Point a, Point b);
    double Thresholdtheta = CV_PI / 36;
    for (unsigned int i = 0; i < bCenter[c1Num].size(); i++)
    {
        for (unsigned int j = 0; j < bCenter_c2Num.size(); j++)
        {
            int bc_dist = lineLength(bCenter[c1Num][i], bCenter_c2Num[j]);
            if (lineLength(bCenter[c1Num][i], bCenter_c2Num[j]) < Threshold)
            {
                vector<Point> temp;
                temp = rotateVecBcenter(lcluster_2p[c1Num][c1_segmentNum],
                                        lcluster_2p[c2Num][c2_segmentNum],
                                        lcluster_2p[c2Num][j]);
                double theta1 = lineDirection(lcluster_2p[c1Num][i][0], lcluster_2p[c1Num][i][1]);
                double theta2 = lineDirection(temp[0], temp[1]);
                double diff_theta = abs(theta1 - theta2);
                if (diff_theta<Thresholdtheta)
                    Num.push_back(Vec2i(i, j));
            }
        }
    }
    return Num;
}
```

vBarycenter.cpp

```
#include "header.h"

Point lineBaryCenter(Point a, Point b);

//计算轮廓重心阵
//lcluster_2p 按两点组合存放的轮廓描绘子
//bCenter 生成重心阵存放的向量
vector<vector<Point>> vBarycenter(vector<vector<vector<Point>>>lcluster_2p, vector<vector<Point>> bCenter)
{
    vector<Point> vec;
    for (int i = 0; i < lcluster_2p.size(); i++)
    {
        vec.clear();
        for (int j = 0; j < lcluster_2p[i].size(); j++)
        {
            Point bc = lineBaryCenter(lcluster_2p[i][j][0], lcluster_2p[i][j][1]);
            vec.push_back(bc);
        }
        bCenter.push_back(vec);
    }
    return bCenter;
}
```

vpairNum.cpp

```
#include "header.h"

//旋转组合标号阵（初步筛选标号阵）
//标号阵—匹配轮廓1中segment_i和轮廓2中segment_j的标号，组合成初步筛选的标号阵
vector<Vec4i> vpairNum(vector<vector<Vec2i>> lineCharacter, int dlThreshold, int daThreshold)
{
    vector<Vec4i> pairNum;
    for (int i = 0; i < lineCharacter.size() - 1; i++)
    {
        for (int m = i + 1; m < lineCharacter.size(); m++)
        {
            for (int j = 0; j < lineCharacter[i].size(); j++)
            {
                for (int n = 0; n < lineCharacter[m].size(); n++)
                {
                    int diff_a = abs(lineCharacter[m][n][1] - lineCharacter[i][j][1]);
                    int diff_l = abs(lineCharacter[m][n][0] - lineCharacter[i][j][0]);
```

```
                    if ((diff_a <= daThreshold) && (diff_l <= dlThreshold))
                        pairNum.push_back(Vec4i(i, j, m, n));
                }
            }
        }
    }
    return pairNum;
}
```