计算机视觉实验报告

16340220 王培钰 电子政务

编译运行：

运行环境：

`Ubuntu 18`

链接库：

`VLFeat` && `CImg`

编译命令：

`g++ -o test main.cpp ImageStitching.cpp ImageStitching.h -lpthread -l X11 -Lglnxa64/ -lvl -O3 -std=c++11`

(linux下编译比较简单，只需将CImg.h文件和用到的vl库放在同目录下，但是要将libvl.so添加到usr下的动态链接库内)

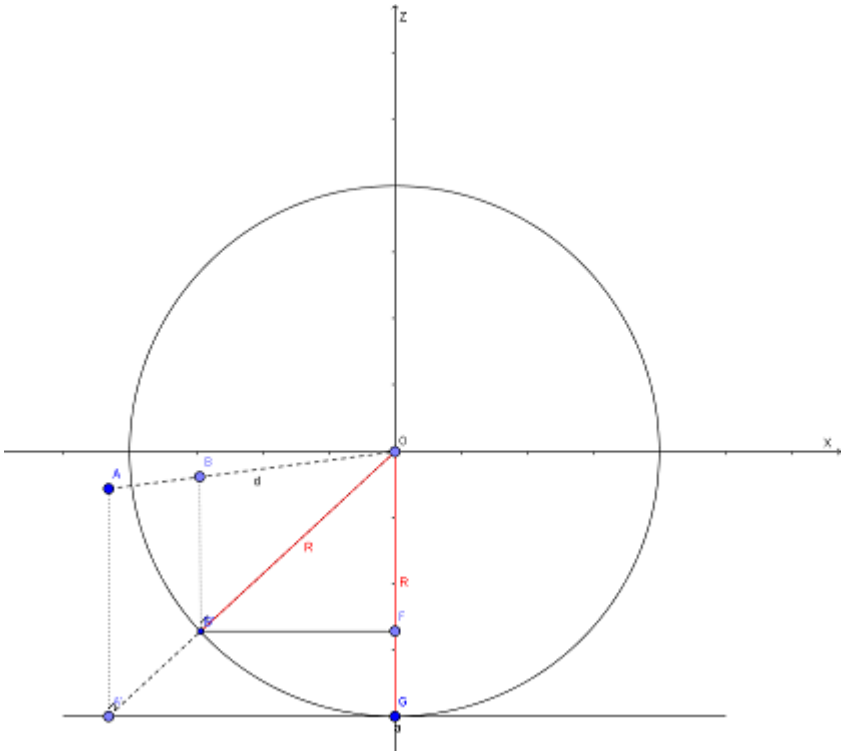
读入图像：

将目录下的多张图片读入，运用了linux环境下提取目录文件的一个目录指针，然后图片存在了CImgList中

```
struct dirent *ptr;
    DIR *dir;
    dir = opendir("TEST-ImageData1");
    while ((ptr = readdir(dir)) != NULL) {
        if(ptr->d_name[0] == '.')
                    continue;
        string file = string("TEST-ImageData1/") + string(ptr->d_name);
        cout << file << endl;
        const char *Ff = file.c_str();
        CImg<float> picture;
        picture.load(Ff);
        imgs.push_back(picture);
    }
```

柱面投影：

在全景图像中，由于摄像头的朝向不同，重合部分部分图像不一定满足视觉一致性的要求，因此先对图像进行投影，使其满足要求，以为后面的拼接做准备。在环形全景中，一般选择柱面投影算法，将图像分别投影到以 像素焦距+摄像头与圆心距离 为半径的圆柱上。投影后的图像为上图摄像头前方的圆弧。从圆弧上看，图像的重合部分已经满足视觉一致性的要求 。



圆柱面坐标转换的公式为：

$$x = \frac{x' - \frac{W}{2}}{k} + \frac{W}{2} = \frac{(x' - \frac{W}{2})\sqrt{R^2 + (x - \frac{W}{2})^2}}{R} + \frac{W}{2}$$

$$y = \frac{y' - \frac{H}{2}}{k} + \frac{H}{2} = \frac{(y' - \frac{H}{2})\sqrt{R^2 + (x - \frac{W}{2})^2}}{R} + \frac{H}{2}$$

可以注意到把x和y写在了等式的左边而x',y'写在了右边，这样做是为了方便进行插值计算 。投影后的图像点坐标未必为整数，而图像的坐标需要为整数，所以必将造成误差 。为了减少这种误差，我们对坐标点进行双线性插值。

其中半径R的值: **R = W / (2*tan(α/2))**，α的取值一般为相机视场角度。

```
CImg<float> ImageStitching::CylindricalProjection(CImg<float> pic) {
    int width = pic._width;
    int height = pic._height;
    int channel = pic._spectrum;
    CImg<float> result(width, height, 1, channel, 0);
    float  R = width / (2*tan(28.0f/2.0f*PI/180.0f));
    for (int x = 0; x < width; x++) {
```

```
        for (int y = 0; y < height; y++) {
            float x0 = x - (float)(width / 2);
            float y0 = y - (float) (height/2);
            float _x = x0*sqrt(R*R + x0*x0) / R + (float) width / 2;
            float _y = y0*sqrt(R*R + x0*x0) / R + (float) height / 2;
            if (_x >= 0 && _x < width && _y >=0 && _y <= height) {
                for (int c = 0;  c < channel; c++) {
                    result(x,y,c) = Interpolation(pic,_x,_y,c);
                }
            }
        }
    }
    return result;
}
```

将图像由RGB空间转为灰度空间：

这一步是为了下面的sift算法做铺垫：

```
CImg<float> ImageStitching::convertTogray(CImg<float> pic) {
    CImg<float> picture(pic._width, pic._height, 1,1,0);
    cimg_forXY(pic,x,y) {
        float R = pic(x,y,0);
        float G = pic(x,y,1);
        float B = pic(x,y,2);
        picture(x,y) =  R*0.299 + G*0.587 + B*0.114;
    }
    return picture;
}
```

SIFT算法查找特征点

SIFT算法的实质是在不同尺度空间上查找关键点(特征点)，并计算关键点方向。SIFT所查找的关键点是一些十分突出，不会因光照，仿射变换和噪音等因素而变化的点，例如角点、边缘点、暗区的亮点以及亮区的暗点。

算法主要分为以下四步：

1. **尺度空间极值检测**：搜索所有尺度上的图像位置。通过高斯微分函数来识别潜在的对于尺度和旋转不变的兴趣点。
2. **关键点定位**：在每个候选位置上，通过一个拟合精细的模型来确定位置和尺度。关键点的选择依据它们的稳定程度。
3. **方向确定**：基于图像局部梯度方向，分配给每个关键点位置一个或多个方向。所有后面的对图像数据的操作都相对于关键点的方向，尺度和位置进行变换。

4. **关键点描述**：在每个关键点周围的邻域内，在选定的尺度上测量图像局部的梯度。这些梯度被变换成一种表示，这种表示允许比较大的局部形状的变形和光照变化。

```
map<vector<float>, VlSiftKeypoint> ImageStitching::SIFTFeatures(CImg<float> pic) {
    int noctaves = 4, nlevels = 2, o_min = 0;
    vl_sift_pix *ImageData = new vl_sift_pix[pic._width*pic._height];
    for (int i = 0; i < pic._width; i++) {
        for (int j = 0; j < pic._height; j++) {
            ImageData[j*pic._width + i] = pic(i,j,0);
        }
    }
    // 定义VlSiftFilt结构体指针
    VlSiftFilt *SiftFilt = NULL;
    // 创建一个新的sift滤波器
    SiftFilt = vl_sift_new(pic._width, pic._height, noctaves, nlevels, o_min);
    //int KeyPoint = 0;
    map<vector<float>, VlSiftKeypoint> Feature;
    if (vl_sift_process_first_octave(SiftFilt, ImageData) != VL_ERR_EOF) {
        while(true) {
            //计算每组中的关键点
            vl_sift_detect(SiftFilt);
            //遍历并绘制每个点
            //KeyPoint += SiftFilt->nkeys;//检测到的关键点的数目
            VlSiftKeypoint *pKeyPoint = SiftFilt->keys;//检测到的关键点
            for (int i = 0; i<SiftFilt->nkeys; i++) {
                VlSiftKeypoint tempKeyPoint = *pKeyPoint;
                pKeyPoint++;
                double angles[4];
                int angleCount = vl_sift_calc_keypoint_orientations(SiftFilt, angles,
&tempKeyPoint);//计算关键点的方向
                for (int j = 0; j<angleCount; j++) {
                    vector<float> Descriptor;
                    double tempAngle = angles[j];
                    vl_sift_pix descriptors[128];
                    // 计算每个方向的描述
                    vl_sift_calc_keypoint_descriptor(SiftFilt, descriptors, &tempKeyPoint,
tempAngle);
                    for (int k = 0; k < 128 ;k++) {
                        Descriptor.push_back(descriptors[k]);
                    }
                    Feature.insert(pair<vector<float>, VlSiftKeypoint>(Descriptor,
tempKeyPoint));
                }

            }
            //下一阶
            if (vl_sift_process_next_octave(SiftFilt) == VL_ERR_EOF)
            {
                break;
            }
            //free(pKeyPoint);
            //KeyPoint = NULL;
        }
```

```
    }
    vl_sift_delete(SiftFilt);
    delete[]ImageData;
    ImageData = NULL;
    return Feature;
}
```

K-d Tree最近邻搜索进行特征点匹配：

k-d树算法可以分为两大部分，一部分是有关k-d树本身这种数据结构建立的算法，另一部分是在建立的k-d树上如何进行最邻近查找的算法。

构建k-d树：

构造kd树的方法：首先构造根节点，根节点对应于整个k维空间，包含所有的实例点，（至于如何选取划分点，有不同的策略。最常用的是一种方法是：对于所有的样本点，统计它们在每个维上的方差，挑选出方差中的最大值，对应的维就是要进行数据切分的维度。数据方差最大表明沿该维度数据点分散得比较开，这个方向上进行数据分割可以获得最好的分辨率；然后再将所有样本点按切分维度的值进行排序，位于正中间的那个数据点选为分裂结点。）。然后利用递归的方法，分别构造k-d树根节点的左右子树。在超矩形区域上选择一个坐标轴（切分维度）和一个分裂结点，以通过此分裂结点且垂直于切分方向坐标轴的直线作为分隔线，将当前超矩形区域分隔成左右或者上下两个子超矩形区域，对应于分裂结点的左右子树的根节点。实例也就被分到两个不相交的区域中。重复此过程直到子区域内没有实例点时终止。终止时的结点为叶结点。

```
算法：createKDTree 构建一棵k-d tree

输入：exm_set 样本集

输出 ： Kd，类型为kd-tree

1. 如果exm_set是空的，则返回空的kd-tree

2.调用分裂结点选择程序（输入是exm_set），返回两个值

       dom_elt:= exm_set中的一个样本点

       split := 分裂维的序号

3.exm_set_left = {exm∈exm_set – dom_elt && exm[split] <= dom_elt[split]}

   exm_set_right = {exm∈exm_set – dom_elt && exm[split] > dom_elt[split]}

4.left = createKDTree(exm_set_left)

right = createKDTree(exm_set_right)
```

k-d tree最近邻搜索算法

在k-d tree树中进行数据的k近邻搜索是特征匹配的重要环节，其目的是检索在k-d tree中与待查询点距离最近的k个数据点。

最近邻搜索是k近邻的特例，也就是1近邻。将1近邻改扩展到k近邻非常容易。下面介绍最简单的k-d tree最近邻搜索算法。

基本的思路很简单：首先通过二叉树搜索（比较待查询节点和分裂节点的分裂维的值，小于等于就进入左子树分支，等于就进入右子树分支直到叶子结点），顺着“搜索路径”很快能找到最近邻的近似点，也就是与待查询点处于同一个子空间的叶子结点；然后再回溯搜索路径，并判断搜索路径上的结点的其他子结点空间中是否可能有距离查询点更近的数据点，如果有可能，则需要跳到其他子结点空间中去搜索（将其他子结点加入到搜索路径）。重复这个过程直到搜索路径为空。下面是k-d tree最近邻搜索的伪代码：

```
算法：kdtreeFindNearest /* k-d tree的最近邻搜索 */

输入：Kd /* k-d tree类型*/
target /* 待查询数据点 */

输出 ： nearest /* 最近邻数据结点 */
dist /* 最近邻和查询点的距离 */

1. 如果Kd是空的，则设dist为无穷大返回
2. 向下搜索直到叶子结点
pSearch = &Kd
while(pSearch != NULL)
{
pSearch加入到search_path中;
if(target[pSearch->split] <= pSearch->dom_elt[pSearch->split]) /* 如果小于就进入左子树 */
{
pSearch = pSearch->left;
}
else
{
pSearch = pSearch->right;
}
}
取出search_path最后一个赋给nearest
dist = Distance(nearest, target);
3. 回溯搜索路径
while(search_path不为空)
{
取出search_path最后一个结点赋给pBack
if(pBack->left为空 && pBack->right为空) /* 如果pBack为叶子结点 */
{
if( Distance(nearest, target) > Distance(pBack->dom_elt, target) )
{
nearest = pBack->dom_elt;
dist = Distance(pBack->dom_elt, target);
}
}
else
{
s = pBack->split;
```

```
if( abs(pBack->dom_elt[s] - target[s]) < dist) /* 如果以target为中心的圆（球或超球），半径为dist的圆与分割超平面相交， 那么就要跳到另一边的子空间去搜索 */
{
if( Distance(nearest, target) > Distance(pBack->dom_elt, target) )
{
nearest = pBack->dom_elt;
dist = Distance(pBack->dom_elt, target);
}
if(target[s] <= pBack->dom_elt[s]) /* 如果target位于pBack的左子空间，那么就要跳到右子空间去搜索 */
pSearch = pBack->right;
else
pSearch = pBack->left; /* 如果target位于pBack的右子空间，那么就要跳到左子空间去搜索 */
if(pSearch != NULL)
pSearch加入到search_path中
}
}
}
```

利用VLFeat实现k-d tree算法的代码：

```
vector<point_pair> ImageStitching::KDtreeMatch(map<vector<float>, VlSiftKeypoint> feature_a,
map<vector<float>, VlSiftKeypoint> feature_b) {
    vector<point_pair> result;

    VlKDForest* forest = vl_kdforest_new(VL_TYPE_FLOAT, 128, 1, VlDistanceL1);

    float *data = new float[128 * feature_a.size()];
    int k = 0;
    for (auto it = feature_a.begin(); it != feature_a.end(); it++) {
        const vector<float> &descriptors = it->first;
        for (int i = 0; i < 128; i++) {
            data[i + 128 * k] = descriptors[i];
        }
        k++;
    }

    vl_kdforest_build(forest, feature_a.size(), data);
    VlKDForestSearcher* searcher = vl_kdforest_new_searcher(forest);
    VlKDForestNeighbor neighbours[2];

    for (auto it = feature_b.begin(); it != feature_b.end(); it++){
        float *temp_data = new float[128];

        for (int i = 0; i < 128; i++) {
            temp_data[i] = (it->first)[i];
        }

        int nvisited = vl_kdforestsearcher_query(searcher, neighbours, 2, temp_data);

        float ratio = neighbours[0].distance / neighbours[1].distance;
        if (ratio < 0.5) {
```

```
            vector<float> des(128);
            for (int j = 0; j < 128; j++) {
                des[j] = data[j + neighbours[0].index * 128];
            }

            VlSiftKeypoint left = feature_a.find(des)->second;
            VlSiftKeypoint right = it->second;
            result.push_back(point_pair(left, right));
        }

        delete[] temp_data;
        temp_data = NULL;
    }

    vl_kdforestsearcher_delete(searcher);
    vl_kdforest_delete(forest);

    delete[] data;
    data = NULL;

    return result;
}
```

RANSAC筛选出特征匹配点：

RANSAC算法在SIFT特征筛选中的主要流程是：

(1) 从样本集中随机抽选一个RANSAC样本，即4个匹配点对

(2) 根据这4个匹配点对计算变换矩阵M

(3) 根据样本集，变换矩阵M，和误差度量函数计算满足当前变换矩阵的一致集consensus，并返回一致集中元素个数

(4) 根据当前一致集中元素个数判断是否最优(最大)一致集，若是则更新当前最优一致集

(5) 更新当前错误概率p，若p大于允许的最小错误概率则重复(1)至(4)继续迭代，直到当前错误概率p小于最小错误概率

$$k = \frac{\log(1-p)}{\log(1-w^m)}$$

其中，p为置信度，一般取0.995；w为"内点"的比例；m为计算模型所需要的最少样本数=4；

```
points ImageStitching::RANSAC(vector<point_pair> pairs) {
    srand(time(0));
    int iterations = ceil(log(1 - 0.99) / log(1 - pow(0.5, 4)));
    vector<int> max_inliner_indexs;
```

```
    while (iterations--) {
        vector<point_pair> random_pairs;
        set<int> seleted_indexs;

        for (int i = 0; i < 4; i++) {
            int index = rand() % pairs.size();
            while (seleted_indexs.find(index) != seleted_indexs.end()) {
                index = rand() % pairs.size();
            }
            seleted_indexs.insert(index);

            random_pairs.push_back(pairs[index]);
        }

        points H = HomographyMatrix(random_pairs);
        //cout << H.x1 << endl;
        vector<int> cur_inliner_indexs;
        for (int i = 0; i < pairs.size(); i++) {
            if (seleted_indexs.find(i) != seleted_indexs.end()) {
                continue;
            }

            float real_x = pairs[i].b.x;
            float real_y = pairs[i].b.y;

            float x = H.x1 * pairs[i].a.x + H.x2 * pairs[i].a.y + H.x3 * pairs[i].a.x *
pairs[i].a.y + H.x4;
            float y = H.x5 * pairs[i].a.x + H.x6 * pairs[i].a.y + H.x7 * pairs[i].a.x *
pairs[i].a.y + H.x8;

            float distance = sqrt((x - real_x) * (x - real_x) + (y - real_y) * (y - real_y));
            if (distance < 4) {
                cur_inliner_indexs.push_back(i);
            }
        }
        if (cur_inliner_indexs.size() > max_inliner_indexs.size()) {
            max_inliner_indexs = cur_inliner_indexs;
        }
    }
    int calc_size = max_inliner_indexs.size();

    CImg<double> A(4, calc_size, 1, 1, 0);
    CImg<double> b(1, calc_size, 1, 1, 0);

    for (int i = 0; i < calc_size; i++) {
        int cur_index = max_inliner_indexs[i];

        A(0, i) = pairs[cur_index].a.x;
        A(1, i) = pairs[cur_index].a.y;
        A(2, i) = pairs[cur_index].a.x * pairs[cur_index].a.y;
        A(3, i) = 1;

        b(0, i) = pairs[cur_index].b.x;
```

```
    }

    CImg<double> x1 = b.get_solve(A);

    for (int i = 0; i < calc_size; i++) {
        int cur_index = max_inliner_indexs[i];

        b(0, i) = pairs[cur_index].b.y;
    }

    CImg<double> x2 = b.get_solve(A);
    return points(x1(0, 0), x1(0, 1), x1(0, 2), x1(0, 3), x2(0, 0), x2(0, 1), x2(0, 2), x2(0,
3));
}
```

图像拼接：

先是进行进行拷贝，将一副图像按照需要拼接的点拷贝到另一幅图像上。但如果是直接拼接的话可能两幅图像对接点显得很不自然。这里使用Cimg的滤波函数进行边缘的滤波平滑。

```
CImg<float> ImageStitching::Blend(CImg<float> pic1, CImg<float> pic2) {
    double sum_a_x = 0;
    double sum_a_y = 0;
    int a_n = 0;
    double sum_overlap_x = 0;
    double sum_overlap_y = 0;
    int overlap_n = 0;
    if (pic1.width() > pic1.height()) {
        for (int x = 0; x < pic1.width(); x++) {
            if (!IsBlack(pic1, x, pic1.height() / 2)) {
                sum_a_x += x;
                a_n++;
            }
            if (!IsBlack(pic1, x, pic1.height() / 2) && !IsBlack(pic2, x, pic1.height() / 2)) {
                sum_overlap_x += x;
                overlap_n++;
            }
        }
    }
    else {
        for (int y = 0; y < pic1.height(); y++) {
            if (!IsBlack(pic1, pic1.width() / 2, y)) {
                sum_a_y += y;
                a_n++;
            }

            if (!IsBlack(pic1, pic1.width() / 2, y) && !IsBlack(pic2, pic2.width() / 2, y)) {
                sum_overlap_y += y;
```

```
                overlap_n++;
            }
        }
    }
    int min_len = (pic1.width() < pic1.height()) ? pic1.width() : pic1.height();
    int n_level = floor(log2(min_len));
    vector<CImg<float> > a_pyramid(n_level);
    vector<CImg<float> > b_pyramid(n_level);
    vector<CImg<float> > mask(n_level);
    // Initialize the base.
    a_pyramid[0] = pic1;
    b_pyramid[0] = pic2;
    mask[0] = CImg<float>(pic1.width(), pic1.height(), 1, 1, 0);
    if (pic1.width() > pic1.height()) {
        if (sum_a_x / a_n < sum_overlap_x / overlap_n) {
            for (int x = 0; x < sum_overlap_x / overlap_n; x++) {
                for (int y = 0; y < pic1.height(); y++) {
                    mask[0](x, y) = 1;
                }
            }
        }
        else {
            for (int x = sum_overlap_x / overlap_n + 1; x < pic1.width(); x++) {
                for (int y = 0; y < pic1.height(); y++) {
                    mask[0](x, y) = 1;
                }
            }
        }
    }
    else {
        if (sum_a_y / a_n < sum_overlap_y / overlap_n) {
            for (int x = 0; x < pic1.width(); x++) {
                for (int y = 0; y < sum_overlap_y / overlap_n; y++) {
                    mask[0](x, y) = 1;
                }
            }
        }
        else {
            for (int x = 0; x < pic1.width(); x++) {
                for (int y = sum_overlap_y / overlap_n; y < pic1.height(); y++) {
                    mask[0](x, y) = 1;
                }
            }
        }
    }
    // Down sampling a and b, building Gaussian pyramids.
    for (int i = 1; i < n_level; i++) {
        a_pyramid[i] = a_pyramid[i - 1].get_blur(2).get_resize(a_pyramid[i - 1].width() / 2,
a_pyramid[i - 1].height() / 2, 1, a_pyramid[i - 1].spectrum(), 3);
        b_pyramid[i] = b_pyramid[i - 1].get_blur(2).get_resize(b_pyramid[i - 1].width() / 2,
b_pyramid[i - 1].height() / 2, 1, b_pyramid[i - 1].spectrum(), 3);
```

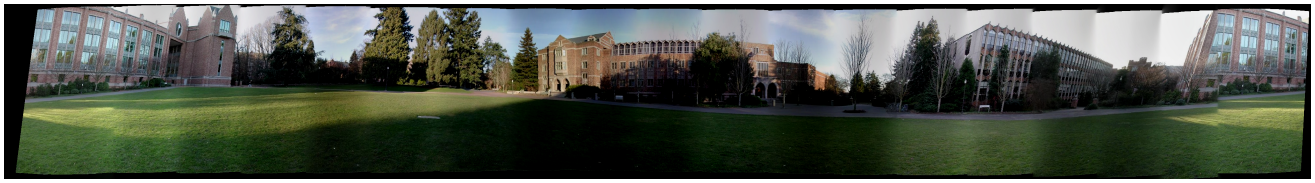
```
        mask[i] = mask[i - 1].get_blur(2).get_resize(mask[i - 1].width() / 2, mask[i -
1].height() / 2, 1, mask[i - 1].spectrum(), 3);
    }
    // Building Laplacian pyramids.
    for (int i = 0; i < n_level - 1; i++) {
        a_pyramid[i] = a_pyramid[i] - a_pyramid[i + 1].get_resize(a_pyramid[i].width(),
a_pyramid[i].height(), 1, a_pyramid[i].spectrum(), 3);
        b_pyramid[i] = b_pyramid[i] - b_pyramid[i + 1].get_resize(b_pyramid[i].width(),
b_pyramid[i].height(), 1, b_pyramid[i].spectrum(), 3);
    }

    vector<CImg<float> > blend_pyramid(n_level);
    for (int i = 0; i < n_level; i++) {
        blend_pyramid[i] = CImg<float>(a_pyramid[i].width(), a_pyramid[i].height(), 1,
a_pyramid[i].spectrum(), 0);
        cimg_forXYC(blend_pyramid[i], x, y, c) {
            blend_pyramid[i](x, y, c) = a_pyramid[i](x, y, c) * mask[i](x, y) + b_pyramid[i](x,
y, c) * (1.0 - mask[i](x, y));
        }
    }

    CImg<float> res = blend_pyramid[n_level - 1];
    for (int i = n_level - 2; i >= 0; i--) {
        res.resize(blend_pyramid[i].width(), blend_pyramid[i].height(), 1,
blend_pyramid[i].spectrum(), 3);
        cimg_forXYC(blend_pyramid[i], x, y, c) {
            res(x, y, c) = blend_pyramid[i](x, y, c) + res(x, y, c);
            if (res(x, y, c) > 255) res(x, y, c) = 255;
            else if (res(x, y, c) < 0) res(x, y, c) = 0;
        }
    }

    return res;
}
```

最后封装成类：

```
class ImageStitching {

private:
    //源图列表
    CImgList<float> imgs;
    //拼接后的图像
    CImg<float> resultImg;

private:
    //柱面投影
    CImg<float> CylindricalProjection(CImg<float> pic);
```

```
    //图像从rgb空间转为灰度空间
    CImg<float> convertTogray(CImg<float> pic);
    //提取图像中的特征点
    map<vector<float>, VlSiftKeypoint> SIFTFeatures(CImg<float> pic);
    //k-d树进行特征点匹配
    vector<point_pair> KDtreeMatch(map<vector<float>, VlSiftKeypoint> feature_a,
map<vector<float>, VlSiftKeypoint> feature_b);
    //利用RANSAC算法求单应矩阵
    points RANSAC(vector<point_pair> pairs);
    //通过单应矩阵扭曲两幅图像内容
    void WarpTwoImg(CImg<float> src, CImg<float> &dst, points H, float offset_x, float
offset_y);
    //移动两幅图像
    void MoveTwoImg(CImg<float> src, CImg<float> &dst, int offset_x, int offset_y);
    //图像拼接
    CImg<float> Blend(CImg<float> pic1, CImg<float> pic2);

public:
    ImageStitching();
    ~ImageStitching();
    void StitchProcess();
};
```

结果拼接出的图像：



实验总结：

程序复杂度过高，第二组图跑了半个小时，不过也跟自己虚拟机分配的内存和处理器不足有关。代码而言一方面是SIFT算法本身复杂度就比较高，网上有提到SIFT的简化版SURF的复杂度会低很多，其次自己在边缘处的一些插值算法以及图像拼接处的平滑等也大大提高了算法复杂度。有时间之后会思考去改进一下。

再就是最大的缺点感觉柱面投影时图片边缘存在许多毛刺，这个本来以为用了插值算法会大大减少这种像素点，结果还是看起来比较明显。同时图像接合处的用滤波的效果感觉不如开始使用opencv库尝试的时候溶解效果好，到时会去读一下opencv中溶解的源码了解一下这类的算法。