第七章作业

作者：曾是少年

二 Bundle Adjustment

2.1 文献阅读(2 分)

我们在第五讲中已经介绍了Bundle Adjustment，指明它可以用于解PnP 问题。现在，我们又在后端中说明了它可以用于解大规模的三维重构问题，但在实时SLAM 场合往往需要控制规模。事实上，Bundle Adjustment 的历史远比我们想象的要长。请阅读Bill Triggs 的经典论文Bundle Adjustment: A Modern Synthesis（见paper/目录）1，了解BA 的发展历史，然后回答下列问题：

1. 为何说Bundle Adjustment is slow 是不对的?

答：原文中如下所示：

"Optimization / bundle adjustment is slow": Such statements often appear in papers introducing yet another heuristic Structure from Motion (SFM) iteration. The claimed slowness is almost always due to the unthinking use of a general-purpose optimization routine that completely ignores the problem structure and sparseness. Real bundle routines are much more efficient than this, and usually considerably more efficient and flexible than the newly suggested method (§6, 7). That is why bundle adjustment remains the dominant structure refinement technique for real applications, after 40 years of research."

翻译："优化/束调整运行缓慢"：这种陈述经常出现在介绍启发式SFM的论文中。他们把BA速度缓慢的原因归咎于通用的优化流程，该例程完全忽略了问题结构的稀疏性。实际的BA例程要高效很多，并且通常比新提出的方法（第6、7节）更加高效灵活。这就是经过40年的研究后，BA依然是实际应用中占主导地位的结构精炼技术。

2. BA 中有哪些需要注意参数化的地方？Pose 和Point 各有哪些参数化方式？有何优缺点。

答：总结一下BA过程首先选择你想要的图里的节点与边的类型，确定它们的参数化形式；
1) 往图里加入实际的节点和边；
2) 选择初值，开始迭代；
3) 每一步迭代中，计算对应于当前估计值的雅可比矩阵和海塞矩阵；
4) 求解稀疏线性方程$H_k\Delta x = -b_k$，得到梯度方向；
5) 继续用GN或LM进行迭代。如果迭代结束，返回优化值。

Point 参数化方式

视觉SLAM中点的参数化表示包括三维坐标XYZ和你深度表示方法。

Open VINS文档中给出了五种特征参数化表示：Global XYZ，Global Inverse Depth，Anchored XYZ，Anchored Inverse Depth，Anchored Inverse Depth (MSCKF Version)，区别在于：

- **Global vs Anchored**：特征点的表示是全局坐标系的坐标还是局部相机坐标系的坐标。
- **XYZ vs Inverse Depth**：使用的XYZ还是逆深度

- **Two different Inverse Depth**：两种不同类型的逆深度参数

三维坐标XYZ：优点在于简单直观；缺点在于不能表示无穷远点；

逆深度：优点在于能够建模无穷远点；在实际应用中，逆深度也具有更好的数值稳定性。

Pose参数化方式

Pose的参数化表示包括欧拉角、四元数、变换矩阵。

1. 欧拉角

   优点：容易理解，形象直观；三个值分别对应x、y、z轴的旋转角度。

   缺点：欧拉角这种方法是要按照一个固定的坐标轴的顺序旋转的，因此不同的顺序会造成不同结果；欧拉角旋转会造成**万向锁现象**，这种现象的发生就是由于上述固定的坐标轴旋转顺序造成的。由于万向锁的存在，欧拉旋转无法实现球面平滑插值。

2. 变换矩阵

   优点：旋转轴可以是任意向量

   缺点：旋转其实只需要知道一个向量+一个角度(共4自由度)，但矩阵却用了16个元素(消耗时间和内存)

3. 四元数

   优点：可以避免万向锁问题；只需要一个4维的四元数就可以执行绕任意过原点的向量的旋转，方便快捷，在某些实现下比旋转矩阵效率更高；而且四元数旋转可以提供平滑插值。

   缺点：比欧拉旋转稍微复杂了一点，因为多了一个维度，理解更困难，不直观。带有约束条件

3. 本文写于2000 年，但是文中提到的很多内容在后面十几年的研究中得到了印证。你能看到哪些方向在后续工作中有所体现？请举例说明。

2.2 BAL-dataset

BAL（Bundle Adjustment in large）数据集是一个大型BA 数据集，它提供了相机与点初始值与观测，你可以用它们进行Bundle Adjustment。现在，请你使用`g2o`，自己定义`Vertex`和`Edge`（不要使用自带的顶点类型，也不要像本书例程那边调用`Ceres`来求导），书写`BAL`上的`BA`程序。你可以挑选其中一个数据，运行你的BA，并给出优化后的点云图。

本题不提供代码框架，请独立完成。

提示：

1. 注意BAL 的投影模型比教材中介绍的多了个负号；

答：（本题我没有思路，因此主要是参考了网络上的代码进行学习验证的）

一般情况下，使用g2o的主要过程如下：

1. 定义顶点和边的类型
2. 构建图
3. 选择优化算法
4. 调用g2o

具体编程过程如下：

1. 打开BAL数据集文件`problem-16-22106-pre.txt`。描述问题

   这一部分调用common.cpp中的BALProblem类来进行

```
//读取BAL数据
string bal_file_name = "problem-16-22106-pre.txt";
//读取BAL数据 构建BAL问题
BALProblem bal_problem(bal_file_name);
//调用Normalize接口对数据进行归一化
bal_problem.Normalize();
//调用Perturb接口对数据添加噪声
bal_problem.Perturb(0.1,0.5,0.5);
```

2. 定义顶点（使用节点来表示相机和路标）

   相机位姿节点

```
//相机位姿节点
//继承的g2o::BaseVertex基类    <优化变量维度,数据类型>
/// 位姿加相机内参的顶点，9维，前三维为so3，接下去为t, f, k1, k2
class VertexPoseAndIntrinsics : public g2o::BaseVertex<9, PoseAndIntrinsics>
{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;

    VertexPoseAndIntrinsics() {}
    //重置
    virtual void setToOriginImpl() override {
        _estimate = PoseAndIntrinsics();
    }
    //更新
    virtual void oplusImpl(const double *update) override {
        _estimate.rotation = SO3d::exp(Vector3d(update[0], update[1],
update[2])) * _estimate.rotation; //李代数的更新方式
        _estimate.translation += Vector3d(update[3], update[4], update[5]);
//平移部分的更新
        _estimate.focal += update[6]; //更新焦距
        _estimate.k1 += update[7]; //更新内参
        _estimate.k2 += update[8]; //更新内参
    }

    /// 根据估计值投影一个点
    Vector2d project(const Vector3d &point) {
        Vector3d pc = _estimate.rotation * point + _estimate.translation;
        pc = -pc / pc[2];
        double r2 = pc.squaredNorm();
        double distortion = 1.0 + r2 * (_estimate.k1 + _estimate.k2 * r2);
        return Vector2d(_estimate.focal * distortion * pc[0],
                        _estimate.focal * distortion * pc[1]);
    }

    //读盘和存盘
    virtual bool read(istream &in) {}

    virtual bool write(ostream &out) const {}
};
```

路标节点

```
class VertexPoint : public g2o::BaseVertex<3, Vector3d> {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;

    VertexPoint() {}

    virtual void setToOriginImpl() override {
        _estimate = Vector3d(0, 0, 0);
    }

    virtual void oplusImpl(const double *update) override {
        _estimate += Vector3d(update[0], update[1], update[2]);
    }

    virtual bool read(istream &in) {}

    virtual bool write(ostream &out) const {}
};
```

3. 投影边

```
// 边的定义
// 观测边
class EdgeProjection : public g2o::BaseBinaryEdge<2, Vector2d,
VertexPoseAndIntrinsics, VertexPoint> {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;

    virtual void computeError() override {
        auto v0 = (VertexPoseAndIntrinsics *) _vertices[0];
        auto v1 = (VertexPoint *) _vertices[1];
        auto proj = v0->project(v1->estimate()); //estimate 可以返回该节点的当
前估计
        _error = proj - _measurement; //计算误差
    }

    // use numeric derivatives
    virtual bool read(istream &in) {}
    virtual bool write(ostream &out) const {}
};
```

4. 选择优化算法（在solve算法中）

```
const int point_block_size = bal_problem.point_block_size();
const int camera_block_size = bal_problem.camera_block_size();
double *points = bal_problem.mutable_points(); //得到point参数的起始地址
double *cameras = bal_problem.mutable_cameras(); //得到camera参数的起始地址

// pose dimension 9, landmark is 3
typedef g2o::BlockSolver<g2o::BlockSolverTraits<9, 3>> BlockSolverType;
```

```
typedef g2o::LinearSolverCSparse<BlockSolverType::PoseMatrixType>
LinearSolverType; //改了一处
// use LM
// 选择使用LM算法进行优化
auto solver = new g2o::OptimizationAlgorithmLevenberg(
    g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>
()));
g2o::SparseOptimizer optimizer;
optimizer.setAlgorithm(solver);
optimizer.setVerbose(true);
```

5. 构建g2o问题

```
    /// build g2o problem
    const double *observations = bal_problem.observations();
    // 添加顶点 Vertex
    vector<VertexPoseAndIntrinsics *> vertex_pose_intrinsics;
    vector<VertexPoint *> vertex_points;
    // 添加相机位姿节点
    for (int i = 0; i < bal_problem.num_cameras(); ++i) {
        VertexPoseAndIntrinsics *v = new VertexPoseAndIntrinsics();
        double *camera = cameras + camera_block_size * i;
        v->setId(i);
        v->setEstimate(PoseAndIntrinsics(camera));
        optimizer.addVertex(v);
        vertex_pose_intrinsics.push_back(v);
    }
    // 添加路标节点
    for (int i = 0; i < bal_problem.num_points(); ++i) {
        VertexPoint *v = new VertexPoint();
        double *point = points + point_block_size * i;
        v->setId(i + bal_problem.num_cameras());
        v->setEstimate(Vector3d(point[0], point[1], point[2]));
        // g2o在BA中需要手动设置待Marg的顶点
        v->setMarginalized(true);
        optimizer.addVertex(v);
        vertex_points.push_back(v);
    }

    // 添加边 edge
    for (int i = 0; i < bal_problem.num_observations(); ++i) {
        EdgeProjection *edge = new EdgeProjection;
        edge->setVertex(0, vertex_pose_intrinsics[bal_problem.camera_index()
[i]]);
        edge->setVertex(1, vertex_points[bal_problem.point_index()[i]]);
        edge->setMeasurement(Vector2d(observations[2 * i + 0],
observations[2 * i + 1]));
        edge->setInformation(Matrix2d::Identity());
        edge->setRobustKernel(new g2o::RobustKernelHuber());
        optimizer.addEdge(edge);
    }
```

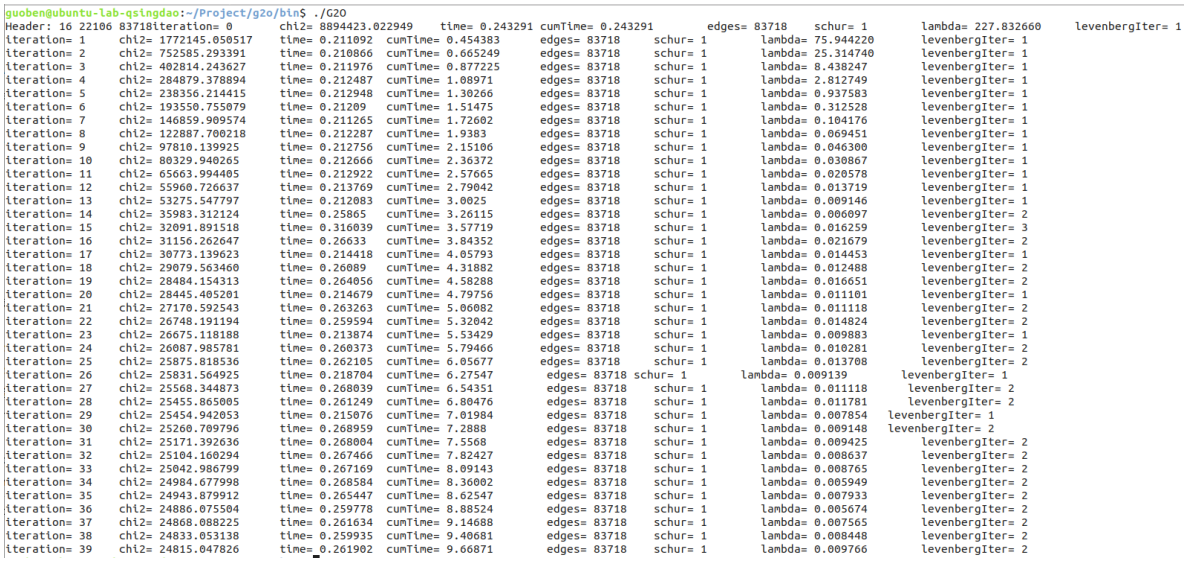
6. 调用g2o

```
optimizer.initializeOptimization();
```

```
optimizer.optimize(40);

// set to bal problem
for (int i = 0; i < bal_problem.num_cameras(); ++i) {
    double *camera = cameras + camera_block_size * i;
    auto vertex = vertex_pose_intrinsics[i];
    auto estimate = vertex->estimate();
    estimate.set_to(camera);
}
for (int i = 0; i < bal_problem.num_points(); ++i) {
    double *point = points + point_block_size * i;
    auto vertex = vertex_points[i];
    for (int k = 0; k < 3; ++k) point[k] = vertex->estimate()[k];
}
```

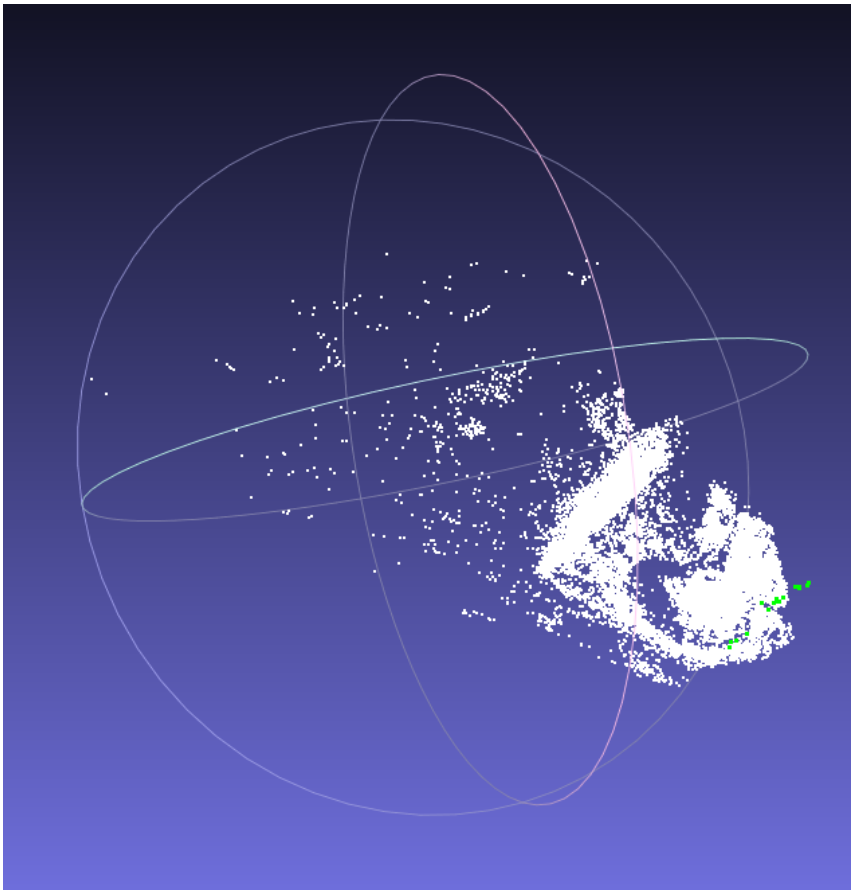
程序运行截图如下：



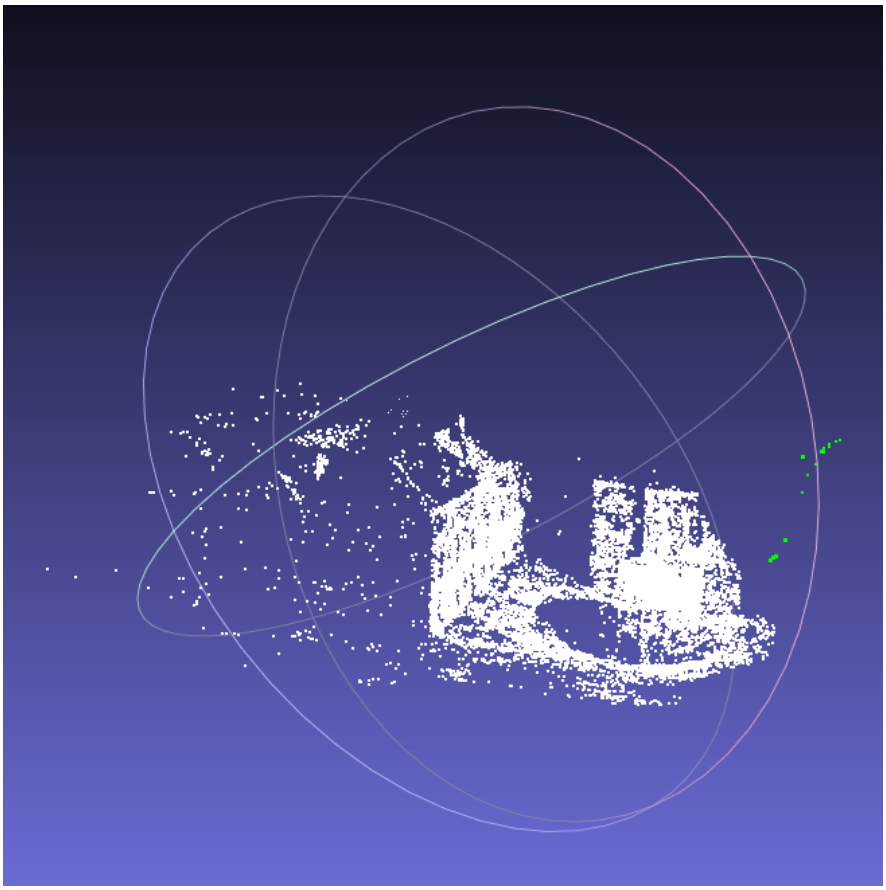
```
guoben@ubuntu-lab-qsingdao:~/Project/g2o/bin$ ./G2O
Header: 16 22106 83718iteration= 0       chi2= 8894423.022949    time= 0.243291 cumTime= 0.243291        edges= 83718    schur= 1        lambda= 227.832660     levenbergIter= 1
iteration= 1     chi2= 1772145.050517    time= 0.211092  cumTime= 0.454383       edges= 83718    schur= 1        lambda= 75.944220       levenbergIter= 1
iteration= 2     chi2= 752585.293391     time= 0.210866  cumTime= 0.665249       edges= 83718    schur= 1        lambda= 25.314740       levenbergIter= 1
iteration= 3     chi2= 402814.243627     time= 0.211976  cumTime= 0.877225       edges= 83718    schur= 1        lambda= 8.438247        levenbergIter= 1
iteration= 4     chi2= 284879.378894     time= 0.212487  cumTime= 1.08971        edges= 83718    schur= 1        lambda= 2.812749        levenbergIter= 1
iteration= 5     chi2= 238356.214415     time= 0.212948  cumTime= 1.30266        edges= 83718    schur= 1        lambda= 0.937583        levenbergIter= 1
iteration= 6     chi2= 193550.755079     time= 0.21209   cumTime= 1.51475        edges= 83718    schur= 1        lambda= 0.312528        levenbergIter= 1
iteration= 7     chi2= 146859.909574     time= 0.211265  cumTime= 1.72602        edges= 83718    schur= 1        lambda= 0.104176        levenbergIter= 1
iteration= 8     chi2= 122887.700218     time= 0.212287  cumTime= 1.9383         edges= 83718    schur= 1        lambda= 0.069451        levenbergIter= 1
iteration= 9     chi2= 97810.139925      time= 0.212756  cumTime= 2.15106        edges= 83718    schur= 1        lambda= 0.046300        levenbergIter= 1
iteration= 10    chi2= 80329.940265      time= 0.212666  cumTime= 2.36372        edges= 83718    schur= 1        lambda= 0.030867        levenbergIter= 1
iteration= 11    chi2= 65663.994405      time= 0.212922  cumTime= 2.57665        edges= 83718    schur= 1        lambda= 0.020578        levenbergIter= 1
iteration= 12    chi2= 55960.726637      time= 0.213769  cumTime= 2.79042        edges= 83718    schur= 1        lambda= 0.013719        levenbergIter= 1
iteration= 13    chi2= 53275.547797      time= 0.212083  cumTime= 3.0025         edges= 83718    schur= 1        lambda= 0.009146        levenbergIter= 1
iteration= 14    chi2= 35983.312124      time= 0.25865   cumTime= 3.26115        edges= 83718    schur= 1        lambda= 0.006097        levenbergIter= 2
iteration= 15    chi2= 32091.891518      time= 0.316039  cumTime= 3.57719        edges= 83718    schur= 1        lambda= 0.016259        levenbergIter= 3
iteration= 16    chi2= 31156.262647      time= 0.26633   cumTime= 3.84352        edges= 83718    schur= 1        lambda= 0.021679        levenbergIter= 2
iteration= 17    chi2= 30773.139623      time= 0.214418  cumTime= 4.05793        edges= 83718    schur= 1        lambda= 0.014453        levenbergIter= 1
iteration= 18    chi2= 29079.563460      time= 0.26089   cumTime= 4.31882        edges= 83718    schur= 1        lambda= 0.012488        levenbergIter= 2
iteration= 19    chi2= 28484.154313      time= 0.264056  cumTime= 4.58288        edges= 83718    schur= 1        lambda= 0.016651        levenbergIter= 2
iteration= 20    chi2= 28445.405201      time= 0.214679  cumTime= 4.79756        edges= 83718    schur= 1        lambda= 0.011101        levenbergIter= 1
iteration= 21    chi2= 27170.592543      time= 0.263263  cumTime= 5.06082        edges= 83718    schur= 1        lambda= 0.011118        levenbergIter= 2
iteration= 22    chi2= 26748.191194      time= 0.259594  cumTime= 5.32042        edges= 83718    schur= 1        lambda= 0.014824        levenbergIter= 2
iteration= 23    chi2= 26675.118188      time= 0.213874  cumTime= 5.53429        edges= 83718    schur= 1        lambda= 0.009883        levenbergIter= 1
iteration= 24    chi2= 26087.985781      time= 0.260373  cumTime= 5.79466        edges= 83718    schur= 1        lambda= 0.010281        levenbergIter= 2
iteration= 25    chi2= 25875.818536      time= 0.262105  cumTime= 6.05677        edges= 83718    schur= 1        lambda= 0.013708        levenbergIter= 2
iteration= 26    chi2= 25831.564925      time= 0.218704  cumTime= 6.27547        edges= 83718 schur= 1       lambda= 0.009139        levenbergIter= 1
iteration= 27    chi2= 25568.344873      time= 0.268039  cumTime= 6.54351        edges= 83718    schur= 1        lambda= 0.011118        levenbergIter= 2
iteration= 28    chi2= 25455.865005      time= 0.261249  cumTime= 6.80476        edges= 83718    schur= 1        lambda= 0.011781        levenbergIter= 2
iteration= 29    chi2= 25454.942053      time= 0.215076  cumTime= 7.01984        edges= 83718    schur= 1        lambda= 0.007854   levenbergIter= 1
iteration= 30    chi2= 25260.709796      time= 0.268959  cumTime= 7.2888         edges= 83718    schur= 1        lambda= 0.009148   levenbergIter= 2
iteration= 31    chi2= 25171.392636      time= 0.268004  cumTime= 7.5568         edges= 83718    schur= 1        lambda= 0.009425        levenbergIter= 2
iteration= 32    chi2= 25104.160294      time= 0.267466  cumTime= 7.82427        edges= 83718    schur= 1        lambda= 0.008637        levenbergIter= 2
iteration= 33    chi2= 25042.986799      time= 0.267169  cumTime= 8.09143        edges= 83718    schur= 1        lambda= 0.008765        levenbergIter= 2
iteration= 34    chi2= 24984.677998      time= 0.268584  cumTime= 8.36002        edges= 83718    schur= 1        lambda= 0.005949        levenbergIter= 2
iteration= 35    chi2= 24943.879912      time= 0.265447  cumTime= 8.62547        edges= 83718    schur= 1        lambda= 0.007933        levenbergIter= 2
iteration= 36    chi2= 24886.075504      time= 0.259778  cumTime= 8.88524        edges= 83718    schur= 1        lambda= 0.005674        levenbergIter= 2
iteration= 37    chi2= 24868.088225      time= 0.261634  cumTime= 9.14688        edges= 83718    schur= 1        lambda= 0.007565        levenbergIter= 2
iteration= 38    chi2= 24833.053138      time= 0.259935  cumTime= 9.40681        edges= 83718    schur= 1        lambda= 0.008448        levenbergIter= 2
iteration= 39    chi2= 24815.047826      time= 0.261902  cumTime= 9.66871        edges= 83718    schur= 1        lambda= 0.009766        levenbergIter= 2
```

使用MeshLab打开生成的两个PLY文件，截图如下：

初始的点云文件



经过给g2o BA优化的点云文件如下：



可以看出，通过给g2o BA优化的点云结构更加紧凑 条理也更加清晰

三 直接法的 Bundle Adjustment (5 分，约 3 小时)

3.1 数学模型

特征点法的 BA 以最小化重投影误差作为优化目标。相对的，如果我们以最小化光度误差为目标，就得到了直接法的 BA。之前我们在直接法 VO 中，谈到了如何用直接法去估计相机位姿。但是直接法亦可用来处理整个 `Bundle Adjustment`。下面，请你推导直接法 BA 的数学模型，并完成它的 g2o 实现。注意本题使用的参数化形式与实际的直接法还有一点不同，我们用 x, y, z 参数化每一个 3D 点，而实际的直接法多采用逆深度参数化 [1]。

本题给定 7 张图片，记为 0.png 至 6.png，每张图片对应的相机位姿初始值为 T_i，以 T_{cw} 形式存储在 `poses.txt` 文件中，其中每一行代表一个相机的位姿，格式如之前作业那样：

$time, t_x, t_y, t_z, q_x, q_y, q_z, q_w$

平移在前，旋转（四元数形式）在后。同时，还存在一个 3D 点集 P，共 N 个点。其中每一个点的初始坐标记作 $p_i = [x, y, z]$ 。每个点还有自己的固定灰度值，我们用 16 个数来描述，这 16 个数为该点周围 4x4的小块读数，记作 $I(p)_i$，顺序见图 1 。换句话说，小块从 `u - 2`,`v - 2` 取到 `u + 1`,`v + 1`，先迭代 `v`。那么，我们知道，可以把每个点投影到每个图像中，然后再看投影后点周围小块与原始的 4x4 小块有多大差异。那么，整体优化目标函数为：

$$\min \sum_{j=1}^{7} \sum_{i=1}^{N} \sum_{W} ||I(p_i) - I_j(\pi(KT_j p_i))||_2^2$$

即最小化任意点在任意图像中投影与其本身颜色之差。其中K 为相机内参（在程序内以全局变量形式给定），π为投影函数，W 指代整个`patch`。下面，请回答：

1. 如何描述任意一点投影在任意一图像中形成的error?

答：

$$error = I(p_i) - I_j(\pi(KT_j p_i))$$

2. 每个error 关联几个优化变量?

答：每个error关联三个优化变量 分别是相机的李代数姿态ξ，三维空间点坐标$P = X, Y, Z$。

3. error 关于各变量的雅可比是什么?

答：

error关于空间点坐标P(X,Y,Z)的导数为对应点的像素梯度，即

$$< Empty\ Math\ Block >$$

$$\frac{\partial e}{\partial \xi} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x XY}{Z^2} & f_x + \frac{fxX^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{fyY^2}{Z^2} & \frac{f_y XY}{Z^2} & \frac{f_y X}{Z} \end{bmatrix}$$

3.2 实现

下面，请你根据上述说明，使用 g2o 实现上述优化，并用 pangolin 绘制优化结果。程序框架见 `code/di-rectBA.cpp` 文件。实现过程中，思考并回答以下问题：

1. 能否不要以 $[x, y, z]^T$ 的形式参数化每个点?

答：可以，还可以使用逆深度的方法来参数化路标点。

2. 取 4x4 的 patch 好吗？取更大的 patch 好还是取小一点的 patch 好?

答：4*4的patch挺好，太小的patch不能反应真正的光度变化。太大的patch不能反应

3. 从本题中，你看到直接法与特征点法在 BA 阶段有何不同?

答：计算误差的方式不同，特征点法在BA阶段最小化的时特征点的重投影误差，直接法最小化的是像素点块的光度误差；

4. 由于图像的差异，你可能需要鲁棒核函数，例如 Huber。此时 Huber 的阈值如何选取?

答：在实践过程中，采用控制变量法对阈值可以做多次测验，取误差最小的阈值作为Huber的阈值。

提示：

1. 构建 Error 之前先要判断点是否在图像中，去除一部分边界的点。
2. 优化之后，Pangolin 绘制的轨迹与地图如图 1 所示。
3. 你也可以不提供雅可比的计算过程，让 g2o 自己计算一个数值雅可比。
4. 以上数据实际取自 DSO[1]。

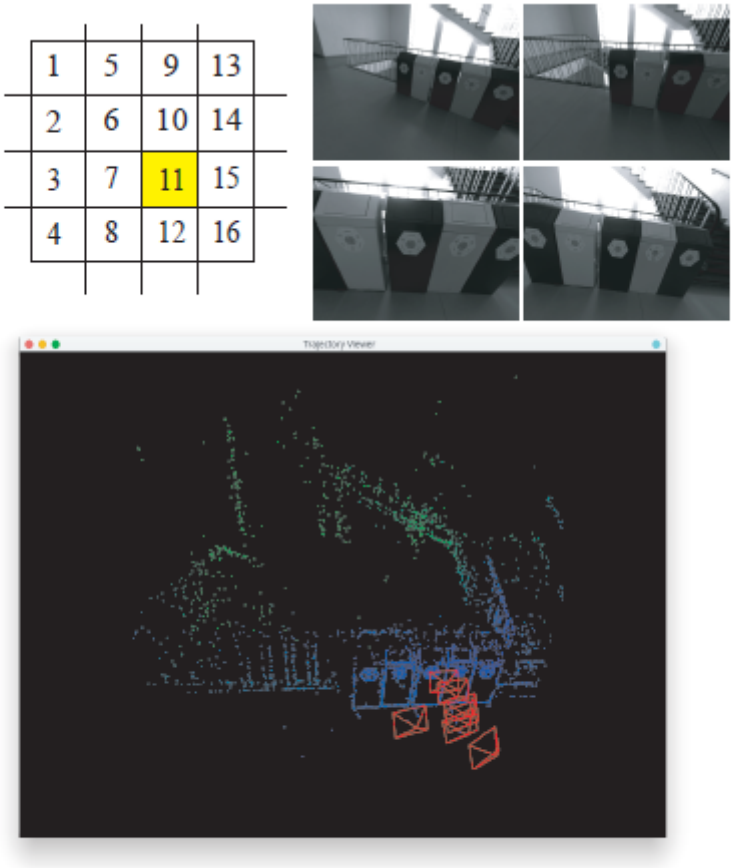
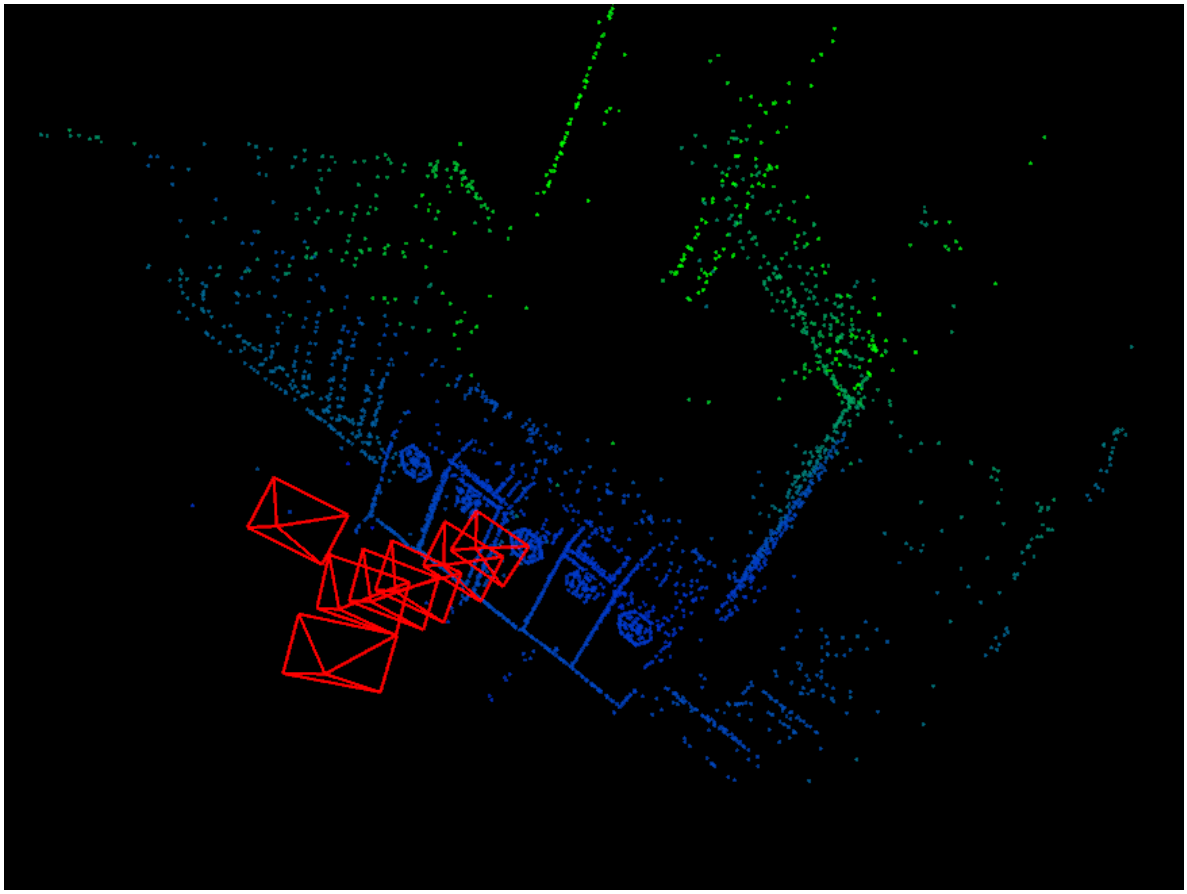
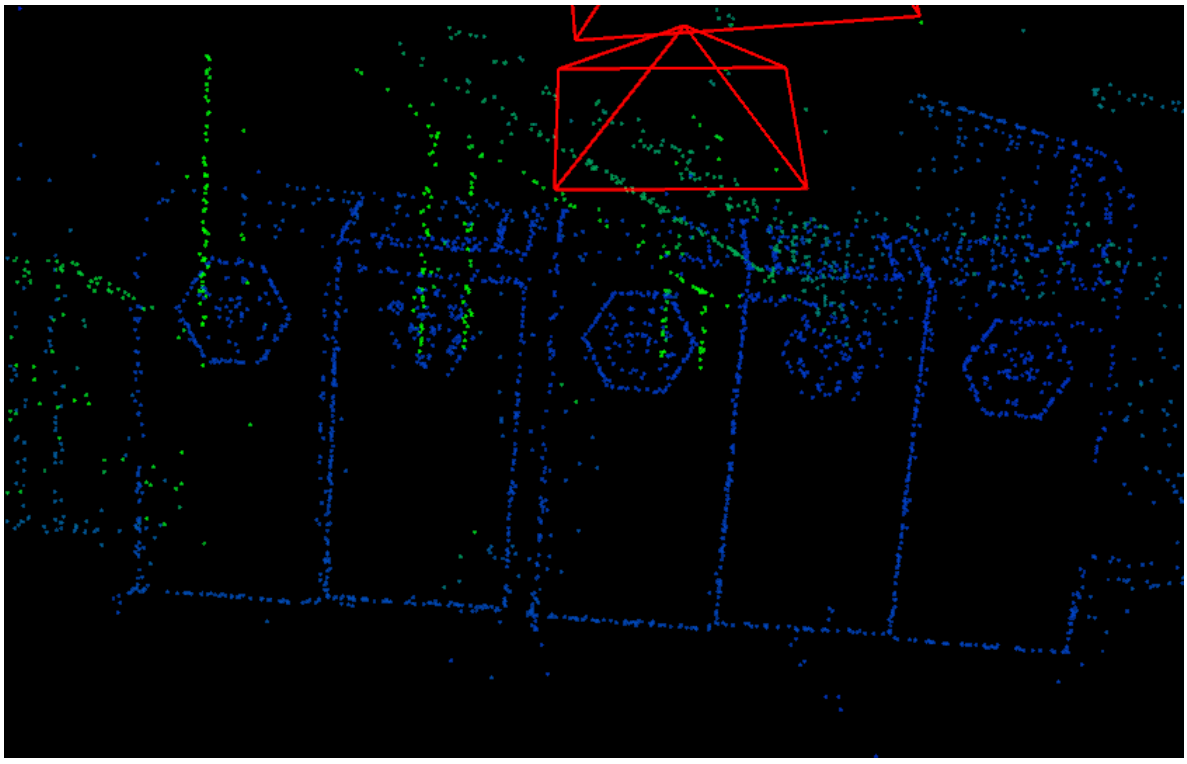


图 1：直接法 BA 的图例。左上：点颜色的定义顺序，其中 11 号点是观测到的位置；右上：图片示例；中间：优化后的相机位置与点云。

最后：

优化前截图：

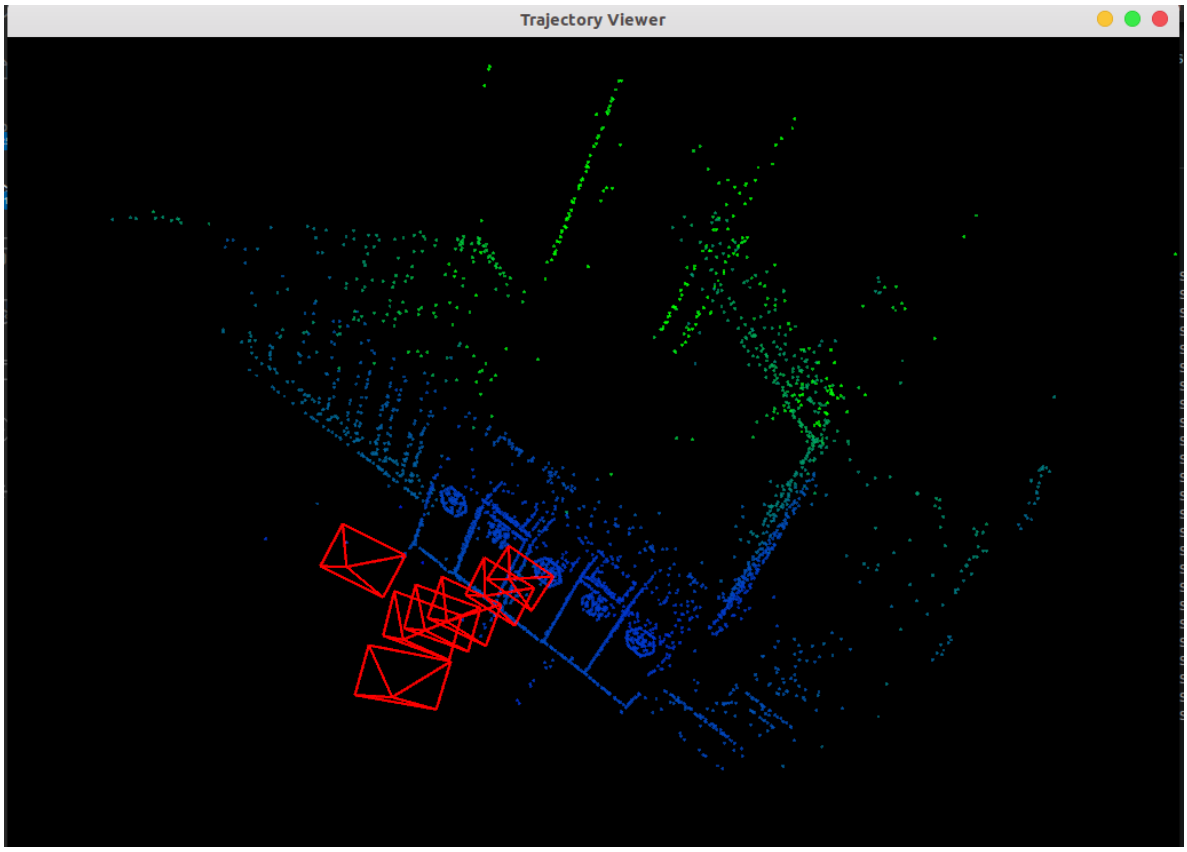


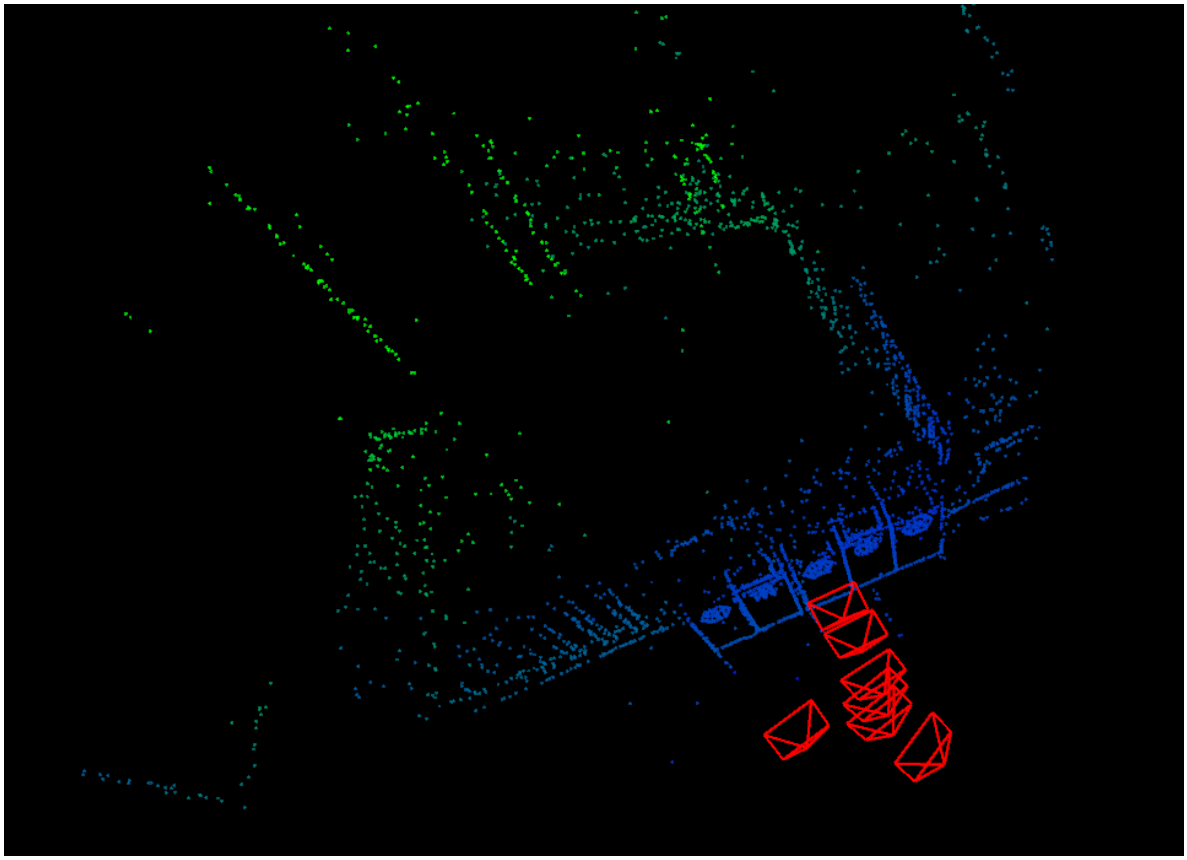


Terminal截图如下所示：

```
poses: 7, points: 4118
images: 7
iteration= 0     chi2= 3786234.227041    time= 0.000205975       cumTime= 0.000205975    edges= 49       schur= 1        lambda= 28747.768162    levenbergIter= 1
iteration= 1     chi2= 3704795.335200    time= 0.000129555       cumTime= 0.00033553     edges= 49       schur= 1        lambda= 9582.589387     levenbergIter= 1
iteration= 2     chi2= 3702534.229287    time= 0.000124981       cumTime= 0.000460511    edges= 49       schur= 1        lambda= 6388.392925     levenbergIter= 1
iteration= 3     chi2= 3642325.732678    time= 0.000123276       cumTime= 0.000583787    edges= 49       schur= 1        lambda= 4258.928617     levenbergIter= 1
iteration= 4     chi2= 3531872.145680    time= 0.000123233       cumTime= 0.00070702     edges= 49       schur= 1        lambda= 2839.285744     levenbergIter= 1
iteration= 5     chi2= 3366925.634407    time= 0.000122493       cumTime= 0.000829513    edges= 49       schur= 1        lambda= 946.428581      levenbergIter= 1
iteration= 6     chi2= 3356806.257256    time= 0.000122421       cumTime= 0.000951934    edges= 49       schur= 1        lambda= 630.952388      levenbergIter= 1
iteration= 7     chi2= 3350912.642360    time= 0.00012221        cumTime= 0.00107414     edges= 49       schur= 1        lambda= 420.634925      levenbergIter= 1
iteration= 8     chi2= 3350553.076095    time= 0.000235237       cumTime= 0.00130938     edges= 49       schur= 1        lambda= 588090249.613658        levenbergIter= 7
iteration= 9     chi2= 3348676.821709    time= 0.000122743       cumTime= 0.00143212     edges= 49       schur= 1        lambda= 392060166.409105        levenbergIter= 1
iteration= 10    chi2= 3346888.278670    time= 0.000160174       cumTime= 0.0015923      edges= 49       schur= 1        lambda= 2090987554.181894       levenbergIter= 3
iteration= 11    chi2= 3346383.650001    time= 0.000122082       cumTime= 0.00171438     edges= 49       schur= 1        lambda= 1393991702.787929       levenbergIter= 1
iteration= 12    chi2= 3346269.529451    time= 0.000122005       cumTime= 0.00183639     edges= 49       schur= 1        lambda= 929327801.858619        levenbergIter= 1
iteration= 13    chi2= 3346129.880584    time= 0.000130131       cumTime= 0.00196652     edges= 49       schur= 1        lambda= 619551867.905746        levenbergIter= 1
iteration= 14    chi2= 3345718.192620    time= 0.000123397       cumTime= 0.00208991     edges= 49       schur= 1        lambda= 413034578.603831        levenbergIter= 1
iteration= 15    chi2= 3345401.519323    time= 0.000159482       cumTime= 0.0022494      edges= 49       schur= 1        lambda= 2202851085.887097       levenbergIter= 3
iteration= 16    chi2= 3345304.481738    time= 0.000121796       cumTime= 0.00237119     edges= 49       schur= 1        lambda= 14685673390.591398      levenbergIter= 1
iteration= 17    chi2= 3345196.373304    time= 0.00012152        cumTime= 0.00249271     edges= 49       schur= 1        lambda= 914709362.982691        levenbergIter= 1
iteration= 18    chi2= 3345027.074667    time= 0.000121673       cumTime= 0.00261438     edges= 49       schur= 1        lambda= 304903120.994230        levenbergIter= 1
iteration= 19    chi2= 3344799.439676    time= 0.000121737       cumTime= 0.00273612     edges= 49       schur= 1        lambda= 203268747.329487        levenbergIter= 1
iteration= 20    chi2= 3344243.350535    time= 0.000122032       cumTime= 0.00285815     edges= 49       schur= 1        lambda= 67756249.109829         levenbergIter= 1
iteration= 21    chi2= 3344225.765151    time= 0.000196911       cumTime= 0.00305506     edges= 49       schur= 1        lambda= 46254932725.643204      levenbergIter= 5
iteration= 22    chi2= 3344211.097551    time= 0.000121352       cumTime= 0.00317642     edges= 49       schur= 1        lambda= 30836621817.095467      levenbergIter= 1
iteration= 23    chi2= 3344205.677083    time= 0.000121371       cumTime= 0.00329779     edges= 49       schur= 1        lambda= 20557747878.063644      levenbergIter= 1
iteration= 24    chi2= 3344205.677083    time= 0.000213399       cumTime= 0.00351119     edges= 49       schur= 1        lambda= 43112722077976928.000000        levenbergIter= 6
```

使用pangolin画出的点云图如下所示：

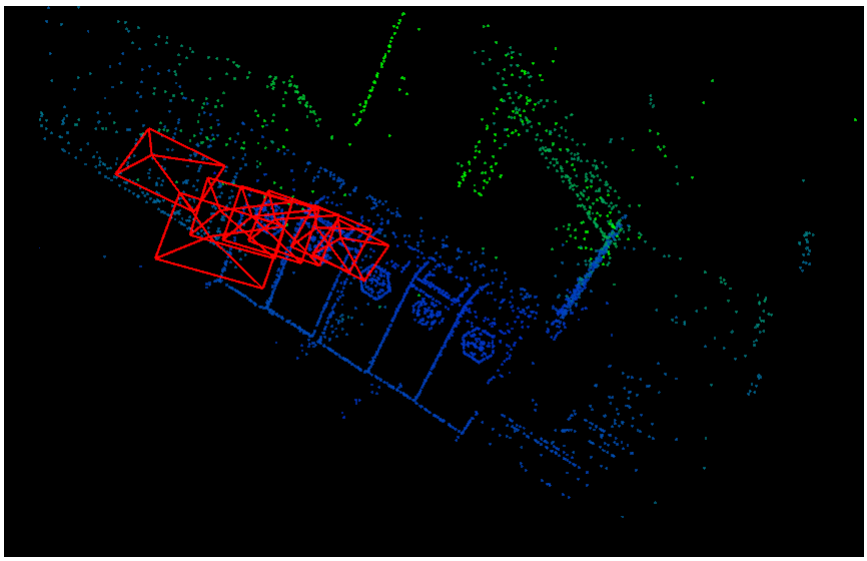




当注释掉鲁棒核函数时，结果如下：

只能优化4次

```
poses: 7, points: 4118
images: 7
iteration= 0     chi2= 4015405.323140    time= 0.000302543       cumTime= 0.000302543    edges= 49       schur= 1        lambda= 341678.969562   levenbergIter= 1
iteration= 1     chi2= 4015155.521767    time= 0.000229294       cumTime= 0.000531837    edges= 49       schur= 1        lambda= 227785.979708   levenbergIter= 1
iteration= 2     chi2= 4009764.531201    time= 0.000225806       cumTime= 0.000757643    edges= 49       schur= 1        lambda= 151857.319805   levenbergIter= 1
iteration= 3     chi2= 4009737.447556    time= 0.000334633       cumTime= 0.00109228     edges= 49       schur= 1        lambda= 212311921296.296204     levenbergIter= 7
iteration= 4     chi2= 4009737.447556    time= 0.000327374       cumTime= 0.00141965     edges= 49       schur= 1        lambda= 5699204740740738252 8.000000     levenbergIter= 7
```



- 在测试过程中发现，让g2o自己计算雅克比的效果并不好。