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
```

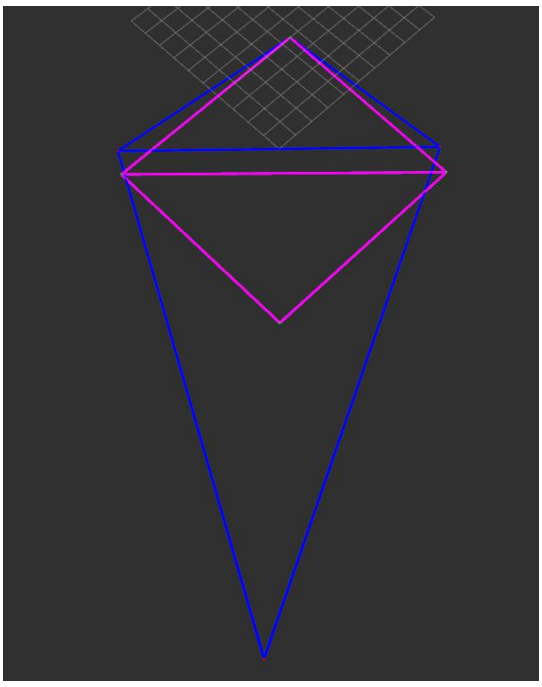
更新顶点

```
//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：

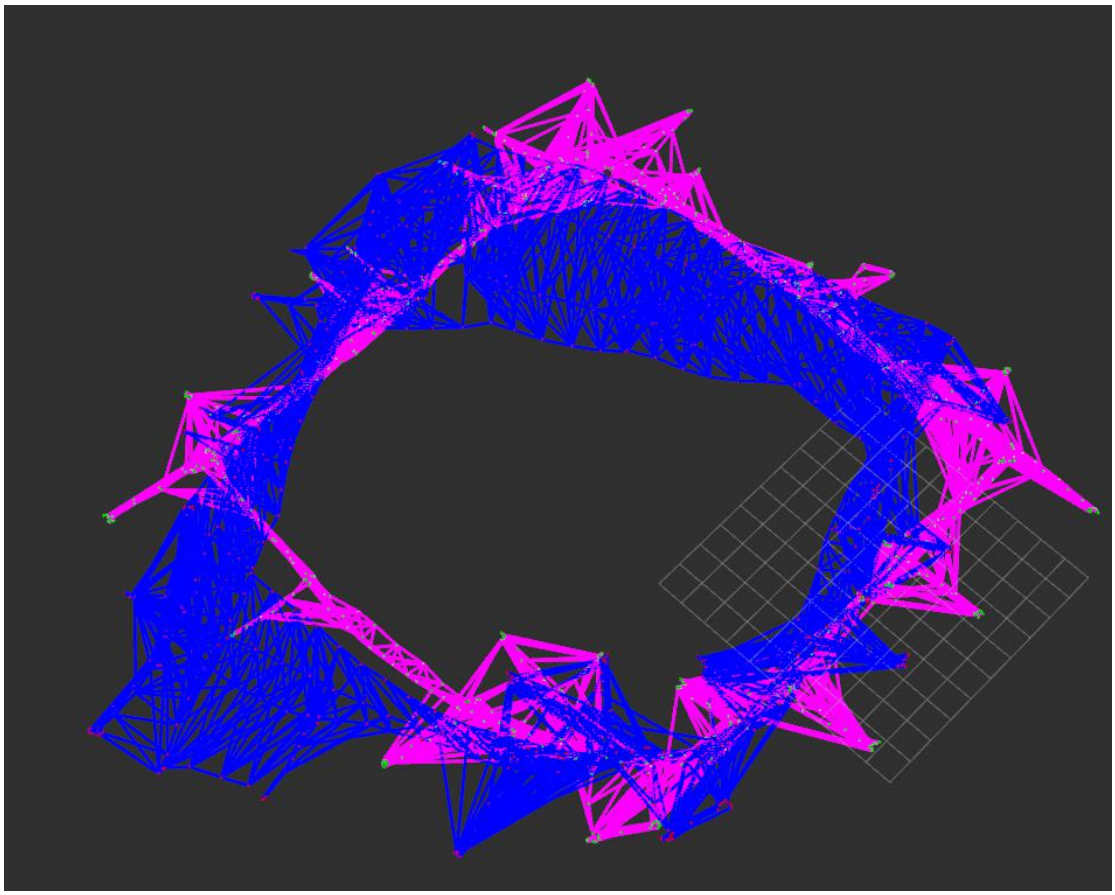
Test 数据集

```
sunm@sunm-Legion:~/work/code/shenlan-2d/6/HW6/LSSLAMProject$ rosrun ls_slam ls_slam
Edges:5
initError:251853
Iterations:0
error: 49360.7
Iterations:1
error: 49356.5
Iterations:2
error: 49356.5
FinalError:49356.5
```



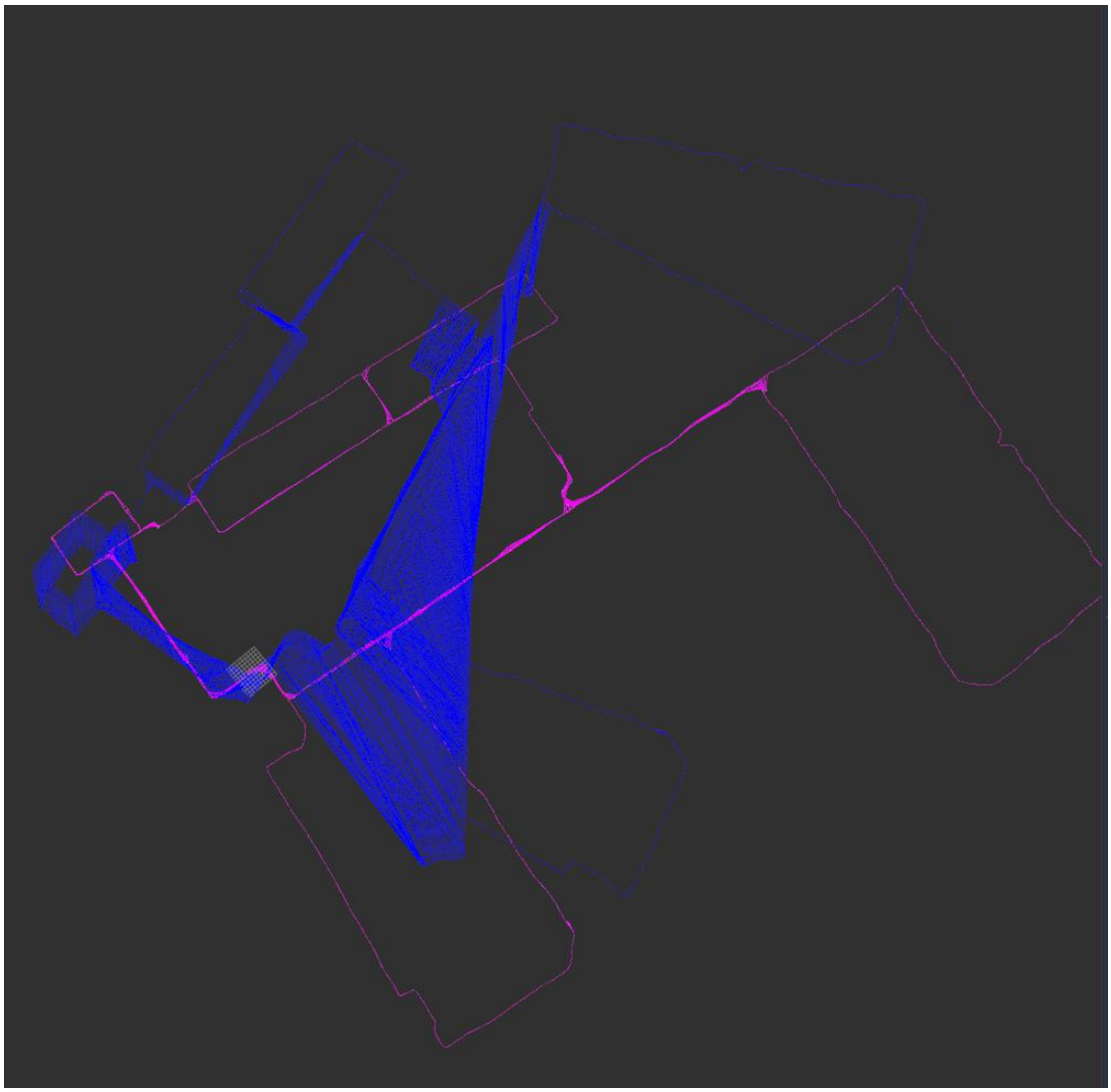
Intel 数据集

```
sunm@sunm-Legion:~/work/code/shenlan-2d/6/HW6/LSSLAMProject$ rosrun ls_slam ls_slam
Edges:3070
initError:2.05092e+06
Iterations:0
Iterations-0-error: 365236
Iterations:1
Iterations-1-error: 134.649
Iterations:2
Iterations-2-error: 65.4025
Iterations:3
Iterations-3-error: 65.402
Iterations:4
Iterations-4-error: 65.402
FinalError:65.402
```



Killian 数据集

```
sunm@sunm-Legion:~/work/code/shenlan-2d/6/HW6/LSSLAMProject$ rosrun ls_slam ls_slam
Edges:3995
initError:3.08592e+08
Iterations:0
Iterations-0-error: 1.39937e+08
Iterations:1
Iterations-1-error: 13466.4
Iterations:2
Iterations-2-error: 10344.7
Iterations:3
Iterations-3-error: 10344.7
Iterations:4
Iterations-4-error: 10344.7
Iterations:5
Iterations-5-error: 10344.7
FinalError:10344.7
```



2. 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

```
sunm@sunm-Legion:~/work/code/shenlan-2d/6/HW6/LSSLAMProject$ rosrun ls_slam ls_slam
Edges:5
initError:251853
Iterations:0
----------
CalcJacobianAndError time: 32 ms
matrix input H b time: 71 ms
CalcJacobianAndError time: 29 ms
matrix input H b time: 71 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 71 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 70 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 70 ms
solve HX=b time: 198
all time: 759 ms

Iterations-0-error: 49360.7
Iterations:1
----------
CalcJacobianAndError time: 28 ms
matrix input H b time: 70 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 71 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 70 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 140 ms
CalcJacobianAndError time: 30 ms
matrix input H b time: 71 ms
solve HX=b time: 189
all time: 810 ms

Iterations-1-error: 49356.5
Iterations:2
----------
CalcJacobianAndError time: 28 ms
matrix input H b time: 71 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 70 ms
CalcJacobianAndError time: 29 ms
matrix input H b time: 71 ms
CalcJacobianAndError time: 28 ms
matrix input H b time: 70 ms
CalcJacobianAndError time: 29 ms
matrix input H b time: 71 ms
solve HX=b time: 215
all time: 786 ms

Iterations-2-error: 49356.5
FinalError:49356.5
```

3、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。

$$\Delta \boldsymbol{x}^* = -\boldsymbol{J}^T(\boldsymbol{x}).$$

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：

$$\boldsymbol{H}\Delta \boldsymbol{x} = -\boldsymbol{J}^T.$$

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：

$$\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{J}(\boldsymbol{x})\, \Delta \boldsymbol{x} = -\boldsymbol{J}(\boldsymbol{x})^T f(\boldsymbol{x}).$$

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。

L-M 更新量的计算如下所示：

$$\left(\boldsymbol{H}+\lambda \boldsymbol{D}^{T} \boldsymbol{D}\right) \Delta \boldsymbol{x}=\boldsymbol{g}.$$

在 Levenberg 方法中，取 D=I，则：

$$\left(\boldsymbol{H}+\lambda \boldsymbol{I}\right) \Delta \boldsymbol{x}=\boldsymbol{g}.$$

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。

4. 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：

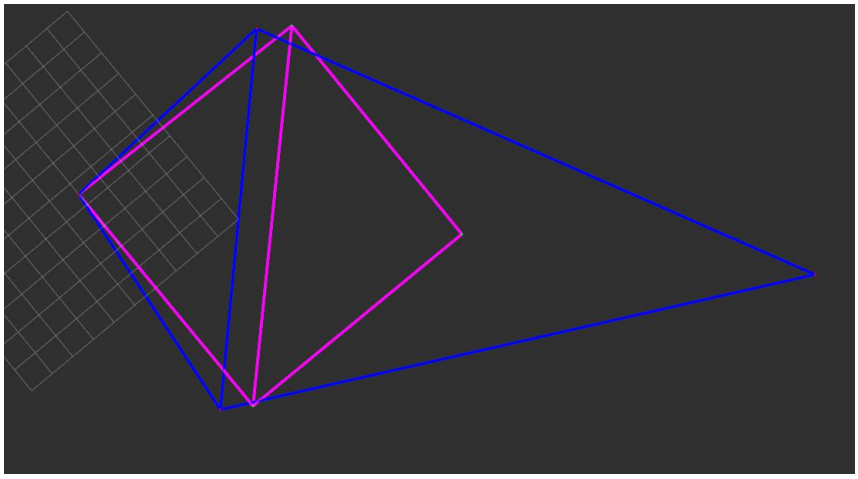
```
typedef g2o::BlockSolver<g2o::BlockSolverTraits<3, 3>> Block; // 每个误差项优化变量维度为 3，误差值维度为 1

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：

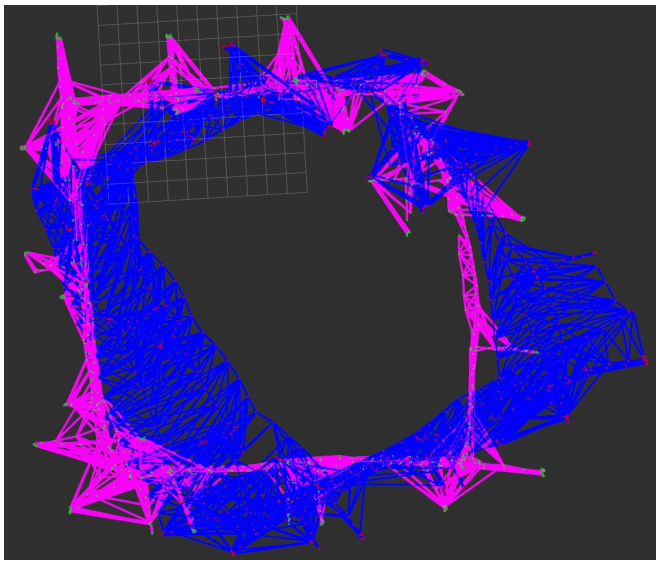
test 数据集：

```
sunm@sunm-Legion:~/work/code/shenlan-2d/6/HW6/LSSLAMProject$ rosrun ls_slam_g2o ls_slam_g2o
Edges:5
initError:251853
iteration= 0     chi2= 49371.307122      time= 0.000651241       cumTime= 0.000651241    edges= 5        schur= 0        lambda= 0.883333        levenbergIter= 1
iteration= 1     chi2= 49356.500170      time= 0.000593388       cumTime= 0.00124463     edges= 5        schur= 0        lambda= 0.294444        levenbergIter= 1
iteration= 2     chi2= 49356.499612      time= 0.000594455       cumTime= 0.00183908     edges= 5        schur= 0        lambda= 0.196296        levenbergIter= 1
iteration= 3     chi2= 49356.499612      time= 0.000601582       cumTime= 0.00244067     edges= 5        schur= 0        lambda= 0.130864        levenbergIter= 1
iteration= 4     chi2= 49356.499612      time= 0.000614256       cumTime= 0.00305492     edges= 5        schur= 0        lambda= 0.261728        levenbergIter= 1
```



intel 数据集：

```
sunn@sunn-Legion:~/work/code/shenlan-2d/6/HW6/LSSLAMProject$ rosrun ls_slam_g2o ls_slam_g2o
Edges:3070
initError:2.05092e+06
iteration= 0     chi2= 27954.856442      time= 11.5735   cumTime= 11.5735       edges= 3070      schur= 0        lambda= 0.150732        levenbergIter= 1
iteration= 1     chi2= 378.775631        time= 11.5856   cumTime= 23.1592        edges= 3070     schur= 0        lambda= 0.050244        levenbergIter= 1
iteration= 2     chi2= 166.438017        time= 11.5972   cumTime= 34.7563        edges= 3070     schur= 0        lambda= 0.016748        levenbergIter= 1
iteration= 3     chi2= 117.424864        time= 11.5781   cumTime= 46.3344        edges= 3070     schur= 0        lambda= 0.011165        levenbergIter= 1
iteration= 4     chi2= 81.450477         time= 11.5303   cumTime= 57.8647        edges= 3070     schur= 0        lambda= 0.007377        levenbergIter= 1
iteration= 5     chi2= 68.231313         time= 11.553    cumTime= 69.4177        edges= 3070     schur= 0        lambda= 0.002866        levenbergIter= 1
iteration= 6     chi2= 65.607210         time= 11.5378   cumTime= 80.9555        edges= 3070     schur= 0        lambda= 0.000955        levenbergIter= 1
iteration= 7     chi2= 65.403279         time= 11.5086   cumTime= 92.4641        edges= 3070     schur= 0        lambda= 0.000318        levenbergIter= 1
iteration= 8     chi2= 65.402048         time= 11.5911   cumTime= 104.055        edges= 3070     schur= 0        lambda= 0.000212        levenbergIter= 1
iteration= 9     chi2= 65.402048         time= 11.5745   cumTime= 115.63         edges= 3070     schur= 0        lambda= 0.000142        levenbergIter= 1
iteration= 10    chi2= 65.402048         time= 11.5798   cumTime= 127.21         edges= 3070     schur= 0        lambda= 0.000094        levenbergIter= 1
iteration= 11    chi2= 65.402048         time= 11.7513   cumTime= 138.961        edges= 3070     schur= 0        lambda= 0.000063        levenbergIter= 1
iteration= 12    chi2= 65.402048         time= 78.8727   cumTime= 217.833        edges= 3070     schur= 0        lambda= 87.940487       levenbergIter= 7
iteration= 13    chi2= 65.402048         time= 34.0641   cumTime= 251.898        edges= 3070     schur= 0        lambda= 469.015928      levenbergIter= 3
iteration= 14    chi2= 65.402048         time= 67.7244   cumTime= 319.622        edges= 3070     schur= 0        lambda= 983597692.240992        levenbergIter= 6
```



Killian 数据集

```
sunm@sunm-Legion:~/work/code/shenlan-2d/6/HW6/LSSLAMProject$ rosrun ls_slam_g2o ls_slam_g2o
Edges:3995
initError:3.08592e+08
iteration= 0     chi2= 6973711.783675    time= 198.084   cumTime= 198.084        edges= 3995     schur= 0        lambda= 42.450053       levenbergIter= 1
iteration= 1     chi2= 477444.019837     time= 201.877   cumTime= 399.96         edges= 3995     schur= 0        lambda= 14.150018       levenbergIter= 1
iteration= 2     chi2= 129918.234974     time= 202.732   cumTime= 602.693        edges= 3995     schur= 0        lambda= 4.716673        levenbergIter= 1
iteration= 3     chi2= 58093.623167      time= 199.945   cumTime= 802.638        edges= 3995     schur= 0        lambda= 1.572224        levenbergIter= 1
iteration= 4     chi2= 37310.104253      time= 198.03    cumTime= 1000.67        edges= 3995     schur= 0        lambda= 0.524075        levenbergIter= 1
iteration= 5     chi2= 24203.405820      time= 199.686   cumTime= 1200.35        edges= 3995     schur= 0        lambda= 0.174692        levenbergIter= 1
iteration= 6     chi2= 17429.389130      time= 196.819   cumTime= 1397.17        edges= 3995     schur= 0        lambda= 0.058231        levenbergIter= 1
iteration= 7     chi2= 15053.951025      time= 196.81    cumTime= 1593.98        edges= 3995     schur= 0        lambda= 0.019410        levenbergIter= 1
iteration= 8     chi2= 13862.008409      time= 197.066   cumTime= 1791.05        edges= 3995     schur= 0        lambda= 0.006470        levenbergIter= 1
iteration= 9     chi2= 12832.051089      time= 199.502   cumTime= 1990.55        edges= 3995     schur= 0        lambda= 0.002157        levenbergIter= 1
iteration= 10    chi2= 11675.800703      time= 199.991   cumTime= 2190.54        edges= 3995     schur= 0        lambda= 0.000719        levenbergIter= 1
iteration= 11    chi2= 10722.720750      time= 199.47    cumTime= 2390.01        edges= 3995     schur= 0        lambda= 0.000240        levenbergIter= 1
iteration= 12    chi2= 10381.563442      time= 196.95    cumTime= 2586.96        edges= 3995     schur= 0        lambda= 0.000080        levenbergIter= 1
iteration= 13    chi2= 10345.270407      time= 198.106   cumTime= 2785.07        edges= 3995     schur= 0        lambda= 0.000027        levenbergIter= 1
iteration= 14    chi2= 10344.666305      time= 196.388   cumTime= 2981.46        edges= 3995     schur= 0        lambda= 0.000009        levenbergIter= 1
iteration= 15    chi2= 10344.665263      time= 196.627   cumTime= 3178.08        edges= 3995     schur= 0        lambda= 0.000006        levenbergIter= 1
iteration= 16    chi2= 10344.665262      time= 197.252   cumTime= 3375.33        edges= 3995     schur= 0        lambda= 0.000004        levenbergIter= 1
iteration= 17    chi2= 10344.665262      time= 196.648   cumTime= 3571.98        edges= 3995     schur= 0        lambda= 0.000003        levenbergIter= 1
iteration= 18    chi2= 10344.665262      time= 786.806   cumTime= 4358.79        edges= 3995     schur= 0        lambda= 0.000112        levenbergIter= 4
iteration= 19    chi2= 10344.665262      time= 592.513   cumTime= 4951.3         edges= 3995     schur= 0        lambda= 0.000598        levenbergIter= 3
iteration= 20    chi2= 10344.665262      time= 1015.46   cumTime= 5966.76        edges= 3995     schur= 0        lambda= 0.408510        levenbergIter= 5
iteration= 21    chi2= 10344.665262      time= 993.419   cumTime= 6960.18        edges= 3995     schur= 0        lambda= 278.876382      levenbergIter= 5
iteration= 22    chi2= 10344.665262      time= 997.588   cumTime= 7957.77        edges= 3995     schur= 0        lambda= 190379.610439   levenbergIter= 5
iteration= 23    chi2= 10344.665262      time= 201.44    cumTime= 8159.21        edges= 3995     schur= 0        lambda= 380759.220877   levenbergIter= 1
```