1. 补充代码，实现高斯牛顿方法对 Pose-Graph 进行优化；（6 分）

[代码]：计算雅克比矩阵和误差：

void CalcJacobianAndError(Eigen::Vector3d *xi*, Eigen::Vector3d *xj*, Eigen::Vector3d *z*,

Eigen::Vector3d& *ei*, Eigen::Matrix3d& *Ai*, Eigen::Matrix3d& *Bi*)

{

*//TODO--Start*

Eigen::Matrix3d trans\_xi = PoseToTrans(xi);

Eigen::Matrix3d trans\_xj = PoseToTrans(xj);

Eigen::Matrix3d trans\_z = PoseToTrans(z);

Eigen::Matrix2d R\_i = trans\_xi.block(0, 0, 2, 2);

Eigen::Matrix2d R\_j = trans\_xj.block(0, 0, 2, 2);

Eigen::Matrix2d R\_ij = trans\_z.block(0, 0, 2, 2);

Eigen::Vector2d t\_i = xi.block(0, 0, 2, 1);

Eigen::Vector2d t\_j = xj.block(0, 0, 2, 1);

Eigen::Vector2d t\_ij = z.block(0, 0, 2, 1);

ei.block(0, 0, 2, 1) = R\_ij.transpose() \* (R\_i.transpose() \* (t\_j - t\_i) - t\_ij);

ei[2] = xj[2] - xi[2] - z[2];

NormalAngle(ei[2]);

Ai.setZero();

Ai.block(0, 0, 2, 2) = -R\_ij.transpose() \* R\_i.transpose();

Ai(2, 2) = -1;

Eigen::Matrix2d derivative\_Ri\_theta;

derivative\_Ri\_theta << -sin(xi[2]), cos(xi[2]), -cos(xi[2]), -sin(xi[2]);

Ai.block(0, 2, 2, 1) = R\_ij.transpose() \* derivative\_Ri\_theta \* (t\_j - t\_i);

Bi.setZero();

Bi.block(0, 0, 2, 2) = R\_ij.transpose() \* R\_i.transpose();

Bi(2, 2) = 1;

*//TODO--end*

}

计算H和b

*//TODO--Start*

b.block(3 \* tmpEdge.xi, 0, 3, 1) += (ei.transpose() \* infoMatrix \* Ai).transpose();

b.block(3 \* tmpEdge.xj, 0, 3, 1) += (ei.transpose() \* infoMatrix \* Bi).transpose();

H.block(3 \* tmpEdge.xi, 3 \* tmpEdge.xi, 3, 3) += Ai.transpose() \* infoMatrix \* Ai;

H.block(3 \* tmpEdge.xj, 3 \* tmpEdge.xj, 3, 3) += Bi.transpose() \* infoMatrix \* Bi;

H.block(3 \* tmpEdge.xi, 3 \* tmpEdge.xj, 3, 3) += Ai.transpose() \* infoMatrix \* Bi;

H.block(3 \* tmpEdge.xj, 3 \* tmpEdge.xi, 3, 3) += Bi.transpose() \* infoMatrix \* Ai;

*//TODO--End*

求解更新量dx

*//TODO--Start*

*// dx = -H.lu().solve(b);*

dx = -H.colPivHouseholderQr().solve(b);

*//TODO-End*

更新顶点

*//进行更新*

*//TODO--Start*

for (size\_t j = 0; j < Vertexs.size(); j++) {

Vertexs[j] += dx.block(3 \* j, 0, 3, 1);

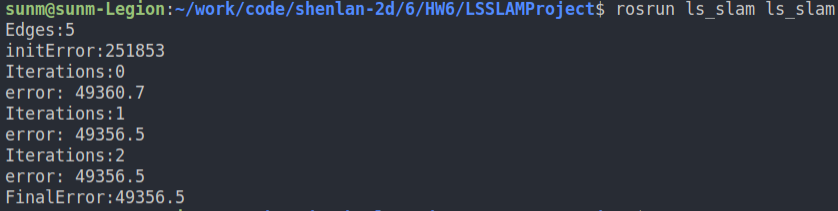
NormalAngle(Vertexs[j][2]);

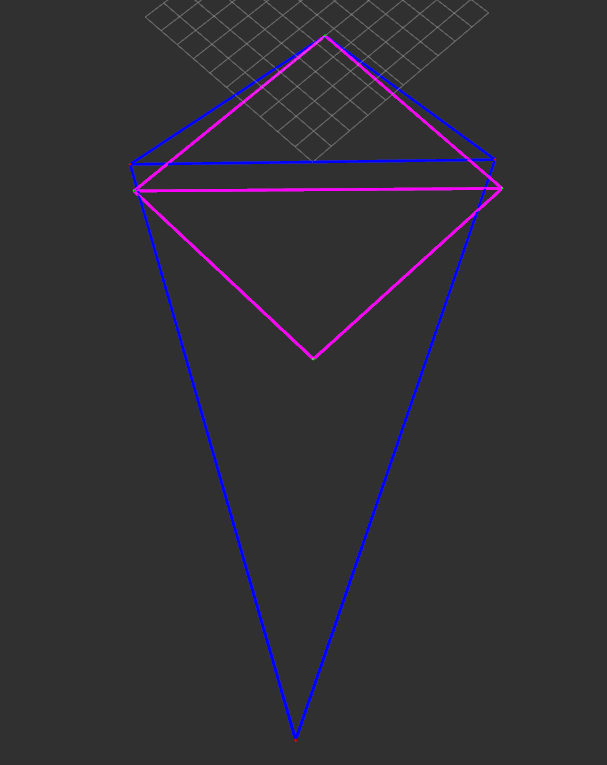
}

*//TODO--End*

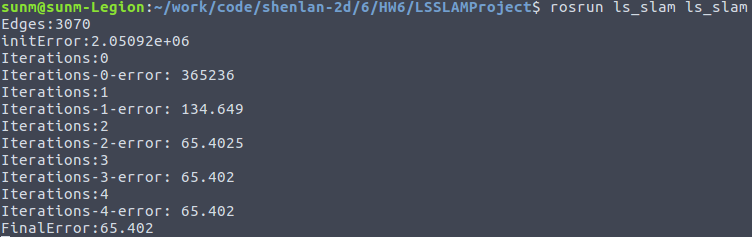
运行效果：

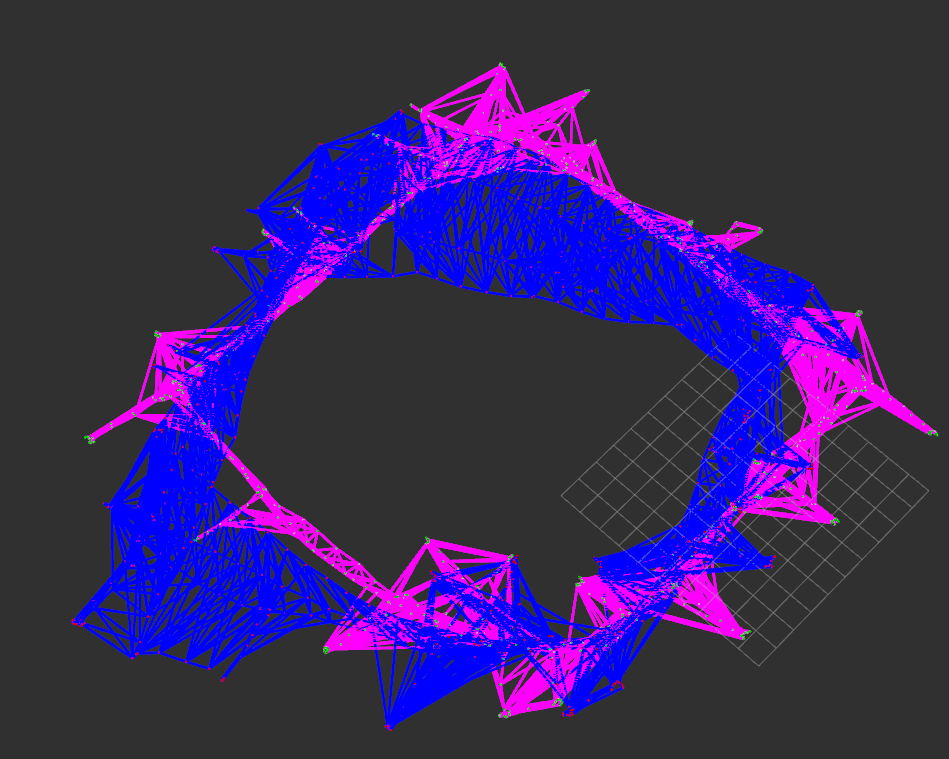
Test数据集



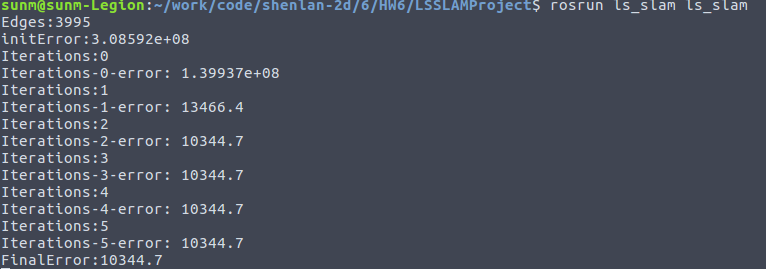


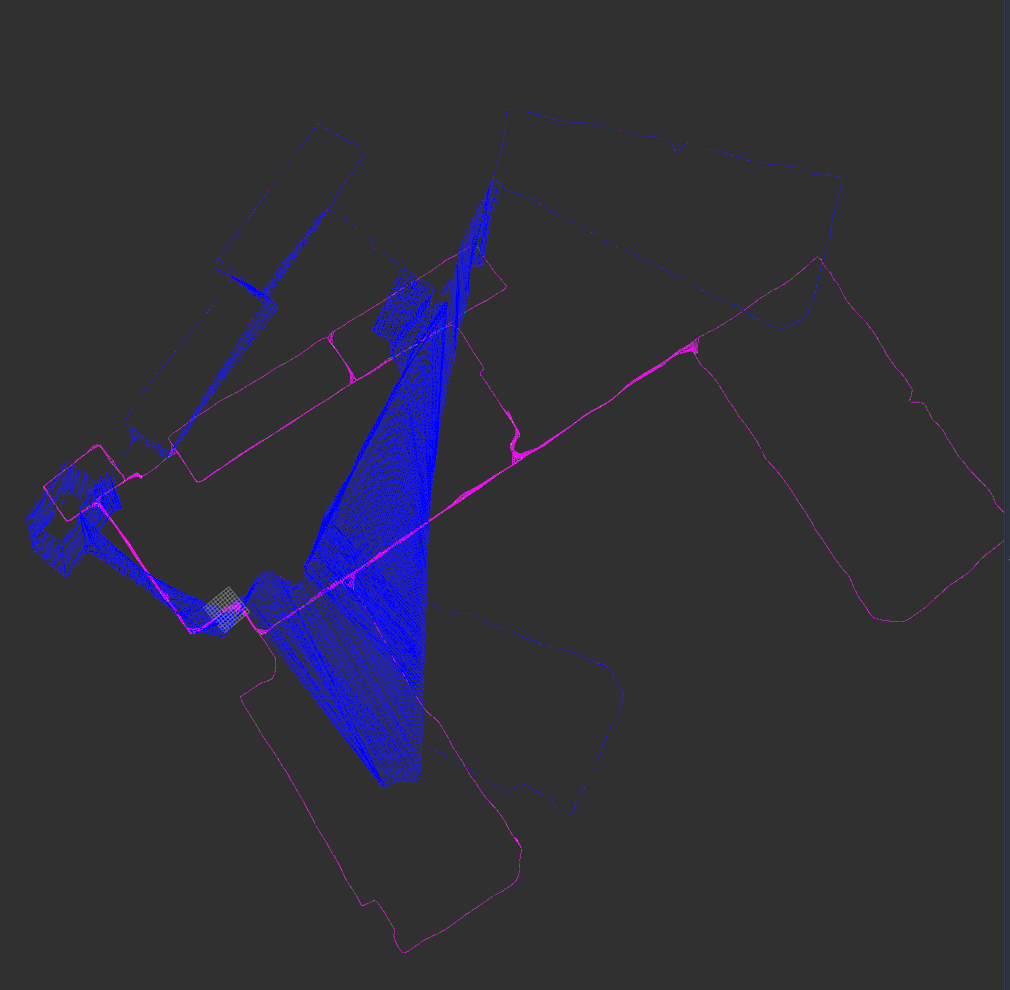
Intel数据集





Killian数据集





1. 简答题，开放性答案：你认为第一题的优化过程中哪个环节耗时最多？是否有什么改进的方法可以进行加速？（2 分）

代码修改，在每一次迭代的过程中的不同位置添加计时函数：

[代码如下所示]：

Eigen::VectorXd LinearizeAndSolve(std::vector<Eigen::Vector3d>& Vertexs,

std::vector<Edge>& Edges)

{

std::cout << "----------" << endl;

double start, all\_start;

all\_start = clock();

//申请内存

Eigen::MatrixXd H(Vertexs.size() \* 3, Vertexs.size() \* 3);

Eigen::VectorXd b(Vertexs.size() \* 3);

H.setZero();

b.setZero();

//固定第一帧

Eigen::Matrix3d I;

I.setIdentity();

H.block(0, 0, 3, 3) += I;

//构造H矩阵　＆ b向量

for (int i = 0; i < Edges.size(); i++) {

//提取信息

Edge tmpEdge = Edges[i];

Eigen::Vector3d xi = Vertexs[tmpEdge.xi];

Eigen::Vector3d xj = Vertexs[tmpEdge.xj];

Eigen::Vector3d z = tmpEdge.measurement;

Eigen::Matrix3d infoMatrix = tmpEdge.infoMatrix;

//计算误差和对应的Jacobian

Eigen::Vector3d ei;

Eigen::Matrix3d Ai;

Eigen::Matrix3d Bi;

start = clock();

CalcJacobianAndError(xi, xj, z, ei, Ai, Bi);

std::cout << "CalcJacobianAndError time: " << clock() - start << " ms" << endl;

start = clock();

//TODO--Start

b.block(3 \* tmpEdge.xi, 0, 3, 1) += (ei.transpose() \* infoMatrix \* Ai).transpose();

b.block(3 \* tmpEdge.xj, 0, 3, 1) += (ei.transpose() \* infoMatrix \* Bi).transpose();

H.block(3 \* tmpEdge.xi, 3 \* tmpEdge.xi, 3, 3) += Ai.transpose() \* infoMatrix \* Ai;

H.block(3 \* tmpEdge.xj, 3 \* tmpEdge.xj, 3, 3) += Bi.transpose() \* infoMatrix \* Bi;

H.block(3 \* tmpEdge.xi, 3 \* tmpEdge.xj, 3, 3) += Ai.transpose() \* infoMatrix \* Bi;

H.block(3 \* tmpEdge.xj, 3 \* tmpEdge.xi, 3, 3) += Bi.transpose() \* infoMatrix \* Ai;

std::cout << "cal block H b time: " << clock() - start << " ms" << endl;

//TODO--End

}

//求解

Eigen::VectorXd dx;

//TODO--Start

// dx = -H.lu().solve(b);

start = clock();

dx = -H.colPivHouseholderQr().solve(b);

std::cout << "solve HX=b time: " << clock() - start << endl;

std::cout << "all time: " << clock() - all\_start << " ms" << endl

<< endl;

//TODO-End

return dx;

}

运行代码输出每一个部分的时间：

可以看出一次迭代的时间大约为 800ms

其中，求解HX=b耗时大约200ms

计算雅克比矩阵的函数耗时大约 30ms \* 5

计算H 和 b 的block涉及到矩阵计算，这部分耗时 70ms \* 5

所以，可以发现虽然HX=b耗时比较多，但是这部分我们是调用函数接口来进行计算的，不好进行优化。另外一个占用时间最多的就是计算H 和 b 的block，这部分如下所示：

b.block(3 \* tmpEdge.xi, 0, 3, 1) += (ei.transpose() \* infoMatrix \* Ai).transpose();

b.block(3 \* tmpEdge.xj, 0, 3, 1) += (ei.transpose() \* infoMatrix \* Bi).transpose();

H.block(3 \* tmpEdge.xi, 3 \* tmpEdge.xi, 3, 3) += Ai.transpose() \* infoMatrix \* Ai;

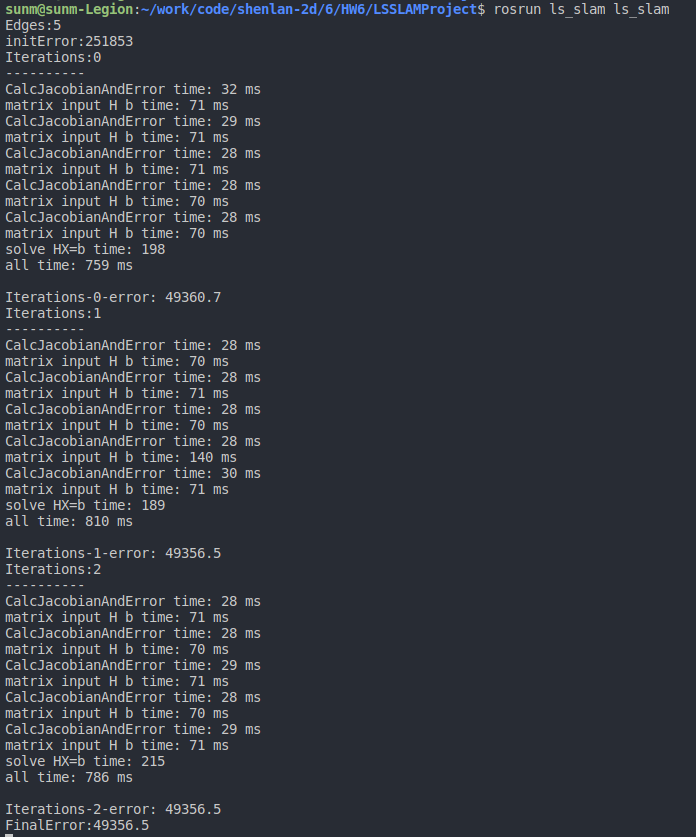
H.block(3 \* tmpEdge.xj, 3 \* tmpEdge.xj, 3, 3) += Bi.transpose() \* infoMatrix \* Bi;

H.block(3 \* tmpEdge.xi, 3 \* tmpEdge.xj, 3, 3) += Ai.transpose() \* infoMatrix \* Bi;

H.block(3 \* tmpEdge.xj, 3 \* tmpEdge.xi, 3, 3) += Bi.transpose() \* infoMatrix \* Ai;

这部分矩阵乘法可以进行优化

可以将矩阵乘法拆分，写成for循环，然后使用多线程进行加速，例如openmp

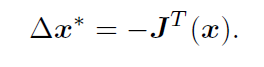


1. 学习相关材料。除了高斯牛顿方法，你还知道哪些非线性优化方法？请简述它们的原理；

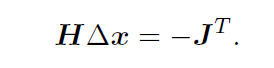
解答：

其他非线性优化方法主要还包括：最速下降法，牛顿法，列文博格-马夸尔特方法，这几种方法简要描述如下：

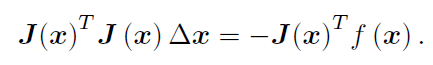
最速下降法：该方法是对目标函数进行一阶泰勒展开，下降速度最快，但是容易出现锯齿zigzag下降，迭代次数较高。计算增量的方向如下所示，然后再确定更新步长。



牛顿法：牛顿法是对目标函数进行二阶泰勒展开，需要计算海森矩阵，在问题规模较大的时候海森矩阵的计算较困难。牛顿法更新量计算如下所示：



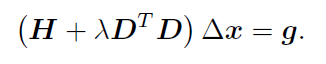
本节课中所提到的高斯牛顿法不是对目标函数进行二阶泰勒展开，而是对误差函数进行二阶泰勒展开，然后在计算平方目标函数（带信息矩阵）。避免了直接计算海森矩阵的复杂性，改为由雅克比矩阵替代。高斯牛顿法的更新量计算如下所示：



LM列文博格-马夸尔特也是对误差函数进行泰勒展开，但是在G-N的基础上，引入了信赖域的约束，对于在信赖域内的更新就接受，对于在信赖域之外的更新就拒绝，同时动态更新信赖域的范围。因为我们的方法是依据泰勒展开来近似的，信赖域的存在保证了泰勒展开的前提条件，即泰勒展开仅在展开点附近近似才有效。

信赖域范围的更新依赖于对上次更新的近似函数下降和实际函数下降的比例，如果比例接近于1,说明近似的比较好，那么下次可以放宽信赖区，加快更新步长。如果比例小于1,说明近似的不好，那么下次就减小信赖区域，减小更新步长。

1. M更新量的计算如下所示：



在Levenberg方法中，取D=I，则：



其中lamada是拉格朗日算子，其将信赖域的约束转化为无约束的优化问题。信赖域的范围大小变化体现在lamada的数值变化上。如果信赖区域大，那么lamada较大，LM算法接近最速下降法，迭代速度较快。如果信赖区域较小，那么lamada较小，LM算法更接近GN，迭代速度较慢，更精确。

1. 将第一题改为使用任意一种非线性优化库进行优化（比如 Ceres Gtsam 或 G2o 等）

[代码]：

使用g2o来优化：

定义g2o顶点：

class my2dSlamVertex : public BaseVertex<3, Eigen::Vector3d> {

public:

EIGEN\_MAKE\_ALIGNED\_OPERATOR\_NEW

my2dSlamVertex() {}

virtual bool read(std::istream& *is*) {}

virtual bool write(std::ostream& *os*) const {}

virtual void setToOriginImpl()

{

\_estimate << 0, 0, 0;

}

virtual void oplusImpl(const double\* *update*)

{

\_estimate += Eigen::Vector3d(update);

}

};

定义g2o边：

class my2dSlamEdge : public BaseBinaryEdge<3, Eigen::Vector3d, my2dSlamVertex, my2dSlamVertex> {

public:

EIGEN\_MAKE\_ALIGNED\_OPERATOR\_NEW

virtual void computeError()

{

const my2dSlamVertex\* v1 = static\_cast<const my2dSlamVertex\*>(\_vertices[0]);

const my2dSlamVertex\* v2 = static\_cast<const my2dSlamVertex\*>(\_vertices[1]);

const Eigen::Vector3d xi = v1->estimate();

const Eigen::Vector3d xj = v2->estimate();

Eigen::Matrix3d Xi = PoseToTrans(xi);

Eigen::Matrix3d Xj = PoseToTrans(xj);

Eigen::Matrix3d Z = PoseToTrans(\_measurement);

Eigen::Matrix3d Ei = Z.inverse() \* Xi.inverse() \* Xj;

\_error = TransToPose(Ei);

}

virtual void linearizeOplus()

{

const my2dSlamVertex\* v1 = static\_cast<const my2dSlamVertex\*>(\_vertices[0]);

const my2dSlamVertex\* v2 = static\_cast<const my2dSlamVertex\*>(\_vertices[1]);

Eigen::Vector3d xi = v1->estimate();

Eigen::Vector3d xj = v2->estimate();

Eigen::Vector3d z = \_measurement;

Eigen::Matrix3d trans\_xi = PoseToTrans(xi);

Eigen::Matrix3d trans\_xj = PoseToTrans(xj);

Eigen::Matrix3d trans\_z = PoseToTrans(\_measurement);

Eigen::Matrix2d R\_i = trans\_xi.block(0, 0, 2, 2);

Eigen::Matrix2d R\_j = trans\_xj.block(0, 0, 2, 2);

Eigen::Matrix2d R\_ij = trans\_z.block(0, 0, 2, 2);

Eigen::Vector2d t\_i = xi.block(0, 0, 2, 1);

Eigen::Vector2d t\_j = xj.block(0, 0, 2, 1);

Eigen::Vector2d t\_ij = z.block(0, 0, 2, 1);

\_jacobianOplusXi.setZero();

\_jacobianOplusXi.block(0, 0, 2, 2) = -R\_ij.transpose() \* R\_i.transpose();

\_jacobianOplusXi(2, 2) = -1;

Eigen::Matrix2d derivative\_Ri\_theta;

derivative\_Ri\_theta << -sin(xi[2]), cos(xi[2]), -cos(xi[2]), -sin(xi[2]);

\_jacobianOplusXi.block(0, 2, 2, 1) = R\_ij.transpose() \* derivative\_Ri\_theta \* (t\_j - t\_i);

\_jacobianOplusXj.setZero();

\_jacobianOplusXj.block(0, 0, 2, 2) = R\_ij.transpose() \* R\_i.transpose();

\_jacobianOplusXj(2, 2) = 1;

}

virtual bool read(istream& *in*) {}

virtual bool write(ostream& *out*) const {}

};

定义g2o优化器并优化：

typedef g2o::BlockSolver<g2o::BlockSolverTraits<3, 3>> Block; *// 每个误差项优化变量维度为3，误差值维度为1*

Block::LinearSolverType\* linearSolver = new g2o::LinearSolverDense<Block::PoseMatrixType>();

Block\* solver\_ptr = new Block(linearSolver);

g2o::OptimizationAlgorithmLevenberg\* solver = new g2o::OptimizationAlgorithmLevenberg(solver\_ptr);

g2o::SparseOptimizer optimizer;

optimizer.setAlgorithm(solver);

optimizer.setVerbose(true);

for (size\_t i = 0; i < Vertexs.size(); i++) {

my2dSlamVertex\* v = new my2dSlamVertex();

v->setEstimate(Vertexs[i]);

v->setId(i);

if (i == 0) {

v->setFixed(true);

}

optimizer.addVertex(v);

}

for (size\_t i = 0; i < Edges.size(); i++) {

my2dSlamEdge\* edge = new my2dSlamEdge();

Edge tmpEdge = Edges[i];

edge->setId(i);

edge->setVertex(0, optimizer.vertices()[tmpEdge.xi]);

edge->setVertex(1, optimizer.vertices()[tmpEdge.xj]);

edge->setMeasurement(tmpEdge.measurement);

edge->setInformation(tmpEdge.infoMatrix);

optimizer.addEdge(edge);

}

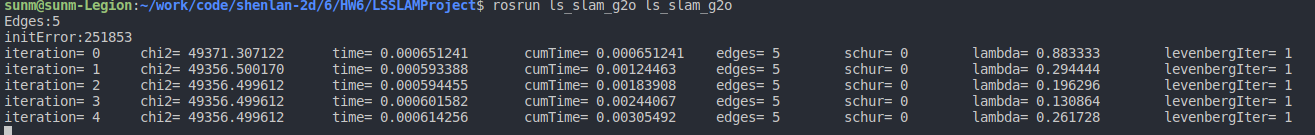
optimizer.setVerbose(true);

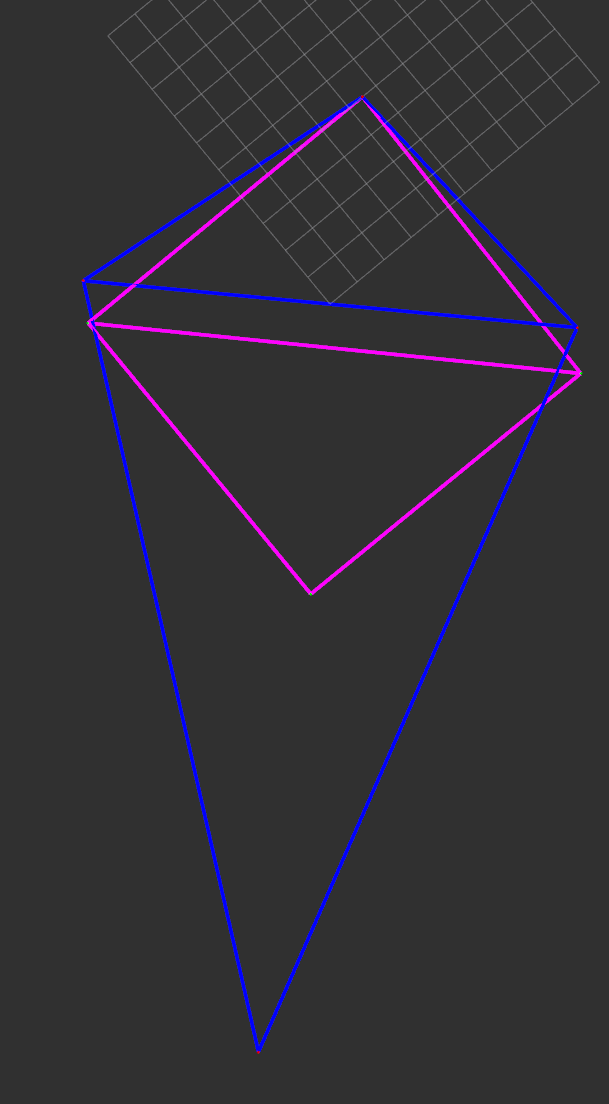
optimizer.initializeOptimization();

optimizer.optimize(100);

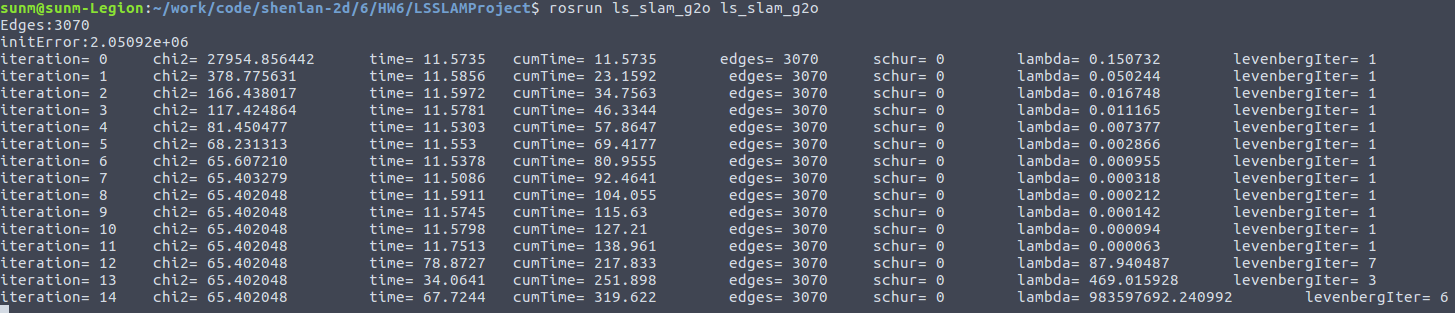
结果如下所示：

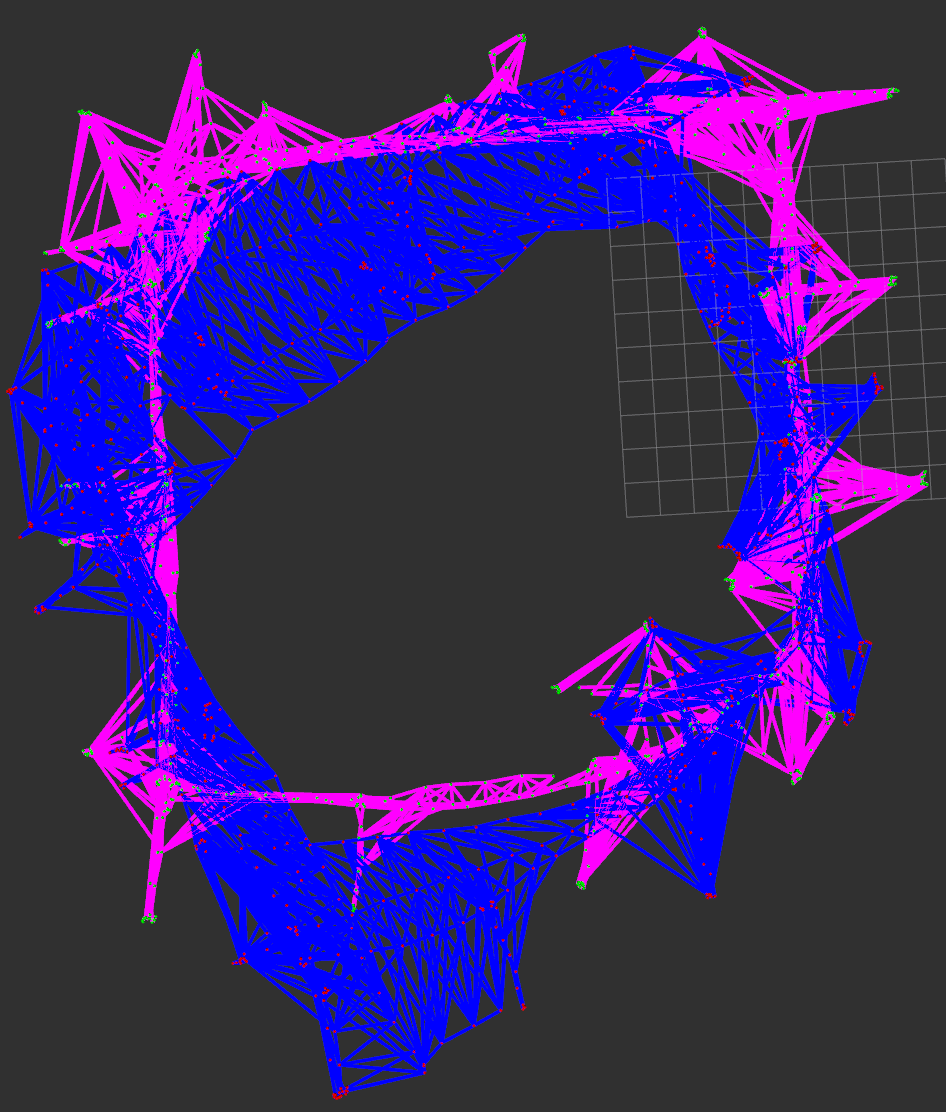
test数据集：





intel数据集：





Killian 数据集

