

# 作业六验报告及测试文档

数据科学与计算机学院 梁育诚 16340133

## 一、实验内容

1. Take pictures on a tripod (or handheld)
2. Warp images to spherical coordinates
3. Extract SIFT features and Match features(by KNN or Hashing)
4. Align neighboring pairs using RANSAC
5. Write out list of neighboring translations
6. Correct for drift
7. Read in warped images and blend them(using multi-scale blending or Poisson blending)
8. Crop the result and import into a viewer

## 二、实验原理

本次实验要做的是全景图的图像拼接。输入一系列图片，输出一张拼接后的图片。图像拼接是一个复杂的过程，其中可以细分为多个部分，每个部分都有自己的数学原理。我将先从整体将整个过程分析，然后逐个步骤分析。

图像拼接的核心在于找到两张图片重叠的部分，然后把他们拼在一起。找到重叠部分从图像处理的角度就是对特征值做吻合度的统计，而拼接则依赖于一系列的算法。找图片特征点常用的算法是 **SIFT**，这里直接用第三方库 **VLFEAT**。整个过程的难点也在于图片特征点之间匹配的估计，通过 **RANSAC** 算法来计算出比较不错的模型，求得两张图片特征点之间的关系模型，从而计算出平均位移。这个平均位移则可以用于后面的拼接了。

首先我们需要对原图进行预处理——曲面变换。这个是为了拼接后的视觉效果而做的。我们的输入是一系列拍摄的平面图片，如果只是直接将他们拼在一起，图片之间的边缘效果会很不好。如果先做柱面变换预处理，最后拼出来的图片就很有立体感，看起来也平滑很多。

然后是特征提取。这里首先要将图片转化为灰度图，这样可以加快特征提取的速度。然后调用 **SIFT** 算法来找到特征点。这里用到了 **VLFEAT** 库，然后参考了网上的一些教程来使用 **SIFT**，获得每张图片的特征点描述子序列。

接着就开始拼接了。我们需要先确定哪些图之间是相邻的，通过对比两幅图之间匹配的特征点数目，超过一定的值就认为两幅图是有大量吻合部分的，所以确定这两张图是相邻的。然后按照顺序来拼接图片。拼接图片要先找到特征点的最近邻，即每个特征点要匹配在另一幅图上的一个特征点。这个方法有很多，可以是盲搜，本次实验中用了搜索效率比较高的 **KD** 树来找最近邻。确定匹配的方向后，使用 **RANSAC** 算法来找到最佳模型，计算出两幅图的相对平均位移，最后将他们拼接在一起就可以了。**RANSAC** 算法又称作随机抽样一致算法，它将一组观测数据集中，通过迭代估计数学模型的参数，将其中的点分为“局内点”和“局外点”。相比起最小二乘拟合，它效率更高，虽然它是一种不确定的算法，但是它有一定的概率得出一个合理的结果。基本假设包括：（1）数据由“局内点”组成。（2）“局外点”是不能适应该模型的数据。（3）除此之外的数据是噪声。算

法的大致过程为：通过反复选择数据中的一组随机子集来达成目标。被选取的子集被假设为局内点，按下述方法进行验证：

①有一个模型适应于假设的局内点，即所有的未知参数都能从假设的局内点计算得出。

②用①中得到的参数模型去测试所有的其它数据，如果某个点适用于估计的模型，则认为它也是局内点。

③如果有足够多的点被归为假设的局内点，那么估计的模型就足够合理。

④然后用所有假设的局内点去重新估计模型，这里一般用最小二乘法，因为它仅仅被初始的假设局内点估计过。

在上面这个迭代过程中，每次产生的参数模型要么因为局内点太少而被舍弃（模型不合理），要么因为比现有的模型更好而被接纳（模型更加合理）。

### 三、实验过程

为了后续拓展方便，这次实验的代码结构按照算法步骤分成了很多个小模块。这样的好处是，日后优化某一个步骤的时候可以直接更改相关部分，而不用改动一整个类。代码按照实验步骤分为：**warping.h**（用于处理图像变换的）、**blending.h**（图像融合拼接）、**align\_feature.h**（RANSAC 算法及特征点对齐）、**stitching.h**（图像拼接的主类）。

1. 自上而下编程，首先实现 **stitching** 类。里面包含了几个函数：

（1）**extractFeatures()**：使用 SIFT 算法提取特征点

```
93 // 使用SIFT提取特征点
94 map<vector<float>, VLSiftKeypoint> extractFeatures(CImg<float>& img) {
95     CImg<float> src(img);
96     float resize_factor;
97     int width = src._width;
98     int height = src._height;
99
100     // 优化计算速度
101     if (width < height) {
102         resize_factor = RESIZE_SIZE / width;
103     }
104     else {
105         resize_factor = RESIZE_SIZE / height;
106     }
107
108     if (resize_factor >= 1) {
109         resize_factor = 1;
110     }
111     else {
112         src.resize(width * resize_factor, height * resize_factor, src.spectrum(), 3);
113     }
114
115     // vl_sift_pix 就是float型数据
116     vl_sift_pix *imageData = new vl_sift_pix[src._height * src._width];
117
118     // 设置SIFT算法过滤参数
119     // Setting SIFT filter params
120     int octaves = 4, nlevels = 2, o_min = 0;
121
122     // 图像二维转一维
123     for (int i = 0; i < src.width(); i++) {
124         for (int j = 0; j < src.height(); j++) {
125             imageData[j * src.width() + i] = src(i, j, 0);
126         }
127     }
```

```

129 // 这个过滤器实现了SIFT检测器和描述符
130 VLSiftFilt *sf = NULL;
131
132 // noctaves: numbers of octaves 组数
133 // nlevels: numbers of levels per octave 每组的层数
134 // o_min: first octave index 第一组的索引号
135
136 sf = vl_sift_new(src.width(), src.height(), noctaves, nlevels, o_min);
137
138 map<vector<float>, VLSiftKeypoint> features; // 记录特征点的描述符，一个特征点有可能有多个描述符，最多有4个
139
140 // Compute the first octave of the DOG scale space
141 // 这个函数开始处理一幅新图像，通过计算它在低层的高斯尺度空间
142 // 它还清空内部记录关键点的缓冲区
143 if (vl_sift_process_first_octave(sf, imageData) != VL_ERR_EOF) {
144     while (1) {
145         // Run the SIFT detector to get the keypoints.
146         vl_sift_detect(sf); // 计算每组中的关键点
147
148         VLSiftKeypoint *pKeyPoint = sf->keys;
149
150         // 遍历每个特征点
151         for (int i = 0; i < sf->nkeys; i++) {
152             VLSiftKeypoint tempKp = *pKeyPoint;
153
154             // 计算并遍历每个点的方向
155             double angles[4];
156
157             // 计算每个极值点的方向，包括主方向和辅方向，最多4个方向
158             int angleCount = vl_sift_calc_keypoint_orientations(sf, angles, &tempKp); // 方向数量
159
160             for (int j = 0; j < angleCount; j++) {
161                 // 计算每个方向的描述符
162                 vl_sift_pix descriptors[128];
163
164                 // 获取特征点的描述子
165                 vl_sift_calc_keypoint_descriptor(sf, descriptors, &tempKp, angles[j]);
166
167                 // 复制到vector
168                 vector<float> des;
169                 int k = 0;
170                 while (k < 128) {
171                     des.push_back(descriptors[k]);
172                     k++;
173                 }
174
175                 // 处理特征点信息
176                 tempKp.x /= resize_factor;
177                 tempKp.y /= resize_factor;
178                 tempKp.ix = tempKp.x;
179                 tempKp.iy = tempKp.y;
180
181                 features.insert(make_pair(des, tempKp)); // 插入到特征点map
182                 pKeyPoint++;
183             }
184             // 这个函数计算高斯尺度空间中的下一组尺度空间图像
185             // 这个函数会清除在前一层空间中检测到的特征点
186             if (vl_sift_process_next_octave(sf) == VL_ERR_EOF) {
187                 break;
188             }
189         }
190     }
191
192     // 释放资源
193     vl_sift_delete(sf);
194     delete[] imageData;
195     imageData = NULL;
196
197     return features;
198 }

```

(2) get\_gray\_image(): 转化为灰度图，加快处理 SIFT 处理速度

```

7 // 获得灰度图，加快处理速度
8 CImg<float> get_gray_image(CImg<float>& image) {
9     CImg<float> res(image._width, image._height);
10    cimg_forXY(image, x, y) {
11        res(x, y) = 0.299 * image(x, y, 0, 0) +
12                    0.587 * image(x, y, 0, 1) +
13                    0.114 * image(x, y, 0, 2);
14    }
15    return res;
16 }

```

(3) getAvgOffset(): 计算出平均位移，包括 x 方向和 y 方向上。

```

28 // 获取平均位移
29 vector<int> getAvgOffset(const vector<POINT_PAIR>& pairs, vector<int>& indices) {
30     int offset_x = 0;
31     int offset_y = 0;
32     int min_x = 1000;
33     int min_y = 1000;
34
35     int size = indices.size();
36     int cnt = 0;
37     for (int i = 0; i < size; i++) {
38
39         int diff_x = pairs[i].a.x - pairs[i].b.x;
40         int diff_y = pairs[i].a.y - pairs[i].b.y;
41         if (diff_x == 0 || diff_y == 0 || abs(diff_y) > 220) {
42             continue;
43         }
44         // 求出位移的和
45         offset_x += diff_x;
46         offset_y += diff_y;
47         // 统计点的数量
48         cnt++;
49
50         // 最小的x
51         if (pairs[i].a.x < min_x) {
52             min_x = pairs[i].a.x;
53         }
54
55         // 最小的y
56         if (pairs[i].a.y < min_y) {
57             min_y = pairs[i].a.y;
58         }
59     }
60
61     // 求平均值
62     offset_x /= cnt;
63     offset_y /= cnt;
64
65     int ans_x = 0;
66     int ans_y = 0;
67     cnt = 0;
68     for (int i = 0; i < size; i++) {
69         int diff_x = pairs[i].a.x - pairs[i].b.x;
70         int diff_y = pairs[i].a.y - pairs[i].b.y;
71         cout << "diff_x " << diff_x << " diff_y " << diff_y << endl;
72
73         if (abs(diff_x - offset_x) < abs(offset_x) / 2 && (abs(diff_y - offset_y) < abs(offset_y) / 2 || abs(offset_y) / 2 < 4)) {
74             ans_x += diff_x;
75             ans_y += diff_y;
76             cnt++;
77         }
78     }
79
80     if (cnt) {
81         ans_x /= cnt;
82         ans_y /= cnt;
83     }
84
85     vector<int> res;
86     res.push_back(ans_x);
87     res.push_back(ans_y);
88     res.push_back(min_x);
89     res.push_back(min_y);
90     return res;
91 }

```

(4) stitching(): 图像拼接的函数，调用其余各个方法。

```

201 // 缝合一系列图片
202 CImg<float> stitching(vector<CImg<float>> & src_imgs) {
203     // 存储特征值等数据
204     vector<map<vector<float>, VLSiftKeypoint>> features(src_imgs.size());
205
206     // 对原素材进行预处理
207     for (int i = 0; i < src_imgs.size(); i++) {
208         // 柱面投影
209         cout << "Cylinder Projection" << endl;
210         src_imgs[i] = CylinderProjection(src_imgs[i]);
211
212         // 转化为灰度图
213         CImg<float> gray = get_gray_image(src_imgs[i]);
214
215         // 对每个灰度图进行特征提取
216         cout << "SIFT: Extract Features" << endl;
217         features[i] = extractFeatures(gray);
218     }
219
220     // 邻居表
221     bool adjacent[20][20] = { false };
222     vector<vector<int>> matching_index(src_imgs.size());
223
224

```

```

225 // 找到每张图片的邻居，确认缝合对象
226 cout << "Find Adjacent images.\n";
227 for (int i = 0; i < src_imgs.size(); i++) {
228     for (int j = i + 1; j < src_imgs.size(); j++) {
229         // 对比两幅图的特征点，求出match的特征点集合
230         vector<POINT_PAIR> pairs = getPointPairsFromFeatures(features[i], features[j]);
231
232         // 如果吻合点数量超过30，则认为两幅图是相邻的
233         if (pairs.size() >= 20) {
234             // 记录相邻关系
235             adjacent[i][j] = true;
236             matching_index[i].push_back(j);
237             cout << "Adjacent: " << i << " and " << j << endl;
238         }
239     }
240 }
241 cout << endl;

243 cout << "Stitching" << endl;
244 int beginIndex = 0;
245
246 // 待拼接队列
247 queue<int> unstitched_idx;
248 unstitched_idx.push(beginIndex);
249
250 // 当前已拼接图片
251 CImg<float> cur_stitched_img = src_imgs[beginIndex];
252
253 while (!unstitched_idx.empty()) {
254     int sourceIndex = unstitched_idx.front();
255     unstitched_idx.pop();
256
257     for (int i = 0; i < matching_index[sourceIndex].size(); i++) {
258         // 与当前图片拼接的图片下标
259         int nextIndex = matching_index[sourceIndex][i];
260
261         if (adjacent[sourceIndex][nextIndex]) {
262             adjacent[sourceIndex][nextIndex] = adjacent[nextIndex][sourceIndex] = false;
263             unstitched_idx.push(nextIndex);
264
265             cout << "get Features.\n";
266             // kd树找最近邻
267             vector<POINT_PAIR> src_to_dst_pairs = getPointPairsFromFeatures(features[sourceIndex], features[nextIndex]);
268             vector<POINT_PAIR> dst_to_src_pairs = getPointPairsFromFeatures(features[nextIndex], features[sourceIndex]);
269             // 找最佳匹配方向
270             if (src_to_dst_pairs.size() > dst_to_src_pairs.size())
271                 ReplacePairs(src_to_dst_pairs, dst_to_src_pairs);
272             else
273                 ReplacePairs(dst_to_src_pairs, src_to_dst_pairs);
274
275             // RANSAC算法
276             cout << "RANSAC" << endl;
277             vector<int> indices = RANSAC(dst_to_src_pairs);
278             // 根据最佳模型，计算平均位移
279             vector<int> offset = getAvgOffset(src_to_dst_pairs, indices);
280
281             cout << "offset_x " << offset[0] << " offset_y" << offset[1] << endl;
282
283             // 使用平均位移信息拼接图片
284             cur_stitched_img = blending(cur_stitched_img, src_imgs[nextIndex], offset[0], offset[1], offset[2], offset[3]);
285             cur_stitched_img.display("mid-process", false);
286         }
287     }
288 }
289 return cur_stitched_img;
290 }
291
292
293

```

使用 SIFT 算法用到了 `vfeat` 库的 SIFT，用法是参考网上博客的，主要作用是对一幅图像提取特征点。SIFT 中使用 128 维的描述子来描述一个特征点，我们关注的是特征点的坐标。

转化为灰度图这个比较简单，不再赘述。计算平均位移的输入是两个参数，一个是两幅图之间的匹配特征点对，另一个是根据模型计算出来的匹配特征点 `vector` 的下标。通过 RANSAC 后，我们剔除了那些局外点，剩下的特征点都是能够较好匹配的，因此对这些特征点计算平均位移。也就是说，将两幅图匹配特征点之间的平均位移，作为两幅图拼接时的平均位移，这样在拼接的时候就能得到一个比较好的结果。

拼接函数是顶层函数，它调用了其他的函数，按照题目给出的思路逐步调用，最后拼接出结果。

2. 然后实现比较简单的 `warping`，原理中也提到，为了获得更好的图像拼接结果，需要对素材进行预处理。这个预处理就是柱面投影，它在生成全景图时有重要作用。投影的本质是计算出新的坐标，数学原理是立体几何的相似三角形，网上的博客有推导公式，参考一下就很容易写出。

```

5 // 对图像进行柱面投影预处理
6 CImg<float> CylinderProjection(CImg<float>& img) {
7     CImg<float> result;
8     result.fill(0.0f);
9
10    int width = img._width;
11    int height = img._height;
12    int depth = img._depth;
13    result.assign(width, height, depth, 3);
14
15    float centerX = width / 2;
16    float centerY = height / 2;
17    float f = width / (2 * tan(PI / 4 / 2));
18
19    cimg_forXY(img, i, j) {
20        // 计算曲面投影后的新坐标
21        float theta = asin((i - centerX) / f);
22        int pointX = (f * tan((i - centerX) / f) + centerX);
23        int pointY = ((j - centerY) / cos(theta) + centerY);
24
25        for (int k = 0; k < depth; k++) {
26            if (pointX >= 0 && pointX < width && pointY >= 0 && pointY < height) {
27                result(i, j, k, 0) = img(pointX, pointY, k, 0);
28                result(i, j, k, 1) = img(pointX, pointY, k, 1);
29                result(i, j, k, 2) = img(pointX, pointY, k, 2);
30            }
31            else {
32                result(i, j, k, 0) = 0;
33                result(i, j, k, 1) = 0;
34                result(i, j, k, 2) = 0;
35            }
36        }
37    }
38    return result;
39 }

```

3. 接着是确认拼接顺序，这个是很关键的一步。对于给出的一系列素材，可能是乱序的，因此我们需要确定那些图片之间是相似的（相似则相邻），这里就利用到了前面提取出来的特征点，通过对比特征点的匹配程度，来判断两幅图是否相似。这一步是特征对齐（Features Align），因此单独分出模块。匹配特征点简单来说就是，在 A 图的特征点中找一个点，然后在 B 图中进行搜索，找到一个特征比较相似的点。我们可以按照这个思路进行穷搜，但是看到博客中有大神介绍使用 KD 树来进行搜索，于是自己就尝试了一下。KD 树的思想是基于数据库的存储，以一个特征点为根建树，通过递归方式来构建 KD 树，最后找到与根节点最近的两个节点，使用达朗贝尔判别法来确定是否匹配。这种方法的效率更好，判断也更加科学。找到这些匹配的特征点后，如果其数量大于 30 个，则我们有充分的理由说明两幅图之间有很多相似的地方，即他们是相邻的。

```

15 // 从两幅图像的特征点集合中找出匹配的点对
16 vector<POINT_PAIR> getPointPairsFromFeatures(map<vector<float>, VLSiftKeypoint>& feature_a,
17 map<vector<float>, VLSiftKeypoint>& feature_b) {
18     // 使用KD树来寻找匹配点集
19     VKDForest* forest = vl_kdforest_new(VL_TYPE_FLOAT, 128, 1, VLDistanceL1);
20
21     // 提取出128维的特征向量
22     float* data = new float[128 * feature_a.size()];
23     int k = 0;
24     for (auto it = feature_a.begin(); it != feature_a.end(); it++) {
25         const vector<float>& descriptors = it->first;
26
27         for (int i = 0; i < 128; i++) {
28             data[i + 128 * k] = descriptors[i];
29         }
30         k++;
31     }
32
33     // 构建kd树
34     vl_kdforest_build(forest, feature_a.size(), data);
35
36     vector<POINT_PAIR> result;
37     // 构建一个searcher
38     VKDForestSearcher* searcher = vl_kdforest_new_searcher(forest);
39     VKDForestNeighbor neighbours[2];
40
41     for (auto it = feature_b.begin(); it != feature_b.end(); it++) {
42         float *temp_data = new float[128];
43
44         for (int i = 0; i < 128; i++) {
45             temp_data[i] = (it->first)[i];
46         }
47
48         // 找最近的两个近邻的距离
49         int nvisited = vl_kdforestsearcher_query(searcher, neighbours, 2, temp_data);

```



```

50
51 // 两个邻居的距离比值 (达朗贝尔判别法)
52 float ratio = neighbours[0].distance / neighbours[1].distance;
53 if (ratio < 0.6) {
54     vector<float> des(128);
55     for (int j = 0; j < 128; j++) {
56         des[j] = data[j + neighbours[0].index * 128];
57     }
58
59     VLSiftKeypoint left = feature_a.find(des)->second;
60     VLSiftKeypoint right = it->second;
61     result.push_back(POINT_PAIR(left, right));
62 }
63
64 delete[] temp_data;
65 temp_data = NULL;
66 }
67 vl_kdforestsearcher_delete(searcher);
68 vl_kdforest_delete(forest);
69
70 delete[] data;
71 data = NULL;
72
73 return result;
74 }

```

4. 然后我们就按照上述确定的相邻关系来进行拼接。这里看了博客，有人建议对拼接方向做一个优化，即确定是从 A 匹配到 B 还是从 B 匹配到 A。这里方法与上一步是一样的，算一下哪个方向的匹配特征点数目多就是了。然后自己写了一个辅助函数用来交换两个点集，只用交换每对 POINT\_PAIR 中的一对特征点。

5. 然后就是重头戏 RANSAC 了。因为前面使用的特征点只是简单的匹配，即使用最近邻的方法来找的。特征点集中间会存在一些离群值，原因是两幅图之间可能有多个地方相似，匹配的时候可能会找到一些实际上不是匹配的点对。使用 RANSAC 的最大作用就是去除这些“局外点”，而使用的模型就是透视变换模型，选取四个点对（共 8 个点），确认一块图像区域。具体过程在原理中已经叙述，实现过程不算很难，在计算模型的时候，其实我们是求解一个透视变换的矩阵，网上搜一下就会有一些求解的算法，引入 Eigen 库可以支持矩阵运算。

```

168 // RANSAC算法
169 vector<int> RANSAC(const vector<POINT_PAIR>& pairs) {
170     if (pairs.size() < NUM_OF_PAIR) {
171         cout << "Not enough pairs.\n";
172         exit(1);
173     }
174
175     // 初始化随机数种子
176     srand(time(0));
177     int iterations = numberOfIteration(CONFIDENCE, INLINER_RATIO, NUM_OF_PAIR);
178
179     // 最佳模型的下标
180     vector<int> max_inliner_indices;
181
182     // 抽样迭代
183     while (iterations--) {
184         vector<POINT_PAIR> random_point_pairs; // 随机点集
185         // 每次选择的集合
186         set<int> selected_indices;
187
188         // 1. 随机选取四个点
189         for (int i = 0; i < NUM_OF_PAIR; i++) {
190             int idx = random(0, pairs.size() - 1);
191             // All pairs are unique
192             while (selected_indices.find(idx) != selected_indices.end()) {
193                 idx = random(0, pairs.size() - 1);
194             }
195             selected_indices.insert(idx); // 存入已选择下集合
196
197             random_point_pairs.push_back(pairs[idx]); // 将该点放入随机点集
198         }
199
200         // 2. 求解变换模型
201         MatrixXf H = getHomographyFromPointPairs(random_point_pairs);

```

```

202
203 // 3. 接受局内点, 拒绝局外点,  $SSD(\pi', H \pi) < \epsilon$ 
204 vector<int> cur_inliner_indices = getIndicesOfInlier(pairs, H, selected_indices);
205
206 // 4. 保留投票数最多的模型 (含局内点最多的H)
207 if (cur_inliner_indices.size() > max_inliner_indices.size())
208     max_inliner_indices = cur_inliner_indices;
209 }
210 // 递归结束
211 // 5. 用所有假设的局内点去重新估计模型 (使用最小二乘法)
212 // MatrixXf H = leastSquareSolution(pairs, max_inliner_indices);
213
214 return max_inliner_indices;
215 }

```

```

77 vector<int> getIndicesOfInlier(const vector<POINT_PAIR>& pairs, MatrixXf& H, set<int>& selected_indices) {
78     vector<int> inliner_indices;
79
80     for (int i = 0; i < pairs.size(); i++) {
81         // 跳过已选择的点
82         if (selected_indices.find(i) != selected_indices.end())
83             continue;
84
85         float real_x = pairs[i].b.x;
86         float real_y = pairs[i].b.y;
87
88         if (real_x == 0 || real_y == 0)
89             continue;
90
91         // 根据透视变换矩阵得到点坐标, 验证模型
92         float x = getXAfterWarping(pairs[i].a.x, pairs[i].a.y, H);
93         float y = getYAfterWarping(pairs[i].a.x, pairs[i].a.y, H);
94
95         // 小于一定距离的视作局内点
96         float dist = sqrt((x - real_x) * (x - real_x) + (y - real_y) * (y - real_y));
97         if (dist < RANSAC_THRESHOLD) {
98             inliner_indices.push_back(i);
99         }
100     }
101     return inliner_indices;
102 }

```

```

77 vector<int> getIndicesOfInlier(const vector<POINT_PAIR>& pairs, MatrixXf& H, set<int>& selected_indices) {
78     vector<int> inliner_indices;
79
80     for (int i = 0; i < pairs.size(); i++) {
81         // 跳过已选择的点
82         if (selected_indices.find(i) != selected_indices.end())
83             continue;
84
85         float real_x = pairs[i].b.x;
86         float real_y = pairs[i].b.y;
87
88         if (real_x == 0 || real_y == 0)
89             continue;
90
91         // 根据透视变换矩阵得到点坐标, 验证模型
92         float x = getXAfterWarping(pairs[i].a.x, pairs[i].a.y, H);
93         float y = getYAfterWarping(pairs[i].a.x, pairs[i].a.y, H);
94
95         // 小于一定距离的视作局内点
96         float dist = sqrt((x - real_x) * (x - real_x) + (y - real_y) * (y - real_y));
97         if (dist < RANSAC_THRESHOLD) {
98             inliner_indices.push_back(i);
99         }
100     }
101     return inliner_indices;
102 }

```

6. 去除掉离群值后, 就是最后的拼接了。因为拼接的算法比较复杂, 而时间有限, 这次作业只做到一个比较简单的直接拼接, 得到一个尚可接受的结果。由匹配的特征点得到平均位移, 按照平均位移将一幅图拼接在另一幅图的左端或右端即可。判断左端还是右端的方法是通过判断位移的正负值来实现, 其余的就是逐个像素点移位操作了。



```

3 // 将两张图按照偏移拼接在一起
4 CImg<float> blending(const CImg<float>& A, const CImg<float>& B, int offset_x, int offset_y, int min_x, int min_y) {
5 // 在当前图的左边拼接
6 if (offset_x > 0) {
7 // 新图片宽度和高度 (B较大)
8 int nwidth = B._width + abs(offset_x);
9 int nheight = B._height + abs(offset_y);
10
11 CImg<float> result(nwidth, nheight, 1, B.spectrum(), 0);
12
13 // 以A为基础构建新图
14 cimg_forXY(A, i, j) {
15 // 减少不必要的赋值 (大于min_x的都是B)
16 if (i > min_x)
17 continue;
18 // 三个色道都要赋值
19 for (int k = 0; k < A.spectrum(); k++)
20 result(i, j, 0, k) = A(i, j, 0, k);
21 }
22
23 // 按照偏移量将B拼接 (A左, B右)
24 cimg_forXY(B, x, y) {
25 // 三个色道都要赋值
26 for (int k = 0; k < B.spectrum(); k++)
27 result(x + offset_x, y + offset_y, 0, k) = B(x, y, 0, k);
28 }
29
30 return result;
31 }
32 // 在当前图的右边拼接 (B左, A右)
33 else {
34 // 新图片宽度和高度 (A较大)
35 int nwidth = A._width + abs(offset_x);
36 int nheight = A._height + abs(offset_y);
37
38 CImg<float> result(nwidth, nheight, 1, A.spectrum(), 0);
39
40 // 以B为基础构建新图
41 cimg_forXY(B, i, j) {
42 // 三个色道都要赋值
43 for (int k = 0; k < B.spectrum(); k++)
44 result(i, j, 0, k) = B(i, j, 0, k);
45 }
46 // 按偏移量将A拼接
47 cimg_forXY(A, i, j) {
48 // 小于min_x的都是B
49 if (i < min_x)
50 continue;
51 // 三个色道都要赋值
52 for (int k = 0; k < A.spectrum(); k++) {
53 result(i - offset_x, j - offset_y, 0, k) = A(i, j, 0, k);
54 }
55 }
56 return result;
57 }
58 }

```

## 四、测试及分析

测试说明：本次实验测试用了四个数据集，其中两个是老师给的，两个是自己拍的。自己拍的照片因为不是水平垂直的，但拼接算法是平移的，所以效果会不是很好，但是基本能做到拼接。

**DataSet1:**

测试结果：



**DataSet2:**  
测试结果:



**DataSet3:**  
测试结果:



**DataSet4:**  
测试结果:



## 五、难点、思考、总结与体会

难点：首先这次作业的内容是非常多的，包含了很多个步骤，每个步骤都有自己的算法和原理，整个图像拼接是将几个步骤的结果结合在一起而实现的，所以一开始的时候无从入手，感觉很困难。后来自己看了老师给的 PPT 和一些论文，对照着算法和概念上网查找了很多资料，对这个过程有了一定的了解后才开始 coding。因为对一些算法概念不是很熟悉，coding 过程中还是要不断地找回资料来看，我认为这个也是学习的一种方式，尤其是初学者，还是要不断地看资料，自己理解以后才能学到东西。

第二个问题就是一些阈值的设置问题。因为在匹配特征点、RANSAC 算法中都牵涉到了使用阈值来判断，这个值到底设多少呢？一开始设置过大，导致图片根本拼接不起来，后来将特征点之间的差值都输出了一下，大概算了下平均，才找到了一个比较好的阈值。

第三个问题是 KD 树的使用。KD 树是在特征点匹配的时候使用的，用于寻找最近邻的时候使用的。本来我用的是穷搜，但是网上看到了许多人都推荐使用这个，便自己学习了一下。虽然耗费了不少时间，因为 KD 的使用也是用到其他库文件，但幸运的是网上也有不少详细的教程可以参考。自己花了一点时间去理解了 KD 树的构造等，这个虽然不是这次作业的重点，但也算是以外收获了。稍微总结一下 KD 树，在平面上用二维坐标表示点，将一个点作为根节点，其邻居作为子节点，首先是确定划分轴，对长的轴进行切分，然后是选出中位数，在该位置进行递归划分，生成一颗子树，这种方法充分利用了二叉树的性质，大大地提



高了搜索效率。

第四个问题虽然不是算法的问题，但也很关键。这次实验用到了两个 C++ 第三方库 **Vlfeat** 和 **Eigen**，通过配置 **VS** 来使用。过程有点麻烦，但其实还是要设置好编译路径、包含库路径等变量。

改进：这次作业由于时间关系，做的还不算很完美，主要在下面几个方面。一是图像拼接部分。这里用到的拼接只是两幅图像之间的直接位移拼接，而更好的做法应该是使用插值、矩阵变换等方式，比如双线性插值等，或者使用 **multiband** 这些算法来实现，这样就可以消除图片之间的明显分界，使得结果更加平滑好看。二是在特征提取的时候，可以使用 **SURF** 算法，据说是比 **SIFT** 更快。

总结：这次作业算是 **CV** 这个课程最复杂的一次了，其中包含了很多步骤，每一个步骤都有自己的数学原理与算法，必须每一步都弄清楚做什么才能把图片拼接起来。整体来说很有挑战性，在比较短时间内做得出一个相对来说还过得去的结果，自己也是比较满意。我希望放假有时间的时候可以对这个项目再做优化，争取能够实现比较完美的拼接。

## 六、参考资料

1. RANSAC 算法: <https://www.cnblogs.com/xrwang/archive/2011/03/09/ransac-1.html>
2. 四顶点校正透视变换的线性方程求解:  
<https://www.cnblogs.com/faith0217/articles/5027490.html>
3. KD 树详解及 KD 树最近邻算法:  
[https://blog.csdn.net/app\\_12062011/article/details/51986805](https://blog.csdn.net/app_12062011/article/details/51986805)
4. ratioTest 剔除低质量匹配点: <https://blog.csdn.net/panda1234lee/article/details/11094483>
5. SIFT 检测特征点之生成 128 维描述子:  
<https://blog.csdn.net/wd1603926823/article/details/46564857?readlog>
6. SIFT 之生成描述子: <https://blog.csdn.net/sulanqing/article/details/16371345>
7. VLFeat 中 SIFT 特征点检测: <http://www.cnblogs.com/pakfahome/p/3605285.html>
8. 图像的柱面投影算法: <https://blog.csdn.net/weixinhum/article/details/50611750>
9. 柱面投影简析: <https://www.cnblogs.com/cheermyang/p/5431170.html>
10. KD 树 API 官方文档: <http://www.vlfeat.org/overview/kdtree.html>
11. 特征点匹配——使用基础矩阵、单应性矩阵的 RANSAC 算法去除误匹配点对:  
<https://blog.csdn.net/lhanchao/article/details/52849446>