```
void get_sift_features(
vector<string>& imagematnames,
vector<vector<KeyPoint>>& keypoints_global,
vector<Mat>& descriptor_global,
vector<vector<Vec3b>>& colors_global
)
{
keypoints_global.clear();
descriptor_global.clear();
Mat imagemat;
//读取图像，获取图像特征点集
Ptr<Feature2D> sift = xfeatures2d::SIFT::create();//0, 3, 0.04, 10);
for (auto it = imagematnames.begin(); it != imagematnames.end(); it++)
{
imagemat = imread(*it);
if (imagemat.empty()) continue;
vector<KeyPoint> keypoints;
Mat descriptor;
sift->detectAndCompute(imagemat, noArray(), keypoints, descriptor);
//特征点过少则排除该图像
if (keypoints.size() <= 20) continue;
keypoints_global.push_back(keypoints);
descriptor_global.push_back(descriptor);
vector<Vec3b> colors(keypoints.size());
for (int i = 0; i < keypoints.size(); ++i)
{
PoinC2f& p = keypoints[i].pt;
colors[i] = imagemat.at<Vec3b>(p.y, p.x);
}
colors_global.push_back(colors);
}
}
void match_sift_features(Mat& query, Mat& train, vector<DMatch>& matches)
{
vector<vector<DMatch>> knn_matches;
BFMatcher matcher(NORM_L2);
matcher.knnMatch(query, train, knn_matches, 2);
//获取满足RatioTest的最小匹配的距离
float min_dist = FLT_MAX;
for (int r = 0; r < knn_matches.size(); ++r)
{
//RatioTest达朗贝尔判别
if (knn_matches[r][0].distance > 0.6*knn_matches[r][1].distance)
continue;
float dist = knn_matches[r][0].distance;
if (dist < min_dist) min_dist = dist;
}
matches.clear();
for (size_t r = 0; r < knn_matches.size(); ++r)
{
//排除不满足RatioTest的点和匹配距离显然过大的点
if (
knn_matches[r][0].distance > 0.6*knn_matches[r][1].distance ||
knn_matches[r][0].distance > 5 * max(min_dist, 10.0f)
)
continue;
```

```
//保存匹配点
matches.push_back(knn_matches[r][0]);
}
}
void match_sift_features_all(vector<Mat>& descriptor_global,
vector<vector<DMatch>>& matches_global)
{
matches_global.clear();
// n个图像，两两顺次有n-1对匹配
// 1-2匹配，2-3匹配，3-4匹配，以此类推
for (int i = 0; i < descriptor_global.size() - 1; ++i)
{
cout << “Matching imagemats ” << i << ” - ” << i + 1 << endl;
vector<DMatch> matches;
match_sift_features(descriptor_global[i], descriptor_global[i + 1], matches);
matches_global.push_back(matches);
}
}
bool find_RandT(Mat& K, vector<PoinC2f>& m1, vector<PoinC2f>& m2, Mat& R, Mat&
T, Mat& mask)
{
//根据内参矩阵获取相机的焦距和相机主点坐标
double f = 0.5*(K.at<double>(0) + K.at<double>(4));
PoinC2d pxpy(K.at<double>(2), K.at<double>(5));
//根据匹配点求取本征矩阵，使用RANSAC，进一步消除失配点
Mat E = findEssentialMat(m1, m2, f, pxpy, RANSAC, 0.999, 1.0, mask);
if (E.empty()) return false;
double feasible_count = countNonZero(mask);
cout << (int)feasible_count << ” -in- ” << m1.size() << endl;
//对于RANSAC而言，Outlier数量大于50%时，结果是不可靠的
if (feasible_count <= 15 || (feasible_count / m1.size()) < 0.6)
return false;
//分解本征矩阵，获取相对变换
int pass_count = recoverPose(E, m1, m2, R, T, f, pxpy, mask);
//同时位于两个相机前方的点的数量要足够大
if (((double)pass_count) / feasible_count < 0.8)
return false;
return true;
}
void get_matched_points(
vector<KeyPoint>& m1,
vector<KeyPoint>& m2,
vector<DMatch> matches,
vector<PoinC2f>& out_m1,
vector<PoinC2f>& out_m2
)
{
out_m1.clear();
out_m2.clear();
for (int i = 0; i < matches.size(); ++i)
{
out_m1.push_back(m1[matches[i].queryIdx].pt);
out_m2.push_back(m2[matches[i].trainIdx].pt);
}
}
void get_matched_colors(
```

```
vector<Vec3b>& color1,
vector<Vec3b>& color2,
vector<DMatch> matches,
vector<Vec3b>& out_color1,
vector<Vec3b>& out_color2
)
{
out_color1.clear();
out_color2.clear();
for (int i = 0; i < matches.size(); ++i)
{
out_color1.push_back(color1[matches[i].queryIdx]);
out_color2.push_back(color2[matches[i].trainIdx]);
}
}
void reconstruct(Mat& K, Mat& R1, Mat& C1, Mat& R2, Mat& C2, vector<PoinC2f>&
m1, vector<PoinC2f>& m2, vector<Point3f>& Mi)
{
//两个相机的投影矩阵[R T]，triangulatePoints必须是float型
Mat P1(3, 4, CV_32FC1);
Mat P2(3, 4, CV_32FC1);
R1.convertTo(P1(Range(0, 3), Range(0, 3)), CV_32FC1);
C1.convertTo(P1.col(3), CV_32FC1);
R2.convertTo(P2(Range(0, 3), Range(0, 3)), CV_32FC1);
C2.convertTo(P2.col(3), CV_32FC1);
Mat fK;
K.convertTo(fK, CV_32FC1);
P1 = fK*P1;
P2 = fK*P2;
//三角化重建
Mat s;
triangulatePoints(P1, P2, m1, m2, s);
Mi.clear();
Mi.reserve(s.cols);
for (int i = 0; i < s.cols; ++i)
{
Mat_<float> col = s.col(i);
col /= col(3); //齐次坐标，需要除以最后一个元素才是真正的坐标值
Mi.push_back(Point3f(col(0), col(1), col(2)));
}
}
void maskout_points(vector<PoinC2f>& m1, Mat& mask)
{
vector<PoinC2f> m1_copy = m1;
m1.clear();
for (int i = 0; i < mask.rows; ++i)
{
if (mask.at<uchar>(i) > 0)
m1.push_back(m1_copy[i]);
}
}
void maskout_colors(vector<Vec3b>& m1, Mat& mask)
{
vector<Vec3b> m1_copy = m1;
m1.clear();
for (int i = 0; i < mask.rows; ++i)
```

```
{
if (mask.at<uchar>(i) > 0)
m1.push_back(m1_copy[i]);
}
}
struct ProjectCost
{
cv::PoinC2d observation;
ProjectCost(cv::PoinC2d& observation)
: observation(observation)
{
}
template <typename T>
bool operator()(const T* const intrinsic, const T* const extrinsic, const T*
const pos3d, T* residuals) const
{
const T* r = extrinsic;
const T* t = &extrinsic[3];
T pos_proj[3];
ceres::AngleAxisRotatePoint(r, pos3d, pos_proj);
// 应用相机平移
pos_proj[0] += t[0];
pos_proj[1] += t[1];
pos_proj[2] += t[2];
const T x = pos_proj[0] / pos_proj[2];
const T y = pos_proj[1] / pos_proj[2];
const T fx = intrinsic[0];
const T fy = intrinsic[1];
const T cx = intrinsic[2];
const T cy = intrinsic[3];
// 应用内参
const T u = fx * x + cx;
const T v = fy * y + cy;
residuals[0] = u - T(observation.x);
residuals[1] = v - T(observation.y);
return true;
}
};
void bundle_adjustment(
Mat& intrinsic,
vector<Mat>& extrinsics,
vector<vector<int>>& correspond_structIdx,
vector<vector<KeyPoint>>& keypoints_global,
vector<Point3d>& Mi
)
{
ceres::Problem problem;
// 载入摄像机外部参数
for (size_t i = 0; i < extrinsics.size(); ++i)
{
problem.AddParameterBlock(extrinsics[i].ptr<double>(), 6);
}
// 修正第一个相机
problem.SetParameterBlockConstant(extrinsics[0].ptr<double>());
// 载入摄像机内部参数
problem.AddParameterBlock(intrinsic.ptr<double>(), 4); // fx, fy, cx, cy
```

```
// 载入点
ceres::LossFunction* loss_function = new ceres::HuberLoss(4); // loss function
make bundle adjustment robuster.
for (size_t imgIdx = 0; imgIdx < correspond_structIdx.size(); ++imgIdx)
{
vector<int>& point3d_ids = correspond_structIdx[imgIdx];
vector<KeyPoint>& keypoints = keypoints_global[imgIdx];
for (size_t pointIdx = 0; pointIdx < point3d_ids.size(); ++pointIdx)
{
int point3d_id = point3d_ids[pointIdx];
if (point3d_id < 0)
continue;
PoinC2d observed = keypoints[pointIdx].pt;
// 模板参数中，第一个为代价函数的类型，第二个为代价的维度，剩下三个分别为代价函
数第一第二还有第三个参数的维度
ceres::CostFunction* cost_function = new
ceres::AutoDiffCostFunction<ProjectCost, 2, 4, 6, 3>(new ProjectCost(observed));
problem.AddResidualBlock(
cost_function,
loss_function,
intrinsic.ptr<double>(), // 内参
extrinsics[imgIdx].ptr<double>(), // 旋转平移
&(Mi[point3d_id].x) // 空间物体点
);
}
}
// 求解Bundle Adjustment
ceres::Solver::Options ceres_config_options;
ceres_config_options.minimizer_progress_to_stdout = false;
ceres_config_options.logging_type = ceres::SILENT;
ceres_config_options.num_threads = 1;
ceres_config_options.preconditioner_type = ceres::JACOBI;
ceres_config_options.linear_solver_type = ceres::SPARSE_SCHUR;
ceres_config_options.sparse_linear_algebra_library_type = ceres::EIGEN_SPARSE;
ceres::Solver::Summary summary;
ceres::Solve(ceres_config_options, &problem, &summary);
if (!summary.IsSolutionUsable())
{
std::cout << “Bundle Adjustment failed.” << std::endl;
}
else
{
// 最小化统计
std::cout << std::endl
<< “approximated RMSE:\n”
<< ” #views: ” << extrinsics.size() << “\n”
<< ” #residuals: ” << summary.num_residuals << “\n”
<< ” Initial RMSE: ” << std::sqrt(summary.initial_cost /
summary.num_residuals) << “\n”
<< ” Final RMSE: ” << std::sqrt(summary.final_cost / summary.num_residuals) <<
“\n”
<< ” Time (s): ” << summary.total_time_in_seconds << “\n”
<< std::endl;
}
}
void get_objpoints_and_imgpoints(
```

```
vector<DMatch>& matches,
vector<int>& struct_indices,
vector<Point3f>& Mi,
vector<KeyPoint>& keypoints,
vector<Point3f>& object_points,
vector<PoinC2f>& imagemat_points)
{
object_points.clear();
imagemat_points.clear();
for (int i = 0; i < matches.size(); ++i)
{
int queryIdx = matches[i].queryIdx;
int trainIdx = matches[i].trainIdx;
int structIdx = struct_indices[queryIdx];
if (structIdx < 0) continue;
object_points.push_back(Mi[structIdx]);
imagemat_points.push_back(keypoints[trainIdx].pt);
}
}
void fusion_Mi(
vector<DMatch>& matches,
vector<int>& struct_indices,
vector<int>& next_struct_indices,
vector<Point3f>& Mi,
vector<Point3f>& next_Mi,
vector<Vec3b>& colors,
vector<Vec3b>& next_colors
)
{
for (int i = 0; i < matches.size(); ++i)
{
int queryIdx = matches[i].queryIdx;
int trainIdx = matches[i].trainIdx;
int structIdx = struct_indices[queryIdx];
if (structIdx >= 0) //若该点在空间中已经存在，则这对匹配点对应的空间点应该是同一个，索引要相同
{
next_struct_indices[trainIdx] = structIdx;
continue;
}
//若该点在空间中已经存在，将该点加入到结构中，且这对匹配点的空间点索引都为新加入的点的索引
Mi.push_back(next_Mi[i]);
colors.push_back(next_colors[i]);
struct_indices[queryIdx] = next_struct_indices[trainIdx] = Mi.size() - 1;
}
}
void init_Mi(
Mat K,
vector<vector<KeyPoint>>& keypoints_global,
vector<vector<Vec3b>>& colors_global,
vector<vector<DMatch>>& matches_global,
vector<Point3f>& Mi,
vector<vector<int>>& correspond_structIdx,
vector<Vec3b>& colors,
vector<Mat>& rotations,
```

```
vector<Mat>& motions
)
{
//计算头两幅图像之间的变换矩阵
vector<PoinC2f> m1, m2;
vector<Vec3b> color2;
Mat R, T; //旋转矩阵和平移向量
Mat mask; //mask中大于零的点代表匹配点，等于零代表失配点
get_matched_points(keypoints_global[0], keypoints_global[1], matches_global[0],
m1, m2);
get_matched_colors(colors_global[0], colors_global[1], matches_global[0],
colors, color2);
find_RandT(K, m1, m2, R, T, mask);
//对头两幅图像进行三维重建
maskout_points(m1, mask);
maskout_points(m2, mask);
maskout_colors(colors, mask);
Mat R0 = Mat::eye(3, 3, CV_64FC1);
Mat T0 = Mat::zeros(3, 1, CV_64FC1);
reconstruct(K, R0, T0, R, T, m1, m2, Mi);
//保存变换矩阵
rotations = { R0, R };
motions = { T0, T };
//将correspond_structIdx的大小初始化为与keypoints_global完全一致
correspond_structIdx.clear();
correspond_structIdx.resize(keypoints_global.size());
for (int i = 0; i < keypoints_global.size(); ++i)
{
correspond_structIdx[i].resize(keypoints_global[i].size(), -1);
}
//填写头两幅图像的结构索引
int idx = 0;
vector<DMatch>& matches = matches_global[0];
for (int i = 0; i < matches.size(); ++i)
{
if (mask.at<uchar>(i) == 0)
continue;
correspond_structIdx[0][matches[i].queryIdx] = idx;
correspond_structIdx[1][matches[i].trainIdx] = idx;
++idx;
}
}
void main()
{
//存放图片文件名
vector<string> img_names;
//内参矩阵，由棋盘格用caliberation.exe标定
Mat K(Matx33d(
2944.23, 0, 1728.69,
0, 2944.15, 2304.81,
0, 0, 1));
vector<vector<Vec3b>> colors_global;
vector<vector<DMatch>> matches_global;
vector<vector<KeyPoint>> keypoints_global;
vector<Mat> descriptor_global;
//提取所有图像的特征
```

```
get_sift_features(img_names, keypoints_global, descriptor_global,
colors_global);
//对所有图像进行相邻顺次的特征匹配
match_sift_features_all(descriptor_global, matches_global);
vector<Point3f> Mi;
vector<vector<int>> correspond_structIdx;
//保存第i副图像中第j个特征点对应的Mi中点的索引
vector<Vec3b> colors;
vector<Mat> rotations;
vector<Mat> motions;
//初始化稀疏重建点云结构
init_Mi(
K,
keypoints_global,
colors_global,
matches_global,
Mi,
correspond_structIdx,
colors,
rotations,
motions
);
//增量方式重建剩余的图像
for (int i = 1; i < matches_global.size(); ++i)
{
vector<Point3f> object_points;
vector<PoinC2f> imagemat_points;
Mat r, R, T;
//Mat mask;
//获取第i幅图像中匹配点对应的三维点，以及在第i+1幅图像中对应的像素点
get_objpoints_and_imgpoints(
matches_global[i],
correspond_structIdx[i],
Mi,
keypoints_global[i+1],
object_points,
imagemat_points
);
//求解变换矩阵
solvePnPRansac(object_points, imagemat_points, K, noArray(), r, T);
//将旋转向量转换为旋转矩阵
Rodrigues(r, R);
//保存变换矩阵
rotations.push_back(R);
motions.push_back(T);
vector<PoinC2f> m1, m2;
vector<Vec3b> color1, color2;
get_matched_points(keypoints_global[i], keypoints_global[i + 1],
matches_global[i], m1, m2);
get_matched_colors(colors_global[i], colors_global[i + 1], matches_global[i],
color1, color2);
//根据之前求得的R，T进行三维重建
vector<Point3f> next_Mi;
reconstruct(K, rotations[i], motions[i], R, T, m1, m2, next_Mi);
//将新的重建结果与之前的融合
fusion_Mi(
```

```
matches_global[i],
correspond_structIdx[i],
correspond_structIdx[i + 1],
Mi,
next_Mi,
colors,
color1
);
}
//Bundle Adjustment
Mat intrinsic(Matx41d(K.at<double>(0, 0), K.at<double>(1, 1), K.at<double>(0,
2), K.at<double>(1, 2)));
vector<Mat> extrinsics;
for (size_t i = 0; i < rotations.size(); ++i)
{
Mat extrinsic(6, 1, CV_64FC1);
Mat r;
Rodrigues(rotations[i], r);
r.copyTo(extrinsic.rowRange(0, 3));
motions[i].copyTo(extrinsic.rowRange(3, 6));
extrinsics.push_back(extrinsic);
}
bundle_adjustment(intrinsic, extrinsics, correspond_structIdx, keypoints_global,
Mi);
//保存
save_Mi(…);
//调用PMVS2
PMVS::PMVS(…);
}
```