实验名称：全景图像拼接

实验要求：

1. 将多张图片合并拼接成一张全景图
2. 可以使用 CImg 库与 vlfeat 库。

实验步骤：

1. 对图像进行 cylindrical warping 操作
2. 提取 SIFT 特征点，匹配特征
3. 使用 RANSAC 来校准邻居间的特征点对
4. 拼接处理后的图像

实验过程：

首先，因为有多张图像需要读取，如果一张一张读取就比较不方便，这里使用了一个叫 dirent.h 的头文件，使用这个头文件需要提供文件夹的名字（存放测试图像的文件夹），然后调用它的功能函数：

```
DIR *dir;
struct dirent *ent;
if ((dir = opendir(folder)) != NULL) {
    while((ent = readdir(dir)) != NULL) {
        if(ent->d_type == DT_REG){
            files.push_back(ent->d_name);
        }
    }
    closedir(dir);
} else {
    /* could not open directory */
    printf("Cannot open directory %s\n", folder);
    exit(EXIT_FAILURE);
}
```

这里只有文件的 d_type 为 DT_REG（普通文件类型）时才是我们想要的。而由上述代码可见，这个头文件实际是在读取文件夹中文件的名称，这样就可以一次获取所有图像了。

1. 对图像进行 cylindrical warping 操作

这里实际上要求为进行 spherical warping 操作，但因相关资料较少，而拥有近似效果的 cylindrical warping 的资料反而不少，所以这里选择了进行 cylindrical warping。

所谓 warping 也就是图像坐标系间的变换，它需要两个独立的算法，一个算法来定义空间变换本身，用它来描述每个像素如何从其输入图像位置移动到输出图像位置；在这里进行 cylindrical warping 的算法为：

$$x' = f * tan(\frac{x - x_c}{f}) + x_c$$

$$y' = \frac{y - y_c}{cos(\frac{x - x_c}{f})} + y_c$$

这里的（x，y）是原图像上的坐标，(x_c, y_c) 为原图像中心处的坐标，(x', y') 为投射到 cylinder 坐标系上的坐标，f 为焦距。得到 cylinder 坐标系上的坐标后，还要再进行一次 inverse warping 得到对应在原图像上的坐标。

在得到坐标对应关系后，需要为相应坐标进行赋值。这里就用到了第二个算法：bilinear interpolation，因为得到的坐标为了精确使用 float 格式表示的，所以很大可能不是准确的在任何一个像素上，这时就需要利用双线性插值来给其赋值。

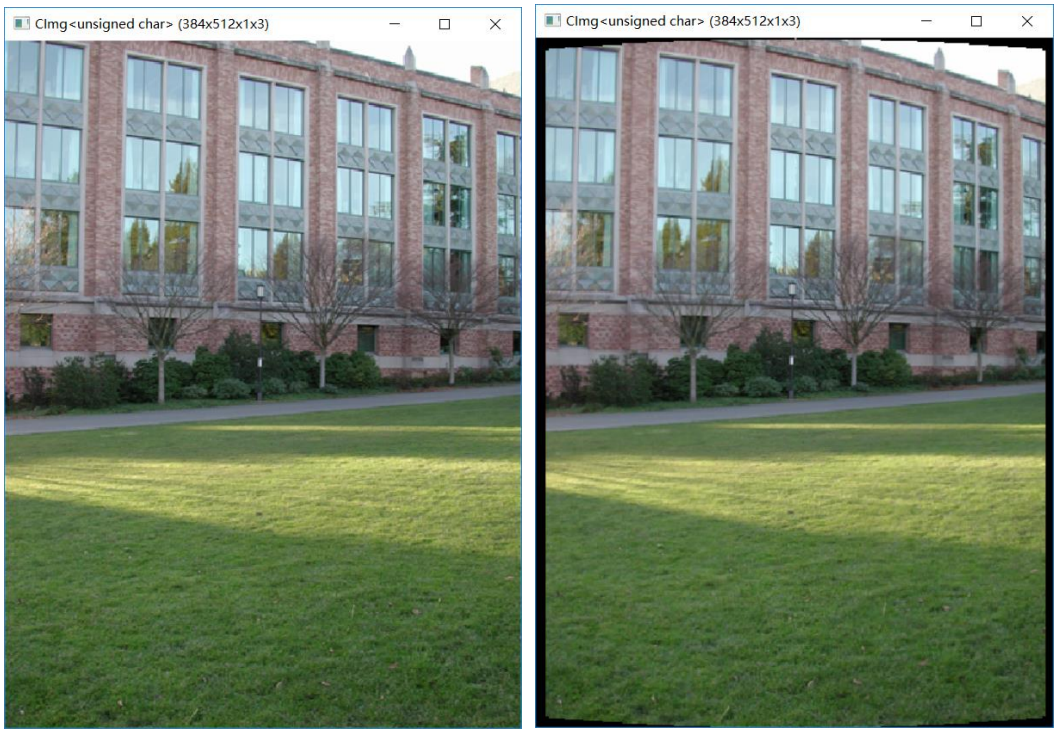
双线性插值的实现：

```
    int x_pos = floor(x);
    float x_u = x - x_pos;
    int xb = (x_pos < image.width() - 1) ? x_pos + 1 : x_pos;

    int y_pos = floor(y);
     float y_v = y - y_pos;
    int yb = (y_pos < image.height() - 1) ? y_pos + 1 : y_pos;

     float P1 = image(x_pos, y_pos, channel) * (1 - x_u) + image(xb,
y_pos, channel) * x_u;
     float P2 = image(x_pos, yb, channel) * (1 - x_u) + image(xb, yb,
channel) * x_u;
    //均衡取四周的值
    return P1 * (1 - y_v) + P2 * y_v;
```

cylindrical warping 的结果：



可以看到右边相比左边，明显有投射为圆柱型的效果。

2. 提取 SIFT 特征点，匹配特征

SIFT 是一种用来侦测与描述影像中的局部性特征的计算机视觉算法，它在空间尺度中寻找极值点，并提取出其位置、尺度、旋转不变量。更进一步的说，它就是用不同尺度（标准差）的高斯函数对图像进行平滑，然后比较平滑后图像的差别。

为了完成 SIFT 算法，首先要建立图像尺度空间（高斯金字塔）并检测极值点，这一步中采用高斯函数来建立尺度空间，其公式为：

$$G(x,y,\sigma)=\frac{1}{2\pi\sigma^2}e^{-(x^2+y^2)/2\sigma^2}.$$

原始影像 I(x, y)在不同的尺度下与高斯函数进行卷积，就可得到尺度空间：

$$L(x,y,\sigma)=G(x,y,\sigma)*I(x,y)$$

组建一组尺度空间后，对上一组尺度空间的最后一幅图像进行二分之一采样来得到下一组尺度空间的第一幅图像，再继续组建尺度空间，得到图像金字塔。

但这一步因为需要遍历所有像素进行卷积而代价很大，可以使用高斯差分尺度空间（DOG）进行优化，即利用不同尺度的高斯差分与原始图像相乘：

$$\begin{aligned}D(x,y,\sigma) &= (G(x,y,k\sigma)-G(x,y,\sigma))*I(x,y)\\ &= L(x,y,k\sigma)-L(x,y,\sigma).\end{aligned}$$

DOG 算子是尺度归一化的 LOG 算子的近似。

为了寻找尺度空间的极值点，需要每个采样点与它所有的相邻点进行比较，这些相邻点是与它同尺度的 8 个相邻点和它上下相邻尺度的 9 个点。

为了使描述符有旋转不变性，可利用图像的局部特征给每个关键点分配方向（梯度方向），这样就得到了关键点的位置、所处尺度、方向三个信息，再为关键点取 16 个种子点，每个种子点有 8 个方向向量信息，对于一个关键点就可以产生 128 维的 SIFT 特征向量，这样的特征向量可以去除尺度变化、旋转等几何变形因素的影响。

当两幅图像的 SIFT 特征向量生成后，就可以采用关键点特征向量的欧式距离来作为两幅图像关键点的相似性判定依据。

但在这次的实际实现中，可以使用 vlfeat 库提供的 sift 算法，实现起来便利许多。主要用到了 VlSiftFIlt 来作为 filter，在利用 vl_sift_new 创建新 VlSiftFilt 实例时，需要提供金字塔阶层、层次等参数。如：

```
    int octave = 4, level = 2, min = 0;
    VlSiftFilt *siftFilt = NULL;
    siftFilt = vl_sift_new(width, height, octave, level, min);
```

之后一个一个的对 octave 进行处理，对每个关键点，提取其对应的 128 维 SIFT 特征向量：

```
double angles[4];
int  angleCount  =  vl_sift_calc_keypoint_orientations(siftFilt,  angles,
&tempKeyp);
for(int j = 0; j < angleCount; ++j){
    double tempAngle = angles[j];
    vl_sift_pix descriptors[128];
    vl_sift_calc_keypoint_descriptor(siftFilt,  descriptors,  &tempKeyp,
tempAngle);
    ……
```

最终使用一个元素类型为 vector<float>（保存向量）和 VlSiftKeypoint 的 map 来保存提取出的 feature。

提取出的关键点：

之后再使用 vlfeat 提供的 kd-tree 算法来对两幅图找它们对应的关键点对，它是利用了特征空间中两个实例点的距离来反映两个实例点的相似程度，再根据相似程度来判断是否为相对应的关键点。使用 vlfeat 提供的 vl_kdforest_build、vl_kdforest_new_searcher、vl_kdforestsearcher_query 等函数可以很方便的建树、找到邻居并比较距离。

得到关键点对后需要进行阈值筛除，这里将阈值设置为 20，即只有相似关键点对大于等于 20 的两幅图才视为匹配的图像。

3. 使用 RANSAC 来校准邻居间的特征点对

RANSAC 算法的输入是一组观测数据（往往有较多的噪声或无效点），这里就是先前找到的关键点对。使用一个模型来测试数据，如果某个点适用于估计的模型，那么认为它也是内点。如果有足够多的点被归类为假设的内点，那么估计的模型就足够合理。然后使用所有的内点取重新估计模型（如使用最小二乘），最后重新评估模型。经过多次迭代后得到最优模型，进而获取满足需求的内点（inliers）。

Ransac 算法中有一些参数需要指定，如每个点为真正内群的机率 w、算法跑 k 次后成功的机率 p，及所选的 n 个点。

这里将 w 设为 0.5，p 设为 0.99，n 设为 4（关键点对）。

在每次迭代运算中，随机选取点对，使用 homography 矩阵运算来计算点对的值所对应的值，在对这些值，获取其中满足内点条件的那些点对，筛除不满足的那些，最后再进行最小二乘处理，返回最终满足内点条件的那些关键点对。

Homography 矩阵运算实现：

```
Homoparam Matching::getHomography(vector<matching_points> &pairs){
    CImg<float> A(4, NUM_OF_PAIR, 1, 1, 0);
    CImg<float> b(1, NUM_OF_PAIR, 1, 1, 0);
    //calc Ax = b
    for(int i = 0; i < NUM_OF_PAIR; ++i){
        A(0, i) = pairs[i].src.x;
        A(1, i) = pairs[i].src.y;
        A(2, i) = pairs[i].src.x * pairs[i].src.y;
        A(3, i) = 1;
        b(0, i) = pairs[i].dst.x;
    }
    CImg<float> x1 = b.get_solve(A);
    for(int i = 0; i < NUM_OF_PAIR; ++i){
        b(0, i) = pairs[i].dst.y;
    }
    CImg<float> x2 = b.get_solve(A);
    return Homoparam(x1(0,0), x1(0,1), x1(0,2), x1(0,3), x2(0,0),
x2(0,1), x2(0,2), x2(0,3));
}
```

这里的 Homoparam 为一个有 8 个参数的结构体。

4. 拼接处理后的图像

这里使用图像金字塔的方法来拼接融合图像。

首先使用高斯金字塔对图像进行下采样，下采样使得图像金字塔的层级越高，图像越小。这里直接使用了 CImg 提供的 get_blur 函数来实现下采样，省去卷积等操作。如：

```
// 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);
}
```

上采样使得图像变大，这里直接用 get_resize 函数：

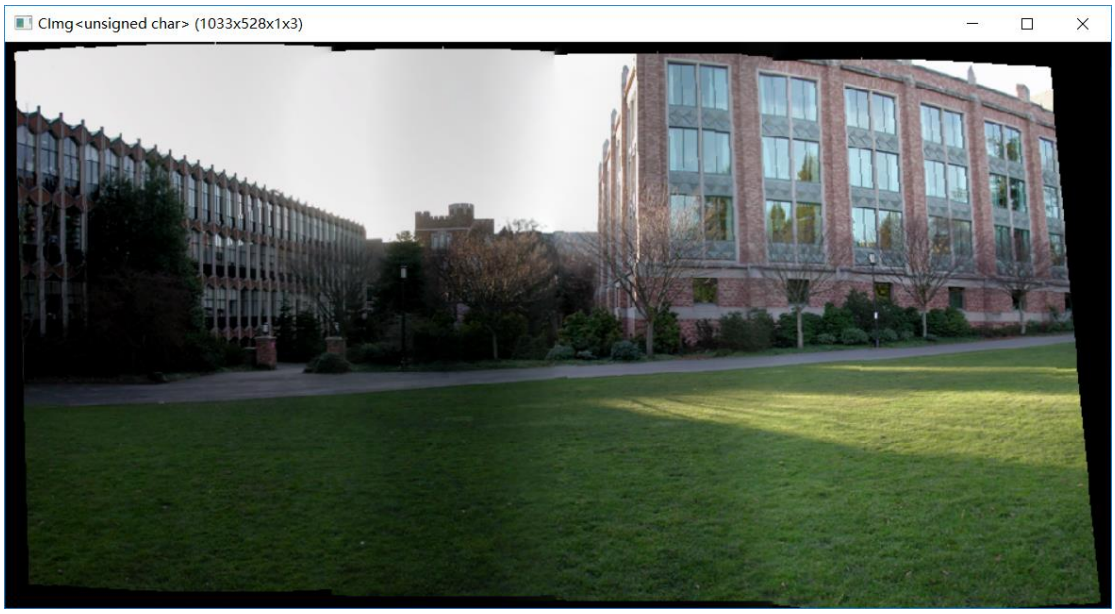
```
// Building Laplacian pyramids.
for (int i = 0; i < n_level - 1; i++) {
    a_pyramid[i] = a_pyramid[i] - a_pyramid[i + 1].get_resize(a_pyramid[i].width(), a_pyramid[i].height(), 1, a_pyramid[i].spectrum(), 3);
    b_pyramid[i] = b_pyramid[i] - b_pyramid[i + 1].get_resize(b_pyramid[i].width(), b_pyramid[i].height(),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

然后利用高斯金字塔与拉普拉斯金字塔融合两幅图像：

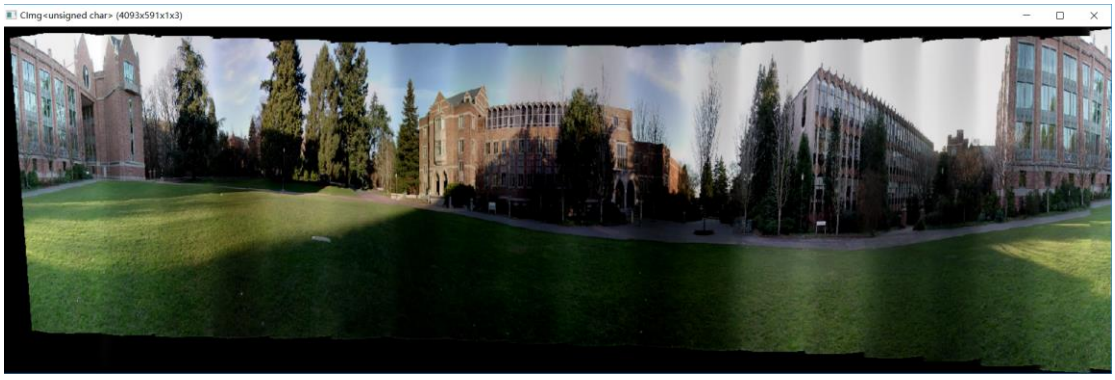
```
for (int i = 0; i < n_level; i++) {
    blend_pyramid[i] = CImg<float>(a_pyramid[i].width(), a_pyramid[i].height(), 1, a_pyramid[i].spectrum(), 0);
    for (int x = 0; x < blend_pyramid[i].width(); x++) {
        for (int y = 0; y < blend_pyramid[i].height(); y++) {
            for (int k = 0; k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

测试结果：

Dataset1：



Dataset2：



自己的测试集 Dataset3：