# Opencv2.4.9源码分析——Stitching（一）

相机镜头所呈现出的景物要比人类的视觉系统所看到的景物要狭小得多，因此一幅图像不可能捕获到我们所看到的整个景物。全景图像拼接给出了这个问题的解决办法，它是把图像间重叠部分拿出来拼接起来，从而得到一幅更大的图像。这种算法也可以用于把一幅图像插入到另一幅图像中。

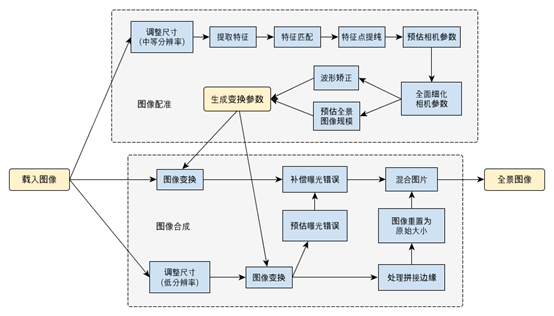


图1 图像拼接执行过程及方法

要想完成图像拼接，所要用到的算法较多，Opencv把这些算法用一张图呈现了处理，如图1所示。下面我们就详细图像拼接算法。

1. 计算特征点
2. 1.1 原理

图像拼接的第一步是计算图像的特征，并得到它们的描述符。好的特征检测的算法有SIFT和SURF，但它们都受到专利的保护，使用上有一定的限制。而ORB算法是一种很好的替代方法，它是利用FAST检测特征，并生成BRIEF描述符。

关于局部特征检测方面的内容，我前面的文章中有一些算法的介绍，在这里就不再做过多的讲解。

1.2 源码

Opencv中应用的是SURF算法和ORB算法，详细的内容为：

class detail::FeaturesFinder表示寻找图像特征的类：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78798687) [copy](http://blog.csdn.net/zhaocj/article/details/78798687)

1. **class** CV\_EXPORTS FeaturesFinder
2. {
3. **public**:
4. **virtual** ~FeaturesFinder() {}
5. //寻找给定图像的特征
6. **void** operator ()(**const** Mat &image, ImageFeatures &features);
7. **void** operator ()(**const** Mat &image, ImageFeatures &features, **const** std::vector<cv::Rect> &rois);
8. **virtual** **void** collectGarbage() {}    //释放已被分配、但还没有被使用的内存
10. **protected**:
11. //虚函数，根据用户所选取的特征类别，调用不同子类的find函数，目前只实现了SURF特征和ORB特征
12. **virtual** **void** find(**const** Mat &image, ImageFeatures &features) = 0;
13. };

在FeaturesFinder类中，重载( )运算符主要的任务是调用find函数来检测图像特征。而find是虚函数，具体是执行FeaturesFinder类的两个子类——SurfFeaturesFinder和OrbFeaturesFinder中的find函数。下面我们就来介绍这两个子类，它们分别表示SURF特征和ORB特征。

SurfFeaturesFinder类的构造函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78798687) [copy](http://blog.csdn.net/zhaocj/article/details/78798687)

1. SurfFeaturesFinder::SurfFeaturesFinder(**double** hess\_thresh, **int** num\_octaves, **int** num\_layers,
2. **int** num\_octaves\_descr, **int** num\_layers\_descr)
3. //hess\_thress表示Hessian矩阵行列式响应值的阈值，默认值为300
4. //num\_octaves表示特征点所用到的图像堆的组数，默认值为3
5. //num\_layers表示特征点所用到的图像堆中每组中的中间层数，默认值为4
6. //num\_octaves\_descr表示描述符所用到的图像堆的组数，默认值为3
7. //num\_layers\_descr表示描述符所用到的图像堆中每组中的中间层数，默认值为4
8. {
9. //如果特征点和描述符所用到的组数和层数都相同，则只需一个Feature2D数据类型来表示特征点和描述符即可，默认就是这种情况
10. **if** (num\_octaves\_descr == num\_octaves && num\_layers\_descr == num\_layers)
11. {
12. //surf为Feature2D数据类型的指针变量，它被创建为SURF特征点
13. surf = Algorithm::create<Feature2D>("Feature2D.SURF");
14. **if**( surf.empty() )    //如果surf没有创建成功，则报错
15. CV\_Error( CV\_StsNotImplemented, "OpenCV was built without SURF support" );
16. surf->set("hessianThreshold", hess\_thresh);    //赋阈值
17. surf->set("nOctaves", num\_octaves);    //赋组数
18. surf->set("nOctaveLayers", num\_layers);    //赋层数
19. }
20. //如果特征点和描述符所用到的组数或层数不相同，则各需要一个数据类型来分别表示特征点和描述符，即特征点用FeatureDetector数据类型，描述符用DescriptorExtractor数据类型
21. **else**
22. {
23. //分别创建SURF特征点detector\_和SURF描述符extractor\_
24. detector\_ = Algorithm::create<FeatureDetector>("Feature2D.SURF");
25. extractor\_ = Algorithm::create<DescriptorExtractor>("Feature2D.SURF");
26. //如果创建失败，则报错
27. **if**( detector\_.empty() || extractor\_.empty() )
28. CV\_Error( CV\_StsNotImplemented, "OpenCV was built without SURF support" );
30. detector\_->set("hessianThreshold", hess\_thresh);    //为特征点赋阈值
31. detector\_->set("nOctaves", num\_octaves);    //为特征点赋组数
32. detector\_->set("nOctaveLayers", num\_layers);    //为特征点赋层数
34. extractor\_->set("nOctaves", num\_octaves\_descr);    //为描述符赋组数
35. extractor\_->set("nOctaveLayers", num\_layers\_descr);    //为描述符赋层数
36. }
37. }

检测SURF特征：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78798687) [copy](http://blog.csdn.net/zhaocj/article/details/78798687)

1. **void** SurfFeaturesFinder::find(**const** Mat &image, ImageFeatures &features)
2. //image表示待处理的输入图像
3. //features表示输出的SURF特征
4. {
5. Mat gray\_image;    //表示待处理的图像
6. //输入图像image必须为8位字符型数据类型，可以是灰度图像，也可以是彩色图像
7. CV\_Assert((image.type() == CV\_8UC3) || (image.type() == CV\_8UC1));
8. **if**(image.type() == CV\_8UC3)    //如果输入图像是彩色图像
9. {
10. cvtColor(image, gray\_image, CV\_BGR2GRAY);    //转换为灰度图像
11. }
12. **else**    //如果输入图像是灰度图像
13. {
14. gray\_image = image;    //赋值
15. }
16. //如果surf为0，说明在SurfFeaturesFinder类的构造函数内，没有创建该变量，也就意味着要分别使用detector\_变量和extractor\_变量分别表示SURF特征点和SURF描述符
17. **if** (surf == 0)
18. {
19. detector\_->detect(gray\_image, features.keypoints);    //得到SURF特征点
20. //得到SURF描述符
21. extractor\_->compute(gray\_image, features.keypoints, features.descriptors);
22. }
23. //在surf不为0的情况下，说明在SurfFeaturesFinder类的构造函数内，已创建该变量，也就意味着只需要用surf变量来表示SURF特征点和SURF描述符即可
24. **else**
25. {
26. Mat descriptors;    //表示描述符
27. //得到特征点和描述符
28. (\*surf)(gray\_image, Mat(), features.keypoints, descriptors);
29. //变换描述符的数据形式，并赋值给features
30. features.descriptors = descriptors.reshape(1, (**int**)features.keypoints.size());
31. }
32. }

OrbFeaturesFinder类的构造函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78798687) [copy](http://blog.csdn.net/zhaocj/article/details/78798687)

1. OrbFeaturesFinder::OrbFeaturesFinder(Size \_grid\_size, **int** n\_features, **float** scaleFactor, **int** nlevels)
2. //\_grid\_size表示根据该值的大小，把输入图像分割成若干个小的区域，分别得到每个小区域的ORB特征，\_grid\_size默认值为Size(3,1)，即把输入图像分割成3个小区域，每个小区域的宽是原输入图像宽的三分之一，小区域的高与原图像相同；而如果不对输入图像进行分割，则\_grid\_size应该为Size(1,1)
3. //n\_features表示最终得到的特征点的数量，默认值为1500
4. //scaleFactor表示尺度金字塔的各层间的取样率，必须大于1，默认值为1.3
5. //nlevels表示尺度金字塔的层数，默认值为5
6. {
7. grid\_size = \_grid\_size;    //赋值
8. //实例化ORB类，它表示分割后每个小区域的ORB特征，并且每个小区域的最多的特征点数量为n\_features×(90+小区域的数量)/(100×小区域的数量)；而如果不对输入图像进行分割，则该类就表示原输入图像的ORB特征，特征点的数量就等于n\_features
9. orb = **new** ORB(n\_features \* (99 + grid\_size.area())/100/grid\_size.area(), scaleFactor, nlevels);
10. }

检测ORB特征：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78798687) [copy](http://blog.csdn.net/zhaocj/article/details/78798687)

1. **void** OrbFeaturesFinder::find(**const** Mat &image, ImageFeatures &features)
2. //image表示待处理的输入图像
3. //features表示输出的ORB特征
4. {
5. Mat gray\_image;    //表示待处理的图像
6. //输入图像image必须为8位字符型数据类型，可以是灰度图像，也可以是三通道或四通道的彩色图像
7. CV\_Assert((image.type() == CV\_8UC3) || (image.type() == CV\_8UC4) || (image.type() == CV\_8UC1));
8. //把彩色图像转换为灰度图像
9. **if** (image.type() == CV\_8UC3) {
10. cvtColor(image, gray\_image, CV\_BGR2GRAY);
11. } **else** **if** (image.type() == CV\_8UC4) {
12. cvtColor(image, gray\_image, CV\_BGRA2GRAY);
13. } **else** **if** (image.type() == CV\_8UC1) {
14. gray\_image=image;
15. } **else** {
16. CV\_Error(CV\_StsUnsupportedFormat, "");
17. }
19. **if** (grid\_size.area() == 1)    //表示没有对输入图像进行分割
20. //直接对原输入图像进行ORB特征检测，得到特征点和描述符
21. (\*orb)(gray\_image, Mat(), features.keypoints, features.descriptors);
22. **else**    //表示对输入图像进行了分割
23. {
24. features.keypoints.clear();    //清除变量
25. features.descriptors.release();    //清除变量
27. std::vector<KeyPoint> points;    //表示每个小区域的特征点
28. Mat descriptors;    //表示每个小区域的描述符
29. //遍历所有分割小区域
30. **for** (**int** r = 0; r < grid\_size.height; ++r)
31. **for** (**int** c = 0; c < grid\_size.width; ++c)
32. {
33. //(xl, yl)表示当前小区域的左上角在原输入图像的坐标
34. **int** xl = c \* gray\_image.cols / grid\_size.width;
35. **int** yl = r \* gray\_image.rows / grid\_size.height;
36. //(xr, yr)表示当前小区域的右下角在原输入图像的坐标
37. **int** xr = (c+1) \* gray\_image.cols / grid\_size.width;
38. **int** yr = (r+1) \* gray\_image.rows / grid\_size.height;
40. // LOGLN("OrbFeaturesFinder::find: gray\_image.empty=" << (gray\_image.empty()?"true":"false") << ", "
41. //     << " gray\_image.size()=(" << gray\_image.size().width << "x" << gray\_image.size().height << "), "
42. //     << " yl=" << yl << ", yr=" << yr << ", "
43. //     << " xl=" << xl << ", xr=" << xr << ", gray\_image.data=" << ((size\_t)gray\_image.data) << ", "
44. //     << "gray\_image.dims=" << gray\_image.dims << "\n");
45. //得到当前小区域的图像
46. Mat gray\_image\_part=gray\_image(Range(yl, yr), Range(xl, xr));
47. // LOGLN("OrbFeaturesFinder::find: gray\_image\_part.empty=" << (gray\_image\_part.empty()?"true":"false") << ", "
48. //     << " gray\_image\_part.size()=(" << gray\_image\_part.size().width << "x" << gray\_image\_part.size().height << "), "
49. //     << " gray\_image\_part.dims=" << gray\_image\_part.dims << ", "
50. //     << " gray\_image\_part.data=" << ((size\_t)gray\_image\_part.data) << "\n");
51. //对当前小区域进行ORB特征检测
52. (\*orb)(gray\_image\_part, Mat(), points, descriptors);
53. //为当前小区域所得到的特征预留空间
54. features.keypoints.reserve(features.keypoints.size() + points.size());
55. //遍历当前小区域的所有特征点，得到这些特征点在原输入图像的坐标，并把它们放入feature中
56. **for** (std::vector<KeyPoint>::iterator kp = points.begin(); kp != points.end(); ++kp)
57. {
58. kp->pt.x += xl;
59. kp->pt.y += yl;
60. features.keypoints.push\_back(\*kp);
61. }
62. //把当前小区域的所有描述符放入feature中
63. features.descriptors.push\_back(descriptors);
64. }
65. }
66. }

1.3 应用

下面我们用拼接算法中的特征点检测方法进行特征点检测：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78798687) [copy](http://blog.csdn.net/zhaocj/article/details/78798687)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. **using** **namespace** cv;
23. **using** **namespace** std;
24. **using** **namespace** detail;
26. **int** main(**int** argc, **char**\*\* argv)
27. {
28. Mat img = imread("1.jpg");    //读入图像
30. Ptr<FeaturesFinder> finder;    //定义FeaturesFinder类
32. finder = **new** SurfFeaturesFinder();    //应用SURF方法
33. //finder = new OrbFeaturesFinder();    //应用ORB方法
35. ImageFeatures features;    //表示特征
37. (\*finder)(img, features);    //特征检测
39. Mat output\_img;
40. //调用drawKeypoints函数绘制特征
41. drawKeypoints(img, features.keypoints, output\_img, Scalar::all(-1));
43. namedWindow("features");
44. imshow("features", output\_img);
46. waitKey(0);
48. **return** 0;
49. }

输出的结果为：



图2 特征检测结果

# Opencv2.4.9源码分析——Stitching（二）

2、计算单应矩阵

2.1 原理

在得到了图像特征点以后，我们就可以根据这些特征点，实现图像匹配，即得到重叠区域。而要把多幅图像拼接成一幅图像，就需要以某幅图像为基准，把其他图像映射到该图像所在的平面。映射的关键所在就是根据重叠区域的特征点计算图像间的单应矩阵。

我们可以通过最近邻算法（K-NN，这里的K表示最近邻特征点的数量）得到与图像A中的某一特征点a最相似的其他任意一幅图像B的K个特征点，对这K个特征点进行距离比较来最终判断是否找到了a的最相似特征点，即配对成功与否。例如我们这里选择K为2，即2-NN算法，图像B中的与a最相似的两个特征点为b和c，设Lab和Lac分别表示a与b和c的距离（距离是通过特征点描述符的比较实现的），并且Lab＜Lac，如果满足下式，则我们认为a的匹配特征点为b：

http://img.blog.csdn.net/20171214101856814（1）

式中，ρ表示匹配阈值。

那么如何得到这K个特征点呢？最简单、也是最粗暴的方法就是计算a与图像B中的所有特征点的距离，这种方法适用于特征点不多的情况。当特征点较多，并且需要匹配的图像也较多时，应用K-D树（K-维树）的算法会更有效。

K-D树是一棵二叉树，树中存储的是一些K维数据，这正好对应于特征点的K维描述符。K-D树本质上是对该K维数据集合的一种K维空间的划分，即树中的每个叶节点都对应于一个K维超矩形区域。当K-D树构建好后，可以很迅速的得到某一数据所属的超矩形区域。所以K-D树非常适合范围搜索和最近邻搜索，但它不适合维数太多的数据搜索。另外，需要注意的是K-NN中的K和K-D树中的K表示的含义是不同的。

首先我们需要利用已知数据构建K-D树（这里的数据指的是图像B中的所有特征点，而特征点是用描述符表示的）。要想构建K-D树，需要解决两个问题：第一是选择哪个维度进行分割；第二是如何把数据分割为左分支和右分支。对于第一个问题，通常的做法是交替循环的依次选择所有的维度，例如在3-D树中，“父节点”用的x维，则“子节点”用y维，“孙节点”用z维，“重孙节点”又回到x维，“重重孙节点”用y维，以此类推。对于第二问题，一般的做法是选择该维度的中值进行分割，小于该值的数据归为左分支，大于该值的数据归为右分支。以上方法可以保证K-D树是平衡的，即每个叶节点与根节点的距离基本相同。

当K-D树构建好了以后，我们就可以搜索待查询数据（这里指的是图像A中的特征点a）的最近邻数据，它的过程是：从根节点出发，依据分叉规则（即选择分割维度和比较中值），沿着搜索路径达到最佳叶节点；然后再从该叶节点出发，回溯搜索路径，找到是否有比最佳叶节点更好的节点，用以替代它。从概念上来说，回溯的过程就是以待查询点为中心，以待查询点到最佳叶节点的距离为半径，画一个超球体，如果该超球体与分割维度所在的超平面相交，则需要搜索该超平面的另一侧（即另一个分支），如果不相交，则无需再搜索另一侧。

特征点的匹配点对得到后，我们就可以利用这些匹配点对评估计算图像间的单应矩阵**H**。

如果平面到平面之间的映射关系是投影映射的话，那么一定存在一个非奇异的3×3的矩阵**H**，当p表示一个平面的任意一点时，那么该点映射到另一平面的点就为**H**p。当相机围绕一点进行旋转而得到不同的两幅图像时（如图3所示），这两幅图像之间就是投影映射，它们之间的关系就可以用矩阵**H**来进行描述，即图像A中的点通过矩阵H就可以映射到图像B中，从而实现两幅图像的拼接。

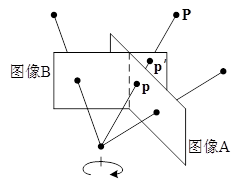


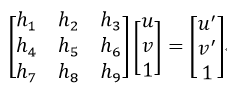
图3 通过旋转相机，得到不同的图像

需要指出的是，当矩阵**H**乘以任意一个不为零的常数时，并不改变两个平面之间的投影映射关系，因此**H**被认为是只有8个自由度的单应矩阵。

设图像A中的某一点的齐次坐标**p**为(u, v,1)，该点在图像B中的匹配点的齐次坐标**p’**为(u’, v’,1)，则：

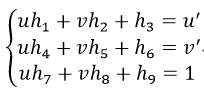
http://img.blog.csdn.net/20171214101913040（2）

即

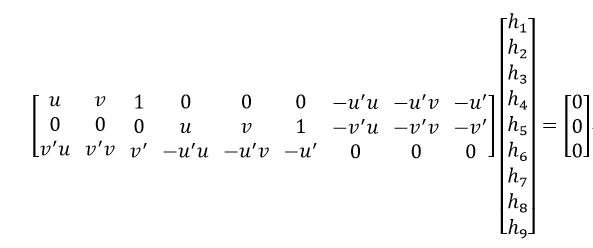
（3）

图像坐标(x, y)所对应的齐次坐标为(x, y, 1)，而齐次坐标(x, y, z)所对应的图像坐标为(x/z, y/z)。

我们把式3展开，则：

（4）

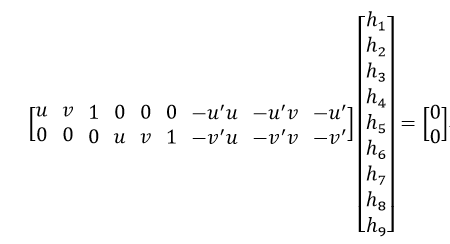
把式4写成矩阵的形式：

（5）

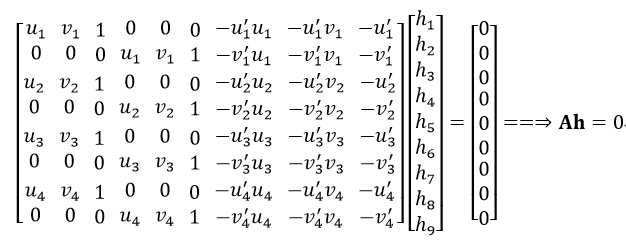
我们观察式5等号左侧的第一个矩阵会发现，第三行是前两行的线性组合，即

http://img.blog.csdn.net/20171214101953711（6）

因此式5应改为

（7）

一对匹配点对可以列写两个方程，那么要想求解只有8个自由度的矩阵**H**只需要四个匹配点对即可，但要求这四对点每三个都不能共线：

（8）

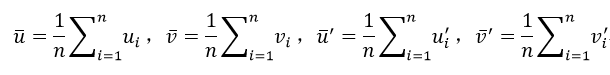
式中，**h**是矩阵**H**的向量形式。**Ah**＝0被称为齐次线性最小二乘问题，因此求解单应矩阵**H**就转换为齐次线性最小二乘问题。该方法也称为直接线性变换（DLT）。

式8表示的是一个齐次线性方程组，式中**A**为矩阵，**h**为相量，我们需要求非零的**h**。而**h**属于**A**的零空间，有时也称为**A**的零（右）相量，因此**h**可以由**A**的奇异值分解（SVD）得到，也就是说并不需要解方程组就可以得到**h**。**A**的奇异值分解形式为：

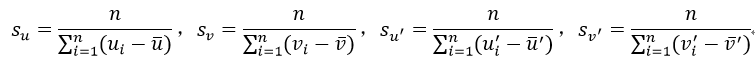
http://img.blog.csdn.net/20171214102021806（9）

则**h**为V的最后一列，再把**h**重新整理，就可得到**H**。

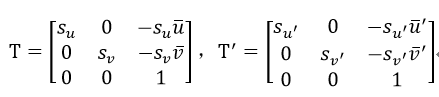
通过式9得到**h**的方法非常依赖于图像的坐标原点及尺度，这会使算法不稳定，产生数值误差。但我们可以通过归一化的方法使数值收敛于正确的结果。归一化的方法是首先平移图像坐标，使图像的坐标原点为匹配点对的重心；然后是改变尺度，使匹配点到平移后的坐标原点的距离为1。匹配点对所在的两幅图像，以及它们的横、纵坐标都必须分别独立的进行上述归一化处理。归一化的具体计算过程为：

（10）

式中，n表示匹配点对，这里n为4，该式得到了坐标平均值，即坐标平移量。

（11）

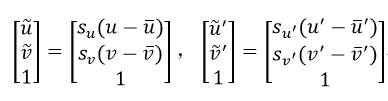
式11得到了坐标尺度，即匹配点到原点的距离为1，如果想使距离为 http://img.blog.csdn.net/20171214100240364（许多文献上，此处的值都是http://img.blog.csdn.net/20171214100240364 ，Opencv用的是1），则只需再乘以 http://img.blog.csdn.net/20171214100240364即可。由式10和式11分别构成两个坐标变换矩阵：

（12）

则两幅图像的坐标分别变为：

http://img.blog.csdn.net/20171214102100794（13）

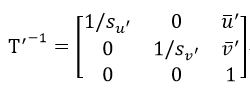
即

（14）

这时，我们就需要用坐标变换后的4个匹配点对计算单应矩阵，得到http://img.blog.csdn.net/20171214100639464 ，它与真实坐标下的**H**的关系为：

http://img.blog.csdn.net/20171214102118173（15）

式中的T’-1很容易由式12中的T’得到：

（16）

下面我们给出归一化直接线性变换的步骤：

●由式10和式11分别计算平移量和尺度；

●由式12计算坐标变换矩阵；

●由式13计算坐标变换后的特征点坐标值，并代入式8得到http://img.blog.csdn.net/20171214100917565，再经过奇异值分解（式9）得到http://img.blog.csdn.net/20171214100639464 ；

●由式15得到最终的单应矩阵**H**。

以上是由4个匹配点对计算出单应矩阵**H**，为了增加鲁棒性，可以用更多的匹配点对计算**H**。此时式8就是一个超定方程组，因为**A**是2n×9的矩阵，而n>4，所以2n>9。当匹配点对n的数量很大时，直接用SVD的方法就不是很方便，这时我们就可以直接应用最小二乘法求解。因为**Ah**＝0，它的误差平方函数f(**h**)可以写为：

http://img.blog.csdn.net/20171214102141218（17）

很显然，我们应该使f(**h**)最小，此时的**h**即为求解。因此我们对f(**h**)求导，并使导数为0，则

http://img.blog.csdn.net/20171214102149615（18）

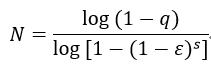
**A**T**A**是对称的9×9的方阵，我们对**A**T**A**进行特征值分解，则**h**应该是**A**T**A**的特征值为0所对应的特征向量，当然因为存在误差，有时会出现没有一个特征值为0的情况，这时**h**等于最接近0的特征值所对应的特征向量。可以很容易的证明，对**A**进行奇异值分解与对**A**T**A**进行特征值分解得到的**h**是完全相等。因此特征值分解的方法也适用于n＝4的情况。

尽管使用了更多的匹配点对，但由于误差的存在，匹配点对的位置可能会出现偏差，即有好的匹配点对（称之为内点），也有不好的匹配点对（称之为外点）。如果用外点计算，则**H**肯定是不正确的，所以我们应该选择内点。那么如何区分内点和外点呢？通常的做法是应用RANSAC（随机抽样一致性，**RAN**dom**SA**mple **C**onsensus）算法。

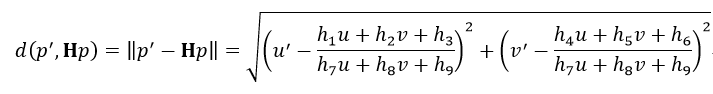
RANSAC算法的核心思想是从样本集中抽取多少次才能保证至少有一次不是外点的概率为q。则据此，可以得到下列关系式：

http://img.blog.csdn.net/20171214102202116（19）

式中，N表示抽取的次数，ε表示样本集中外点的比例，s表示每一次抽取样本的数量，在这里s为4，因为求单应矩阵**H**至少需要4个匹配点对，因此我们需要每一次抽取4个匹配点对。式19又可以表示为：

（20）

式中，q是需要事先确定的参数，q越大，得到的内点就越可信。ε由重映射误差的方法得到，即当得到了单应矩阵**H**后，可以把图像A中的特征点p重映射到图像B中，计算映射后的点与它的匹配点对的特征点p’之间的几何距离：

（21）

而重映射误差为d2(p’,**H**p)。如果该距离小于事先确定好的阈值η，则认为这对匹配点对是内点，否则是外点。计算所有的匹配点对，这样就可以统计出内点和外点的数量，从而得到ε，进而由式20得到N。

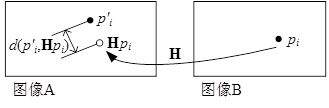


图4 重映射误差

下面我们给出RANSAC计算单应矩阵**H**的步骤：

●给出参数q和η，以及迭代次数N；

●进入迭代，一共需迭代N次：

▲ 随机抽取4个匹配点对，要求任意3个不共线；

▲ 由归一化直接线性变换计算单应矩阵**H**；

▲ 由重映射误差方法（式21）计算内点的数量，并得到参数ε；

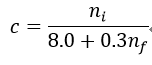
▲ 由式20更新最大迭代次数N；

●在迭代循环内已得到最大的内点数量，由这些内点计算单应矩阵，得到最终的**H**。

另外，我们还需要用下列的关系式来说明内点数ni和匹配点对数量nf之间的关系：

http://img.blog.csdn.net/20171214102255498（22）

如果满足式22，则说明图像间有很好的匹配。或者我们用一个匹配置信度c来衡量：

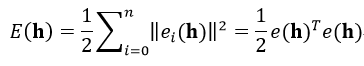
（23）

c越大越好，当然如果c大于3（3为实验数据），则认为这两幅图像十分接近，可以被看成是一幅图像了；而如果c小于一定的值，则认为这两幅图像没有重叠的区域，所以就无法拼接在一起。

应用RANSAC算法的主要目的是得到内点，而用全体内点经过特征值分解计算得到的**H**误差较大，可以说比较粗糙，因此我们还需要用更加“精细”的方法得到精确的**H**。由式21可知，**H**应该使重映射误差（即匹配点对之间的几何距离）达到最小。因此这是一个参数优化估计求极值的问题。基于LM算法（Levenberg-Marquardt）的优化是目前应用较为广泛的一种无条件约束优化方法，该算法是在逼近某个极小点时平方收敛，因此它是专门用于误差平方和最小化的方法。该算法的特点是，它是高斯牛顿法和梯度下降法的结合，既有高斯牛顿法的快速收敛性，也有梯度下降法的全局搜索特性。

下面我们就给出应用LM算法计算**H**的方法。

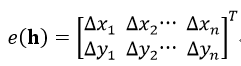
首先给出误差指标函数，即所有内点的重映射误差的平方和：

（24）

式中，n表示内点的总数，**h**是单应矩阵**H**的向量形式，第i对匹配点对的重映射误差ei为：

http://img.blog.csdn.net/20171214102325079（25）

而e(**h**)则为：

（26）

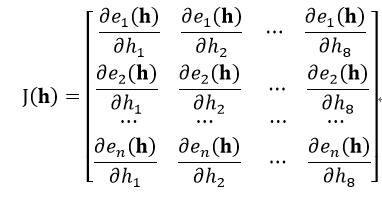
LM算法用于迭代计算**h**的公式为：

http://img.blog.csdn.net/20171214102343426（27）

式中，**h**的上标表示迭代的次数，步长∆**h**为：

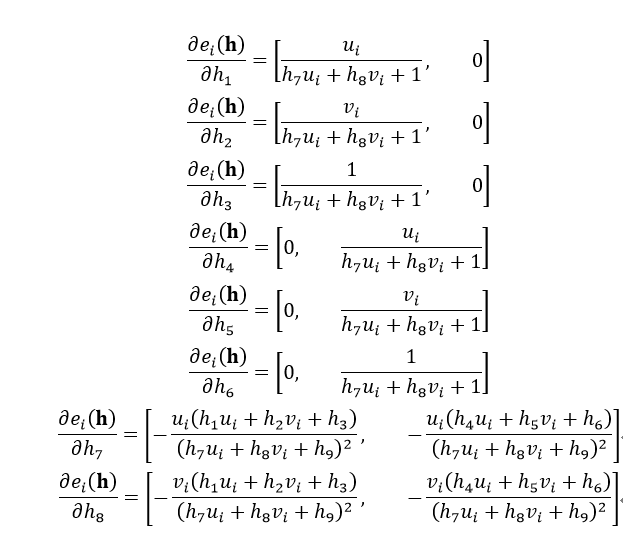
http://img.blog.csdn.net/20171214102353218（28）

式中，I为单位矩阵，λ为大于0的常数，如果λ为0，则LM算法为高斯牛顿法，如果λ取值很大，则LM算法接近梯度下降法。因此当每迭代成功一步时，需把λ减小一些，这样在接近误差目标的时候，逐渐与高斯牛顿法相似。J(**h**)为式24的E(**h**)的雅可比矩阵（Jacobian矩阵），我们在前面已分析过了，**H**是只有8个自由度的单应矩阵，所以我们可以把**H**进行归一化处理，使h9为1，则J(**h**)只需对**H**中前8个元素求偏导数：

（29）

式中，n表示待拼接的图像数量。

下面我们就分别给出ei(**h**)（式25，其中h9＝1）对**h**的偏导数：

（30）

由于ei(**h**)是由横坐标和纵坐标两个值组成的，因此在计算E(**h**)（式24）和∆**h**（式28）时，先分别对横、纵坐标单独进行相关运算，然后再把横、纵坐标下的两个值相加。

下面我们给出LM算法计算**h**的步骤：

●把RANSAC最后计算得到的**h**作为LM算法的初始**h**，并初始化λ为0.001；

●进入迭代，当满足最大迭代次数或∆**h**小于一定程度（即误差目标函数变化很小）时，退出迭代；

▲ 由式28计算∆**h**；

▲比较E(**h**+∆**h**)和E(**h**)的大小，如果E(**h**+∆**h**)小于E(**h**)，则**h**←**h**+∆**h**，λ←λ/10；如果E(**h**+∆**h**)大于E(**h**)，则λ←10λ。

2.2 源码

class detail::FeaturesMatcher表示特征匹配：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **class** CV\_EXPORTS FeaturesMatcher
2. {
3. **public**:
4. **virtual** ~FeaturesMatcher() {}
5. //两个重载( )运算符
6. //第一个重载( )运算符负责的是两幅图像之间的匹配，它主要是调用match函数，features1和features2分别表示待匹配的两幅图像的特征，matches\_info表示匹配信息
7. **void** operator ()(**const** ImageFeatures &features1, **const** ImageFeatures &features2,
8. MatchesInfo& matches\_info) { match(features1, features2, matches\_info); }
9. //第二个重载( )运算符负责的是所有待拼接图像的匹配，它可以实现并行处理，在执行的过程需要调用第一个重载( )运算符
10. **void** operator ()(**const** std::vector<ImageFeatures> &features, std::vector<MatchesInfo> &pairwise\_matches,
11. **const** cv::Mat &mask = cv::Mat());
12. //判断是否能够进行并行处理
13. **bool** isThreadSafe() **const** { **return** is\_thread\_safe\_; }
15. **virtual** **void** collectGarbage() {}    //释放已被分配、但还没有被使用的内存
17. **protected**:
18. FeaturesMatcher(**bool** is\_thread\_safe = **false**) : is\_thread\_safe\_(is\_thread\_safe) {}
19. //虚函数，调用子类BestOf2NearestMatcher的match函数，即应用2-NN方法实现匹配，目前只实现了该方法的匹配
20. **virtual** **void** match(**const** ImageFeatures &features1, **const** ImageFeatures &features2,
21. MatchesInfo& matches\_info) = 0;
23. **bool** is\_thread\_safe\_;    //表示是否可以进行并行处理
24. };

FeaturesMatcher类的第二个重载( )运算符：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **void** FeaturesMatcher::operator ()(**const** vector<ImageFeatures> &features, vector<MatchesInfo> &pairwise\_matches,
2. **const** Mat &mask)
3. //features表示所有图像的特征
4. //pairwise\_matches表示两幅图像之间的匹配信息
5. //mask表示图像掩码，指出哪些图像不参与匹配
6. {
7. **const** **int** num\_images = **static\_cast**<**int**>(features.size());    //得到待拼接的图像数量
8. //确保图像掩码的格式正确
9. CV\_Assert(mask.empty() || (mask.type() == CV\_8U && mask.cols == num\_images && mask.rows));
10. Mat\_<uchar> mask\_(mask);    //复制
11. **if** (mask\_.empty())    //都置1，表示都进行匹配
12. mask\_ = Mat::ones(num\_images, num\_images, CV\_8U);
14. vector<pair<**int**,**int**> > near\_pairs;    //定义特征点匹配对
15. //两个嵌套for循环表示得到不同的两个待拼接图像
16. **for** (**int** i = 0; i < num\_images - 1; ++i)
17. **for** (**int** j = i + 1; j < num\_images; ++j)
18. //如果这两幅图像都得到了特征点，并且它们需要匹配，则把这两幅图像组成一个匹配图像对，放入near\_pairs中
19. **if** (features[i].keypoints.size() > 0 && features[j].keypoints.size() > 0 && mask\_(i, j))
20. near\_pairs.push\_back(make\_pair(i, j));
21. //重新定义pairwise\_matches的大小
22. pairwise\_matches.resize(num\_images \* num\_images);
23. //实例化MatchPairsBody结构
24. MatchPairsBody body(\***this**, features, pairwise\_matches, near\_pairs);
25. //如果并行处理可靠，则并行处理匹配图像对的特征点匹配，否则串行处理
26. **if** (is\_thread\_safe\_)
27. parallel\_for\_(Range(0, **static\_cast**<**int**>(near\_pairs.size())), body);
28. **else**
29. body(Range(0, **static\_cast**<**int**>(near\_pairs.size())));
30. LOGLN\_CHAT("");
31. }

MatchPairsBody结构：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **struct** MatchPairsBody : ParallelLoopBody
2. {
3. //构造函数
4. MatchPairsBody(FeaturesMatcher &\_matcher, **const** vector<ImageFeatures> &\_features,
5. vector<MatchesInfo> &\_pairwise\_matches, vector<pair<**int**,**int**> > &\_near\_pairs)
6. : matcher(\_matcher), features(\_features),
7. pairwise\_matches(\_pairwise\_matches), near\_pairs(\_near\_pairs) {}
8. //重载( )运算符，完成特征点匹配
9. **void** operator ()(**const** Range &r) **const**
10. {
11. **const** **int** num\_images = **static\_cast**<**int**>(features.size());    //待拼接的图像数量
12. **for** (**int** i = r.start; i < r.end; ++i)    //遍历匹配对
13. {
14. **int** from = near\_pairs[i].first;    //提取出当前匹配对的第一幅图像
15. **int** to = near\_pairs[i].second;    //提取出当前匹配对的第二幅图像
16. **int** pair\_idx = from\*num\_images + to;    //得到当前匹配对的索引
17. //执行FeaturesMatcher类的第一个重载( )运算符，即调用该类的match函数
18. matcher(features[from], features[to], pairwise\_matches[pair\_idx]);
19. //得到两个匹配对的图像索引
20. pairwise\_matches[pair\_idx].src\_img\_idx = from;
21. pairwise\_matches[pair\_idx].dst\_img\_idx = to;
22. //另一种索引的形式，因为第i幅图像与第j幅图像匹配，也等于第j幅图像与第i幅图像匹配
23. **size\_t** dual\_pair\_idx = to\*num\_images + from;
25. pairwise\_matches[dual\_pair\_idx] = pairwise\_matches[pair\_idx];
26. pairwise\_matches[dual\_pair\_idx].src\_img\_idx = to;
27. pairwise\_matches[dual\_pair\_idx].dst\_img\_idx = from;
28. //把第一种索引形式下得到的单应矩阵H复制给第二种索引形式下的H
29. **if** (!pairwise\_matches[pair\_idx].H.empty())
30. pairwise\_matches[dual\_pair\_idx].H = pairwise\_matches[pair\_idx].H.inv();
31. //交换两个匹配下的索引值
32. **for** (**size\_t** j = 0; j < pairwise\_matches[dual\_pair\_idx].matches.size(); ++j)
33. std::swap(pairwise\_matches[dual\_pair\_idx].matches[j].queryIdx,
34. pairwise\_matches[dual\_pair\_idx].matches[j].trainIdx);
35. LOG(".");
36. }
37. }
39. FeaturesMatcher &matcher;
40. **const** vector<ImageFeatures> &features;
41. vector<MatchesInfo> &pairwise\_matches;
42. vector<pair<**int**,**int**> > &near\_pairs;
44. **private**:
45. **void** operator =(**const** MatchPairsBody&);
46. };

BestOf2NearestMatcher是FeaturesMatcher的一个子类，它的作用是应用2-NN方法（即找到两个最近邻的特征点）实现特征匹配，这里我们主要介绍它的构造函数和match函数。

BestOf2NearestMatcher类的构造函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. BestOf2NearestMatcher::BestOf2NearestMatcher(**bool** try\_use\_gpu, **float** match\_conf, **int** num\_matches\_thresh1, **int** num\_matches\_thresh2)
2. //try\_use\_gpu表示是否使用GPU，缺省值为false
3. //match\_conf用来衡量两个最近邻的特征点的相似程度，即式1中的ρ，该值在0和1之间，越接近1，两个特征点就越不相近，缺省值为0.3
4. //num\_matches\_thresh1表示两幅图像匹配点对的数量阈值，小于该值，则说明两幅图像没有重叠的地方，无需拼接，缺省值为6
5. //num\_matches\_thresh2表示内点的阈值，小于该值，说明再用内点来计算单应矩阵就不再有意义，缺省值为6
6. {
7. #if defined(HAVE\_OPENCV\_GPU) && !defined(DYNAMIC\_CUDA\_SUPPORT)
8. **if** (try\_use\_gpu && getCudaEnabledDeviceCount() > 0)
9. impl\_ = **new** GpuMatcher(match\_conf);
10. **else**
11. #else
12. (**void**)try\_use\_gpu;    //清空该变量
13. #endif
14. //实例化CpuMatcher类，CpuMatcher也是FeaturesMatcher类的一个子类，它主要负责特征匹配，而不负责单应矩阵的计算
15. impl\_ = **new** CpuMatcher(match\_conf);
17. is\_thread\_safe\_ = impl\_->isThreadSafe();    //赋值，表示可以并行处理图像匹配
18. num\_matches\_thresh1\_ = num\_matches\_thresh1;    //阈值赋值
19. num\_matches\_thresh2\_ = num\_matches\_thresh2;    //阈值赋值
20. }

BestOf2NearestMatcher类的match函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **void** BestOf2NearestMatcher::match(**const** ImageFeatures &features1, **const** ImageFeatures &features2,
2. MatchesInfo &matches\_info)
3. //features1和features2分别表示待匹配的两幅图像的特征
4. //matches\_info表示匹配信息
5. {
6. //调用CpuMatcher类中的match函数，得到匹配信息matches\_info，该函数在后面会详细介绍
7. (\*impl\_)(features1, features2, matches\_info);
9. // Check if it makes sense to find homography
10. //判断两幅图像的匹配点对的数量是否达到了设置的阈值，如果小于该阈值，说明两幅图像没有重叠的地方，无需再进行拼接
11. **if** (matches\_info.matches.size() < **static\_cast**<**size\_t**>(num\_matches\_thresh1\_))
12. **return**;
14. // Construct point-point correspondences for homography estimation
15. //定义两个矩阵，用于保存两幅图像的匹配点对的坐标
16. Mat src\_points(1, **static\_cast**<**int**>(matches\_info.matches.size()), CV\_32FC2);
17. Mat dst\_points(1, **static\_cast**<**int**>(matches\_info.matches.size()), CV\_32FC2);
18. //遍历所有匹配点对，得到匹配点对的特征点坐标
19. **for** (**size\_t** i = 0; i < matches\_info.matches.size(); ++i)
20. {
21. **const** DMatch& m = matches\_info.matches[i];
22. //得到第一幅图像的当前匹配点对的特征点坐标
23. Point2f p = features1.keypoints[m.queryIdx].pt;
24. //以图像的中心处为坐标原点，得到此时的特征点坐标，因为默认情况下是以图像的左上角为坐标原点的
25. p.x -= features1.img\_size.width \* 0.5f;
26. p.y -= features1.img\_size.height \* 0.5f;
27. src\_points.at<Point2f>(0, **static\_cast**<**int**>(i)) = p;    //特征点坐标赋值
29. //得到第二幅图像的当前匹配点对的特征点坐标
30. p = features2.keypoints[m.trainIdx].pt;
31. //以图像的中心处为坐标原点，得到此时的特征点坐标，因为默认情况下是以图像的左上角为坐标原点的
32. p.x -= features2.img\_size.width \* 0.5f;
33. p.y -= features2.img\_size.height \* 0.5f;
34. dst\_points.at<Point2f>(0, **static\_cast**<**int**>(i)) = p;    //特征点坐标赋值
35. }
37. // Find pair-wise motion
38. //利用所有的匹配点对得到单应矩阵，findHomography函数在后面有详细的讲解，src\_points和dst\_points分别表示两幅图像的特征点，它们是匹配点对的关系，matches\_info.inliers\_mask表示内点的掩码，即哪些特征点属于内点，CV\_RANSAC表示使用RANSAC的方法来得到单应矩阵
39. matches\_info.H = findHomography(src\_points, dst\_points, matches\_info.inliers\_mask, CV\_RANSAC);
40. //计算单应矩阵的行列式的值，如果小于一个很小的值，则直接退出该函数
41. **if** (std::abs(determinant(matches\_info.H)) < numeric\_limits<**double**>::epsilon())
42. **return**;
44. // Find number of inliers
45. matches\_info.num\_inliers = 0;    //匹配点对的内点数先清零
46. //由内点掩码得到内点数
47. **for** (**size\_t** i = 0; i < matches\_info.inliers\_mask.size(); ++i)
48. **if** (matches\_info.inliers\_mask[i])
49. matches\_info.num\_inliers++;
51. // These coeffs are from paper M. Brown and D. Lowe. "Automatic Panoramic Image Stitching
52. // using Invariant Features"
53. //计算匹配置信度c，式23
54. matches\_info.confidence = matches\_info.num\_inliers / (8 + 0.3 \* matches\_info.matches.size());
56. // Set zero confidence to remove matches between too close images, as they don't provide
57. // additional information anyway. The threshold was set experimentally.
58. //如果匹配置信度太大（大于3，3为实验数据），则认为这两幅图像十分接近，可以被看成是一幅图像，因此无需匹配，并要把置信度重新赋值为0
59. matches\_info.confidence = matches\_info.confidence > 3. ? 0. : matches\_info.confidence;
61. // Check if we should try to refine motion
62. //判断内点数是否小于阈值，如果小于，则直接退出
63. **if** (matches\_info.num\_inliers < num\_matches\_thresh2\_)
64. **return**;
66. // Construct point-point correspondences for inliers only
67. //重新创建src\_points和dst\_points这两个矩阵，用来存储内点
68. src\_points.create(1, matches\_info.num\_inliers, CV\_32FC2);
69. dst\_points.create(1, matches\_info.num\_inliers, CV\_32FC2);
70. **int** inlier\_idx = 0;    //表示内点索引
71. **for** (**size\_t** i = 0; i < matches\_info.matches.size(); ++i)    //遍历匹配点对，得到内点
72. {
73. **if** (!matches\_info.inliers\_mask[i])    //不是内点
74. **continue**;
76. **const** DMatch& m = matches\_info.matches[i];    //赋值
78. Point2f p = features1.keypoints[m.queryIdx].pt;    //第一幅图像的内点坐标
79. p.x -= features1.img\_size.width \* 0.5f;    //把坐标原点移至图像中心处
80. p.y -= features1.img\_size.height \* 0.5f;
81. src\_points.at<Point2f>(0, inlier\_idx) = p;    //赋值
83. p = features2.keypoints[m.trainIdx].pt;    //第二幅图像的内点坐标
84. p.x -= features2.img\_size.width \* 0.5f;    //把坐标原点移至图像中心处
85. p.y -= features2.img\_size.height \* 0.5f;
86. dst\_points.at<Point2f>(0, inlier\_idx) = p;    //赋值
88. inlier\_idx++;    //索引计数
89. }
91. // Rerun motion estimation on inliers only
92. //最后再用内点来计算单应矩阵，其实在上次调用findHomography函数时，findHomography函数内部已实现了用内点计算单应矩阵，这里又执行了一遍是为了增强鲁棒性
93. matches\_info.H = findHomography(src\_points, dst\_points, CV\_RANSAC);
94. }

CpuMatcher类中的match函数主要负责特征匹配：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **void** CpuMatcher::match(**const** ImageFeatures &features1, **const** ImageFeatures &features2, MatchesInfo& matches\_info)
2. {
3. //确保两幅图像的特征描述符的类型一致，以及第二幅图像的描述符的格式正确
4. CV\_Assert(features1.descriptors.type() == features2.descriptors.type());
5. CV\_Assert(features2.descriptors.depth() == CV\_8U || features2.descriptors.depth() == CV\_32F);
7. #ifdef HAVE\_TEGRA\_OPTIMIZATION
8. **if** (tegra::match2nearest(features1, features2, matches\_info, match\_conf\_))
9. **return**;
10. #endif
12. matches\_info.matches.clear();    //清空
13. //定义K-D树形式的索引
14. Ptr<flann::IndexParams> indexParams = **new** flann::KDTreeIndexParams();
15. //定义搜索参数
16. Ptr<flann::SearchParams> searchParams = **new** flann::SearchParams();
18. **if** (features2.descriptors.depth() == CV\_8U)
19. {
20. indexParams->setAlgorithm(cvflann::FLANN\_INDEX\_LSH);
21. searchParams->setAlgorithm(cvflann::FLANN\_INDEX\_LSH);
22. }
23. //使用FLANN方法匹配，定义matcher变量
24. FlannBasedMatcher matcher(indexParams, searchParams);
25. vector< vector<DMatch> > pair\_matches;    //表示邻域特征点
26. MatchesSet matches;    //表示匹配点对
28. // Find 1->2 matches
29. //在第二幅图像中，找到与第一幅图像的特征点最相近的两个特征点
30. matcher.knnMatch(features1.descriptors, features2.descriptors, pair\_matches, 2);
31. **for** (**size\_t** i = 0; i < pair\_matches.size(); ++i)    //遍历这两次匹配结果
32. {
33. //如果相近的特征点少于2个，则继续下个匹配
34. **if** (pair\_matches[i].size() < 2)
35. **continue**;
36. //得到两个最相近的特征点
37. **const** DMatch& m0 = pair\_matches[i][0];
38. **const** DMatch& m1 = pair\_matches[i][1];
39. //比较这两个最相近的特征点的相似程度，当满足一定条件时（用match\_conf\_变量来衡量），才能认为匹配成功
40. **if** (m0.distance < (1.f - match\_conf\_) \* m1.distance)    //式1
41. {
42. //把匹配点对分别保存在matches\_info和matches中
43. matches\_info.matches.push\_back(m0);
44. matches.insert(make\_pair(m0.queryIdx, m0.trainIdx));
45. }
46. }
47. LOG("\n1->2 matches: " << matches\_info.matches.size() << endl);
49. // Find 2->1 matches
50. pair\_matches.clear();    //变量清零
51. //在第一幅图像中，找到与第二幅图像的特征点最相近的两个特征点
52. matcher.knnMatch(features2.descriptors, features1.descriptors, pair\_matches, 2);
53. **for** (**size\_t** i = 0; i < pair\_matches.size(); ++i)    //遍历这两次匹配结果
54. {
55. //如果相近的特征点少于2个，则继续下个匹配
56. **if** (pair\_matches[i].size() < 2)
57. **continue**;
58. //得到两个最相近的特征点
59. **const** DMatch& m0 = pair\_matches[i][0];
60. **const** DMatch& m1 = pair\_matches[i][1];
61. **if** (m0.distance < (1.f - match\_conf\_) \* m1.distance)    //表明匹配成功，式1
62. //如果当前的匹配点对还没有被上一次调用knnMatch函数时得到，则需要把这次的匹配点对保存下来
63. **if** (matches.find(make\_pair(m0.trainIdx, m0.queryIdx)) == matches.end())
64. matches\_info.matches.push\_back(DMatch(m0.trainIdx, m0.queryIdx, m0.distance));
65. }
66. LOG("1->2 & 2->1 matches: " << matches\_info.matches.size() << endl);
67. }

findHomography函数的作用是得到单应矩阵**H**，它有两种形式，但归根到底都是执行的下面这种形式：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. cv::Mat cv::findHomography( InputArray \_points1, InputArray \_points2,
2. **int** method, **double** ransacReprojThreshold, OutputArray \_mask )
3. //\_points1和\_points2分别表示两幅图像匹配点对的特征点
4. //method表示计算单应矩阵的方法
5. //ransacReprojThreshold表示重映射误差阈值η，该参数仅在RANSAC算法中使用
6. //\_mask表示输出掩码，用以区分内点和外点
7. //该函数返回单应矩阵
8. {
9. //得到两幅图像匹配点对的特征点矩阵的形式
10. Mat points1 = \_points1.getMat(), points2 = \_points2.getMat();
11. **int** npoints = points1.checkVector(2);    //匹配点对的数量
12. //确保匹配点对数量大于0，并且两幅图像的匹配点对数量和类型要一致
13. CV\_Assert( npoints >= 0 && points2.checkVector(2) == npoints &&
14. points1.type() == points2.type());
16. Mat H(3, 3, CV\_64F);    //表示单应矩阵
17. CvMat \_pt1 = points1, \_pt2 = points2;    //赋值
18. //c\_mask表示\_mask的另一种形式，p\_mask表示c\_mask的指针
19. CvMat matH = H, c\_mask, \*p\_mask = 0;
20. **if**( \_mask.needed() )    //如果需要掩码\_mask
21. {
22. \_mask.create(npoints, 1, CV\_8U, -1, **true**);    //创建\_mask
23. p\_mask = &(c\_mask = \_mask.getMat());    //赋值
24. }
25. **bool** ok = cvFindHomography( &\_pt1, &\_pt2, &matH, method, ransacReprojThreshold, p\_mask ) > 0;    //调用cvFindHomography函数
26. **if**( !ok )    //如果没有得到单应矩阵
27. H = Scalar(0);    //赋值单应矩阵为0矩阵
28. **return** H;    //返回单应矩阵
29. }

具体计算单应矩阵的函数cvFindHomography：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. CV\_IMPL **int**
2. cvFindHomography( **const** CvMat\* objectPoints, **const** CvMat\* imagePoints,
3. CvMat\* \_\_H, **int** method, **double** ransacReprojThreshold,
4. CvMat\* mask )
5. {
6. **const** **double** confidence = 0.995;    //表示式20中的q
7. **const** **int** maxIters = 2000;    //初始化RANSAC算法的最大迭代次数
8. **const** **double** defaultRANSACReprojThreshold = 3;    //表示重映射误差阈值η的默认值
9. **bool** result = **false**;    //表示单应矩阵的计算是否成功
10. //m和M分别表示imagePoints和objectPoints的齐次坐标下的值，tempMask表示掩码
11. Ptr<CvMat> m, M, tempMask;
13. **double** H[9];    //代表单应矩阵H的向量形式h
14. CvMat matH = cvMat( 3, 3, CV\_64FC1, H );    //表示单应矩阵
15. **int** count;    //表示特征点的数量
16. //确保输入的两个特征点是矩阵的形式
17. CV\_Assert( CV\_IS\_MAT(imagePoints) && CV\_IS\_MAT(objectPoints) );
19. count = MAX(imagePoints->cols, imagePoints->rows);    //得到特征点的数量
20. CV\_Assert( count >= 4 );    //确保特征点的数量必须不小于4个
21. //如果输入的重映射误差阈值η不大于0，则重新设置为默认值
22. **if**( ransacReprojThreshold <= 0 )
23. ransacReprojThreshold = defaultRANSACReprojThreshold;
24. //把特征点imagePoints的直角坐标转换为齐次坐标m
25. m = cvCreateMat( 1, count, CV\_64FC2 );
26. cvConvertPointsHomogeneous( imagePoints, m );
27. //把特征点objectPoints的直角坐标转换为齐次坐标M
28. M = cvCreateMat( 1, count, CV\_64FC2 );
29. cvConvertPointsHomogeneous( objectPoints, M );
31. **if**( mask )    //如果定义了掩码mask，则必须确保该变量的格式正确
32. {
33. CV\_Assert( CV\_IS\_MASK\_ARR(mask) && CV\_IS\_MAT\_CONT(mask->type) &&
34. (mask->rows == 1 || mask->cols == 1) &&
35. mask->rows\*mask->cols == count );
36. }
37. //创建tempMask，并赋值为1
38. **if**( mask || count > 4 )
39. tempMask = cvCreateMat( 1, count, CV\_8U );
40. **if**( !tempMask.empty() )
41. cvSet( tempMask, cvScalarAll(1.) );
42. //定义CvHomographyEstimator类变量，表示用4个匹配点对估计计算单应矩阵
43. CvHomographyEstimator estimator(4);
44. //基于method值的不同，应用不同的算法来计算单应矩阵，在这里，我们用的是RANSAC算法，所以调用的是runRANSAC函数，该函数会在后面给出详细的介绍
45. **if**( count == 4 )
46. method = 0;
47. **if**( method == CV\_LMEDS )    //最小中值法
48. result = estimator.runLMeDS( M, m, &matH, tempMask, confidence, maxIters );
49. **else** **if**( method == CV\_RANSAC )    //RANSAC算法
50. result = estimator.runRANSAC( M, m, &matH, tempMask, ransacReprojThreshold, confidence, maxIters);
51. **else**    //最小二乘法
52. result = estimator.runKernel( M, m, &matH ) > 0;
53. //如果应用的是最小中值法或RANSAC算法，并且得到了正确的单应矩阵，则需要应用所有的内点得到最终的单应矩阵
54. **if**( result && count > 4 )
55. {
56. //在M中提取出内点
57. icvCompressPoints( (CvPoint2D64f\*)M->data.ptr, tempMask->data.ptr, 1, count );
58. //在m中提取出内点，并得到内点的数量count
59. count = icvCompressPoints( (CvPoint2D64f\*)m->data.ptr, tempMask->data.ptr, 1, count );
60. //重新定义m和M的长度，因为前count个元素就是内点
61. M->cols = m->cols = count;
62. //如果是RANSAC方法，则再用所有的内点得到最终的单应矩阵matH
63. **if**( method == CV\_RANSAC )
64. estimator.runKernel( M, m, &matH );
65. //应用LM算法计算单应函数，refine函数在后面会给出详细的介绍
66. estimator.refine( M, m, &matH, 10 );
67. }
69. **if**( result )    //转换单应矩阵的格式
70. cvConvert( &matH, \_\_H );
72. **if**( mask && tempMask )    //转换内点掩码mask的格式
73. {
74. **if**( CV\_ARE\_SIZES\_EQ(mask, tempMask) )
75. cvCopy( tempMask, mask );
76. **else**
77. cvTranspose( tempMask, mask );
78. }
80. **return** (**int**)result;    //返回
81. }

在前面的cvFindHomography函数中，用到了CvHomographyEstimator类，而它是CvModelEstimator2的子类，在该父类中实现了具体的计算单应矩阵的不同算法，在这里我们只介绍runRANSAC函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **bool** CvModelEstimator2::runRANSAC( **const** CvMat\* m1, **const** CvMat\* m2, CvMat\* model,
2. CvMat\* mask0, **double** reprojThreshold,
3. **double** confidence, **int** maxIters )
4. //m1和m2分别为两幅图像的特征点
5. //model表示输出得到的单应矩阵
6. //mask0表示输出的掩码
7. //reprojThreshold表示重映射误差阈值
8. //confidence表示式20中的q，该值为0.995
9. //maxIters表示最大迭代次数
10. {
11. **bool** result = **false**;    //表示输出标志变量
12. cv::Ptr<CvMat> mask = cvCloneMat(mask0);    //复制
13. //models分别表示为单应矩阵、重映射误差和掩码
14. cv::Ptr<CvMat> models, err, tmask;
15. cv::Ptr<CvMat> ms1, ms2;    //表示计算单应矩阵时要应用的特征点矩阵形式
17. **int** iter, niters = maxIters;    //分别表示迭代索引和最大迭代次数
18. //count表示特征点的数量，maxGoodCount表示最佳单应矩阵下的内点的数量
19. **int** count = m1->rows\*m1->cols, maxGoodCount = 0;
20. //确保m1、m2和mask0的大小一致
21. CV\_Assert( CV\_ARE\_SIZES\_EQ(m1, m2) && CV\_ARE\_SIZES\_EQ(m1, mask) );
22. //全局变量modelPoints表示计算单应矩阵所需要的特征点的数量，在这里，该值在定义CvHomographyEstimator类时，已通过构造函数被赋值为4
23. **if**( count < modelPoints )    //特征点的数量不能小于modelPoints
24. **return** **false**;
25. //全局变量modelSize和maxBasicSolutions分别表示单应矩阵尺寸大小和分辨率，在这里，这两个值在定义CvHomographyEstimator类时，已通过构造函数分别被默认为cvSize(3,3)和1
26. //创建models、err和tmask
27. models = cvCreateMat( modelSize.height\*maxBasicSolutions, modelSize.width, CV\_64FC1 );
28. err = cvCreateMat( 1, count, CV\_32FC1 );
29. tmask = cvCreateMat( 1, count, CV\_8UC1 );
30. //创建ms1和ms2
31. **if**( count > modelPoints )    //特征点的数量大于modelPoints
32. {
33. ms1 = cvCreateMat( 1, modelPoints, m1->type );
34. ms2 = cvCreateMat( 1, modelPoints, m2->type );
35. }
36. **else**    //特征点的数量等于modelPoints
37. {
38. niters = 1;    //表示只迭代一次
39. ms1 = cvCloneMat(m1);
40. ms2 = cvCloneMat(m2);
41. }
43. **for**( iter = 0; iter < niters; iter++ )    //进入迭代循环
44. {
45. **int** i, goodCount, nmodels;    //goodCount表示内点的数量
46. //如果特征点的数量大于modelPoints，则需要随机抽取出modelPoints个特征点
47. **if**( count > modelPoints )
48. {
49. //分别从m1和m2中经过不大于300次的迭代，抽取出modelPoints个特征点，放入ms1和ms2中，并且要求modelPoints个特征点中任意3点都不能共线
50. **bool** found = getSubset( m1, m2, ms1, ms2, 300 );
51. **if**( !found )    //如果此次的迭代，没能成功抽取出modelPoints个特征点
52. {
53. **if**( iter == 0 )    //如果是第0次迭代，则退出该函数
54. **return** **false**;
55. **break**;    //退出迭代循环
56. }
57. }
58. //调用CvHomographyEstimator::runKernel函数，利用最小二乘法计算单应矩阵models，该函数在后面给出详细的介绍，如果执行runKernel时，得到了单应函数，则nmodels为1，否则为0
59. nmodels = runKernel( ms1, ms2, models );
60. **if**( nmodels <= 0 )    //此次迭代没有得到单应矩阵，则进行下次迭代
61. **continue**;
62. //得到了单应矩阵，此循环只执行一次，因为此时nmodels为1
63. **for**( i = 0; i < nmodels; i++ )
64. {
65. CvMat model\_i;
66. //把3×3的单应矩阵models转换为1×9的行向量的形式model\_i
67. cvGetRows( models, &model\_i, i\*modelSize.height, (i+1)\*modelSize.height );
68. //以掩码的形式得到内点，findInliers在后面给出详细的介绍
69. goodCount = findInliers( m1, m2, &model\_i, err, tmask, reprojThreshold );
70. //如果当前得到内点数量比以前的多，则更新最佳单应矩阵
71. **if**( goodCount > MAX(maxGoodCount, modelPoints-1) )
72. {
73. std::swap(tmask, mask);    //更新掩码
74. cvCopy( &model\_i, model );    //更新单应矩阵
75. maxGoodCount = goodCount;    //更新最大内点数
76. //重新计算最大迭代次数niters，后面有详细介绍
77. niters = cvRANSACUpdateNumIters( confidence,
78. (**double**)(count - goodCount)/count, modelPoints, niters );
79. }
80. }
81. }
83. **if**( maxGoodCount > 0 )    //得到了单应矩阵，重新赋值掩码
84. {
85. **if**( mask != mask0 )
86. cvCopy( mask, mask0 );
87. result = **true**;    //表示得到了单应矩阵
88. }
90. **return** result;    //函数返回
91. }

前面有多个地方调用了runKernel函数，它的本质是利用最小二乘法计算单应矩阵：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **int** CvHomographyEstimator::runKernel( **const** CvMat\* m1, **const** CvMat\* m2, CvMat\* H )
2. //m1和m2表示匹配点对的特征点
3. //H表示得到的单应矩阵
4. {
5. **int** i, count = m1->rows\*m1->cols;    //count表示特征点的数量
6. **const** CvPoint2D64f\* M = (**const** CvPoint2D64f\*)m1->data.ptr;    //得到矩阵m1的指针
7. **const** CvPoint2D64f\* m = (**const** CvPoint2D64f\*)m2->data.ptr;    //得到矩阵m2的指针
8. //LtL表示式18中的待分解的矩阵ATA，W表示特征值，V表示特征向量
9. **double** LtL[9][9], W[9][1], V[9][9];
10. //把上面定义的数组转换为矩阵的形式
11. CvMat \_LtL = cvMat( 9, 9, CV\_64F, LtL );
12. CvMat matW = cvMat( 9, 1, CV\_64F, W );
13. CvMat matV = cvMat( 9, 9, CV\_64F, V );
14. CvMat \_H0 = cvMat( 3, 3, CV\_64F, V[8] );    //表示最后一个特征向量，即为式18的h
15. CvMat \_Htemp = cvMat( 3, 3, CV\_64F, V[7] );    //表示临时用到的向量变量
16. //cM和cm表示位移量，sM和sm表示尺度
17. CvPoint2D64f cM={0,0}, cm={0,0}, sM={0,0}, sm={0,0};
18. //按式10计算位移量
19. **for**( i = 0; i < count; i++ )
20. {
21. cm.x += m[i].x; cm.y += m[i].y;
22. cM.x += M[i].x; cM.y += M[i].y;
23. }
24. cm.x /= count; cm.y /= count;
25. cM.x /= count; cM.y /= count;
26. //按式11计算坐标尺度
27. **for**( i = 0; i < count; i++ )
28. {
29. sm.x += fabs(m[i].x - cm.x);
30. sm.y += fabs(m[i].y - cm.y);
31. sM.x += fabs(M[i].x - cM.x);
32. sM.y += fabs(M[i].y - cM.y);
33. }
34. //式11中的分母值不能太小
35. **if**( fabs(sm.x) < DBL\_EPSILON || fabs(sm.y) < DBL\_EPSILON ||
36. fabs(sM.x) < DBL\_EPSILON || fabs(sM.y) < DBL\_EPSILON )
37. **return** 0;    //函数返回，0表示计算失败
38. sm.x = count/sm.x; sm.y = count/sm.y;
39. sM.x = count/sM.x; sM.y = count/sM.y;
41. **double** invHnorm[9] = { 1./sm.x, 0, cm.x, 0, 1./sm.y, cm.y, 0, 0, 1 };    //式16
42. **double** Hnorm2[9] = { sM.x, 0, -cM.x\*sM.x, 0, sM.y, -cM.y\*sM.y, 0, 0, 1 };    //式12中的T
43. CvMat \_invHnorm = cvMat( 3, 3, CV\_64FC1, invHnorm );    //矩阵形式
44. CvMat \_Hnorm2 = cvMat( 3, 3, CV\_64FC1, Hnorm2 );    //矩阵形式
46. cvZero( &\_LtL );    //\_LtL清零
47. **for**( i = 0; i < count; i++ )    //遍历所有的匹配点对
48. {
49. **double** x = (m[i].x - cm.x)\*sm.x, y = (m[i].y - cm.y)\*sm.y;    //表示式14的第一项
50. **double** X = (M[i].x - cM.x)\*sM.x, Y = (M[i].y - cM.y)\*sM.y;    //表示式14的第二项
51. **double** Lx[] = { X, Y, 1, 0, 0, 0, -x\*X, -x\*Y, -x };    //表示式7中的第一行
52. **double** Ly[] = { 0, 0, 0, X, Y, 1, -y\*X, -y\*Y, -y };    //表示式7中的第二行
53. **int** j, k;
54. //因为ATA是对称矩阵，所以这里只需得到该矩阵的右上角元素即可
55. **for**( j = 0; j < 9; j++ )
56. **for**( k = j; k < 9; k++ )
57. LtL[j][k] += Lx[j]\*Lx[k] + Ly[j]\*Ly[k];
58. }
59. //把ATA的右上角元素复制到左下角，构成完整的ATA
60. cvCompleteSymm( &\_LtL );
62. //cvSVD( &\_LtL, &matW, 0, &matV, CV\_SVD\_MODIFY\_A + CV\_SVD\_V\_T );
63. cvEigenVV( &\_LtL, &matV, &matW );    //对ATA进行特征值分解
64. //下面两个矩阵乘法实现了式15
65. cvMatMul( &\_invHnorm, &\_H0, &\_Htemp );
66. cvMatMul( &\_Htemp, &\_Hnorm2, &\_H0 );
67. //对H进行归一化处理，使h9为1
68. cvConvertScale( &\_H0, H, 1./\_H0.data.db[8] );
70. **return** 1;
71. }

得到内点的函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **int** CvModelEstimator2::findInliers( **const** CvMat\* m1, **const** CvMat\* m2,
2. **const** CvMat\* model, CvMat\* \_err,
3. CvMat\* \_mask, **double** threshold )
4. //m1和m2表示匹配点对的特征点
5. //model表示单应矩阵H
6. //\_err表示重映射误差
7. //\_mask表示内点掩码，内点是用掩码的方式表示的
8. //threshold表示重映射误差的阈值η
9. {
10. //count表示特征点的数量，goodCount表示最终得到的内点数量
11. **int** i, count = \_err->rows\*\_err->cols, goodCount = 0;
12. **const** **float**\* err = \_err->data.fl;    //表示误差指针
13. uchar\* mask = \_mask->data.ptr;    //表示掩码指针
14. //调用子类中的computeReprojError函数，见下面的分析
15. computeReprojError( m1, m2, model, \_err );
16. threshold \*= threshold;    //误差阈值的平方
17. //遍历所有特征点，以掩码的形式记录下误差小于阈值的特征点，即内点，并计数
18. **for**( i = 0; i < count; i++ )
19. goodCount += mask[i] = err[i] <= threshold;
20. **return** goodCount;    //返回内点数量
21. }

计算重映射误差，即式21：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. oid CvHomographyEstimator::computeReprojError( **const** CvMat\* m1, **const** CvMat\* m2,
2. **const** CvMat\* model, CvMat\* \_err )
3. //m1和m2为匹配点对
4. //model表示单应矩阵H
5. //\_err表示所有匹配点对的重映射误差，即式21几何距离的平方
6. {
7. **int** i, count = m1->rows\*m1->cols;    //count表示特征点的数量
8. **const** CvPoint2D64f\* M = (**const** CvPoint2D64f\*)m1->data.ptr;    //特征点复制指针
9. **const** CvPoint2D64f\* m = (**const** CvPoint2D64f\*)m2->data.ptr;    //特征点复制指针
10. **const** **double**\* H = model->data.db;    //单应矩阵复制指针
11. **float**\* err = \_err->data.fl;    //误差复制指针
13. **for**( i = 0; i < count; i++ )    //遍历所有特征点，计算重映射误差
14. {
15. **double** ww = 1./(H[6]\*M[i].x + H[7]\*M[i].y + 1.);    //式21中分式的分母部分
16. **double** dx = (H[0]\*M[i].x + H[1]\*M[i].y + H[2])\*ww - m[i].x;    //式21中x坐标之差
17. **double** dy = (H[3]\*M[i].x + H[4]\*M[i].y + H[5])\*ww - m[i].y;    //式21中y坐标之差
18. err[i] = (**float**)(dx\*dx + dy\*dy);    //得到当前匹配点对的重映射误差
19. }
20. }

利用LM算法计算单应矩阵**H**：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. **bool** CvHomographyEstimator::refine( **const** CvMat\* m1, **const** CvMat\* m2, CvMat\* model, **int** maxIters )
2. //m1和m2分别是匹配点对的内点
3. //model为输入输出变量，输入表示LM算法的初始化H，输出表示LM算法最终得到的H
4. //maxIters为LM算法的最大迭代次数
5. {
6. //实例化CvLevMarq类，表示LM算法，CvLevMarq类构造函数的第一个参数表示需要优化变量的数量，因为这里我们要得到H的前8个元素，所以该变量设置为8；CvLevMarq类构造函数的第二个参数这里没有用到，所以该变量设置为0；CvLevMarq类构造函数的第三个参数表示LM算法的迭代终止条件
7. CvLevMarq solver(8, 0, cvTermCriteria(CV\_TERMCRIT\_ITER+CV\_TERMCRIT\_EPS, maxIters, DBL\_EPSILON));
8. **int** i, j, k, count = m1->rows\*m1->cols;    //count表示特征点的数量
9. **const** CvPoint2D64f\* M = (**const** CvPoint2D64f\*)m1->data.ptr;    //m1指针复制
10. **const** CvPoint2D64f\* m = (**const** CvPoint2D64f\*)m2->data.ptr;    //m2指针复制
11. //modelPart和solver.param都表示的是单应矩阵
12. CvMat modelPart = cvMat( solver.param->rows, solver.param->cols, model->type, model->data.ptr );
13. cvCopy( &modelPart, solver.param );    //复制，初始化用于LM算法的单应矩阵
15. **for**(;;)    //死循环
16. {
17. **const** CvMat\* \_param = 0;    //\_param表示单应矩阵
18. CvMat \*\_JtJ = 0, \*\_JtErr = 0;    //\_JtJ表示式28中的JTJ，\_JtErr表示式28中的JTe
19. **double**\* \_errNorm = 0;    //\_errNorm表示式24的误差指标函数
20. //调用updateAlt函数，由式27和式28计算更新迭代后单应矩阵\_param
21. **if**( !solver.updateAlt( \_param, \_JtJ, \_JtErr, \_errNorm ))
22. **break**;    //满足迭代终止条件，则退出死循环
23. //遍历所有内点，计算更新\_JtJ、\_JtErr和\_errNorm
24. **for**( i = 0; i < count; i++ )
25. {
26. **const** **double**\* h = \_param->data.db;    //单应矩阵赋值
27. **double** Mx = M[i].x, My = M[i].y;    //得到m1的坐标
28. **double** ww = h[6]\*Mx + h[7]\*My + 1.;    //得到式30中分母部分
29. ww = fabs(ww) > DBL\_EPSILON ? 1./ww : 0;    //倒数
30. //计算式25中的横、纵坐标的分式部分
31. **double** \_xi = (h[0]\*Mx + h[1]\*My + h[2])\*ww;
32. **double** \_yi = (h[3]\*Mx + h[4]\*My + h[5])\*ww;
33. **double** err[] = { \_xi - m[i].x, \_yi - m[i].y };    //式25
34. **if**( \_JtJ || \_JtErr )
35. {
36. **double** J[][8] =    //式30
37. {
38. { Mx\*ww, My\*ww, ww, 0, 0, 0, -Mx\*ww\*\_xi, -My\*ww\*\_xi },
39. { 0, 0, 0, Mx\*ww, My\*ww, ww, -Mx\*ww\*\_yi, -My\*ww\*\_yi }
40. };
41. //计算式28中的JTJ和JTe
42. **for**( j = 0; j < 8; j++ )
43. {
44. **for**( k = j; k < 8; k++ )
45. \_JtJ->data.db[j\*8+k] += J[0][j]\*J[0][k] + J[1][j]\*J[1][k];
46. \_JtErr->data.db[j] += J[0][j]\*err[0] + J[1][j]\*err[1];
47. }
48. }
49. **if**( \_errNorm )    //计算式24
50. \*\_errNorm += err[0]\*err[0] + err[1]\*err[1];
51. }
52. }
54. cvCopy( solver.param, &modelPart );    //复制单应矩阵
55. **return** **true**;
56. }

更新RANSAC迭代次数，即计算式20：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. CV\_IMPL **int**
2. cvRANSACUpdateNumIters( **double** p, **double** ep,
3. **int** model\_points, **int** max\_iters )
4. //p表示式20中的q
5. //ep表示式20中的ε
6. //model\_points表示单应矩阵所需要的匹配点对数，这里为4
7. //max\_iters表示迭代的次数
8. //该函数返回更新后的迭代次数N，即式20的结果
9. {
10. **if**( model\_points <= 0 )    //这里model\_points是等于4的
11. CV\_Error( CV\_StsOutOfRange, "the number of model points should be positive" );
12. //确保p和ep都在0和1之间
13. p = MAX(p, 0.);
14. p = MIN(p, 1.);
15. ep = MAX(ep, 0.);
16. ep = MIN(ep, 1.);
18. // avoid inf's & nan's
19. **double** num = MAX(1. - p, DBL\_MIN);    //式20中分子log里面的部分
20. **double** denom = 1. - pow(1. - ep,model\_points);    //式20中分母log里面的部分
21. **if**( denom < DBL\_MIN )    //分母不能太小
22. **return** 0;
24. num = log(num);    //式20的分子
25. denom = log(denom);    //式20的分母
26. //计算式20，更新迭代次数N
27. **return** denom >= 0 || -num >= max\_iters\*(-denom) ?
28. max\_iters : cvRound(num/denom);
29. }

2.3 应用

下面我们给出用拼接算法中的匹配方法进行特征匹配的程序：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78799194) [copy](http://blog.csdn.net/zhaocj/article/details/78799194)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. #include <iomanip>
23. **using** **namespace** cv;
24. **using** **namespace** std;
25. **using** **namespace** detail;
27. **int** main(**int** argc, **char**\*\* argv)
28. {
30. vector<Mat> imgs;    //表示待拼接的图像矢量队列
31. Mat img = imread("1.jpg");    //读取两幅图像，并存入队列
32. imgs.push\_back(img);
33. img = imread("2.jpg");
34. imgs.push\_back(img);
36. Ptr<FeaturesFinder> finder;    //特征检测
37. finder = **new** SurfFeaturesFinder();
38. vector<ImageFeatures> features(2);
39. (\*finder)(imgs[0], features[0]);
40. (\*finder)(imgs[1], features[1]);
42. vector<MatchesInfo> pairwise\_matches;    //特征匹配
43. BestOf2NearestMatcher matcher(**false**, 0.3f, 6, 6);    //定义特征匹配器，2NN方法
44. matcher(features, pairwise\_matches);    //进行特征匹配
46. Mat dispimg;    //两幅图像合并成一幅图像显示
47. dispimg.create(Size(imgs[0].cols+imgs[1].cols, max(imgs[1].rows,imgs[1].rows)), CV\_8UC3);
48. Mat imgROI = dispimg(Rect(0, 0, (**int**)(imgs[0].cols), (**int**)(imgs[0].rows)));
49. resize(imgs[0], imgROI, Size((**int**)(imgs[0].cols), (**int**)(imgs[0].rows)));
50. imgROI = dispimg(Rect((**int**)(imgs[0].cols), 0, (**int**)(imgs[1].cols), (**int**)(imgs[1].rows)));
51. resize(imgs[1], imgROI, Size((**int**)(imgs[1].cols), (**int**)(imgs[1].rows)));
53. Point2f p1, p2;    //分别表示两幅图像内的匹配点对
54. **int** j=0;
55. **for** (**size\_t** i = 0; i < pairwise\_matches[1].matches.size(); ++i)    //遍历匹配点对
56. {
57. **if** (!pairwise\_matches[1].inliers\_mask[i])    //不是内点，则继续下一次循环
58. **continue**;
60. **const** DMatch& m = pairwise\_matches[1].matches[i];    //得到内点的匹配点对
61. p1 = features[0].keypoints[m.queryIdx].pt;
62. p2 = features[1].keypoints[m.trainIdx].pt;
63. p2.x += features[0].img\_size.width;    //p2在合并图像上的坐标
65. line(dispimg, p1, p2, Scalar(0,0,255), 1, CV\_AA);    //画直线
67. **if**(j++==10)    //内点数量较多，我们只显示10个
68. **break**;
69. }
71. //在终端显示内点数量和单应矩阵
72. cout<<"内点数量："<<endl;
73. cout<<setw(10)<<pairwise\_matches[1].matches.size()<<endl<<endl;
75. **const** **double**\* h = **reinterpret\_cast**<**const** **double**\*>(pairwise\_matches[1].H.data);
76. cout<<"单应矩阵："<<endl;
77. cout<<setw(10)<<(**int**)(h[0]+0.5)<<setw(6)<<(**int**)(h[1]+0.5)<<setw(6)<<(**int**)(h[2]+0.5)<<endl;
78. cout<<setw(10)<<(**int**)(h[3]+0.5)<<setw(6)<<(**int**)(h[4]+0.5)<<setw(6)<<(**int**)(h[5]+0.5)<<endl;
79. cout<<setw(10)<<(**int**)(h[6]+0.5)<<setw(6)<<(**int**)(h[7]+0.5)<<setw(6)<<(**int**)(h[8]+0.5)<<endl;
81. imshow("匹配显示", dispimg);    //显示匹配图像
82. waitKey(0);
84. **return** 0;
85. }

终端显示的结果为：

内点数量：

      474

单应矩阵：

        1     0   -465

        0     1    -30

        0     0      1

匹配图像如图5所示。



图5 匹配图像

# Opencv2.4.9源码分析——Stitching（三）

3、相机参数评估

3.1 原理

相机参数的评估也称为相机定标。要想理解这部分内容，首先应该从成像原理开始讲起。

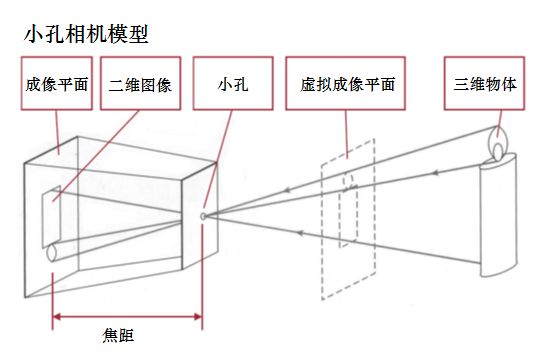


图6 小孔成像原理

从图6可以看出，真实物体通过小孔映射到成像平面上，小孔到成像平面的距离称为焦距f。在成像平面上的图像是镜像倒立的，所以为了研究方便，在小孔和物体之间定义一个虚拟成像平面（在后面，我们把该平面也称为成像平面），它与小孔的距离也为焦距，则两个成像平面的图像大小是相同的，但虚拟成像平面上的图像与原物体的方向是一样的。

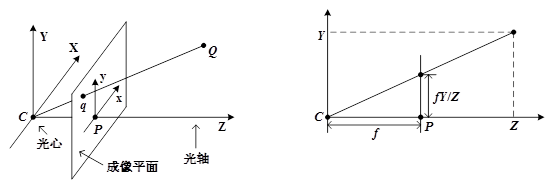
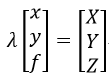


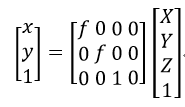
图7 成像的几何模型

我们以小孔为坐标原点建立一个三维直角坐标系XYZ（如图7所示），坐标原点C称为相机的光心。成像平面xy平行于XY，并且距离光心C为f，其中Z轴定义为光轴，它与成像平面xy的交点为P，因此CP＝f。设空间中的任一点Q的坐标为（X, Y, Z），该点映射到成像平面的点q的坐标为(x, y,f)。

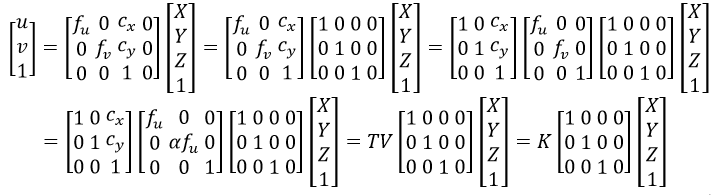
由几何知识可得：

（31）

式中，λ＝Z/f，则x＝fX/Z，y＝fY/Z。因此空间中Q点的三维坐标映射到成像平面的二维坐标q点在齐次坐标下的线性映射关系为：

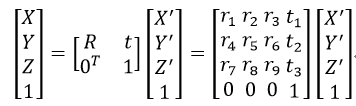
（32）

如图7所示，在成像平面（坐标系为xy）的坐标原点为P，但图像（设坐标系为uv）的坐标原点一般在左上角，所以这两个坐标系之间需要通过平移来进行转换。另外成像平面xy是长度单位，而相机图像传感器的单位是像素，因此像素与长度之间也是需要转换的，而且水平和垂直的像素往往是不相同的，所以横纵轴的转换系数是不一致。因此式32改写为：

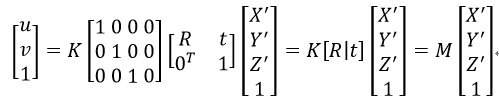
（33）

式中，fu和fv为焦距f在横纵轴的长度和像素的转换，它们之间的关系可以写为fv＝αfu，(cx, cy)为坐标平移。因为矩阵K的参数都是基于相机内部自身的参数，因此K称为相机内参数，K＝TV。

相机除了具有内参数外，还有外参数。式33中的(X, Y, Z)称为相机坐标，而它与真实的世界坐标(X’, Y’, Z’)还存在欧几里得变换，即：

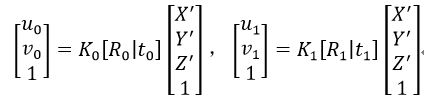
（34）

式中，R为3×3的旋转矩阵，t为3×1的平移向量。式34代入式33，得

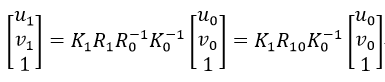
（35）

式中，[R|t]称为相机外参数，M称为投影矩阵。

如果得到的两幅图像如图3所示的那样，即相机在三角架上通过旋转得到的两幅图像，则对于同一个世界坐标上的点(X’, Y’, Z’)，在两幅图像上的坐标点分别为(u0, v0)和(u1, v1)，即：

（36）

由于相机只做旋转处理（或者我们认为物体离相机很远），则t0＝t1＝0，从而得到(u0, v0)和(u1, v1)的关系为：

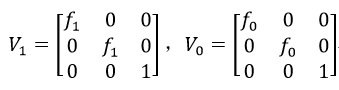
（37）

式中，R10为由图像1到图像0的相对旋转矩阵，R10＝R1R0-1。由式2可得**：**

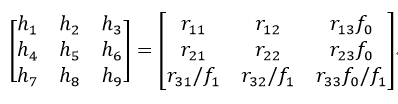
http://img.blog.csdn.net/20171215091937150（38）

为简单起见，我们设两幅图像的相机内参数中的坐标平移都为0，即K＝V，因为抛弃参数T对图像拼接影响不大。

我们在评估焦距时，还需定义f1u＝f1v＝f1，f0u＝f0v＝f0，即设图像的长宽等像素：

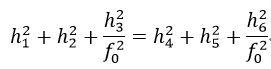
（39）

则式38表示为

（40）

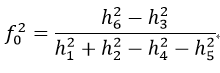
式中，R10＝[rij]。

由式40我们就可以得到焦距f0和f1：观察矩阵R10可知，R10前两行一定有相同的范数，并且是正交的，因此

（41）

http://img.blog.csdn.net/20171215092019621（42）

由式41可得：

（43）

由式42可得：

http://img.blog.csdn.net/20171215092045186（44）

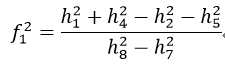
由式43和式44得到了两个f0，选取哪个呢？比较式43和式44中分母部分的绝对值的大小，如果式43的分母大，则选择分式大的值作为f0，否则如果式44的分母大，则选择值小的作为f0。

同理，矩阵R10的前两列也一定有相同的范数，并且也是正交的，因此

http://img.blog.csdn.net/20171215092054869（45）

http://img.blog.csdn.net/20171215092103077（46）

由式45可得

（47）

由式46可得

http://img.blog.csdn.net/20171215092121278（48）

比较式47和式48中分母部分的绝对值的大小，如果式48的分母大，则选择分式大的值作为f1，否则如果式47的分母大，则选择值小的作为f1。

如果两幅图像的焦距相同，则最终这两幅图像的焦距f为：

http://img.blog.csdn.net/20171215092131293（49）

当评估计算得到多个f时，可取这些f的中值作为所有相机的焦距。

焦距得到后，我们就可以由式38得到R10：

http://img.blog.csdn.net/20171215092141598（50）

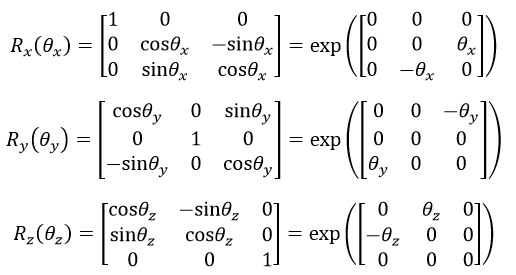
则R01（由图像0到图像1的相对旋转矩阵）为：

http://img.blog.csdn.net/20171215092150336（51）

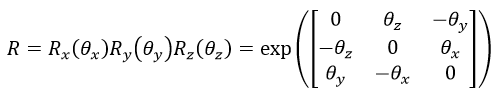
则

http://img.blog.csdn.net/20171215092202010（52）

对于刚性物体，它的旋转都是沿笛卡尔坐标系的x轴、y轴和z轴旋转，则分别沿着这三个轴的旋转矩阵定义为：

（53）

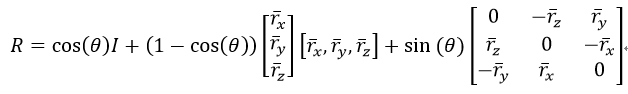
则旋转矩阵R表示为：

（54）

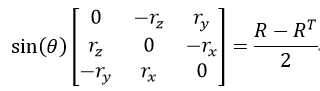
三维旋转除了可以用旋转矩阵描述外，还可以用旋转向量r描述，即r＝[rx, ry, rz]T。旋转向量的长度（模）表示绕轴逆时针旋转的角度θ。旋转向量和旋转矩阵可以通过Rodrigues算法进行转换。由旋转向量转换为旋转矩阵的Rodrigues算法描述如下：

http://img.blog.csdn.net/20171215092235693（55）

http://img.blog.csdn.net/20171215092245071（56）

（57）

式中，I为3×3的单位矩阵。而由旋转矩阵转换为旋转向量的Rodrigues算法公式为：

（58）

当有多幅图像需要拼接为一幅图像时，是要以其中一幅图像为基准，其他图像都要旋转到该基准图像平面上的，所以就需要找到基准平面。这里用到的算法为最大生成树算法。

待拼接图像的排列是无序的，而且我们事先是不知道它们之间的关系的，我们只知道它们之间的单应矩阵，而单应矩阵是由图像间的内点计算评估得到的。由此我们可以构造一个无向图G，G的节点为图像，G的边为内点数，然后利用并查集在该G中得到一棵最大生成树。

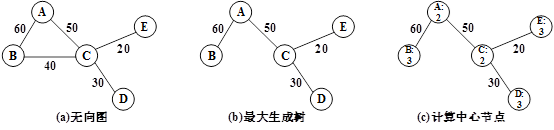


图8 最大生成树

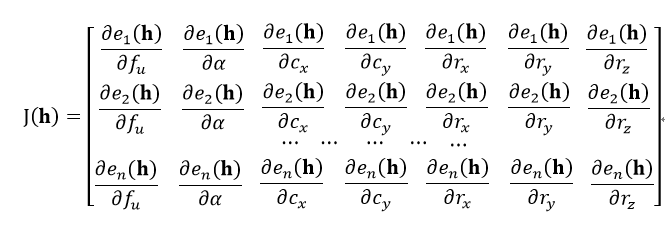
图8为用于拼接的最大生成树的一个例子，图(a)为无向图，节点为图像（A、B、C、D、E），节点间的边为内点数。图(b)为最大生成树，由图像C到图像B要经过最大的边连接，所以要经过图像A，而图像C和图像B之间的连接就需要去掉了。

我们把树的中心节点作为基准图像。中心节点的确定方法为：计算每一个节点到所有叶节点的距离，把其中的最大值作为该节点的值；然后选择这些值中最小者作为中心节点。这里的距离指的是节点间的节点数。如图8(c)所示，节点A和C为中心节点。中心节点可能是1个，也可能是2个，如果是2个，则选择其中一个即可。

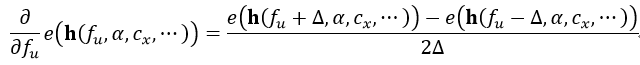
基于以上方法，我们得到了相机的内外参数，但这样得到的参数忽略了多个图像间的约束，而且会产生累计误差。这时，我们就需要用到光束平差法（Bundle Adjustment）来精确化相机参数。光束指的是相机光心“发射”出来的光束（或射线），它透过相片达到物点，因此相片中的点应该和物点处于一个光束线上，但当两者不共线时，我们就需要对结构和视角参数进行调节，以达到最优解甚至共线的目的。最优化一般采用前文介绍过的LM算法。

应用于光束平差法的LM算法，误差指标函数可以有两个，一个是重映射误差，另一个是射线发散误差。

重映射误差公式为式25（即一个内点要有x轴和y轴两个误差值），而单应矩阵**H**是由式38得到。也就是说**H**是由相机的内外参数得到。相机的内外参数一共有7个：fu、α、cx、cy、rx、ry和rz。前4个参数是内参数（见式33），后3个参数是外参数（即式55中的旋转向量的三个元素）。因此式25中的**h**为**h**(fu, α, cx, cy, rx, ry,rz)，由此得到J(**h**)为：

（59）

式中，n表示待拼接的图像数量，也就是所有的相机。有时为了计算方便，导数可以用差分近似，例如我们要计算e对fu的偏导，则

（60）

式中，∆表示一个很小的数。

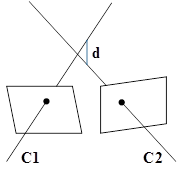
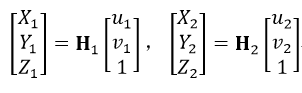


图9 射线发散概念

第二种误差指标函数是基于射线发散原理。如图9所示，不同的相机发出的射线透过相片后达到同一个物点，但由于误差，两条射线不会相交，或者称为两条射线发散了，我们就把这两条射线间的最短距离d定义为射线发散误差。

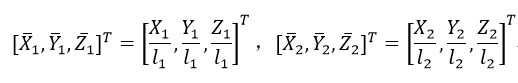
下面，我们不加证明的给出射线发散误差的计算公式：

设(u1, v1, 1)和(u2, v2, 1)为两幅不同图像的同一特征点的齐次坐标，则由单应矩阵**H**1和**H**2分别得到它们的物点坐标为：

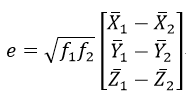
（61）

分别对坐标进行归一化处理：

http://img.blog.csdn.net/20171215092416593（62）

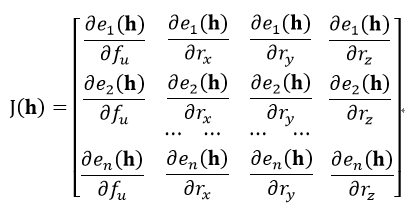
（63）

为了简化计算，射线间的最短距离可以表示为基于不同焦距f1和f2下的归一化后的物点坐标之差：

（64）

式64说明在计算射线发散误差时，每一个内点有x轴、y轴和z轴三个误差值。

我们假设射线就是相机的光轴，因此单应矩阵**H**中的参数α＝1，cx＝cy＝0，所以式25中**h**只是基于4个参数的变量，即**h**(fu, rx,ry, rz)，由此得到射线发散误差的J(**h**)为：

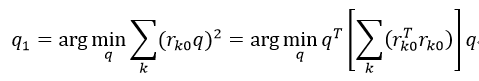
（65）

前面介绍的光束平差法会引起波形效应，即拼接的图像会呈现蛇形分布，这是因为真实拍摄相片时不大可能都保持水平而不倾斜的，也就是重力轴没有垂直于图像。因此我们就需要引入一个全局校正矩阵Q，用于“拉直”拼接图像，该方法也成为波形校正。

校正的目的应该是，在第k个相机旋转矩阵Rk乘以Q后，全局y轴应该与图像的x轴垂直，这种约束条件可以表示为：

http://img.blog.csdn.net/20171215092456840（66）

式中，i＝(1, 0, 0)，j＝(0, 1, 0)，该式的含义是Rk的第一行rk0垂直于Q的第二列q1。式66的这类约束问题可以看成是最小二乘问题：

（67）

因此，q1是矩量矩阵∑rTr的最小特征值所对应的特征向量。矩阵Q的其他列的经验公式为：

http://img.blog.csdn.net/20171215092519227（68）

式中，×表示矩阵的点乘（即对应元素相乘，与MATLAB中的点乘相同），分母表示对分子取模，即q0是归一化的结果。

http://img.blog.csdn.net/20171215092530870（69）

最终的全局校正矩阵Q＝[q0q1q2]，而用Q乘以各个相机的R即实现了波形校正。如果要进行垂直校正拉直，则需选择最大特征值对应的特征向量。

3.2 源码

Estimator类是相机参数评估器的基类，HomographyBasedEstimator和BundleAdjusterBase都是Estimator类的子类。HomographyBasedEstimator类主要负责相机参数的计算评估，BundleAdjusterBase类主要通过光束平差法使相机参数精确化。

我们先来看HomographyBasedEstimator类，它的主要内容就是estimate函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** HomographyBasedEstimator::estimate(**const** vector<ImageFeatures> &features, **const** vector<MatchesInfo> &pairwise\_matches,
2. vector<CameraParams> &cameras)
3. //features表示所有待拼接图像的特征
4. //pairwise\_matches表示匹配点对
5. //cameras表示相机参数信息
6. {
7. LOGLN("Estimating rotations...");
8. #if ENABLE\_LOG
9. int64 t = getTickCount();
10. #endif
12. **const** **int** num\_images = **static\_cast**<**int**>(features.size());    //得到待拼接图像的数量
14. #if 0
15. // Robustly estimate focal length from rotating cameras
16. vector<Mat> Hs;
17. **for** (**int** iter = 0; iter < 100; ++iter)
18. {
19. **int** len = 2 + rand()%(pairwise\_matches.size() - 1);
20. vector<**int**> subset;
21. selectRandomSubset(len, pairwise\_matches.size(), subset);
22. Hs.clear();
23. **for** (**size\_t** i = 0; i < subset.size(); ++i)
24. **if** (!pairwise\_matches[subset[i]].H.empty())
25. Hs.push\_back(pairwise\_matches[subset[i]].H);
26. Mat\_<**double**> K;
27. **if** (Hs.size() >= 2)
28. {
29. **if** (calibrateRotatingCamera(Hs, K))
30. cin.get();
31. }
32. }
33. #endif
34. //is\_focals\_estimated\_为HomographyBasedEstimator类的全局变量，表示是否需要进行焦距评估
35. **if** (!is\_focals\_estimated\_)    //需要焦距评估，因为焦距还没有被评估
36. {
37. // Estimate focal length and set it for all cameras
38. vector<**double**> focals;    //表示所有图像的焦距
39. //评估焦距，estimateFocal在后面给出详细介绍
40. estimateFocal(features, pairwise\_matches, focals);
41. cameras.assign(num\_images, CameraParams());    //为相机参数分配空间
42. **for** (**int** i = 0; i < num\_images; ++i)
43. cameras[i].focal = focals[i];    //相机焦距赋值
44. }
45. **else**    //不需要焦距评估
46. {
47. //得到相机内参数的cx, cy
48. **for** (**int** i = 0; i < num\_images; ++i)
49. {
50. cameras[i].ppx -= 0.5 \* features[i].img\_size.width;
51. cameras[i].ppy -= 0.5 \* features[i].img\_size.height;
52. }
53. }
55. // Restore global motion
56. Graph span\_tree;    //定义最大生成树
57. vector<**int**> span\_tree\_centers;    //表示最大生成树的中心节点
58. //得到最大生成树，该函数在后面给出了详细的介绍
59. findMaxSpanningTree(num\_images, pairwise\_matches, span\_tree, span\_tree\_centers);
60. //以中心节点的图像为基准，计算其他图像与该图像的旋转矩阵，CalcRotation结构体在后面给出了详细的介绍
61. span\_tree.walkBreadthFirst(span\_tree\_centers[0], CalcRotation(num\_images, pairwise\_matches, cameras));
63. // As calculations were performed under assumption that p.p. is in image center
64. **for** (**int** i = 0; i < num\_images; ++i)    //得到相机参数中的坐标平移
65. {
66. cameras[i].ppx += 0.5 \* features[i].img\_size.width;
67. cameras[i].ppy += 0.5 \* features[i].img\_size.height;
68. }
70. LOGLN("Estimating rotations, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
71. }

评估相机焦距：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** estimateFocal(**const** vector<ImageFeatures> &features, **const** vector<MatchesInfo> &pairwise\_matches,
2. vector<**double**> &focals)
3. //features表示图像的特征
4. //pairwise\_matches表示匹配点对
5. //focals表示相机焦距
6. {
7. **const** **int** num\_images = **static\_cast**<**int**>(features.size());    //待拼接的图像数量
8. focals.resize(num\_images);    //定义focals向量变量的长度
10. vector<**double**> all\_focals;    //表示焦距
11. //嵌套循环，得到每两个图像之间的匹配信息
12. **for** (**int** i = 0; i < num\_images; ++i)
13. {
14. **for** (**int** j = 0; j < num\_images; ++j)
15. {
16. //得到第i幅图像与第j幅图像之间的匹配信息
17. **const** MatchesInfo &m = pairwise\_matches[i\*num\_images + j];
18. **if** (m.H.empty())    //没有单应矩阵，说明它们之间不能拼接
19. **continue**;    //进入下一个循环
20. **double** f0, f1;    //表示这两幅图像的焦距
21. **bool** f0ok, f1ok;    //表示是否得到了这两幅图像的焦距
22. //从单应矩阵中得到焦距，focalsFromHomography函数在后面给出介绍
23. focalsFromHomography(m.H, f0, f1, f0ok, f1ok);
24. **if** (f0ok && f1ok)    //得到了两幅图像的焦距
25. all\_focals.push\_back(sqrt(f0 \* f1));    //式49，把焦距存入all\_focals内
26. }
27. }
29. **if** (**static\_cast**<**int**>(all\_focals.size()) >= num\_images - 1)
30. {
31. **double** median;    //表示焦距中值
33. std::sort(all\_focals.begin(), all\_focals.end());    //所有焦距排序
34. //得到所有焦距的中值，作为最终的焦距
35. **if** (all\_focals.size() % 2 == 1)
36. median = all\_focals[all\_focals.size() / 2];
37. **else**
38. median = (all\_focals[all\_focals.size() / 2 - 1] + all\_focals[all\_focals.size() / 2]) \* 0.5;
40. **for** (**int** i = 0; i < num\_images; ++i)    //为所有相机的焦距赋同样的值
41. focals[i] = median;    //焦距赋值
42. }
43. //按照正常的方面没有得到焦距的处理方法：焦距为所有拼接图像的长宽之和的平均值
44. **else**
45. {
46. LOGLN("Can't estimate focal length, will use naive approach");
47. **double** focals\_sum = 0;
48. **for** (**int** i = 0; i < num\_images; ++i)    //所有图像的长宽之和
49. focals\_sum += features[i].img\_size.width + features[i].img\_size.height;
50. **for** (**int** i = 0; i < num\_images; ++i)
51. focals[i] = focals\_sum / num\_images;    //平均值
52. }
53. }

从单应矩阵中得到焦距：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** focalsFromHomography(**const** Mat& H, **double** &f0, **double** &f1, **bool** &f0\_ok, **bool** &f1\_ok)
2. //H表示单应矩阵
3. //f0和f1分别表示单应矩阵H所转换的两幅图像的焦距
4. //f0\_ok和f1\_ok分别表示f0和f1是否评估正确
5. {
6. //确保H的数据类型和格式正确
7. CV\_Assert(H.type() == CV\_64F && H.size() == Size(3, 3));
9. **const** **double**\* h = **reinterpret\_cast**<**const** **double**\*>(H.data);    //赋值指针
10. //分别表示式43和式44，或式47和式48的分母
11. **double** d1, d2; // Denominators
12. //分别表示式43和式44，或式47和式48
13. **double** v1, v2; // Focal squares value candidates
15. f1\_ok = **true**;
16. d1 = h[6] \* h[7];    //式48的分母
17. d2 = (h[7] - h[6]) \* (h[7] + h[6]);    //式47的分母
18. v1 = -(h[0] \* h[1] + h[3] \* h[4]) / d1;    //式48
19. v2 = (h[0] \* h[0] + h[3] \* h[3] - h[1] \* h[1] - h[4] \* h[4]) / d2;    //式47
20. **if** (v1 < v2) std::swap(v1, v2);    //使v1不小于v2
21. //表示到底选取式47还是式48作为f1
22. **if** (v1 > 0 && v2 > 0) f1 = sqrt(std::abs(d1) > std::abs(d2) ? v1 : v2);
23. **else** **if** (v1 > 0) f1 = sqrt(v1);    //v2小于0，则f1一定是v1的平方根
24. **else** f1\_ok = **false**;    //v1和v2都小于0，则没有得到f1
26. f0\_ok = **true**;
27. d1 = h[0] \* h[3] + h[1] \* h[4];    //式44的分母
28. d2 = h[0] \* h[0] + h[1] \* h[1] - h[3] \* h[3] - h[4] \* h[4];    //式43的分母
29. v1 = -h[2] \* h[5] / d1;    //式44
30. v2 = (h[5] \* h[5] - h[2] \* h[2]) / d2;    //式43
31. **if** (v1 < v2) std::swap(v1, v2);    //使v1不小于v2
32. //表示到底选取式44还是式43作为f0
33. **if** (v1 > 0 && v2 > 0) f0 = sqrt(std::abs(d1) > std::abs(d2) ? v1 : v2);
34. **else** **if** (v1 > 0) f0 = sqrt(v1);    //v2小于0，则f1一定是v1的开根号
35. **else** f0\_ok = **false**;    //v1和v2都小于0，则没有得到f1
36. }

构建最大生成树：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** findMaxSpanningTree(**int** num\_images, **const** vector<MatchesInfo> &pairwise\_matches,
2. Graph &span\_tree, vector<**int**> ¢ers)
3. //num\_images表示待拼接图像的数量，也是最大生成树的节点数
4. //pairwise\_matches表示图像间的拼接信息
5. //span\_tree表示最大生成树
6. //centers表示最大生成树的中心节点
7. {
8. Graph graph(num\_images);    //定义无向图G
9. vector<GraphEdge> edges;    //定义无向图G的边
11. // Construct images graph and remember its edges
12. //遍历待拼接图像，得到无向图G的边，即内点数
13. **for** (**int** i = 0; i < num\_images; ++i)
14. {
15. **for** (**int** j = 0; j < num\_images; ++j)
16. {
17. //如果图像i和j没有单应矩阵，则说明这两幅图像不重叠，不能拼接
18. **if** (pairwise\_matches[i \* num\_images + j].H.empty())
19. **continue**;
20. //得到图像i和j的内点数
21. **float** conf = **static\_cast**<**float**>(pairwise\_matches[i \* num\_images + j].num\_inliers);
22. graph.addEdge(i, j, conf);    //为G添加边
23. edges.push\_back(GraphEdge(i, j, conf));    //添加到边队列中
24. }
25. }
27. DisjointSets comps(num\_images);    //实例化DisjointSets类，表示定义一个并查集
28. span\_tree.create(num\_images);    //创建生成树
29. //表示生成树的幂，即节点间的连接数，例如某节点的幂为3，说明该节点与其他3个节点相连接
30. vector<**int**> span\_tree\_powers(num\_images, 0);
32. // Find maximum spanning tree
33. //按无向图G的边的大小（内点数）从小到大排序
34. sort(edges.begin(), edges.end(), greater<GraphEdge>());
35. **for** (**size\_t** i = 0; i < edges.size(); ++i)    //从小到大遍历无向图G的边
36. {
37. **int** comp1 = comps.findSetByElem(edges[i].from);    //得到该边的起始节点的集合
38. **int** comp2 = comps.findSetByElem(edges[i].to);    //得到该边的终止节点的集合
39. //两种不相等，说明是一个新的边，需要通过并查集添加到生成树中
40. **if** (comp1 != comp2)
41. {
42. comps.mergeSets(comp1, comp2);    //合并这两个节点
43. //为生成树添加该边
44. span\_tree.addEdge(edges[i].from, edges[i].to, edges[i].weight);
45. span\_tree.addEdge(edges[i].to, edges[i].from, edges[i].weight);
46. //节点幂的累加
47. span\_tree\_powers[edges[i].from]++;
48. span\_tree\_powers[edges[i].to]++;
49. }
50. }
52. // Find spanning tree leafs
53. vector<**int**> span\_tree\_leafs;    //表示生成树的叶节点
54. //生成树的节点的幂为1，则为叶节点
55. **for** (**int** i = 0; i < num\_images; ++i)    //遍历图像
56. **if** (span\_tree\_powers[i] == 1)    //表示该图像为叶节点
57. span\_tree\_leafs.push\_back(i);    //放入队列中
59. // Find maximum distance from each spanning tree vertex
60. vector<**int**> max\_dists(num\_images, 0);    //表示节点与叶节点的最大距离
61. vector<**int**> cur\_dists;    //表示节点与叶节点的当前距离
62. **for** (**size\_t** i = 0; i < span\_tree\_leafs.size(); ++i)    //遍历叶节点
63. {
64. cur\_dists.assign(num\_images, 0);    //初始化
65. //得到该叶节点到其他节点的距离，IncDistance表示距离的累加，即节点的累计
66. span\_tree.walkBreadthFirst(span\_tree\_leafs[i], IncDistance(cur\_dists));
67. //遍历所有节点，更新节点到叶节点的最大距离
68. **for** (**int** j = 0; j < num\_images; ++j)
69. max\_dists[j] = max(max\_dists[j], cur\_dists[j]);
70. }
72. // Find min-max distance
73. **int** min\_max\_dist = max\_dists[0];    //表示所有最大距离中的最小值
74. **for** (**int** i = 1; i < num\_images; ++i)    //遍历所有节点
75. **if** (min\_max\_dist > max\_dists[i])
76. min\_max\_dist = max\_dists[i];    //得到最大距离中的最小值
78. // Find spanning tree centers
79. centers.clear();    //表示中心节点，清零
80. **for** (**int** i = 0; i < num\_images; ++i)    //遍历所有节点
81. **if** (max\_dists[i] == min\_max\_dist)
82. centers.push\_back(i);    //保存最大距离中的最小值所对应的节点
83. //确保中心节点的数量必须大于0并小于3
84. CV\_Assert(centers.size() > 0 && centers.size() <= 2);
85. }

计算旋转矩阵：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **struct** CalcRotation
2. {
3. CalcRotation(**int** \_num\_images, **const** vector<MatchesInfo> &\_pairwise\_matches, vector<CameraParams> &\_cameras)
4. : num\_images(\_num\_images), pairwise\_matches(&\_pairwise\_matches[0]), cameras(&\_cameras[0]) {}
6. **void** operator ()(**const** GraphEdge &edge)
7. {
8. **int** pair\_idx = edge.from \* num\_images + edge.to;    //表示匹配点对的索引
9. //构造式51中的参数K0
10. Mat\_<**double**> K\_from = Mat::eye(3, 3, CV\_64F);    //初始化
11. K\_from(0,0) = cameras[edge.from].focal;    //表示式33的fu
12. //表示式33的fv
13. K\_from(1,1) = cameras[edge.from].focal \* cameras[edge.from].aspect;
14. K\_from(0,2) = cameras[edge.from].ppx;    //表示式33的cx
15. K\_from(1,2) = cameras[edge.from].ppy;    //表示式33的cy
16. //构造式51中的参数K1
17. Mat\_<**double**> K\_to = Mat::eye(3, 3, CV\_64F);    //初始化
18. K\_to(0,0) = cameras[edge.to].focal;    //表示式33的fu
19. K\_to(1,1) = cameras[edge.to].focal \* cameras[edge.to].aspect;    //表示式33的fv
20. K\_to(0,2) = cameras[edge.to].ppx;    //表示式33的cx
21. K\_to(1,2) = cameras[edge.to].ppy;    //表示式33的cy
23. Mat R = K\_from.inv() \* pairwise\_matches[pair\_idx].H.inv() \* K\_to;    //式51
24. //式52，可见CameraParams变量中R实际存储的是相机旋转矩阵变量的逆
25. cameras[edge.to].R = cameras[edge.from].R \* R;
26. }
28. **int** num\_images;    //表示待拼接图像的数量
29. **const** MatchesInfo\* pairwise\_matches;    //表示匹配图像的信息
30. CameraParams\* cameras;    //表示相机参数
31. };

下面我们给出BundleAdjusterBase类的讲解。BundleAdjusterBase类的构造函数的主要作用是为每个相机参数数量（num\_params\_per\_cam，如果是重映射误差，则num\_params\_per\_cam应为7，如果是射线发散误差，则num\_params\_per\_cam应为4）和每个内点误差数量（num\_errs\_per\_measurement，如果是重映射误差，则num\_errs\_per\_measurement应为2，即x轴和y轴的误差，如果是射线发散误差，则num\_errs\_per\_measurement应为3，即x轴、y轴和z轴的误差）赋值，并且初始化setRefinementMask（表示需要精确化的相机内参数矩阵K的掩码矩阵）、setConfThresh（表示内点阈值conf\_thresh\_，也称作置信度阈值），以及setTermCriteria（表示LM算法的迭代终止条件）。

BundleAdjusterBase类的一个重要函数是estimate，它的作用是细化相机参数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterBase::estimate(**const** vector<ImageFeatures> &features,
2. **const** vector<MatchesInfo> &pairwise\_matches,
3. vector<CameraParams> &cameras)
4. //features表示图像特征
5. //pairwise\_matches表示图像匹配信息
6. //cameras表示相机参数
7. {
8. LOG\_CHAT("Bundle adjustment");
9. #if ENABLE\_LOG
10. int64 t = getTickCount();
11. #endif
13. num\_images\_ = **static\_cast**<**int**>(features.size());    //表示待拼接图像的数量
14. features\_ = &features[0];    //图像特征变量的首地址赋值
15. pairwise\_matches\_ = &pairwise\_matches[0];    //表示匹配信息变量的首地址指针
16. //初始化LM算法所需的参数，setUpInitialCameraParams为虚函数，实际是调用BundleAdjusterBase类的子类
17. setUpInitialCameraParams(cameras);
19. // Leave only consistent image pairs
20. edges\_.clear();    //edges\_表示图像间内点数，该变量先清零
21. //只保留那些内点数大于置信度阈值的图像匹配对
22. **for** (**int** i = 0; i < num\_images\_ - 1; ++i)
23. {
24. **for** (**int** j = i + 1; j < num\_images\_; ++j)
25. {
26. //得到图像i和图像j的匹配信息
27. **const** MatchesInfo& matches\_info = pairwise\_matches\_[i \* num\_images\_ + j];
28. **if** (matches\_info.confidence > conf\_thresh\_)    //内点数大于阈值
29. edges\_.push\_back(make\_pair(i, j));    //保留这个图像匹配对
30. }
31. }
33. // Compute number of correspondences
34. total\_num\_matches\_ = 0;    //total\_num\_matches\_表示所有保留下来的内点数
35. **for** (**size\_t** i = 0; i < edges\_.size(); ++i)    //遍历图像匹配对，计算total\_num\_matches\_
36. total\_num\_matches\_ += **static\_cast**<**int**>(pairwise\_matches[edges\_[i].first \* num\_images\_ +
37. edges\_[i].second].num\_inliers);
38. //实例化CvLevMarq类，表示LM算法，CvLevMarq类构造函数的第一个参数表示需要优化变量的数量；CvLevMarq类构造函数的第二个参数表示误差数量；CvLevMarq类构造函数的第三个参数表示LM算法的迭代终止条件
39. CvLevMarq solver(num\_images\_ \* num\_params\_per\_cam\_,
40. total\_num\_matches\_ \* num\_errs\_per\_measurement\_,
41. term\_criteria\_);
43. Mat err, jac;    //err表示LM算法的误差，jac表示雅可比参数
44. CvMat matParams = cam\_params\_;    //表示相机参数矩阵
45. cvCopy(&matParams, solver.param);    //复制，初始化用于LM算法的相机参数
47. **int** iter = 0;    //表示迭代次数
48. **for**(;;)    //LM算法的迭代循环
49. {
50. **const** CvMat\* \_param = 0;    //表示LM算法计算得到的相机参数
51. CvMat\* \_jac = 0;    //表示LM计算得到的雅可比参数
52. CvMat\* \_err = 0;    //表示LM计算得到的误差
53. //调用update函数，得到相机参数\_param，即式27和式28
54. **bool** proceed = solver.update(\_param, \_jac, \_err);
56. cvCopy(\_param, &matParams);    //复制，得到相机参数
58. **if** (!proceed || !\_err)    //如果该次迭代不成功，或误差为0
59. **break**;    //退出LM算法
60. //如果\_jac不为零，则说明下次循环时，solver.update函数需要雅可比参数
61. **if** (\_jac)
62. {
63. calcJacobian(jac);    //计算雅可比参数，该函数为虚函数
64. CvMat tmp = jac;
65. cvCopy(&tmp, \_jac);    //复制
66. }
67. //如果\_err不为零，则说明下次循环时，solver.update函数需要误差参数
68. **if** (\_err)
69. {
70. calcError(err);    //计算误差，该函数为虚函数
71. LOG\_CHAT(".");
72. iter++;    //累计迭代次数
73. CvMat tmp = err;
74. cvCopy(&tmp, \_err);    //复制
75. }
76. }
77. //终端输出信息
78. LOGLN\_CHAT("");
79. LOGLN\_CHAT("Bundle adjustment, final RMS error: " << sqrt(err.dot(err) / total\_num\_matches\_));
80. LOGLN\_CHAT("Bundle adjustment, iterations done: " << iter);
81. //LM算法结束后，得到最终的精确的相机参数，该函数为虚函数
82. obtainRefinedCameraParams(cameras);
84. // Normalize motion to center image
85. //利用精确的相机参数，再次计算基于中心图像的其他图像的相对旋转矩阵
86. Graph span\_tree;
87. vector<**int**> span\_tree\_centers;
88. //利用最大生成树算法计算中心图像
89. findMaxSpanningTree(num\_images\_, pairwise\_matches, span\_tree, span\_tree\_centers);
90. //前面我们介绍过，相机旋转矩阵CameraParams.R在前面的程序中其实是旋转矩阵的逆矩阵，现在让CameraParams.R是基于中心图像的相对旋转矩阵
91. Mat R\_inv = cameras[span\_tree\_centers[0]].R.inv();    //中心图像的旋转矩阵
92. **for** (**int** i = 0; i < num\_images\_; ++i)    //得到其他图像的相对旋转矩阵
93. cameras[i].R = R\_inv \* cameras[i].R;
94. //终端输出信息
95. LOGLN\_CHAT("Bundle adjustment, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
96. }

BundleAdjusterBase类有两个子类——BundleAdjusterReproj和BundleAdjusterRay，它们分别表示光束平差法的两个实现方法——重映射方法和射线发散方法，也就是误差指标函数的两种形式。

下面我们先给出BundleAdjusterReproj类。setUpInitialCameraParams函数表示初始化相机参数，即在LM算法迭代之前，要赋予相机参数初值：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterReproj::setUpInitialCameraParams(**const** vector<CameraParams> &cameras)
2. //cameras表示相机参数的初始化值
3. {
4. //定义表示待精确化的所有参数的矩阵大小
5. cam\_params\_.create(num\_images\_ \* 7, 1, CV\_64F);
6. SVD svd;
7. //遍历所有的图像（即所有相机），初始化相机参数
8. **for** (**int** i = 0; i < num\_images\_; ++i)
9. {
10. cam\_params\_.at<**double**>(i \* 7, 0) = cameras[i].focal;    //初始化fu
11. cam\_params\_.at<**double**>(i \* 7 + 1, 0) = cameras[i].ppx;    //初始化cx
12. cam\_params\_.at<**double**>(i \* 7 + 2, 0) = cameras[i].ppy;    //初始化cy
13. cam\_params\_.at<**double**>(i \* 7 + 3, 0) = cameras[i].aspect;    //初始化α
14. //表示得到满足正交关系的旋转矩阵R
15. svd(cameras[i].R, SVD::FULL\_UV);
16. Mat R = svd.u \* svd.vt;
17. **if** (determinant(R) < 0)
18. R \*= -1;
20. Mat rvec;
21. Rodrigues(R, rvec);    //旋转矩阵R由Rodrigues算法得到旋转向量rvec
22. CV\_Assert(rvec.type() == CV\_32F);
23. cam\_params\_.at<**double**>(i \* 7 + 4, 0) = rvec.at<**float**>(0, 0);    //初始化rx
24. cam\_params\_.at<**double**>(i \* 7 + 5, 0) = rvec.at<**float**>(1, 0);    //初始化ry
25. cam\_params\_.at<**double**>(i \* 7 + 6, 0) = rvec.at<**float**>(2, 0);    //初始化rz
26. }
27. }

obtainRefinedCameraParams函数表示得到最终的精确的相机参数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterReproj::obtainRefinedCameraParams(vector<CameraParams> &cameras) **const**
2. //cameras表示最终得到的相机参数
3. {
4. **for** (**int** i = 0; i < num\_images\_; ++i)    //遍历所有的图像（即所有相机），得到相机参数
5. {
6. cameras[i].focal = cam\_params\_.at<**double**>(i \* 7, 0);    //得到fu
7. cameras[i].ppx = cam\_params\_.at<**double**>(i \* 7 + 1, 0);   //得到cx
8. cameras[i].ppy = cam\_params\_.at<**double**>(i \* 7 + 2, 0);    //得到cy
9. cameras[i].aspect = cam\_params\_.at<**double**>(i \* 7 + 3, 0);    //得到α
11. Mat rvec(3, 1, CV\_64F);
12. rvec.at<**double**>(0, 0) = cam\_params\_.at<**double**>(i \* 7 + 4, 0);    //得到rx
13. rvec.at<**double**>(1, 0) = cam\_params\_.at<**double**>(i \* 7 + 5, 0);    //得到ry
14. rvec.at<**double**>(2, 0) = cam\_params\_.at<**double**>(i \* 7 + 6, 0);    //得到rz
15. Rodrigues(rvec, cameras[i].R);    //旋转向量rvec由Rodrigues算法得到旋转矩阵R
17. Mat tmp;
18. cameras[i].R.convertTo(tmp, CV\_32F);    //变换数据类型
19. cameras[i].R = tmp;    //赋值
20. }
21. }

calcError函数用于计算误差，即式25：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterReproj::calcError(Mat &err)
2. //err表示计算得到的误差
3. {
4. err.create(total\_num\_matches\_ \* 2, 1, CV\_64F);    //定义误差矩阵
6. **int** match\_idx = 0;    //表示重映射误差的索引
7. //遍历最大生成树的边
8. **for** (**size\_t** edge\_idx = 0; edge\_idx < edges\_.size(); ++edge\_idx)
9. {
10. **int** i = edges\_[edge\_idx].first;    //表示边的始端图像
11. **int** j = edges\_[edge\_idx].second;    //表示边的终端图像
12. **double** f1 = cam\_params\_.at<**double**>(i \* 7, 0);    //始端图像的fu
13. **double** f2 = cam\_params\_.at<**double**>(j \* 7, 0);    //终端图像的fu
14. **double** ppx1 = cam\_params\_.at<**double**>(i \* 7 + 1, 0);    //始端图像的cx
15. **double** ppx2 = cam\_params\_.at<**double**>(j \* 7 + 1, 0);    //终端图像的cx
16. **double** ppy1 = cam\_params\_.at<**double**>(i \* 7 + 2, 0);    //始端图像的cy
17. **double** ppy2 = cam\_params\_.at<**double**>(j \* 7 + 2, 0);    //终端图像的cy
18. **double** a1 = cam\_params\_.at<**double**>(i \* 7 + 3, 0);    //始端图像的α
19. **double** a2 = cam\_params\_.at<**double**>(j \* 7 + 3, 0);    //终端图像的α
21. **double** R1[9];
22. Mat R1\_(3, 3, CV\_64F, R1);
23. Mat rvec(3, 1, CV\_64F);
24. rvec.at<**double**>(0, 0) = cam\_params\_.at<**double**>(i \* 7 + 4, 0);    //始端图像的rx
25. rvec.at<**double**>(1, 0) = cam\_params\_.at<**double**>(i \* 7 + 5, 0);    //始端图像的ry
26. rvec.at<**double**>(2, 0) = cam\_params\_.at<**double**>(i \* 7 + 6, 0);    //始端图像的rz
27. //旋转向量rvec由Rodrigues算法得到始端图像的旋转矩阵R
28. Rodrigues(rvec, R1\_);
30. **double** R2[9];
31. Mat R2\_(3, 3, CV\_64F, R2);
32. rvec.at<**double**>(0, 0) = cam\_params\_.at<**double**>(j \* 7 + 4, 0);    //终端图像的rx
33. rvec.at<**double**>(1, 0) = cam\_params\_.at<**double**>(j \* 7 + 5, 0);    //终端图像的ry
34. rvec.at<**double**>(2, 0) = cam\_params\_.at<**double**>(j \* 7 + 6, 0);    //终端图像的rz
35. //旋转向量rvec由Rodrigues算法得到终端图像的旋转矩阵R
36. Rodrigues(rvec, R2\_);
38. **const** ImageFeatures& features1 = features\_[i];    //得到始端图像的特征
39. **const** ImageFeatures& features2 = features\_[j];    //得到终端图像的特征
40. //得到两者的图像匹配信息
41. **const** MatchesInfo& matches\_info = pairwise\_matches\_[i \* num\_images\_ + j];
42. //为始端图像的相机内参数矩阵K赋值
43. Mat\_<**double**> K1 = Mat::eye(3, 3, CV\_64F);
44. K1(0,0) = f1; K1(0,2) = ppx1;
45. K1(1,1) = f1\*a1; K1(1,2) = ppy1;
46. //为终端图像的相机内参数矩阵K赋值
47. Mat\_<**double**> K2 = Mat::eye(3, 3, CV\_64F);
48. K2(0,0) = f2; K2(0,2) = ppx2;
49. K2(1,1) = f2\*a2; K2(1,2) = ppy2;
50. //得到两者的相对单应矩阵H，式38
51. Mat\_<**double**> H = K2 \* R2\_.inv() \* R1\_ \* K1.inv();
53. **for** (**size\_t** k = 0; k < matches\_info.matches.size(); ++k)    //遍历匹配点对
54. {
55. **if** (!matches\_info.inliers\_mask[k])    //表示不是内点
56. **continue**;
58. **const** DMatch& m = matches\_info.matches[k];    //表示内点信息
59. //表示内点点对
60. Point2f p1 = features1.keypoints[m.queryIdx].pt;
61. Point2f p2 = features2.keypoints[m.trainIdx].pt;
62. //由单应矩阵得到p1点的三维空间坐标
63. **double** x = H(0,0)\*p1.x + H(0,1)\*p1.y + H(0,2);
64. **double** y = H(1,0)\*p1.x + H(1,1)\*p1.y + H(1,2);
65. **double** z = H(2,0)\*p1.x + H(2,1)\*p1.y + H(2,2);
66. //把p1点的三维空间坐标重新映射到p2点所在平面，并比较两者的差，即重映射误差
67. err.at<**double**>(2 \* match\_idx, 0) = p2.x - x/z;
68. err.at<**double**>(2 \* match\_idx + 1, 0) = p2.y - y/z;
69. match\_idx++;    //累加匹配索引值
70. }
71. }
72. }

calcJacobian函数用于计算雅可比矩阵，即式59：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterReproj::calcJacobian(Mat &jac)
2. //jac表示得到的雅可比矩阵
3. {
4. //定义雅可比矩阵的大小
5. jac.create(total\_num\_matches\_ \* 2, num\_images\_ \* 7, CV\_64F);
6. jac.setTo(0);    //清零
8. **double** val;
9. **const** **double** step = 1e-4;    //表示步长，即式60中的∆
11. **for** (**int** i = 0; i < num\_images\_; ++i)    //遍历所有的匹配图像，即式59的所有行
12. {
13. **if** (refinement\_mask\_.at<uchar>(0, 0))    //计算雅可比矩阵的fu项
14. {
15. val = cam\_params\_.at<**double**>(i \* 7, 0);    //提取fu
16. cam\_params\_.at<**double**>(i \* 7, 0) = val - step;    //fu－∆
17. calcError(err1\_);    //计算因fu变换而引起的误差
18. cam\_params\_.at<**double**>(i \* 7, 0) = val + step;    //fu＋∆
19. calcError(err2\_);    //计算因fu变换而引起的误差
20. calcDeriv(err1\_, err2\_, 2 \* step, jac.col(i \* 7));    //式60
21. cam\_params\_.at<**double**>(i \* 7, 0) = val;    //重新赋值fu，以备其他参数计算
22. }
23. **if** (refinement\_mask\_.at<uchar>(0, 2))    //计算雅可比矩阵的cx项
24. {
25. val = cam\_params\_.at<**double**>(i \* 7 + 1, 0);    //提取cx
26. cam\_params\_.at<**double**>(i \* 7 + 1, 0) = val - step;    //cx－∆
27. calcError(err1\_);    //计算因cx变换而引起的误差
28. cam\_params\_.at<**double**>(i \* 7 + 1, 0) = val + step;    //cx＋∆
29. calcError(err2\_);    //计算因cx变换而引起的误差
30. calcDeriv(err1\_, err2\_, 2 \* step, jac.col(i \* 7 + 1));    //式60
31. cam\_params\_.at<**double**>(i \* 7 + 1, 0) = val;    //重新赋值cx，以备其他参数计算
32. }
33. **if** (refinement\_mask\_.at<uchar>(1, 2))    //计算雅可比矩阵的cy项
34. {
35. val = cam\_params\_.at<**double**>(i \* 7 + 2, 0);    //提取cy
36. cam\_params\_.at<**double**>(i \* 7 + 2, 0) = val - step;    //cy－∆
37. calcError(err1\_);    //计算因cy变换而引起的误差
38. cam\_params\_.at<**double**>(i \* 7 + 2, 0) = val + step;    //cy＋∆
39. calcError(err2\_);    //计算因cy变换而引起的误差
40. calcDeriv(err1\_, err2\_, 2 \* step, jac.col(i \* 7 + 2));    //式60
41. cam\_params\_.at<**double**>(i \* 7 + 2, 0) = val;    //重新赋值cy，以备其他参数计算
42. }
43. **if** (refinement\_mask\_.at<uchar>(1, 1))    //计算雅可比矩阵的α项
44. {
45. val = cam\_params\_.at<**double**>(i \* 7 + 3, 0);    //提取α
46. cam\_params\_.at<**double**>(i \* 7 + 3, 0) = val - step;    //α－∆
47. calcError(err1\_);    //计算因α变换而引起的误差
48. cam\_params\_.at<**double**>(i \* 7 + 3, 0) = val + step;    //α＋∆
49. calcError(err2\_);    //计算因α变换而引起的误差
50. calcDeriv(err1\_, err2\_, 2 \* step, jac.col(i \* 7 + 3));    //式60
51. cam\_params\_.at<**double**>(i \* 7 + 3, 0) = val;    //重新赋值α，以备其他参数计算
52. }
53. **for** (**int** j = 4; j < 7; ++j)    //计算rx, ry, rz
54. {
55. val = cam\_params\_.at<**double**>(i \* 7 + j, 0);
56. cam\_params\_.at<**double**>(i \* 7 + j, 0) = val - step;
57. calcError(err1\_);
58. cam\_params\_.at<**double**>(i \* 7 + j, 0) = val + step;
59. calcError(err2\_);
60. calcDeriv(err1\_, err2\_, 2 \* step, jac.col(i \* 7 + j));
61. cam\_params\_.at<**double**>(i \* 7 + j, 0) = val;
62. }
63. }
64. }

下面我们先给出BundleAdjusterRay类：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterRay::setUpInitialCameraParams(**const** vector<CameraParams> &cameras)
2. {
3. //定义表示待精确化的所有参数的矩阵大小
4. cam\_params\_.create(num\_images\_ \* 4, 1, CV\_64F);
5. SVD svd;
6. //遍历所有的图像（即所有相机），初始化相机参数
7. **for** (**int** i = 0; i < num\_images\_; ++i)
8. {
9. cam\_params\_.at<**double**>(i \* 4, 0) = cameras[i].focal;    //初始化fu
10. //表示得到满足正交关系的旋转矩阵R
11. svd(cameras[i].R, SVD::FULL\_UV);
12. Mat R = svd.u \* svd.vt;
13. **if** (determinant(R) < 0)
14. R \*= -1;
16. Mat rvec;
17. Rodrigues(R, rvec);    //旋转矩阵R由Rodrigues算法得到旋转向量rvec
18. CV\_Assert(rvec.type() == CV\_32F);
19. cam\_params\_.at<**double**>(i \* 4 + 1, 0) = rvec.at<**float**>(0, 0);    //初始化rx
20. cam\_params\_.at<**double**>(i \* 4 + 2, 0) = rvec.at<**float**>(1, 0);    //初始化ry
21. cam\_params\_.at<**double**>(i \* 4 + 3, 0) = rvec.at<**float**>(2, 0);    //初始化rz
22. }
23. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterRay::obtainRefinedCameraParams(vector<CameraParams> &cameras) **const**
2. {
3. **for** (**int** i = 0; i < num\_images\_; ++i)    //遍历所有的图像（即所有相机），得到相机参数
4. {
5. cameras[i].focal = cam\_params\_.at<**double**>(i \* 4, 0);    //得到fu
7. Mat rvec(3, 1, CV\_64F);
8. rvec.at<**double**>(0, 0) = cam\_params\_.at<**double**>(i \* 4 + 1, 0);    //得到rx
9. rvec.at<**double**>(1, 0) = cam\_params\_.at<**double**>(i \* 4 + 2, 0);    //得到ry
10. rvec.at<**double**>(2, 0) = cam\_params\_.at<**double**>(i \* 4 + 3, 0);    //得到rz
11. Rodrigues(rvec, cameras[i].R);    //旋转向量rvec由Rodrigues算法得到旋转矩阵R
13. Mat tmp;
14. cameras[i].R.convertTo(tmp, CV\_32F);    //变换数据类型
15. cameras[i].R = tmp;    //赋值
16. }
17. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterRay::calcError(Mat &err)
2. {
3. err.create(total\_num\_matches\_ \* 3, 1, CV\_64F);    //定义误差矩阵
5. **int** match\_idx = 0;    //表示重映射误差的索引
6. //遍历最大生成树的边
7. **for** (**size\_t** edge\_idx = 0; edge\_idx < edges\_.size(); ++edge\_idx)
8. {
9. **int** i = edges\_[edge\_idx].first;    //表示边的始端图像
10. **int** j = edges\_[edge\_idx].second;    //表示边的终端图像
11. **double** f1 = cam\_params\_.at<**double**>(i \* 4, 0);    //始端图像的fu
12. **double** f2 = cam\_params\_.at<**double**>(j \* 4, 0);    //终端图像的fu
14. **double** R1[9];
15. Mat R1\_(3, 3, CV\_64F, R1);
16. Mat rvec(3, 1, CV\_64F);
17. rvec.at<**double**>(0, 0) = cam\_params\_.at<**double**>(i \* 4 + 1, 0);    //始端图像的rx
18. rvec.at<**double**>(1, 0) = cam\_params\_.at<**double**>(i \* 4 + 2, 0);    //始端图像的ry
19. rvec.at<**double**>(2, 0) = cam\_params\_.at<**double**>(i \* 4 + 3, 0);    //始端图像的rz
20. //旋转向量rvec由Rodrigues算法得到始端图像的旋转矩阵R
21. Rodrigues(rvec, R1\_);
23. **double** R2[9];
24. Mat R2\_(3, 3, CV\_64F, R2);
25. rvec.at<**double**>(0, 0) = cam\_params\_.at<**double**>(j \* 4 + 1, 0);    //终端图像的rx
26. rvec.at<**double**>(1, 0) = cam\_params\_.at<**double**>(j \* 4 + 2, 0);    //终端图像的ry
27. rvec.at<**double**>(2, 0) = cam\_params\_.at<**double**>(j \* 4 + 3, 0);    //终端图像的rz
28. //旋转向量rvec由Rodrigues算法得到终端图像的旋转矩阵R
29. Rodrigues(rvec, R2\_);
31. **const** ImageFeatures& features1 = features\_[i];    //得到始端图像的特征
32. **const** ImageFeatures& features2 = features\_[j];    //得到终端图像的特征
33. //得到两者的图像匹配信息
34. **const** MatchesInfo& matches\_info = pairwise\_matches\_[i \* num\_images\_ + j];
35. //为始端图像的相机内参数矩阵K赋值
36. Mat\_<**double**> K1 = Mat::eye(3, 3, CV\_64F);
37. K1(0,0) = f1; K1(0,2) = features1.img\_size.width \* 0.5;
38. K1(1,1) = f1; K1(1,2) = features1.img\_size.height \* 0.5;
39. //为终端图像的相机内参数矩阵K赋值
40. Mat\_<**double**> K2 = Mat::eye(3, 3, CV\_64F);
41. K2(0,0) = f2; K2(0,2) = features2.img\_size.width \* 0.5;
42. K2(1,1) = f2; K2(1,2) = features2.img\_size.height \* 0.5;
43. //计算两个单应矩阵
44. Mat\_<**double**> H1 = R1\_ \* K1.inv();
45. Mat\_<**double**> H2 = R2\_ \* K2.inv();
47. **for** (**size\_t** k = 0; k < matches\_info.matches.size(); ++k)    //遍历匹配点对
48. {
49. **if** (!matches\_info.inliers\_mask[k])    //表示不是内点
50. **continue**;
52. **const** DMatch& m = matches\_info.matches[k];    //表示内点信息
54. Point2f p1 = features1.keypoints[m.queryIdx].pt;    //表示内点的一个点
55. //由单应矩阵得到p1点的三维空间坐标
56. **double** x1 = H1(0,0)\*p1.x + H1(0,1)\*p1.y + H1(0,2);
57. **double** y1 = H1(1,0)\*p1.x + H1(1,1)\*p1.y + H1(1,2);
58. **double** z1 = H1(2,0)\*p1.x + H1(2,1)\*p1.y + H1(2,2);
59. **double** len = sqrt(x1\*x1 + y1\*y1 + z1\*z1);    //式62
60. x1 /= len; y1 /= len; z1 /= len;    //式63
62. Point2f p2 = features2.keypoints[m.trainIdx].pt;    //表示内点的另一个点
63. //由单应矩阵得到p2点的三维空间坐标
64. **double** x2 = H2(0,0)\*p2.x + H2(0,1)\*p2.y + H2(0,2);
65. **double** y2 = H2(1,0)\*p2.x + H2(1,1)\*p2.y + H2(1,2);
66. **double** z2 = H2(2,0)\*p2.x + H2(2,1)\*p2.y + H2(2,2);
67. len = sqrt(x2\*x2 + y2\*y2 + z2\*z2);    //式62
68. x2 /= len; y2 /= len; z2 /= len;    //式63
70. **double** mult = sqrt(f1 \* f2);    //式64的根号内的部分
71. //式64
72. err.at<**double**>(3 \* match\_idx, 0) = mult \* (x1 - x2);
73. err.at<**double**>(3 \* match\_idx + 1, 0) = mult \* (y1 - y2);
74. err.at<**double**>(3 \* match\_idx + 2, 0) = mult \* (z1 - z2);
76. match\_idx++;    //累计
77. }
78. }
79. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** BundleAdjusterRay::calcJacobian(Mat &jac)
2. {
3. //定义雅可比矩阵的大小，式65
4. jac.create(total\_num\_matches\_ \* 3, num\_images\_ \* 4, CV\_64F);
6. **double** val;
7. **const** **double** step = 1e-3;    //表示步长，即式60中的∆
9. **for** (**int** i = 0; i < num\_images\_; ++i)    //遍历所有的匹配图像，即式65的所有行
10. {
11. **for** (**int** j = 0; j < 4; ++j)    //遍历4个待精确的参数fu, rx, ry, rz
12. {
13. val = cam\_params\_.at<**double**>(i \* 4 + j, 0);
14. cam\_params\_.at<**double**>(i \* 4 + j, 0) = val - step;
15. calcError(err1\_);
16. cam\_params\_.at<**double**>(i \* 4 + j, 0) = val + step;
17. calcError(err2\_);
18. calcDeriv(err1\_, err2\_, 2 \* step, jac.col(i \* 4 + j));
19. cam\_params\_.at<**double**>(i \* 4 + j, 0) = val;
20. }
21. }
22. }

下面给出波形校正函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. **void** waveCorrect(vector<Mat> &rmats, WaveCorrectKind kind)
2. //rmats表示所有相机的旋转矩阵
3. //kind表示波形校正的方式，是水平校正（WAVE\_CORRECT\_HORIZ）还是垂直校正（WAVE\_CORRECT\_VERT）
4. {
5. LOGLN("Wave correcting...");
6. #if ENABLE\_LOG
7. int64 t = getTickCount();    //用于计时
8. #endif
9. //在前面已经分析过，程序中的旋转矩阵其实是公式中旋转矩阵的逆
10. Mat moment = Mat::zeros(3, 3, CV\_32F);    //表示矩量矩阵
11. **for** (**size\_t** i = 0; i < rmats.size(); ++i)    //遍历所有的R，得到矩量矩阵
12. {
13. Mat col = rmats[i].col(0);    //提取旋转矩阵的第一列
14. moment += col \* col.t();    //得到式67的方括号内的部分
15. }
16. Mat eigen\_vals, eigen\_vecs;    //表示特征值和特征向量
17. eigen(moment, eigen\_vals, eigen\_vecs);    //计算矩量矩阵的特征向量
19. Mat rg1;    //表示校正矩阵的第二列，即式67的q1
20. **if** (kind == WAVE\_CORRECT\_HORIZ)    //水平校正
21. rg1 = eigen\_vecs.row(2).t();    //最小特征值对应的特征向量
22. **else** **if** (kind == WAVE\_CORRECT\_VERT)    //垂直校正
23. rg1 = eigen\_vecs.row(0).t();    //最大特征值对应的特征向量
24. **else**    //其他情况
25. CV\_Error(CV\_StsBadArg, "unsupported kind of wave correction");
27. Mat img\_k = Mat::zeros(3, 1, CV\_32F);
28. **for** (**size\_t** i = 0; i < rmats.size(); ++i)
29. img\_k += rmats[i].col(2);    //计算∑r2
30. Mat rg0 = rg1.cross(img\_k);    //得到校正矩阵的第一列
31. rg0 /= norm(rg0);    //归一化处理，式68
33. Mat rg2 = rg0.cross(rg1);    //得到校正矩阵的第三列，式69
35. **double** conf = 0;    //表示置信度
36. //根据置信度，调整校正矩阵的第一列和第二列
37. **if** (kind == WAVE\_CORRECT\_HORIZ)    //水平校正
38. {
39. **for** (**size\_t** i = 0; i < rmats.size(); ++i)
40. conf += rg0.dot(rmats[i].col(0));
41. **if** (conf < 0)
42. {
43. rg0 \*= -1;
44. rg1 \*= -1;
45. }
46. }
47. **else** **if** (kind == WAVE\_CORRECT\_VERT)    //垂直校正
48. {
49. **for** (**size\_t** i = 0; i < rmats.size(); ++i)
50. conf -= rg1.dot(rmats[i].col(0));
51. **if** (conf < 0)
52. {
53. rg0 \*= -1;
54. rg1 \*= -1;
55. }
56. }
58. Mat R = Mat::zeros(3, 3, CV\_32F);    //表示校正矩阵
59. //构造校正矩阵
60. Mat tmp = R.row(0);
61. Mat(rg0.t()).copyTo(tmp);
62. tmp = R.row(1);
63. Mat(rg1.t()).copyTo(tmp);
64. tmp = R.row(2);
65. Mat(rg2.t()).copyTo(tmp);
67. **for** (**size\_t** i = 0; i < rmats.size(); ++i)    //遍历所有的R
68. rmats[i] = R \* rmats[i];    //波形校正
70. LOGLN("Wave correcting, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
71. }

在图像拼接时，有时会出现输入的图像不属于全景图像的时候，因此我们还需要应用leaveBiggestComponent函数把这些图像剔除掉，只保留全景图像集合。应用并查集可以很方便的得到全景图像集合，而图像间能否拼接，依据的是匹配置信度c值的大小：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. vector<**int**> leaveBiggestComponent(vector<ImageFeatures> &features,  vector<MatchesInfo> &pairwise\_matches,
2. **float** conf\_threshold)
3. //features表示图像特征
4. //pairwise\_matches表示匹配点对的信息
5. //conf\_threshold表示匹配置信度c，即式23
6. //返回能够拼接在一起的全景图像集合的索引
7. {
8. **const** **int** num\_images = **static\_cast**<**int**>(features.size());    //得到待拼接图像的数量
10. DisjointSets comps(num\_images);    //实例化DisjointSets类，表示定义一个并查集
11. //遍历待拼接图像对，得到能够拼接在一起的所有图像
12. **for** (**int** i = 0; i < num\_images; ++i)
13. {
14. **for** (**int** j = 0; j < num\_images; ++j)
15. {
16. //如果匹配置信度过小，则继续下一次循环，即图像i和图像j无法拼接
17. **if** (pairwise\_matches[i\*num\_images + j].confidence < conf\_threshold)
18. **continue**;
19. **int** comp1 = comps.findSetByElem(i);    //提取出第i幅图像的所在集合
20. **int** comp2 = comps.findSetByElem(j);    //提取出第j幅图像的所在集合
21. **if** (comp1 != comp2)    //表示图像i和图像j不属于同一集合
22. comps.mergeSets(comp1, comp2);    //合并两个集合
23. }
24. }
25. //得到元素最多的那个集合，即全景图像集合，max\_comp表示该集合内元素（即图像）的数量
26. **int** max\_comp = **static\_cast**<**int**>(max\_element(comps.size.begin(), comps.size.end()) - comps.size.begin());
27. //表示我们所需要的全景图像集合的元素索引，即max\_comp所表示的那个集合
28. vector<**int**> indices;
29. vector<**int**> indices\_removed;    //表示不是全景图像集合的元素索引
30. **for** (**int** i = 0; i < num\_images; ++i)    //遍历所有图像
31. **if** (comps.findSetByElem(i) == max\_comp)    //得到全景图像集合
32. indices.push\_back(i);    //得到图像索引
33. **else**
34. indices\_removed.push\_back(i);    //不是全景图像集合的元素索引
36. vector<ImageFeatures> features\_subset;    //表示特征子集
37. vector<MatchesInfo> pairwise\_matches\_subset;    //表示匹配子集
38. //遍历全景图像集合，得到特征子集和匹配子集
39. **for** (**size\_t** i = 0; i < indices.size(); ++i)
40. {
41. features\_subset.push\_back(features[indices[i]]);    //得到特征子集
42. **for** (**size\_t** j = 0; j < indices.size(); ++j)    //得到匹配子集
43. {
44. pairwise\_matches\_subset.push\_back(pairwise\_matches[indices[i]\*num\_images + indices[j]]);
45. pairwise\_matches\_subset.back().src\_img\_idx = **static\_cast**<**int**>(i);
46. pairwise\_matches\_subset.back().dst\_img\_idx = **static\_cast**<**int**>(j);
47. }
48. }
49. //如果子集包括了所有图像，则直接退出，无需再执行下去
50. **if** (**static\_cast**<**int**>(features\_subset.size()) == num\_images)
51. **return** indices;    //返回索引值
52. //终端输出
53. LOG("Removed some images, because can't match them or there are too similar images: (");
54. LOG(indices\_removed[0] + 1);
55. **for** (**size\_t** i = 1; i < indices\_removed.size(); ++i)
56. LOG(", " << indices\_removed[i]+1);
57. LOGLN(").");
58. LOGLN("Try to decrease the match confidence threshold and/or check if you're stitching duplicates.");
59. //特征和匹配信息重新赋值
60. features = features\_subset;
61. pairwise\_matches = pairwise\_matches\_subset;
63. **return** indices;    //返回索引值
64. }

3.3 应用

下面给出相机参数评估部分的应用，得到的参数在终端内输出：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78809143) [copy](http://blog.csdn.net/zhaocj/article/details/78809143)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. #include <iomanip>
23. **using** **namespace** cv;
24. **using** **namespace** std;
25. **using** **namespace** detail;
27. **int** main(**int** argc, **char**\*\* argv)
28. {
30. vector<Mat> imgs;    //得到待拼接图像队列
31. Mat img = imread("1.jpg");
32. imgs.push\_back(img);
33. img = imread("2.jpg");
34. imgs.push\_back(img);
36. Ptr<FeaturesFinder> finder;    //特征检测
37. finder = **new** SurfFeaturesFinder();
38. vector<ImageFeatures> features(2);
39. (\*finder)(imgs[0], features[0]);
40. (\*finder)(imgs[1], features[1]);
42. vector<MatchesInfo> pairwise\_matches;    //特征匹配
43. BestOf2NearestMatcher matcher(**false**, 0.3f, 6, 6);
44. matcher(features, pairwise\_matches);
46. HomographyBasedEstimator estimator;    //定义参数评估器
47. vector<CameraParams> cameras;    //表示相机参数矢量队列
48. estimator(features, pairwise\_matches, cameras);    //相机参数评估
50. cout<<"相机参数的初次评估："<<endl;    //终端输出
51. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
52. {
53. Mat R;
54. cameras[i].R.convertTo(R, CV\_32F);    //数据类型转换
55. cameras[i].R = R;
56. cout<<"第"<<i+1<<"个相机的内参数"<< ":\n" << cameras[i].K()<<endl;
57. cout<<"第"<<i+1<<"个相机的旋转参数"<< ":\n" << R<<endl;
58. cout<<"第"<<i+1<<"个相机的焦距"<< ":\n" << cameras[i].focal<<endl;
59. }
61. Ptr<detail::BundleAdjusterBase> adjuster;    //光束平差法，精确相机参数
62. adjuster = **new** detail::BundleAdjusterReproj();    //重映射误差方法
63. //adjuster = new detail::BundleAdjusterRay();    //射线发散误差方法
65. adjuster->setConfThresh(1);    //设置匹配置信度，该值为1
66. (\*adjuster)(features, pairwise\_matches, cameras);    //相机参数的精确评估
68. cout<<"相机参数的精确评估："<<endl;    //终端输出
69. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
70. {
71. cout<<"第"<<i+1<<"个相机的内参数"<< ":\n" << cameras[i].K()<<endl;
72. cout<<"第"<<i+1<<"个相机的旋转参数"<< ":\n" << cameras[i].R<<endl;
73. }
75. vector<Mat> rmats;
76. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
77. rmats.push\_back(cameras[i].R.clone());    //复制相机的旋转参数
78. waveCorrect(rmats, WAVE\_CORRECT\_HORIZ);    //进行波形校正
79. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
80. cameras[i].R = rmats[i];    //赋值
82. cout<<"波形校正："<<endl;    //终端输出
83. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
84. {
85. cout<<"第"<<i+1<<"个相机的旋转参数"<< ":\n" << cameras[i].R<<endl;
86. }
87. **return** 0;
88. }

最后的终端输出为：

相机参数的初次评估：

第1个相机的内参数:

[8828.962705830743,  0,  1632;

  0,  8828.962705830743,  916;

  0,  0,  1]

第1个相机的旋转参数:

[1,  0, 0;

  0,  1,  0;

  0,  0,  1]

第1个相机的焦距：

8828.96

第2个相机的内参数:

[8828.962705830743,  0,  1632;

  0,  8828.962705830743,  916;

  0,  0,  1]

第2个相机的旋转参数:

[0.96937662,  -0.010653165,  0.2045967;

 0.014337449,  0.9847675,  0.016997401;

 -0.1385226,  -0.023688598,  0.97040582]

第2个相机的焦距：

8828.96

相机参数的精确评估：

第1个相机的内参数:

[9870.390441988358,  0,  1741.665818978845;

  0,  12324.5895921202,  916.6138494133298;

  0,  0,  1]

第1个相机的旋转参数:

[1,  -2.066372e-009, 0;

 1.6589183e-009,  1,  0;

  0,  0,  1]

第2个相机的内参数:

[9869.984266558649,  0,  1516.680944868967;

  0,  12323.20985076497,  862.0564899921652;

  0,  0,  1]

第2个相机的旋转参数:

[0.98646808,  -0.012547051,  0.16347259;

 0.011426224,  0.99990433,  0.0077948803;

 -0.16355476,  -0.0058215279,  0.98651713]

波形校正：

第1个相机的旋转参数:

[0.99661297,  0.0057310974,  -0.082034826;

  -1.0345258e-010, 0.99756861,  0.069691904;

 0.082234778,  -0.069455855,  0.9941898]

第2个相机的旋转参数:

[0.99660957,  -0.006296434,  0.082034826;

 -9.3132257e-010,  0.99706745,  0.076528184;

 -0.082276106,  -0.07626871,  0.99368697]

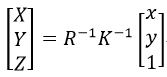
# Opencv2.4.9源码分析——Stitching（四）

4、图像投影变换

4.1 原理

前文我们已经说过，每幅图像是相机在不同角度下拍摄得到的，它们并不在同一个投影平面上，如果对重叠部分直接进行拼接，则会破坏实际场景的视觉一致性。所以我们需要在拼接之前，对图像进行投影变换，即对图像进行扭曲变形。

设图像中某像素点的二维坐标为(x, y)，它所对应的世界坐标为(X, Y, Z)，两者之间的关系为：

（70）

式中，R为旋转矩阵，K为相机的内参数矩阵。像素点可以映射到不同的表面上，最简单的是映射到平面上，设(u, v)为映射后的二维坐标，则

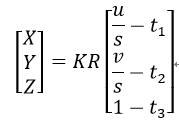
http://img.blog.csdn.net/20171218093209232（71）

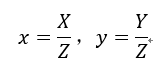
如果世界坐标有平移，并且投影图像有尺度的变化，则式71改写为：

http://img.blog.csdn.net/20171218093217543（72）

式中，s表示尺度，与相机焦距成正比，t1、t2和t3表示三个坐标轴的位移。

上面的变换是由源图像变换到投影图像上，即由(x, y)映射为(u, v)，我们称为正向投影。如果是由投影图像变换为源图像，我们称为反向投影。反向投影的公式为：

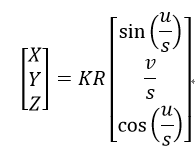
（73）

（74）

平面投影简单，但拼接图像较多时，视觉效果并不好。另一种常见的投影方式是柱面投影。柱面投影是以相机为圆柱中心点，相机焦距为半径的一个柱面作为投影面。它的投影图像与投影到的圆柱表面的位置无关，柱面全景图像可在水平方向上满足360度环视，具有较好的视觉效果，并且柱面投影也符合我们对相机位置的假设（相机只做旋转动作）。柱面投影后的坐标为：

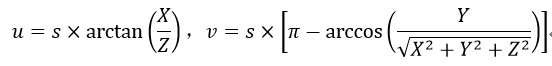
http://img.blog.csdn.net/20171218093246893（75）

柱面投影的反向映射关系为：

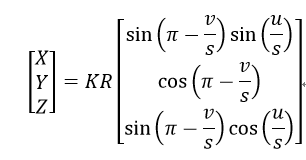
（76）

由(X, Y, Z)得到(x, y)的公式也是式74。

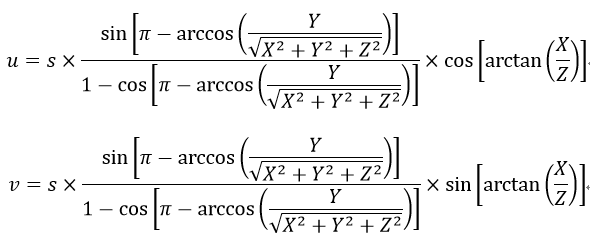
球面投影是将图像序列投影到以一点为坐标中心的球面上。人的眼睛在看东西时的原理就类似于球面投影，因此，以视点为中心的球面投影模型是最自然的投影模型。但是球面投影模型也存在着一些缺点，比如球面上的像素点不是行列均匀排列的关系，球面不能展开成平面，这些都使得很多图像处理算法很难用在平面投影上。球面投影的正向映射为：

（77）

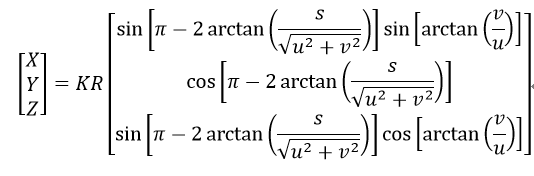
球面投影的反向映射为：

（78）

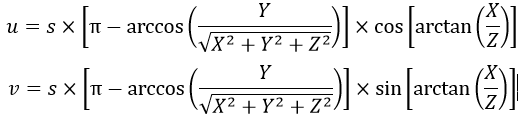
立方体投影是为了克服球形投影缺点而提出的投影模型。这种投影模型的优点是方便计算机处理与储存图像。立方体投影的正向映射为：

（79）

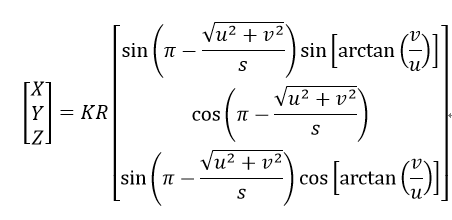
立方体投影的反向映射为：

（80）

鱼眼投影图像具有较大的视角，非常适用于导航、监视和检测等方面。它的正向映射为：

（81）

鱼眼投影的反向映射为：

（82）

在图像拼接过程中，我们首先需要把图像进行正向映射，又因为最终图像还是要在平面上进行展示，所以还是需要再进行反向映射。最终被映射到的平面就是全景图像所在的平面，这是因为在上一步，我们已经通过最大生成树得到了基准图像，相机的内参数都是基于该基准图像的，所以所有的图像最终都映射到了该基准图像所在的平面上，这样就构成了一幅全景图像。

4.2 源码

RotationWarper类是只处理因旋转而引起的图像扭曲的接口类，它是RotationWarperBase类的基类：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **template** <**class** P>
2. **class** CV\_EXPORTS RotationWarperBase : **public** RotationWarper
3. {
4. **public**:
5. //表示投影图像的像素点，pt为源像素点，它通过P.mapForward函数得到投影点（该函数的返回值），K为相机的内参数，R为相机的旋转矩阵，通过P.setCameraParams函数设置
6. Point2f warpPoint(**const** Point2f &pt, **const** Mat &K, **const** Mat &R);
7. //由给定的相机数据建立投影关系，src\_size为源图像区域，xmap和ymap是分别表示坐标值由两次映射的值，该函数返回投影图区域
8. Rect buildMaps(Size src\_size, **const** Mat &K, **const** Mat &R, Mat &xmap, Mat &ymap);
9. //表示由源图src经buildMaps函数得到投影图像dst，interp\_mode和border\_mode分别表示投影时用到的插值算法和边界扩展方法，该函数返回dst在最终的全景图像投影后的左上角坐标
10. Point warp(**const** Mat &src, **const** Mat &K, **const** Mat &R, **int** interp\_mode, **int** border\_mode,
11. Mat &dst);
12. //与buildMaps函数相类似，只不过该函数使用的是P.mapForward函数
13. **void** warpBackward(**const** Mat &src, **const** Mat &K, **const** Mat &R, **int** interp\_mode, **int** border\_mode,
14. Size dst\_size, Mat &dst);
15. //表示确定扭曲图像区域
16. Rect warpRoi(Size src\_size, **const** Mat &K, **const** Mat &R);
18. **float** getScale() **const** { **return** projector\_.scale; }    //得到尺度
19. **void** setScale(**float** val) { projector\_.scale = val; }    //设置尺度
21. **protected**:
23. // Detects ROI of the destination image. It's correct for any projection.
24. //该虚函数用于得到目标图像的区域
25. **virtual** **void** detectResultRoi(Size src\_size, Point &dst\_tl, Point &dst\_br);
27. // Detects ROI of the destination image by walking over image border.
28. // Correctness for any projection isn't guaranteed.
29. //该函数仅由源图像的边界得到目标图像的区域
30. **void** detectResultRoiByBorder(Size src\_size, Point &dst\_tl, Point &dst\_br);
32. P projector\_;    //表示投影的方法
33. };

下面我们给出RotationWarperBase类中主要函数的介绍：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **template** <**class** P>
2. Point2f RotationWarperBase<P>::warpPoint(**const** Point2f &pt, **const** Mat &K, **const** Mat &R)
3. //pt表示投射的源点
4. //K表示相机的内参数
5. //R表示相机的旋转参数
6. //该函数返回投射点
7. {
8. projector\_.setCameraParams(K, R);    //设置相机参数
9. Point2f uv;    //表示投射映射点
10. projector\_.mapForward(pt.x, pt.y, uv.x, uv.y);    //前向投影，得到投射点
11. **return** uv;    //返回投射点
12. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **template** <**class** P>
2. Rect RotationWarperBase<P>::buildMaps(Size src\_size, **const** Mat &K, **const** Mat &R, Mat &xmap, Mat &ymap)
3. //src\_size表示源图的区域
4. //K表示相机的内参数
5. //R表示相机的旋转参数
6. //xmap和ymap分别表示返回横纵坐标的前向映射后再反向映射的值
7. //该函数返回投影后的区域尺寸
8. {
9. projector\_.setCameraParams(K, R);    //设置相机参数
11. Point dst\_tl, dst\_br;    //表示投影区域的左上角坐标和右下角坐标
12. //得到映射后的左上角坐标dst\_tl和右下角坐标dst\_br
13. detectResultRoi(src\_size, dst\_tl, dst\_br);
14. //创建xmap和ymap矩阵大小
15. xmap.create(dst\_br.y - dst\_tl.y + 1, dst\_br.x - dst\_tl.x + 1, CV\_32F);
16. ymap.create(dst\_br.y - dst\_tl.y + 1, dst\_br.x - dst\_tl.x + 1, CV\_32F);
18. **float** x, y;    //表示反向投影映射后的x轴和y轴坐标值
19. //遍历投影区域，再进行反向映射
20. **for** (**int** v = dst\_tl.y; v <= dst\_br.y; ++v)
21. {
22. **for** (**int** u = dst\_tl.x; u <= dst\_br.x; ++u)
23. {
24. //反向投影
25. projector\_.mapBackward(**static\_cast**<**float**>(u), **static\_cast**<**float**>(v), x, y);
26. xmap.at<**float**>(v - dst\_tl.y, u - dst\_tl.x) = x;    //赋值
27. ymap.at<**float**>(v - dst\_tl.y, u - dst\_tl.x) = y;    //赋值
28. }
29. }
31. **return** Rect(dst\_tl, dst\_br);    //返回投影映射区域
32. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **template** <**class** P>
2. Point RotationWarperBase<P>::warp(**const** Mat &src, **const** Mat &K, **const** Mat &R, **int** interp\_mode, **int** border\_mode,
3. Mat &dst)
4. //src表示源图
5. //K表示相机内参数
6. //R表示相机的旋转参数
7. //interp\_mode表示插值模式
8. //border\_mode表示边界扩充模式
9. //dst表示投影映射图
10. //该函数返回投影映射图的左上角在基准图像坐标系下的坐标，即全景图像坐标系下的坐标
11. {
12. Mat xmap, ymap;
13. Rect dst\_roi = buildMaps(src.size(), K, R, xmap, ymap);    //调用buildMaps函数
15. dst.create(dst\_roi.height + 1, dst\_roi.width + 1, src.type());    //创建大小
16. //按xmap和ymap对src进行重映射，得到dst
17. remap(src, dst, xmap, ymap, interp\_mode, border\_mode);
19. **return** dst\_roi.tl();    //返回左上角坐标
20. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **template** <**class** P>
2. Rect RotationWarperBase<P>::warpRoi(Size src\_size, **const** Mat &K, **const** Mat &R)
3. //src表示源图
4. //K表示相机内参数
5. //R表示相机的旋转参数
6. //返回投影矩形区域
7. {
8. projector\_.setCameraParams(K, R);    //设置相机参数
10. Point dst\_tl, dst\_br;
11. detectResultRoi(src\_size, dst\_tl, dst\_br);    //得到映射区域
13. **return** Rect(dst\_tl, Point(dst\_br.x + 1, dst\_br.y + 1));    //返回映射矩形区域
14. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **template** <**class** P>
2. **void** RotationWarperBase<P>::detectResultRoi(Size src\_size, Point &dst\_tl, Point &dst\_br)
3. //src\_size表示源图像区域
4. //dst\_tl和dst\_br分别表示返回得到的投影后区域的左上角坐标和右下角坐标
5. {
6. //下面4个变量分别表示左上角和右下角x轴和y轴的值
7. **float** tl\_uf = std::numeric\_limits<**float**>::max();    //先初始化为最大值
8. **float** tl\_vf = std::numeric\_limits<**float**>::max();    //先初始化为最大值
9. **float** br\_uf = -std::numeric\_limits<**float**>::max();    //先初始化为最小值
10. **float** br\_vf = -std::numeric\_limits<**float**>::max();    //先初始化为最小值
12. **float** u, v;
13. **for** (**int** y = 0; y < src\_size.height; ++y)    //遍历源图区域
14. {
15. **for** (**int** x = 0; x < src\_size.width; ++x)
16. {
17. //前向映射
18. projector\_.mapForward(**static\_cast**<**float**>(x), **static\_cast**<**float**>(y), u, v);
19. tl\_uf = std::min(tl\_uf, u); tl\_vf = std::min(tl\_vf, v);    //更新左上角坐标
20. br\_uf = std::max(br\_uf, u); br\_vf = std::max(br\_vf, v);    //更新右下角坐标
21. }
22. }
23. //得到最终的左上角和右下角坐标
24. dst\_tl.x = **static\_cast**<**int**>(tl\_uf);
25. dst\_tl.y = **static\_cast**<**int**>(tl\_vf);
26. dst\_br.x = **static\_cast**<**int**>(br\_uf);
27. dst\_br.y = **static\_cast**<**int**>(br\_vf);
28. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **template** <**class** P>
2. **void** RotationWarperBase<P>::detectResultRoiByBorder(Size src\_size, Point &dst\_tl, Point &dst\_br)
3. {
4. //下面4个变量分别表示左上角和右下角x轴和y轴的值
5. **float** tl\_uf = std::numeric\_limits<**float**>::max();    //先初始化为最大值
6. **float** tl\_vf = std::numeric\_limits<**float**>::max();    //先初始化为最大值
7. **float** br\_uf = -std::numeric\_limits<**float**>::max();    //先初始化为最小值
8. **float** br\_vf = -std::numeric\_limits<**float**>::max();    //先初始化为最小值
10. **float** u, v;
11. **for** (**float** x = 0; x < src\_size.width; ++x)    //遍历源图的横坐标
12. {
13. projector\_.mapForward(**static\_cast**<**float**>(x), 0, u, v);    //上边映射
14. tl\_uf = std::min(tl\_uf, u); tl\_vf = std::min(tl\_vf, v);
15. br\_uf = std::max(br\_uf, u); br\_vf = std::max(br\_vf, v);
16. //下边映射
17. projector\_.mapForward(**static\_cast**<**float**>(x), **static\_cast**<**float**>(src\_size.height - 1), u, v);
18. tl\_uf = std::min(tl\_uf, u); tl\_vf = std::min(tl\_vf, v);
19. br\_uf = std::max(br\_uf, u); br\_vf = std::max(br\_vf, v);
20. }
21. **for** (**int** y = 0; y < src\_size.height; ++y)    //遍历源图的纵坐标
22. {
23. projector\_.mapForward(0, **static\_cast**<**float**>(y), u, v);    左边映射
24. tl\_uf = std::min(tl\_uf, u); tl\_vf = std::min(tl\_vf, v);
25. br\_uf = std::max(br\_uf, u); br\_vf = std::max(br\_vf, v);
26. //右边映射
27. projector\_.mapForward(**static\_cast**<**float**>(src\_size.width - 1), **static\_cast**<**float**>(y), u, v);
28. tl\_uf = std::min(tl\_uf, u); tl\_vf = std::min(tl\_vf, v);
29. br\_uf = std::max(br\_uf, u); br\_vf = std::max(br\_vf, v);
30. }
31. //得到坐标
32. dst\_tl.x = **static\_cast**<**int**>(tl\_uf);
33. dst\_tl.y = **static\_cast**<**int**>(tl\_vf);
34. dst\_br.x = **static\_cast**<**int**>(br\_uf);
35. dst\_br.y = **static\_cast**<**int**>(br\_vf);
36. }

投影方法的基类结构为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **struct** CV\_EXPORTS ProjectorBase
2. {
3. //设置相机参数，该函数见后面的介绍
4. **void** setCameraParams(**const** Mat &K = Mat::eye(3, 3, CV\_32F),
5. **const** Mat &R = Mat::eye(3, 3, CV\_32F),
6. **const** Mat &T = Mat::zeros(3, 1, CV\_32F));
8. **float** scale;    //表示尺度
9. **float** k[9];    //表示相机的内参数矩阵K，用向量形式表示
10. //在前面的程序分析中，我们已强调过，程序中所表示的旋转矩阵r其实是公式中旋转矩阵R的逆
11. **float** rinv[9];    //表示相机旋转矩阵r的逆（就是R），用向量形式表示
12. **float** r\_kinv[9];    //表示rK-1（就是R-1K-1），用向量形式表示
13. **float** k\_rinv[9];    //表示Kr-1（就是KR），用向量形式表示
14. **float** t[3];    //表示三个方向的平移量
15. };

设置相机的内外参数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **void** ProjectorBase::setCameraParams(**const** Mat &K, **const** Mat &R, **const** Mat &T)
2. //K表示相机的内参数
3. //R表示相机的旋转参数
4. //T表示相机的平移量
5. {
6. //确保三个输入参数正确
7. CV\_Assert(K.size() == Size(3, 3) && K.type() == CV\_32F);
8. CV\_Assert(R.size() == Size(3, 3) && R.type() == CV\_32F);
9. CV\_Assert((T.size() == Size(1, 3) || T.size() == Size(3, 1)) && T.type() == CV\_32F);
11. Mat\_<**float**> K\_(K);    //复制
12. //把矩阵形式的K转换为向量形式的k
13. k[0] = K\_(0,0); k[1] = K\_(0,1); k[2] = K\_(0,2);
14. k[3] = K\_(1,0); k[4] = K\_(1,1); k[5] = K\_(1,2);
15. k[6] = K\_(2,0); k[7] = K\_(2,1); k[8] = K\_(2,2);
17. Mat\_<**float**> Rinv = R.t();    //得到r的逆，即R-1
18. //得到向量形式的rinv
19. rinv[0] = Rinv(0,0); rinv[1] = Rinv(0,1); rinv[2] = Rinv(0,2);
20. rinv[3] = Rinv(1,0); rinv[4] = Rinv(1,1); rinv[5] = Rinv(1,2);
21. rinv[6] = Rinv(2,0); rinv[7] = Rinv(2,1); rinv[8] = Rinv(2,2);
23. Mat\_<**float**> R\_Kinv = R \* K.inv();    //得到rK-1，即R-1K-1
24. //得到向量形式的r\_kinv
25. r\_kinv[0] = R\_Kinv(0,0); r\_kinv[1] = R\_Kinv(0,1); r\_kinv[2] = R\_Kinv(0,2);
26. r\_kinv[3] = R\_Kinv(1,0); r\_kinv[4] = R\_Kinv(1,1); r\_kinv[5] = R\_Kinv(1,2);
27. r\_kinv[6] = R\_Kinv(2,0); r\_kinv[7] = R\_Kinv(2,1); r\_kinv[8] = R\_Kinv(2,2);
29. Mat\_<**float**> K\_Rinv = K \* Rinv;    //得到Kr-1，即KR
30. //得到向量形式的k\_rinv
31. k\_rinv[0] = K\_Rinv(0,0); k\_rinv[1] = K\_Rinv(0,1); k\_rinv[2] = K\_Rinv(0,2);
32. k\_rinv[3] = K\_Rinv(1,0); k\_rinv[4] = K\_Rinv(1,1); k\_rinv[5] = K\_Rinv(1,2);
33. k\_rinv[6] = K\_Rinv(2,0); k\_rinv[7] = K\_Rinv(2,1); k\_rinv[8] = K\_Rinv(2,2);
35. Mat\_<**float**> T\_(T.reshape(0, 3));    //复制
36. //把矩阵形式的T转换为向量形式的t
37. t[0] = T\_(0,0); t[1] = T\_(1,0); t[2] = T\_(2,0);
38. }

各种扭曲方法都是以RotationWarperBase为基类，各种投影方法都是以ProjectorBase为基类，扭曲和投影是一一对应的，通过不同的投影算法实现不同的图像扭曲。Opencv实现了许多投影算法，有平面、柱面、球面、鱼眼、立方体、压缩直线、压缩直线人像、panini（弯曲）、panini人像、Mercator（正轴等角柱面）、横向Mercator、球面人像、柱面人像、平面人像等投影算法。只要投影算法掌握了，通过映射得到图像扭曲就很容易，下面我们就重点介绍平面、柱面、球面、鱼眼、立方体这几种投影类，这些投影类主要就是实现了正向和反向映射算法。

平面投影：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** PlaneProjector::mapForward(**float** x, **float** y, **float** &u, **float** &v)    //正向
3. //x和y表示源图像的坐标
4. //u和v表示投影图像的坐标
5. {
6. //式70
7. **float** x\_ = r\_kinv[0] \* x + r\_kinv[1] \* y + r\_kinv[2];
8. **float** y\_ = r\_kinv[3] \* x + r\_kinv[4] \* y + r\_kinv[5];
9. **float** z\_ = r\_kinv[6] \* x + r\_kinv[7] \* y + r\_kinv[8];
10. //式72
11. x\_ = t[0] + x\_ / z\_ \* (1 - t[2]);
12. y\_ = t[1] + y\_ / z\_ \* (1 - t[2]);
13. //式72
14. u = scale \* x\_;
15. v = scale \* y\_;
16. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** PlaneProjector::mapBackward(**float** u, **float** v, **float** &x, **float** &y)    //反向
3. //u和v表示投影图像的坐标
4. //x和y表示源图像的坐标
5. {
6. u = u / scale - t[0];
7. v = v / scale - t[1];
8. //式73
9. **float** z;
10. x = k\_rinv[0] \* u + k\_rinv[1] \* v + k\_rinv[2] \* (1 - t[2]);
11. y = k\_rinv[3] \* u + k\_rinv[4] \* v + k\_rinv[5] \* (1 - t[2]);
12. z = k\_rinv[6] \* u + k\_rinv[7] \* v + k\_rinv[8] \* (1 - t[2]);
13. //式74
14. x /= z;
15. y /= z;
16. }

柱面投影：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** CylindricalProjector::mapForward(**float** x, **float** y, **float** &u, **float** &v)    //正向
3. {
4. //式70
5. **float** x\_ = r\_kinv[0] \* x + r\_kinv[1] \* y + r\_kinv[2];
6. **float** y\_ = r\_kinv[3] \* x + r\_kinv[4] \* y + r\_kinv[5];
7. **float** z\_ = r\_kinv[6] \* x + r\_kinv[7] \* y + r\_kinv[8];
8. //式75
9. u = scale \* atan2f(x\_, z\_);
10. v = scale \* y\_ / sqrtf(x\_ \* x\_ + z\_ \* z\_);
11. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** CylindricalProjector::mapBackward(**float** u, **float** v, **float** &x, **float** &y)    //反向
3. {
4. u /= scale;
5. v /= scale;
7. **float** x\_ = sinf(u);
8. **float** y\_ = v;
9. **float** z\_ = cosf(u);
10. //式76
11. **float** z;
12. x = k\_rinv[0] \* x\_ + k\_rinv[1] \* y\_ + k\_rinv[2] \* z\_;
13. y = k\_rinv[3] \* x\_ + k\_rinv[4] \* y\_ + k\_rinv[5] \* z\_;
14. z = k\_rinv[6] \* x\_ + k\_rinv[7] \* y\_ + k\_rinv[8] \* z\_;
15. //式74
16. **if** (z > 0) { x /= z; y /= z; }
17. **else** x = y = -1;
18. }

球面投影：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** SphericalProjector::mapForward(**float** x, **float** y, **float** &u, **float** &v)    //正向
3. {
4. //式70
5. **float** x\_ = r\_kinv[0] \* x + r\_kinv[1] \* y + r\_kinv[2];
6. **float** y\_ = r\_kinv[3] \* x + r\_kinv[4] \* y + r\_kinv[5];
7. **float** z\_ = r\_kinv[6] \* x + r\_kinv[7] \* y + r\_kinv[8];
8. //式77
9. u = scale \* atan2f(x\_, z\_);
10. **float** w = y\_ / sqrtf(x\_ \* x\_ + y\_ \* y\_ + z\_ \* z\_);
11. v = scale \* (**static\_cast**<**float**>(CV\_PI) - acosf(w == w ? w : 0));
12. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** SphericalProjector::mapBackward(**float** u, **float** v, **float** &x, **float** &y)    //反向
3. {
4. u /= scale;
5. v /= scale;
6. //式78
7. **float** sinv = sinf(**static\_cast**<**float**>(CV\_PI) - v);
8. **float** x\_ = sinv \* sinf(u);
9. **float** y\_ = cosf(**static\_cast**<**float**>(CV\_PI) - v);
10. **float** z\_ = sinv \* cosf(u);
12. **float** z;
13. x = k\_rinv[0] \* x\_ + k\_rinv[1] \* y\_ + k\_rinv[2] \* z\_;
14. y = k\_rinv[3] \* x\_ + k\_rinv[4] \* y\_ + k\_rinv[5] \* z\_;
15. z = k\_rinv[6] \* x\_ + k\_rinv[7] \* y\_ + k\_rinv[8] \* z\_;
16. //式74
17. **if** (z > 0) { x /= z; y /= z; }
18. **else** x = y = -1;
19. }

立方体投影：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** StereographicProjector::mapForward(**float** x, **float** y, **float** &u, **float** &v)    //正向
3. {
4. //式70
5. **float** x\_ = r\_kinv[0] \* x + r\_kinv[1] \* y + r\_kinv[2];
6. **float** y\_ = r\_kinv[3] \* x + r\_kinv[4] \* y + r\_kinv[5];
7. **float** z\_ = r\_kinv[6] \* x + r\_kinv[7] \* y + r\_kinv[8];
8. //式79
9. **float** u\_ = atan2f(x\_, z\_);
10. **float** v\_ = (**float**)CV\_PI - acosf(y\_ / sqrtf(x\_ \* x\_ + y\_ \* y\_ + z\_ \* z\_));
12. **float** r = sinf(v\_) / (1 - cosf(v\_));
14. u = scale \* r \* cos(u\_);
15. v = scale \* r \* sin(u\_);
16. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** StereographicProjector::mapBackward(**float** u, **float** v, **float** &x, **float** &y)    //反向
3. {
4. u /= scale;
5. v /= scale;
7. **float** u\_ = atan2f(v, u);
8. **float** r = sqrtf(u\*u + v\*v);
9. **float** v\_ = 2 \* atanf(1.f / r);
11. **float** sinv = sinf((**float**)CV\_PI - v\_);
12. **float** x\_ = sinv \* sinf(u\_);
13. **float** y\_ = cosf((**float**)CV\_PI - v\_);
14. **float** z\_ = sinv \* cosf(u\_);
15. //式80
16. **float** z;
17. x = k\_rinv[0] \* x\_ + k\_rinv[1] \* y\_ + k\_rinv[2] \* z\_;
18. y = k\_rinv[3] \* x\_ + k\_rinv[4] \* y\_ + k\_rinv[5] \* z\_;
19. z = k\_rinv[6] \* x\_ + k\_rinv[7] \* y\_ + k\_rinv[8] \* z\_;
20. //式74
21. **if** (z > 0) { x /= z; y /= z; }
22. **else** x = y = -1;
23. }

鱼眼投影：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** FisheyeProjector::mapForward(**float** x, **float** y, **float** &u, **float** &v)    //正向
3. {
4. //式70
5. **float** x\_ = r\_kinv[0] \* x + r\_kinv[1] \* y + r\_kinv[2];
6. **float** y\_ = r\_kinv[3] \* x + r\_kinv[4] \* y + r\_kinv[5];
7. **float** z\_ = r\_kinv[6] \* x + r\_kinv[7] \* y + r\_kinv[8];
8. //式81
9. **float** u\_ = atan2f(x\_, z\_);
10. **float** v\_ = (**float**)CV\_PI - acosf(y\_ / sqrtf(x\_ \* x\_ + y\_ \* y\_ + z\_ \* z\_));
12. u = scale \* v\_ \* cosf(u\_);
13. v = scale \* v\_ \* sinf(u\_);
14. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **inline**
2. **void** FisheyeProjector::mapBackward(**float** u, **float** v, **float** &x, **float** &y)    //反向
3. {
4. u /= scale;
5. v /= scale;
6. //式82
7. **float** u\_ = atan2f(v, u);
8. **float** v\_ = sqrtf(u\*u + v\*v);
10. **float** sinv = sinf((**float**)CV\_PI - v\_);
11. **float** x\_ = sinv \* sinf(u\_);
12. **float** y\_ = cosf((**float**)CV\_PI - v\_);
13. **float** z\_ = sinv \* cosf(u\_);
15. **float** z;
16. x = k\_rinv[0] \* x\_ + k\_rinv[1] \* y\_ + k\_rinv[2] \* z\_;
17. y = k\_rinv[3] \* x\_ + k\_rinv[4] \* y\_ + k\_rinv[5] \* z\_;
18. z = k\_rinv[6] \* x\_ + k\_rinv[7] \* y\_ + k\_rinv[8] \* z\_;
19. //式74
20. **if** (z > 0) { x /= z; y /= z; }
21. **else** x = y = -1;
22. }

前面介绍了映射变换算法的几个重要函数，下面给出编写映射变换程序时还需要用到的一些其他类。

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. WarperCreator类表示映射变换的生成器：
2. **class** WarperCreator
3. {
4. **public**:
5. **virtual** ~WarperCreator() {}
6. //生成映射变换器，scale表示映射尺度
7. **virtual** Ptr<detail::RotationWarper> create(**float** scale) **const** = 0;
8. };

具体算法的映射变换器类都是WarperCreator类的子类，如平面映射PlaneWarper，柱面映射CylindricalWarper，球面映射SphericalWarper等等，它们的内容基本相似，我们只给出PlaneWarper类：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. **class** PlaneWarper : **public** WarperCreator
2. {
3. **public**:
4. Ptr<detail::RotationWarper> create(**float** scale) **const** { **return** **new** detail::PlaneWarper(scale); }
5. };

4.3 应用

图像映射投影变换的应用：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78829736) [copy](http://blog.csdn.net/zhaocj/article/details/78829736)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. #include <iomanip>
23. **using** **namespace** cv;
24. **using** **namespace** std;
25. **using** **namespace** detail;
27. **int** main(**int** argc, **char**\*\* argv)
28. {
30. vector<Mat> imgs;    //输入图像
31. Mat img = imread("1.jpg");
32. imgs.push\_back(img);
33. img = imread("2.jpg");
34. imgs.push\_back(img);
36. Ptr<FeaturesFinder> finder;    //特征检测
37. finder = **new** SurfFeaturesFinder();
38. vector<ImageFeatures> features(2);
39. (\*finder)(imgs[0], features[0]);
40. (\*finder)(imgs[1], features[1]);
42. vector<MatchesInfo> pairwise\_matches;    //特征匹配
43. BestOf2NearestMatcher matcher(**false**, 0.3f, 6, 6);
44. matcher(features, pairwise\_matches);
46. HomographyBasedEstimator estimator;    //相机参数评估
47. vector<CameraParams> cameras;
48. estimator(features, pairwise\_matches, cameras);
49. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
50. {
51. Mat R;
52. cameras[i].R.convertTo(R, CV\_32F);
53. cameras[i].R = R;
54. }
56. Ptr<detail::BundleAdjusterBase> adjuster;    //光束平差法，精确相机参数
57. adjuster = **new** detail::BundleAdjusterReproj();
58. adjuster->setConfThresh(1);
59. (\*adjuster)(features, pairwise\_matches, cameras);
61. vector<Mat> rmats;
62. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
63. rmats.push\_back(cameras[i].R.clone());
64. waveCorrect(rmats, WAVE\_CORRECT\_HORIZ);    //波形校正
65. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
66. cameras[i].R = rmats[i];
68. vector<Point> corners(2);    //表示映射变换后图像的左上角坐标
69. vector<Mat> masks\_warped(2);    //表示映射变换后的图像掩码
70. vector<Mat> images\_warped(2);    //表示映射变换后的图像
71. vector<Size> sizes(2);    //表示映射变换后的图像尺寸
72. vector<Mat> masks(2);    //表示源图的掩码
74. **for** (**int** i = 0; i < 2; ++i)    //初始化源图的掩码
75. {
76. masks[i].create(imgs[i].size(), CV\_8U);    //定义尺寸大小
77. masks[i].setTo(Scalar::all(255));    //全部赋值为255，表示源图的所有区域都使用
78. }
80. Ptr<WarperCreator> warper\_creator;    //定义图像映射变换创造器
81. //warper\_creator = new cv::PlaneWarper();    //平面投影
82. //warper\_creator = new cv::CylindricalWarper();    //柱面投影
83. //warper\_creator = new cv::SphericalWarper();    //球面投影
84. //warper\_creator = new cv::FisheyeWarper();    //鱼眼投影
85. warper\_creator = **new** cv::StereographicWarper();    //立方体投影
87. //定义图像映射变换器，设置映射的尺度为相机的焦距，所有相机的焦距都相同
88. Ptr<RotationWarper> warper = warper\_creator->create(**static\_cast**<**float**>(cameras[0].focal));
89. **for** (**int** i = 0; i < 2; ++i)
90. {
91. Mat\_<**float**> K;
92. cameras[i].K().convertTo(K, CV\_32F);    //转换相机内参数的数据类型
93. //对当前图像镜像投影变换，得到变换后的图像以及该图像的左上角坐标
94. corners[i] = warper->warp(imgs[i], K, cameras[i].R, INTER\_LINEAR, BORDER\_REFLECT, images\_warped[i]);
95. sizes[i] = images\_warped[i].size();    //得到尺寸
96. //得到变换后的图像掩码
97. warper->warp(masks[i], K, cameras[i].R, INTER\_NEAREST, BORDER\_CONSTANT, masks\_warped[i]);
98. }
100. //通过掩码，只得到映射变换后的图像
101. **for**(**int** k =0;k<2;k++)
102. {
103. **for**(**int** i=0;i<sizes[k].height;i++)
104. {
105. **for**(**int** j=0;j<sizes[k].width;j++)
106. {
107. **if**(masks\_warped[k].at<uchar>(i, j)==0)    //掩码
108. {
109. images\_warped[k].at<Vec3b>(i, j)[0]=0;
110. images\_warped[k].at<Vec3b>(i, j)[1]=0;
111. images\_warped[k].at<Vec3b>(i, j)[2]=0;
112. }
113. }
114. }
115. }
117. imwrite("warp1.jpg", images\_warped[0]);
118. imwrite("warp2.jpg", images\_warped[1]);
120. **return** 0;
121. }

1.jpg和2.jpg仍然是第2.3节程序中的那两幅图像，则经过立体投影后的图像为：

图10 立体投影图

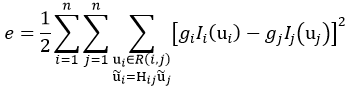
# Opencv2.4.9源码分析——Stitching（五）

5、曝光补偿

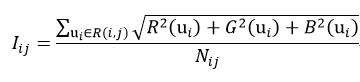
5.1 原理

即使通过几何投影，图像之间可以做到很好的拼接，但如果不同图像之间有不同的曝光程度，那么拼接图像中的重叠部分也会出现明显的边缘，这样就使图像看起来十分不自然。因此，我们还需要对每幅图像进行曝光补偿，来使所有图像具有相同的曝光程度。

目前，常用的曝光补偿方法有增益补偿和分块补偿这两种方法。增益补偿就是为每幅图像赋予一个增益系数，使重叠部分的图像强度相等或相似。它可以利用误差函数来实现：

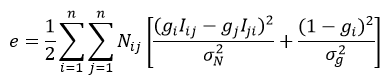
（83）

式中，gi和gj为图像i和图像j的增益系数，R(i,j)表示图像i和图像j的重叠部分，Ii(ui)表示图像i在重叠部分R(i,j)的强度平均值Iij：

（84）

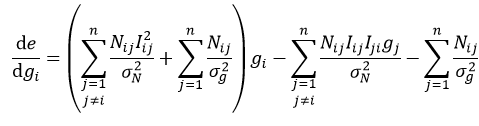
式中，R、G和B分别表示彩色图像的红、绿和蓝分量的强度值，Nij表示重叠部分R(i,j)的像素数量。

很显然，当g＝0时，式83所表示的误差函数为零，也就是最优解，这明显不是我们想要的值；另外为了提高鲁棒性，增益补偿误差函数的经验公式为：

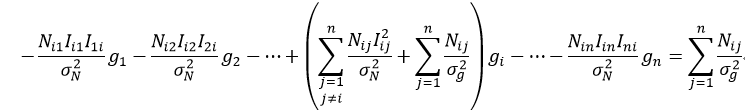
（85）

式中，σN和σg分别表示误差和增益的标准差，σN＝10（如果强度范围为0～255），σg＝0.1。

式85是增益系数g的一个二次目标函数，该函数可以通过使其导数为0得到闭合形式的解。式85对gi的导数为：

（86）

使该导数为0，并展开成以g1，g2，……，gn为变量的等式：

（87）

e对所有的g求导，这样就建立起了n个如式87一样的线性方程。求解该方程组，则最终得到了n幅图像的n个增益系数g。

第二种曝光补偿方法是分块补偿，它的原理是把图像分割成大小相同的不同块，这样以块为单位进行增益补偿，即每一块都有一个增益系数。该方法的补偿精度更好，并且还可以缓解增益补偿算法的复杂性。块的尺寸大小的经验值为32×32。分块补偿的结果会使拼接图像呈现“块”的形态，因此还需要对每一块的增益系数进行平滑滤波处理，具体来说，对同一幅图像的所有块的增益系数应用分割线性滤波的方法，分割滤波的核为[1/4, 1/2, 1/4]。

5.2 源码

曝光补偿的基类：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78883794) [copy](http://blog.csdn.net/zhaocj/article/details/78883794)

1. **class** CV\_EXPORTS ExposureCompensator
2. {
3. **public**:
4. **virtual** ~ExposureCompensator() {}
6. **enum** { NO, GAIN, GAIN\_BLOCKS };
7. **static** Ptr<ExposureCompensator> createDefault(**int** type);
9. **void** feed(**const** std::vector<Point> &corners, **const** std::vector<Mat> &images,
10. **const** std::vector<Mat> &masks);
11. **virtual** **void** feed(**const** std::vector<Point> &corners, **const** std::vector<Mat> &images,
12. **const** std::vector<std::pair<Mat,uchar> > &masks) = 0;
13. **virtual** **void** apply(**int** index, Point corner, Mat &image, **const** Mat &mask) = 0;
14. };

ExposureCompensator类中的createDefault函数表示创建曝光补偿器：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78883794) [copy](http://blog.csdn.net/zhaocj/article/details/78883794)

1. Ptr<ExposureCompensator> ExposureCompensator::createDefault(**int** type)
2. //type表示曝光补偿的方法
3. //该函数返回具体曝光补偿方法的函数指针
4. {
5. **if** (type == NO)    //表示不进行曝光补偿
6. **return** **new** NoExposureCompensator();    //指向NoExposureCompensator类
7. **if** (type == GAIN)    //表示应用增益补偿方法进行曝光补偿
8. **return** **new** GainCompensator();    //指向GainCompensator类
9. **if** (type == GAIN\_BLOCKS)    //表示应该分块补偿方法进行曝光补偿
10. **return** **new** BlocksGainCompensator();    //指向BlocksGainCompensator类
11. //没有其他的曝光补偿算法
12. CV\_Error(CV\_StsBadArg, "unsupported exposure compensation method");
13. **return** NULL;
14. }

ExposureCompensator类中有两个feed函数，其中的实函数为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78883794) [copy](http://blog.csdn.net/zhaocj/article/details/78883794)

1. **void** ExposureCompensator::feed(**const** vector<Point> &corners, **const** vector<Mat> &images,
2. **const** vector<Mat> &masks)
3. //corners表示images在全景图像坐标系下的左上角坐标
4. //images表示投影变换后的各个图像
5. //masks表示上一步图像投影后的图像掩码
6. {
7. vector<pair<Mat,uchar> > level\_masks;    //表示为masks赋予一个等级系数
8. //为level\_masks赋值
9. **for** (**size\_t** i = 0; i < masks.size(); ++i)
10. level\_masks.push\_back(make\_pair(masks[i], 255));
11. feed(corners, images, level\_masks);    //调用feed的虚函数
12. }

三种实现曝光补偿方法的类——NoExposureCompensator、GainCompensator和BlocksGainCompensator都是以ExposureCompensator类作为基类。NoExposureCompensator类表示不进行曝光补偿，其中的feed和apply函数都是空函数。GainCompensator类表示的是增益补偿方法，其中的feed函数表示计算增益系数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78883794) [copy](http://blog.csdn.net/zhaocj/article/details/78883794)

1. **void** GainCompensator::feed(**const** vector<Point> &corners, **const** vector<Mat> &images,
2. **const** vector<pair<Mat,uchar> > &masks)
3. {
4. LOGLN("Exposure compensation...");    //终端输出
5. #if ENABLE\_LOG
6. int64 t = getTickCount();    //用于计时
7. #endif
8. //确保各个输入参数的数量的一致性
9. CV\_Assert(corners.size() == images.size() && images.size() == masks.size());
11. **const** **int** num\_images = **static\_cast**<**int**>(images.size());    //表示待拼接的图像数量
12. //N表示两幅图像间重叠部分的像素数量，即式84中的Nij
13. Mat\_<**int**> N(num\_images, num\_images); N.setTo(0);
14. //I表示两幅图像间重叠部分的第一个元素所代表的图像的强度，即式84的Iij
15. Mat\_<**double**> I(num\_images, num\_images); I.setTo(0);
17. //Rect dst\_roi = resultRoi(corners, images);
18. Mat subimg1, subimg2;
19. Mat\_<uchar> submask1, submask2, intersect;
20. //遍历相交的图像对
21. **for** (**int** i = 0; i < num\_images; ++i)
22. {
23. **for** (**int** j = i; j < num\_images; ++j)
24. {
25. Rect roi;    //表示图像i和图像j的相交部分
26. //判断图像i和图像j是否有相交部分
27. **if** (overlapRoi(corners[i], corners[j], images[i].size(), images[j].size(), roi))
28. {
29. //subimg1和subimg2分别表示图像i和图像j相交部分在各自图像上的子图像
30. subimg1 = images[i](Rect(roi.tl() - corners[i], roi.br() - corners[i]));
31. subimg2 = images[j](Rect(roi.tl() - corners[j], roi.br() - corners[j]));
32. //表示把subimg1和subimg2分别赋予submask1和submask2
33. submask1 = masks[i].first(Rect(roi.tl() - corners[i], roi.br() - corners[i]));
34. submask2 = masks[j].first(Rect(roi.tl() - corners[j], roi.br() - corners[j]));
35. //intersect表示掩码的相交部分
36. intersect = (submask1 == masks[i].second) & (submask2 == masks[j].second);
37. //得到Nij
38. N(i, j) = N(j, i) = max(1, countNonZero(intersect));
40. **double** Isum1 = 0, Isum2 = 0;    //表示式84的分子部分
41. **for** (**int** y = 0; y < roi.height; ++y)    //遍历重叠部分的行
42. {
43. //表示当前行的首地址
44. **const** Point3\_<uchar>\* r1 = subimg1.ptr<Point3\_<uchar> >(y);
45. **const** Point3\_<uchar>\* r2 = subimg2.ptr<Point3\_<uchar> >(y);
46. **for** (**int** x = 0; x < roi.width; ++x)    //遍历当前行的每个像素
47. {
48. //计算相交重叠部分的强度，式（84）的分子部分
49. **if** (intersect(y, x))
50. {
51. Isum1 += sqrt(**static\_cast**<**double**>(sqr(r1[x].x) + sqr(r1[x].y) + sqr(r1[x].z)));
52. Isum2 += sqrt(**static\_cast**<**double**>(sqr(r2[x].x) + sqr(r2[x].y) + sqr(r2[x].z)));
53. }
54. }
55. }
56. I(i, j) = Isum1 / N(i, j);    //式（84），得到Iij
57. I(j, i) = Isum2 / N(i, j);    //式（84），得到Iji
58. }
59. }
60. }
62. **double** alpha = 0.01;    //表示σN的平方的倒数
63. **double** beta = 100;    //表示σg的平方的倒数
65. Mat\_<**double**> A(num\_images, num\_images); A.setTo(0);
66. Mat\_<**double**> b(num\_images, 1); b.setTo(0);
67. //计算由式87组成的方程组的系数
68. **for** (**int** i = 0; i < num\_images; ++i)
69. {
70. **for** (**int** j = 0; j < num\_images; ++j)
71. {
72. b(i, 0) += beta \* N(i, j);    //式87中等号右边的项
73. A(i, i) += beta \* N(i, j);    //式87中括号中的第二项
74. **if** (j == i) **continue**;
75. A(i, i) += 2 \* alpha \* I(i, j) \* I(i, j) \* N(i, j);    //式87中gi前的系数
76. A(i, j) -= 2 \* alpha \* I(i, j) \* I(j, i) \* N(i, j);    //式87中除gi以外的gj前的系数
77. }
78. }
79. //求解线性方程组，得到所有增益系数g的向量形式gains\_
80. solve(A, b, gains\_);    //A × gains\_ = b
81. //在终端显示运行时间
82. LOGLN("Exposure compensation, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
83. }

在计算得到了增益系数g后，就可以利用apply函数进行曝光补偿：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78883794) [copy](http://blog.csdn.net/zhaocj/article/details/78883794)

1. **void** GainCompensator::apply(**int** index, Point /\*corner\*/, Mat &image, **const** Mat &/\*mask\*/)
2. //index表示待拼接的图像索引
3. //image表示补偿得到的图像
4. {
5. image \*= gains\_(index, 0);    //增益系数乘以图像，就实现了曝光补偿
6. }
8. 分块补偿方法的feed函数：
9. **void** BlocksGainCompensator::feed(**const** vector<Point> &corners, **const** vector<Mat> &images,
10. **const** vector<pair<Mat,uchar> > &masks)
11. {
12. //确保各个输入参数的数量的一致性
13. CV\_Assert(corners.size() == images.size() && images.size() == masks.size());
15. **const** **int** num\_images = **static\_cast**<**int**>(images.size());    //表示待拼接的图像数量
17. vector<Size> bl\_per\_imgs(num\_images);    //表示每个图像的块数量
18. vector<Point> block\_corners;    //表示块的左上角在全景图像上的坐标
19. vector<Mat> block\_images;    //表示块图像
20. vector<pair<Mat,uchar> > block\_masks;    //表示块的掩码
22. // Construct blocks for gain compensator
23. **for** (**int** img\_idx = 0; img\_idx < num\_images; ++img\_idx)    //遍历所有图像，建立块
24. {
25. //bl\_per\_img表示当前图像的块数量，其中全局变量bl\_width\_和bl\_height\_表示块的宽和高，它们都被初始化为32，也就是块的理论尺寸为32×32
26. Size bl\_per\_img((images[img\_idx].cols + bl\_width\_ - 1) / bl\_width\_,
27. (images[img\_idx].rows + bl\_height\_ - 1) / bl\_height\_);
28. //bl\_width和bl\_height表示当前图像的块尺寸的实际宽和高
29. **int** bl\_width = (images[img\_idx].cols + bl\_per\_img.width - 1) / bl\_per\_img.width;
30. **int** bl\_height = (images[img\_idx].rows + bl\_per\_img.height - 1) / bl\_per\_img.height;
31. bl\_per\_imgs[img\_idx] = bl\_per\_img;    //赋值
32. //遍历图像的每个块
33. **for** (**int** by = 0; by < bl\_per\_img.height; ++by)
34. {
35. **for** (**int** bx = 0; bx < bl\_per\_img.width; ++bx)
36. {
37. //bl\_tl表示当前块的左上角坐标点
38. Point bl\_tl(bx \* bl\_width, by \* bl\_height);
39. //bl\_br表示当前块的右下角坐标点
40. Point bl\_br(min(bl\_tl.x + bl\_width, images[img\_idx].cols),
41. min(bl\_tl.y + bl\_height, images[img\_idx].rows));
42. //block\_corners矢量变量赋值
43. block\_corners.push\_back(corners[img\_idx] + bl\_tl);
44. //block\_images矢量变量赋值
45. block\_images.push\_back(images[img\_idx](Rect(bl\_tl, bl\_br)));
46. //block\_masks矢量变量赋值
47. block\_masks.push\_back(make\_pair(masks[img\_idx].first(Rect(bl\_tl, bl\_br)),
48. masks[img\_idx].second));
49. }
50. }
51. }
52. //实例化GainCompensator，每个块都应用增益补偿方法
53. GainCompensator compensator;
54. compensator.feed(block\_corners, block\_images, block\_masks);    //得到块补偿系数
55. vector<**double**> gains = compensator.gains();    //得到增益系数
56. //全局变量gain\_maps\_表示所有块的增益
57. gain\_maps\_.resize(num\_images);
59. Mat\_<**float**> ker(1, 3);    //表示分割线性滤波的核
60. ker(0,0) = 0.25; ker(0,1) = 0.5; ker(0,2) = 0.25;
62. **int** bl\_idx = 0;
63. **for** (**int** img\_idx = 0; img\_idx < num\_images; ++img\_idx)    //遍历所有的图像
64. {
65. Size bl\_per\_img = bl\_per\_imgs[img\_idx];    //提取出当前图像的块数量
66. gain\_maps\_[img\_idx].create(bl\_per\_img);    //创建当前图像的gain\_maps\_
67. //遍历当前图像的所有块，得到块的增益系数
68. **for** (**int** by = 0; by < bl\_per\_img.height; ++by)
69. **for** (**int** bx = 0; bx < bl\_per\_img.width; ++bx, ++bl\_idx)
70. gain\_maps\_[img\_idx](by, bx) = **static\_cast**<**float**>(gains[bl\_idx]);
71. //两次应用分割线性滤波，平滑同一图像的块增益系数
72. sepFilter2D(gain\_maps\_[img\_idx], gain\_maps\_[img\_idx], CV\_32F, ker, ker);
73. sepFilter2D(gain\_maps\_[img\_idx], gain\_maps\_[img\_idx], CV\_32F, ker, ker);
74. }
75. }

应用分块补偿，对图像进行曝光补偿的apply函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78883794) [copy](http://blog.csdn.net/zhaocj/article/details/78883794)

1. **void** BlocksGainCompensator::apply(**int** index, Point /\*corner\*/, Mat &image, **const** Mat &/\*mask\*/)
2. {
3. CV\_Assert(image.type() == CV\_8UC3);    //确保image的正确
5. Mat\_<**float**> gain\_map;
6. //得到该图像的块增益系数
7. **if** (gain\_maps\_[index].size() == image.size())
8. gain\_map = gain\_maps\_[index];
9. **else**
10. resize(gain\_maps\_[index], gain\_map, image.size(), 0, 0, INTER\_LINEAR);
11. //对该图像的红、绿、蓝分量进行增益补偿
12. **for** (**int** y = 0; y < image.rows; ++y)
13. {
14. **const** **float**\* gain\_row = gain\_map.ptr<**float**>(y);
15. Point3\_<uchar>\* row = image.ptr<Point3\_<uchar> >(y);
16. **for** (**int** x = 0; x < image.cols; ++x)
17. {
18. row[x].x = saturate\_cast<uchar>(row[x].x \* gain\_row[x]);
19. row[x].y = saturate\_cast<uchar>(row[x].y \* gain\_row[x]);
20. row[x].z = saturate\_cast<uchar>(row[x].z \* gain\_row[x]);
21. }
22. }
23. }

5.3 应用

下面我们曝光补偿的应用：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78883794) [copy](http://blog.csdn.net/zhaocj/article/details/78883794)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. #include <iomanip>
23. **using** **namespace** cv;
24. **using** **namespace** std;
25. **using** **namespace** detail;
27. **int** main(**int** argc, **char**\*\* argv)
28. {
29. vector<Mat> imgs;    //输入图像
30. Mat img = imread("1.jpg");
31. imgs.push\_back(img);
32. img = imread("2.jpg");
33. imgs.push\_back(img);
35. Ptr<FeaturesFinder> finder;    //特征检测
36. finder = **new** SurfFeaturesFinder();
37. vector<ImageFeatures> features(2);
38. (\*finder)(imgs[0], features[0]);
39. (\*finder)(imgs[1], features[1]);
41. vector<MatchesInfo> pairwise\_matches;    //特征匹配
42. BestOf2NearestMatcher matcher(**false**, 0.3f, 6, 6);
43. matcher(features, pairwise\_matches);
45. HomographyBasedEstimator estimator;    //相机参数评估
46. vector<CameraParams> cameras;
47. estimator(features, pairwise\_matches, cameras);
48. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
49. {
50. Mat R;
51. cameras[i].R.convertTo(R, CV\_32F);
52. cameras[i].R = R;
53. }
55. Ptr<detail::BundleAdjusterBase> adjuster;    //光束平差法，精确相机参数
56. adjuster = **new** detail::BundleAdjusterReproj();
57. adjuster->setConfThresh(1);
58. (\*adjuster)(features, pairwise\_matches, cameras);
60. vector<Mat> rmats;
61. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
62. rmats.push\_back(cameras[i].R.clone());
63. waveCorrect(rmats, WAVE\_CORRECT\_HORIZ);    //波形校正
64. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
65. cameras[i].R = rmats[i];
67. //图像映射变换
68. vector<Point> corners(2);
69. vector<Mat> masks\_warped(2);
70. vector<Mat> images\_warped(2);
71. vector<Mat> masks(2);
72. **for** (**int** i = 0; i < 2; ++i)
73. {
74. masks[i].create(imgs[i].size(), CV\_8U);
75. masks[i].setTo(Scalar::all(255));
76. }
77. Ptr<WarperCreator> warper\_creator;
78. warper\_creator = **new** cv::PlaneWarper();    //平面投影
79. Ptr<RotationWarper> warper = warper\_creator->create(**static\_cast**<**float**>(cameras[0].focal));
80. **for** (**int** i = 0; i < 2; ++i)
81. {
82. Mat\_<**float**> K;
83. cameras[i].K().convertTo(K, CV\_32F);
84. corners[i] = warper->warp(imgs[i], K, cameras[i].R, INTER\_LINEAR, BORDER\_REFLECT, images\_warped[i]);
85. warper->warp(masks[i], K, cameras[i].R, INTER\_NEAREST, BORDER\_CONSTANT, masks\_warped[i]);
86. }
88. //创建曝光补偿器，应用增益补偿方法
89. Ptr<ExposureCompensator> compensator =
90. ExposureCompensator::createDefault(ExposureCompensator::GAIN);
91. compensator->feed(corners, images\_warped, masks\_warped);    //得到曝光补偿器
92. **for**(**int** i=0;i<2;++i)    //应用曝光补偿器，对图像进行曝光补偿
93. {
94. compensator->apply(i, corners[i], images\_warped[i], masks\_warped[i]);
95. }
97. imwrite("warped1.jpg", images\_warped[0]);    //存储图像
98. imwrite("warped2.jpg", images\_warped[1]);
100. **return** 0;
101. }

我们对图11中的两幅图像进行曝光补偿。很明显，两幅图像的曝光程度是不同的，尤其是两者的重叠区域，左侧要比右侧暗一些。图12是曝光补偿后的图像，经过补偿后，左侧图像变亮了，而右侧图像却变暗了。

图11 原图

图12 曝光补偿的结果

# Opencv2.4.9源码分析——Stitching（六）

6、寻找接缝线

6.1 原理

拼接图像的另一个重要的步骤是找到图像重叠部分内的一条接缝线，该接缝是重叠部分最相似的像素的连线。当确定了接缝线后，在重叠部分，线的一侧只选择该侧的图像部分，线的另一侧只选择这一侧的图像部分，而不是把重叠部分的两幅图像简单融合起来。这么做的目的可以避免图像的模糊及伪像。

目前，常用的寻找接缝线的方法有三种：逐点法、动态规划法和图割法。

逐点法比较简单，它的原理就是重叠部分内的像素距离哪个图像更近，就用哪个图像中的相应像素值。这种方法并不是基于像素值，只是基于距离准则。所以这种方法并不能真正得到最佳的接缝线，寻找的结果比较粗糙原始，仅仅是能够分割重叠部分而已。

动态规划法可以快速的找到最佳的接缝线，并且所需的内存较少，因此该方法很适合在移动终端设备上进行高清晰度的全景拼接。

设图像1和图像2之间有重叠部分，我们需要得到它们之间的最佳接缝，首先定义重叠部分的误差表面函数：

http://img.blog.csdn.net/20180101111326814（88）

式中，I1，I2表示两幅图像各自的重叠部分，││,||表示范数。

一般来说，接缝线有三个限制条件：一是如果重叠区域是宽大于高，则接缝是横向走向的，而如果重叠区域是宽小于高，则接缝是纵向走向的，也就是要保证接缝线要有一定的长度；二是如果是横向的接缝，则不允许有绝对垂直的接缝线，而如果是纵向的接缝，则不允许有绝对水平的接缝线；三是重叠区域是矩形，接缝线从矩形的一边出发，必须到达与该边平行的另一边结束。

假设重叠区域的宽小于高，则接缝线是垂直的，我们需要横向遍历e值，并且计算所有可能到达当前像素(i, j)的路径的累积最小误差E：

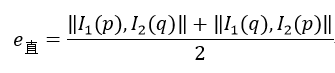
http://img.blog.csdn.net/20180101111339202（89）

式中，min中的三项内容分别表示当前像素与其左上侧、上侧和右上侧的E。在E中最后一行的最小值表明已经到达了最小垂直路径的尽头，那么我们就可以追溯，从而得到最佳路径，即接缝线。

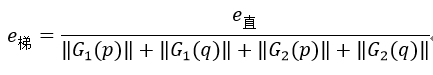
同理，重叠区域的宽大于高也能够做出类似的计算过程。

我们可以看出，动态规划算法非常适用于对式89的求解，这也是该寻找最佳接缝线方法名称的由来。

对于彩色图像，计算式88可以有两种方法：直接法和梯度法。设p和q是两个相邻像素，直接法的计算公式为：

（90）

梯度法的计算公式为：

（91）

式中，G表示灰度图像的p到q方向上的梯度。

最后一种方法是图割法。它的主要思想是将图像看成是一个有向图G={V, E}，V表示节点，E表示边。节点分为两种：普通节点和终端节点。图像的像素可以作为普通节点，图像的聚类结果可以看成是终端节点，即某一聚类中的普通节点会汇聚于一个终端节点，一般图只有两个终端节点：s和t，也就是说一般图割法只能够把图像分割成两类S和T。边指的是节点之间的连接，对于普通节点，只有相邻节点可以连接，该边被称为n-links，而所有普通节点都可与属于它们的终端节点之间有连接，该边被称为t-links。用最大流图割法可以得到哪些普通节点属于终端节点s，那么剩下的普通节点就会属于t。

Opencv中所实现的最大流图割算法是基于Boykov和Kolmogorov所提出的算法。该算法是以s和t为根，以得到的两棵无不相交的搜索树S和T为目标，它迭代的重复以下三个阶段：

生长阶段：搜索树S和T的生长，直到两棵树生长到了一起，形成了从s到t的通路（即为流）；

增广阶段：该通路被增广，使相连的两棵树又重新分开；

收养阶段：搜索树S和T收养孤立的节点（称为孤儿）。

在生长阶段，如果找不到任何一条从s到t的通路，则迭代结束。在增广阶段，开始孤儿集合是空集，而该阶段结束后，可能会出现一些孤儿。在该阶段，我们需要得到通路上最小的容量值（也称为边的权值），然后所有的容量值都减去该值（即通过流量更新容量值），则该通路上至少会有一个权值为0的边（称为饱和边），假设饱和边上的两个节点为p和q，并且在正向通路中是由p流向q，如果p和q都属于树S，则q为孤儿，如果p和q都属于树T，则p为孤儿。在收养阶段，检查孤儿集合中的所有孤儿的邻域像素，连接那些它们的边的容量值不为0的像素，如果最终可以从该孤儿连通到s或t，则该孤儿就属于树S或树T，否则由该孤儿组成的树就属于自由节点集，在下次迭代时的生长阶段，树S和树T就通过连接自由节点集中的节点来使两棵树生长发育的。

如何用最大流图割法寻找接缝线呢？接缝线把重叠区域分割成了两个区域，那么我们通过最大流图割法得到树S和树T，即得到了被分割的两个区域，则区域之间的接缝线自然就可以得到。

普通节点之间的初始容量值（即边的权值）也可以通过直接法和梯度法计算得到，它们的计算公式分别与式90和式91相同。终端节点与普通节点之间的初始容量值可以定义为一个很大的值，当重叠区域中的节点只属于图像1或图像2时，节点与相应的终端节点的边赋予该值，其他情况都置为0，表示它们与终端节点无直接连接。

6.2 源码

SeamFinder类是寻找缝的基类，该类内有一个最重要的虚函数是find，它的作用是寻找缝：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **virtual** **void** find(**const** std::vector<Mat> &src, **const** std::vector<Point> &corners,
2. std::vector<Mat> &masks) = 0;

它的三个输入参数分别为：src表示待拼接的图像，corners表示待拼接图像在最终的全景图像坐标系内的左上角的坐标位置，masks输入时表示上一步得到的映射变换掩码，输出时表示接缝掩码，也就是该参数是要更新输出的，输出时如果像素值更新为0了，则表示该像素是在接缝的另一侧，拼接时是不需要该像素的。

基于SeamFinder类有四个子类，分别对应于四种不同的寻找缝的算法——NoSeamFinder（无需寻找缝）、PairwiseSeamFinder（逐点寻找缝算法）、DpSeamFinder（动态规划寻找缝算法）和GraphCutSeamFinder（图割寻找缝算法）。

我们先来介绍PairwiseSeamFinder类。逐点寻找缝算法又可分为不同的方法，Opencv只实现了Voronoi图方法，实现该方法的类为VoronoiSeamFinder类，它是PairwiseSeamFinder类的子类。VoronoiSeamFinder类的find函数为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** VoronoiSeamFinder::find(**const** vector<Size> &sizes, **const** vector<Point> &corners,
2. vector<Mat> &masks)
3. {
4. LOGLN("Finding seams...");
5. **if** (sizes.size() == 0)    //没有图像，则返回
6. **return**;
8. #if ENABLE\_LOG
9. int64 t = getTickCount();    //计时
10. #endif
11. //参数赋值
12. sizes\_ = sizes;
13. corners\_ = corners;
14. masks\_ = masks;
15. run();    //调用run函数
16. //计时输出
17. LOGLN("Finding seams, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
18. }

VoronoiSeamFinder类没有实现run函数，因此这里的find函数调用的是它的父类PairwiseSeamFinder类的run函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** PairwiseSeamFinder::run()
2. {
3. //遍历图像对
4. **for** (**size\_t** i = 0; i < sizes\_.size() - 1; ++i)
5. {
6. **for** (**size\_t** j = i + 1; j < sizes\_.size(); ++j)
7. {
8. Rect roi;    //表示图像间的重叠部分
9. //如果图像i和图j之间有重叠的区域，则执行findInPair函数
10. **if** (overlapRoi(corners\_[i], corners\_[j], sizes\_[i], sizes\_[j], roi))
11. findInPair(i, j, roi);
12. }
13. }
14. }

findInPair函数表示寻找两幅图像之间的接缝：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** VoronoiSeamFinder::findInPair(**size\_t** first, **size\_t** second, Rect roi)
2. //first和second表示两幅图像的尺寸大小，roi表示图像间重叠部分
3. {
4. **const** **int** gap = 10;
5. //分别表示图像first和图像second的重叠部分的掩码，它们要比roi四周多gap个距离
6. Mat submask1(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_8U);
7. Mat submask2(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_8U);
9. Size img1 = sizes\_[first], img2 = sizes\_[second];    //得到两幅图像的尺寸
10. Mat mask1 = masks\_[first], mask2 = masks\_[second];    //得到两幅图像的掩码
11. Point tl1 = corners\_[first], tl2 = corners\_[second];    //得到两幅图像的左上角坐标
13. // Cut submasks with some gap
14. //遍历重叠部分，得到submask1和submask2
15. **for** (**int** y = -gap; y < roi.height + gap; ++y)
16. {
17. **for** (**int** x = -gap; x < roi.width + gap; ++x)
18. {
19. //y1和x1表示重叠部分相对于图像first的坐标
20. **int** y1 = roi.y - tl1.y + y;
21. **int** x1 = roi.x - tl1.x + x;
22. //把图像first的掩码赋值给submask1
23. //表示当前坐标在图像first内
24. **if** (y1 >= 0 && x1 >= 0 && y1 < img1.height && x1 < img1.width)
25. submask1.at<uchar>(y + gap, x + gap) = mask1.at<uchar>(y1, x1);
26. **else**    //表示当前坐标不在图像first内
27. submask1.at<uchar>(y + gap, x + gap) = 0;
28. //y2和x2表示重叠部分相对于图像second的坐标
29. **int** y2 = roi.y - tl2.y + y;
30. **int** x2 = roi.x - tl2.x + x;
31. //把图像second的掩码赋值给submask2
32. //表示当前坐标在图像second内
33. **if** (y2 >= 0 && x2 >= 0 && y2 < img2.height && x2 < img2.width)
34. submask2.at<uchar>(y + gap, x + gap) = mask2.at<uchar>(y2, x2);
35. **else**    //表示当前坐标不在图像second内
36. submask2.at<uchar>(y + gap, x + gap) = 0;
37. }
38. }
39. //collision表示submask1和submask2交集，roi区域可能有些区域是无效的部分（由于映射的原因），而collision（像素值为1的部分）则是图像first和图像second的真正有意义部分的重叠区域，我们是需要在collision区域内寻找接缝的
40. Mat collision = (submask1 != 0) & (submask2 != 0);
41. //复制submask1和submask2为unique1和unique2，并且把collision部分的像素清零
42. Mat unique1 = submask1.clone(); unique1.setTo(0, collision);
43. Mat unique2 = submask2.clone(); unique2.setTo(0, collision);
45. Mat dist1, dist2;    //表示距离矩阵
46. //分别得到unique1和unique2内为0的像素值与最近的非0值之间的L1距离，显然为0的区域肯定要比collision大
47. distanceTransform(unique1 == 0, dist1, CV\_DIST\_L1, 3);
48. distanceTransform(unique2 == 0, dist2, CV\_DIST\_L1, 3);
49. //得到缝矩阵，dist1 < dist2成立，说明离first图像近，则seam内相应的像素为1，否则为0
50. Mat seam = dist1 < dist2;
51. //遍历roi内的所有像素
52. **for** (**int** y = 0; y < roi.height; ++y)
53. {
54. **for** (**int** x = 0; x < roi.width; ++x)
55. {
56. //如果seam内的像素值为1，说明collision区域内的相应像素离图像first近，所以该点的值选择图像first的像素值，因此需把mask2清零，反之把mask1清零。也就是在重叠部分，哪些区域应该属于图像first，哪些区域属于图像second。
57. **if** (seam.at<uchar>(y + gap, x + gap))
58. mask2.at<uchar>(roi.y - tl2.y + y, roi.x - tl2.x + x) = 0;
59. **else**
60. mask1.at<uchar>(roi.y - tl1.y + y, roi.x - tl1.x + x) = 0;
61. }
62. }
63. }

DpSeamFinder类利用动态规划算法寻找缝，它的构造函数为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. DpSeamFinder::DpSeamFinder(CostFunction costFunc) : costFunc\_(costFunc) {}

其中CostFunction为：

enum CostFunction { COLOR, COLOR\_GRAD };

它表示动态规划寻找接缝的两种不同方法，直接法和梯度法，默认是直接法。

DpSeamFinder类的find函数为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::find(**const** vector<Mat> &src, **const** vector<Point> &corners, vector<Mat> &masks)
2. {
3. LOGLN("Finding seams...");
4. #if ENABLE\_LOG
5. int64 t = getTickCount();    //用于计时
6. #endif
8. **if** (src.size() == 0)    //没有图像，则返回
9. **return**;
11. vector<pair<**size\_t**, **size\_t**> > pairs;    //表示图像对
13. **for** (**size\_t** i = 0; i+1 < src.size(); ++i)
14. **for** (**size\_t** j = i+1; j < src.size(); ++j)
15. pairs.push\_back(make\_pair(i, j));    //每两幅图像组成一个图像对
16. //根据图像位置（基于左上角坐标），对图像对进行排序
17. sort(pairs.begin(), pairs.end(), ImagePairLess(src, corners));
18. reverse(pairs.begin(), pairs.end());    //逆序
20. **for** (**size\_t** i = 0; i < pairs.size(); ++i)    //遍历图像对，
21. {
22. **size\_t** i0 = pairs[i].first, i1 = pairs[i].second;
23. //寻找i0和i1这两幅图像的接缝，该函数在后面有介绍
24. process(src[i0], src[i1], corners[i0], corners[i1], masks[i0], masks[i1]);
25. }
27. LOGLN("Finding seams, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
28. }

寻找两幅图像间的接缝：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::process(
2. **const** Mat &image1, **const** Mat &image2, Point tl1, Point tl2,
3. Mat &mask1, Mat &mask2)
4. //image1和image2分别表示待处理的两幅图像
5. //tl1和tl2分别表示这两幅图像的左上角坐标
6. //mask1和mask2分别表示这两幅图像的掩码
7. {
8. CV\_Assert(image1.size() == mask1.size());    //确保图像与掩码尺寸相同
9. CV\_Assert(image2.size() == mask2.size());
10. //intersectTl和intersectBr分别表示image1和image2的交集区域（即重叠部分）的左上角和右下角坐标
11. Point intersectTl(std::max(tl1.x, tl2.x), std::max(tl1.y, tl2.y));
13. Point intersectBr(std::min(tl1.x + image1.cols, tl2.x + image2.cols),
14. std::min(tl1.y + image1.rows, tl2.y + image2.rows));
15. //如果if条件成立，则说明image1和image2没有重叠区域，因此退出该函数
16. **if** (intersectTl.x >= intersectBr.x || intersectTl.y >= intersectBr.y)
17. **return**; // there are no conflicts
18. //unionTl\_和unionBr\_分别表示image1和image2的并集区域（即外围最大部分的矩形）的左上角和右下角坐标
19. unionTl\_ = Point(std::min(tl1.x, tl2.x), std::min(tl1.y, tl2.y));
21. unionBr\_ = Point(std::max(tl1.x + image1.cols, tl2.x + image2.cols),
22. std::max(tl1.y + image1.rows, tl2.y + image2.rows));
23. //得到image1和image2的并集区域的尺寸大小
24. unionSize\_ = Size(unionBr\_.x - unionTl\_.x, unionBr\_.y - unionTl\_.y);
25. //定义两幅图像的掩码大小，先清零
26. mask1\_ = Mat::zeros(unionSize\_, CV\_8U);
27. mask2\_ = Mat::zeros(unionSize\_, CV\_8U);
28. //tmp为mask1\_中image1的区域
29. Mat tmp = mask1\_(Rect(tl1.x - unionTl\_.x, tl1.y - unionTl\_.y, mask1.cols, mask1.rows));
30. mask1.copyTo(tmp);    //把image1的掩码mask1赋值给mask1\_
31. //tmp为mask2\_中image2的区域
32. tmp = mask2\_(Rect(tl2.x - unionTl\_.x, tl2.y - unionTl\_.y, mask2.cols, mask2.rows));
33. mask2.copyTo(tmp);    //把image2的掩码mask2赋值给mask2\_
35. // find both images contour masks
36. //在并集中分别得到边界掩码contour1mask\_和contour2mask\_
37. contour1mask\_ = Mat::zeros(unionSize\_, CV\_8U);    //先清零
38. contour2mask\_ = Mat::zeros(unionSize\_, CV\_8U);
39. //遍历并集区域
40. **for** (**int** y = 0; y < unionSize\_.height; ++y)
41. {
42. **for** (**int** x = 0; x < unionSize\_.width; ++x)
43. {
44. **if** (mask1\_(y, x) &&
45. ((x == 0 || !mask1\_(y, x-1)) || (x == unionSize\_.width-1 || !mask1\_(y, x+1)) ||
46. (y == 0 || !mask1\_(y-1, x)) || (y == unionSize\_.height-1 || !mask1\_(y+1, x))))
47. {
48. contour1mask\_(y, x) = 255;    //得到image1的边界掩码
49. }
51. **if** (mask2\_(y, x) &&
52. ((x == 0 || !mask2\_(y, x-1)) || (x == unionSize\_.width-1 || !mask2\_(y, x+1)) ||
53. (y == 0 || !mask2\_(y-1, x)) || (y == unionSize\_.height-1 || !mask2\_(y+1, x))))
54. {
55. contour2mask\_(y, x) = 255;    //得到image2的边界掩码
56. }
57. }
58. }
59. //以下三个函数在后面都有介绍
60. findComponents();
62. findEdges();
64. resolveConflicts(image1, image2, tl1, tl2, mask1, mask2);
65. }

利用漫水填充法分割图像寻找组件，不同的组件对应于图像的不同区域：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::findComponents()
2. {
3. // label all connected components and get information about them
5. ncomps\_ = 0;    //表示组件的数量
6. labels\_.create(unionSize\_);    //定义标签的大小
7. states\_.clear();    //状态清空
8. tls\_.clear();    //tls表示各种组件的左上角坐标
9. brs\_.clear();    //brs表示各种组件的右下角坐标
10. contours\_.clear();    //组件区域的边界
11. //遍历两幅图像的并集区域，为labels\_赋值
12. **for** (**int** y = 0; y < unionSize\_.height; ++y)
13. {
14. **for** (**int** x = 0; x < unionSize\_.width; ++x)
15. {
16. **if** (mask1\_(y, x) && mask2\_(y, x))    //表示两幅图像的公共交集的部分
17. labels\_(y, x) = numeric\_limits<**int**>::max();
18. **else** **if** (mask1\_(y, x))    //表示只有image1的部分
19. labels\_(y, x) = numeric\_limits<**int**>::max()-1;
20. **else** **if** (mask2\_(y, x))    //表示只有image2的部分
21. labels\_(y, x) = numeric\_limits<**int**>::max()-2;
22. **else**    //表示没有任何图像信息
23. labels\_(y, x) = 0;
24. }
25. }
26. //再次遍历两幅图像的并集区域，
27. **for** (**int** y = 0; y < unionSize\_.height; ++y)
28. {
29. **for** (**int** x = 0; x < unionSize\_.width; ++x)
30. {
31. //表示当前像素是并集区域内有图像信息的
32. **if** (labels\_(y, x) >= numeric\_limits<**int**>::max()-2)
33. {
34. **if** (labels\_(y, x) == numeric\_limits<**int**>::max())    //交集部分
35. states\_.push\_back(INTERS);    //状态赋值
36. **else** **if** (labels\_(y, x) == numeric\_limits<**int**>::max()-1)    //只有image1部分
37. states\_.push\_back(FIRST);    //状态赋值
38. **else** **if** (labels\_(y, x) == numeric\_limits<**int**>::max()-2)    //只有image2部分
39. states\_.push\_back(SECOND);    //状态赋值
40. //利用漫水填充算法，从当前像素点出发，填充++ncomps\_值
41. floodFill(labels\_, Point(x, y), ++ncomps\_);
42. //存入当前像素点，表示当前组件的左上角坐标
43. tls\_.push\_back(Point(x, y));
44. //表示当前组件的右下角坐标，+1的含义是用于区分左上角坐标
45. brs\_.push\_back(Point(x+1, y+1));
46. //因为已经得到了一个组件，所以这时需要在contours\_内预留一段空间，用于存储当前组件的边界像素
47. contours\_.push\_back(vector<Point>());
48. }
50. **if** (labels\_(y, x))    //表示当前像素属于某幅图像
51. {
52. **int** l = labels\_(y, x);    //当前标签
53. **int** ci = l-1;    //标签值减1，表示当前组件
55. //确定当前组件的左上角和右下角坐标，在双重for循环体内，不断进入该if语句，所以实现了当前组件的左上角和右下角坐标的不断扩展，最终得到准确的值
56. tls\_[ci].x = std::min(tls\_[ci].x, x);
57. tls\_[ci].y = std::min(tls\_[ci].y, y);
58. brs\_[ci].x = std::max(brs\_[ci].x, x+1);
59. brs\_[ci].y = std::max(brs\_[ci].y, y+1);
60. //if条件成立，说明当前像素为当前组件的边界
61. **if** ((x == 0 || labels\_(y, x-1) != l) || (x == unionSize\_.width-1 || labels\_(y, x+1) != l) ||
62. (y == 0 || labels\_(y-1, x) != l) || (y == unionSize\_.height-1 || labels\_(y+1, x) != l))
63. {
64. contours\_[ci].push\_back(Point(x, y));    //存入当前组件的边界像素
65. }
66. }
67. }
68. }
69. }

得到组件间的边缘：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::findEdges()
2. {
3. // find edges between components
4. //wedges表示边缘权值，即边缘像素的数量，pair中的两个参数表示两个组件的索引
5. map<pair<**int**, **int**>, **int**> wedges; // weighted edges
6. //遍历组件，初始化wedges为0
7. **for** (**int** ci = 0; ci < ncomps\_-1; ++ci)
8. {
9. **for** (**int** cj = ci+1; cj < ncomps\_; ++cj)
10. {
11. wedges[make\_pair(ci, cj)] = 0;
12. wedges[make\_pair(cj, ci)] = 0;
13. }
14. }
16. **for** (**int** ci = 0; ci < ncomps\_; ++ci)    //遍历组件
17. {
18. **for** (**size\_t** i = 0; i < contours\_[ci].size(); ++i)    //遍历当前组件的边界
19. {
20. **int** x = contours\_[ci][i].x;    //边缘像素坐标
21. **int** y = contours\_[ci][i].y;
22. **int** l = ci + 1;    //当前组件的标签
23. //if条件成立，表示当前像素为左侧边缘，此时当前组件与其左侧组件间的wedges权值加1
24. **if** (x > 0 && labels\_(y, x-1) && labels\_(y, x-1) != l)
25. {
26. wedges[make\_pair(ci, labels\_(y, x-1)-1)]++;
27. wedges[make\_pair(labels\_(y, x-1)-1, ci)]++;
28. }
29. //if条件成立，表示当前像素为上侧边缘，此时当前组件与其上侧组件间的wedges权值加1
30. **if** (y > 0 && labels\_(y-1, x) && labels\_(y-1, x) != l)
31. {
32. wedges[make\_pair(ci, labels\_(y-1, x)-1)]++;
33. wedges[make\_pair(labels\_(y-1, x)-1, ci)]++;
34. }
35. //if条件成立，表示当前像素为右侧边缘，此时当前组件与其右侧组件间的wedges权值加1
36. **if** (x < unionSize\_.width-1 && labels\_(y, x+1) && labels\_(y, x+1) != l)
37. {
38. wedges[make\_pair(ci, labels\_(y, x+1)-1)]++;
39. wedges[make\_pair(labels\_(y, x+1)-1, ci)]++;
40. }
41. //if条件成立，表示当前像素为下侧边缘，此时当前组件与其下侧组件间的wedges权值加1
42. **if** (y < unionSize\_.height-1 && labels\_(y+1, x) && labels\_(y+1, x) != l)
43. {
44. wedges[make\_pair(ci, labels\_(y+1, x)-1)]++;
45. wedges[make\_pair(labels\_(y+1, x)-1, ci)]++;
46. }
47. }
48. }
49. //edges\_表示两个组件是相邻的，因为它们之间有边界
50. edges\_.clear();    //清空
51. //遍历wedges
52. **for** (**int** ci = 0; ci < ncomps\_-1; ++ci)
53. {
54. **for** (**int** cj = ci+1; cj < ncomps\_; ++cj)
55. {
56. //得到组件ci和cj之间的wedges
57. map<pair<**int**, **int**>, **int**>::iterator itr = wedges.find(make\_pair(ci, cj));
58. **if** (itr != wedges.end() && itr->second > 0)    //表示ci和cj是相邻的
59. edges\_.insert(itr->first);    //ci和cj存入edges中
60. //得到组件cj和ci之间的wedges
61. itr = wedges.find(make\_pair(cj, ci));
62. **if** (itr != wedges.end() && itr->second > 0)    //表示cj和ci是相邻的
63. edges\_.insert(itr->first);    //cj和ci存入edges中
64. }
65. }
66. }

具体计算接缝的部分：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::resolveConflicts(
2. **const** Mat &image1, **const** Mat &image2, Point tl1, Point tl2, Mat &mask1, Mat &mask2)
3. {
4. **if** (costFunc\_ == COLOR\_GRAD)    //梯度法
5. computeGradients(image1, image2);    //调用computeGradients，计算梯度
7. // resolve conflicts between components
9. **bool** hasConflict = **true**;    //标识变量
10. **while** (hasConflict)
11. {
12. **int** c1 = 0, c2 = 0;
13. hasConflict = **false**;    //变换标识
14. //遍历edges\_，得到相连组件
15. **for** (set<pair<**int**, **int**> >::iterator itr = edges\_.begin(); itr != edges\_.end(); ++itr)
16. {
17. c1 = itr->first;    //得到连接边缘的一个组件
18. c2 = itr->second;    //得到连接边缘的另一个组件
19. //c1有交集部分，并且它不是与c2组件的交集
20. **if** ((states\_[c1] & INTERS) && (states\_[c1] & (~INTERS)) != states\_[c2])
21. {
22. hasConflict = **true**;    //赋值为true
23. **break**;    //退出for循环
24. }
25. }
27. **if** (hasConflict)
28. {
29. **int** l1 = c1+1, l2 = c2+1;    //表示c1和c2的标签
30. //表示c1只有一个邻近组件，说明c1不是交集，不需要在该组件寻找接缝线
31. **if** (hasOnlyOneNeighbor(c1))
32. {
33. // if the first components has only one adjacent component
34. //遍历c1区域，把标签为l1的像素改为l2，即合并两个组件
35. **for** (**int** y = tls\_[c1].y; y < brs\_[c1].y; ++y)
36. **for** (**int** x = tls\_[c1].x; x < brs\_[c1].x; ++x)
37. **if** (labels\_(y, x) == l1)
38. labels\_(y, x) = l2;
39. //改变组件c1的状态
40. states\_[c1] = states\_[c2] == FIRST ? SECOND : FIRST;
41. }
42. //表示c1有不止一个邻近组件，说明c1是交集部分，要在该组件寻找接缝线
43. **else**
44. {
45. // if the first component has more than one adjacent component
47. Point p1, p2;    //表示接缝的起点和终点
48. //利用getSeamTips函数得到p1和p2，该函数在后面给出介绍
49. **if** (getSeamTips(c1, c2, p1, p2))
50. {
51. vector<Point> seam;    //表示接缝
52. **bool** isHorizontalSeam;    //表示接缝的基本走向
53. //利用estimateSeam函数得到接缝seam，该函数在后面给出介绍
54. **if** (estimateSeam(image1, image2, tl1, tl2, c1, p1, p2, seam, isHorizontalSeam))
55. //更新标签，该函数在后面给出介绍
56. updateLabelsUsingSeam(c1, c2, seam, isHorizontalSeam);
57. }
58. //重新为c1的状态赋值
59. states\_[c1] = states\_[c2] == FIRST ? INTERS\_SECOND : INTERS\_FIRST;
60. }
62. **const** **int** c[] = {c1, c2};    //表示组件
63. **const** **int** l[] = {l1, l2};    //表示标签
65. **for** (**int** i = 0; i < 2; ++i)    //遍历c1和c2两个组件
66. {
67. // update information about the (i+1)-th component
68. //得到当前组件的左上角和右下角坐标
69. **int** x0 = tls\_[c[i]].x, x1 = brs\_[c[i]].x;
70. **int** y0 = tls\_[c[i]].y, y1 = brs\_[c[i]].y;
71. //重新赋值
72. tls\_[c[i]] = Point(numeric\_limits<**int**>::max(), numeric\_limits<**int**>::max());
73. brs\_[c[i]] = Point(numeric\_limits<**int**>::min(), numeric\_limits<**int**>::min());
74. contours\_[c[i]].clear();    //当前组件的边缘像素清零
76. **for** (**int** y = y0; y < y1; ++y)    //遍历当前组件
77. {
78. **for** (**int** x = x0; x < x1; ++x)
79. {
80. **if** (labels\_(y, x) == l[i])
81. {
82. //更新当前组件的区域面积
83. tls\_[c[i]].x = std::min(tls\_[c[i]].x, x);
84. tls\_[c[i]].y = std::min(tls\_[c[i]].y, y);
85. brs\_[c[i]].x = std::max(brs\_[c[i]].x, x+1);
86. brs\_[c[i]].y = std::max(brs\_[c[i]].y, y+1);
88. **if** ((x == 0 || labels\_(y, x-1) != l[i]) || (x == unionSize\_.width-1 || labels\_(y, x+1) != l[i]) ||
89. (y == 0 || labels\_(y-1, x) != l[i]) || (y == unionSize\_.height-1 || labels\_(y+1, x) != l[i]))
90. {
91. //给出当前组件的边缘像素
92. contours\_[c[i]].push\_back(Point(x, y));
93. }
94. }
95. }
96. }
97. }
99. // remove edges
100. //删除c1和c2所形成的边缘
101. edges\_.erase(make\_pair(c1, c2));
102. edges\_.erase(make\_pair(c2, c1));
103. }
104. }
106. // update masks
107. //更新掩码
108. **int** dx1 = unionTl\_.x - tl1.x, dy1 = unionTl\_.y - tl1.y;    //图像1的左上角坐标
109. **int** dx2 = unionTl\_.x - tl2.x, dy2 = unionTl\_.y - tl2.y;    //图像2的左上角坐标
110. //遍历mask2，更新掩码2
111. **for** (**int** y = 0; y < mask2.rows; ++y)
112. {
113. **for** (**int** x = 0; x < mask2.cols; ++x)
114. {
115. **int** l = labels\_(y - dy2, x - dx2);    //当前像素的标签
116. //if成立，说明当前像素应该用图像1的内容，所以掩码2清零
117. **if** (l > 0 && (states\_[l-1] & FIRST) && mask1.at<uchar>(y - dy2 + dy1, x - dx2 + dx1))
118. mask2.at<uchar>(y, x) = 0;
119. }
120. }
121. //遍历mask1，更新掩码1
122. **for** (**int** y = 0; y < mask1.rows; ++y)
123. {
124. **for** (**int** x = 0; x < mask1.cols; ++x)
125. {
126. **int** l = labels\_(y - dy1, x - dx1);    //当前像素的标签
127. //if成立，说明当前像素应该用图像2的内容，所以掩码1清零
128. **if** (l > 0 && (states\_[l-1] & SECOND) && mask2.at<uchar>(y - dy1 + dy2, x - dx1 + dx2))
129. mask1.at<uchar>(y, x) = 0;
130. }
131. }
132. }

计算彩色图像image1和image2的水平和垂直梯度：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::computeGradients(**const** Mat &image1, **const** Mat &image2)
2. {
3. //确保image1和image2是彩色图像
4. CV\_Assert(image1.channels() == 3 || image1.channels() == 4);
5. CV\_Assert(image2.channels() == 3 || image2.channels() == 4);
6. CV\_Assert(costFunction() == COLOR\_GRAD);    //确保是梯度法
8. Mat gray;
9. //把彩色图像image1转换为灰度图像
10. **if** (image1.channels() == 3)
11. cvtColor(image1, gray, CV\_BGR2GRAY);
12. **else** **if** (image1.channels() == 4)
13. cvtColor(image1, gray, CV\_BGRA2GRAY);
15. Sobel(gray, gradx1\_, CV\_32F, 1, 0);    //得到image1的水平梯度gradx1\_
16. Sobel(gray, grady1\_, CV\_32F, 0, 1);    //得到image1的垂直梯度grady1\_
17. //把彩色图像image2转换为灰度图像
18. **if** (image2.channels() == 3)
19. cvtColor(image2, gray, CV\_BGR2GRAY);
20. **else** **if** (image2.channels() == 4)
21. cvtColor(image2, gray, CV\_BGRA2GRAY);
23. Sobel(gray, gradx2\_, CV\_32F, 1, 0);    //得到image2的水平梯度gradx2\_
24. Sobel(gray, grady2\_, CV\_32F, 0, 1);    //得到image2的垂直梯度grady2\_
25. }

计算接缝的起点和终点：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **bool** DpSeamFinder::getSeamTips(**int** comp1, **int** comp2, Point &p1, Point &p2)
2. //comp1和comp2分别表示相邻的组件
3. //p1和p2分别表示得到返回comp1中，聚类最远的两个聚类中的像素点，即接缝的起点和终点
4. //返回为true，则说明得到了p1和p2，否则返回false
5. {
6. CV\_Assert(states\_[comp1] & INTERS);    //确保comp1为图像间的重叠部分
8. // find special points
10. vector<Point> specialPoints;    //表示一些特殊的点，即靠近其他边界的点
11. **int** l2 = comp2+1;    //表示comp2的标签
13. **for** (**size\_t** i = 0; i < contours\_[comp1].size(); ++i)    //遍历comp1的边缘像素
14. {
15. **int** x = contours\_[comp1][i].x;    //得到当前边缘像素的坐标
16. **int** y = contours\_[comp1][i].y;
17. //if条件成立，说明当前像素临近并集的边界，并且靠近comp2
18. **if** (closeToContour(y, x, contour1mask\_) &&
19. closeToContour(y, x, contour2mask\_) &&
20. ((x > 0 && labels\_(y, x-1) == l2) ||
21. (y > 0 && labels\_(y-1, x) == l2) ||
22. (x < unionSize\_.width-1 && labels\_(y, x+1) == l2) ||
23. (y < unionSize\_.height-1 && labels\_(y+1, x) == l2)))
24. {
25. specialPoints.push\_back(Point(x, y));    //存储这些特殊的点
26. }
27. }
28. //特殊点数量太少，不能形成聚类，则直接返回
29. **if** (specialPoints.size() < 2)
30. **return** **false**;
32. // find clusters
34. vector<**int**> labels;
35. //把specialPoints中的相距不大于10个单位的点聚成一类，并赋予相应的标签labels
36. cv::partition(specialPoints, labels, ClosePoints(10));
37. //得到最大聚类的像素数量
38. **int** nlabels = \*max\_element(labels.begin(), labels.end()) + 1;
39. **if** (nlabels < 2)    //聚类中的像素数量太少，无法得到中心像素，则直接返回
40. **return** **false**;
42. vector<Point> sum(nlabels);    //表示各个聚类中像素点坐标值之和
43. vector<vector<Point> > points(nlabels);    //用于存储各个聚类的像素
45. **for** (**size\_t** i = 0; i < specialPoints.size(); ++i)    //遍历specialPoints
46. {
47. sum[labels[i]] += specialPoints[i];
48. points[labels[i]].push\_back(specialPoints[i]);
49. }
51. // select two most distant clusters
53. **int** idx[2] = {-1,-1};    //表示距离最大的两个聚类的索引
54. //表示两个聚类间的最大距离，先初始化为最小值
55. **double** maxDist = -numeric\_limits<**double**>::max();
56. //两两聚类遍历，得到距离最大的两个聚类
57. **for** (**int** i = 0; i < nlabels-1; ++i)
58. {
59. **for** (**int** j = i+1; j < nlabels; ++j)
60. {
61. //分别表示聚类1和聚类2的特殊点的数量
62. **double** size1 = **static\_cast**<**double**>(points[i].size()), size2 = **static\_cast**<**double**>(points[j].size());
63. //分别表示聚类1和聚类2的横、纵坐标的均值，代表聚类的中心
64. **double** cx1 = cvRound(sum[i].x / size1), cy1 = cvRound(sum[i].y / size1);
65. **double** cx2 = cvRound(sum[j].x / size2), cy2 = cvRound(sum[j].y / size1);
66. //聚类i和聚类j间的距离
67. **double** dist = (cx1 - cx2) \* (cx1 - cx2) + (cy1 - cy2) \* (cy1 - cy2);
68. **if** (dist > maxDist)
69. {
70. maxDist = dist;    //更新最大距离
71. idx[0] = i;    //更新两个聚类
72. idx[1] = j;
73. }
74. }
75. }
77. // select two points closest to the clusters' centers
79. Point p[2];
80. //遍历距离最大的那两个聚类，得到聚类内离中心最近的点
81. **for** (**int** i = 0; i < 2; ++i)
82. {
83. **double** size = **static\_cast**<**double**>(points[idx[i]].size());    //聚类内的特殊点的数量
84. **double** cx = cvRound(sum[idx[i]].x / size);    //聚类的横坐标的均值
85. **double** cy = cvRound(sum[idx[i]].y / size);    //聚类的纵坐标的均值
87. **size\_t** closest = points[idx[i]].size();
88. **double** minDist = numeric\_limits<**double**>::max();    //先设置一个最大值
90. **for** (**size\_t** j = 0; j < points[idx[i]].size(); ++j)    //遍历聚类内的特殊点
91. {
92. //得到当前特殊点与该聚类中心的距离
93. **double** dist = (points[idx[i]][j].x - cx) \* (points[idx[i]][j].x - cx) +
94. (points[idx[i]][j].y - cy) \* (points[idx[i]][j].y - cy);
95. **if** (dist < minDist)
96. {
97. minDist = dist;    //更新最小距离
98. closest = j;    //更新聚类索引
99. }
100. }
102. p[i] = points[idx[i]][closest];    //得到该点
103. }
105. p1 = p[0];    //得到这两个像素点
106. p2 = p[1];
107. **return** **true**;
108. }

计算得到接缝：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **bool** DpSeamFinder::estimateSeam(
2. **const** Mat &image1, **const** Mat &image2, Point tl1, Point tl2, **int** comp,
3. Point p1, Point p2, vector<Point> &seam, **bool** &isHorizontal)
4. //image1和image2分别表示重叠的两幅图像
5. //tl1和tl2分别表示这两幅图像在全景图像中的左上角的坐标
6. //comp表示这两幅图像的重叠部分，即交集组件
7. //p1和p2分别表示接缝的起点和终点
8. //seam表示得到的接缝
9. //isHorizontal表示接缝的基本走向是否为水平
10. //如果得到了接缝，该函数返回true，否则返回false
11. {
12. CV\_Assert(states\_[comp] & INTERS);    //确保comp为交集组件
14. Mat\_<**float**> costV, costH;    //表示垂直和水平的代价
15. //计算两幅图像间重叠区域的水平和垂直的代价值，该函数在后面给出介绍
16. computeCosts(image1, image2, tl1, tl2, comp, costV, costH);
18. Rect roi(tls\_[comp], brs\_[comp]);    //定义重叠区域的矩形形式
19. Point src = p1 - roi.tl();    //src表示p1点在roi的绝对位置
20. Point dst = p2 - roi.tl();    //dst表示p2点在roi的绝对位置
21. **int** l = comp+1;    //表示comp1的标签
23. // estimate seam direction
25. **bool** swapped = **false**;    //定义一个标签，表示src和dst交换
26. //由src和dst构成的矩形如果是宽大于高，则isHorizontal为true，否则为false
27. isHorizontal = std::abs(dst.x - src.x) > std::abs(dst.y - src.y);
29. **if** (isHorizontal)    //宽大于高
30. {
31. **if** (src.x > dst.x)    //比较横坐标
32. {
33. std::swap(src, dst);    //交换，使src在dst的左边
34. swapped = **true**;
35. }
36. }
37. **else** **if** (src.y > dst.y)    //高大于宽，则比较纵坐标
38. {
39. swapped = **true**;
40. std::swap(src, dst);    //交换，使src在dst的上边
41. }
43. // find optimal control
45. Mat\_<uchar> control = Mat::zeros(roi.size(), CV\_8U);    //表示控制量，即接缝的走向
46. Mat\_<uchar> reachable = Mat::zeros(roi.size(), CV\_8U);    //表示已遍历到该像素
47. Mat\_<**float**> cost = Mat::zeros(roi.size(), CV\_32F);    //表示代价值，即累积最小误差
49. reachable(src) = 1;    //从src开始遍历
50. cost(src) = 0.f;
52. **int** nsteps;    //表示步长
53. pair<**float**, **int**> steps[3];    //表示接缝的走向，并计算式89中min中的三个元素
55. **if** (isHorizontal)    //宽大于高，接缝的走向基本上是从左向右
56. {
57. **for** (**int** x = src.x+1; x <= dst.x; ++x)    //遍历重叠部分
58. {
59. **for** (**int** y = 0; y < roi.height; ++y)
60. {
61. // seam follows along upper side of pixels
63. nsteps = 0;    //每次循环，步长都要清零
65. **if** (labels\_(y + roi.y, x + roi.x) == l)    //当前像素的标签为l
66. {
67. **if** (reachable(y, x-1))
68. steps[nsteps++] = make\_pair(cost(y, x-1) + costH(y, x-1), 1);
69. **if** (y > 0 && reachable(y-1, x-1))
70. steps[nsteps++] = make\_pair(cost(y-1, x-1) + costH(y-1, x-1) + costV(y-1, x), 2);
71. **if** (y < roi.height-1 && reachable(y+1, x-1))
72. steps[nsteps++] = make\_pair(cost(y+1, x-1) + costH(y+1, x-1) + costV(y, x), 3);
73. }
75. **if** (nsteps)    //表示该次循环，已经得到了steps变量
76. {
77. //在steps数组中，得到最小值，即式89
78. pair<**float**, **int**> opt = \*min\_element(steps, steps + nsteps);
79. cost(y, x) = opt.first;    //当前像素的代价值
80. control(y, x) = (uchar)opt.second;    //当前像素的控制量
81. reachable(y, x) = 255;    //表示已遍历到该点
82. }
83. }
84. }
85. }
86. **else**    //高大于宽，接缝的走向基本上是从上向下
87. {
88. **for** (**int** y = src.y+1; y <= dst.y; ++y)    //遍历重叠部分
89. {
90. **for** (**int** x = 0; x < roi.width; ++x)
91. {
92. // seam follows along left side of pixels
94. nsteps = 0;    //每次循环，步长都要清零
96. **if** (labels\_(y + roi.y, x + roi.x) == l)    //当前像素的标签为l
97. {
98. **if** (reachable(y-1, x))
99. steps[nsteps++] = make\_pair(cost(y-1, x) + costV(y-1, x), 1);
100. **if** (x > 0 && reachable(y-1, x-1))
101. steps[nsteps++] = make\_pair(cost(y-1, x-1) + costV(y-1, x-1) + costH(y, x-1), 2);
102. **if** (x < roi.width-1 && reachable(y-1, x+1))
103. steps[nsteps++] = make\_pair(cost(y-1, x+1) + costV(y-1, x+1) + costH(y, x), 3);
104. }
106. **if** (nsteps)    //表示该次循环，已经得到了steps变量
107. {
108. //在steps数组中，得到最小值，即式89
109. pair<**float**, **int**> opt = \*min\_element(steps, steps + nsteps);
110. cost(y, x) = opt.first;    //当前像素的代价值
111. control(y, x) = (uchar)opt.second;    //当前像素的控制量
112. reachable(y, x) = 255;    //表示已遍历到该点
113. }
114. }
115. }
116. }
118. **if** (!reachable(dst))    //没有遍历到dst点（终点），则直接退出
119. **return** **false**;
121. // restore seam
123. Point p = dst;    //接缝的终点
124. seam.clear();    //接缝清零
125. seam.push\_back(p + roi.tl());    //终点放入seam中
127. **if** (isHorizontal)    //宽大于高
128. {
129. **for** (; p.x != src.x; seam.push\_back(p + roi.tl()))
130. {
131. //根据控制量，得到接缝的坐标，并把它存放入seam内
132. **if** (control(p) == 2) p.y--;
133. **else** **if** (control(p) == 3) p.y++;
134. p.x--;
135. }
136. }
137. **else**    //高大于宽
138. {
139. **for** (; p.y != src.y; seam.push\_back(p + roi.tl()))
140. {
141. //根据控制量，得到接缝的坐标，并把它存放入seam内
142. **if** (control(p) == 2) p.x--;
143. **else** **if** (control(p) == 3) p.x++;
144. p.y--;
145. }
146. }
148. **if** (!swapped)    //确保起点为p1，终点为p2
149. reverse(seam.begin(), seam.end());
151. CV\_Assert(seam.front() == p1);    //确保起点为p1，终点为p2
152. CV\_Assert(seam.back() == p2);
153. **return** **true**;
154. }

计算代价的函数，程序并不是直接计算重叠区域相对像素之间的差值平方和，而是根据接缝线的走向，计算两者之间水平方向或垂直方向上相差一个像素之间的差值平方和，即式90，因此代价值分为水平值和垂直值两类的：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::computeCosts(
2. **const** Mat &image1, **const** Mat &image2, Point tl1, Point tl2,
3. **int** comp, Mat\_<**float**> &costV, Mat\_<**float**> &costH)
4. {
5. CV\_Assert(states\_[comp] & INTERS);    //确保comp为交集部分
7. // compute costs
9. **float** (\*diff)(**const** Mat&, **int**, **int**, **const** Mat&, **int**, **int**) = 0;    //定义一个函数指针
10. //根据图像类型的不同，diff函数指针指向不同的函数，函数的本质都是比较两幅图像中相对应元素之间的差值，即红、绿、蓝三通道强度差值平方之和
11. **if** (image1.type() == CV\_32FC3 && image2.type() == CV\_32FC3)
12. diff = diffL2Square3<**float**>;    //三通道
13. **else** **if** (image1.type() == CV\_8UC3 && image2.type() == CV\_8UC3)
14. diff = diffL2Square3<uchar>;    //三通道
15. **else** **if** (image1.type() == CV\_32FC4 && image2.type() == CV\_32FC4)
16. diff = diffL2Square4<**float**>;    //四通道
17. **else** **if** (image1.type() == CV\_8UC4 && image2.type() == CV\_8UC4)
18. diff = diffL2Square4<uchar>;    //四通道
19. **else**
20. CV\_Error(CV\_StsBadArg, "both images must have CV\_32FC3(4) or CV\_8UC3(4) type");
22. **int** l = comp+1;    //表示标签
23. Rect roi(tls\_[comp], brs\_[comp]);    //得到comp的矩形形式
24. //得到image1和image2左上角的相对坐标
25. **int** dx1 = unionTl\_.x - tl1.x, dy1 = unionTl\_.y - tl1.y;
26. **int** dx2 = unionTl\_.x - tl2.x, dy2 = unionTl\_.y - tl2.y;
27. //定义一个常数，用来表示无效的值
28. **const** **float** badRegionCost = normL2(Point3f(255.f, 255.f, 255.f),
29. Point3f(0.f, 0.f, 0.f));
31. costV.create(roi.height, roi.width+1);    //创建costV的大小
33. **for** (**int** y = roi.y; y < roi.br().y; ++y)    //遍历roi区域
34. {
35. **for** (**int** x = roi.x; x < roi.br().x+1; ++x)
36. {
37. **if** (labels\_(y, x) == l && x > 0 && labels\_(y, x-1) == l)    //当前像素的标签为l
38. {
39. //在水平方向，计算式90
40. **float** costColor = (diff(image1, y + dy1, x + dx1 - 1, image2, y + dy2, x + dx2) +
41. diff(image1, y + dy1, x + dx1, image2, y + dy2, x + dx2 - 1)) / 2;
42. **if** (costFunc\_ == COLOR)    //直接法
43. costV(y - roi.y, x - roi.x) = costColor;
44. **else** **if** (costFunc\_ == COLOR\_GRAD)    //梯度法
45. {
46. //得到式91的分母部分
47. **float** costGrad = std::abs(gradx1\_(y + dy1, x + dx1)) + std::abs(gradx1\_(y + dy1, x + dx1 - 1)) +
48. std::abs(gradx2\_(y + dy2, x + dx2)) + std::abs(gradx2\_(y + dy2, x + dx2 - 1)) + 1.f;
49. costV(y - roi.y, x - roi.x) = costColor / costGrad;    //式91
50. }
51. }
52. **else**
53. costV(y - roi.y, x - roi.x) = badRegionCost;    //无效的值
54. }
55. }
57. costH.create(roi.height+1, roi.width);    //创建costV的大小
59. **for** (**int** y = roi.y; y < roi.br().y+1; ++y)    //遍历roi区域
60. {
61. **for** (**int** x = roi.x; x < roi.br().x; ++x)
62. {
63. **if** (labels\_(y, x) == l && y > 0 && labels\_(y-1, x) == l)    //当前像素的标签为l
64. {
65. //在垂直方向，计算式90
66. **float** costColor = (diff(image1, y + dy1 - 1, x + dx1, image2, y + dy2, x + dx2) +
67. diff(image1, y + dy1, x + dx1, image2, y + dy2 - 1, x + dx2)) / 2;
68. **if** (costFunc\_ == COLOR)    //直接法
69. costH(y - roi.y, x - roi.x) = costColor;
70. **else** **if** (costFunc\_ == COLOR\_GRAD)        //梯度法
71. {
72. //得到式91的分母部分
73. **float** costGrad = std::abs(grady1\_(y + dy1, x + dx1)) + std::abs(grady1\_(y + dy1 - 1, x + dx1)) +
74. std::abs(grady2\_(y + dy2, x + dx2)) + std::abs(grady2\_(y + dy2 - 1, x + dx2)) + 1.f;
75. costH(y - roi.y, x - roi.x) = costColor / costGrad;    //式91
76. }
77. }
78. **else**
79. costH(y - roi.y, x - roi.x) = badRegionCost;    //无效的值
80. }
81. }
82. }

利用接缝来进行更新标签：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** DpSeamFinder::updateLabelsUsingSeam(
2. **int** comp1, **int** comp2, **const** vector<Point> &seam, **bool** isHorizontalSeam)
3. {
4. //表示comp1的掩码，先清零
5. Mat\_<**int**> mask = Mat::zeros(brs\_[comp1].y - tls\_[comp1].y,
6. brs\_[comp1].x - tls\_[comp1].x, CV\_32S);
7. //把comp1中的边缘的掩码赋值为255
8. **for** (**size\_t** i = 0; i < contours\_[comp1].size(); ++i)
9. mask(contours\_[comp1][i] - tls\_[comp1]) = 255;
10. //把comp1中的接缝的掩码赋值为255
11. **for** (**size\_t** i = 0; i < seam.size(); ++i)
12. mask(seam[i] - tls\_[comp1]) = 255;
14. // find connected components after seam carving
16. **int** l1 = comp1+1, l2 = comp2+1;    //得到两个组件的标签
18. **int** ncomps = 0;    //表示组件数量
19. //在mask1中，对除了边缘和接缝以为的区域利用漫水填充算法进行分割
20. **for** (**int** y = 0; y < mask.rows; ++y)
21. **for** (**int** x = 0; x < mask.cols; ++x)
22. **if** (!mask(y, x) && labels\_(y + tls\_[comp1].y, x + tls\_[comp1].x) == l1)
23. floodFill(mask, Point(x, y), ++ncomps);
25. **for** (**size\_t** i = 0; i < contours\_[comp1].size(); ++i)    //遍历comp1组件边缘像素
26. {
27. **int** x = contours\_[comp1][i].x - tls\_[comp1].x;    //相对坐标
28. **int** y = contours\_[comp1][i].y - tls\_[comp1].y;
30. **bool** ok = **false**;    //标识变量
31. **static** **const** **int** dx[] = {-1, +1, 0, 0, -1, +1, -1, +1};    //表示8邻域
32. **static** **const** **int** dy[] = {0, 0, -1, +1, -1, -1, +1, +1};
34. **for** (**int** j = 0; j < 8; ++j)    //遍历当前像素的8邻域
35. {
36. **int** c = x + dx[j];
37. **int** r = y + dy[j];
38. //表示当前像素的8邻域属于由上一步漫水填充算法得到的分割区域
39. **if** (c >= 0 && c < mask.cols && r >= 0 && r < mask.rows &&
40. mask(r, c) && mask(r, c) != 255)
41. {
42. ok = **true**;    //赋值
43. mask(y, x) = mask(r, c);    //属于同一个分割区域
44. }
45. }
47. **if** (!ok)
48. mask(y, x) = 0;    //表示当前像素为掩码像素
49. }
51. **if** (isHorizontalSeam)    //宽大于高
52. {
53. **for** (**size\_t** i = 0; i < seam.size(); ++i)    //遍历接缝
54. {
55. **int** x = seam[i].x - tls\_[comp1].x;    //相对坐标
56. **int** y = seam[i].y - tls\_[comp1].y;
57. //当前像素的下一行像素的掩码不为0或255
58. **if** (y < mask.rows-1 && mask(y+1, x) && mask(y+1, x) != 255)
59. mask(y, x) = mask(y+1, x);    //赋值
60. **else**
61. mask(y, x) = 0;
62. }
63. }
64. **else**    //高大于宽
65. {
66. **for** (**size\_t** i = 0; i < seam.size(); ++i)    //遍历接缝
67. {
68. **int** x = seam[i].x - tls\_[comp1].x;    //相对坐标
69. **int** y = seam[i].y - tls\_[comp1].y;
70. //当前像素的右边像素的掩码不为0或255
71. **if** (x < mask.cols-1 && mask(y, x+1) && mask(y, x+1) != 255)
72. mask(y, x) = mask(y, x+1);    //赋值
73. **else**
74. mask(y, x) = 0;
75. }
76. }
78. // find new components connected with the second component and
79. // with other components except the ones we are working with
81. map<**int**, **int**> connect2;    //表示与第2个组件相邻
82. map<**int**, **int**> connectOther;    //表示与其他组件相邻
84. **for** (**int** i = 1; i <= ncomps; ++i)    //遍历分割区域
85. {
86. connect2.insert(make\_pair(i, 0));    //初始化
87. connectOther.insert(make\_pair(i, 0));
88. }
90. **for** (**size\_t** i = 0; i < contours\_[comp1].size(); ++i)    //遍历comp1组件的边缘像素
91. {
92. **int** x = contours\_[comp1][i].x;    //当前像素的相对坐标
93. **int** y = contours\_[comp1][i].y;
94. //当前像素的4邻域范围内只要有一个像素的标签为l2
95. **if** ((x > 0 && labels\_(y, x-1) == l2) ||
96. (y > 0 && labels\_(y-1, x) == l2) ||
97. (x < unionSize\_.width-1 && labels\_(y, x+1) == l2) ||
98. (y < unionSize\_.height-1 && labels\_(y+1, x) == l2))
99. {
100. //记录该像素，并且数量累计
101. connect2[mask(y - tls\_[comp1].y, x - tls\_[comp1].x)]++;
102. }
103. //当前像素的4邻域范围内只要有一个像素的标签为其他值
104. **if** ((x > 0 && labels\_(y, x-1) != l1 && labels\_(y, x-1) != l2) ||
105. (y > 0 && labels\_(y-1, x) != l1 && labels\_(y-1, x) != l2) ||
106. (x < unionSize\_.width-1 && labels\_(y, x+1) != l1 && labels\_(y, x+1) != l2) ||
107. (y < unionSize\_.height-1 && labels\_(y+1, x) != l1 && labels\_(y+1, x) != l2))
108. {
109. //记录该像素，并且数量累计
110. connectOther[mask(y - tls\_[comp1].y, x - tls\_[comp1].x)]++;
111. }
112. }
114. vector<**int**> isAdjComp(ncomps + 1, 0);    //表示邻近组件
115. //遍历connect2内的像素
116. **for** (map<**int**, **int**>::iterator itr = connect2.begin(); itr != connect2.end(); ++itr)
117. {
118. //comp1组件内边缘像素的数量
119. **double** len = **static\_cast**<**double**>(contours\_[comp1].size());
120. //通过比较connect2和connectOther的数量来为isAdjComp赋值
121. isAdjComp[itr->first] = itr->second / len > 0.05 && connectOther.find(itr->first)->second / len < 0.1;
122. }
124. // update labels
125. //更新标签
126. **for** (**int** y = 0; y < mask.rows; ++y)
127. **for** (**int** x = 0; x < mask.cols; ++x)
128. **if** (mask(y, x) && isAdjComp[mask(y, x)])
129. labels\_(y + tls\_[comp1].y, x + tls\_[comp1].x) = l2;
130. }

GraphCutSeamFinder类为图割法实现类，它的构造函数为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. GraphCutSeamFinder::GraphCutSeamFinder(**int** cost\_type, **float** terminal\_cost, **float** bad\_region\_penalty)
2. : impl\_(**new** Impl(cost\_type, terminal\_cost, bad\_region\_penalty)) {}
3. //cost\_type表示图割法的类型，直接法还是梯度法，默认为梯度法
4. //terminal\_cost表示终端节点与普通节点之间的初始边权值，默认值为10000
5. //bad\_region\_penalty表示图中无效边的权值，默认值为1000

它主要实例化了impl\_类为Impl，而GraphCutSeamFinder类中的find函数为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** GraphCutSeamFinder::find(**const** vector<Mat> &src, **const** vector<Point> &corners,
2. vector<Mat> &masks)
3. {
4. impl\_->find(src, corners, masks);
5. }

所以对于GraphCutSeamFinder类，重点应该了解它内部的Impl类。

Impl类的find函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** GraphCutSeamFinder::Impl::find(**const** vector<Mat> &src, **const** vector<Point> &corners,
2. vector<Mat> &masks)
3. {
4. // Compute gradients
5. //计算梯度
6. dx\_.resize(src.size());    //表示水平梯度
7. dy\_.resize(src.size());    //表示垂直梯度
8. Mat dx, dy;    //表示当前图像的水平梯度和垂直梯度
9. **for** (**size\_t** i = 0; i < src.size(); ++i)    //遍历所有待拼接的图像
10. {
11. CV\_Assert(src[i].channels() == 3);    //确保图像是三通道的彩色图像
12. Sobel(src[i], dx, CV\_32F, 1, 0);    //得到当前图像的红绿蓝各个通道的水平梯度dx
13. Sobel(src[i], dy, CV\_32F, 0, 1);    //得到当前图像的红绿蓝各个通道的垂直梯度dy
14. dx\_[i].create(src[i].size(), CV\_32F);    //定义大小
15. dy\_[i].create(src[i].size(), CV\_32F);    //定义大小
16. **for** (**int** y = 0; y < src[i].rows; ++y)    //遍历当前图像的行
17. {
18. **const** Point3f\* dx\_row = dx.ptr<Point3f>(y);    //当前图像水平梯度行指针
19. **const** Point3f\* dy\_row = dy.ptr<Point3f>(y);    //当前图像垂直梯度行指针
20. **float**\* dx\_row\_ = dx\_[i].ptr<**float**>(y);    //所有图像的水平梯度行指针
21. **float**\* dy\_row\_ = dy\_[i].ptr<**float**>(y);    //所有图像的垂直梯度行指针
22. **for** (**int** x = 0; x < src[i].cols; ++x)    //遍历当前行的每个元素
23. {
24. dx\_row\_[x] = normL2(dx\_row[x]);    //三通道水平梯度平方和
25. dy\_row\_[x] = normL2(dy\_row[x]);    //三通道垂直梯度平方和
26. }
27. }
28. }
29. PairwiseSeamFinder::find(src, corners, masks);    //调用find函数
30. }

PairwiseSeamFinder类中的find函数主要是调用了run函数，而run函数在前面已给出介绍，它实际又调用了findInPair函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** GraphCutSeamFinder::Impl::findInPair(**size\_t** first, **size\_t** second, Rect roi)
2. {
3. Mat img1 = images\_[first], img2 = images\_[second];    //表示相交的两幅图像
4. Mat dx1 = dx\_[first], dx2 = dx\_[second];    //表示这两幅图像的水平梯度
5. Mat dy1 = dy\_[first], dy2 = dy\_[second];    //表示这两幅图像的垂直梯度
6. Mat mask1 = masks\_[first], mask2 = masks\_[second];    //表示这两幅图像的掩码
7. Point tl1 = corners\_[first], tl2 = corners\_[second];    //表示这两幅图像的左上角坐标
9. **const** **int** gap = 10;
10. //分别定义这两幅图像相应的子图像，它们的面积要大一些
11. Mat subimg1(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_32FC3);
12. Mat subimg2(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_32FC3);
13. Mat submask1(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_8U);
14. Mat submask2(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_8U);
15. Mat subdx1(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_32F);
16. Mat subdy1(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_32F);
17. Mat subdx2(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_32F);
18. Mat subdy2(roi.height + 2 \* gap, roi.width + 2 \* gap, CV\_32F);
20. // Cut subimages and submasks with some gap
21. **for** (**int** y = -gap; y < roi.height + gap; ++y)    //遍历重叠区域
22. {
23. **for** (**int** x = -gap; x < roi.width + gap; ++x)
24. {
25. **int** y1 = roi.y - tl1.y + y;    //当前像素在图像1中的坐标
26. **int** x1 = roi.x - tl1.x + x;
27. **if** (y1 >= 0 && x1 >= 0 && y1 < img1.rows && x1 < img1.cols)
28. //属于图像1，则赋值相应的值
29. {
30. subimg1.at<Point3f>(y + gap, x + gap) = img1.at<Point3f>(y1, x1);
31. submask1.at<uchar>(y + gap, x + gap) = mask1.at<uchar>(y1, x1);
32. subdx1.at<**float**>(y + gap, x + gap) = dx1.at<**float**>(y1, x1);
33. subdy1.at<**float**>(y + gap, x + gap) = dy1.at<**float**>(y1, x1);
34. }
35. **else**    //不属于图像1，则赋值为0
36. {
37. subimg1.at<Point3f>(y + gap, x + gap) = Point3f(0, 0, 0);
38. submask1.at<uchar>(y + gap, x + gap) = 0;
39. subdx1.at<**float**>(y + gap, x + gap) = 0.f;
40. subdy1.at<**float**>(y + gap, x + gap) = 0.f;
41. }
43. **int** y2 = roi.y - tl2.y + y;    //当前像素在图像2中的坐标
44. **int** x2 = roi.x - tl2.x + x;
45. **if** (y2 >= 0 && x2 >= 0 && y2 < img2.rows && x2 < img2.cols)
46. //属于图像2，则赋值相应的值
47. {
48. subimg2.at<Point3f>(y + gap, x + gap) = img2.at<Point3f>(y2, x2);
49. submask2.at<uchar>(y + gap, x + gap) = mask2.at<uchar>(y2, x2);
50. subdx2.at<**float**>(y + gap, x + gap) = dx2.at<**float**>(y2, x2);
51. subdy2.at<**float**>(y + gap, x + gap) = dy2.at<**float**>(y2, x2);
52. }
53. **else**    //不属于图像1，则赋值为0
54. {
55. subimg2.at<Point3f>(y + gap, x + gap) = Point3f(0, 0, 0);
56. submask2.at<uchar>(y + gap, x + gap) = 0;
57. subdx2.at<**float**>(y + gap, x + gap) = 0.f;
58. subdy2.at<**float**>(y + gap, x + gap) = 0.f;
59. }
60. }
61. }
62. //表示普通节点数量
63. **const** **int** vertex\_count = (roi.height + 2 \* gap) \* (roi.width + 2 \* gap);
64. //表示边数量
65. **const** **int** edge\_count = (roi.height - 1 + 2 \* gap) \* (roi.width + 2 \* gap) +
66. (roi.width - 1 + 2 \* gap) \* (roi.height + 2 \* gap);
67. GCGraph<**float**> graph(vertex\_count, edge\_count);    //定义图割变量
69. **switch** (cost\_type\_)    //根据不同的方法，调用不同的函数
70. {
71. **case** GraphCutSeamFinder::COST\_COLOR:    //直接法
72. setGraphWeightsColor(subimg1, subimg2, submask1, submask2, graph);
73. **break**;
74. **case** GraphCutSeamFinder::COST\_COLOR\_GRAD:    //梯度法
75. setGraphWeightsColorGrad(subimg1, subimg2, subdx1, subdx2, subdy1, subdy2,
76. submask1, submask2, graph);
77. **break**;
78. **default**:    //目前只实现了上述那两种算法
79. CV\_Error(CV\_StsBadArg, "unsupported pixel similarity measure");
80. }
82. graph.maxFlow();    //得到最大流
84. **for** (**int** y = 0; y < roi.height; ++y)    //遍历重叠区域，更新掩码
85. {
86. **for** (**int** x = 0; x < roi.width; ++x)
87. {
88. //当前像素属于终端节点s
89. **if** (graph.inSourceSegment((y + gap) \* (roi.width + 2 \* gap) + x + gap))
90. {
91. **if** (mask1.at<uchar>(roi.y - tl1.y + y, roi.x - tl1.x + x))
92. mask2.at<uchar>(roi.y - tl2.y + y, roi.x - tl2.x + x) = 0;    //掩码图像2
93. }
94. **else**    //当前像素属于终端节点t
95. {
96. **if** (mask2.at<uchar>(roi.y - tl2.y + y, roi.x - tl2.x + x))
97. mask1.at<uchar>(roi.y - tl1.y + y, roi.x - tl1.x + x) = 0;    //掩码图像1
98. }
99. }
100. }
101. }

图割中的直接法：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** GraphCutSeamFinder::Impl::setGraphWeightsColor(**const** Mat &img1, **const** Mat &img2,
2. **const** Mat &mask1, **const** Mat &mask2, GCGraph<**float**> &graph)
3. {
4. **const** Size img\_size = img1.size();    //得到重叠区域的尺寸
6. // Set terminal weights
7. **for** (**int** y = 0; y < img\_size.height; ++y)    //遍历重叠区域
8. {
9. **for** (**int** x = 0; x < img\_size.width; ++x)
10. {
11. **int** v = graph.addVtx();    //为图添加普通节点
12. //为普通节点与两个终端节点添加边，即定义其初始权值
13. graph.addTermWeights(v, mask1.at<uchar>(y, x) ? terminal\_cost\_ : 0.f,
14. mask2.at<uchar>(y, x) ? terminal\_cost\_ : 0.f);
15. }
16. }
18. // Set regular edge weights
19. **const** **float** weight\_eps = 1.f;    //设置一个规则边权值
20. **for** (**int** y = 0; y < img\_size.height; ++y)    //遍历重叠区域
21. {
22. **for** (**int** x = 0; x < img\_size.width; ++x)
23. {
24. **int** v = y \* img\_size.width + x;    //表示当前普通节点，即当前像素
25. **if** (x < img\_size.width - 1)
26. {
27. //计算当前像素与其右侧像素的边的权值，式90的分子部分
28. **float** weight = normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
29. normL2(img1.at<Point3f>(y, x + 1), img2.at<Point3f>(y, x + 1)) +
30. weight\_eps;
31. //如果当前像素和其右侧像素是被mask1和mask2掩码掉的，则该边无效
32. **if** (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y, x + 1) ||
33. !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y, x + 1))
34. weight += bad\_region\_penalty\_;    //赋于一个较大的值
35. //为当前像素与其右侧像素之间的边赋予初始权值，权值是双向的
36. graph.addEdges(v, v + 1, weight, weight);
37. }
38. **if** (y < img\_size.height - 1)
39. {
40. //计算当前像素与其下侧像素的边的权值，式90的分子部分
41. **float** weight = normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
42. normL2(img1.at<Point3f>(y + 1, x), img2.at<Point3f>(y + 1, x)) +
43. weight\_eps;
44. //如果当前像素和其下侧像素是被mask1和mask2掩码掉的，则该边无效
45. **if** (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y + 1, x) ||
46. !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y + 1, x))
47. weight += bad\_region\_penalty\_;    //赋于一个较大的值
48. //为当前像素与其下侧像素之间的边赋予初始权值，权值是双向的
49. graph.addEdges(v, v + img\_size.width, weight, weight);
50. }
51. }
52. }
53. }

图割中的梯度法：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. **void** GraphCutSeamFinder::Impl::setGraphWeightsColorGrad(
2. **const** Mat &img1, **const** Mat &img2, **const** Mat &dx1, **const** Mat &dx2,
3. **const** Mat &dy1, **const** Mat &dy2, **const** Mat &mask1, **const** Mat &mask2,
4. GCGraph<**float**> &graph)
5. {
6. **const** Size img\_size = img1.size();    //得到重叠区域的尺寸
8. // Set terminal weights
9. **for** (**int** y = 0; y < img\_size.height; ++y)    //遍历重叠区域
10. {
11. **for** (**int** x = 0; x < img\_size.width; ++x)
12. {
13. **int** v = graph.addVtx();    //为图添加普通节点
14. //为普通节点与两个终端节点添加边，即定义其初始权值
15. graph.addTermWeights(v, mask1.at<uchar>(y, x) ? terminal\_cost\_ : 0.f,
16. mask2.at<uchar>(y, x) ? terminal\_cost\_ : 0.f);
17. }
18. }
20. // Set regular edge weights
21. **const** **float** weight\_eps = 1.f;    //设置一个规则边权值
22. **for** (**int** y = 0; y < img\_size.height; ++y)    //遍历重叠区域
23. {
24. **for** (**int** x = 0; x < img\_size.width; ++x)
25. {
26. **int** v = y \* img\_size.width + x;    //表示当前普通节点，即当前像素
27. **if** (x < img\_size.width - 1)
28. {
29. //计算当前像素与其右侧像素的梯度，即式91的分母部分
30. **float** grad = dx1.at<**float**>(y, x) + dx1.at<**float**>(y, x + 1) +
31. dx2.at<**float**>(y, x) + dx2.at<**float**>(y, x + 1) + weight\_eps;
32. //计算当前像素与其右侧像素的边权值，即式91
33. **float** weight = (normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
34. normL2(img1.at<Point3f>(y, x + 1), img2.at<Point3f>(y, x + 1))) / grad +
35. weight\_eps;
36. //如果当前像素和其右侧像素是被mask1和mask2掩码掉的，则该边无效
37. **if** (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y, x + 1) ||
38. !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y, x + 1))
39. weight += bad\_region\_penalty\_;
40. //为当前像素与其右侧像素之间赋予权值，权值是双向的
41. graph.addEdges(v, v + 1, weight, weight);
42. }
43. **if** (y < img\_size.height - 1)
44. {
45. //计算当前像素与其下侧像素的梯度，即式91的分母部分
46. **float** grad = dy1.at<**float**>(y, x) + dy1.at<**float**>(y + 1, x) +
47. dy2.at<**float**>(y, x) + dy2.at<**float**>(y + 1, x) + weight\_eps;
48. //计算当前像素与其下侧像素的边权值，即式91
49. **float** weight = (normL2(img1.at<Point3f>(y, x), img2.at<Point3f>(y, x)) +
50. normL2(img1.at<Point3f>(y + 1, x), img2.at<Point3f>(y + 1, x))) / grad +
51. weight\_eps;
52. //如果当前像素和其下侧像素是被mask1和mask2掩码掉的，则该边无效
53. **if** (!mask1.at<uchar>(y, x) || !mask1.at<uchar>(y + 1, x) ||
54. !mask2.at<uchar>(y, x) || !mask2.at<uchar>(y + 1, x))
55. weight += bad\_region\_penalty\_;    //赋于一个较大的值
56. //为当前像素与其下侧像素之间赋予权值，权值是双向的
57. graph.addEdges(v, v + img\_size.width, weight, weight);
58. }
59. }
60. }
61. }

关于图割算法的源码解析这里就不再介绍，详细内容可以参看下面这些网文：

http://blog.csdn.net/u011574296/article/details/52983211

http://www.voidcn.com/article/p-tsizwupq-h.html

http://www.myexception.cn/go/1895078.html

6.3 应用

下面我们给出寻找接缝线的应用：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78944867) [copy](http://blog.csdn.net/zhaocj/article/details/78944867)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. #include <iomanip>
23. **using** **namespace** cv;
24. **using** **namespace** std;
25. **using** **namespace** detail;
27. **int** main(**int** argc, **char**\*\* argv)
28. {
29. vector<Mat> imgs;    //输入图像
30. Mat img = imread("1.jpg");
31. imgs.push\_back(img);
32. img = imread("2.jpg");
33. imgs.push\_back(img);
35. Ptr<FeaturesFinder> finder;    //特征检测
36. finder = **new** SurfFeaturesFinder();
37. vector<ImageFeatures> features(2);
38. (\*finder)(imgs[0], features[0]);
39. (\*finder)(imgs[1], features[1]);
41. vector<MatchesInfo> pairwise\_matches;    //特征匹配
42. BestOf2NearestMatcher matcher(**false**, 0.3f, 6, 6);
43. matcher(features, pairwise\_matches);
45. HomographyBasedEstimator estimator;    //相机参数评估
46. vector<CameraParams> cameras;
47. estimator(features, pairwise\_matches, cameras);
48. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
49. {
50. Mat R;
51. cameras[i].R.convertTo(R, CV\_32F);
52. cameras[i].R = R;
53. }
55. Ptr<detail::BundleAdjusterBase> adjuster;    //光束平差法，精确化相机参数
56. adjuster = **new** detail::BundleAdjusterReproj();
57. adjuster->setConfThresh(1);
58. (\*adjuster)(features, pairwise\_matches, cameras);
60. vector<Mat> rmats;
61. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
62. rmats.push\_back(cameras[i].R.clone());
63. waveCorrect(rmats, WAVE\_CORRECT\_HORIZ);    //波形校正
64. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
65. cameras[i].R = rmats[i];
67. //图像映射变换
68. vector<Point> corners(2);
69. vector<Mat> masks\_warped(2);
70. vector<Mat> images\_warped(2);
71. vector<Mat> masks(2);
72. **for** (**int** i = 0; i < 2; ++i)
73. {
74. masks[i].create(imgs[i].size(), CV\_8U);
75. masks[i].setTo(Scalar::all(255));
76. }
77. Ptr<WarperCreator> warper\_creator;
78. warper\_creator = **new** cv::PlaneWarper();
79. Ptr<RotationWarper> warper = warper\_creator->create(**static\_cast**<**float**>(cameras[0].focal));
80. **for** (**int** i = 0; i < 2; ++i)
81. {
82. Mat\_<**float**> K;
83. cameras[i].K().convertTo(K, CV\_32F);
84. corners[i] = warper->warp(imgs[i], K, cameras[i].R, INTER\_LINEAR, BORDER\_REFLECT, images\_warped[i]);
85. warper->warp(masks[i], K, cameras[i].R, INTER\_NEAREST, BORDER\_CONSTANT, masks\_warped[i]);
86. }
88. //曝光补偿
89. Ptr<ExposureCompensator> compensator =
90. ExposureCompensator::createDefault(ExposureCompensator::GAIN);
91. compensator->feed(corners, images\_warped, masks\_warped);
92. **for**(**int** i=0;i<2;++i)
93. {
94. compensator->apply(i, corners[i], images\_warped[i], masks\_warped[i]);
95. }
97. Ptr<SeamFinder> seam\_finder;    //定义接缝线寻找器
98. //seam\_finder = new NoSeamFinder();    //无需寻找接缝线
99. //seam\_finder = new VoronoiSeamFinder();    //逐点法
100. //seam\_finder = new DpSeamFinder(DpSeamFinder::COLOR);    //动态规范法
101. //seam\_finder = new DpSeamFinder(DpSeamFinder::COLOR\_GRAD);
102. //图割法
103. //seam\_finder = new GraphCutSeamFinder(GraphCutSeamFinder::COST\_COLOR);
104. seam\_finder = **new** GraphCutSeamFinder(GraphCutSeamFinder::COST\_COLOR\_GRAD);
106. vector<Mat> images\_warped\_f(2);
107. **for** (**int** i = 0; i < 2; ++i)    //图像数据类型转换
108. images\_warped[i].convertTo(images\_warped\_f[i], CV\_32F);
109. //得到接缝线的掩码图像masks\_warped
110. seam\_finder->find(images\_warped\_f, corners, masks\_warped);
112. //通过canny边缘检测，得到掩码边界，其中有一条边界就是接缝线
113. **for**(**int** k=0;k<2;k++)
114. Canny(masks\_warped[k], masks\_warped[k], 3, 9,3 );
116. //为了使接缝线看得更清楚，这里使用了膨胀运算来加粗边界线
117. vector<Mat> dilate\_img(2);
118. Mat element = getStructuringElement(MORPH\_RECT, Size(10, 10));    //定义结构元素
120. **for**(**int** k =0; k<2;k++)    //遍历两幅图像
121. {
122. dilate(masks\_warped[k], dilate\_img[k],element);    //膨胀运算
123. //在映射变换图上画出接缝线，在这里只是为了呈现出的一种效果，所以并没有区分接缝线和其他掩码边界
124. **for**(**int** y = 0; y < images\_warped[k].rows; y++)
125. {
126. **for**(**int** x = 0; x < images\_warped[k].cols; x++)
127. {
128. **if**(dilate\_img[k].at<uchar>(y, x) == 255)    //掩码边界
129. {
130. images\_warped[k].at<Vec3b>(y, x)[0]=255;
131. images\_warped[k].at<Vec3b>(y, x)[1]=0;
132. images\_warped[k].at<Vec3b>(y, x)[2]=255;
133. }
134. }
135. }
136. }
138. imwrite("seam1.jpg", images\_warped[0]);    //存储图像
139. imwrite("seam2.jpg", images\_warped[1]);
141. **return** 0;
142. }

最终得到的结果为：

图13 接缝线

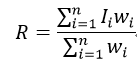
# [Opencv2.4.9源码分析——Stitching（七）](http://blog.csdn.net/zhaocj/article/details/78960325)

7、融合

7.1 原理

在上一步中，虽然我们已经得到了接缝线，但如果只是简单的对接缝线的两侧选取不同的图像，那么对于重叠区域，在接缝线处的过度会出现不连贯的现象，在视觉上会显得有些突兀。因此我们还需要在接缝线两侧，对不同图像进行融合处理来克服上述不足之处。应用于图像拼接的融合算法有两种常用的方法：羽化和多频段融合。

羽化的原理是对边界进行平滑虚化，通过渐变的方法达到自然衔接的效果。在应用于图像拼接时，羽化是只对接缝线两侧的区域进行处理，它的公式为：

（92）

式中，*R*表示由*n*幅图像重叠后经过羽化处理后而得到的新图像，*Ii*表示第*i*幅图像在接缝线两侧区域内的部分，*wi*表示*Ii*的权值，它是当前像素到达第*i*幅图像最近边界的距离。从式92可以看出，羽化算法本质上就是加权平均的过程。另外羽化处理还可以通过设置锐度参数，来调整羽化平滑处理的虚化程度和羽化面积。

羽化算法虽然简单，但当重叠部分有细微的不重合的时候，图像的高频部分会出现较为明显的模糊情况。

为了能够保留图像的高频成分（即图像的细节部分），则需要应用多频段融合方法，它通过建立拉普拉斯（带通滤波器）金字塔，使各个频段上的信息都保留并融合在一起。

我们下面给出多频段融合方法的具体执行步骤：首先分别建立各个图像的拉普拉斯金字塔，然后针对重叠区域，把它们的金字塔的相同层应用式92进行合并，最后对该合并后的金字塔进行逆拉普拉斯变换，从而得到最终的融合图像。

拉普拉斯金字塔是通过高斯金字塔得到。高斯金字塔的上一层图像是对下一层图像进行高斯模糊（卷积高斯内核）再降采样（隔点采样）得到的。而拉普拉斯金字塔的各层图像是由高斯金字塔的相同层减去它的上一层的扩展（即先升采样，再卷积高斯内核）得到的，即

http://img.blog.csdn.net/20180103135138540（93）

式中，*L*和*G*分别表示拉普拉斯和高斯金字塔，拉普拉斯金字塔的顶层图像就是高斯金字塔的顶层图像，下标*n*表示的是金字塔的层数，底层为0，并且*G*0为图像原图，expand表示扩展运算。拉普拉斯金字塔是由底层向顶层逐层构建得到的。图14示意了拉普拉斯金字塔的建立方法。

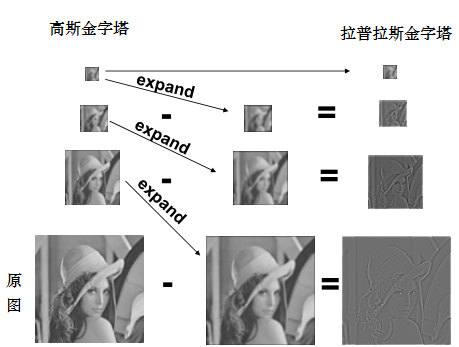


图14 拉普拉斯金字塔

当得到了不同图像的拉普拉斯金字塔后，我们仍然可以应用式92对不同的区域的不同层进行合并，同样也得到了一个金字塔，我们称为合并金字塔。其中式92的权值，在这里就是掩码，而各层的掩码也是通过建立金字塔得到，也就是需要为掩码建立一个高斯金字塔，金字塔的底层就是该图的掩码。

逆拉普拉斯变换的计算公式为：

http://img.blog.csdn.net/20180103135148480（94）

式中，*R*为由式92得到的合并金字塔，*S*为融合金字塔，其中，*S*的顶层为*R*的顶层，*S*是从顶层向底层计算得到的，最终得到的融合金字塔的底层图像就是我们想要的融合图像。图15和图16分别表示了合并金字塔和融合金字塔建立过程。

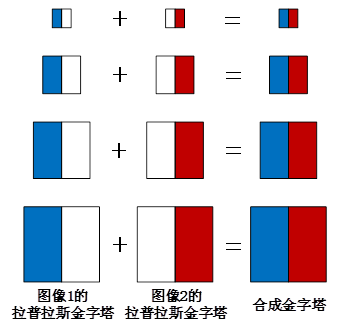


图15 合成金字塔创建示意图

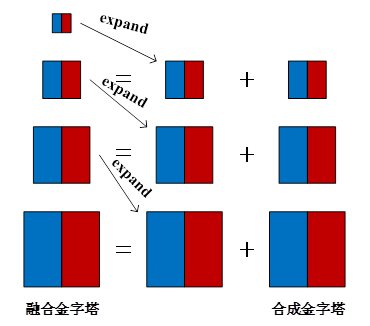


图16融合金字塔创建示意图

7.2 源码

图像融合的基类Blender为：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **class** CV\_EXPORTS Blender
2. {
3. **public**:
4. **virtual** ~Blender() {}
6. **enum** { NO, FEATHER, MULTI\_BAND };    //表示融合算法的类别
7. //该函数的主要作用是根据不同的算法类别type，实例化并得到不同的子类
8. **static** Ptr<Blender> createDefault(**int** type, **bool** try\_gpu = **false**);
9. //prepare函数表示事先得到全景图像的Mat变量，就是为了像素赋值，先准备好全景图像的区域、尺寸
10. **void** prepare(**const** std::vector<Point> &corners, **const** std::vector<Size> &sizes);
11. **virtual** **void** prepare(Rect dst\_roi);
12. **virtual** **void** feed(**const** Mat &img, **const** Mat &mask, Point tl);    //预处理图像
13. **virtual** **void** blend(Mat &dst, Mat &dst\_mask);    //执行融合算法
15. **protected**:
16. //表示最终得到的全景图像和它的掩码
17. Mat dst\_, dst\_mask\_;
18. Rect dst\_roi\_;    //表示最终得到的全景图像的矩形变量
19. };

第一个prepare函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** Blender::prepare(**const** vector<Point> &corners, **const** vector<Size> &sizes)
2. //corners表示待拼接图像在全景图像中的左上角坐标
3. //sizes表示映射变换后待拼接图像的尺寸
4. {
5. //利用resultRoi函数得到最终的全景图像的尺寸
6. //调用另一个prepare函数，该函数的主要作用是初始化为dst\_，dst\_mask\_和dst\_roi\_
7. prepare(resultRoi(corners, sizes));
8. }

为dst\_和dst\_mask\_在img图像的区域内赋值，父类Blender类本质上没有进行任何融合，所以该类的feed函数就是简单赋值：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** Blender::feed(**const** Mat &img, **const** Mat &mask, Point tl)
2. //img表示待拼接的图像
3. //mask表示该图像的掩码
4. //tl表示该图像在全景图像的左上角坐标
5. {
6. CV\_Assert(img.type() == CV\_16SC3);    //确保img类型正确
7. CV\_Assert(mask.type() == CV\_8U);    //确保mask类型正确
8. **int** dx = tl.x - dst\_roi\_.x;    //表示该图像在最终的全景图像的左上角的横坐标
9. **int** dy = tl.y - dst\_roi\_.y;    //表示该图像在最终的全景图像的左上角的纵坐标
11. **for** (**int** y = 0; y < img.rows; ++y)    //遍历图像的行
12. {
13. //得到各个变量的行首地址指针
14. **const** Point3\_<**short**> \*src\_row = img.ptr<Point3\_<**short**> >(y);
15. Point3\_<**short**> \*dst\_row = dst\_.ptr<Point3\_<**short**> >(dy + y);
16. **const** uchar \*mask\_row = mask.ptr<uchar>(y);
17. uchar \*dst\_mask\_row = dst\_mask\_.ptr<uchar>(dy + y);
19. **for** (**int** x = 0; x < img.cols; ++x)    //遍历当前行的每个像素
20. {
21. **if** (mask\_row[x])    //当前像素没有被掩码掉
22. dst\_row[dx + x] = src\_row[x];    //赋值
23. dst\_mask\_row[dx + x] |= mask\_row[x];    //赋值
24. }
25. }
26. }

基类的blend并没有执行任何融合算法，只是简单的赋值，当融合类别为NO时，其实也就是调用的该基类：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** Blender::blend(Mat &dst, Mat &dst\_mask)
2. //dst和dst\_mask表示最终得到的全景图像和掩码
3. {
4. dst\_.setTo(Scalar::all(0), dst\_mask\_ == 0);    //为掩码部分赋0值
5. dst = dst\_;    //赋值
6. dst\_mask = dst\_mask\_;    //赋值
7. dst\_.release();    //释放内存
8. dst\_mask\_.release();    //释放内容
9. }

下面介绍羽化融合算法FeatherBlender类的相关函数，它的父类为Blender类：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** FeatherBlender::prepare(Rect dst\_roi)
2. {
3. Blender::prepare(dst\_roi);    //调用Blender::prepare函数
4. //全局变量dst\_weight\_map\_表示最终得到的全景图像的权值映射图像，在这里初始化该变量
5. dst\_weight\_map\_.create(dst\_roi.size(), CV\_32F);
6. dst\_weight\_map\_.setTo(0);
7. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** FeatherBlender::feed(**const** Mat &img, **const** Mat &mask, Point tl)
2. {
3. CV\_Assert(img.type() == CV\_16SC3);    //确保img类型正确
4. CV\_Assert(mask.type() == CV\_8U);    //确保mask类型正确
5. //该函数在后面给出介绍，全局变量sharpness\_表示锐化系数，默认值为0.02，全局变量weight\_map\_表示图像img的权值映射图，也就是经过阈值处理后的距离图像
6. createWeightMap(mask, sharpness\_, weight\_map\_);
7. **int** dx = tl.x - dst\_roi\_.x;    //表示该图像在最终的全景图像的左上角的横坐标
8. **int** dy = tl.y - dst\_roi\_.y;    //表示该图像在最终的全景图像的左上角的纵坐标
10. **for** (**int** y = 0; y < img.rows; ++y)    //遍历图像的行
11. {
12. //得到各个变量的行首地址指针
13. **const** Point3\_<**short**>\* src\_row = img.ptr<Point3\_<**short**> >(y);
14. Point3\_<**short**>\* dst\_row = dst\_.ptr<Point3\_<**short**> >(dy + y);
15. **const** **float**\* weight\_row = weight\_map\_.ptr<**float**>(y);
16. **float**\* dst\_weight\_row = dst\_weight\_map\_.ptr<**float**>(dy + y);
18. **for** (**int** x = 0; x < img.cols; ++x)    //遍历当前行的每个像素，为相关变量赋值
19. {
20. //dst\_为src\_的加权结果，即式92的分子部分，如果不是重叠区域，则只有一幅图像，所以加号是不起作用的
21. dst\_row[dx + x].x += **static\_cast**<**short**>(src\_row[x].x \* weight\_row[x]);
22. dst\_row[dx + x].y += **static\_cast**<**short**>(src\_row[x].y \* weight\_row[x]);
23. dst\_row[dx + x].z += **static\_cast**<**short**>(src\_row[x].z \* weight\_row[x]);
24. dst\_weight\_row[dx + x] += weight\_row[x];    //式92的分母部分
25. }
26. }
27. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** FeatherBlender::blend(Mat &dst, Mat &dst\_mask)
2. {
3. //该函数在后面给出介绍
4. normalizeUsingWeightMap(dst\_weight\_map\_, dst\_);
5. //WEIGHT\_EPS = 1e-5f，表示很小的一个数
6. //dst\_mask\_为二值图像，权值很小为0，否则为1
7. dst\_mask\_ = dst\_weight\_map\_ > WEIGHT\_EPS;
8. Blender::blend(dst, dst\_mask);    //调用Blender:: blend函数，为dst和dst\_mask赋值
9. }

生成权值映射图：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** createWeightMap(**const** Mat &mask, **float** sharpness, Mat &weight)
2. //mask表示掩码
3. //sharpness表示锐度，系统默认值为0.02
4. //weight表示返回的权值映射图，也就是经过阈值处理后的距离图像
5. {
6. CV\_Assert(mask.type() == CV\_8U);    //再次确保mask类型正确
7. //在二值掩码图像mask内，得到每个像素到达最近的零值像素的距离，得到距离图像weight
8. distanceTransform(mask, weight, CV\_DIST\_L1, 3);
9. //调用threshold函数，首先把距离图像weight乘以锐度sharpness，此时如果距离图像像素值大于阈值1，则等于1，否则保留原值，即保留数值较小的那些像素，结果再次赋值给weight
10. threshold(weight \* sharpness, weight, 1.f, 1.f, THRESH\_TRUNC);
11. }

normalizeUsingWeightMap函数实现了式92：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** normalizeUsingWeightMap(**const** Mat& weight, Mat& src)
2. //weight表示权值映射图像
3. //src作为输入表示加权处理后的图像，作为输出表示归一化后的图像
4. {
5. #ifdef HAVE\_TEGRA\_OPTIMIZATION
6. **if**(tegra::normalizeUsingWeightMap(weight, src))
7. **return**;
8. #endif
9. CV\_Assert(src.type() == CV\_16SC3);    //确保src类型符合要求
11. **if**(weight.type() == CV\_32FC1)    //weight类型为CV\_32FC1
12. {
13. //遍历全景图像
14. **for** (**int** y = 0; y < src.rows; ++y)
15. {
16. //获取当前行的首地址指针
17. Point3\_<**short**> \*row = src.ptr<Point3\_<**short**> >(y);
18. **const** **float** \*weight\_row = weight.ptr<**float**>(y);
20. **for** (**int** x = 0; x < src.cols; ++x)
21. {
22. //对重叠区域，进行归一化处理，式92
23. //而对非重叠区域，还是原始像素值
24. row[x].x = **static\_cast**<**short**>(row[x].x / (weight\_row[x] + WEIGHT\_EPS));
25. row[x].y = **static\_cast**<**short**>(row[x].y / (weight\_row[x] + WEIGHT\_EPS));
26. row[x].z = **static\_cast**<**short**>(row[x].z / (weight\_row[x] + WEIGHT\_EPS));
27. }
28. }
29. }
30. **else**    //weight类型为CV\_16SC1
31. {
32. CV\_Assert(weight.type() == CV\_16SC1);    //确保weight类型为CV\_16SC1
33. //遍历全景图像
34. **for** (**int** y = 0; y < src.rows; ++y)
35. {
36. //获取当前行的首地址指针
37. **const** **short** \*weight\_row = weight.ptr<**short**>(y);
38. Point3\_<**short**> \*row = src.ptr<Point3\_<**short**> >(y);
40. **for** (**int** x = 0; x < src.cols; ++x)
41. {
42. **int** w = weight\_row[x] + 1;    //权值
43. //对重叠区域，进行归一化处理，式92
44. //而对非重叠区域，还是原始像素值
45. row[x].x = **static\_cast**<**short**>((row[x].x << 8) / w);
46. row[x].y = **static\_cast**<**short**>((row[x].y << 8) / w);
47. row[x].z = **static\_cast**<**short**>((row[x].z << 8) / w);
48. }
49. }
50. }
51. }

下面给出多频段融合算法MultiBandBlender类的相关函数，它的父类也是Blender类。我们首先给出它的构造函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. MultiBandBlender::MultiBandBlender(**int** try\_gpu, **int** num\_bands, **int** weight\_type)
2. //try\_gpu表示是否应用图像处理器
3. //num\_bands表示频段的数量，缺省值为5，即金字塔一共有5层
4. //weight\_type表示权值的数据类型
5. {
6. //把num\_bands赋值给全局变量actual\_num\_bands\_
7. setNumBands(num\_bands);
8. #if defined(HAVE\_OPENCV\_GPU) && !defined(DYNAMIC\_CUDA\_SUPPORT)
9. can\_use\_gpu\_ = try\_gpu && gpu::getCudaEnabledDeviceCount();
10. #else
11. (**void**)try\_gpu;
12. can\_use\_gpu\_ = **false**;
13. #endif
14. //确保权值的数据类型为CV\_32F或CV\_16S
15. CV\_Assert(weight\_type == CV\_32F || weight\_type == CV\_16S);
16. weight\_type\_ = weight\_type;    //赋值
17. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** MultiBandBlender::prepare(Rect dst\_roi)
2. {
3. dst\_roi\_final\_ = dst\_roi;    //赋值，表示最终的全景图像
5. // Crop unnecessary bands
6. //图像两边长的最大值
7. **double** max\_len = **static\_cast**<**double**>(max(dst\_roi.width, dst\_roi.height));
8. //设置真正的频段数量
9. num\_bands\_ = min(actual\_num\_bands\_, **static\_cast**<**int**>(ceil(log(max\_len) / log(2.0))));
11. // Add border to the final image, to ensure sizes are divided by (1 << num\_bands\_)
12. //增加全景图像的边长，以确保能够完成num\_bands\_次的降采样
13. dst\_roi.width += ((1 << num\_bands\_) - dst\_roi.width % (1 << num\_bands\_)) % (1 << num\_bands\_);
14. dst\_roi.height += ((1 << num\_bands\_) - dst\_roi.height % (1 << num\_bands\_)) % (1 << num\_bands\_);
16. Blender::prepare(dst\_roi);    //调用Blender::prepare函数
17. //全局变量dst\_pyr\_laplace\_表示拉普拉斯金字塔
18. dst\_pyr\_laplace\_.resize(num\_bands\_ + 1);    //金字塔的层数赋值
19. dst\_pyr\_laplace\_[0] = dst\_;    //第0层（底层）为原图
20. //全局变量dst\_band\_weights\_表示各个频段的权值，与拉普拉斯金字塔相对应
21. dst\_band\_weights\_.resize(num\_bands\_ + 1);
22. dst\_band\_weights\_[0].create(dst\_roi.size(), weight\_type\_);    //为底层大小类型赋值
23. dst\_band\_weights\_[0].setTo(0);    //初始化为0
24. //为除了底层以外的其他金字塔层赋值
25. **for** (**int** i = 1; i <= num\_bands\_; ++i)    //遍历金字塔的层
26. {
27. //定义各个层的尺寸和数据类型
28. dst\_pyr\_laplace\_[i].create((dst\_pyr\_laplace\_[i - 1].rows + 1) / 2,
29. (dst\_pyr\_laplace\_[i - 1].cols + 1) / 2, CV\_16SC3);
30. dst\_band\_weights\_[i].create((dst\_band\_weights\_[i - 1].rows + 1) / 2,
31. (dst\_band\_weights\_[i - 1].cols + 1) / 2, weight\_type\_);
32. dst\_pyr\_laplace\_[i].setTo(Scalar::all(0));    //初始化为0
33. dst\_band\_weights\_[i].setTo(0);    //初始化为0
34. }
35. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** MultiBandBlender::feed(**const** Mat &img, **const** Mat &mask, Point tl)
2. {
3. //确保图像和掩码的数据类型正确
4. CV\_Assert(img.type() == CV\_16SC3 || img.type() == CV\_8UC3);
5. CV\_Assert(mask.type() == CV\_8U);
7. // Keep source image in memory with small border
8. //得到新的左上角和右下角坐标，新的矩形要比img大
9. **int** gap = 3 \* (1 << num\_bands\_);
10. Point tl\_new(max(dst\_roi\_.x, tl.x - gap),
11. max(dst\_roi\_.y, tl.y - gap));
12. Point br\_new(min(dst\_roi\_.br().x, tl.x + img.cols + gap),
13. min(dst\_roi\_.br().y, tl.y + img.rows + gap));
15. // Ensure coordinates of top-left, bottom-right corners are divided by (1 << num\_bands\_).
16. // After that scale between layers is exactly 2.
17. //
18. // We do it to avoid interpolation problems when keeping sub-images only. There is no such problem when
19. // image is bordered to have size equal to the final image size, but this is too memory hungry approach.
20. //为了便于降采样，需要对左上角、右下角进行调整
21. tl\_new.x = dst\_roi\_.x + (((tl\_new.x - dst\_roi\_.x) >> num\_bands\_) << num\_bands\_);
22. tl\_new.y = dst\_roi\_.y + (((tl\_new.y - dst\_roi\_.y) >> num\_bands\_) << num\_bands\_);
23. **int** width = br\_new.x - tl\_new.x;
24. **int** height = br\_new.y - tl\_new.y;
25. width += ((1 << num\_bands\_) - width % (1 << num\_bands\_)) % (1 << num\_bands\_);
26. height += ((1 << num\_bands\_) - height % (1 << num\_bands\_)) % (1 << num\_bands\_);
27. br\_new.x = tl\_new.x + width;
28. br\_new.y = tl\_new.y + height;
29. **int** dy = max(br\_new.y - dst\_roi\_.br().y, 0);
30. **int** dx = max(br\_new.x - dst\_roi\_.br().x, 0);
31. tl\_new.x -= dx; br\_new.x -= dx;
32. tl\_new.y -= dy; br\_new.y -= dy;
33. //下面4个变量表示扩充的四边的长度
34. **int** top = tl.y - tl\_new.y;
35. **int** left = tl.x - tl\_new.x;
36. **int** bottom = br\_new.y - tl.y - img.rows;
37. **int** right = br\_new.x - tl.x - img.cols;
39. // Create the source image Laplacian pyramid
40. Mat img\_with\_border;    //表示边界扩充后的图像
41. //对img通过边界反射的方式填充扩充了边界后的像素，得到img\_with\_border
42. copyMakeBorder(img, img\_with\_border, top, bottom, left, right,
43. BORDER\_REFLECT);
44. vector<Mat> src\_pyr\_laplace;    //表示拉普拉斯金字塔
45. **if** (can\_use\_gpu\_ && img\_with\_border.depth() == CV\_16S)
46. createLaplacePyrGpu(img\_with\_border, num\_bands\_, src\_pyr\_laplace);
47. **else**
48. //创建拉普拉斯金字塔，该函数在后面给出介绍
49. createLaplacePyr(img\_with\_border, num\_bands\_, src\_pyr\_laplace);
51. // Create the weight map Gaussian pyramid
52. Mat weight\_map;    //表示权值映射图
53. //表示权值高斯金字塔
54. vector<Mat> weight\_pyr\_gauss(num\_bands\_ + 1);
55. //把mask传递给weight\_map
56. **if**(weight\_type\_ == CV\_32F)
57. {
58. mask.convertTo(weight\_map, CV\_32F, 1./255.);    //weight\_map = mask / 255
59. }
60. **else**// weight\_type\_ == CV\_16S
61. {
62. mask.convertTo(weight\_map, CV\_16S);    //weight\_map = mask
63. add(weight\_map, 1, weight\_map, mask != 0);    //weight\_map = weight\_map + 1
64. }
65. //对weight\_map通过赋予一个恒定常数的方式填充扩充了边界后的像素，得到weight\_pyr\_gauss[0]
66. copyMakeBorder(weight\_map, weight\_pyr\_gauss[0], top, bottom, left, right, BORDER\_CONSTANT);
67. //建立权值高斯金字塔
68. **for** (**int** i = 0; i < num\_bands\_; ++i)
69. pyrDown(weight\_pyr\_gauss[i], weight\_pyr\_gauss[i + 1]);
70. //得到金字塔底层图像的左上角和右下角坐标
71. **int** y\_tl = tl\_new.y - dst\_roi\_.y;
72. **int** y\_br = br\_new.y - dst\_roi\_.y;
73. **int** x\_tl = tl\_new.x - dst\_roi\_.x;
74. **int** x\_br = br\_new.x - dst\_roi\_.x;
76. // Add weighted layer of the source image to the final Laplacian pyramid layer
77. **if**(weight\_type\_ == CV\_32F)
78. {
79. **for** (**int** i = 0; i <= num\_bands\_; ++i)    //遍历金字塔各个层
80. {
81. **for** (**int** y = y\_tl; y < y\_br; ++y)    //遍历当前层的各个行
82. {
83. **int** y\_ = y - y\_tl;
84. //得到各个图像的当前行的首地址指针
85. **const** Point3\_<**short**>\* src\_row = src\_pyr\_laplace[i].ptr<Point3\_<**short**> >(y\_);
86. Point3\_<**short**>\* dst\_row = dst\_pyr\_laplace\_[i].ptr<Point3\_<**short**> >(y);
87. **const** **float**\* weight\_row = weight\_pyr\_gauss[i].ptr<**float**>(y\_);
88. **float**\* dst\_weight\_row = dst\_band\_weights\_[i].ptr<**float**>(y);
90. **for** (**int** x = x\_tl; x < x\_br; ++x)    //遍历当前行的各个像素
91. {
92. **int** x\_ = x - x\_tl;
93. //式92的分子
94. dst\_row[x].x += **static\_cast**<**short**>(src\_row[x\_].x \* weight\_row[x\_]);
95. dst\_row[x].y += **static\_cast**<**short**>(src\_row[x\_].y \* weight\_row[x\_]);
96. dst\_row[x].z += **static\_cast**<**short**>(src\_row[x\_].z \* weight\_row[x\_]);
97. dst\_weight\_row[x] += weight\_row[x\_];    //式92的分母
98. }
99. }
100. //表示的含义是降采样的图像面积缩小了一倍
101. x\_tl /= 2; y\_tl /= 2;
102. x\_br /= 2; y\_br /= 2;
103. }
104. }
105. **else**    // weight\_type\_ == CV\_16S
106. {
107. **for** (**int** i = 0; i <= num\_bands\_; ++i)    //遍历金字塔的各个层
108. {
109. **for** (**int** y = y\_tl; y < y\_br; ++y)    //遍历当前层的各行
110. {
111. **int** y\_ = y - y\_tl;
112. //得到各个图像的当前行的首地址指针
113. **const** Point3\_<**short**>\* src\_row = src\_pyr\_laplace[i].ptr<Point3\_<**short**> >(y\_);
114. Point3\_<**short**>\* dst\_row = dst\_pyr\_laplace\_[i].ptr<Point3\_<**short**> >(y);
115. **const** **short**\* weight\_row = weight\_pyr\_gauss[i].ptr<**short**>(y\_);
116. **short**\* dst\_weight\_row = dst\_band\_weights\_[i].ptr<**short**>(y);
118. **for** (**int** x = x\_tl; x < x\_br; ++x)    //遍历当前行的各个像素
119. {
120. **int** x\_ = x - x\_tl;
121. //式92的分子
122. dst\_row[x].x += **short**((src\_row[x\_].x \* weight\_row[x\_]) >> 8);
123. dst\_row[x].y += **short**((src\_row[x\_].y \* weight\_row[x\_]) >> 8);
124. dst\_row[x].z += **short**((src\_row[x\_].z \* weight\_row[x\_]) >> 8);
125. dst\_weight\_row[x] += weight\_row[x\_];    //式92的分母
126. }
127. }
128. //表示的含义是降采样的图像面积缩小了一倍
129. x\_tl /= 2; y\_tl /= 2;
130. x\_br /= 2; y\_br /= 2;
131. }
132. }
133. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** MultiBandBlender::blend(Mat &dst, Mat &dst\_mask)
2. {
3. //遍历金字塔的各个层，得到归一化后的各个层的图像，计算式92
4. **for** (**int** i = 0; i <= num\_bands\_; ++i)
5. normalizeUsingWeightMap(dst\_band\_weights\_[i], dst\_pyr\_laplace\_[i]);
7. **if** (can\_use\_gpu\_)
8. restoreImageFromLaplacePyrGpu(dst\_pyr\_laplace\_);
9. **else**
10. restoreImageFromLaplacePyr(dst\_pyr\_laplace\_);    //得到融合金字塔
12. dst\_ = dst\_pyr\_laplace\_[0];    //得到融合金字塔的底层图像，即最终的多频段融合图像
13. //把融合图像还原回融合之前的图像尺寸
14. dst\_ = dst\_(Range(0, dst\_roi\_final\_.height), Range(0, dst\_roi\_final\_.width));
15. dst\_mask\_ = dst\_band\_weights\_[0] > WEIGHT\_EPS;    //得到融合图像的掩码
16. //把掩码尺寸还原回融合之前的图像尺寸
17. dst\_mask\_ = dst\_mask\_(Range(0, dst\_roi\_final\_.height), Range(0, dst\_roi\_final\_.width));
18. dst\_pyr\_laplace\_.clear();    //释放内存
19. dst\_band\_weights\_.clear();    //释放内存
21. Blender::blend(dst, dst\_mask);    //调用Blender::blend函数
22. }

在多频段融合算法中，创建拉普拉斯金字塔函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** createLaplacePyr(**const** Mat &img, **int** num\_levels, vector<Mat> &pyr)
2. //img表示原图
3. //num\_levels表示金字塔的层数，也就是多频段的频段数
4. //pyr表示最终得到的金字塔
5. {
6. #ifdef HAVE\_TEGRA\_OPTIMIZATION
7. **if**(tegra::createLaplacePyr(img, num\_levels, pyr))
8. **return**;
9. #endif
11. pyr.resize(num\_levels + 1);    //建立金字塔
13. **if**(img.depth() == CV\_8U)    //图像像素为8位
14. {
15. **if**(num\_levels == 0)    //表示不需要建立拉普拉斯金字塔
16. {
17. img.convertTo(pyr[0], CV\_16S);    //原图赋予底层
18. **return**;    //函数不再做任何处理，直接返回
19. }
21. Mat downNext;    //表示降采样后的图像
22. Mat current = img;    //表示当前层的图像
23. //调用pyrDown函数，对img图像进行高斯模糊和降采样，得到downNext
24. pyrDown(img, downNext);
26. **for**(**int** i = 1; i < num\_levels; ++i)    //遍历金字塔的各个层
27. {
28. Mat lvl\_up;    //表示对当前图像进行升采样得到的图像
29. Mat lvl\_down;    //表示对当前图像进行降采样得到的图像
30. //调用pyrDown函数，得到lvl\_down
31. pyrDown(downNext, lvl\_down);
32. //调用pyrUp函数，得到lvl\_up
33. pyrUp(downNext, lvl\_up, current.size());    //式93中的expand运算
34. //式93，pyr[i-1]= current - lvl\_up
35. subtract(current, lvl\_up, pyr[i-1], noArray(), CV\_16S);
37. current = downNext;    //更新current
38. downNext = lvl\_down;    //更新downNext
39. }
40. //得到金字塔的顶层图像
41. {
42. Mat lvl\_up;
43. pyrUp(downNext, lvl\_up, current.size());
44. subtract(current, lvl\_up, pyr[num\_levels-1], noArray(), CV\_16S);
46. downNext.convertTo(pyr[num\_levels], CV\_16S);
47. }
48. }
49. **else**    //图像像素的深度为其他情况
50. {
51. pyr[0] = img;    //金字塔的底层图像
52. **for** (**int** i = 0; i < num\_levels; ++i)    //遍历金字塔的各个层
53. pyrDown(pyr[i], pyr[i + 1]);    //得到各个层的图像
54. Mat tmp;
55. **for** (**int** i = 0; i < num\_levels; ++i)    //再次遍历金字塔的各个层
56. {
57. pyrUp(pyr[i + 1], tmp, pyr[i].size());    //式93中的expand运算
58. subtract(pyr[i], tmp, pyr[i]);    //式93
59. }
60. }
61. }

得到融合金字塔：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. **void** restoreImageFromLaplacePyr(vector<Mat> &pyr)
2. //pyr作为输入表示拉普拉斯金字塔，作为输出表示融合金字塔
3. {
4. **if** (pyr.empty())
5. **return**;
6. Mat tmp;
7. **for** (**size\_t** i = pyr.size() - 1; i > 0; --i)    //遍历拉普拉斯金字塔的各层
8. {
9. pyrUp(pyr[i], tmp, pyr[i - 1].size());    //式94中的expand运算
10. add(tmp, pyr[i - 1], pyr[i - 1]);    //式94
11. }
12. }

7.3 应用

下面给出融合的应用：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78960325) [copy](http://blog.csdn.net/zhaocj/article/details/78960325)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. #include <iomanip>
23. **using** **namespace** cv;
24. **using** **namespace** std;
25. **using** **namespace** detail;
27. **int** main(**int** argc, **char**\*\* argv)
28. {
29. vector<Mat> imgs;    //输入图像
30. Mat img = imread("1.jpg");
31. imgs.push\_back(img);
32. img = imread("2.jpg");
33. imgs.push\_back(img);
35. Ptr<FeaturesFinder> finder;    //特征检测
36. finder = **new** SurfFeaturesFinder();
37. vector<ImageFeatures> features(2);
38. (\*finder)(imgs[0], features[0]);
39. (\*finder)(imgs[1], features[1]);
41. vector<MatchesInfo> pairwise\_matches;    //特征匹配
42. BestOf2NearestMatcher matcher(**false**, 0.3f, 6, 6);
43. matcher(features, pairwise\_matches);
45. HomographyBasedEstimator estimator;    //相机参数评估
46. vector<CameraParams> cameras;
47. estimator(features, pairwise\_matches, cameras);
48. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
49. {
50. Mat R;
51. cameras[i].R.convertTo(R, CV\_32F);
52. cameras[i].R = R;
53. }
55. Ptr<detail::BundleAdjusterBase> adjuster;    //相机参数精确评估
56. adjuster = **new** detail::BundleAdjusterReproj();
57. adjuster->setConfThresh(1);
58. (\*adjuster)(features, pairwise\_matches, cameras);
60. vector<Mat> rmats;
61. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
62. rmats.push\_back(cameras[i].R.clone());
63. waveCorrect(rmats, WAVE\_CORRECT\_HORIZ);    //波形校正
64. **for** (**size\_t** i = 0; i < cameras.size(); ++i)
65. cameras[i].R = rmats[i];
67. //图像映射变换
68. vector<Point> corners(2);
69. vector<Mat> masks\_warped(2);
70. vector<Mat> images\_warped(2);
71. vector<Size> sizes(2);
72. vector<Mat> masks(2);
73. **for** (**int** i = 0; i < 2; ++i)
74. {
75. masks[i].create(imgs[i].size(), CV\_8U);
76. masks[i].setTo(Scalar::all(255));
77. }
78. Ptr<WarperCreator> warper\_creator;
79. warper\_creator = **new** cv::PlaneWarper();
80. Ptr<RotationWarper> warper = warper\_creator->create(**static\_cast**<**float**>(cameras[0].focal));
81. **for** (**int** i = 0; i < 2; ++i)
82. {
83. Mat\_<**float**> K;
84. cameras[i].K().convertTo(K, CV\_32F);
85. corners[i] = warper->warp(imgs[i], K, cameras[i].R, INTER\_LINEAR, BORDER\_REFLECT, images\_warped[i]);
86. sizes[i] = images\_warped[i].size();
87. warper->warp(masks[i], K, cameras[i].R, INTER\_NEAREST, BORDER\_CONSTANT, masks\_warped[i]);
88. }
90. //曝光补偿
91. Ptr<ExposureCompensator> compensator =
92. ExposureCompensator::createDefault(ExposureCompensator::GAIN);
93. compensator->feed(corners, images\_warped, masks\_warped);
94. **for**(**int** i=0;i<2;++i)
95. {
96. compensator->apply(i, corners[i], images\_warped[i], masks\_warped[i]);
97. }
99. //在后面，我们还需要用到映射变换图的掩码masks\_warped，因此这里为该变量添加一个副本masks\_seam
100. vector<Mat> masks\_seam(2);
101. **for**(**int** i = 0; i<2;i++)
102. masks\_warped[i].copyTo(masks\_seam[i]);
104. //寻找接缝线
105. Ptr<SeamFinder> seam\_finder;
106. seam\_finder = **new** GraphCutSeamFinder(GraphCutSeamFinder::COST\_COLOR\_GRAD);
107. vector<Mat> images\_warped\_f(2);
108. **for** (**int** i = 0; i < 2; ++i)
109. images\_warped[i].convertTo(images\_warped\_f[i], CV\_32F);
110. seam\_finder->find(images\_warped\_f, corners, masks\_seam);
112. //图像融合
113. Ptr<Blender> blender;    //定义图像融合器
115. //blender = Blender::createDefault(Blender::NO, false);    //简单融合方法
116. /\*\*\*\*羽化融合方法\*\*\*\*\*\*\*\*\*
117. blender = Blender::createDefault(Blender::FEATHER, false);
118. FeatherBlender\* fb = dynamic\_cast<FeatherBlender\*>(static\_cast<Blender\*>(blender));
119. fb->setSharpness(0.005);    //设置羽化锐度
120. \*\*\*\*\*\*\*\*\*\*\*\*\*\*/
121. blender = Blender::createDefault(Blender::MULTI\_BAND, **false**);    //多频段融合
122. MultiBandBlender\* mb = **dynamic\_cast**<MultiBandBlender\*>(**static\_cast**<Blender\*>(blender));
123. //设置频段数，即金字塔层数，原则上频段越多越好
124. mb->setNumBands(8);
126. blender->prepare(corners, sizes);    //生成全景图像区域
128. //在融合的时候，最重要的是在接缝线两侧进行处理，而上一步在寻找接缝线后得到的掩码的边界就是接缝线处，因此我们还需要在接缝线两侧开辟一块区域用于融合处理，这一处理过程对羽化方法尤为关键
129. //应用膨胀算法缩小掩码面积
130. vector<Mat> dilate\_img(2);
131. Mat element = getStructuringElement(MORPH\_RECT, Size(20, 20));    //定义结构元素
132. vector<Mat> images\_warped\_s(2);
133. **for**(**int** k=0;k<2;k++)    //遍历所有图像
134. {
136. images\_warped\_f[k].convertTo(images\_warped\_s[k], CV\_16S);    //改变数据类型
137. dilate(masks\_seam[k], masks\_seam[k], element);    //膨胀运算
138. //映射变换图的掩码和膨胀后的掩码相“与”，从而使扩展的区域仅仅限于接缝线两侧，其他边界处不受影响
139. masks\_seam[k] = masks\_seam[k] & masks\_warped[k];
140. blender->feed(images\_warped\_s[k], masks\_seam [k], corners[k]);    //初始化数据
141. }
143. Mat result, result\_mask;
144. //完成融合操作，得到全景图像result和它的掩码result\_mask
145. blender->blend(result, result\_mask);
147. imwrite("pano.jpg", result);    //存储全景图像
148. **return** 0;
149. }

最终的全景图像为：



图17 全景图像

# [Opencv2.4.9源码分析——Stitching（八）](http://blog.csdn.net/zhaocj/article/details/78967041)

8、完整的拼接程序

下面给出完整的拼接程序：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78967041) [copy](http://blog.csdn.net/zhaocj/article/details/78967041)

1. #include "opencv2/core/core.hpp"
2. #include "highgui.h"
3. #include "opencv2/imgproc/imgproc.hpp"
4. #include "opencv2/features2d/features2d.hpp"
5. #include "opencv2/nonfree/nonfree.hpp"
6. #include "opencv2/legacy/legacy.hpp"
8. #include "opencv2/stitching/detail/autocalib.hpp"
9. #include "opencv2/stitching/detail/blenders.hpp"
10. #include "opencv2/stitching/detail/camera.hpp"
11. #include "opencv2/stitching/detail/exposure\_compensate.hpp"
12. #include "opencv2/stitching/detail/matchers.hpp"
13. #include "opencv2/stitching/detail/motion\_estimators.hpp"
14. #include "opencv2/stitching/detail/seam\_finders.hpp"
15. #include "opencv2/stitching/detail/util.hpp"
16. #include "opencv2/stitching/detail/warpers.hpp"
17. #include "opencv2/stitching/warpers.hpp"
19. #include <iostream>
20. #include <fstream>
21. #include <string>
22. #include <iomanip>
23. **using** **namespace** cv;
24. **using** **namespace** std;
25. **using** **namespace** detail;
27. **int** main(**int** argc, **char**\*\* argv)
28. {
29. vector<Mat> imgs;    //输入9幅图像
30. Mat img;
31. img = imread("1.jpg");
32. imgs.push\_back(img);
33. img = imread("2.jpg");
34. imgs.push\_back(img);
35. img = imread("3.jpg");
36. imgs.push\_back(img);
37. img = imread("4.jpg");
38. imgs.push\_back(img);
39. img = imread("5.jpg");
40. imgs.push\_back(img);
41. img = imread("6.jpg");
42. imgs.push\_back(img);
43. img = imread("7.jpg");
44. imgs.push\_back(img);
45. img = imread("8.jpg");
46. imgs.push\_back(img);
47. img = imread("9.jpg");
48. imgs.push\_back(img);
50. **int** num\_images = 9;    //图像数量
52. Ptr<FeaturesFinder> finder;    //定义特征寻找器
53. finder = **new** SurfFeaturesFinder();    //应用SURF方法寻找特征
54. //finder = new OrbFeaturesFinder();    //应用ORB方法寻找特征
55. vector<ImageFeatures> features(num\_images);    //表示图像特征
56. **for** (**int** i =0 ;i<num\_images;i++)
57. (\*finder)(imgs[i], features[i]);    //特征检测
59. vector<MatchesInfo> pairwise\_matches;    //表示特征匹配信息变量
60. BestOf2NearestMatcher matcher(**false**, 0.3f, 6, 6);    //定义特征匹配器，2NN方法
61. matcher(features, pairwise\_matches);    //进行特征匹配
63. HomographyBasedEstimator estimator;    //定义参数评估器
64. vector<CameraParams> cameras;    //表示相机参数
65. estimator(features, pairwise\_matches, cameras);    //进行相机参数评估
67. **for** (**size\_t** i = 0; i < cameras.size(); ++i)    //转换相机旋转参数的数据类型
68. {
69. Mat R;
70. cameras[i].R.convertTo(R, CV\_32F);
71. cameras[i].R = R;
72. }
74. Ptr<detail::BundleAdjusterBase> adjuster;    //光束平差法，精确相机参数
75. adjuster = **new** detail::BundleAdjusterReproj();    //重映射误差方法
76. //adjuster = new detail::BundleAdjusterRay();    //射线发散误差方法
78. adjuster->setConfThresh(1);    //设置匹配置信度，该值设为1
79. (\*adjuster)(features, pairwise\_matches, cameras);    //精确评估相机参数
81. vector<Mat> rmats;
82. **for** (**size\_t** i = 0; i < cameras.size(); ++i)    //复制相机的旋转参数
83. rmats.push\_back(cameras[i].R.clone());
84. waveCorrect(rmats, WAVE\_CORRECT\_HORIZ);    //进行波形校正
85. **for** (**size\_t** i = 0; i < cameras.size(); ++i)    //相机参数赋值
86. cameras[i].R = rmats[i];
87. rmats.clear();    //清变量
89. vector<Point> corners(num\_images);    //表示映射变换后图像的左上角坐标
90. vector<Mat> masks\_warped(num\_images);    //表示映射变换后的图像掩码
91. vector<Mat> images\_warped(num\_images);    //表示映射变换后的图像
92. vector<Size> sizes(num\_images);    //表示映射变换后的图像尺寸
93. vector<Mat> masks(num\_images);    //表示源图的掩码
95. **for** (**int** i = 0; i < num\_images; ++i)    //初始化源图的掩码
96. {
97. masks[i].create(imgs[i].size(), CV\_8U);    //定义尺寸大小
98. masks[i].setTo(Scalar::all(255));    //全部赋值为255，表示源图的所有区域都使用
99. }
101. Ptr<WarperCreator> warper\_creator;    //定义图像映射变换创造器
102. warper\_creator = **new** cv::PlaneWarper();    //平面投影
103. //warper\_creator = new cv::CylindricalWarper();    //柱面投影
104. //warper\_creator = new cv::SphericalWarper();    //球面投影
105. //warper\_creator = new cv::FisheyeWarper();    //鱼眼投影
106. //warper\_creator = new cv::StereographicWarper();    //立方体投影
108. //定义图像映射变换器，设置映射的尺度为相机的焦距，所有相机的焦距都相同
109. Ptr<RotationWarper> warper = warper\_creator->create(**static\_cast**<**float**>(cameras[0].focal));
110. **for** (**int** i = 0; i < num\_images; ++i)
111. {
112. Mat\_<**float**> K;
113. cameras[i].K().convertTo(K, CV\_32F);    //转换相机内参数的数据类型
114. //对当前图像镜像投影变换，得到变换后的图像以及该图像的左上角坐标
115. corners[i] = warper->warp(imgs[i], K, cameras[i].R, INTER\_LINEAR, BORDER\_REFLECT, images\_warped[i]);
116. sizes[i] = images\_warped[i].size();    //得到尺寸
117. //得到变换后的图像掩码
118. warper->warp(masks[i], K, cameras[i].R, INTER\_NEAREST, BORDER\_CONSTANT, masks\_warped[i]);
119. }
121. imgs.clear();    //清变量
122. masks.clear();
124. //创建曝光补偿器，应用增益补偿方法
125. Ptr<ExposureCompensator> compensator =
126. ExposureCompensator::createDefault(ExposureCompensator::GAIN);
127. compensator->feed(corners, images\_warped, masks\_warped);    //得到曝光补偿器
128. **for**(**int** i=0;i<num\_images;++i)    //应用曝光补偿器，对图像进行曝光补偿
129. {
130. compensator->apply(i, corners[i], images\_warped[i], masks\_warped[i]);
131. }
133. //在后面，我们还需要用到映射变换图的掩码masks\_warped，因此这里为该变量添加一个副本masks\_seam
134. vector<Mat> masks\_seam(num\_images);
135. **for**(**int** i = 0; i<num\_images;i++)
136. masks\_warped[i].copyTo(masks\_seam[i]);
138. Ptr<SeamFinder> seam\_finder;    //定义接缝线寻找器
139. //seam\_finder = new NoSeamFinder();    //无需寻找接缝线
140. //seam\_finder = new VoronoiSeamFinder();    //逐点法
141. //seam\_finder = new DpSeamFinder(DpSeamFinder::COLOR);    //动态规范法
142. //seam\_finder = new DpSeamFinder(DpSeamFinder::COLOR\_GRAD);
143. //图割法
144. //seam\_finder = new GraphCutSeamFinder(GraphCutSeamFinder::COST\_COLOR);
145. seam\_finder = **new** GraphCutSeamFinder(GraphCutSeamFinder::COST\_COLOR\_GRAD);
147. vector<Mat> images\_warped\_f(num\_images);
148. **for** (**int** i = 0; i < num\_images; ++i)    //图像数据类型转换
149. images\_warped[i].convertTo(images\_warped\_f[i], CV\_32F);
151. images\_warped.clear();    //清内存
153. //得到接缝线的掩码图像masks\_seam
154. seam\_finder->find(images\_warped\_f, corners, masks\_seam);
156. vector<Mat> images\_warped\_s(num\_images);
157. Ptr<Blender> blender;    //定义图像融合器
159. //blender = Blender::createDefault(Blender::NO, false);    //简单融合方法
160. //羽化融合方法
161. //blender = Blender::createDefault(Blender::FEATHER, false);
162. //FeatherBlender\* fb = dynamic\_cast<FeatherBlender\*>(static\_cast<Blender\*>(blender));
163. //fb->setSharpness(0.005);    //设置羽化锐度
165. blender = Blender::createDefault(Blender::MULTI\_BAND, **false**);    //多频段融合
166. MultiBandBlender\* mb = **dynamic\_cast**<MultiBandBlender\*>(**static\_cast**<Blender\*>(blender));
167. mb->setNumBands(8);   //设置频段数，即金字塔层数
169. blender->prepare(corners, sizes);    //生成全景图像区域
171. //在融合的时候，最重要的是在接缝线两侧进行处理，而上一步在寻找接缝线后得到的掩码的边界就是接缝线处，因此我们还需要在接缝线两侧开辟一块区域用于融合处理，这一处理过程对羽化方法尤为关键
172. //应用膨胀算法缩小掩码面积
173. vector<Mat> dilate\_img(num\_images);
174. Mat element = getStructuringElement(MORPH\_RECT, Size(20, 20));    //定义结构元素
175. **for**(**int** k=0;k<num\_images;k++)
176. {
177. images\_warped\_f[k].convertTo(images\_warped\_s[k], CV\_16S);    //改变数据类型
178. dilate(masks\_seam[k], masks\_seam[k], element);    //膨胀运算
179. //映射变换图的掩码和膨胀后的掩码相“与”，从而使扩展的区域仅仅限于接缝线两侧，其他边界处不受影响
180. masks\_seam[k] = masks\_seam[k] & masks\_warped[k];
181. blender->feed(images\_warped\_s[k], masks\_seam[k], corners[k]);    //初始化数据
182. }
184. masks\_seam.clear();    //清内存
185. images\_warped\_s.clear();
186. masks\_warped.clear();
187. images\_warped\_f.clear();
189. Mat result, result\_mask;
190. //完成融合操作，得到全景图像result和它的掩码result\_mask
191. blender->blend(result, result\_mask);
193. imwrite("pano.jpg", result);    //存储全景图像
195. **return** 0;
196. }

最终的输出图像为：



图18 平面映射全景图像

输入的9幅图像的尺寸都为979×550，最终共耗时10分钟左右。全景拼接程序十分消耗内存空间，如果图像的尺寸较大，而且图像数量较多的话，不仅耗时较长，而且很可能由于内存不足而报错。另外，经过多次实验看出，如果图像尺寸大、数量多，还会引起拼接不正确的现象。

下面几幅图像分别为不同映射得到的全景图像。



图19 柱面映射全景图像



图20 球面映射全景图像



图21 鱼眼映射全景图像（该图调整了角度，以便于显示观看）



图22 立体映射全景图像（该图调整了角度，以便于显示观看）

# [Opencv2.4.9源码分析——Stitching（九）](http://blog.csdn.net/zhaocj/article/details/78989548)

9、Stitcher类

图像拼接方法用到的算法较多，内容较复杂，可能对于一些人来说用起来过于繁琐，因此Opencv把拼接算法封装到了Stitcher类中，这样就不必理会拼接算法中的具体实现过程。

下面我们就简单介绍一下Stitcher类中的一些常用的函数：

用系统缺省值创建图像拼接器stitcher：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher Stitcher::createDefault(**bool** try\_use\_gpu)
2. //try\_use\_gpu表示是否应用图像处理器
3. {
4. Stitcher stitcher;    //实例化Stitcher类
5. //设置全局变量registr\_resol\_为0.6，registr\_resol\_表示图像配准分辨率
6. stitcher.setRegistrationResol(0.6);
7. //设置全局变量seam\_est\_resol\_为0.1，seam\_est\_resol\_表示接缝分辨率
8. stitcher.setSeamEstimationResol(0.1);
9. //ORIG\_RESOL = -1，设置全局变量compose\_resol\_为ORIG\_RESOL，compose\_resol\_表示合成分辨率
10. stitcher.setCompositingResol(ORIG\_RESOL);
11. //设置全局变量conf\_thresh\_为1，conf\_thresh\_表示匹配置信度，即式25
12. stitcher.setPanoConfidenceThresh(1);
13. //设置全局变量do\_wave\_correct\_为true，do\_wave\_correct\_表示是否进行波形校正
14. stitcher.setWaveCorrection(**true**);
15. //设置全局变量wave\_correct\_kind\_为WAVE\_CORRECT\_HORIZ，wave\_correct\_kind\_表示波形校正的方式，这里是用的水平校正
16. stitcher.setWaveCorrectKind(detail::WAVE\_CORRECT\_HORIZ);
17. //设置全局变量features\_matcher\_为BestOf2NearestMatcher()，features\_matcher\_表示特征匹配方法，这里是用2-NN方法
18. stitcher.setFeaturesMatcher(**new** detail::BestOf2NearestMatcher(try\_use\_gpu));
19. //设置全局变量bundle\_adjuster\_为BundleAdjusterRay()，bundle\_adjuster\_表示光束平差法，这里用的是射线发散方法
20. stitcher.setBundleAdjuster(**new** detail::BundleAdjusterRay());
22. #if defined(HAVE\_OPENCV\_GPU) && !defined(DYNAMIC\_CUDA\_SUPPORT)
23. **if** (try\_use\_gpu && gpu::getCudaEnabledDeviceCount() > 0)
24. {
25. #if defined(HAVE\_OPENCV\_NONFREE)
26. stitcher.setFeaturesFinder(**new** detail::SurfFeaturesFinderGpu());
27. #else
28. stitcher.setFeaturesFinder(**new** detail::OrbFeaturesFinder());
29. #endif
30. stitcher.setWarper(**new** SphericalWarperGpu());
31. stitcher.setSeamFinder(**new** detail::GraphCutSeamFinderGpu());
32. }
33. **else**
34. #endif
35. {
36. #ifdef HAVE\_OPENCV\_NONFREE    //表示可以使用NONFREE
37. //设置全局变量features\_finder\_为SurfFeaturesFinder()，features\_finder\_表示特征检测的方法，在这里是用SURF算法
38. stitcher.setFeaturesFinder(**new** detail::SurfFeaturesFinder());
39. #else    //表示不可以使用NONFREE
40. //设置全局变量features\_finder\_为OrbFeaturesFinder()，在这里用的是ORB算法
41. stitcher.setFeaturesFinder(**new** detail::OrbFeaturesFinder());
42. #endif
43. //设置全局变量warper\_为SphericalWarper()，warper\_表示图像投影变换的方法，在这里是球面投影方法
44. stitcher.setWarper(**new** SphericalWarper());
45. //设置全局变量seam\_finder\_为GraphCutSeamFinderBase::COST\_COLOR，seam\_finder\_表示寻找接缝线的算法，在这里是图割法，而且误差表面函数是直接法
46. stitcher.setSeamFinder(**new** detail::GraphCutSeamFinder(detail::GraphCutSeamFinderBase::COST\_COLOR));
47. }
48. //设置全局变量exposure\_comp\_为BlocksGainCompensator()，exposure\_comp\_表示曝光补偿方法，在这里是分块增益补偿方法
49. stitcher.setExposureCompensator(**new** detail::BlocksGainCompensator());
50. //设置全局变量blender\_为MultiBandBlender()，blender\_表示融合算法，在这里是多频段融合方法
51. stitcher.setBlender(**new** detail::MultiBandBlender(try\_use\_gpu));
53. **return** stitcher;    //返回Stitcher类
54. }

estimateTransform函数用于匹配图像，并评估相机的旋转参数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher::Status Stitcher::estimateTransform(InputArray images)
2. //images表示待拼接的输入图像
3. {
4. //调用另一种形式的estimateTransform函数
5. **return** estimateTransform(images, vector<vector<Rect> >());
6. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher::Status Stitcher::estimateTransform(InputArray images, **const** vector<vector<Rect> > &rois)
2. //images表示待拼接的输入图像
3. //rois表示输入图像中感兴趣的矩形区域，即只对该区域进行拼接
4. {
5. images.getMatVector(imgs\_);    //图像赋值
6. rois\_ = rois;    //赋值
8. Status status;
9. //调用matchImages函数，用于匹配图像，该函数在后面给出介绍
10. **if** ((status = matchImages()) != OK)
11. **return** status;
12. //调用estimateCameraParams函数，用于评估相机参数，该函数在后面给出介绍
13. estimateCameraParams();
15. **return** OK;
16. }

composePanorama函数用于合并拼接图像：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher::Status Stitcher::composePanorama(OutputArray pano)
2. //pano表示最终得到的全景图像
3. {
4. //调用另一种形式的composePanorama函数
5. **return** composePanorama(vector<Mat>(), pano);
6. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher::Status Stitcher::composePanorama(InputArray images, OutputArray pano)
2. //images表示经过变形处理后的图像
3. //pano表示最终得到的全景图像
4. {
5. LOGLN("Warping images (auxiliary)... ");
7. vector<Mat> imgs;
8. images.getMatVector(imgs);    //赋值
9. //如果imgs不为空，即输入参数images有值，则需要把images的相关值赋予imgs\_
10. **if** (!imgs.empty())    //如果imgs不为空，即有值
11. {
12. CV\_Assert(imgs.size() == imgs\_.size());    //确保图像数量相同
14. Mat img;
15. //定义seam\_est\_imgs\_的数量，seam\_est\_imgs\_表示在进行接缝线算法后得到的图像集合，由于尺度变换，该图像的尺寸与源图尺寸会不一样
16. seam\_est\_imgs\_.resize(imgs.size());
18. **for** (**size\_t** i = 0; i < imgs.size(); ++i)    //遍历各个图像，改变图像尺寸
19. {
20. imgs\_[i] = imgs[i];    //赋值
21. //根据尺度seam\_scale\_变换当前图像的尺寸
22. resize(imgs[i], img, Size(), seam\_scale\_, seam\_scale\_);
23. seam\_est\_imgs\_[i] = img.clone();    //图像赋值
24. }
25. //下面两个变量分别表示图像seam\_est\_imgs\_和图像imgs\_的子集，这两个集合的图像只是图像尺寸不同
26. vector<Mat> seam\_est\_imgs\_subset;
27. vector<Mat> imgs\_subset;
28. //indices\_表示真正可以拼接在一幅全景图像的待拼接图像的数量，即可能有些图像是不属于拼接图像的，indices\_由leaveBiggestComponent得到
29. **for** (**size\_t** i = 0; i < indices\_.size(); ++i)    //遍历图像
30. {
31. //分别得到图像seam\_est\_imgs\_和图像imgs\_的子集
32. imgs\_subset.push\_back(imgs\_[indices\_[i]]);    //放入队列
33. seam\_est\_imgs\_subset.push\_back(seam\_est\_imgs\_[indices\_[i]]);    //放入队列
34. }
36. seam\_est\_imgs\_ = seam\_est\_imgs\_subset;    //赋值
37. imgs\_ = imgs\_subset;    //赋值
38. }
40. Mat &pano\_ = pano.getMatRef();    //赋值
42. #if ENABLE\_LOG
43. int64 t = getTickCount();    //用于计时
44. #endif
45. //表示经过映射变换后，各个图像在全景图像坐标系上的左上角坐标
46. vector<Point> corners(imgs\_.size());
47. vector<Mat> masks\_warped(imgs\_.size());    //表示映射变换图的掩码
48. vector<Mat> images\_warped(imgs\_.size());    //表示映射变换图
49. vector<Size> sizes(imgs\_.size());    //表示映射变换图的尺寸
50. vector<Mat> masks(imgs\_.size());    //表示映射变换前的图像掩码
52. // Prepare image masks
53. **for** (**size\_t** i = 0; i < imgs\_.size(); ++i)    //遍历图像，准备掩码
54. {
55. masks[i].create(seam\_est\_imgs\_[i].size(), CV\_8U);    //定义尺寸大小
56. masks[i].setTo(Scalar::all(255));    //掩码矩阵先赋值为255，表示不掩码
57. }
59. // Warp images and their masks
60. //定义图像映射变换类，这里的warped\_image\_scale\_就是焦距，seam\_work\_aspect\_表示接缝尺度与匹配尺度的比值，warped\_image\_scale\_和seam\_work\_aspect\_的乘积表示映射变换的尺度
61. Ptr<detail::RotationWarper> w = warper\_->create(**float**(warped\_image\_scale\_ \* seam\_work\_aspect\_));
62. **for** (**size\_t** i = 0; i < imgs\_.size(); ++i)    //遍历所有图像
63. {
64. Mat\_<**float**> K;
65. cameras\_[i].K().convertTo(K, CV\_32F);    //得到当前相机的内参数
66. //依据seam\_work\_aspect\_的大小调整相机的内参数，之所以要调整内参数，是因为得到内参数的图像与要映射变换的图像的尺寸不一样
67. K(0,0) \*= (**float**)seam\_work\_aspect\_;    //表示fu
68. K(0,2) \*= (**float**)seam\_work\_aspect\_;    //表示cx
69. K(1,1) \*= (**float**)seam\_work\_aspect\_;    //表示fv
70. K(1,2) \*= (**float**)seam\_work\_aspect\_;    //表示cy
71. //对图像进行映射变换，得到图像映射图images\_warped
72. corners[i] = w->warp(seam\_est\_imgs\_[i], K, cameras\_[i].R, INTER\_LINEAR, BORDER\_REFLECT, images\_warped[i]);
73. sizes[i] = images\_warped[i].size();    //得到尺寸
74. //对掩码进行映射变换，得到掩码映射图masks\_warped
75. w->warp(masks[i], K, cameras\_[i].R, INTER\_NEAREST, BORDER\_CONSTANT, masks\_warped[i]);
76. }
78. vector<Mat> images\_warped\_f(imgs\_.size());
79. **for** (**size\_t** i = 0; i < imgs\_.size(); ++i)    //变换图像的数据类型
80. images\_warped[i].convertTo(images\_warped\_f[i], CV\_32F);
82. LOGLN("Warping images, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
84. // Find seams
85. //生成曝光补偿器
86. exposure\_comp\_->feed(corners, images\_warped, masks\_warped);
87. seam\_finder\_->find(images\_warped\_f, corners, masks\_warped);    //寻找接缝线
89. // Release unused memory
90. //释放一些内存空间
91. seam\_est\_imgs\_.clear();
92. images\_warped.clear();
93. images\_warped\_f.clear();
94. masks.clear();
96. LOGLN("Compositing...");
97. #if ENABLE\_LOG
98. t = getTickCount();
99. #endif
101. Mat img\_warped, img\_warped\_s;    //表示不同数据类型的映射变换图像
102. //分别表示不同过程中的掩码图像
103. Mat dilated\_mask, seam\_mask, mask, mask\_warped;
105. //double compose\_seam\_aspect = 1;
106. **double** compose\_work\_aspect = 1;    //表示合成尺度与匹配尺度之比
107. **bool** is\_blender\_prepared = **false**;    //表示融合器是否准备好
109. **double** compose\_scale = 1;    //表示合成尺度
110. **bool** is\_compose\_scale\_set = **false**;    //表示合成尺度是否设置好
112. Mat full\_img, img;
113. **for** (**size\_t** img\_idx = 0; img\_idx < imgs\_.size(); ++img\_idx)    //遍历所有图像
114. {
115. LOGLN("Compositing image #" << indices\_[img\_idx] + 1);
117. // Read image and resize it if necessary
118. full\_img = imgs\_[img\_idx];    //得到当前图像
119. **if** (!is\_compose\_scale\_set)    //compose\_scale还没有被设置
120. {
121. //compose\_resol\_被初始化为-1，因此compose\_scale并没有被重新赋值
122. **if** (compose\_resol\_ > 0)
123. compose\_scale = min(1.0, sqrt(compose\_resol\_ \* 1e6 / full\_img.size().area()));
124. //表示compose\_scale已被设置，则在下次遍历图像时，无需再调整映射变换图的尺寸和左上角坐标
125. is\_compose\_scale\_set = **true**;
127. // Compute relative scales
128. //compose\_seam\_aspect = compose\_scale / seam\_scale\_;
129. //得到合成尺度与匹配尺度之比
130. compose\_work\_aspect = compose\_scale / work\_scale\_;
132. // Update warped image scale
133. //调整映射变换尺度
134. warped\_image\_scale\_ \*= **static\_cast**<**float**>(compose\_work\_aspect);
135. //根据调整后的映射变换尺度，重新得到映射变换器
136. w = warper\_->create((**float**)warped\_image\_scale\_);
138. // Update corners and sizes
139. //更新映射变换图像的参数：左上角坐标和尺寸
140. **for** (**size\_t** i = 0; i < imgs\_.size(); ++i)    //遍历所有图像
141. {
142. // Update intrinsics
143. //更新相机的内参数
144. cameras\_[i].focal \*= compose\_work\_aspect;
145. cameras\_[i].ppx \*= compose\_work\_aspect;
146. cameras\_[i].ppy \*= compose\_work\_aspect;
148. // Update corner and size
149. Size sz = full\_img\_sizes\_[i];    //表示当前图像尺寸
150. **if** (std::abs(compose\_scale - 1) > 1e-1)    //调整图像尺寸
151. {
152. sz.width = cvRound(full\_img\_sizes\_[i].width \* compose\_scale);
153. sz.height = cvRound(full\_img\_sizes\_[i].height \* compose\_scale);
154. }
156. Mat K;
157. cameras\_[i].K().convertTo(K, CV\_32F);    //转换相机参数的数据格式
158. Rect roi = w->warpRoi(sz, K, cameras\_[i].R);    //得到映射变换图的矩形
159. corners[i] = roi.tl();    //映射变换图在全景图像坐标系上的左上角坐标
160. sizes[i] = roi.size();    //得到映射变换图的尺寸
161. }
162. }
163. //依据合成尺度compose\_scale，调整图像的尺寸
164. **if** (std::abs(compose\_scale - 1) > 1e-1)
165. resize(full\_img, img, Size(), compose\_scale, compose\_scale);
166. **else**
167. img = full\_img;
168. full\_img.release();    //释放一些内存空间
169. Size img\_size = img.size();
171. Mat K;
172. cameras\_[img\_idx].K().convertTo(K, CV\_32F);    //转换相机参数的数据格式
174. // Warp the current image
175. //对改变了尺寸的当前图像进行映射变换
176. w->warp(img, K, cameras\_[img\_idx].R, INTER\_LINEAR, BORDER\_REFLECT, img\_warped);
178. // Warp the current image mask
179. mask.create(img\_size, CV\_8U);
180. mask.setTo(Scalar::all(255));
181. //映射变换当前图像的掩码，得到映射变换图掩码
182. w->warp(mask, K, cameras\_[img\_idx].R, INTER\_NEAREST, BORDER\_CONSTANT, mask\_warped);
184. // Compensate exposure
185. //完成曝光补偿
186. exposure\_comp\_->apply((**int**)img\_idx, corners[img\_idx], img\_warped, mask\_warped);
188. img\_warped.convertTo(img\_warped\_s, CV\_16S);    //改变映射变换图的数据类型
189. img\_warped.release();    //释放一些内存空间
190. img.release();
191. mask.release();
193. // Make sure seam mask has proper size
194. //对因合成而改变图像尺寸之前的映射变换掩码（即对该函数内seam\_finder\_->find函数得到的掩码）进行膨胀运算，使掩码面积缩小，从而扩展了接缝线附近的区域
195. dilate(masks\_warped[img\_idx], dilated\_mask, Mat());
196. //再把掩码图像的尺寸调整为mask\_warped（即调整到因合成而改变图像尺寸下的映射变换掩码，由该函数内exposure\_comp\_->apply函数得到）的大小尺寸，
197. resize(dilated\_mask, seam\_mask, mask\_warped.size());
198. //“与”操作，得到更准确的掩码图像，即mask\_warped仅仅是扩展了接缝线附近的区域
199. mask\_warped = seam\_mask & mask\_warped;
201. //在该循环体的第一次循环的时候，准备融合器，即定义好全景图像的大小
202. **if** (!is\_blender\_prepared)
203. {
204. blender\_->prepare(corners, sizes);    //准备融合器
205. is\_blender\_prepared = **true**;    //置1，避免重复调用blender\_->prepare
206. }
208. // Blend the current image
209. //把当前图像的数据添加进融合器内
210. blender\_->feed(img\_warped\_s, mask\_warped, corners[img\_idx]);
211. }
213. Mat result, result\_mask;
214. blender\_->blend(result, result\_mask);    //得到全景图像result及它的掩码result\_mask
216. LOGLN("Compositing, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
218. // Preliminary result is in CV\_16SC3 format, but all values are in [0,255] range,
219. // so convert it to avoid user confusing
220. result.convertTo(pano\_, CV\_8U);    //得到全景图像pano\_
222. **return** OK;
223. }

如果对拼接的执行过程不熟悉，那么也是不清楚estimateTransform函数和composePanorama函数的具体含义和执行顺序的。好在Opencv还定义了stitch函数，它完全回避了拼接的具体执行过程，也就是直接调用该函数即可：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher::Status Stitcher::stitch(InputArray images, OutputArray pano)
2. //images表示输入的待拼接图像
3. //pano表示最终得到的全景图像
4. {
5. Status status = estimateTransform(images);    //调用estimateTransform函数
6. **if** (status != OK)
7. **return** status;
8. **return** composePanorama(pano);    //调用composePanorama函数
9. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher::Status Stitcher::stitch(InputArray images, **const** vector<vector<Rect> > &rois, OutputArray pano)
2. //images表示输入的待拼接图像
3. //rois表示输入图像中感兴趣的矩形区域，即只对该区域进行拼接
4. //pano表示最终得到的全景图像
5. {
6. Status status = estimateTransform(images, rois);    //调用estimateTransform函数
7. **if** (status != OK)
8. **return** status;
9. **return** composePanorama(pano);    //调用composePanorama函数
10. }

下面我们再给出前面函数用到的两个重要函数：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. Stitcher::Status Stitcher::matchImages()    //图像匹配
2. {
3. **if** ((**int**)imgs\_.size() < 2)    //只有两幅以上的图像才能拼接
4. {
5. LOGLN("Need more images");
6. **return** ERR\_NEED\_MORE\_IMGS;
7. }
9. work\_scale\_ = 1;    //表示匹配尺度
10. seam\_work\_aspect\_ = 1;    //表示work\_scale\_和seam\_scale\_的比值
11. seam\_scale\_ = 1;    //表接缝尺度
12. **bool** is\_work\_scale\_set = **false**;    //表示work\_scale\_是否被设置
13. **bool** is\_seam\_scale\_set = **false**;    //表示seam\_scale\_是否被设置
14. Mat full\_img, img;
15. features\_.resize(imgs\_.size());    //定义特征点的矢量数量
16. seam\_est\_imgs\_.resize(imgs\_.size());    //定义seam\_est\_imgs\_的矢量数量
17. full\_img\_sizes\_.resize(imgs\_.size());    //full\_img\_sizes\_表示图像尺寸
19. LOGLN("Finding features...");
20. #if ENABLE\_LOG
21. int64 t = getTickCount();
22. #endif
24. **for** (**size\_t** i = 0; i < imgs\_.size(); ++i)    //遍历所有图像，寻找特征点
25. {
26. full\_img = imgs\_[i];    //得到当前图像
27. full\_img\_sizes\_[i] = full\_img.size();    //得到当前图像的尺寸
28. //registr\_resol\_被设置为0.6，所以执行的是else内容
29. **if** (registr\_resol\_ < 0)
30. {
31. img = full\_img;
32. work\_scale\_ = 1;
33. is\_work\_scale\_set = **true**;
34. }
35. **else**
36. {
37. //没有设置work\_scale\_，则此时需要设置work\_scale\_
38. **if** (!is\_work\_scale\_set)
39. {
40. //由图像配准分辨率和图像面积设置匹配尺度
41. work\_scale\_ = min(1.0, sqrt(registr\_resol\_ \* 1e6 / full\_img.size().area()));
42. is\_work\_scale\_set = **true**;    //避免再次循环重复设置work\_scale\_
43. }
44. //由work\_scale\_调整图像尺寸
45. resize(full\_img, img, Size(), work\_scale\_, work\_scale\_);
46. }
47. //没有设置seam\_scale\_，则此时需要设置seam\_scale\_
48. **if** (!is\_seam\_scale\_set)
49. {
50. //由接缝分辨率和图像面积设置接缝尺度
51. seam\_scale\_ = min(1.0, sqrt(seam\_est\_resol\_ \* 1e6 / full\_img.size().area()));
52. //设置seam\_work\_aspect\_
53. seam\_work\_aspect\_ = seam\_scale\_ / work\_scale\_;
54. is\_seam\_scale\_set = **true**;    //避免再次循环重复设置seam\_scale\_
55. }
57. **if** (rois\_.empty())    //表示图像的所有区域都是寻找特征点的区域
58. (\*features\_finder\_)(img, features\_[i]);    //寻找图像img的特征点
59. **else**    //否则在指定区域内寻找特征点
60. {
61. vector<Rect> rois(rois\_[i].size());    //定义区域矩形
62. **for** (**size\_t** j = 0; j < rois\_[i].size(); ++j)    //得到该区域在当前图像的位置
63. {
64. Point tl(cvRound(rois\_[i][j].x \* work\_scale\_), cvRound(rois\_[i][j].y \* work\_scale\_));
65. Point br(cvRound(rois\_[i][j].br().x \* work\_scale\_), cvRound(rois\_[i][j].br().y \* work\_scale\_));
66. rois[j] = Rect(tl, br);    //得到区域矩形
67. }
68. //在图像img中的rois区域内寻找特征点
69. (\*features\_finder\_)(img, features\_[i], rois);
70. }
71. features\_[i].img\_idx = (**int**)i;    //赋值索引
72. LOGLN("Features in image #" << i+1 << ": " << features\_[i].keypoints.size());
74. resize(full\_img, img, Size(), seam\_scale\_, seam\_scale\_);    //改变图像尺寸
75. seam\_est\_imgs\_[i] = img.clone();    //赋值
76. }
78. // Do it to save memory
79. features\_finder\_->collectGarbage();    //收集内存碎片
80. full\_img.release();    //释放内存
81. img.release();    //释放内存
83. LOGLN("Finding features, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
85. LOG("Pairwise matching");
86. #if ENABLE\_LOG
87. t = getTickCount();
88. #endif
89. (\*features\_matcher\_)(features\_, pairwise\_matches\_, matching\_mask\_);    //特征点匹配
90. features\_matcher\_->collectGarbage();    //收集内存碎片
91. LOGLN("Pairwise matching, time: " << ((getTickCount() - t) / getTickFrequency()) << " sec");
93. // Leave only images we are sure are from the same panorama
94. //调用leaveBiggestComponent函数，得到全景图像集合（即输入图像中有可能有些图像并不能构成全景图像），indices\_为全景图像集合的图像索引
95. indices\_ = detail::leaveBiggestComponent(features\_, pairwise\_matches\_, (**float**)conf\_thresh\_);
96. vector<Mat> seam\_est\_imgs\_subset;    //表示seam\_est\_imgs\_的子集
97. vector<Mat> imgs\_subset;    //表示imgs\_的子集
98. vector<Size> full\_img\_sizes\_subset;    //表示imgs\_subset集合中图像的尺寸
99. **for** (**size\_t** i = 0; i < indices\_.size(); ++i)    //遍历全景图像集合中的所有图像
100. {
101. imgs\_subset.push\_back(imgs\_[indices\_[i]]);    //赋值
102. seam\_est\_imgs\_subset.push\_back(seam\_est\_imgs\_[indices\_[i]]);
103. full\_img\_sizes\_subset.push\_back(full\_img\_sizes\_[indices\_[i]]);
104. }
105. seam\_est\_imgs\_ = seam\_est\_imgs\_subset;    //赋值
106. imgs\_ = imgs\_subset;
107. full\_img\_sizes\_ = full\_img\_sizes\_subset;
109. **if** ((**int**)imgs\_.size() < 2)    //图像数量必须要大于2
110. {
111. LOGLN("Need more images");
112. **return** ERR\_NEED\_MORE\_IMGS;
113. }
115. **return** OK;
116. }

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. **void** Stitcher::estimateCameraParams()    //相机参数评估
2. {
3. detail::HomographyBasedEstimator estimator;    //表示相机参数评估器
4. estimator(features\_, pairwise\_matches\_, cameras\_);    //得到相机参数cameras\_
6. **for** (**size\_t** i = 0; i < cameras\_.size(); ++i)    //遍历所有的相机
7. {
8. Mat R;
9. cameras\_[i].R.convertTo(R, CV\_32F);    //变换旋转参数的数据类型
10. cameras\_[i].R = R;    //得到旋转参数
11. LOGLN("Initial intrinsic parameters #" << indices\_[i] + 1 << ":\n " << cameras\_[i].K());
12. }
14. bundle\_adjuster\_->setConfThresh(conf\_thresh\_);    //设置匹配置信度
15. //利用光束平差法精确化相机参数cameras\_
16. (\*bundle\_adjuster\_)(features\_, pairwise\_matches\_, cameras\_);
18. // Find median focal length and use it as final image scale
19. vector<**double**> focals;    //表示焦距
20. **for** (**size\_t** i = 0; i < cameras\_.size(); ++i)    //遍历所有相机，得到焦距
21. {
22. LOGLN("Camera #" << indices\_[i] + 1 << ":\n" << cameras\_[i].K());
23. focals.push\_back(cameras\_[i].focal);
24. }
26. std::sort(focals.begin(), focals.end());    //焦距排序
27. //把焦距的中间值作为图像映射变换尺度warped\_image\_scale\_
28. **if** (focals.size() % 2 == 1)
29. warped\_image\_scale\_ = **static\_cast**<**float**>(focals[focals.size() / 2]);
30. **else**
31. warped\_image\_scale\_ = **static\_cast**<**float**>(focals[focals.size() / 2 - 1] + focals[focals.size() / 2]) \* 0.5f;
33. **if** (do\_wave\_correct\_)    //进行波形校正
34. {
35. vector<Mat> rmats;
36. **for** (**size\_t** i = 0; i < cameras\_.size(); ++i)
37. rmats.push\_back(cameras\_[i].R);    //得到相机旋转参数
38. detail::waveCorrect(rmats, wave\_correct\_kind\_);    //波形校正
39. **for** (**size\_t** i = 0; i < cameras\_.size(); ++i)
40. cameras\_[i].R = rmats[i];    //校正后的旋转参数
41. }
42. }

通过以上分析，我们会发现，Stitcher类实现的拼接与第8节给出的实现方法最大的不同之处在于，Stitcher类在拼接的各个阶段，借助于尺度不断缩小图像的尺寸：特征检测与匹配是在work\_scale\_（匹配尺度）下进行的，映射变换是在seam\_scale\_（接缝尺度）下进行的，而图像合成又是在compose\_scale（合成尺度）下进行的。这么做的优点显而易见，既提高了效率，又节省了内存开销，但最终得到的全景图像的尺寸缩小了。

下面我们给出利用Stitcher类实现的图像拼接：

**[cpp]** [view plain](http://blog.csdn.net/zhaocj/article/details/78989548) [copy](http://blog.csdn.net/zhaocj/article/details/78989548)

1. #include <iostream>
2. #include <fstream>
3. #include <string>
4. #include "opencv2/opencv\_modules.hpp"
5. #include "opencv2/highgui/highgui.hpp"
6. #include "opencv2/stitching/stitcher.hpp"
7. **using** **namespace** std;
8. **using** **namespace** cv;
9. **using** **namespace** cv::detail;
11. **int** main( )
12. {
13. vector<Mat> imgs;    //输入9幅图像
14. Mat img;
15. img = imread("4.jpg");
16. imgs.push\_back(img);
17. img = imread("5.jpg");
18. imgs.push\_back(img);
19. img = imread("6.jpg");
20. imgs.push\_back(img);
21. img = imread("7.jpg");
22. imgs.push\_back(img);
23. img = imread("8.jpg");
24. imgs.push\_back(img);
25. img = imread("9.jpg");
26. imgs.push\_back(img);
27. img = imread("10.jpg");
28. imgs.push\_back(img);
29. img = imread("11.jpg");
30. imgs.push\_back(img);
31. img = imread("12.jpg");
32. imgs.push\_back(img);
34. **int** num\_images = 9;
36. Mat pano;    //全景图像
37. Stitcher stitcher = Stitcher::createDefault(**false**);    //定义全景图像拼接器
38. Stitcher::Status status = stitcher.stitch(imgs, pano);    //图像拼接
40. **if** (status != Stitcher::OK)
41. {
42. cout << "Can't stitch images, error code = " << **int**(status) << endl;
43. **return** -1;
44. }
46. imwrite("pano.jpg", pano);    //存储图像
47. **return** 0;
48. }

该程序输入的9幅图像与第8节中的9幅图像完全相同，最终用时不到4分钟。

我们最后总结一下Stitcher类所使用的各种默认算法：SURF特征检测算法（NONFREE条件下），2-NN特征匹配算法，光束平差法中的射线发散法，进行了波形水平校正，球面投影变换，曝光补偿法中的分块增益补偿法，直接图割法，以及多频段融合算法。