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

    1. 计算jacobian矩阵和error；

```
//TODO--Start
// Eigen::Matrix3d zij = PoseToTrans(xi).inverse() * PoseToTrans(xj);
// Eigen::Vector3d eij = TransToPose(PoseToTrans(z).inverse() * zij);
Eigen::Matrix2d RijT;
RijT << cos(z(2)),sin(z(2)),
       -sin(z(2)),cos(z(2));
Eigen::Matrix2d RiT;
RiT << cos(xi(2)),sin(xi(2)),
      -sin(xi(2)),cos(xi(2));
Eigen::Matrix2d dRiT;
dRiT << -sin(xi(2)), cos(xi(2)),
        -cos(xi(2)),-sin(xi(2));

ei.block(0, 0, 2, 1) = RijT * (RiT * (xj.block(0, 0, 2, 1) - xi.block(0, 0, 2, 1)) - z.block(0, 0, 2, 1));
ei(2) = xj(2) - xi(2) - z(2);
if (ei(2) > M_PI)
    ei(2) -= 2 * M_PI;
else if (ei(2) < -M_PI)
    ei(2) += 2 * M_PI;
Ai.block(0, 0, 2, 2) = - RijT * RiT;
Ai.block(0, 2, 2, 1) = RijT * dRiT * (xj.block(0, 0, 2, 1) - xi.block(0, 0, 2, 1));
Ai.block(2, 0, 1, 3) << 0, 0, -1;
Bi.setIdentity();
Bi.block(0, 0, 2, 2) = RijT * RiT;
//TODO--end
```

    2. 构造H矩阵和b向量以及求解；

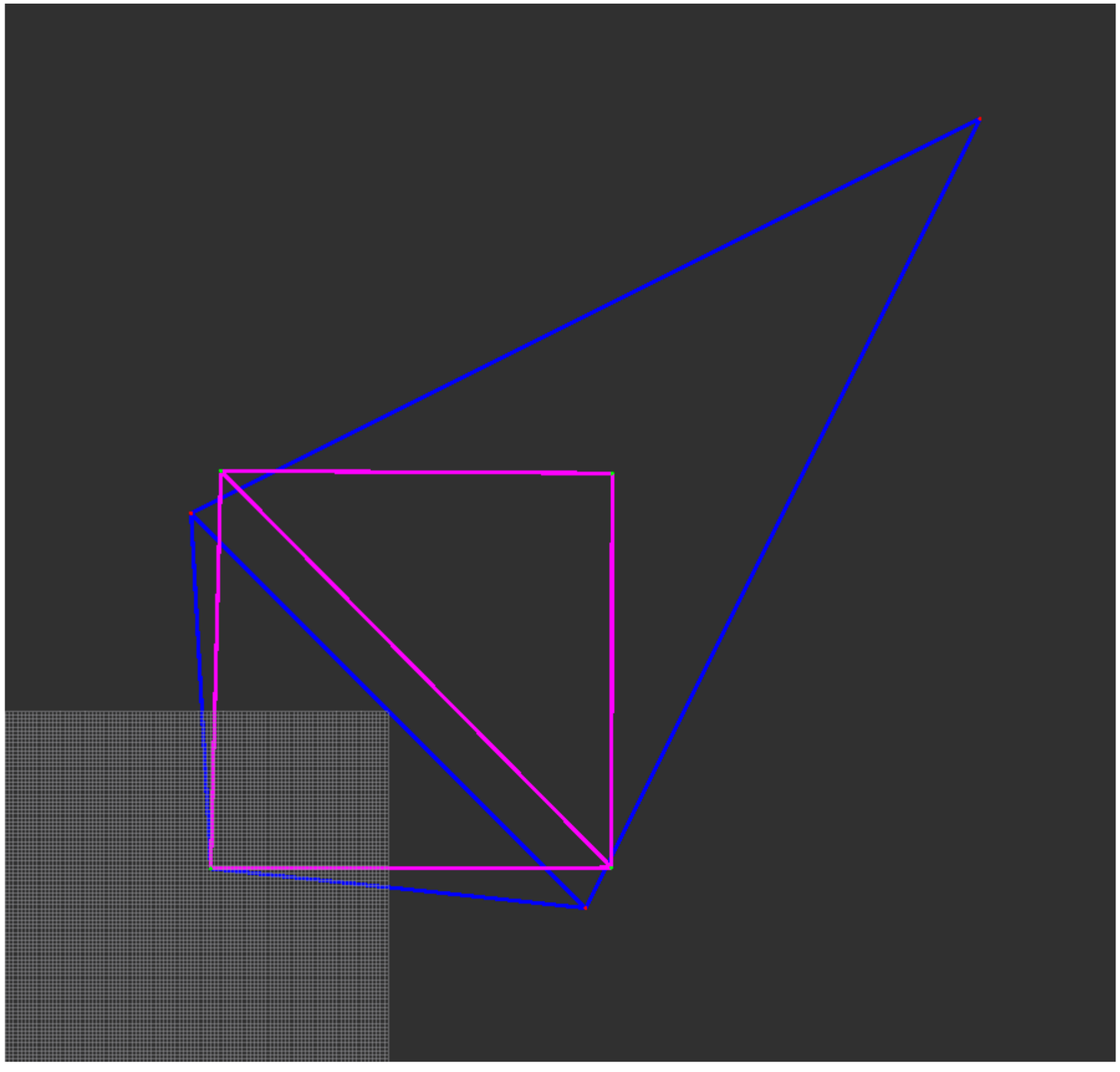
```
//构造H矩阵  &  b向量
for(int i = 0; i < Edges.size();i++)
{
    //提取信息
    Edge tmpEdge = Edges[i];
    Eigen::Vector3d xi = Vertexs[tmpEdge.xi];
    Eigen::Vector3d xj = Vertexs[tmpEdge.xj];
    Eigen::Vector3d z = tmpEdge.measurement;
    Eigen::Matrix3d infoMatrix = tmpEdge.infoMatrix;

    //计算误差和对应的Jacobian
    Eigen::Vector3d ei;
    Eigen::Matrix3d Ai;
    Eigen::Matrix3d Bi;
    CalcJacobianAndError(xi,xj,z,ei,Ai,Bi);

    //TODO--Start
    b.block(3*tmpEdge.xi, 0, 3, 1) += Ai.transpose() * infoMatrix * ei;
    b.block(3*tmpEdge.xj, 0, 3, 1) += Bi.transpose() * infoMatrix * ei;
    H.block(3*tmpEdge.xi, 3*tmpEdge.xi, 3, 3) += Ai.transpose() * infoMatrix * Ai;
    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;
    H.block(3*tmpEdge.xj, 3*tmpEdge.xj, 3, 3) += Bi.transpose() * infoMatrix * Bi;
    //TODO--End
}

//求解
Eigen::VectorXd dx;
//TODO--Start
dx = H.colPivHouseholderQr().solve(-b);
// dx = H.llt().solve(-b);
//TODO-End
```

1. 更新x。

```
//TODO--Start
for(int j = 0; j < Vertexs.size(); ++j)
{
    //when j = 0, delta x = 0, I should check
    Vertexs[j](0) += dx(j*3);
    Vertexs[j](1) += dx(j*3+1);
    Vertexs[j](2) += dx(j*3+2);
    if (Vertexs[j](2) > M_PI)
        Vertexs[j](2) -= 2 * M_PI;
    else if (Vertexs[j](2) < -M_PI)
        Vertexs[j](2) += 2 * M_PI;
}
//TODO--End
```

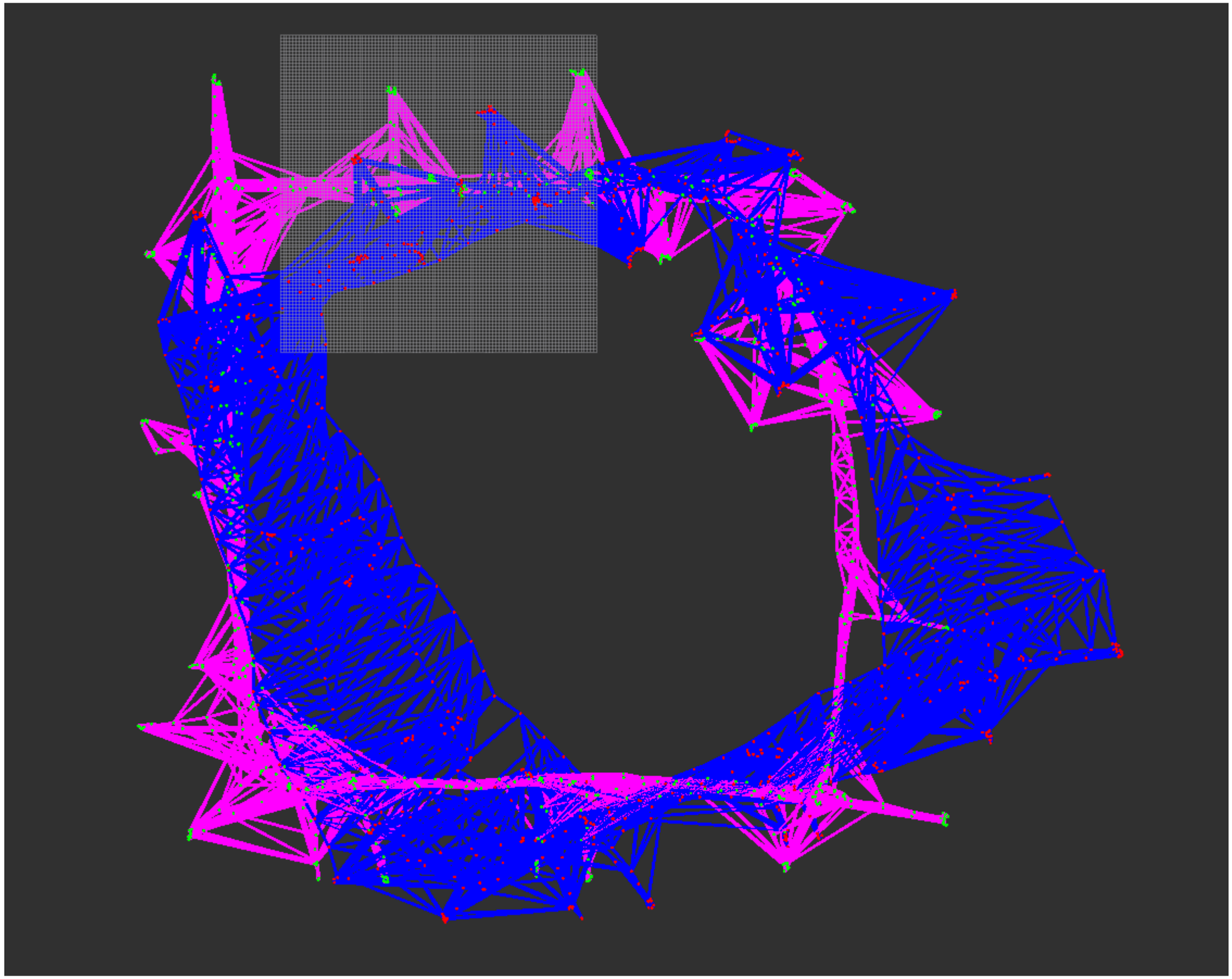
三组数据运行结果如下所示：
test_quadrat数据集：

```
$ rosrun ls_slam ls_slam
Edges:5
initError:251853
Iterations:0
Iterations:1
Iterations:2
FinalError:49356.5
```



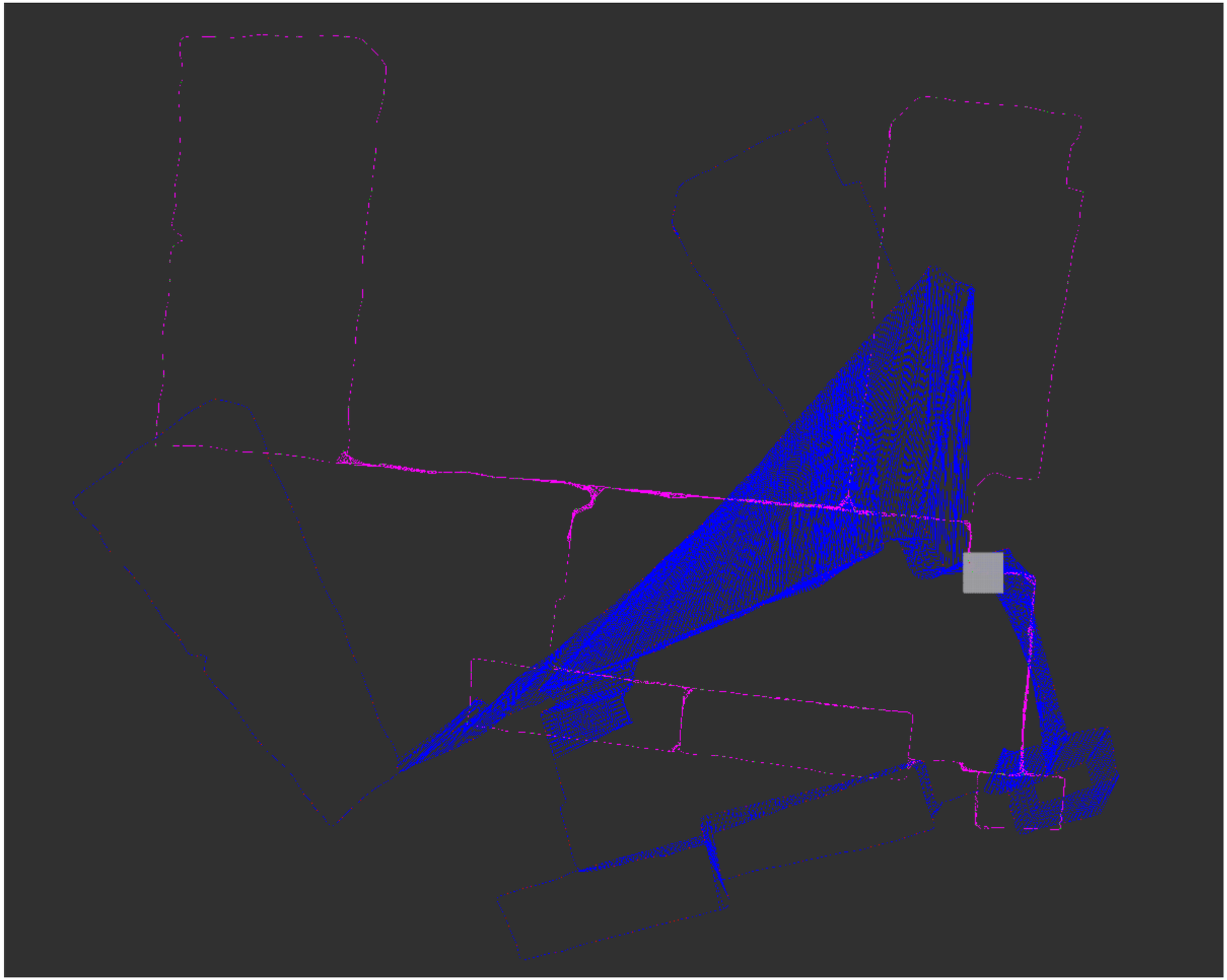
intel数据集：

```
$ rosrun ls_slam ls_slam
Edges:3070
initError:2.05092e+06
Iterations:0
Iterations:1
Iterations:2
Iterations:3
Iterations:4
FinalError:65.402
```



killian数据集：

```
$ rosrun ls_slam ls_slam
Edges:3995
initError:3.08592e+08
Iterations:0
Iterations:1
Iterations:2
Iterations:3
Iterations:4
Iterations:5
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）

加入测试代码：
在求解位置加入测试代码，如下：

```
//求解
Eigen::VectorXd dx;
auto last_time = std::chrono::high_resolution_clock::now();
//TODO--Start
dx = H.colPivHouseholderQr().solve(-b);
//TODO-End
auto current_time = std::chrono::high_resolution_clock::now();
std::cout  << "solve time(ms): " << std::chrono::duration_cast<std::chrono::milliseconds>(current_time - last_time).count() << std::endl;
```

在迭代处加入测试代码，求总耗时：

```
auto last_time = std::chrono::high_resolution_clock::now();
for(int i = 0; i < maxIteration;i++)
{
    std::cout <<"Iterations:"<<i<<std::endl;
    Eigen::VectorXd dx = LinearizeAndSolve(Vertexs,Edges);

    //进行更新
    //TODO--Start
    for(int j = 0; j < Vertexs.size(); ++j)
    {
        //when j = 0, delta x = 0, I should check
        Vertexs[j](0) += dx(j*3);
        Vertexs[j](1) += dx(j*3+1);
        Vertexs[j](2) += dx(j*3+2);
        if (Vertexs[j](2) > M_PI)
            Vertexs[j](2) -= 2 * M_PI;
        else if (Vertexs[j](2) < -M_PI)
            Vertexs[j](2) += 2 * M_PI;
    }
    //TODO--End

    double maxError = -1;
    for(int k = 0; k < 3 * Vertexs.size();k++)
    {
        if(maxError < std::fabs(dx(k)))
        {
            maxError = std::fabs(dx(k));
        }
    }

    if(maxError < epsilon)
        break;
}
auto current_time = std::chrono::high_resolution_clock::now();
std::cout  << "all time(ms): " << std::chrono::duration_cast<std::chrono::milliseconds>(current_time - last_time).count() << std::endl;
```

以intel数据集作为测试数据集：

```
$ rosrun ls_slam ls_slam
Edges:3070
initError:2.05092e+06
Iterations:0
solve time(ms): 145168
Iterations:1
solve time(ms): 145980
Iterations:2
solve time(ms): 148969
Iterations:3
solve time(ms): 143156
Iterations:4
solve time(ms): 143954
all time(ms): 727791
FinalError:65.402
```

经过测试可知，总共迭代5次，求解过程耗时727227ms，总耗时727791ms，除求解过程其他耗时564ms，所以求解过程耗时最多。
加速方法：求解过程加速，H为稀疏矩阵，可以使用eigen中的稀疏矩阵求解器进行求解；求解J矩阵加速，可以使用openmp进行多线程加速。
使用稀疏矩阵求解器进行求解，修改代码如下：

```
//求解
Eigen::VectorXd dx;
auto last_time = std::chrono::high_resolution_clock::now();
//TODO--Start
Eigen::SimplicialCholesky<Eigen::SparseMatrix<double> > solver;
Eigen::SparseMatrix<double> A = H.sparseView();
dx = solver.compute(A).solve(-b);
//TODO-End
```

运行结果如下所示：

```
$ rosrun ls_slam ls_slam
Edges:3070
initError:2.05092e+06
Iterations:0
solve time(ms): 93
Iterations:1
solve time(ms): 95
Iterations:2
solve time(ms): 93
Iterations:3
solve time(ms): 94
Iterations:4
solve time(ms): 93
all time(ms): 999
FinalError:65.402
```

总共迭代5次，求解过程耗时468ms，总耗时999ms，除求解过程其他耗时531ms，求解过程耗时大大减少。

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；（2 分）

除了高斯牛顿法（GN法），还有一阶（最速下降法）和二阶（牛顿法）梯度法以及列文伯格-马夸尔特算法（LM法）。
将目标函数在 x 附近进行泰勒展开，如果保留一阶梯度，那么增量的方向为：

$$\Delta x^* = -J^T(x)$$

如果保留二阶梯度信息，增量的解为：

$$H\Delta x = -J^T$$

最速下降法过于贪心，容易走出锯齿路线，反而增加了迭代次数。而牛顿法则需要计算目标函数的 H 矩阵，这在问题规模较大时非常困难，我们通常倾向于避免 H 的计算。
高斯牛顿法它的思想是将 $f(x)$ 进行一阶的泰勒展开（请注意不是目标函数 $f(x)^2$ ），最终得到增量方程为：

$$J^T(x)J(x)\Delta x = -J^T(x)f(x)$$

对比牛顿法可见， 高斯牛顿法用 J^TJ 作为牛顿法中二阶 Hessian 矩阵的近似，从而省略了计算 H 的过程。然而 H 矩阵存在可能为奇异矩阵或者病态 (illcondition) 的情况，此时增量的稳定性较差，导致算法不收敛。更严重的是，就算我们假设 H 非奇异也非病态，如果我们求出来的步长 Δx 太大，也会导致我们采用的局部近似（一阶泰勒展开）不够准确，这样一来我们甚至都无法保证它的迭代收敛，哪怕是让目标函数变得更大都是有可能的。
Levenberg-Marquadt 方法在一定程度上修正了这些问题，该方法的简化形式为：

$$(H + \lambda I)\Delta x = g$$

我们看到，当参数 λ 比较小时， H 占主要地位，这说明二次近似模型在该范围内是比较好的， LM 方法更接近于 GN 法。另一方面，当 λ 比较大时， λI 占据主要地位， LM更接近于一阶梯度下降法（即最速下降），这说明附近的二次近似不够好。LM 的求解方式，可在一定程度上避免线性方程组的系数矩阵的非奇异和病态问题，提供更稳定更准确的增量 Δx。

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）。（2 分）

将第一题使用G2O进行优化，使用顶点g2o::VertexSE2，边g2o::EdgeSE2，插入代码如下：

```cpp
double initError = ComputeError(Vertexs,Edges);
std::cout <<"initError:"<<initError<<std::endl;

auto last_time = std::chrono::high_resolution_clock::now();
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3
Block::LinearSolverType* linearSolver = new g2o::LinearSolverCSparse<Block::PoseMatrixType>();
Block* solver_ptr = new Block(linearSolver);
g2o::OptimizationAlgorithmGaussNewton* solver = new g2o::OptimizationAlgorithmGaussNewton( solver_ptr );
g2o::SparseOptimizer optimizer;
optimizer.setAlgorithm(solver);

for (size_t i = 0; i < Vertexs.size(); i++) {
    VertexSE2* v = new VertexSE2();
    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++) {
    EdgeSE2* edge = new EdgeSE2();

    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();
SparseOptimizerTerminateAction* terminateAction = new SparseOptimizerTerminateAction;
terminateAction->setGainThreshold(1e-4);
optimizer.addPostIterationAction(terminateAction);
optimizer.optimize(100);

for (size_t i = 0; i < Vertexs.size(); i++) {
    VertexSE2* v = static_cast<VertexSE2*>(optimizer.vertices()[i]);
    Vertexs[i] = v->estimate().toVector();
}
auto current_time = std::chrono::high_resolution_clock::now();
std::cout  << "all time(ms): " << std::chrono::duration_cast<std::chrono::milliseconds>(current_time - last_time).count() << std::endl;

double finalError  = ComputeError(Vertexs,Edges);

std::cout <<"FinalError:"<<finalError<<std::endl;
```

解决编译问题后，运行如下：

```
$ rosrun ls_slam ls_slam
Edges:3070
initError:2.05092e+06
iteration= 0     chi2= 365235.618940     time= 0.00546132        cumTime= 0.00546132     edges= 3070     schur= 0
iteration= 1     chi2= 134.648697        time= 0.00350787        cumTime= 0.00896919     edges= 3070     schur= 0
iteration= 2     chi2= 65.402453         time= 0.00361978        cumTime= 0.012589       edges= 3070     schur= 0
iteration= 3     chi2= 65.402048         time= 0.00415684        cumTime= 0.0167458      edges= 3070     schur= 0
all time(ms): 28
FinalError:65.402
```

线性方程求解器设置为CSparse，同样设置为高斯牛顿方法，总共迭代4次（比我们自己的程序少迭代一次，具体原因后面有时间可以查看），总耗时28ms，总耗时比优化后的代码（999ms）再次大幅减少。