```cpp
#include <g2o/core/base_vertex.h>
#include <g2o/core/base_binary_edge.h>
#include <g2o/core/block_solver.h>
#include <g2o/core/optimization_algorithm_levenberg.h>
#include <g2o/solvers/csparse/linear_solver_csparse.h>
#include <g2o/core/robust_kernel_impl.h>
#include <iostream>

#include "common.h"
#include "sophus/se3.hpp"
#include "ceres/rotation.h"
#include "glog/logging.h"
using namespace Sophus;
using namespace Eigen;
using namespace std;

/// ¿¿¿¿¿¿¿¿
struct PoseAndIntrinsics {
    PoseAndIntrinsics() {}

    /// set from given data address
    explicit PoseAndIntrinsics(double *data_addr) {
        rotation = SO3d::exp(Vector3d(data_addr[0], data_addr[1], data_addr[2]));
        translation = Vector3d(data_addr[3], data_addr[4], data_addr[5]);
        focal = data_addr[6];
        k1 = data_addr[7];
        k2 = data_addr[8];
    }

    /// ¿¿¿¿¿¿¿¿
    void set_to(double *data_addr) {
        auto r = rotation.log();
        for (int i = 0; i < 3; ++i) data_addr[i] = r[i];
        for (int i = 0; i < 3; ++i) data_addr[i + 3] = translation[i];
        data_addr[6] = focal;
        data_addr[7] = k1;
        data_addr[8] = k2;
    }

    SO3d rotation;
    Vector3d translation = Vector3d::Zero();
    double focal = 0;
    double k1 = 0, k2 = 0;
};

/// ¿¿¿¿¿¿¿¿¿¿9¿¿¿¿¿¿so3¿¿¿¿¿t, f, k1, k2
class VertexPoseAndIntrinsics : public g2o::BaseVertex<9, PoseAndIntrinsics> {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;

    VertexPoseAndIntrinsics() {}

    virtual void setToOriginImpl() override {
        _estimate = PoseAndIntrinsics();
    }

    virtual void oplusImpl(const double *update) override {
        _estimate.rotation = SO3d::exp(Vector3d(update[0], update[1], update[2])) * _
estimate.rotation;
        _estimate.translation += Vector3d(update[3], update[4], update[5]);
        _estimate.focal += update[6];
        _estimate.k1 += update[7];
        _estimate.k2 += update[8];
    }

    /// ¿¿¿¿¿¿¿¿¿¿
    Vector2d project(const Vector3d &point) {
        Vector3d pc = _estimate.rotation * point + _estimate.translation;
        pc = -pc / pc[2];
```

```
69          double r2 = pc.squaredNorm();
70          double distortion = 1.0 + r2 * (_estimate.k1 + _estimate.k2 * r2);
71          return Vector2d(_estimate.focal * distortion * pc[0],
72                          _estimate.focal * distortion * pc[1]);
73      }
74
75      virtual bool read(istream &in) {}
76
77      virtual bool write(ostream &out) const {}
78  };
79
80  class VertexPoint : public g2o::BaseVertex<3, Vector3d> {
81  public:
82      EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
83
84      VertexPoint() {}
85
86      virtual void setToOriginImpl() override {
87          _estimate = Vector3d(0, 0, 0);
88      }
89
90      virtual void oplusImpl(const double *update) override {
91          _estimate += Vector3d(update[0], update[1], update[2]);
92      }
93
94      virtual bool read(istream &in) {}
95
96      virtual bool write(ostream &out) const {}
97  };
98
99  class EdgeProjection :
100     public g2o::BaseBinaryEdge<2, Vector2d, VertexPoseAndIntrinsics, VertexPoint> {
101 public:
102     EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
103
104     virtual void computeError() override {
105         auto v0 = (VertexPoseAndIntrinsics *) _vertices[0];
106         auto v1 = (VertexPoint *) _vertices[1];
107         auto proj = v0->project(v1->estimate());
108         _error = proj - _measurement;
109     }
110
111     // use numeric derivatives
112     virtual bool read(istream &in) {}
113
114     virtual bool write(ostream &out) const {}
115
116 };
117
118 void SolveBA(BALProblem &bal_problem);
119
120 int main(int argc, char **argv) {
121
122     if (argc != 2) {
123         cout << "usage: bundle_adjustment_g2o bal_data.txt" << endl;
124         return 1;
125     }
126
127     BALProblem bal_problem(argv[1]);
128     bal_problem.Normalize();
129     bal_problem.Perturb(0.1, 0.5, 0.5);
130     bal_problem.WriteToPLYFile("initial.ply");
131     SolveBA(bal_problem);
132     bal_problem.WriteToPLYFile("final.ply");
133
134     return 0;
135 }
136
137 void SolveBA(BALProblem &bal_problem) {
```

```
138     const int point_block_size = bal_problem.point_block_size();
139     const int camera_block_size = bal_problem.camera_block_size();
140     double *points = bal_problem.mutable_points();
141     double *cameras = bal_problem.mutable_cameras();
142 
143     // pose dimension 9, landmark is 3
144     typedef g2o::BlockSolver<g2o::BlockSolverTraits<9, 3>> BlockSolverType;
145     typedef g2o::LinearSolverCSparse<BlockSolverType::PoseMatrixType> LinearSolverTyp
    e;
146     // use LM
147     auto solver = new g2o::OptimizationAlgorithmLevenberg(
148         g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
149     g2o::SparseOptimizer optimizer;
150     optimizer.setAlgorithm(solver);
151     optimizer.setVerbose(true);
152 
153     /// build g2o problem
154     const double *observations = bal_problem.observations();
155     // vertex
156     vector<VertexPoseAndIntrinsics *> vertex_pose_intrinsics;
157     vector<VertexPoint *> vertex_points;
158     for (int i = 0; i < bal_problem.num_cameras(); ++i) {
159         VertexPoseAndIntrinsics *v = new VertexPoseAndIntrinsics();
160         double *camera = cameras + camera_block_size * i;
161         v->setId(i);
162         v->setEstimate(PoseAndIntrinsics(camera));
163         optimizer.addVertex(v);
164         vertex_pose_intrinsics.push_back(v);
165     }
166     for (int i = 0; i < bal_problem.num_points(); ++i) {
167         VertexPoint *v = new VertexPoint();
168         double *point = points + point_block_size * i;
169         v->setId(i + bal_problem.num_cameras());
170         v->setEstimate(Vector3d(point[0], point[1], point[2]));
171         // g2o¿BA¿¿¿¿¿¿¿Marg¿¿¿
172         v->setMarginalized(true);
173         optimizer.addVertex(v);
174         vertex_points.push_back(v);
175     }
176 
177     // edge
178     for (int i = 0; i < bal_problem.num_observations(); ++i) {
179         EdgeProjection *edge = new EdgeProjection;
180         edge->setVertex(0, vertex_pose_intrinsics[bal_problem.camera_index()[i]]);
181         edge->setVertex(1, vertex_points[bal_problem.point_index()[i]]);
182         edge->setMeasurement(Vector2d(observations[2 * i + 0], observations[2 * i + 1
    ]));
183         edge->setInformation(Matrix2d::Identity());
184         edge->setRobustKernel(new g2o::RobustKernelHuber());
185         optimizer.addEdge(edge);
186     }
187 
188     optimizer.initializeOptimization();
189     optimizer.optimize(40);
190 
191     // set to bal problem
192     for (int i = 0; i < bal_problem.num_cameras(); ++i) {
193         double *camera = cameras + camera_block_size * i;
194         auto vertex = vertex_pose_intrinsics[i];
195         auto estimate = vertex->estimate();
196         estimate.set_to(camera);
197     }
198     for (int i = 0; i < bal_problem.num_points(); ++i) {
199         double *point = points + point_block_size * i;
200         auto vertex = vertex_points[i];
201         for (int k = 0; k < 3; ++k) point[k] = vertex->estimate()[k];
202     }
203 }
```