第七章作业

Student name: Francisrk

Due date: March 13th, 2022

1 第 1 题

已阅。

2 第 2 题

2.1 为何说 Bundle Adjustment is slow 是不对的？

文献 [1] 中提到：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.

之前认为 BA 慢的原因是人们没有意识到问题的结构和稀疏性，直接对 H 求逆的计算量非常大，也就显得比较慢，但是实际上 H 具有稀疏性，是可以通过诸如使用 Shur 消元对 H 进行 Marginalization，从而大大降低计算量，而经过处理之后的 BA 问题的效率通常比很多新的方法的效率更高，更灵活。

2.2 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。

1. 需要参数化的有 3D points(路标点 y)，3D Rotation(相机外参数 (R,t) 或者说是相机的位姿)，相机校准 (camera calibration) 也就是相机的内参数、投影后的像素坐标。

2. Pose：变换矩阵，欧拉角 (Euler Angles)，四元数 (Quaternions)

   (a) 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)。

   (b) 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。

   (c) 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. Point：三维坐标点 (X,Y,Z)，逆深度。

   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：

   (a) 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。

   (b) XYZ vs Inverse Depth：使用的 XYZ 还是逆深度。

   (c) Two different Inverse Depth：两种不同类型的逆深度参数。

2.3 * 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。

3.4 节的 Intensity- based methods 就是 BA 在直接法中的应用。第 5 节 Network Structure 可以对应到 SLAM 中的图优化模型；H 的稀疏性可以实现 BA 实时，在 07 年的 PTAM 上实现。

2.4 BAL-dataset

总结一下 BA 过程:

1. 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；
2. 往图里加入实际的节点和边；
3. 选择初值，开始迭代；
4. 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；
5. 求解稀疏线性方程 $Hk\Delta x = -bk$，得到梯度方向；
6. 继续用 GN 或 LM 进行迭代。如果迭代结束，返回优化值。

这里选择了教材上的例程中的数据 problem-16-22106-pre.txt 文件核心代码如 Listing1 所示：

```
//
// Created by wrk on 2022/3/28.
//

#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"

using namespace Sophus;
using namespace Eigen;
```

```
using namespace std;

/*
 * 问题梳理:
 * 1.首先，我们需要同时优化相机的位姿和路标点，相机位姿和路标点分别是顶点，
 * 2.然后，误差=观测-预测，采用重投影误差来当做误差
 * 3.最后，将所有的边和顶点插入到图中，构建g2o优化问题求解即可
 * 问题是第二步怎么求重投影？没有深度信息得不到3d点，还是说直接就给了3d的？
 * */

//先定义如何存放姿态，内参，畸变系数
struct PoseAndIntrinsics
{
    PoseAndIntrinsics() {}  //构造函数

    explicit PoseAndIntrinsics(double *data_addr)  //explicit之后只能用()进行初始化，不能用等号进行赋值
    {
        rotation = SO3d::exp(Vector3d(data_addr[0], data_addr[1], data_addr[2]));  //指数映射转换成SO3，罗德里格斯公式
        translation = Vector3d(data_addr[3], data_addr[4], data_addr[5]);
        focal = data_addr[6];
        k1 = data_addr[7];
        k2 = data_addr[8];
    }

    //将估计值放入内存
    void set_to(double *data_addr)
    {
        auto r = rotation.log();   //对数变换得se3
        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;  //旋转SO3
    Vector3d translation;  //平移向量
    double focal = 0;  // 焦距
    double k1=0,k2=0;   //畸变系数
};

class VertexPoseAndIntrinsics: public g2o::BaseVertex<9, PoseAndIntrinsics>{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;  //重写new，使内存对齐
```

```
62
63      VertexPoseAndIntrinsics() {}
64
65      //
66      virtual void setToOriginImpl() override
67      {
68          _estimate = PoseAndIntrinsics();  //这就是要估计的对象: 9维数据都需
        要被估计
69      }
70
71      // 估计更新
72      virtual void oplusImpl(const double *update) override{
73          _estimate.rotation = SO3d::exp(Vector3d(update[0], update[1], update
        [2])) * _estimate.rotation;
74          _estimate.translation += Vector3d(update[3],update[4],update[5]);
75          _estimate.focal += update[6];
76          _estimate.k1 += update[7];
77          _estimate.k2 += update[8];
78      }
79
80      //根据估计值投影一个点(估计的是相机系下的3d点)
81      Vector2d project(const Vector3d &point){
82          Vector3d pc = _estimate.rotation * point + _estimate.translation;
83          pc = -pc / pc[2];
84          /*
85           * 这个畸变的我没看懂
86           * */
87          double r2 = pc.squaredNorm();
88          double distortion = 1.0 + r2 * (_estimate.k1 + _estimate.k2 * r2);
89          return Vector2d(_estimate.focal * distortion * pc[0],
90                          _estimate.focal * distortion * pc[1]);
91      }
92
93      virtual bool read(istream &in) {}
94
95      virtual bool write(ostream &out) const {}
96  };
97
98  class VertexLandMark: public g2o::BaseVertex<3, Eigen::Vector3d>{
99  public:
100     EIGEN_MAKE_ALIGNED_OPERATOR_NEW;  //重写new, 使内存对齐
101
102     VertexLandMark() {}
103
104     virtual void setToOriginImpl() override
105     {
```

```
106         _estimate = Eigen::Vector3d(0,0,0);  //这就是要估计的对象: 9维数据都需要被估计
107     }
108 
109     virtual void oplusImpl(const double *update) override{
110         _estimate += Eigen::Vector3d(update[0], update[1], update[2]);
111     }
112 
113     virtual bool read(istream &in) {}
114 
115     virtual bool write(ostream &out) const {}
116 };
117 
118 //传入参数参数2 : 观测值 (这里是3D点在像素坐标系下的投影坐标) 的维度
119 //参数Vector : 观测值类型, piexl.x, piexl.y
120 //参数VertexSBAPointXYZ: 第一个顶点类型
121 //参数VertexSE3Expmap : 第二个顶点类型
122 class EdgeProjection: public g2o::BaseBinaryEdge<2, Vector2d, VertexPoseAndIntrinsics, VertexLandMark>{
123 public:
124     EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
125 
126 //    EdgeProjection(double x): BaseBinaryEdge(),
127     virtual void computeError() override  //使用override显式地声明虚函数覆盖
128     {
129         VertexPoseAndIntrinsics * v0 = dynamic_cast<VertexPoseAndIntrinsics *>(_vertices[0]);  //读取位姿9维顶点
130         VertexLandMark * v1 = dynamic_cast<VertexLandMark *>(_vertices[1]);  //读取位姿9维顶点
131         // 利用估计的相机位姿和路标点的3d坐标将3d坐标重投影称像素坐标, 与观测数据_measurement(实际上就是传进来的Vector2d)计算error,
132         // 再由g2o自己计算雅可比, 或者我么你自己定义那个2*6的雅可比
133         auto proj = v0->project(v1->estimate());
134         _error = proj - _measurement;    //为什么不是观测-预测??
135     }
136 
137     // use numeric derivatives
138     virtual bool read(istream &in) {}
139 
140     virtual bool write(ostream &out) const {}
141 
142 
143 };
144 
145 
146 /*构建g2o问题并求解*/
```

```
147 void SolveBA(BALProblem &bal_problem) {
148     const int point_block_size = bal_problem.point_block_size();
149     const int camera_block_size = bal_problem.camera_block_size();  //位姿,
        内参, 畸变系数
150     double *points = bal_problem.mutable_points();       //  观测点的起始地址
151     double *cameras = bal_problem.mutable_cameras();     //  camera参数的起始
        地址
152 
153     // pose dimension 9,  landmark is 3
154     // 1, 2定义blocksolver和linearsolver类型
155     typedef g2o::BlockSolver<g2o::BlockSolverTraits<9, 3>> BlockSolverType;
156     // 用到对应的LinearSolver就得include对应的.h文件, 比如这里的
        linear_solver_csparse,h
157     typedef g2o::LinearSolverCSparse<BlockSolverType::PoseMatrixType>
        LinearSolverType;
158     // 3.选择优化算法, 创建总求解器
159     auto solver = new g2o::OptimizationAlgorithmLevenberg(
160             g2o::make_unique<BlockSolverType>(g2o::make_unique<
        LinearSolverType>()));
161     // 4.创建稀疏优化器
162     g2o::SparseOptimizer optimizer;
163     optimizer.setAlgorithm(solver);
164     optimizer.setVerbose(true);
165     // 5.定义图的顶点和边, 添加到稀疏优化器中
166     const double *observations = bal_problem.observations();  //这个观测值就
        是前面的4维数据<camera_index_1> <point_index_1> <x_1> <y_1>
167     vector<VertexPoseAndIntrinsics *> vertex_pose_intrinsics;  //9维顶点临时
        变量
168     vector<VertexLandMark *> vertex_points;  //3位landmark顶点临时变量
169     // 插入相机位姿顶点: 3维罗德里格斯旋转向量R, 3维平移t, 1维焦距f, 2维径向
        畸变系数
170     for (int i = 0; i < bal_problem.num_cameras(); ++i) {
171         VertexPoseAndIntrinsics *v = new VertexPoseAndIntrinsics();
172         double *camera = cameras + camera_block_size * i;
173         v->setId(i);
174         v->setEstimate(PoseAndIntrinsics(camera));
175         optimizer.addVertex(v);
176         vertex_pose_intrinsics.push_back(v);
177     }
178     // 插入路标点(这个)
179     for (int i = 0; i < bal_problem.num_points(); ++i)
180     {
181         VertexLandMark *v = new VertexLandMark();
182         double *point = points + point_block_size * i;
183         v->setId(i + bal_problem.num_cameras());  //从camera后面开始继续编号
184         v->setEstimate(Vector3d(point[0], point[1], point[2]));
```

```
185         // g2o在BA中需要手动设置待Marg的顶点,在这里设置就是要把路标点给marg
        掉
186         v->setMarginalized(true);
187         optimizer.addVertex(v);
188         vertex_points.push_back(v);
189     }
190 
191     //
192     for(int i=0; i < bal_problem.num_observations(); ++i)  //观测的数量,2个
        值算一组观测, 所以取值的时候2*i
193     {
194         EdgeProjection *edge = new EdgeProjection;
195         edge->setVertex(0, vertex_pose_intrinsics[bal_problem.camera_index()
        [i]]);
196         edge->setVertex(1, vertex_points[bal_problem.point_index()[i]]);
197         edge->setMeasurement(Vector2d(observations[2*i+0], observations[2*i
        +1]));
198         edge->setInformation(Matrix2d::Identity());  //使用单位阵作为协方差
        矩阵
199         edge->setRobustKernel(new g2o::RobustKernelHuber());
200         optimizer.addEdge(edge);
201     }
202 
203     optimizer.initializeOptimization();
204     optimizer.optimize(40);
205 
206     //优化完成, 保存
207     //更新相机位姿那9维数据
208     for(int i=0; i<bal_problem.num_cameras(); ++i)
209     {
210         double *camera = cameras + camera_block_size * i;  //找camera地址
211         auto vertex = vertex_pose_intrinsics[i];  //取出优化后的9维顶点
212         auto estimate = vertex->estimate();  //读取估计值
213         estimate.set_to(camera);    //保存至9维顶点
214     }
215     //更新三维点坐标
216     for(int i=0; i<bal_problem.num_points(); ++i)
217     {
218         double *point = points + point_block_size * i;  //找三维点地址
219         auto vertex = vertex_points[i];
220         for(int k=0; k<3; ++k)
221             point[k] = vertex->estimate()[k];
222     }
223 }
224 
225 
```

```
226 int main(int argc, char** argv)
227 {
228     if (argc != 2) {
229         cout << "usage: BAL_g2o bal_data.txt" << endl;
230         return 1;
231     }
232     BALProblem bal_problem(argv[1]);
233     bal_problem.Normalize();
234     bal_problem.Perturb(0.1, 0.5, 0.5);  //R,t,P标准差，加入噪声
235     bal_problem.WriteToPLYFile("initial_g2o.ply");
236     SolveBA(bal_problem);
237     bal_problem.WriteToPLYFile("final_g2o.ply");
238     return 0;
239 }
```

Listing 1: BAL_g2o 核心代码

大致使用教材上的例程，加上自己的理解，整理成了博客 [2]：
选择教材上的数据进行实验，最终运行的结果如图 2.1 和图 2.2 所示

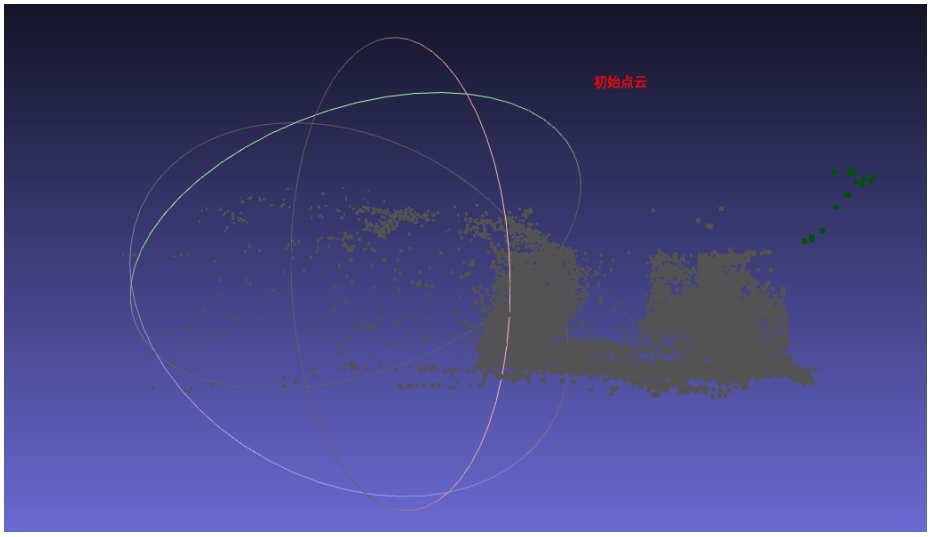

图 2.1 g2o 优化前的点云

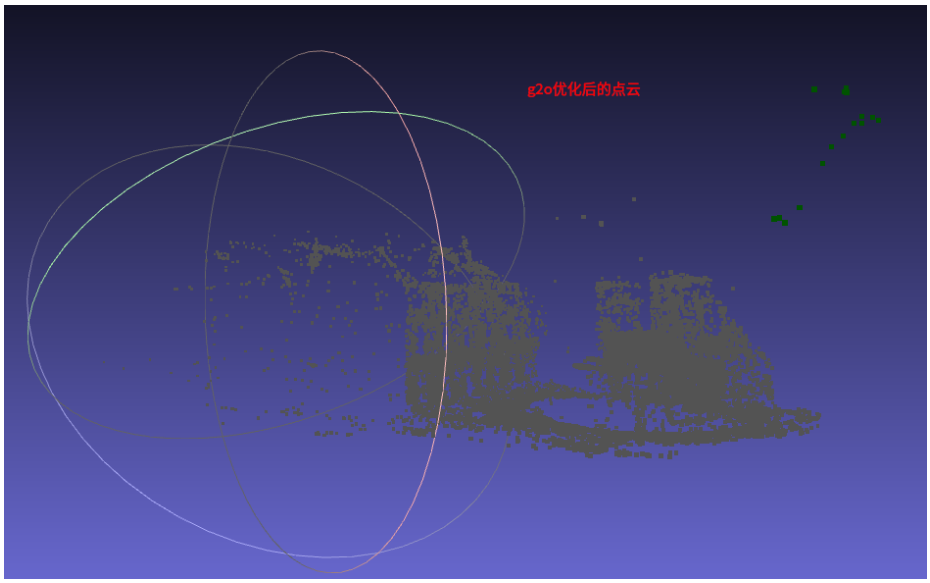

图 2.2 g2o 优化后的点云

3 直接法的 Bundle Adjustment

3.1 数学模型

1. 如何描述任意一点投影在任意一图像中形成的 error?

$$e = I(\boldsymbol{p}_i) - I_j(\pi(\boldsymbol{K}\boldsymbol{T}_j\boldsymbol{p}_i)) \tag{3.1}$$

2. 每个 error 关联几个优化变量?

   每个 error 关联了 2 组变量，共 9 个变量。第 1 组是相机位姿李代数 (6 自由度)。第 2 组是三维路标点位置：$[x, y, z]^T$。

3. error 关于各变量的雅可比是什么?

   这里的雅可比跟之前的一样，光度误差对路标点的导数是像素梯度 * 投影方程对相机系下的 3d 点的导数，见课本 P220。

像素梯度: 这里采用中心差分进行求导，参照 ch6 作业

$$\frac{\partial \boldsymbol{I}_2}{\partial \boldsymbol{u}} = -\frac{1}{2}\begin{bmatrix} I_2(u+1,v) - I_2(u-1,v) & I_2(u,v+1) - I_2(u,v-1) \end{bmatrix}_{1\times 2} \tag{3.2}$$

投影方程对相机系下的三维点的导数:

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{q}} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} \end{bmatrix} \tag{3.3}$$

相机系下三维点对相机位姿李代数的导数:

$$\frac{\partial \boldsymbol{q}}{\partial \delta \boldsymbol{\xi}} = \begin{bmatrix} \boldsymbol{I}, & -\boldsymbol{q}^\wedge \end{bmatrix} \tag{3.4}$$

将 (3.3) 与 (3.4) 合并得

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial \delta \boldsymbol{\xi}} &= \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{q}} * \frac{\partial \boldsymbol{q}}{\partial \delta \boldsymbol{\xi}} \\ &= \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x XY}{Z^2} & f_x + \frac{f_x X^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{f_y Y^2}{Z^2} & \frac{f_y XY}{Z^2} & \frac{f_y X}{Z} \end{bmatrix}_{2\times 6} \end{aligned} \tag{3.5}$$

需要说明，因为边连接的是变换之前的路标点 P，我们要优化这个路标点的位置，因为 q=TP，q 是变换之后的 3d 点，而 $\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{q}}$ 只是对变换之后的 3d 路标点的导数，为了对变换之前的路标点求导，所以还需要再乘个变换矩阵 T。所以误差对于 3d 坐标点的雅可比 (维数 16×3):

$$\boldsymbol{J}_1 = -\frac{\partial \boldsymbol{I}_2}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{q}} \frac{\partial \boldsymbol{q}}{\partial \boldsymbol{P}} = -\frac{\partial \boldsymbol{I}_2}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{q}} \boldsymbol{T} \tag{3.6}$$

误差对于相机位姿李代数的雅可比 (维数 16×6):

$$\boldsymbol{J}_2 = \frac{\partial \boldsymbol{I}_2}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \delta \boldsymbol{\xi}} \tag{3.7}$$

3.2 实现

1. 能否不要以 $[x, y, z]^T$ 的形式参数化每个点?

   可以，还可以使用逆深度的方法来参数化每个点，这种方式可以表示无限远点。

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

   4*4 的 patch 应该是一个比较适中的大小，patch 过大会导致计算量大，过小则会导致鲁棒性不强。

3. 从本题中，你看到直接法与特征点法在 BA 阶段有何不同？

   最大的不同是误差定义的不同，而误差又关联着优化的变量，所以优化变量也不同。

   直接法是计算光度误差，同时优化相机位姿和 3d 路标点（如果使用了带深度的 3d 点，如教材 P221 上所述，就不用优化路标点了，不知道我这个理解对不对，如果后面看到这里请再思考！）

   而特征点法的误差则是重投影误差，只优化相机位姿，因为路标点已经匹配完成。

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？

   Huber 阈值应该是根据多次实验，按照经验来确定的。

代码部分如 Listing2 所示

```
1  //
2  // Created by xiang on 1/4/18.
3  // this program shows how to perform direct bundle adjustment
4  //
5  #include <iostream>
6
7  #include <g2o/core/base_unary_edge.h>
8  #include <g2o/core/base_binary_edge.h>
9  #include <g2o/core/base_vertex.h>
10 #include <g2o/core/block_solver.h>
11 #include <g2o/core/optimization_algorithm_levenberg.h>
12 #include <g2o/solvers/dense/linear_solver_dense.h>
13 #include <g2o/core/robust_kernel.h>
14 #include <g2o/core/robust_kernel_impl.h>
15 #include <g2o/types/sba/types_six_dof_expmap.h>
16
17 #include <Eigen/Core>
18 #include <sophus/se3.hpp>
19 #include <opencv2/opencv.hpp>
20
21 #include <pangolin/pangolin.h>
22 #include <boost/format.hpp>
```

```
23
24 using namespace std;
25 using namespace Eigen;
26
27 typedef vector<Sophus::SE3d, Eigen::aligned_allocator<Sophus::SE3d>> VecSE3;
       //装相机位姿
28 typedef vector<Eigen::Vector3d, Eigen::aligned_allocator<Eigen::Vector3d>>
     VecVec3d;   //装3d路标点
29
30 // global variables
31 string pose_file = "./poses.txt";
32 string points_file = "./points.txt";
33
34 // intrinsics
35 float fx = 277.34;
36 float fy = 291.402;
37 float cx = 312.234;
38 float cy = 239.777;
39
40 // bilinear interpolation  双线性插值读取图像的灰度值
41 inline float GetPixelValue(const cv::Mat &img, float x, float y) {
42     uchar *data = &img.data[int(y) * img.step + int(x)];
43     float xx = x - floor(x);
44     float yy = y - floor(y);
45     return float(
46             (1 - xx) * (1 - yy) * data[0] +
47             xx * (1 - yy) * data[1] +
48             (1 - xx) * yy * data[img.step] +
49             xx * yy * data[img.step + 1]
50     );
51 }
52
53 // g2o vertex that use sophus::SE3 as pose  自定义位姿顶点，数据类型是SE3d
54 class VertexSophus : public g2o::BaseVertex<6, Sophus::SE3d> {
55 public:
56     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
57
58     VertexSophus() {}
59
60     ~VertexSophus() {}
61
62
63     virtual void setToOriginImpl() {
64         _estimate = Sophus::SE3d();
65     }
66
```

```
67      //根据估计值投影一个点(估计的是相机系下的3d点)
68      Vector2f project(const Vector3d &point)
69      {
70          //KTP 读取SE(3), 转换, 投影为像素坐标, 访问像素灰度值计算光度误差
71          Sophus::SE3d Tcw(estimate());
72          Vector3d point_cam3d = Tcw * point;
73          float u = fx * point_cam3d[0]/point_cam3d[2] + cx;
74          float v = fy * point_cam3d[1]/point_cam3d[2] + cy;
75          return Vector2f(u,v);
76      }
77
78      //更新
79      virtual void oplusImpl(const double *update_) {
80          //计算se(3), 再由se(3)为SE(3)
81          Eigen::Map<const Eigen::Matrix<double, 6, 1>> update(update_);
82          //保存估计值, 相当于_estimate = Sophus::SE3d::exp(update) *
        _estimate;
83          setEstimate(Sophus::SE3d::exp(update) * estimate());
84      }
85
86      virtual bool read(std::istream &is) {}
87
88      virtual bool write(std::ostream &os) const {}
89  };
90
91  // TODO edge of projection error, implement it
92  // 16x1 error, which is the errors in patch   16个像素点的光度差之和
93  typedef Eigen::Matrix<double,16,1> Vector16d;
94  class EdgeDirectProjection : public g2o::BaseBinaryEdge<16, Vector16d, g2o::
        VertexSBAPointXYZ, VertexSophus>  //一个是SBA的XYZ自带边, 一个是自定义
        的边
95          {
96  public:
97      EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
98
99      //边构造函数
100     EdgeDirectProjection(float *color, cv::Mat &target) {
101         this->origColor = color;
102         this->targetImg = target;
103     }
104
105     ~EdgeDirectProjection() {}
106
107     virtual void computeError() override {
108         // TODO START YOUR CODE HERE
109         // compute projection error ...
```

```cpp
        //projected = KTP  需要使用SE(3)
        g2o::VertexSBAPointXYZ* vertexPw = static_cast<g2o::VertexSBAPointXYZ *>(_vertices[0]);
        VertexSophus* vertexTcw = static_cast<VertexSophus *>(_vertices[1]);
        Vector2f proj = vertexTcw->project(vertexPw->estimate());
        //判断是否越界，若越界，则将error该位置1，并setLevel(1)不知道啥意思，是记录好坏的吗？
        if(proj[0]<-2 || proj[0]+2>targetImg.cols || proj[1]<-2 || proj[1]+2>targetImg.rows)
        {
            this->setLevel(1);  //设置level为1，标记为outlier，下次不再对该边进行优化
            for(int i=0; i<16; ++i)  _error[i] = 0;  //_error是16*1的向量
        }
        else{
            for(int i=-2; i<2;++i){
                for(int j=-2; j<2; ++j){
                    /******************************************************/
                    int num = 4 * i + j + 10;   //为什么要加10？？？？？？？？？？？？？？？？？？？？？？？
                    /******************************************************/
                    //_measurement是一个16*1的向量，所以_error也是16*1
                    _error[num] = origColor[num] - GetPixelValue(targetImg, proj[0]+i, proj[1]+j);
                }
            }
        }
        // END YOUR CODE HERE
    }
    // Let g2o compute jacobian for you  自己不算了,后面再算
    virtual void linearizeOplus() override
    {

        //分别计算3d点和李代数的雅可比
//        J_I2u
//        J_uq
//        J_qxi
        if(level()==1)
        {
            _jacobianOplusXi = Eigen::Matrix<double, 16, 3>::Zero();  //因为_error是(D,1)的，D=16是4*4的patch，所以对第一个顶点的雅可比就是16*2*2*3=16*3的
            _jacobianOplusXj = Eigen::Matrix<double, 16, 6>::Zero();  //同理，对第二个顶点(李代数顶点)的雅可比是16*2*2*3*3*6=16*6的
```

```
145             return;
146         }
147 
148         g2o::VertexSBAPointXYZ *vertexPw = static_cast<g2o::
    VertexSBAPointXYZ *> (_vertices[0]);
149         VertexSophus *vertexTcw = static_cast<VertexSophus *> (_vertices[1])
    ;
150         Vector2f proj = vertexTcw->project(vertexPw->estimate());
151 
152         double X = vertexPw->estimate()[0], Y = vertexPw->estimate()[1], Z =
     vertexPw->estimate()[2];
153         double inv_z = 1.0 / Z, inv_z2 = inv_z * inv_z;  //cur中的3D坐标X'Y'
    Z'
154         float u = proj[0], v = proj[1];
155 
156         Eigen::Matrix<double,1,2> J_I2u;    //像素梯度
157         Eigen::Matrix<double,2,3> J_uq;     //投影方程对3d点导数
158         Eigen::Matrix<double,3,6> J_qxi = Eigen::Matrix<double,3,6>::Zero();
         //3d点对李代数导数
159 
160         J_uq(0,0) = fx * inv_z;
161         J_uq(0,1) = 0;
162         J_uq(0,2) = -(fx*X) * inv_z2;
163         J_uq(1,0) = 0;
164         J_uq(1,1) = fy * inv_z;
165         J_uq(1,2) = -fy * Y * inv_z2;
166 
167         //这个矩阵见P187笔记[I| -P'^]
168         J_qxi(0,0) = 1;
169         J_qxi(0,4) = Z;
170         J_qxi(0,5) = -Y;
171         J_qxi(1,1) = 1;
172         J_qxi(1,3) = -Z;
173         J_qxi(1,5) = X;
174         J_qxi(2,2) = 1;
175         J_qxi(2,3) = -Y;
176         J_qxi(2,4) = -X;
177 
178         for(int x=-2; x<2; ++x)
179             for(int y=-2; y<2; ++y)
180             {
181                 //这num到底是什么意思？？？实际上是计算patch中第几个，num
    =4*(x+2)+(y+2)=4x+y+10
182                 int num = 4 * x + y + 10;
183 
184                 //中心差分计算像素梯度
```

```
185                 J_I2u(0,0) = (1.0 / 2) * (GetPixelValue(targetImg, u+1+x, v+
    y)-GetPixelValue(targetImg, u-1+x, v+y));
186                 J_I2u(0,1) = (1.0 / 2) * (GetPixelValue(targetImg, u+x, v+1+
    y)-GetPixelValue(targetImg, u+x, v-1+y));
187
188                 /********为什么还要乘以一个变换矩阵？******/
189 //                _jacobianOplusXi.block<1, 3>(num, 0) = -J_I2u * J_uq *
    vertexTcw->estimate().rotationMatrix();
190                 _jacobianOplusXi.block<1, 3>(num, 0) = -J_I2u * J_uq ;
191
192                 _jacobianOplusXj.block<1, 6>(num, 0) = -J_I2u * J_uq * J_qxi
    ;
193
194             }
195     }
196
197     virtual bool read(istream &in) {}
198
199     virtual bool write(ostream &out) const {}
200
201 private:
202     cv::Mat targetImg;  // the target image
203     float *origColor = nullptr;   // 16 floats, the color of this point
204 };
205
206 // plot the poses and points for you, need pangolin
207 void Draw(const VecSE3 &poses, const VecVec3d &points, string title);
208
209 int main(int argc, char **argv) {
210
211     // read poses and points
212     VecSE3 poses;
213     VecVec3d points;
214     ifstream fin(pose_file);  //读取相机位姿（7张图，7个位姿）
215
216     //读取位姿
217     while (!fin.eof())
218     {
219         double timestamp = 0;
220         fin >> timestamp;
221         if (timestamp == 0) break;
222         double data[7];
223         for (auto &d: data) fin >> d;
224         //四元数和平移向量来构建SE3
225         poses.push_back(Sophus::SE3d
226         (
```

```
227                 Eigen::Quaterniond(data[6], data[3], data[4], data[5]),
228                 Eigen::Vector3d(data[0], data[1], data[2])
229         ));
230         if (!fin.good()) break;
231     }
232     fin.close();
233 
234     //读取3d路标点XYZ
235     vector<float *> color;
236     fin.open(points_file);
237     while (!fin.eof())
238     {
239         double xyz[3] = {0};
240         for (int i = 0; i < 3; i++) fin >> xyz[i];
241         if (xyz[0] == 0) break;
242         points.push_back(Eigen::Vector3d(xyz[0], xyz[1], xyz[2]));
243         float *c = new float[16];
244         for (int i = 0; i < 16; i++) fin >> c[i];
245         color.push_back(c);
246 
247         if (fin.good() == false) break;
248     }
249     fin.close();
250 
251     cout << "poses: " << poses.size() << ", points: " << points.size() <<
      endl;
252 
253 
254     // read images 读取所有图片
255     vector<cv::Mat> images;
256     boost::format fmt("./%d.png");
257     for (int i = 0; i < 7; i++)
258     {
259         images.push_back(cv::imread((fmt % i).str(), 0));
260     }
261 
262     // build optimization problem
263     typedef g2o::BlockSolver<g2o::BlockSolverTraits<6, 3>> DirectBlock;  //
      求解的向量是6 * 1的
264     typedef g2o::LinearSolverDense<DirectBlock::PoseMatrixType>
      LinearSolverType;
265 //    DirectBlock::LinearSolverType *linearSolver = new g2o::
      LinearSolverDense<DirectBlock::PoseMatrixType>();
266 //    DirectBlock *solver_ptr = new DirectBlock(linearSolver);
267     auto solver = new g2o::OptimizationAlgorithmLevenberg(g2o::make_unique<
      DirectBlock>(g2o::make_unique<LinearSolverType>()));
```

```
268     g2o::SparseOptimizer optimizer;
269     optimizer.setAlgorithm(solver);
270     optimizer.setVerbose(true);
271 
272     // TODO add vertices, edges into the graph optimizer
273     vector<g2o::VertexSBAPointXYZ *> vertex_points;  //3位landmark顶点临时变量
274     vector<VertexSophus *> vertex_pose;  //pose顶点临时变量
275 
276     // START YOUR CODE HERE
277     //插入路标顶点
278     for(int i=0; i<points.size(); ++i)
279     {
280         g2o::VertexSBAPointXYZ *v = new g2o::VertexSBAPointXYZ;
281         v->setId(i);
282         v->setEstimate(points[i]);
283         v->setMarginalized(true);   //设置边缘化路标点
284         optimizer.addVertex(v);
285         vertex_points.push_back(v);
286     }
287     //插入位姿顶点
288     for(int i=0; i<poses.size(); ++i)
289     {
290         VertexSophus *v = new VertexSophus();
291         v->setId(i + points.size());
292         v->setEstimate(poses[i]);
293         optimizer.addVertex(v);
294         vertex_pose.push_back(v);
295     }
296 
297     //插入边
298     for(int c=0; c<poses.size(); ++c)
299         for(int p=0; p<points.size(); ++p)
300         {
301             EdgeDirectProjection *edge = new EdgeDirectProjection(color[p], images[c]);  //每个图中的每个点都插入到优化图中，都有一条边
302             //先point后pose
303             edge->setVertex(0, dynamic_cast<g2o::VertexSBAPointXYZ *>(optimizer.vertex(p)));
304             edge->setVertex(1, dynamic_cast<VertexSophus *>(optimizer.vertex(points.size()+c)));
305 //            edge->setMeasurement(Vector16d );
306             // 信息矩阵可直接设置为 error_dim*error_dim 的单位阵
307             edge->setInformation(Eigen::Matrix<double, 16, 16>::Identity());
308             // 设置Huber核函数，减小错误点影响，加强鲁棒性
309             g2o::RobustKernelHuber *rk = new g2o::RobustKernelHuber;
```

```
310             rk->setDelta(1.0);  //A squared error above delta^2 is
    considered as outlier in the data
311             edge->setRobustKernel(rk);
312             optimizer.addEdge(edge);
313         }
314
315     // END YOUR CODE HERE
316
317     Draw(poses, points, string("before"));
318
319     // perform optimization
320     optimizer.initializeOptimization(0);
321     optimizer.optimize(100);
322
323     // TODO fetch data from the optimizer
324     // START YOUR CODE HERE
325     for(int c=0; c<poses.size(); ++c)
326         for(int p=0; p<points.size(); ++p)
327         {
328             points[p] = dynamic_cast<g2o::VertexSBAPointXYZ *>(optimizer.
    vertex(p))->estimate();
329             poses[c] = dynamic_cast<VertexSophus *>(optimizer.vertex(points.
    size()+c))->estimate();
330         }
331
332     // END YOUR CODE HERE
333
334     // plot the optimized points and poses
335     Draw(poses, points, "after");
336
337     // 看看这数据有没有什么不一样的？怎么看优化的对不对？
338
339
340     // delete color data
341     for (auto &c: color) delete[] c;
342     return 0;
343 }
344
345 void Draw(const VecSE3 &poses, const VecVec3d &points, string title) {
346     if (poses.empty() || points.empty()) {
347         cerr << "parameter is empty!" << endl;
348         return;
349     }
350
351     // create pangolin window and plot the trajectory
352     pangolin::CreateWindowAndBind(title, 1024, 768);
```

```
353     glEnable(GL_DEPTH_TEST);
354     glEnable(GL_BLEND);
355     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
356 
357     pangolin::OpenGlRenderState s_cam(
358             pangolin::ProjectionMatrix(1024, 768, 500, 500, 512, 389, 0.1,
        1000),
359             pangolin::ModelViewLookAt(0, -0.1, -1.8, 0, 0, 0, 0.0, -1.0,
        0.0)
360     );
361 
362     pangolin::View &d_cam = pangolin::CreateDisplay()
363             .SetBounds(0.0, 1.0, pangolin::Attach::Pix(175), 1.0, -1024.0f /
         768.0f)
364             .SetHandler(new pangolin::Handler3D(s_cam));
365 
366 
367     while (pangolin::ShouldQuit() == false) {
368         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
369 
370         d_cam.Activate(s_cam);
371         glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
372 
373         // draw poses
374         float sz = 0.1;
375         int width = 640, height = 480;
376         for (auto &Tcw: poses) {
377             glPushMatrix();
378             Sophus::Matrix4f m = Tcw.inverse().matrix().cast<float>();
379             glMultMatrixf((GLfloat *) m.data());
380             glColor3f(1, 0, 0);
381             glLineWidth(2);
382             glBegin(GL_LINES);
383             glVertex3f(0, 0, 0);
384             glVertex3f(sz * (0 - cx) / fx, sz * (0 - cy) / fy, sz);
385             glVertex3f(0, 0, 0);
386             glVertex3f(sz * (0 - cx) / fx, sz * (height - 1 - cy) / fy, sz);
387             glVertex3f(0, 0, 0);
388             glVertex3f(sz * (width - 1 - cx) / fx, sz * (height - 1 - cy) /
    fy, sz);
389             glVertex3f(0, 0, 0);
390             glVertex3f(sz * (width - 1 - cx) / fx, sz * (0 - cy) / fy, sz);
391             glVertex3f(sz * (width - 1 - cx) / fx, sz * (0 - cy) / fy, sz);
392             glVertex3f(sz * (width - 1 - cx) / fx, sz * (height - 1 - cy) /
    fy, sz);
393             glVertex3f(sz * (width - 1 - cx) / fx, sz * (height - 1 - cy) /
```

```
    fy, sz);
394             glVertex3f(sz * (0 - cx) / fx, sz * (height - 1 - cy) / fy, sz);
395             glVertex3f(sz * (0 - cx) / fx, sz * (height - 1 - cy) / fy, sz);
396             glVertex3f(sz * (0 - cx) / fx, sz * (0 - cy) / fy, sz);
397             glVertex3f(sz * (0 - cx) / fx, sz * (0 - cy) / fy, sz);
398             glVertex3f(sz * (width - 1 - cx) / fx, sz * (0 - cy) / fy, sz);
399             glEnd();
400             glPopMatrix();
401         }
402
403         // points
404         glPointSize(2);
405         glBegin(GL_POINTS);
406         for (size_t i = 0; i < points.size(); i++) {
407             glColor3f(0.0, points[i][2]/4, 1.0-points[i][2]/4);
408             glVertex3d(points[i][0], points[i][1], points[i][2]);
409         }
410         glEnd();
411
412         pangolin::FinishFrame();
413         usleep(5000);   // sleep 5 ms
414     }
415 }
```

Listing 2: Direct_BA.cpp

优化前的点云图如图 3.1 所示，优化后的如图 3.2 所示

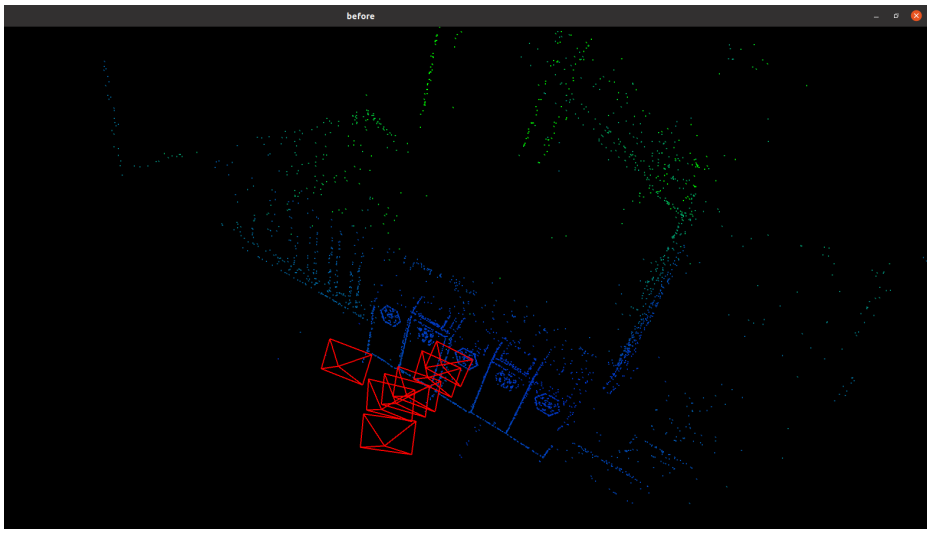

图 3.1 g2o 优化前的点云

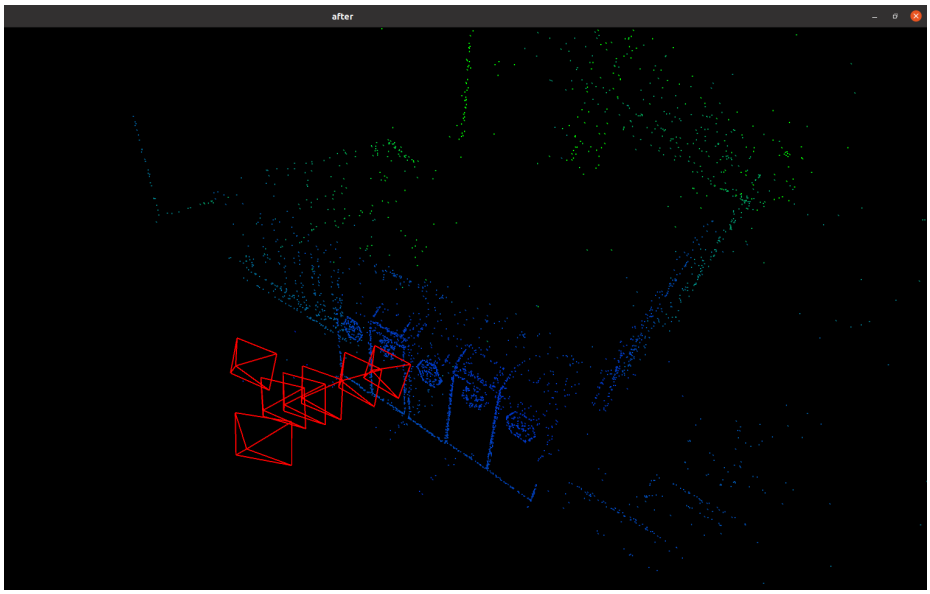

图 3.2 g2o 优化后的点云

实验了自己写的雅可比和让 g2o 自己写雅可比的性能对比，自己写的雅可比优化速度要快一些，且边的代价也要小一些，对比分别如图 3.3 和 3.4 所示:

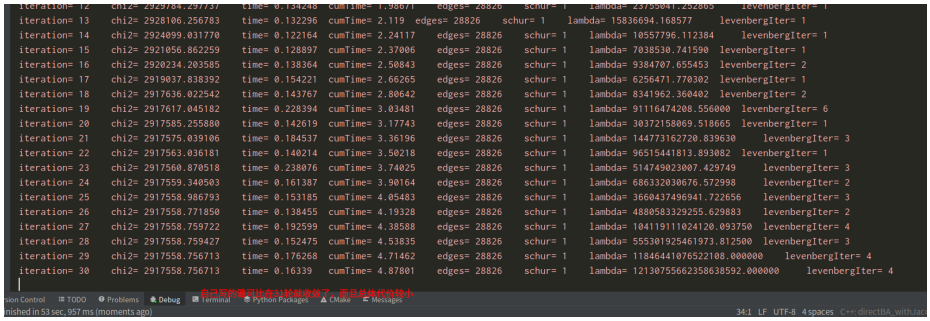

图 3.3 自己实现雅可比的运行结果

图 3.4 g2o 自动求雅可比的运行结果

参考文献

[1] J. Engel, V. Koltun, and D. Cremers, "Direct sparse odometry," arXiv preprint arXiv:1607.02565, 2016.

[2] https://blog.csdn.net/qq_37746927/article/details/123787617